Chapter 3

WMI: Instrumentation

In the previous Chapter I covered how to make use of the existing support for WMI that Windows provides. This proved very useful when you wanted to access data about a local or remote computer. I also covered the powerful event features that WMI supports. However, this is only half the story; you can also create your own providers that can be queried against, just like any native provider supplied by Microsoft. Now take a moment to think about what this means. Using WMI, you can now create objects on one computer and then remotely control and monitor those objects. As you have seen there is a lot of power within WMI to obtain data and register for changes using events. Now you can apply that power to your own objects. As mentioned there are a number of ISVs already providing WMI support, and you are encouraged to consult the documentation or ask the supplier about their support for WMI.

Manipulating Class Instances

Before I explain how to add WMI support to your objects there are still a few things about WMI which I haven't covered. I've explained how you can retrieve instances and return the data they provide. However, WMI also allows you to modify the data of the instance you return.

In addition to allowing you to modify data, WMI also supports asynchronous processing using delegates.

Modifying Instances

I've covered how you can expose your classes to WMI and make them accessible to any WMI compliant application. However, there are times when you will want to modify rather than just view this data. When you create your own WMI enabled objects you will also be able to write or update the data exposed by your classes.

To modify data you only need to have an instance of the class which you want to modify, set the write enabled properties and then call the Put method. This method also allows you to execute the operation with the following overloaded options:

```
Function Put() As ManagementPath
Function Put(PutOptions) As ManagementPath
Sub Put(ManagementOperationObserver)
Sub Put(ManagementOperationObserver, PutOptions)
```

* **PutOptions**: This object allows you to set options that determine how updates will be performed, such as CreateOnly, UpdateOnly, UpdateOrCreate (default).
* **ManagementOperationObserver**: This allows you to execute your updates asynchronously, by specifying a method to execute when the operation finishes.

To demonstrate this let's assume you want to modify the volume name of drive C:. First we need to locate the class instance which we want to modify, which is done with the following WQL:

```
SELECT * FROM Win32_LogicalDisk where DeviceID='C:'
```

Next we need to change the VolumeName property of the class to the new name:

```
objDrive.Properties("VolumeName").Value = <new name>
```

Then once we have changed the relevant property or properties we need to save this information back to the object. To do this we simply call the Put method:

```
objDrive.Put()
```

So if we put this all together as an example we get the following:

```
Shared Sub ModifyVolumeName()
    Dim objScope As New ManagementScope("\\.\root\cimv2")
    Dim objWMISearcher As New ManagementObjectSearcher(objScope, _
        New SelectQuery("SELECT * FROM Win32_LogicalDisk where DeviceID='C:'"))
    Dim objDrive As ManagementObject
    Try
        For Each objDrive In objWMISearcher.Get
            objDrive.Properties("VolumeName").Value = _
                InputBox("Enter New Name for Drive C:", "Change Caption")
            objDrive.Put()
        Next
    Catch ex As Exception
        MessageBox.Show(ex.Message)
    Finally
    End Try
End Sub
```

In this example I allow the user to choose the new name which is then applied. The result is then applied to the property, which is then saved with the Put method. You will of course need the required permissions on your computer to run this sample.

Deleting Instances

We have seen how easy it is to retrieve class instances and make use of them. However, there will be times when you'd like to remove an instance too. To remove an instance, we simply get the instance object and call the Delete method. The following line of code is all that is required once you have obtained the instance of the class:

```
objWMIClass.Delete()
```

Once you call this method the instance is removed and cannot be restored so consider carefully before deleting instances. There is no way to know if an instance can be deleted or not, however it is reasonable to expect that if a class supports creation of instances, it also supports their deletion.

Writing Data Asynchronously

In previous example where you changed the volume name of the local drive you knew that the action would be done fairly quickly as it's done locally. However, assume that the class property you wish to change will take an unknown amount of time to execute and you'd rather not have your users wait for the process to finish. In this case you can execute the Put method asynchronously by proving a ManagementOperationObserver class which defines the method that will be called when the process is complete. So modifying the previous example we end up with the following:

```
Shared Sub ModifyVolumeNameAsync()
    Dim objScope As New ManagementScope("\\.\root\cimv2")
    Dim objWMISearcher As New ManagementObjectSearcher(objScope, _
      New SelectQuery("SELECT * FROM Win32_LogicalDisk where DeviceID='C:'"))
    Dim objDrive As ManagementObject
    Try
      Dim objObserver As New ManagementOperationObserver()
      AddHandler objObserver.Completed, _
        AddressOf WQL_Samples.ModifyVolumeNameAsyncComplete
      For Each objDrive In objWMISearcher.Get
        objDrive.Properties("VolumeName").Value = _
          InputBox("Enter New Name for Drive C:", "Change Caption")
        ' Here we tell the method how to handle the async operation
        objDrive.Put(objObserver)
      Next
    Catch ex As Exception
      MessageBox.Show(ex.Message)
    End Try
End Sub
Shared Sub ModifyVolumeNameAsyncComplete(ByVal sender As Object, _
    ByVal e As CompletedEventArgs)
    MessageBox.Show("Drive Volume update is complete.")
End Sub
```

The core difference here is that we pass a delegate object to the put method and have a method ready to be notified when the process finishes. This example also shows that you can use shared methods instead of instance methods for your delegates. How delegates work is covered in more detail in the MSMQ Chapter that follows and in the online help.

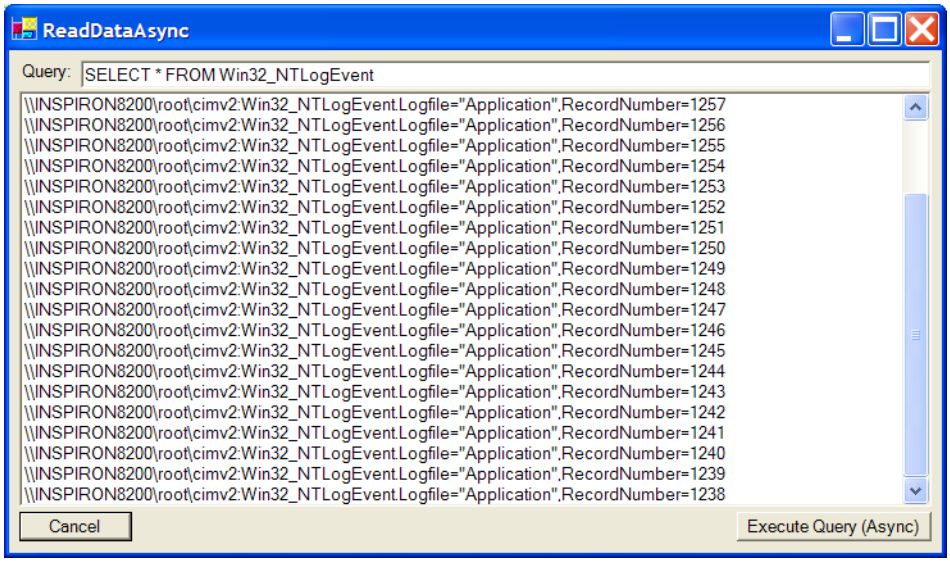

Figure 3-1. Exposing Data

Reading Data Asynchronously

WMI also supports the asynchronous reading of data too. The process is a little different from updating but does require the use of the ManagementOperationObserver class. You have two main events which you can make use of:

* **ObjectReady**: This fires anytime a full object has been received ready for processing. This is ideal for processing data as it becomes available.
* **Completed**: This fires when all the data has been received.

The following example demonstrates how to read data from the event log asynchronously, and even gives the user the option to stop the reading at any time. This gives a much better user experience as the user may not wish to see all 10,000 entries in your event log.

This example makes use of the ObjectReady event as it's the most useful for asynchronous processing; see Figure 3-1. Although you can use AddHandler to define the events, I've gone with the more VB way by using WithEvents. They are both acceptable and C# developers would have to choose AddHandler as they have no other choice:

```
Public Class ReadDataAsync
    Inherits System.Windows.Forms.Form
    WithEvents m_objObserver As New ManagementOperationObserver()
    Dim m_blnProcess As Boolean = False

    Private Sub btnExecute_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles btnExecute.Click
      ' This is where all the good stuff is!
      Try ' Errors in your WQL are common so we will watch for that.
        ' Define where to run our WQL Query
```

```
        ' Set the class level variable to say we are currently processing.
        m_blnProcess = True

        ' Define the query to be run Async.
        dim objWMISearcher as ManagementObjectSearcher = _
            New ManagementObjectSearcher(Me.txtQuery.Text)
        Dim objWMIClass As ManagementBaseObject

        ' You can use this method below if you don't want to use WithEvents
        'AddHandler m_objObserver.ObjectReady, _
        '    AddressOf ReadDataAsync.ModifyUserNameAsyncComplete

        Me.lstResults.Items.Clear() ' Clear previous results
        ' Execute the query – results retrieved by the ObjectReady event
        objWMISearcher.Get(m_objObserver)
    Catch ex As Exception
        ' Opps we have an error
        MessageBox.Show("ERROR: " & ex.Message)
    End Try
End Sub

Private Sub m_objObserver_ObjectReady(ByVal sender As Object, _
        ByVal e As System.Management.ObjectReadyEventArgs) _
        Handles m_objObserver.ObjectReady

    ' Add the returned data to the listbox on the form.
    Me.lstResults.Items.Add(e.NewObject.ToString)

    ' This can lead to a tight loop as all the results come in.
    Application.DoEvents()

    ' Check if we should be continuing processing of the events
    If Not m_blnProcess Then
        ' Stop the processing of events by cancelling the operation
        CType(sender, ManagementOperationObserver).Cancel()
    End If
    ' Move the selected item to the last position so we can see it.
    Me.lstResults.SelectedIndex = Me.lstResults.Items.Count
End Sub

Private Sub btnCancel_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles btnCancel.Click
    ' Set the class level variable to False, this will be examined each time
    ' the ObjectReady event is fired, so we can stop processing at any time.
    m_blnProcess = False
```

```
    End Sub
End Class
```

The core differences between how you would receive data synchronously compared to asynchronously is the inclusion of the ManagementOperationObserver class. When passed as an argument of the Get method, it fires events as data is being collected. The only thing you need to do is hook up to the events by using either WithEvents as I've done in the example or using AddHandler. Once the event fires you get a reference to the newly created object via e.NewObject, which represents the WMI Class returned by the query. I've included a little extra code to allow you to stop processing by setting the m_blnProcess variable to False.

Making use of the asynchronous features of WMI can allow you to build more responsive applications, which don't require a great deal of extra effort. As an exercise I've left the Query Tool synchronous for you to upgrade to the asynchronous model.

Security

When instrumenting a provider in .NET, it uses a new WMI feature known as a decoupled provider subsystem which enables embedding the provider into the application; allowing for more efficient operation. This permits WMI to interact with the application directly instead of indirectly through the program's API. Decoupling the provider from WMI also puts the application in control of the provider lifespan, instead of WMI.

> **Tip:** If you are running in a workgroup, you may need to override the default settings for remote network access to allow remote calls to work. Navigate to Control Panel>Administrative Tools>Local Security Policy. From here navigate to Local Policies>Security Options. Then change the Network access: Sharing and security model for local accounts to the classic option. This allows you to use a username and password to access the remote computer. The guest account doesn't have sufficient rights to run WMI queries.

Classic providers that are loaded and unloaded by WinMgmt and run in the controlled host are allowed and actually encouraged to impersonate the calling client when retrieving information for this client. In the case of a decoupled provider, it's hosted in an application that can be run by any user, impersonation is not allowed and only identify-level connection is supported. The provider always operates in the context of the user who is running the application, and performs an access check for the identity of the calling client before forwarding the management information requested. The decoupled provider also provides in the registration mechanism a security descriptor to define the users who are allowed to provide information for this application.

Providers load into the provider subsystem within the NetworkService security account. This account is intended for services that have no need for extensive privileges but have the need to communicate remotely with other systems. By using this account the potential risk that a corrupted or compromised provider could take out the entire computer (or domain, in the case of a domain controller) is eliminated. It also ensures that

no privileged information is exposed to a user in case the provider does not properly impersonate the client's context.

.NET Classes

When executing queries in WMI, you can adjust the connection options on the ManagementScope class. You can supply details of a remote computer as well as a username, password and connection settings.

```
Dim objScope As ManagementScope
objscope = New ManagementScope("\\remotecomputer\namespace")
With objScope.Options
   .Username = "Fred"
   .Password = "Barny"
   .Impersonation = ImpersonationLevel.Impersonate
   .Authentication = AuthenticationLevel.Unchanged
End With
```

Having defined the scope, you then pass this object into the ManagementObjectSearcher class. The important thing to understand is the connection between ManagementScope and the ManagementObjectSearcher class.

```
Dim objWMISearcher As ManagementObjectSearcher
objWMISearcher = New ManagementObjectSearcher(objScope, objQuery)
```

Tables 3-1 and Table 3-2 detail the options available when passing authentication details.

Table 3-1. ImpersonationLevel enumeration

Member name	Description
Anonymous	Anonymous COM impersonation level that hides the identity of the caller. Calls to WMI may fail with this impersonation level.
Default	Default impersonation.
Delegate	Delegate-level COM impersonation level that allows objects to permit other objects to use the credentials of the caller. This level, which will work with WMI calls but may constitute an unnecessary security risk, is supported only under Windows 2000.
Identify	Identify-level COM impersonation level that allows objects to query the credentials of the caller. Calls to WMI may fail with this impersonation level.
Impersonate	Impersonate-level COM impersonation level that allows objects to use the credentials of the caller. This is the recommended impersonation level for WMI calls.

Table 3-2. AuthenticationLevel enumeration

Member name	Description
Call	Call-level COM authentication.
Connect	Connect-level COM authentication.
Default	The default COM authentication level. WMI uses the default Windows

	Authentication setting.
None	No COM authentication.
Packet	Packet-level COM authentication.
PacketIntegrity	Packet Integrity-level COM authentication.
PacketPrivacy	Packet Privacy-level COM authentication.
Unchanged	Authentication level should remain as it was before.

Adding WMI Support

Now that you know much of what there is to know about querying, modifying and deleting WMI classes its time to build your own. The process of adding WMI support to your own objects and applications is known as Instrumentation; making them easy to diagnose and profile. This section starts by introducing you to what is required to instrument an application. I will then demonstrate how you might use this in the real world. Later Chapters will extend this to cover more specific topics such as performance counters and scheduling.

Instrumenting Your Applications

When you add support for WMI to your own applications it is known as instrumentation; which is the process of adding management events, performance counters, and trace information to an application. This allows monitoring tools to track the current status and performance characteristics of your application. You can use instrumentation to provide the following support to your applications:

* Performance analysis and runtime profiling.
* Problem diagnosis.
* Expose application data.
* Application configuration.

There are a number of classes provided by .NET that aid in the instrumentation of our applications. These classes help to abstract even further the underlying detail of WMI.

The classes provided in the System.Management.Instrumentation namespace reduce the amount of work required to give your applications WMI support; they also help you to generate WMI events. This namespace allows you to do the following:

* Instrument an application.
* Expose application (delegate-based) events as WMI events.
* Create manageable objects.
* Define and use relationships between manageable objects.

WMI uses an object-oriented schema which has many similarities with .NET metadata. This makes it very intuitive for .NET programmers and makes the process of exposing application objects easier to understand. This similarity also allows application objects to be mapped directly to WMI objects, which makes instrumenting the application relatively easy. You could make use of instrumentation to:

* Send events from components within your application.
* Provide objects that aid in application configuration.
* Expose runtime data; for example, performance characteristics.

Providing this type of instrumentation within your application greatly assists problem diagnosis and can be used to quickly alert operators to conditions that require immediate attention. Automating responses to problem situations can also be developed which consumers can subscribe to via the WMI event model. The use of performance metrics can be used to locate bottlenecks within your application and diagnose problems associated with your application.

.NET Classes

The .NET framework provides a number of classes and features that allow you to add instrumentation to your applications with relative ease. Below are the main namespaces and their role within instrumentation:

* **System.Management.Instrumentation:** Provides classes and attributes to help instrument .NET applications.
* **System.Diagnostics:** Provides support for developing custom performance counters. This will be covered in-depth in Chapter 8.
* **System.Management:** Provides a set of managed-code classes through which .NET applications can easily access and manipulate management information. This has already been covered in the previous Chapter.

Figure 3-1 shows how these classes relate to the overall architecture. Although not shown in the figure the System.Management.Instrumentation namespace is used to provide both data and to publish events to the CIMOM. This namespace handles all the hard work for us and as you will see next, makes the process of adding WMI support almost trivial.

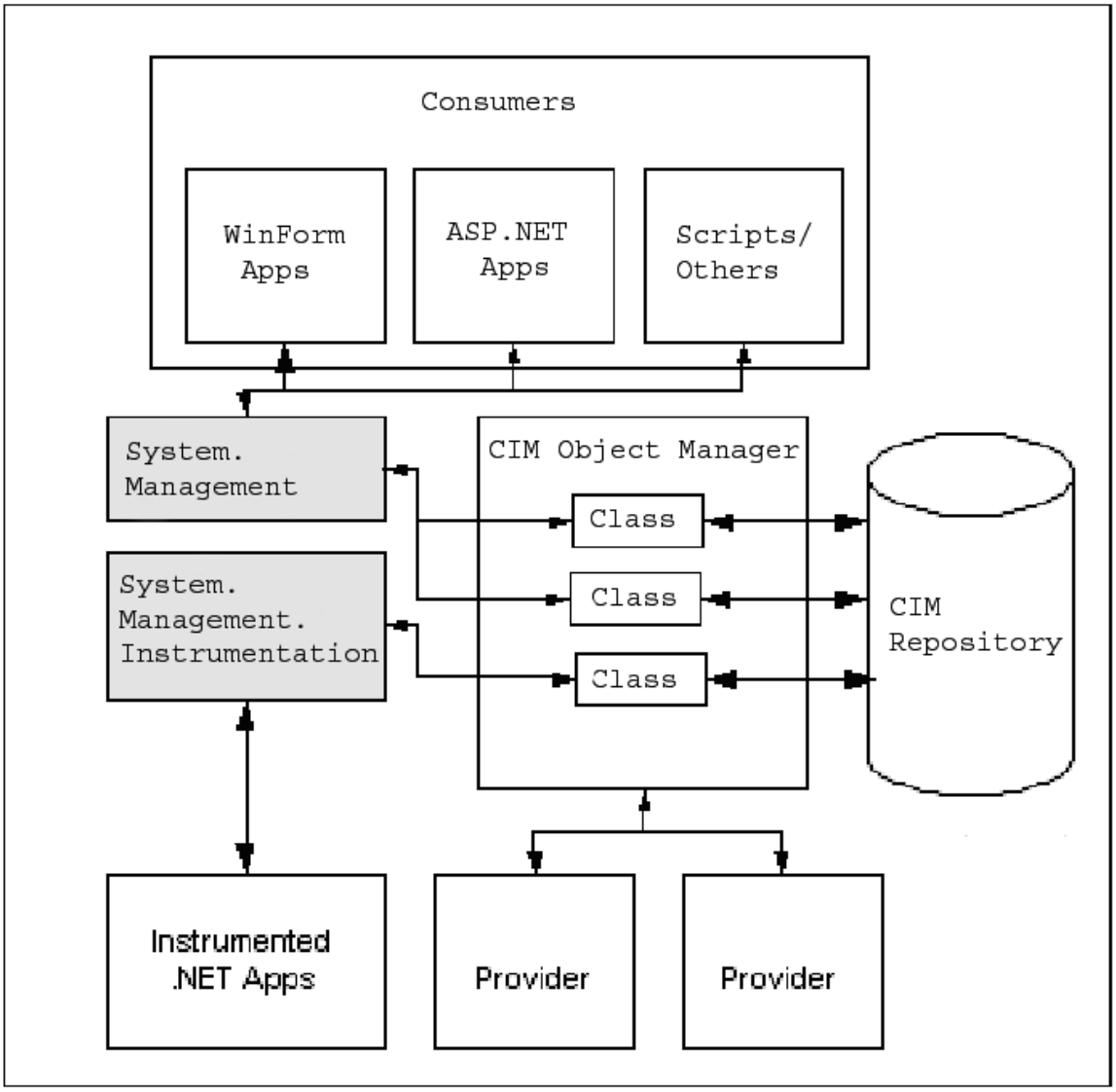

Figure 3-1. WMI Architecture

Exposing Data

One of the powerful features that the .NET WMI classes provide is the ability to expose your own objects data. Suppose that you have a web application that has an object for each currently logged on user. Now, assume you want to know who is currently logged on. To do this you could write yourself a web page which makes use of the object instances on the server. You may also code into your objects some type of remoting, which you could then program against. These options will work however they do require a bit of extra work; especially if the administrator then asks to be able to apply criteria.

Another alternative would be to make your objects WMI enabled and expose the class instances through WMI. By adding a few lines of code to your classes you are able to expose them to any WMI compliant application, including the query tool we built in the

previous Chapter. Providing this level of functionality previously, that is pre .NET was complicated and required C++ to do. Now with the advent of .NET you can provide this level of functionality to your objects with only a few lines of code, which are even easy to understand!

To do this, you add to your .NET class the InstrumentionClass attribute thereby identifying it as a managed WMI class alternatively you can inherit from the Instance class found in the System.Management.Instrumentation namespace. This then automatically generates the WMI schema associated with your managed code class. Instances of the class are exposed as WMI instances with all of the property values mapped, even references between objects of the class and objects of other classes in the application are mapped to WMI associations (which are relationships between management objects defined in schema).

The following example demonstrates our scenario. The code is kept as simple as possible so you can understand the WMI features being used:

Want to highlight the WMI specific areas in bold or similar

```
Option Strict On
' 1 & 2. Define your references
Imports System.Management
Imports System.Management.Instrumentation

' 3. Define the namespace for your objects
<Assembly: Instrumented("root\Samples")>

' This class is self contained and is used to automatically install all the
' WMI Schema data into the CIM. You must however use the InstallUtil.exe
' for this class to run.

' 4. Define the code for the Installer class: Simple copy and paste
<System.ComponentModel.RunInstaller(True)> _
Public Class SalesInstaller
    Inherits DefaultManagementProjectInstaller
End Class

' This is the main class which is used to log users onto the system.
Public Class LoggonUsers
' 5. Inherit from the base class
    Inherits Instance
    Dim m_colUsers As New ArrayList()

    Sub New()
        ' This line registers the current instance with WMI, so its vital!
        ' Also it should be the LAST line when initialising your object.
        ' 6. Publish instance to WMI
```

```
        MyBase.Published = True
    End Sub

    Public Sub Logon(ByVal Username As String)
        m_colUsers.Add(New User(Username))
        Dim objNewUserEventHandler As New NewUserEventHandler()
        objNewUserEventHandler.Username = Username
        objNewUserEventHandler.Fire()
    End Sub

    Public ReadOnly Property Count() As Integer
        Get
            Return m_colUsers.Count
        End Get
    End Property

    '<IgnoreMember()> _
    Public ReadOnly Property Items() As Object
        Get
            Return m_colUsers
        End Get
    End Property
End Class

' This class is only using Attributes to mark it as
' a WMI managed class. Allowing it to inherit from
' another class if required.
' 5. Use attributes to make use of the instrumentation class
<InstrumentationClass(InstrumentationType.Instance)> _
Public Class User
    Private m_strUsername As String
    Private m_dtmLoggedInAt As DateTime

    Sub New()
        Me.New("") ' Supply default user of Blank
    End Sub

    Sub New(ByVal Username As String)
        m_dtmLoggedInAt = DateTime.Now()
        Me.Username = Username

        ' This line registers the current instance with WMI, so its vital!
        ' Also it should be the LAST line when initialising your object.
        ' Note: this uses a different technique to the LoggonUsers object.
        ' 6. Publish instance to WMI
        System.Management.Instrumentation.Instrumentation.Publish(Me)
```

```
    End Sub

    Public Property Username() As String
      Get
        Return m_strUsername
      End Get
      Set(ByVal Value As String)
        m_strUsername = Value
      End Set
    End Property

    ' This property is dynamic and will change depending on the
    ' amount of time the user has been logged in.
    ReadOnly Property MinutesLoggedIn() As Long
      Get
        Return DateTime.Now.Subtract(m_dtmLoggedInAt).Minutes
      End Get
    End Property
  End Class
```

This example uses two objects one to hold the collection of individual users (LoggonUsers) and the other the actual individual user (User) object. You can associate a user as being logged on with the following code:

```
m_objUsers.Logon(Me.txtUsername.Text)
```

Notice that I use a module level variable m_objUsers when I call the Logon method of the LoggonUsers class. This is because you need to keep a reference to any of the instance objects you create, this is important because WMI is only a layer onto of your objects. If your object instances go out of scope or are destroyed then the WMI layer will not be able to expose them as they will not exist. This makes sense; how can WMI expose instances of objects that don't exist. To that extent once you close your application down WMI looses the references to the objects you've created.

Now if we examine the code it demonstrates that the following steps are required to expose your data using WMI:

1. Add a reference to System.Configuration.Install. This is required for the DefaultManagementProjectInstaller class which registers your objects with WMI.
2. Add a reference to System.Management. This is required for all the instrumentation and WMI classes.
3. Define the namespace where you want your objects to reside. If you don't supply this, they will be put into root\Default by default:

   ```
   <Assembly: Instrumented("root\Samples")>
   ```

4. To ensure that your objects can be installed into the CIM or WMI you must include an installer class. The good news is you only have to inherit from a base class which contains all the necessary code:

   ```
   <System.ComponentModel.RunInstaller(True)> _
   ```



```
Public Class SalesInstaller
Inherits DefaultManagementProjectInstaller
End Class
```

5. Now to enable WMI support; you have two choices. You can either inherit from the base class Instance or apply attributes which will do the same thing; if you have already inherited from another class:

```
' Inheriting from System.Management.Instrumentation.Instance
Public Class LoggonUsers
Inherits Instance
…
End Class

' Using Attributes serves the same purpose as inheriting from Instance
<InstrumentationClass(InstrumentationType.Instance)> _
Public Class User
…
End Class
```

6. The only thing left to do now is to inform WMI when an instance of your object is ready to be published, and therefore exposed by WMI. How you do this depends on how you enabled WMI support as shown below:

```
' When inheriting from System.Management.Instrumentation.Instance
Sub New()
…
MyBase.Published = True
End Sub

' When using Attributes
Sub New()
…
System.Management.Instrumentation.Instrumentation.Publish(Me)
End Sub
```

7. Although that covers everything you need to do with the code itself there is one last action that needs to be done. Firstly, build your solution to generate an exe or dll. Next run the following command line utility against it; you may need to use the .NET command prompt otherwise the file may not be found in your DOS path.

```
installutil <assembly dll or exe>
```

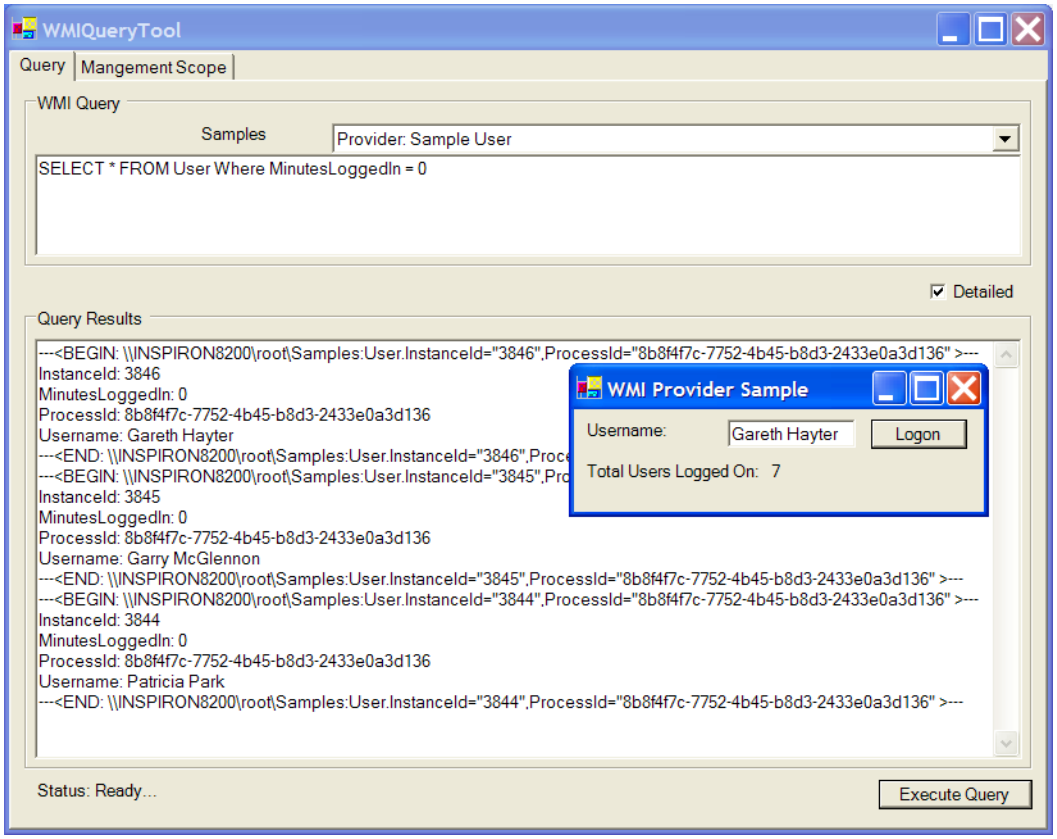

Figure 3-3. Exposing Data

Once you have completed all the above steps you are now ready to start creating instances of your objects. The sample demonstrates this by building a simple GUI that registers a user by calling the Logon method of the class we just exposed to WMI. You will find this example on the main form under the WMI Provider Examples, where you can select Add New Users. This will bring up the dialog as shown in Figure 3-3. Once you have created a few users, you can then use the query tool to start querying the objects you've added which will now be available through WMI.

> **Note:** As of version 1.0 of the framework the instrumentation classes won't expose the methods of your classes. It's hoped that this will be introduced in a later version of the framework.

Looking at Figure 3-3, you can see that I have a total of 7 users logged on which I added using the form. Now assume I'm an administrator who needs to know how mnay users have logged onto the system in the last minute. Our original code didn't provide any functionality to return this information. However, using WQL and WMI we can run a simple query against our objects to gain this information. The following query was used to return back a total of 3 users out of the 7; results of which can be seen in Figure 3-3.

```
SELECT * FROM User Where MinutesLoggedIn = 0
```

This simple query demonstrates the power that was given to our objects for free. It only required a few extra lines of code to make them WMI enabled. If should be noted

that if our application was installed on a remote computer and we had the necessary access rights, we could query it just as easily without having to install any extra software. Next we will add another exciting aspect of WMI to our objects, events.

Exposing Events

Events are probably one of the most exciting aspects of WMI in that it allows you to notify consumers of your objects when a specific condition changes or of some other event. This notification will also work across the enterprise thanks to the built in remoting capability of WMI. You use similar techniques for exposing events as you do to expose object data.

Expanding upon the previous example, I will add an event that will fire anytime a user logs on. This event can then be monitored in the same way as was discussed previously in the Event Queries topic in the previous Chapter.

The following code only shows the core changes from the previous example; I will go over the differences in detail afterwards:

```
' This is an Event Class used to fire a custom event that
' will be exposed by WMI.
Public Class NewUserEventHandler
    ' 5. Inherit from BaseEvent for your custom events
    Inherits BaseEvent
    Public Username As String
End Class

Public Class LoggonUsers
    …
    Public Sub Logon(ByVal Username As String)
        m_colUsers.Add(New User(Username))
        Dim objNewUserEventHandler As New NewUserEventHandler()
        objNewUserEventHandler.Username = Username
        ' 6. Inform WMI that the event has fired
        objNewUserEventHandler.Fire()
    End Sub
    …
End Class
```

Now looking at the code it demonstrates the following steps required to expose an event using WMI:

Steps 1 to 4 of exposing data are still required when exposing events.

5. Next we have two options which are very similar to those when exposing data. You can either inherit from the base class BaseEvent or apply attributes which will do the same thing; if you have already inherited from another class:

```
' Inheriting from System.Management.Instrumentation.BaseEvent
```



```
Public Class NewUserEventHandler
Inherits BaseEvent
…
End Class

' Using Attributes serves the same purpose as inheriting from BaseEvent
<InstrumentationClass(InstrumentationType.Event)> _
Public Class NewUserEventHandler
…
End Class
```

6. Now all you need to do is fire the event at the appropriate time, by creating an instance of the event object. You can then either call its Fire method if you've inherited from EventBase or use the shared Fire method of the Instrumentation class by passing the event object as a parameter if you used attributes.

```
' When inheriting from System.Management.Instrumentation.BaseEvent
Dim objNewUserEventHandler As New NewUserEventHandler()
objNewUserEventHandler.Username = Username
objNewUserEventHandler.Fire()

' When using Attributes
Dim objNewUserEvent As New NewUserEventHandler()
objNewUserEvent.Username = Username
System.Management.Instrumentation.Instrumentation.Fire(objNewUserEvent)
```

7. Although that covers everything you need to do with the code itself there is one last action that needs to be done. Firstly, build your solution to generate an exe or dll. Next run the following command line utility against it.

```
installutil <assembly dll or exe>
```

I will cover how you can monitor these custom events from within VS.NET in the next section. Although I've mentioned this a few times, it's worth reiterating that this new event can be used remotely without the need to write stubs or proxies. If you allow yourself to think about the possibilities this brings you may rethink some of your architectures.

WMI Server Explorer Extensions

Microsoft has been working on extensions for Server Explorer to make it easier to use WMI resources. At the time of writing they have a Beta 2 of the extensions for download to work with VS.NET 2002 and have just released an RTM version for VS.NET 2003. You can see from Figure 3-4 what it looks like when installed.

As you can see there are a number of providers available similar to the ones mentioned earlier. You can also see how easy it is to identify parts of the system simply by navigating through the tree. Click on a class to see the properties that relate to that class in the properties window.

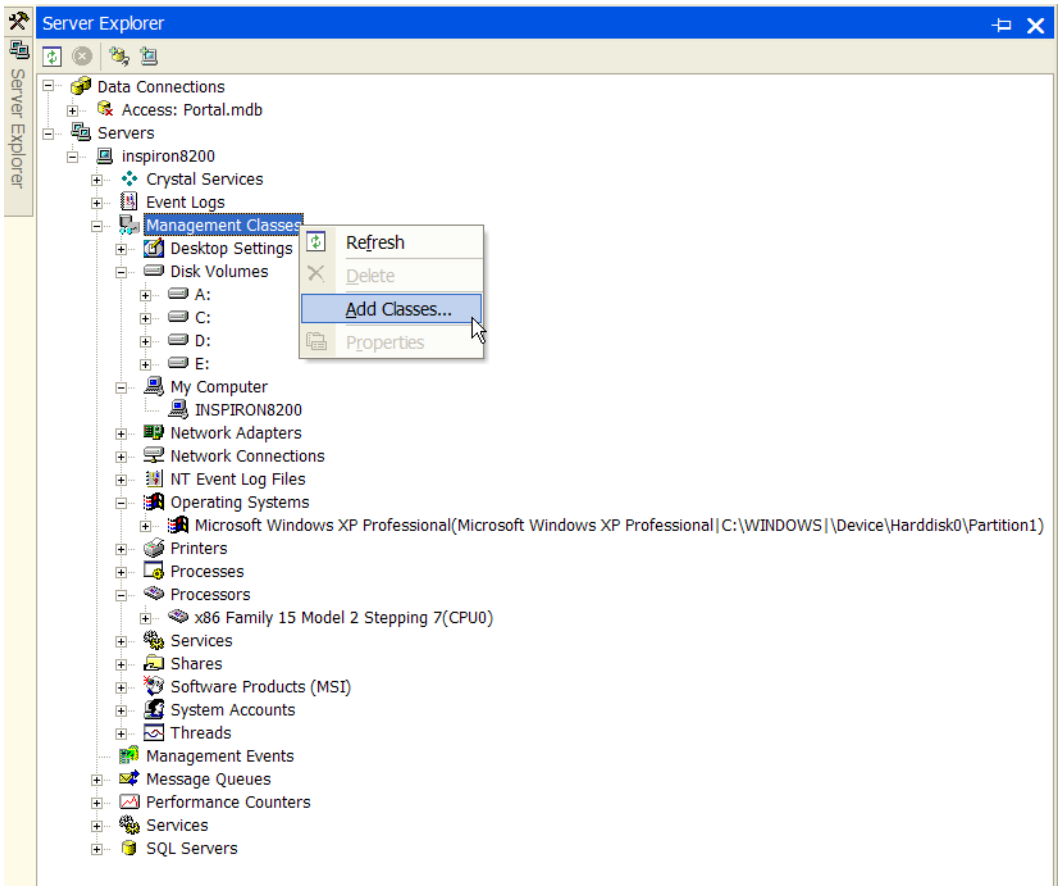

Figure 3-4. WMI Server Explorer Extensions

Although there is a host of information available, as I've mentioned earlier this is only the tip of the iceberg and the WMI extensions allow you to see the whole iceberg if you want. This includes your own custom providers like the one we just built. You can add extra WMI classes by using the Add Classes context menu as shown in Figure 3-4. This will bring up the Add Classes dialog which allows you to browse the available WMI classes. Given that few people are very familiar with the WMI namespace structure, the Add Classes dialog provides a very handy search feature. Let's take as an example you wanted to know the details of the batteries on your laptop. Typing the keyword 'batteries' returns the Win32_Battery class within the root\cimv2 namespace as shown in Figure 3-5. If you don't find what you want on the first find, pressing the find button again will continue the search through all available namespaces.

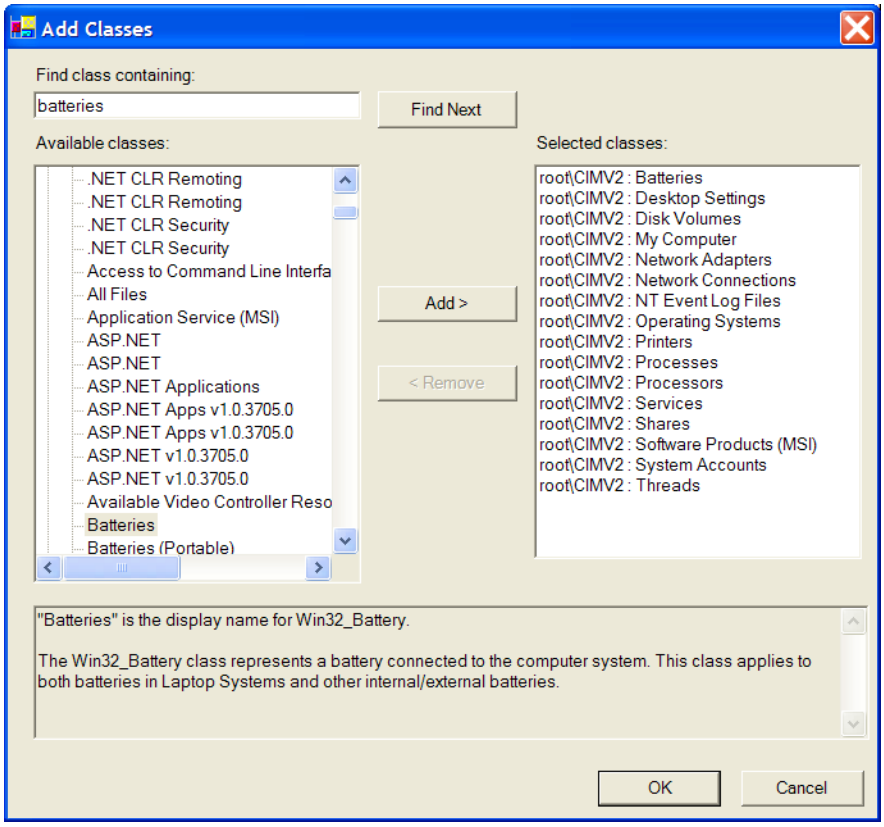

Figure 3-5. Searching WMI Namespaces

Although not shown in Figure 3-5, there are many other namespaces other than root\cimv2 and when you first open the Add Classes dialog you'll see all the available namespaces. If you download the WMI SDK you will also get a number of tools that come with WMI such as an object browser. You may find these useful when trying to find particular class resources. However, given that the WMI extensions are integrated into VS.NET and the searching is easier, my preference is to use VS.NET.

Once you have added a class to Server Explorer, you will be able to see its properties in the Property Browser and expand its nodes using Server Explorer. When you expand an object node you can reveal other management objects that are semantically related to that one. For example by expanding a process node you will be able to see which DLLs have been loaded by that process.

Strongly Typed Classes

One of the problems with the WMI object model that Microsoft has tried to alleviate with WMI extensions is the lack of early binding. As mentioned before WMI whether you're using the COM or System.Management class versions all suffer from late binding. This means that object properties are accessed through a properties collection, and its methods are accessed through a methods collection with all values being returned as System.Object. Although this is good for generic management applications it does prove problematic for developers by not having any type checking and requires more supporting

documentation. With WMI extensions you have a new feature which will create an early bound wrapper class for any resource you try to use. That is instead of having to remember all the late bound properties for a class you will have a strongly typed managed class with the relevant properties and methods. This then allows you to take advantage of IntelliSense and early binding and so reducing problematic code.

So the code that would have been written to retrieve the free disk space available on a disk through a late-bound object (of type ManagementObject) in VB.NET would be like this which can be found in the Server Explorer Extensions section of the samples.

```
Shared Sub LateBound()
    Dim objWMISearcher As New ManagementObjectSearcher_
                    ("SELECT * FROM Win32_LogicalDisk WHERE DeviceID='C:'")
    Dim objDrive As ManagementObject

    For Each objDrive In objWMISearcher.Get
      MsgBox("You have :" & objDrive.Properties("FreeSpace").Value. _
                    ToString & " bytes free.")
    Next
End Sub
```

Now if you contrast this to a strongly typed version then you could write the code to be more like this:

```
Shared Sub EarlyBound()
    Dim objLogicalDisk As New ROOT.CIMV2.LogicalDisk("C:")

    MsgBox("You have :" & objLogicalDisk.FreeSpace.ToString & _
                    " bytes free.")
End Sub
```

The early bound version is obviously easier to read and would require little documentation to support it, as it's very intuitive. The WMI extensions provide an easy method for creating these strongly typed wrappers by dragging a node onto a form or a component in design mode. This action generates a wrapper which is added to your project. A strongly typed data field is then added to your form which is then bound to the specific WMI object that you had selected for the drag and drop operation.

> **Tip:** The code that generates the strongly typed class is actually part of the Framework. You can call the method GetStronglyTypedClassCode on the System.Management.ManagementClass class. This will allow you to generate the class code for VB.NET, C# or Jscript depending on your parameters.

In the example, you select the logical disk node from Server Explorer and drop it onto your form; then a file named Win32_LogicalDisk.vb, which contains the definition of the wrapper class, is added to your project. The resulting code for the LogicalDisk class works out at almost 2,000 lines of code! In addition to this, the LogicalDisk1 field is added to your component, then instantiated and bound to a specific process inside the

InitializeComponent method. If however, you didn't want to create the wrapper with a GUI interface i.e. you wanted to write a console application. Then you can still create the wrapper by using the Generate Managed Class context menu of the class you're interested in see Figure 3-6. Then you can simply create the object like any other, which is what I have done for the "WMI Server Explorer Extensions" examples. From a maintenance viewpoint the strongly typed version wins hands down, although it can add a little code bloat.

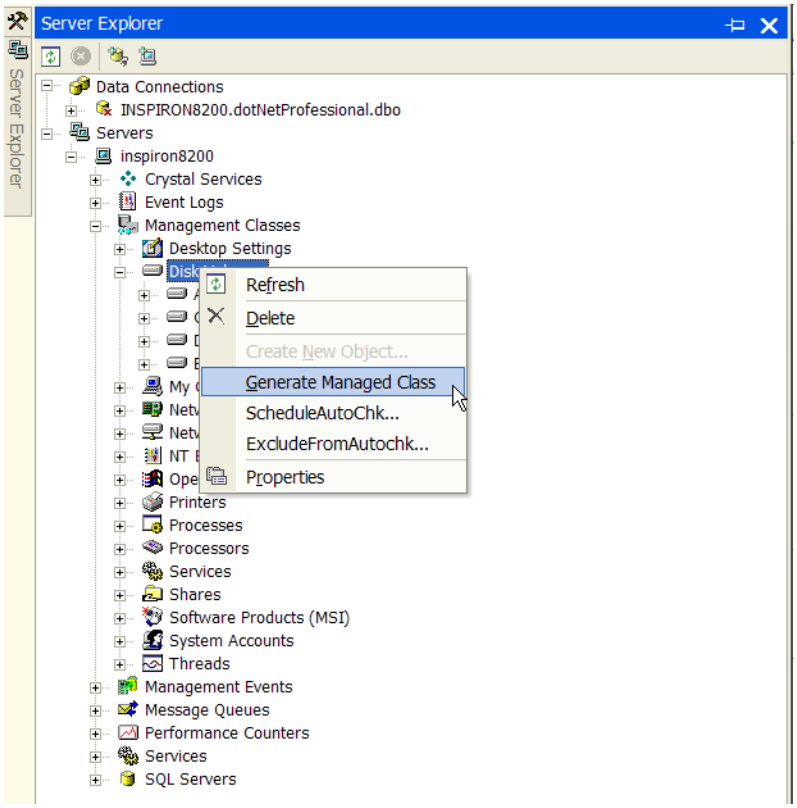

Figure 3-6. Building a WMI Query

Management Events

The WMI extensions also provide access to the events that WMI provide called management events. You can access this from the Management Events node in Server Explorer. Events can be quite useful in a variety of development scenarios. You can monitor for specific system changes in distributed applications, be notified when specified thresholds are crossed, and view the events that your instrumented application fires, or test a WMI event provider you are developing. The really useful part of this is that once you register to listen for an event the event results are displayed within Server Explorer, this means you don't actually need to code anything to test your custom events.

By default the Management Events node doesn't have any children. You can add an event by selecting the Add Event Query from the context menu.

Building Queries

You can use the Management Events feature of Server Explorer for several purposes:

* Test event handlers in custom providers
* Monitor an existing event handler in a WMI class

You can create a query that will always notify you of an event by using the Managed Events node in Server Explorer. Selecting the context menu Add Event Query, displays the dialog in Figure 3-7.

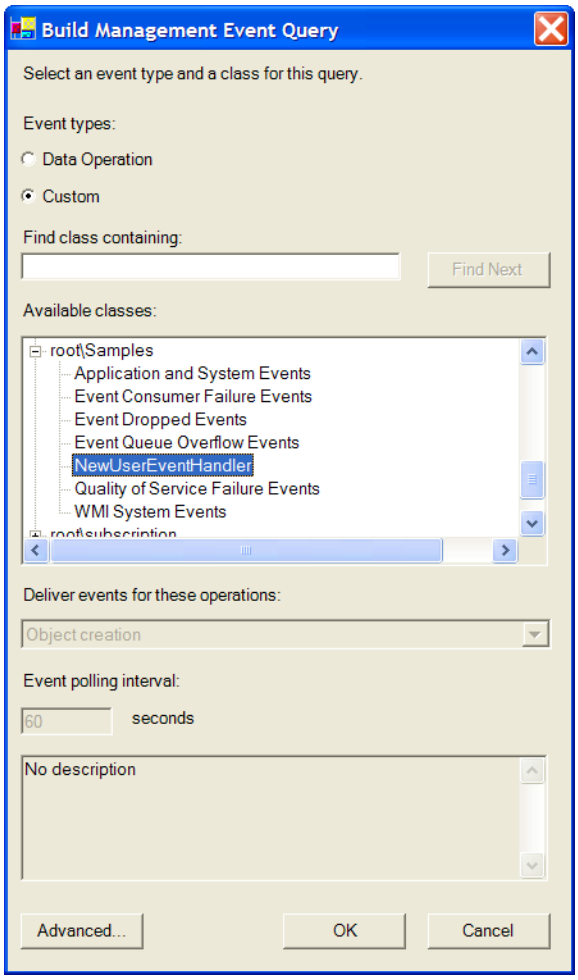

Figure 3-7. Building a WMI Query

If you want to be notified of a system or intrinsic event select the Data Operation option for the event type. However given that you've created your own custom provider which has a custom event, which is an extrinsic event type we will select custom. Once you change the event type to custom the dialog will fill with all the possible extrinsic events for all the classes. Navigating to your custom class under the root\samples namespace, shows all the events available for that namespace. You'll notice that the event

NewUserEventHandler is also there, which is the one we will register. By selecting the event you're interested in and then clicking OK, you have setup Server Explorer to respond to that event. The event query you have created using this UI will appear as a child node under Management Events and will start listening for the appropriate events. You can test this by running the Add New Users sample.

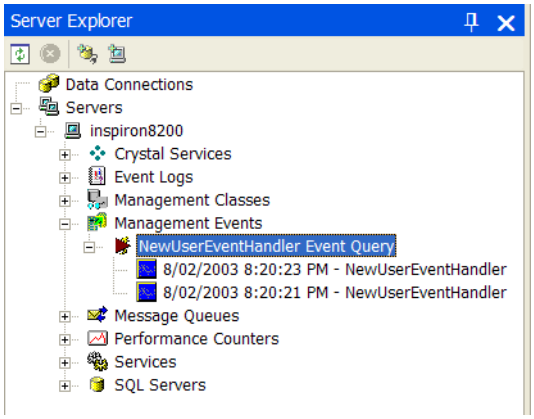

Figure 3-8. Receiving event nofifications in Server Explorer

It is important to note that the query builder UI is somewhat limited and cannot produce some valid WQL event queries (for example, it assumes that the query conditions are always simply constructed using the logical AND). If you need to create more sophisticated query expressions using a logical OR or parentheses, you can always modify the query string by changing the query property of the event query node in the Property Browser window. This will stop the current event subscription and start a new one.

Once an event matching your query criteria arrives, it will be printed to the output window in Visual.Studio.NET, and added to the event node (see Figure 3-8). You need to select Refresh from the event node's context menu to see the event as a child to the event node. You can stop and start event subscriptions from the context menu of the event node. You can also clear out the old event nodes this way.

Receiving Event Notifications

Now that we have added an event query to Server Explorer and have tested that it works, we can take this one step further. As with anything that is available in Server Explorer, you can drag and drop the event query you have created and drop it onto a form. Doing this adds an instance of the System.Management.ManagementEventWatcher class to your component and sets it to the query that you built. During my testing however it appears that it doesn't work as advertised (remember this is a Beta); as it seems to use the default namespace instead of the one you've defined. This however is a small inconvenience; drop the query onto a form; name it wmiNewUserEvent; then apply the following code to get it all working.

```
' Initialize the object
Me.wmiNewUserEvent.Scope = New Management.ManagementScope("\\.\root\Samples")
```



```
Me.wmiNewUserEvent.Start()

' Define the method that will accept the event
Private Sub OnNewUser(ByVal sender As Object, ByVal e As _
   System.Management.EventArrivedEventArgs) Handles _
   wmiNewUserEvent.EventArrived

   MessageBox.Show("The following user just logged in: " & _
      e.NewEvent.Properties("Username").Value.ToString)
End Sub
```

This code firstly initializes the instance of the ManagementEventWatcher class by specifying the correct management scope; which is not set after the drag and drop. Next we tie the EventArrived event of the ManagementEventWatcher instance to the method which will be executed; I've used the Handles keyword, however C# programmers would have to use an AddHandler statement instead. Now when a new user is added, you will see a dialog box alerting you to the fact a new user has logged on.

Summary

This Chapter built on the knowledge we gained in the previous Chapter, and we learned that WMI is not only powerful out of the box, but can be made more powerful by using instrumentation. The ability to make our applications part of the WMI system allows us to develop sophisticated applications quickly and easily. We can extend our applications so that WMI tools can query our applications data and even register for events that our application has.

Although mentioned a few times, it's worth reiterating that Microsoft is not the only supplier of WMI providers. This could make using third party products easier, especially if they don't provide a decent event model in their API.

Resources:

WMI Managed Extensions for VS.NET 2003:

http://www.microsoft.com/downloads/details.aspx?FamilyID=62d91a63-1253-4ea6-8599-68fb3ef77de1&DisplayLang=en

COM API for WMI:

http://msdn2.microsoft.com/en-us/library/Aa389276.aspx

WMI SDK (a must have download!)

http://msdn.microsoft.com/downloads/sdks/wmi/default.asp

Update: http://www.microsoft.com/downloads/details.aspx?FamilyID=c2b1e300-f358-4523-b479-f53d234cdccf&DisplayLang=en (seems the original link no longer works and they have bundled it into the main SDK)

