6

The WMI Events Scripting

6.1 Objective

At this stage of the discussion, with the knowledge acquired from previous chapters about WQL and the WMI scripting API, we are ready to explore the WMI event scripting techniques. Up to now, each time we developed a script that runs on top of WMI, the code performed some actions with the manageable entities. The script did not perform monitoring functions. Some monitoring was performed using the permanent *SMTP* event consumer (see Chapter 2), but this did not involve scripting. Moreover, only one event type was used from the collection proposed by WMI. As WMI is designed to support event notifications, it makes sense that the event notification capabilities are also part of the WMI scripting API. The event scripting technique relies on the WMI event architecture, and some aspects of its usage are very similar to the asynchronous scripting techniques described in the previous chapter. The WMI event technology is probably one of the most powerful features offered by WMI. It offers different event types to trigger any action in an application. In this chapter, we examine the various event types available, how they are organized, and how to take full advantage of this technology from the scripting world.

6.2 Event notification

Managing an environment means more than simply performing tasks with the manageable entities; it also means being able to respond to situation changes. This implies that the technology used to perform the management tasks is able to detect changes. These changes are considered by WMI as events. WMI supports event detections and their delivery to WMI clients. Of course, since many changes may occur in a manageable environment, it is clear that only occurrences of interest must be reported. Moreover, if an

event is detected, WMI must know where to deliver the event. And last but not least, once the event is delivered, some actions must be taken regarding the detected change. These steps involve the presence of several mechanisms to support event detections.

WMI is able to detect changes by itself, such as changes in the CIM repository. Besides changes performed in the CIM repository, the real-world manageable entities are also subject to change. To help with these detection tasks, WMI uses some specific providers called event providers. For instance, the *SNMP* event provider, the *NT Event Log* event provider, and the *Registry* event provider are some typical WMI event providers.

To receive events, WMI clients must subscribe to WMI events. A WMI client can be an application such as **WBEMTEST.exe**, a script developed on top of the WMI scripting API, or any other applications developed on top of the WMI API. The WMI client, which creates the subscription, is called the subscription builder and provides two elements: the events in which the subscriber is interested and the WMI client request to be processed when the events occur. The subscription builder describes which event is of interest using a WQL event query. The WQL event query acts as a filter for events. For instance, in Chapter 3 we used an event query like

```
SELECT * FROM __InstanceModificationEvent WITHIN 2 Where
                                          TargetInstance ISA 'Win32_Service' And
                                          TargetInstance.Name='SNMP' And
                                          TargetInstance.State='Stopped'
```

6.2.1 Event subscriptions

When a client submits a WQL event query, it performs an event subscription. There are two subscription types: temporary subscriptions and permanent subscriptions. The subscription type determines the type of consumer used to perform an action triggered by a WMI event. So, let's examine the differences between the two existing subscriptions.

6.2.1.1 Temporary subscription

Temporary subscriptions are created by an application interested in receiving certain events. In the context of a temporary subscription, the application will receive events only if the application is running. If the application stops, the application will stop receiving events. When the application terminates, it must cancel its subscription to finish the event notification smoothly and properly with regard to WMI. A script written on top of the WMI scripting API is a typical example of an application performing a temporary subscription. Of course, it can be any other application type; tempo-

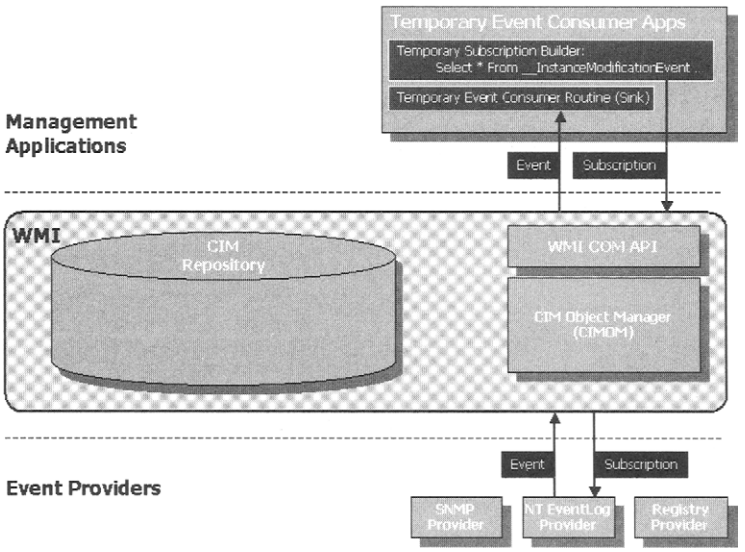

Figure 6.1
The temporary event consumer and its relation to the CIM repository and WMI.

rary subscriptions are not dedicated to scripts. Later, we see that an application can receive events synchronously, semisynchronously, or asynchronously. For instance, with an asynchronous notification, a temporary subscription includes a pointer to the application containing the sink that receives the events (with an **SWbemSink** object such as used in the previous chapter). This allows applications to handle received events and perform the related actions in parallel processing. Figure 6.1 represents a logical organization of the various items participating in a temporary event subscription using an asynchronous notification.

It is important to note that Figure 6.1 does not represent what happens when the event occurs. The figure logically differentiates the items to clarify their roles. The subscription builder contains the sink routine that receives events. The sink routine can be executed asynchronously and represents the temporary event consumer. In case of a synchronous or semisynchronous event, there is no explicit sink routine as with an asynchronous event notification. In this case the consumer waits for an event to occur. We consider the differences between these scripting techniques later in this chapter in Sections 6.5.1 and 6.5.2. In any case, regardless of the notification type, the consumer receives the WMI event. The consumer logic coding and WMI scripting API used determine the technique used to perform the action and how the consumer manages the received event.

Since the temporary subscription has a lifetime equal to the lifetime of the application, nothing related to the subscription is permanently or tem-

porarily stored in the CIM repository. A subscription filter (WQL event query) is passed to WMI with the *ExecNotificationQuery* method (for synchronous and semisynchronous notifications) and the *ExecNotificationQueryAsync* method (for the asynchronous notifications) of the **SWbemServices** object. A temporary subscription is stored in memory.

6.2.1.2 Permanent subscription

A subscription builder performing a permanent subscription is interested in performing a specific action each time an event occurs. When the application supposed to perform the action is not running, WMI launches the application. To make the subscription permanent, WMI stores the subscription in the CIM repository. This means that the subscription is effective until the subscription builder removes the subscription explicitly, which means removing instances created in the CIM repository. This is an important difference from the temporary subscription. Figure 6.2 shows the logical organization of a permanent event subscription. With a permanent subscription, the sink routine must be a COM component (called a permanent event consumer).

Because this type of subscription is permanent, two types of instances are created in the CIM repository as follows:

1. **An instance of the event filter**: As with the temporary subscription, the set of events that trigger the action of a permanent subscriber is specified with a WQL event query. This query uses the

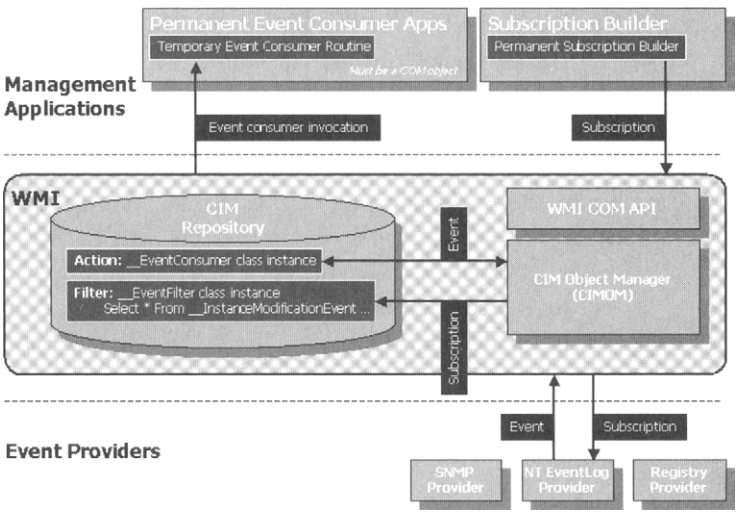

Figure 6.2 *The permanent event consumer and its relation to the CIM repository and WMI.*

same syntax and statements as a temporary subscription, but it is stored in an instance of the *__EventFilter* system class.

2. **An instance of the event consumer class:** Because the subscription is permanent, it uses a permanent event consumer. Each permanent event consumer defines its specific classes in the CIM repository. This is part of a registration process made during its installation. Usually, an event consumer comes with its MOF file containing the registration data and its exposed classes. The event consumer classes are subclasses of the *__EventConsumer* system class. A permanent subscription uses one instance of the permanent event consumer classes.

It is important to note that the subscription builder and the permanent event consumer can be two totally different things. For instance, the subscription builder can be a compiled MOF file or a script creating the required instances, (i.e., an event query filter instance with the event consumer instance) where the permanent consumer can be a .dll provided with WMI (i.e., *SMTP* event consumer).

Note that the CIM repository contains some WMI system classes especially designed to support permanent subscriptions. In addition to the *__EventFilter* and the *__EventConsumer* system classes, WMI implements other system classes related to the event notifications. Let's see what these classes are.

6.2.2 Event system classes

Before going further into the event notification, we must briefly examine a set of classes especially created for WMI to support the event instrumentation with its related components. Figure 6.3 shows a tree view of the system classes used by WMI events.

The most important system classes related to the WMI event notifications are as follows:

- ***__Event:*** This system class is used as a superclass to create all intrinsic event system classes (*__Intrinsic*), extrinsic event system classes (*__Extrinsic*), and timer system class (*__TimerEvent*) proposed by WMI. In Section 6.2.5 we examine the differences between the intrinsic, extrinsic, and timer event notifications.
- ***__AggregateEvent:*** This system class is used as a class template to represent the aggregated events. In Chapter 3, when we talked about the

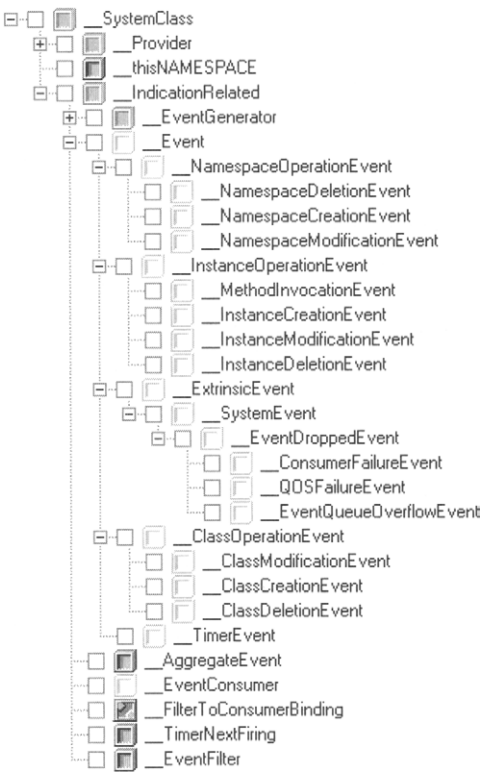

Figure 6.3
The system classes for the event notifications.

WQL event queries, we used the **GROUP** statement to get a single notification to represent a group of events. This type of query typically creates an aggregated event using the *__AggregateEvent* system class.

- *__EventConsumer:* This system class is a superclass for the classes exposed by the permanent event consumers. A permanent event consumer registered in the CIM repository has its own set of classes. This set of classes is derived from the *__EventConsumer* class. Each time a

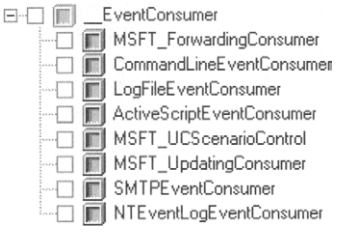

Figure 6.4
The classes supported by some event consumers.

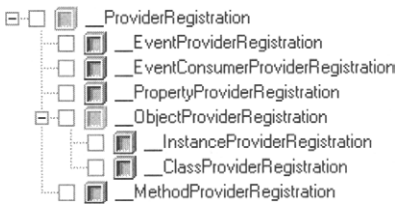

Figure 6.5
The system classes for the provider registrations.

permanent event consumer is subscribed to a WMI event, an instance of the class derived from the *__EventConsumer* system class is created.

- ***__EventFilter:*** This system class is a superclass for the class representing the WQL event query filter made by the subscription builders. This class is associated with the *__EventConsumer* class with the association class *__FilterToConsumerBinding*. So, every instance of an event filter should be associated to an instance of a permanent event consumer, which represents the action to be executed. The *__FilterToConsumerBinding* association class is the link between an event and the corresponding action.
- ***__EventGenerator:*** This system class is a superclass creating specific classes used to trigger timer events.

To summarize, these system classes are used to support event notifications. Each time an event is detected by WMI or reported to WMI, an instance of a class derived from *__Event* system class is created.

The last class to mention is the *__ProviderRegistration* system class. This class is not an event system class; however, it is a superclass template used to register the different type of WMI providers. Figure 6.5 shows the list of subclasses created from the *__ProviderRegistration* system class. For example, the registration of an event provider creates an instance made from the *__EventProviderRegistration* subclass. In the same way, the registration of an event consumer provider creates an instance made from the *__EventConsumerProviderRegistration* subclass. Each registered WMI provider creates an instance from one of these system classes in the CIM repository. Moreover, in addition to this created instance, each provider registration also creates an instance of the *__Win32Provider* system class.

6.2.3 Event consumers

Having both temporary and permanent subscriptions implies that we have two types of event consumers: temporary event consumers and permanent event consumers.

6.2.3.1 Temporary event consumers

As previously mentioned, the temporary consumer is nothing more than a routine that receives an event notification that matches the temporary subscription filter submitted by the subscription builder (see Figure 6.1). In the case of an asynchronous notification, an **SWbemSink** object contains a pointer to the sink routine. It is passed as a parameter when performing the subscription with the *ExecNotificationQueryAsync* method of the **SWbemServices** object. Because the notification query can also be executed synchronously, semisynchronously, or asynchronously, this has a direct impact on the logic and the structure used by the temporary event consumer. In any case, as long as the application hosting the sink routine is running, each matching event is forwarded to the temporary event consumer.

6.2.3.2 Permanent event consumers

A permanent event consumer must be implemented as a COM object. It can be an in-process .dll, a local server, or a remote server. Permanent consumers implemented as local and remote servers require the distributed COM protocol. Under Windows Millennium, Windows NT 4.0, Windows 2000, and Windows.NET Server, DCOM runs by default. For the Windows 95 and 98 platforms, DCOM must be installed as a separated component from the Microsoft Web site (http://www.microsoft.com/com/dcom/dcom95/download.asp for Windows 95 and http://www.microsoft.com/com/dcom/dcom98/download.asp for Windows 98).

By default, WMI comes with a set of permanent event consumers, some of which are mentioned in Chapter 2:

- The *WMI Event Viewer* event consumer
- The *SMTP* event consumer
- The *Log File* event consumer
- The *NT Event Log* event consumer
- The *Command-Line* event consumer
- The *Active Script* event consumer

In Chapter 2, we used the *WMI Event Viewer* event consumer and the *SMTP* event consumer. Because the event registrations of these permanent consumers are stored in the CIM repository with two associated instances (one made from the *__EventConsumer* system class and the other made from *__EventFilter* system class), the even registrations persist through restarting the operating system. When an event matching the WQL filter occurs, WMI determines whether the consumer is active. If the consumer is

not already active because the consumer is registered in the CIM repository with instances made from the *__EventConsumerProviderRegistration* and *__Win32Provider* system classes, or in the system registry as a COM object, WMI locates the consumer and loads it to deliver the event notification.

The permanent event consumers are registered in the CIM repository by creating an instance of the *__EventConsumerProviderRegistration* system class (see Figure 6.5). We can locate the permanent event consumers with the script written in Chapter 4 to locate instances of a given class across namespaces (see Sample 4.30, "A generic routine to display the SWbemPropertySet object"). The script must be started with the class name of the instances to find. In this case, the class must be the *__EventConsumerProviderRegistration* system class. Started on a Windows.NET Server system, the output is as follows:

```
C:\>BrowseNameSpaceForInstancesWithAPI.wsf __EventConsumerProviderRegistration
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. All rights reserved.

Root
Root/SECURITY
Root/MSAPPS
Root/RSOP
Root/RSOP/User
Root/RSOP/User/ms_409
Root/RSOP/User/S_1_5_21_1454471165_1647877149_725345543_1008
Root/RSOP/User/S_1_5_21_1454471165_1647877149_725345543_500
Root/RSOP/Computer
Root/RSOP/Computer/ms_409
Root/Cli
Root/snmp
Root/snmp/localhost
Root/MSCluster
Root/WMI
Root/WMI/ms_409
Root/CIMV2
  ConsumerClassNames (wbemCimtypeString) = Microsoft_SA_AlertEmailConsumer
  *provider (wbemCimtypeReference) =
            \\.\Root\cimv2:__Win32Provider.Name="Microsoft_SA_AlertEmailConsumerProvider"
  ConsumerClassNames (wbemCimtypeString) = EventViewerConsumer
  *provider (wbemCimtypeReference) = \\.\root\cimv2:__Win32Provider.Name="EventViewerConsumer"
  ConsumerClassNames (wbemCimtypeString) = ClearClientConfigAlertEventConsumerClass
  *provider (wbemCimtypeReference) =
            \\.\ROOT\CIMV2:__Win32Provider.Name="ClearClientConfigAlertConsumer"
  ConsumerClassNames (wbemCimtypeString) = ActiveScriptEventConsumer
  *provider (wbemCimtypeReference) = \\.\Root\CIMv2:__Win32Provider.Name="ActiveScriptEventConsumer"
  ConsumerClassNames (wbemCimtypeString) = CmdTriggerConsumer
  *provider (wbemCimtypeReference) = \\.\Root\cimv2:__Win32Provider.Name="CmdTriggerConsumer"
  ConsumerClassNames (wbemCimtypeString) = SMTPEventConsumer
  *provider (wbemCimtypeReference) = \\.\Root\CIMv2:__Win32Provider.Name="SMTPEventConsumer"
  ConsumerClassNames (wbemCimtypeString) = LogFileEventConsumer
  *provider (wbemCimtypeReference) = \\.\Root\CIMv2:__Win32Provider.Name="LogFileEventConsumer"
  ConsumerClassNames (wbemCimtypeString) = CommandLineEventConsumer
  *provider (wbemCimtypeReference) = \\.\Root\CIMv2:__Win32Provider.Name="CommandLineEventConsumer"
  ConsumerClassNames (wbemCimtypeString) = SANTEventLogFilterEventConsumer
```

```
  *provider (wbemCimtypeReference) =
           \\.\root\cimv2:__Win32Provider.Name="SANTEventLogFilterEventConsumerProvider"
  ConsumerClassNames (wbemCimtypeString) = NTEventLogEventConsumer
  *provider (wbemCimtypeReference) = \\.\Root\CIMv2:__Win32Provider.Name="NTEventLogEventConsumer"
Root/CIMV2/ms_409
Root/CIMV2/Applications
Root/CIMV2/Applications/MicrosoftIE
Root/MicrosoftActiveDirectory
Root/MicrosoftIISv2
Root/Policy
Root/Policy/ms_409
Root/MicrosoftDNS
Root/MicrosoftNLB
Root/Microsoft
Root/Microsoft/HomeNet
Root/DEFAULT
Root/DEFAULT/ms_409
Root/directory
Root/directory/LDAP
Root/directory/LDAP/ms_409
Root/subscription
  ConsumerClassNames (wbemCimtypeString) = NTEventLogEventConsumer
  *provider (wbemCimtypeReference) =
           \\.\root\subscription:__Win32Provider.Name="NTEventLogEventConsumer"
  ConsumerClassNames (wbemCimtypeString) = LogFileEventConsumer
  *provider (wbemCimtypeReference) =
           \\.\root\subscription:__Win32Provider.Name="LogFileEventConsumer"
  ConsumerClassNames (wbemCimtypeString) = MSFT_UpdatingConsumer
  ConsumerClassNames (wbemCimtypeString) = MSFT_UCScenarioControl
  *provider (wbemCimtypeReference) =
           \\.\root\subscription:__Win32Provider.Name="Microsoft WMI Updating Consumer Provider"
  ConsumerClassNames (wbemCimtypeString) = SMTPEventConsumer
  *provider (wbemCimtypeReference) = \\.\root\subscription:__Win32Provider.Name="SMTPEventConsumer"
  ConsumerClassNames (wbemCimtypeString) = CommandLineEventConsumer
  *provider (wbemCimtypeReference) =
           \\.\root\subscription:__Win32Provider.Name="CommandLineEventConsumer"
  ConsumerClassNames (wbemCimtypeString) = MSFT_ForwardingConsumer
  *provider (wbemCimtypeReference) =
           \\.\root\subscription:__Win32Provider.Name="Microsoft WMI Forwarding Consumer Provider"
  ConsumerClassNames (wbemCimtypeString) = ActiveScriptEventConsumer
  *provider (wbemCimtypeReference) =
           \\.\root\subscription:__Win32Provider.Name="ActiveScriptEventConsumer"
Root/subscription/ms_409
Root/registry
Root/NetFrameworkv1
Root/NetFrameworkv1/ms_409
```

Across all namespaces available in the CIM repository under Windows.NET Server, we find a collection of permanent event consumers. The most interesting ones are summarized in Table 6.1.

6.2.4 Event providers

Chapter 2 shows that WMI providers are components that act between CIMOM and the manageable entities. Since WMI providers allow access to the manageable entities, the CIM repository contains their registration and

Table 6.1 *The Most Common Permanent Event Consumers*

Namespace	Event Consumer Provider	Consumer Class Name
Root/CIMV2	EventConsumerProvider	TriggerEventConsumer
Root/subscription	NTEventLogEventConsumer	NTEventLogEventConsumer
	LogFileEventConsumer	LogFileEventConsumer
	SMTPEventConsumer	SMTPEventConsumer
	CommandLineEventConsumer	CommandLineEventConsumer
	ActiveScriptEventConsumer	ActiveScriptEventConsumer

the classes they support. WMI classifies the providers in several categories with regard to the type of request they service. For instance, in the previous section we talked about permanent event consumer providers. Regardless of whether they are event consumers, they are providers first! Each provider has its own particularities and determines the management capabilities of a real-world entity. Because the nature of the providers available is determinant, it has a direct impact on WMI scripting possibilities (classes they support, update operations they support, events supported, etc.).

For now, let's concentrate on the event providers. Event providers deliver events to WMI, and event consumer providers consume events delivered by WMI. An event provider is nothing other than a COM component providing an event notification to WMI when the submitted subscription matches the WQL event query. In turn, WMI forwards the notification to the event subscriber. As previously mentioned, the *SNMP* event provider, the NT *Event Log* event provider, and the *Registry* event provider are some typical event providers. Where is WMI offering other event providers? Let's try to answer this question by examining the CIM repository.

Event providers are registered in the CIM repository with an instance of the *__EventProviderRegistration* system class. The *__EventProviderRegistration* is a subclass of the *__ProviderRegistration* system class as shown in Figure 6.5. The registration is made per WMI namespace. Therefore, each instance of the *__EventProviderRegistration* class we find corresponds to an event provider registration made in a particular namespace. The same tactic used to locate the permanent event consumers across all namespaces is used to locate event providers. When the script runs on Windows.NET Server to retrieve instances of the *__EventProviderRegistration* system class, the script output is as follows:

```
C:\ >BrowseNameSpaceForInstancesWithAPI.Wsf __EventProviderRegistration
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. All rights reserved.
```

```
Root
Root/SECURITY
Root/MSAPPS
Root/RSOP
Root/RSOP/User
Root/RSOP/User/ms_409
Root/RSOP/User/S_1_5_21_1454471165_1647877149_725345543_1008
Root/RSOP/User/S_1_5_21_1454471165_1647877149_725345543_500
Root/RSOP/Computer
Root/RSOP/Computer/ms_409
Root/Cli
Root/snmp
Root/snmp/localhost
  EventQueryList (wbemCimtypeString) = select * from SnmpExtendedNotification
  *provider (wbemCimtypeReference) =
            \\.\root\snmp\localhost:__Win32Provider.Name="MS_SNMP_REFERENT_EVENT_PROVIDER"
  EventQueryList (wbemCimtypeString) = select * from SnmpNotification
  *provider (wbemCimtypeReference) =
            \\.\root\snmp\localhost:__Win32Provider.Name="MS_SNMP_ENCAPSULATED_EVENT_PROVIDER"
Root/MSCluster
  EventQueryList (wbemCimtypeString) = select * from MSCluster_Event
  *provider (wbemCimtypeReference) = \\.\Root\MSCluster:__Win32Provider.Name="Cluster Event Provider"
Root/WMI
  EventQueryList (wbemCimtypeString) = select * from WMIEvent
  *provider (wbemCimtypeReference) = \\.\Root\WMI:__Win32Provider.Name="WMIEventProv"
Root/WMI/ms_409
Root/CIMV2
  EventQueryList (wbemCimtypeString) = select * from Msft_WmiProvider_OperationEvent
  *provider (wbemCimtypeReference) = \\.\root\cimv2:__Win32Provider.Name="ProviderSubSystem"
  EventQueryList (wbemCimtypeString) = select * from MSFT_WMI_GenericNonCOMEvent
  EventQueryList (wbemCimtypeString) = select * from MSFT_NcProvEvent
  *provider (wbemCimtypeReference) = \\.\root\cimv2:__Win32Provider.Name="Standard Non-COM Event Provider"
  EventQueryList (wbemCimtypeString) = select * from Win32_PowerManagementEvent
  *provider (wbemCimtypeReference) =
            \\.\Root\CIMV2:__Win32Provider.Name="MS_Power_Management_Event_Provider"
  EventQueryList (wbemCimtypeString) = select * from Win32_SystemConfigurationChangeEvent
  *provider (wbemCimtypeReference) = \\.\Root\CIMV2:__Win32Provider.Name="SystemConfigurationChangeEvents"
  EventQueryList (wbemCimtypeString) = select * from Win32_ComputerShutdownEvent
  *provider (wbemCimtypeReference) = \\.\Root\CIMV2:__Win32Provider.Name="MS_Shutdown_Event_Provider"
  EventQueryList (wbemCimtypeString) = select * from Win32_VolumeChangeEvent
  *provider (wbemCimtypeReference) = \\.\Root\CIMV2:__Win32Provider.Name="VolumeChangeEvents"
  EventQueryList (wbemCimtypeString) = select * from MSFT_ForwardedEvent
  *provider (wbemCimtypeReference) =
            \\.\root\cimv2:__Win32Provider.Name="Microsoft WMI Forwarding Event Provider"
  EventQueryList (wbemCimtypeString) = select * from Win32_IP4RouteTableEvent
  *provider (wbemCimtypeReference) = \\.\Root\CIMV2:__Win32Provider.Name="RouteEventProvider"
  EventQueryList (wbemCimtypeString) = select * from Win32_ProcessStartTrace
  EventQueryList (wbemCimtypeString) = select * from Win32_ProcessStopTrace
  EventQueryList (wbemCimtypeString) = select * from Win32_ThreadStartTrace
  EventQueryList (wbemCimtypeString) = select * from Win32_ThreadStopTrace
  EventQueryList (wbemCimtypeString) = select * from Win32_ModuleLoadTrace
  *provider (wbemCimtypeReference) = \\.\Root\CIMV2:__Win32Provider.Name="WMI Kernel Trace Event Provider"
  EventQueryList (wbemCimtypeString) = select * from MSFT_WmiEssEvent
  *provider (wbemCimtypeReference) =
            \\.\root\cimv2:__Win32Provider.Name="WMI Self-Instrumentation Event Provider"
  EventQueryList (wbemCimtypeString) = select * from __InstanceCreationEvent where TargetInstance isa
"Win32_NTLogEvent"
  *provider (wbemCimtypeReference) = \\.\Root\CIMV2:__Win32Provider.Name="MS_NT_EVENTLOG_EVENT_PROVIDER"
  EventQueryList (wbemCimtypeString) = select * from __InstanceModificationEvent where TargetInstance isa
```

```
"Win32_LocalTime"
  EventQueryList (wbemCimtypeString) =
            select * from __InstanceModificationEvent where TargetInstance isa "Win32_UTCTime"
  *provider (wbemCimtypeReference) = __Win32Provider="Win32ClockProvider"
  EventQueryList (wbemCimtypeString) = select * from Microsoft_SA_AlertEvent
  *provider (wbemCimtypeReference) = \\.\ROOT\CIMV2:__Win32Provider.Name="AppllianceManager"
  EventQueryList (wbemCimtypeString) = select * from MSFT_SCMEvent
  *provider (wbemCimtypeReference) = \\.\root\cimv2:__Win32Provider.Name="SCM Event Provider"
Root/CIMV2/ms_409
Root/CIMV2/Applications
Root/CIMV2/Applications/MicrosoftIE
Root/MicrosoftActiveDirectory
Root/MicrosoftIISv2
Root/Policy
Root/Policy/ms_409
Root/MicrosoftDNS
Root/MicrosoftNLB
  EventQueryList (wbemCimtypeString) = select * from WMIEvent
  *provider (wbemCimtypeReference) = \\.\Root\MicrosoftNLB:__Win32Provider.Name="WMIEventProv"
Root/Microsoft
Root/Microsoft/HomeNet
Root/DEFAULT
  EventQueryList (wbemCimtypeString) = select * from RegistryEvent
  *provider (wbemCimtypeReference) = \\.\Root\Default:__Win32Provider.Name="RegistryEventProvider"
Root/DEFAULT/ms_409
Root/directory
Root/directory/LDAP
Root/directory/LDAP/ms_409
Root/subscription
  EventQueryList (wbemCimtypeString) =
            SELECT * FROM __InstanceOperationEvent WHERE TargetInstance ISA "MSFT_TemplateBase"
  *provider (wbemCimtypeReference) =
            \\.\root\subscription:__Win32Provider.Name="Microsoft WMI Template Event Provider"
  EventQueryList (wbemCimtypeString) = select * from MSFT_TransientEggTimerEvent
  EventQueryList (wbemCimtypeString) =
            select * from __InstanceOperationEvent where TargetInstance isa "MSFT_TransientStateBase"
  *provider (wbemCimtypeReference) =
            \\.\root\subscription:__Win32Provider.Name="Microsoft WMI Transient Event Provider"
  EventQueryList (wbemCimtypeString) = SELECT * FROM MSFT_FCTraceEventBase
  *provider (wbemCimtypeReference) = \\.\root\subscription:__Win32Provider.Name="Microsoft WMI
Forwarding Consumer Trace Event Provider"
  EventQueryList (wbemCimtypeString) = select * from MSFT_TransientRebootEvent
  *provider (wbemCimtypeReference) = \\.\root\subscription:__Win32Provider.Name="Microsoft WMI Transient
Reboot Event Provider"
  EventQueryList (wbemCimtypeString) = SELECT * FROM MSFT_UCTraceEventBase
  EventQueryList (wbemCimtypeString) = SELECT * FROM MSFT_UCEventBase
  *provider (wbemCimtypeReference) = \\.\root\subscription:__Win32Provider.Name="Microsoft WMI Updating
Consumer Event Provider"
Root/subscription/ms_409
Root/registry
Root/NetFrameworkv1
Root/NetFrameworkv1/ms_409
```

The most interesting event providers are summarized in Table 6.2.

Each event provider has an associated WQL event query stored in a property called *EventQueryList*, which is the WQL event query supported by the provider. Before going into detail, let's see what notification types are

Table 6.2 *The Event Providers*

Namespace	Event Provider	Event Query Ilst
Root/CIMV2	MS_Power_Management_Event_Provider	select * from Win32_PowerManagementEvent
	SystemConfigurationChangeEvents	select * from Win32_SystemConfigurationChangeEvent
	MS_Shutdown_Event_Provider	select * from Win32_ComputerShutdownEvent
	VolumeChangeEvents	select * from Win32_VolumeChangeEvent
	Microsoft WMI Forwarding Event Provider	select * from MSFT_ForwardedEvent
	RouteEventProvider	select * from Win32_IP4RouteTableEvent
	WMI Kernel Trace Event Provider	select * from Win32_ProcessStartTrace
		select * from Win32_ProcessStopTrace
		select * from Win32_ThreadStartTrace
		select * from Win32_ThreadStopTrace
		select * from Win32_ModuleLoadTrace
	MS_NT_EVENTLOG_EVENT_PROVIDER	select * from __InstanceCreationEvent where TargetInstance isa "Win32_NTLogEvent"
	Win32ClockProvider	select * from __InstanceModificationEvent where TargetInstance isa "Win32_CurrentTime"
Root/DEFAULT	RegistryEventProvider	select * from RegistryEvent
Root/MicrosoftCluster	Cluster Event Provider	select * from MicrosoftCluster_Event
Root/snmp/localhost	MS_SNMP_REFERENT_EVENT_PROVIDER	select * from SnmpExtendedNotification
	MS_SNMP_ENCAPSULATED_EVENT_PROVIDER	select * from SnmpNotification

Table 6.2 *The Event Providers (continued)*

Namespace	Event Provider	Event Query list
Root/subscription	Microsoft WMI Template Event Provider	select * from __InstanceOperationEvent WHERE TargetInstance ISA "MSFT_TemplateBase"
	Microsoft WMI Transient Event Provider	select * from MSFT_TransientEggTimerEvent
		select * from __InstanceOperationEvent where TargetInstance isa "MSFT_TransientStateBase"
	Microsoft WMI Forwarding Consumer Trace Event Provider	select * from MSFT_FCTraceEventBase
	Microsoft WMI Transient Reboot Event Provider	select * from MSFT_TransientRebootEvent
	Microsoft WMI Updating Consumer Event Provider	select * from MSFT_UCTraceEventBase
		select * from MSFT_UCEventBase
Root/WMI	WMIEventProv	select * from WMIEvent

available from WMI. Each event provider must conform to the different natures of notification available from WMI.

6.2.5 Event notification types

The events are reported to WMI in different ways and are classified as intrinsic events or extrinsic events. In addition to this classification, WMI offers another type of event not related to the CIM repository or the real-world manageable entities. These events are timer events and behave differently.

Intrinsic events are related to creation, modification, or deletion made in the CIM repository in regard to a namespace, a class, or a class instance. Extrinsic events are defined by the event provider designer and correspond to events that are not already described by an intrinsic event. For instance, a change to a registry key or an SNMP trap arrival is a typical extrinsic event. Timer events are events that occur at a particular time and day or repeatedly at regular intervals. Let's start with the intrinsic events.

6.2.5.1 Intrinsic events

WMI creates intrinsic events for data stored statically in the CIM repository and acquires information by inspecting the repository. There are different event types represented by a set of system classes derived from the *__IntrinsicEvent* system class. There is a set of system classes for classes, instances, and namespace creation, modification, and deletion (see Figure 6.6). Each time an intrinsic event is delivered by WMI, it corresponds to one instance of the event classes listed in Table 6.3. Based on the WQL event filter, this event instance will be delivered to the consumer.

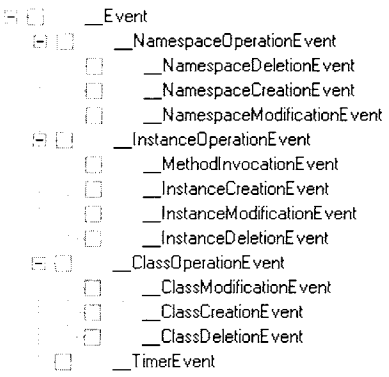

Figure 6.6 *The intrinsic events.*

Table 6.3 *The Intrinsic Events*

__NamespaceCreationEvent	Notifies the consumer when a namespace is created
TargetNamespace	Contains a copy of the __Namespace instance that was created. The Name property of the __Namespace instance indicates which namespace was created.
__NamespaceDeletionEvent	Notifies the consumer when a namespace is deleted
TargetNamespace	Contains a copy of the __Namespace instance that was deleted. The Name property of the __Namespace instance indicates which namespace was deleted.
__NamespaceModificationEvent	Notifies the consumer when a namespace is modified
PreviousNamespace	Contains a copy of the original version of the __Namespace instance. The Name property of this instance indicates which namespace was modified.
TargetNamespace	Contains a copy of the __Namespace instance that was modified. The Name property of the __Namespace instance indicates which namespace was modified.
__ClassCreationEvent	Notifies the consumer when a class is created
TargetClass	Contains a copy of the newly-created class reported by the class creation event.
__ClassDeletionEvent	Notifies the consumer when a class is deleted
TargetClass	Contains a copy of the newly-deleted class reported by the class deletion event.
__ClassModificationEvent	Notifies the consumer when a class is modified
PreviousClass	Contains a copy of the original version of the class.
TargetClass	Contains a copy of the newly-modified class reported by the class modification event.
__InstanceCreationEvent	Notifies the consumer when a instance is created
TargetInstance	Contains a copy of the instance that was created.
__InstanceDeletionEvent	Notifies the consumer when a instance is deleted
TargetInstance	Contains a copy of the instance that was deleted.
__InstanceModificationEvent	Notifies the consumer when a instance is modified
PreviousInstance	Contains a copy of the instance prior to modification.
TargetInstance	Contains the new version of the changed instance.

Each of these instances represents the specific events with one or more properties embedding the object that is subject to the event (*Target* or *PreviousInstance*, *Class* or *Namespace*). We described this object encapsulation in Chapter 3 when discussing the WQL event queries.

A real-world manageable entity is represented in CIM by a dynamic instance provided by a WMI provider. If a dynamic instance is subject to

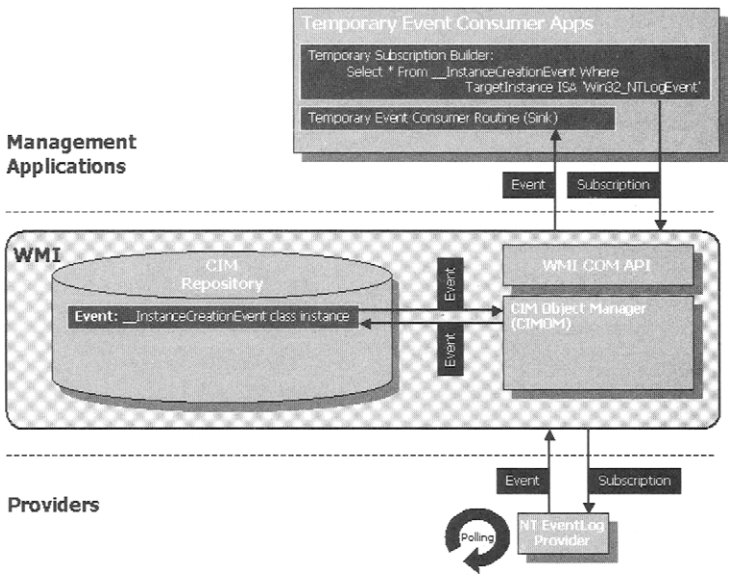

Figure 6.7
The intrinsic events and their relation to the CIM repository and WMI.

change, the WMI provider creates the intrinsic events for dynamic data. To perform this, a WMI event provider must be available, and the polling is executed at the level of the provider itself. This logic is represented in Figure 6.7

Having a WMI event provider for dynamic instances provides the best performance because it allows WMI to delegate the polling tasks to the provider. A typical case of an event provider delivering intrinsic events is the *NT Event Log* event provider. It uses the intrinsic event system class *__InstanceCreationEvent* for the notification creation. The WQL event query to use is

```
Select * From __InstanceCreationEvent Where TargetInstance ISA 'Win32_NTLogEvent'
```

There are two important things to note:

1. The *NT Event Log* is an event provider; there is no need to specify the **WITHIN** WQL statement.
2. The event notification is classified as an intrinsic event because the event provider delivers an event already defined in the CIM repository as a subclass of *__IntrinsicEvent* (*__InstanceCreationEvent*).

Figure 6.8 shows sample output of a temporary event consumer (**WBEMTEST.exe**) using the *NT Event Log* event provider and an instance of the intrinsic event system class *__InstanceCreationEvent*.

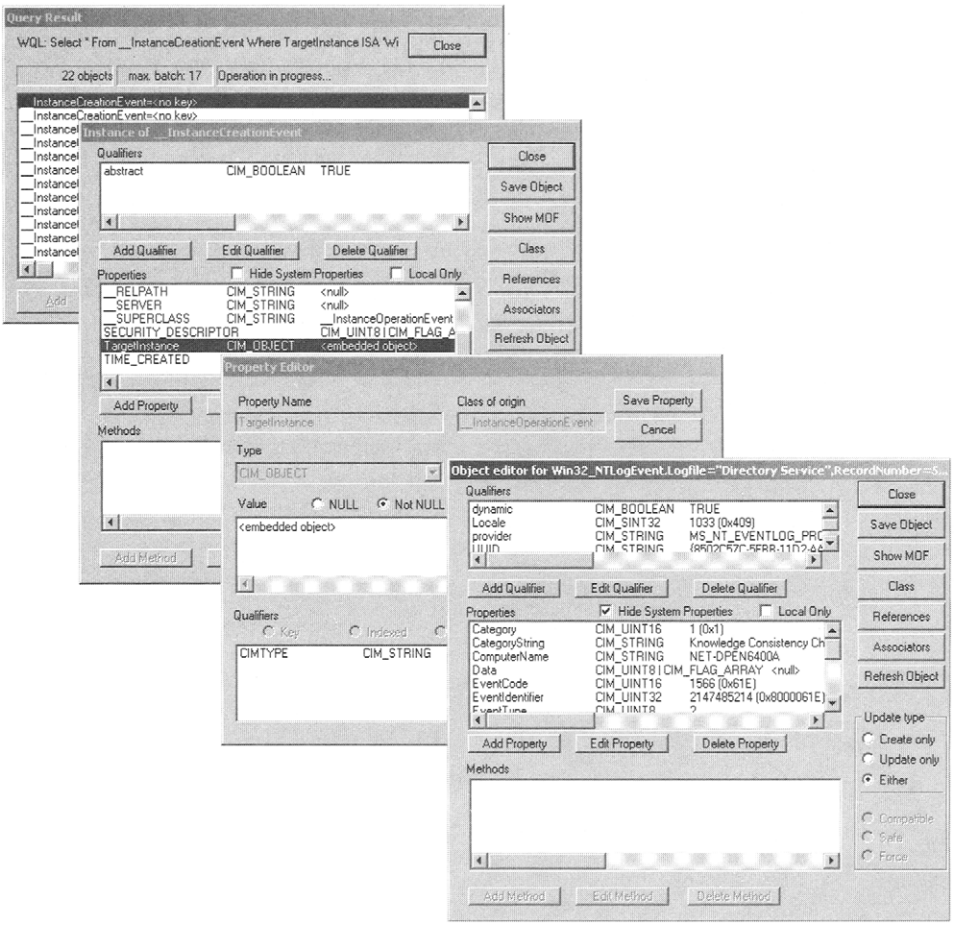

Figure 6.8 *An example of an intrinsic event with the NT Event Log event provider.*

For the facility, the event is created with a small WSH script using the *LogEvent* method of the **WshShell** object. Of course, any other means can be used to create an NT Event Log entry.

Sample 6.1 *A WSH script to create to generate a WMI event by creating an event log entry*

```
 .:
 6:Option Explicit
 .:
10:Set WshShell = Wscript.CreateObject("Wscript.Shell")
11:
12:WshShell.LogEvent 0, "Event Log entry for TEST."
13:
14:Set WshShell = Nothing
```

It is possible that a dynamic instance has no event provider; in such a case, the event consumer must register its WQL event query for polling with WMI by using the **WITHIN** statement. In such a case, WMI monitors the dynamic instance at a regular time interval for changes and delivers the intrinsic event. A typical case is when an application wants to monitor a dynamic class provided by the *Win32* providers. When discovering the WQL event queries discussed in Chapter 3, the WQL event query samples monitored the Windows services status changes. The *Win32* provider supporting this class is not implemented as an event provider, and the polling technique with the **WITHIN** statement was used. As a reminder, the WQL event query used was

```
Select * From __InstanceModificationEvent Within 30 Where TargetInstance ISA 'Win32_Service'
```

This query tracks the creation of an *__InstanceModificationEvent* instance embedding the *Win32_Service* instance modified. There are two important things to note in this query:

1. The *Win32* provider is not an event provider; it is mandatory to specify the **WITHIN** statement.
2. The event notification is classified as an intrinsic event because WMI delivers a predefined event of the CIM repository as a subclass of *__IntrinsicEvent* (*__InstanceModificationEvent*).

This logic is represented in Figure 6.9.

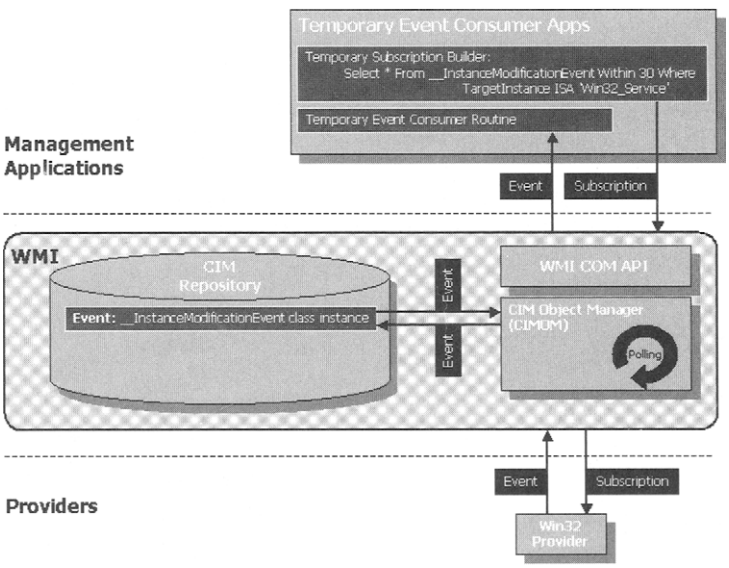

Figure 6.9 *The temporary event consumer and its relation to the CIM repository and WMI.*

If you need more information about intrinsic classes (such as the properties exposed by the classes), you can use the **LoadCIMinXL.wsf** script developed in Chapter 4. This script has been especially developed for this purpose. It self-documents the definition of every class created in the CIM repository. Of course, you can also use the **WMI CIM Studio** to examine the class content.

6.2.5.2 *Extrinsic events*

Extrinsic events are not represented by a fixed set of classes like intrinsic events. The classes representing extrinsic events are determined by the event provider capabilities and may correspond to a wide range of changes that can occur. It is the responsibility of the event provider developer to define the set of classes that best represent the changes monitored. Also, as opposed to intrinsic events, extrinsic events imply the presence of an event provider. By default, when WMI is installed, several event providers delivering extrinsic events are available. In the **Root\CIMv2** namespace, for instance, there is one extrinsic event class, called *Win32_PowerManagementEvent* class, supported by the *Power Management* event provider (see Figure 6.10).

In the **Root\WMI** namespace, Microsoft implements some event classes from the WMI provider mainly related to the NDIS interface (see Figure 6.11).

In the **Root\SNMP\SMIR** namespace, the *SNMP* event provider offers an important set of event classes related to SNMP events (see Figure 6.12).

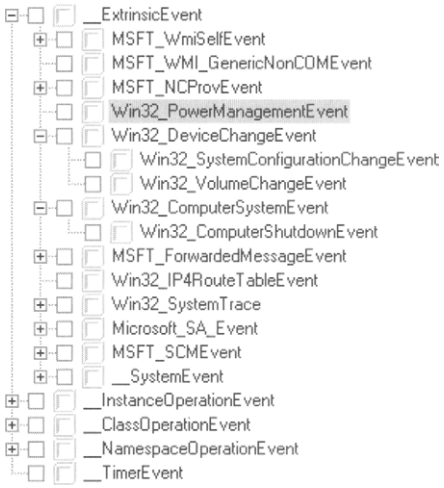

Figure 6.10
The extrinsic events in the Root\CIMv2 namespace.

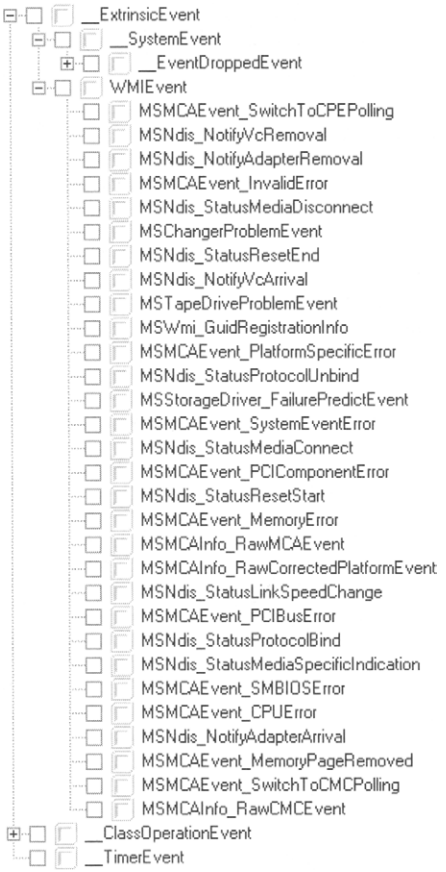

Figure 6.11
The extrinsic events in the Root\WMI namespace.

Last but not least, in the **Root\Default** namespace, the *Registry* event provider exposes three event classes as shown in Figure 6.13. As we see in Figure 6.13, the *Registry* event provider delivers an event called *RegistryTreeChangeEvent*. For instance, the formulation of the WQL event query using this class is as follows:

```
Select * FROM RegistryTreeChangeEvent Where Hive='HKEY_LOCAL_MACHINE' AND Rootpath='Software'
```

There are two important things to note in this WQL query:

1. The *Registry* provider does not require the **WITHIN** statement to detect events because it is an event provider.
2. The event notification is classified as an extrinsic event because it provides a set of event classes different from the standard intrinsic event classes supported by WMI. The event provider delivers specific events to WMI as a subclass of *__ExtrinsicEvent* (i.e.,

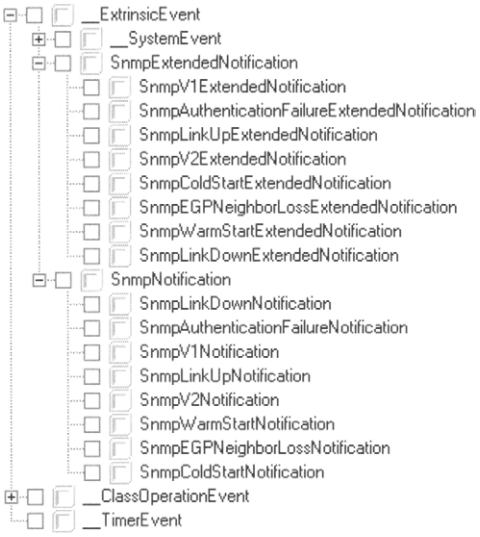

Figure 6.12
The extrinsic events in the Root\SNMP\SMIR namespace.

RegistryTreeChangeEvent), which is a subclass of the *__Event* system class. Actually, it is impossible to have a specific set of defined extrinsic event classes without an event provider.

As for intrinsic events, instances of leaf classes are delivered to consumers.

Figure 6.14 shows the sample output of a temporary event consumer (**WBEMTEST.exe**) using the *Registry* event provider and an instance of the extrinsic event system class *RegistryTreeChangeEvent*. To make this work with **WBEMTEST.exe**, do not forget to connect to the **Root\Default** namespace and select "Notification Query." Any change in the registry hive HKEY_LOCAL_MACHINE under the subkey "software" will create an event.

In Figure 6.14, the *LegalNoticeCaption* key is modified with the **REGEDIT.exe** tool. As soon as the value is modified, the event consumer (**WBEMTEST.exe**) receives the extrinsic event notification.

All extrinsic event providers expose their sets of classes for event notifications. Independent of the nature of the WMI providers, each provider is

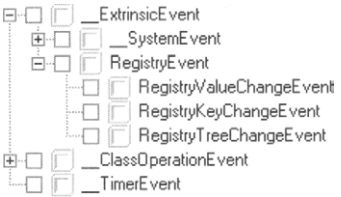

Figure 6.13
The extrinsic events in the Root\Default namespace.

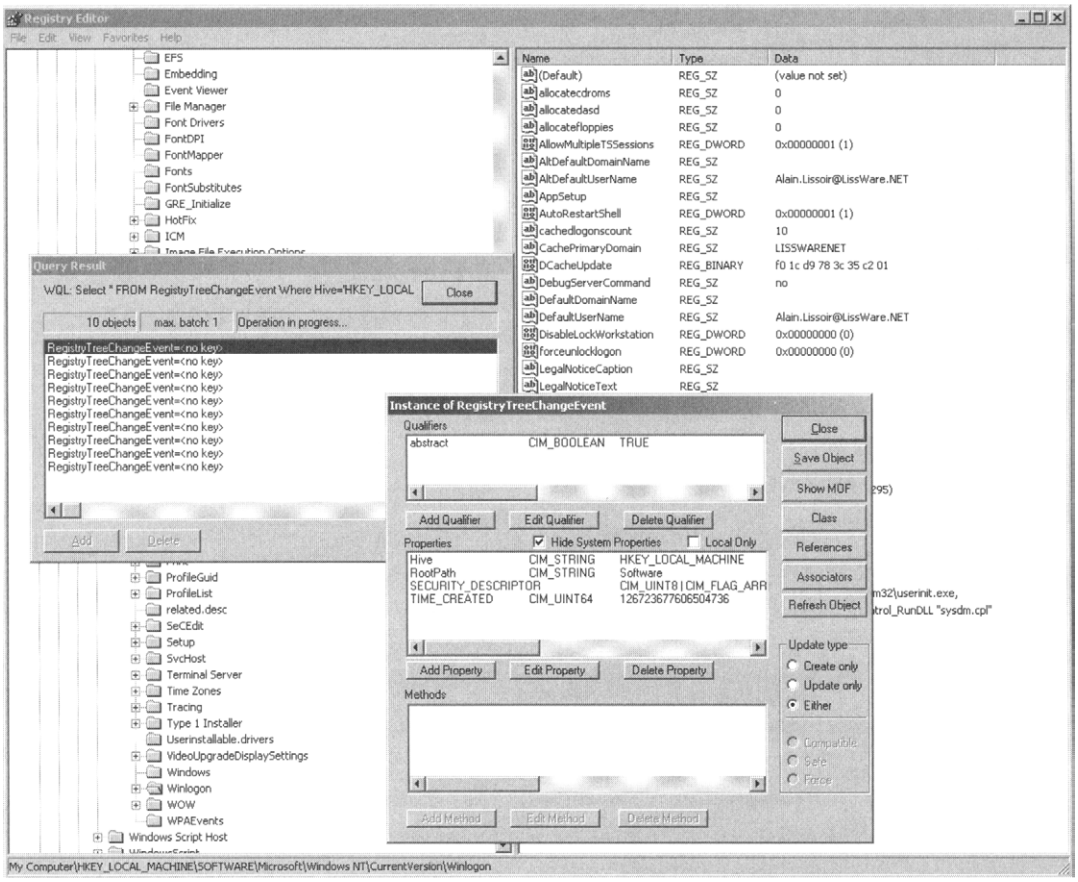

Figure 6.14 *An example of an extrinsic event with the Registry event provider.*

unique and brings its own set of classes. You can examine these classes using the **LoadCIMinXL.wsf** script (developed in Chapter 4) or with **WMI CIM Studio**.

6.2.5.3 *Timer events*

Timer events are totally different from intrinsic or extrinsic events because they are not related to a class, instance, or namespace creation, deletion, or modification. Timer events are events that occur in two situations:

1. At a particular time and day
2. Repeatedly at regular intervals

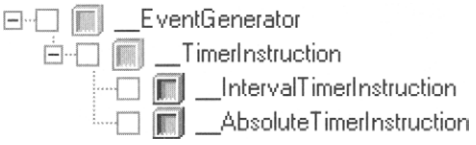

Figure 6.15
The timer events.

A subscription builder must explicitly configure timer events in order for them to occur. These events use two system classes derived from the *__TimerInstruction* system class (see Figure 6.15):

1. ***__AbsoluteTimerInstruction:*** This system class represents events that occur at a particular time and day.
2. ***__IntervalTimerInstruction:*** This system class represents event that occur repeatedly at regular intervals.

As soon as an instance of one of these classes is created, the corresponding timer event occurs, and the registered consumer receives an instance of the *__TimerEvent* system class from WMI. The *__TimerEvent* system class is derived from the *__Event* system class.

6.2.5.3.1 Interval timer event

To create an instance of a timer event, the WMI scripting API or a simple MOF file can be used. First, we use a MOF file for simplicity and the **WEBMTEST.exe** tool as an event consumer of the event. The MOF file used to create an instance of the *__IntervalTimerInstruction* is as follows:

```
instance of __IntervalTimerInstruction
{
  TimerId = "MyIntervalTimerEvent";
  IntervalBetweenEvents = 5000;
};
```

The *__IntervalTimerInstruction* class exposes two properties: the *TimerID* and the *IntervalBetweenEvents*. The *TimerID* is the key of the class and must use a unique name for the instance. The *IntervalBetweenEvents* expresses a delay in milliseconds, which is the time interval to use for the timer event. Next, the MOF file must be compiled with **MOFCOMP.exe**:

```
C:\>MOFCOMP -N:ROOT\CIMv2 IntervalTimerEvent.mof
```

The WQL event query to formulate with **WBEMTEST.exe** is

```
Select * From __TimerEvent Where TimerID = 'MyIntervalTimerEvent'
```

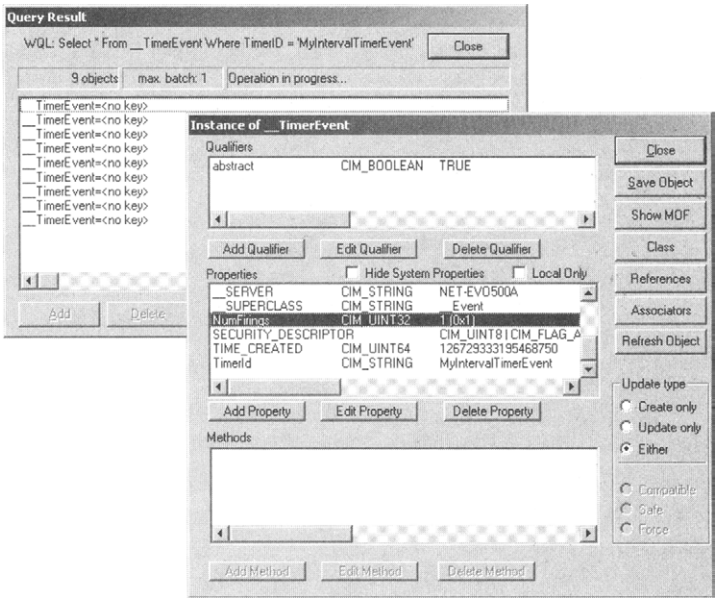

Figure 6.16 *The timer event properties.*

Because all timer events return an instance of the *__TimerEvent* system class, the query performs the selection on this class. To narrow the scope of the query (in case of several timer events), the WQL query uses the *TimerID* specified in the MOF file. Figure 6.16 shows the output of the timer event captured with the temporary consumer **WBEMTEST.exe**.

Instead of using a MOF file to create an instance of the *__IntervalTimerInstruction* class, it is possible to use the WMI scripting API. The script uses the set of WMI objects seen throughout Chapter 4.

Sample 6.2 *Creating an __IntervalTimerInstruction instance with a script*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
14:
15:    <script language="VBscript">
16:    <![CDATA[
..:
20:    Const cComputerName = "LocalHost"
21:    Const cWMINameSpace = "root/cimv2"
22:    Const cWMIClass = "__IntervalTimerInstruction"
23:    Const cTimerID = "MyIntervalTimerEvent"
24:    Const cIntervalBetweenEvents = 5000
..:
30:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
```

```
31:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
32:    Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
33:    Set objWMIClass = objWMIServices.Get (cWMIClass)  34:
35:    Set objWMIInstance = objWMIClass.SpawnInstance_
36:    objWMIInstance.TimerID = cTimerID
37:    objWMIInstance.IntervalBetweenEvents = cIntervalBetweenEvents
38:    objWMIInstance.Put_ (wbemChangeFlagCreateOrUpdate Or wbemFlagReturnWhenComplete)
39:
40:    WScript.Echo cWMIClass & " instance created."
..:
46:    ]]>
47:    </script>
48:  </job>
49:</package>
```

Lines 30 to 32 perform the usual WMI namespace connection. Next, the script creates an instance of the *__IntervalTimerInstruction* system class (lines 33 and 35). Once the instance is available, the script sets the miscellaneous parameters (lines 36 and 37). Before terminating the script, the created instance is committed in the CIM repository (line 38). From now on, every 5,000 ms (5 sec), a *__TimerEvent* notification is triggered by WMI to any registered subscriber for this event. WMI stops generating this event as soon as the *__IntervalTimerInstruction* instance is deleted. The way to proceed is shown in Sample 6.3.

Sample 6.3 *Deleting an __IntervalTimerInstruction instance with a script*

```
1:<?xml version="1.0"?>
.:
8:<package>
9:  <job>
..:
13:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
14:
15:    <script language="VBscript">
16:    <![CDATA[
..:
20:    Const cComputerName = "LocalHost"
21:    Const cWMINameSpace = "root/cimv2"
22:    Const cWMIClass = "__IntervalTimerInstruction"
23:    Const cTimerID = "MyIntervalTimerEvent"
..:
28:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
29:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
30:    Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
31:    Set objWMIInstance = objWMIServices.Get (cWMIClass & "='" & cTimerID & "'")
32:
33:    objWMIInstance.Delete_
34:
35:    WScript.Echo cWMIClass & " instance deleted."
..:
40:    ]]>
41:    </script>
42:  </job>
43:</package>
```

This script is pretty easy to understand. The script gets the *__IntervalTimerInstruction* instance of the corresponding *TimerID* (line 31) and then deletes that instance with the *Delete_* method exposed by the **SWbemObject** representing the *__IntervalTimerInstruction* instance (line 33).

You can perform a simple test. First, subscribe to the event using the **WBEMTEST.exe** tool, and then run the script shown in Sample 6.2 to create the *__IntervalTimerInstruction* instance. As soon as the instance is created, **WBEMTEST.exe** receives the *__TimerEvent* instances at regular intervals. Next, execute the script shown in Sample 6.3 to delete the *__IntervalTimerInstruction* instance; you now see that **WBEMTEST.exe** no longer receives *__TimerEvent* instances from WMI.

6.2.5.3.2 Absolute timer event

The second timer event available from WMI is the absolute timer event. Like the *__IntervalTimerInstruction*, an instance of the *__AbsoluteTimerInstruction* class is created. The *__AbsoluteTimerInstruction* exposes two properties: the *TimerID* and the *EventDateTime*. The *TimerID* is the key property of this class and works as the *TimerID* property of the *__IntervalTimerInstruction* class. The event *EventDateTime* property allows the configuration of a precise date and time when the timer event has to occur. The date and time uses a DMTF format, as discussed in the previous chapter:

```
yyyymmddHHMMSS.mmmmmmsUUU
```

As soon as the date and time match, a *__TimerEvent* notification is triggered by WMI to the subscribed consumers. We can create the *__AbsoluteTimerInstruction* instance using a MOF file, but we can also create the *__AbsoluteTimerInstruction* instance with a script. Sample 6.4 performs this task. The script uses a date equal to 20020804135000.000000+060, which means that WMI will forward an event to the subscribed consumers on August 04, 2002 at 13:50 - GMT+1. Keep in mind that if daylight saving time is active, the hour to specify must be adapted accordingly. For instance, in Central Europe, some countries located in GMT+1 during the winter use a time moved forward by one hour during the summer. To receive an event at 13:50 during the summer, the *EventDateTime* value must be set to August 04, 2002 at 13:50 - GMT+2 (20020804135000.000000+120).

Sample 6.4 *Creating an __AbsoluteTimerInstruction instance with a script*

```
1:<?xml version="1.0"?>
.:
8:<package>
9:  <job>
..:
```

```
13:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
14:
15:    <script language="VBscript">
16:    <![CDATA[
..:
20:    Const cComputerName = "LocalHost"
21:    Const cWMINameSpace = "root/cimv2"
22:    Const cWMIClass = "__AbsoluteTimerInstruction"
23:    Const cTimerId = "MyAbsoluteTimerEvent"
24:    Const cSkipIfPassed = False
25:    Const cEventDateTime = "20020804135000.000000+060"
..:
31:    WScript.Echo "Absolute timer scheduled for " & cEventDateTime
32:
33:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
34:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
35:    Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
36:    Set objWMIClass = objWMIServices.Get (cWMIClass)
37:
38:    Set objWMIInstance = objWMIClass.SpawnInstance_
39:    objWMIInstance.TimerID = cTimerID
40:    objWMIInstance.SkipIfPassed = cSkipIfPassed
41:    objWMIInstance.EventDateTime = cEventDateTime
42:    objWMIInstance.Put_ (wbemChangeFlagCreateOrUpdate Or wbemFlagReturnWhenComplete)
43:
44:    WScript.Echo cWMIClass & " instance created."
..:
50:    ]]>
51:    </script>
52:  </job>
53:</package>
```

In Sample 6.4, the date and time are specified in the DMTF format. With the help of the **SWbemDateTime** helper object (see Chapter 5), it is possible to specify the date and time in another format. Sample 6.5 shows this modification.

Sample 6.5 *Creating an __AbsoluteTimerInstruction instance using the SWBemDateTime object*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
14:    <object progid="WbemScripting.SWbemDateTime" id="objWMIDateTime" reference="true"/>
15:
16:    <script language="VBscript">
17:    <![CDATA[
..:
21:    Const cComputerName = "LocalHost"
22:    Const cWMINameSpace = "root/cimv2"
23:    Const cWMIClass = "__AbsoluteTimerInstruction"
24:    Const cTimerId = "MyAbsoluteTimerEvent"
25:    Const cSkipIfPassed = False
26:    Const cTriggerDateTime = "June 16 2002 15:20:00"
..:
```

```
32:    objWMIDateTime.SetVarDate CDate (cTriggerDateTime), True
33:    WScript.Echo "Absolute timer scheduled for " & objWMIDateTime.GetVarDate (True)
34:    WScript.Echo "                              " & objWMIDateTime.Value
35:
36:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
37:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
38:    Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
39:    Set objWMIClass = objWMIServices.Get (cWMIClass)
40:
41:    Set objWMIInstance = objWMIClass.SpawnInstance_
42:    objWMIInstance.TimerID = cTimerID
43:    objWMIInstance.SkipIfPassed = cSkipIfPassed
44:    objWMIInstance.EventDateTime = objWMIDateTime.Value
45:    objWMIInstance.Put_ (wbemChangeFlagCreateOrUpdate Or wbemFlagReturnWhenComplete)
46:
47:    WScript.Echo cWMIClass & " instance created."
..:
53:    ]]>
54:    </script>
55:  </job>
56:</package>
```

Both scripts use exactly the same logic to create the timer instance. It is important to note that each script creates the *__AbsoluteTimerInstruction* only! They don't act as consumers of the timer event. For the moment, we continue to use the **WBEMTEST.exe** tool as a consumer. Writing a script acting as a consumer of a WMI event is discussed later in Section 6.5. To subscribe to the event, the WQL event query to formulate in **WBEMTEST.exe** is:

```
Select * From __TimerEvent Where TimerID = 'MyAbsoluteTimerEvent'
```

Now, if you want to delete the *__AbsoluteTimerInstruction* instance previously created, you can run a script similar to Sample 6.6.

Sample 6.6 *Deleting an __AbsoluteTimerInstruction instance with a script*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
14:
15:    <script language="VBscript">
16:    <![CDATA[
..:
20:    Const cComputerName = "LocalHost"
21:    Const cWMINameSpace = "root/cimv2"
22:    Const cWMIClass = "__AbsoluteTimerInstruction"
23:    Const cTimerID = "MyAbsoluteTimerEvent"
..:
28:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
29:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
30:    Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
```

```
31:    Set objWMIInstance = objWMIServices.Get (cWMIClass & "='" & cTimerID & "'")
32:
33:    objWMIInstance.Delete_
34:
35:    WScript.Echo cWMIClass & " instance deleted."
..:
40:    ]]>
41:    </script>
42:  </job>
43:</package>
```

6.2.5.3.3 The *Win32_CurrentTime* alternative

The *Win32_CurrentTime* class represents the current time in the system and can be used as an alternate method to trigger events at regular time intervals. However, the *Win32_CurrentTime* class is not part of the WMI core event instrumentation. This class is supported by the *Clock* provider, which is implemented as an event provider. We examine this class and its usage in detail when exploring the *Clock* provider in Chapter 3 of the second book, *Leveraging Windows Management Instrumentation (WMI) Scripting* (ISBN 1555582990).

6.3 Using the permanent event consumers

Beside the permanent *SMTP* event consumer and the permanent *WMI Event Viewer* event consumer used in Chapter 2, we saw that WMI also includes some other interesting event consumers. In this section, we review how to use these consumers and the pitfalls or restrictions of using them. These permanent WMI event consumers are used in the same event context as the previous exercise with the permanent *SMTP* event consumer. In Chapter 2, the consumers were registered to receive an event when a change in a *Win32_Service* instance modification occurs. For simplicity, we will use the same event condition here.

6.3.1 Log File event consumer

The *Log File* event consumer writes a string in a flat file used as a log. The string and flat file name are defined in the event consumer class instance. The consumer allows the definition of a maximum size for the log file. Each time a new log file is created, it generates log files numbered from 001 to 999, when the maximum size of the current file is reached. If the log does not exist, the consumer creates the file. If the file is located in a subdirectory, the subdirectory must exist; otherwise, no log file will be created. This consumer has a few parameters. This event consumer is registered in the CIM repository with the **WbemCons.mof** file located in %SystemRoot%\

System32\Wbem. Under Windows 2000 and Windows.NET Server, it is registered in the **Root\Subscription** namespace. To use the event consumer from the **Root\CIMv2** namespace, it must be registered under that namespace with the following command line:

```
C:\>mofcomp -N:Root\CIMv2 %SystemRoot%\System32\Wbem\WbemCons.Mof
Microsoft (R) 32-bit MOF Compiler Version 5.1.3590.0
Copyright (c) Microsoft Corp. 1997-2001. All rights reserved.
Parsing MOF file: J:\WINDOWS\System32\Wbem\WbemCons.Mof
MOF file has been successfully parsed
Storing data in the repository...
Done!
```

Because the *Command-Line* event consumer and the *NT Event Log* event consumer are also defined in the MOF file, it is not necessary to perform this registration to use the two next consumers. For reference purposes, the section of the MOF file regarding the *Log File* event consumer is as follows:

```
 1:class LogFileEventConsumer : __EventConsumer
 2:{
 3:    [key]
 4:    string Name;
 5:    [Not_Null,
 6:     Description("Fully qualified path name for the log file."
 7:                 "If file does not exist, it will be created."
 8:                 "If directory does not exist, file will not be created.")]
 9:    string Filename;
10:    string Text;
11:    [Description("Maximum size to which file is allowed to grow.  It will "
12:                 "be archived when it exceeds this size.  Archived files "
13:                 "have an extension of .001 through .999.  A value of zero "
14:                 "will be interpreted to mean 'do not archive.' ")]
15:    uint64  MaximumFileSize = 65535;
16:    [Description("If FALSE or NULL, file will not be Unicode.")]
17:    boolean IsUnicode;
18:};
..:
66:instance of __Win32Provider as $P1
67:{
68:    Name = "LogFileEventConsumer";
69:    Clsid = "{266c72d4-62e8-11d1-ad89-00c04fd8fdff}";
70:};
71:
72:instance of __EventConsumerProviderRegistration
73:{
74:    Provider = $P1;
75:    ConsumerClassNames = {"LogFileEventConsumer"};
76:};
..:
..:
```

To use the consumer, it is necessary to create an instance of the consumer with its associated filter by using the following MOF file shown in Sample 6.7.

Sample 6.7 *A MOF file to associate the permanent log file event consumer and any change notification coming from the Win32_Service instances (LogFileConsumerInstanceReg.mof)*

```
 1://-----------------------------------
 2://   Sample for LOGFileEventConsumer
 3://-----------------------------------
 4:
 5:// Create the Event Filter
 6:instance of __EventFilter as $ef
 7:{
 8:  Name = "FilterForWIN32_Services";
 9:  Query = "SELECT * FROM __InstanceModificationEvent WITHIN 10 Where "
10:          "TargetInstance ISA 'Win32_Service'";
11:  QueryLanguage = "WQL";
12:};
13:
14:// create the consumer
15:instance of LOGFileEventConsumer as $LOGFileec
16:{
17:  Name = "LOGFileForSvc";
18:  Filename = "C:\\WMIServiceWatcher.LOG";
19:  Text = "Service %TargetInstance.DisplayName% is %TargetInstance.State%.";
20:  MaximumFileSize = 1024;
21:  IsUnicode = False;
22:};
23:
24:// bind the filter and the consumer
25:instance of __FilterToConsumerBinding
26:{
27:  Filter = $ef;
28:  Consumer = $LOGFileec;
29:};
```

From lines 6 to 12 we recognize the filter instance definition; from lines 15 to 22 we have the *Log File* event consumer instance definition with its parameters. At lines 25 to 29, we have the link between the filter and the associated *Log File* event consumer instance. Once the MOF file is loaded in the CIM repository, nothing else is needed. As soon as a Windows service modification occurs, the event is triggered and the *Log File* event consumer creates a log file called **WMIServiceWatcher.log** in the root of the C disk:. The file contains a string as defined in the *text* property (line 19), and a new log file is created every 1,024 bytes. Table 6.4 summarizes the parameters available from the log file event consumer.

The *Log File* permanent event consumer subscription can also be achieved with a script on top of the WMI scripting API. In this case, the script must create the three required instances for the *__EventFilter* class,

Table 6.4 *The Log File Permanent Event Consumer Class Properties*

Name	Qualifiers: Key String containing the unique name for this consumer.
Filename	Access type: Read/write String naming the file, including the path, to which the log entries are appended.
Text	Access type: Read/write Qualifiers: Template Template string for the text of the log entry.
MaximumFileSize	Access type: Read/write Unsigned 64-bit integer which sets the maximum size of the log file (in bytes). If the primary file exceeds its maximum size, the contents are moved to another file and the primary file is emptied. A value of zero (0) is interpreted as meaning no size limit, and NULL as meaning a 65,535 byte limit. The size of the file is checked before a write operation; therefore, you can have a file that is slightly larger than the specified size limit. The next write operation catches it and starts a new file. The naming structure for the backup file is as follows: If the original filename is 8.3, the extension is replaces by a string of the format "001", "002", and so on with the smallest number larger than all those previously used chosen. If "999" has been used, then the number chosen is the smallest unused number. if the original filename is not 8.3, the suffix described above is appended to the filename.
IsUnicode	Access type: Read/write Boolean indicating whether the log file is a unicode or a Multi Byte Code (MBC) text file.

the *LOGFileEventConsumer* class, and the *__FilterToConsumerBinding* class. This is the purpose of Sample 6.8.

Sample 6.8 *Creating a subscription for a permanent event consumer*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <script language="VBScript" src="..\Functions\TinyErrorHandler.vbs" />
14:
15:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
16:
17:    <script language="VBscript">
18:    <![CDATA[
..:
22:    Const cComputerName = "LocalHost"
23:    Const cWMINameSpace = "root/cimv2"
24:    Const cWMIEventFilterClass = "__EventFilter"
25:    Const cConsumerClass = "LogFileEventConsumer"
26:    Const cWMIFilterToConsumerBindingClass = "__FilterToConsumerBinding"
27:
```

```
28:     ' Consumer Instance data ------------------------------------------------------------
29:     Const cConsumerName = "LOGFileForSvc"
30:     Const cLOGFileName = "C:\WMIServiceWatcher.LOG"
31:     Const cLOGText = "Service %TargetInstance.DisplayName% is %TargetInstance.State%."
32:     Const cLOGMaximumFileSize = 1024
33:     Const cLOGIsUnicode = False
34:
35:     ' __EventFilter Instance data --------------------------------------------------------
36:     Const cWMIEventFilterName = "FilterForWIN32_Services"
37:     Const cWMIEventFilterQuery = "SELECT * FROM __InstanceModificationEvent WITHIN 10 Where
                                    TargetInstance ISA 'Win32_Service'"
..:
47:     objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
48:     objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
49:     Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
..:
52:     ' Create the Consumer instance -------------------------------------------------------
53:     Set objWMIClass = objWMIServices.Get (cConsumerClass)
..:
56:     Set objWMIInstance = objWMIClass.SpawnInstance_
57:     objWMIInstance.Name = cConsumerName
58:     objWMIInstance.FileName = cLOGFileName
59:     objWMIInstance.Text = cLOGText
60:     objWMIInstance.MaximumFileSize = cLOGMaximumFileSize
61:     objWMIInstance.IsUnicode = cLOGIsUnicode
62:     objWMIInstance.Put_ (wbemChangeFlagCreateOrUpdate Or wbemFlagReturnWhenComplete)
..:
65:     Set objWMIInstance = objWMIServices.Get (cConsumerClass & "='" & cConsumerName & "'")
66:     strConsumerRef = objWMIInstance.Path_.Path
67:
68:     WScript.Echo "WMI Consumer Instance instance created."
69:
70:     ' Create the __EventFilter instance ---------------------------------------------------
71:     Set objWMIClass = objWMIServices.Get (cWMIEventFilterClass)
..:
74:     Set objWMIInstance = objWMIClass.SpawnInstance_
75:     objWMIInstance.Name = cWMIEventFilterName
76:     objWMIInstance.Query = cWMIEventFilterQuery
77:     objWMIInstance.QueryLanguage = "WQL"
78:     objWMIInstance.Put_ (wbemChangeFlagCreateOrUpdate Or wbemFlagReturnWhenComplete)
..:
81:     Set objWMIInstance = objWMIServices.Get (cWMIEventFilterClass & "='" & cWMIEventFilterName & "'")
82:     strEventFilterRef = objWMIInstance.Path_.Path
83:
84:     WScript.Echo "WMI __EventFilter instance created."
85:
86:     ' Create the __FilterToConsumerBinding instance --------------------------------------
87:     Set objWMIClass = objWMIServices.Get (cWMIFilterToConsumerBindingClass)
..:
90:     Set objWMIInstance = objWMIClass.SpawnInstance_
91:     objWMIInstance.Filter = strEventFilterRef
92:     objWMIInstance.Consumer = strConsumerRef
93:     objWMIInstance.Put_ (wbemChangeFlagCreateOrUpdate Or wbemFlagReturnWhenComplete)
..:
96:     WScript.Echo "WMI __FilterToConsumerBinding instance created."
..:
102:    ]]>
103:    </script>
104:  </job>
105:</package>
```

The script creates the necessary instances to allow the notification to a particular consumer. Before, the consumer was a temporary consumer, and we used the **WBEMTEST.exe** tool. Now, we use a permanent event consumer, and the three required instances must be created accordingly: the consumer instance (lines 52 to 68), the *__EventFilter* instance (lines 70 to 84), and the *__FilterToConsumerBinding* instance (lines 86 to 96). The interesting particularity resides in the *__FilterToConsumerBinding* instance creation. Because this instance is an association instance that contains references, its references must be initialized with the complete object path of the consumer instance (lines 66 and 92) and the *__EventFilter* instance (lines 89 and 91). Except for this specific required information to create each instance, there is nothing new. This script can be used to create the required instance for any type of event consumer. Only the parameters must be reviewed (lines 29 to 33 and lines 57 to 61).

6.3.2 NT Event Log event consumer

In the same philosophy as before, instead of creating text lines in a flat file, WMI provides a permanent event consumer to log information in the NT application event log. By default, this consumer is also registered in the **Root\Subscription** name space. The section of the **WbemCons.mof** defining the class for the *NT Event Log* permanent event consumer is shown below:

```
..:
..:
53:[description("Logs events into NT event log")]
54:class NTEventLogEventConsumer : __EventConsumer
55:{
56:     [key] string Name;
57:     string UNCServerName;
58:     string SourceName;
59:     [not_null] uint32 EventID;
60:     uint32 EventType = 1;
61:     uint32 Category;
62:     uint32 NumberOfInsertionStrings = 0;
63:     string InsertionStringTemplates[] = {""};
64:};
..:
90:instance of __Win32Provider as $P3
91:{
92:     Name = "NTEventLogEventConsumer";
93:     Clsid = "{266c72e6-62e8-11d1-ad89-00c04fd8fdff}";
94:};
95:
96:instance of __EventConsumerProviderRegistration
97:{
```

```
98:     Provider = $P3;
99:     ConsumerClassNames = {"NTEventLogEventConsumer"};
100:};
```

As before, an instance of the consumer class associated with an event filter must be created with the help of a MOF file as shown in Sample 6.9.

Sample 6.9 *A MOF file to associate the permanent NT event log event consumer and any change notification coming from the Win32_Service instances (NTEventLogConsumerInstanceReg.mof)*

```
 1://----------------------------------------
 2://   Sample for NTEventLogEventConsumer
 3://----------------------------------------
 4:
 5:// Create the Event Filter
 6:instance of __EventFilter as $ef
 7:{
 8:  Name = "FilterForWIN32_Services";
 9:  Query = "SELECT * FROM __InstanceModificationEvent WITHIN 10 Where "
10:         "TargetInstance ISA 'Win32_Service'";
11:  QueryLanguage = "WQL";
12:};
13:
14:// create the consumer
15:instance of NTEventLogEventConsumer as $NTEventLogec
16:{
17:  Name = "NTEventLogForSvc";
18:  UNCServerName = "";
19:  SourceName = "WMI NTEventLog event consumer";
20:  EventID = 100;
21:  EventType = 2; // 0=Information, 1=Error, 2=Warning
22:  Category = 0;
23:  NumberOfInsertionStrings = 2;
24:  InsertionStringTemplates = {"This is a WMI NTEventLog permanent event consumer action.",
25:                              "Service %TargetInstance.DisplayName% is %TargetInstance.State%."};
26:};
27:
28:// bind the filter and the consumer
29:instance of __FilterToConsumerBinding
30:{
31:  Filter = $ef;
32:  Consumer = $NTEventLogec;
33:};
```

Like any of the other MOF files registering the permanent event consumer class, the three traditional sections are defined. The only change is in the section defining the consumer instance (lines 15 to 26). Once registered, any change to a Windows service creates a warning (line 21) event record in the application event log with a source name "WMI NTEventLog event consumer" (line 19) and an event ID equal to 100 (line 20). Two insertion strings are also associated with the event log trace as defined in lines 23 to 24. Note that in line 18 no server name is given. This implies

that the message is logged on the local machine. If the message must be logged on a remote machine, rights must be configured accordingly, and the server name must be provided in UNC notation. Table 6.5 summarizes the parameters available from the *NT Event Log* event consumer.

Table 6.5 *The NT Event Log Permanent Event Consumer Class Properties*

Name	Access type: Read/write Qualifiers: Key String containing the unique name of this consumer.
UNCServerName	Access type: Read/write String naming the computer on which to log the event, or NULL if the event is to be logged on the local server.
SourceName	Access type: Read/write String naming the source in which the message is to be found. The customer is assumed to have registered a DLL with the necessary messages.
EventID	Access type: Read/write Qualifiers: Not null Unsigned 32-bit integer identifying the event message in the message DLL. This property cannot be NULL.
EventType	Access type: Read/write Unsigned 32-bit integer specifying the type of event being raised. This parameter can have one of the following values which are defined in Winnt.h and Ntelfapi.h. EVENTLOG_SUCCESS Successful event EVENTLOG_ERROR_TPYE Error event EVENTLOG_WARNING_TYPE Warning event EVENTLOG_INFORMATION_TYPE Information event EVENTLOG_AUDIT_SUCCESS Success audit type EVENTLOG_AUDIT_FAILURE Failure audit type
Category	Access type: Read/write Unsigned 32-bit integer specifying the event category. This is source-specific information and can have any value.
NumberOfInsertionStrings	Access type: Read/write Unsigned 32-bit integer specifying the number of elements in the InsertionStringTemplates array.
InsertionStringTemplates	Access type: Read/write Qualifiers: Template Array of string templates used as the insertion strings for the event log record.

6.3.3 Command-Line event consumer

The *Command-Line* event consumer allows the startup of any Win32 application. It can be a command line (a new command prompt session) or any other Win32 application. The consumer also exposes some parameters to define the window style or the lifetime of the started process. This consumer is also defined in the **WbemCons.mof** file and registered, by default, in the **Root\Subscription** namespace. Here is the section related to its class registration in the CIM repository:

```
..:
..:
20:class CommandLineEventConsumer : __EventConsumer
21:{
22:     [key]
23:     string Name;
24:     string ExecutablePath;
25:     [Template]
26:     string CommandLineTemplate;
27:     boolean UseDefaultErrorMode = FALSE;
28:     boolean CreateNewConsole = FALSE;
29:     boolean CreateNewProcessGroup = FALSE;
30:     boolean CreateSeparateWowVdm = FALSE;
31:     boolean CreateSharedWowVdm = FALSE;
32:     sint32 Priority = 32;
33:     string WorkingDirectory;
34:     string DesktopName;
35:     [Template]
36:     string WindowTitle;
37:     uint32 XCoordinate;
38:     uint32 YCoordinate;
39:     uint32 XSize;
40:     uint32 YSize;
41:     uint32 XNumCharacters;
42:     uint32 YNumCharacters;
43:     uint32 FillAttribute;
44:     uint32 ShowWindowCommand;
45:     boolean ForceOnFeedback = FALSE;
46:     boolean ForceOffFeedback = FALSE;
47:     boolean RunInteractively = FALSE;
48:     [description("Number of seconds that child process is allowed to run"
49:                  "if zero, process will not be terminated")]
50:     uint32 KillTimeout = 0;
..:
78:instance of __Win32Provider as $P2
79:{
80:     Name = "CommandLineEventConsumer";
81:     Clsid = "{266c72e5-62e8-11d1-ad89-00c04fd8fdff}";
82:};
83:
84:instance of __EventConsumerProviderRegistration
85:{
86:     Provider = $P2;
87:     ConsumerClassNames = {"CommandLineEventConsumer"};
88:};
..:
..:
```

To create an instance of this permanent event consumer, the following MOF file can be used as shown in Sample 6.10.

Sample 6.10 *A MOF file to associate the permanent command-line event consumer and any change notification coming from the Win32_Service instances (CommandLineConsumerInstanceReg.mof)*

```
1://----------------------------------------
2://   Sample for CommandLineEventConsumer
3://----------------------------------------
4:
5:// Create the Event Filter
6:instance of __EventFilter as $ef
7:{
8:  Name = "FilterForWIN32_Services";
9:  Query = "SELECT * FROM __InstanceModificationEvent WITHIN 10 Where "
10:         "TargetInstance ISA 'Win32_Service'";
11:  QueryLanguage = "WQL";
12:};
13:
14:// create the consumer
15:instance of CommandLineEventConsumer as $CommandLineec
16:{
17:  Name = "CommandLineForSvc";
18:  ExecutablePath = "Cmd.exe";
19:  WindowTitle = "Started for 10 seconds (Service "
20:                "%TargetInstance.DisplayName% is %TargetInstance.State%) ...";
21:  KillTimeout = 10;
22:  WorkingDirectory = "C:\\";
23:  FillAttribute = 144;
24:  RunInteractively = true;
25:};
26:
27:// bind the filter and the consumer
28:instance of __FilterToConsumerBinding
29:{
30:  Filter = $ef;
31:  Consumer = $CommandLineec;
32:};
```

The MOF continues to use the same structure as before. The only change resides in the permanent consumer instance definition to define the parameters related to this particular consumer. Once registered, when a Windows service change occurs, the command-line consumer starts a new command prompt (line 18) with a default directory of C:\ (line 22) and with a window title defined in lines 19 and 20. The lifetime of the command prompt is 10 seconds as defined in line 21. The foreground and the background color is a value combination as defined in Table 6.6. The process is set to run interactively (line 24). Nothing else is needed except to

Table 6.6 *The Command-Line Permanent Event Consumer Class Properties*

Name	Access type: Read/write Qualifiers: Key String property that specifies the unique name of this consumer.
ExecutablePath	Access type: Read-only Qualifiers: Key String specifying the module to execute. The string can specify the full path and file name of the module to execute or it can specify a partial name. In the case of a partial name, the current drive and current directory will be assumed. The ExecutablePath parameter can be NULL. In that case, the module name must be the first white space-delimited token in the CommandLine-Template string. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin; otherwise, the file name is ambiguous.
CommandLineTemplate	Access type: Read/write Qualifiers: Template Template string specifying the process to be launched. This parameter can be NULL. In that case, the method uses ExecutablePath as the command line.
UseDefaultErrorMode	Access type: Read/write Boolean indicating the error mode.
CreateNewConsole	Access type: Read/write Boolean indicating that the new process has a new console, instead of inheriting the parent's console.
CreateNewProcessGroup	Access type: Read/write Boolean indicating that the new process is the root process of a new process group. The process group includes all processes that are descendants of this root process. The process identifier of the new process group is the same as this process identifier. Process groups are used by the GenerateConsoleCtrlEvent method to enable sending a Ctrl+C or Ctrl+Break signal to a group of console processes.
CreateSeparateWowVdm	Access type: Read/write Windows_NT/Windows_2000 only. Boolean indicating whether the new process runs in a private Virtual DOS Machine (VDM). This is only valid when starting a 16-bit Windows-based application. If set to false, all 16-bit Windows-based applications run as threads in a single, shared VDM. See Remarks.
CreateSharedWowVdm	Access type: Read/write Windows_NT/Windows_2000 only. Boolean indicating whether the CreateProcess method is to run the new process in the shared Virtual DOS Machine (VDM). This property is able to override the DefaultSeparateVDM switch in the Windows section of Win.ini if it is set to true.
Priority	Access type: Read/write Signed 32-bit integer that determines the scheduling priorities of the process's threads. The following table lists the priority levels available.

Table 6.6 *The Command-Line Permanent Event Consumer Class Properties (continued)*

Priority	0x00000020 Indicates a normal process with no special scheduling needs. 0x00000040 Indicates a process whose threads run only when the system is idle and which are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle priority class is inherited by child processes. 0x00000080 Indicates a process that performs high priority, time-critical tasks. The threads of a high-priiority class process preempt the threads of normal-priority or idle-priority class processes. An example is the Task List, which must respond quickly when called by the user, regardless of the load on the system. Use extreme care when using the high-priority class, as a CPU-bound application with a high-priority class can use nearly all available cycles. 0x00000100 Indicates a process that has the highest possible priiority. The threads of a real-time priority class process preempt the threads of all other processes, including operating system processes performing important tasks. For
WorkingDirectory	Access type: Read/write String specifying the working directory for this process.
DesktopName	Access type: Read/write String specifying the name of the desktop.
WindowTitle	Access type: Read/write Qualifiers: Template String specifying the title that appears on the title bar of the process.
XCoordinate	Access type: Read/write Unsigned 32-bit integer which specifies the x-offset (pixels) from the left edge of the screen to the left edge of the window, if a new window is created.
YCoordinate	Access type: Read/write Unsigned 32-bit integer which indicates the y-offset (pixels) from the top edge of the screen to the top edge of the window, if a new window is created.
XSize	Access type: Read/write Unsigned 32-bit integer which indicates the width (pixels) of the new window, if a new window is created.
YSize	Access type: Read/write Unsigned 32-bit integer which indicates the height (pixels) of the new window, if a new window is created.

Table 6.6 *The Command-Line Permanent Event Consumer Class Properties (continued)*

XNumCharacters	Access type: Read/write Unsigned 32-bit integer which specifies the screen buffer width in character columns, if a new console window is created. This property is ignored in a GUI process.
YNumCharacters	Access type: Read/write Unsigned 32-bit integer which specifies the screen buffer height in characters rows, if a new console window is created. This property is ignored in a GUI process.
FillAttributes	Access type: Read/write Unsigned 32-bit integer which specifies the initial text and background colors, if a new console window is created in a console application. This property is ignored in a GUI application. The value can be any combination of the following values: 0x00000001 blue foreground 0x00000002 green foreground 0x00000004 red foreground 0x00000008 foreground intensity 0x00000010 background blue 0x00000020 background green 0X00000040 background red 0X00000080 background intensity
ShowWindowCommand	Access type: Read/write Unsigned 32-bit integer which specifies the default value the first time ShowWindow is called, for GUI processes.
ForceOnFeedback	Access type: Read/write Boolean indicating that the cursor is in feedback mode for two seconds after CreateProcess is called. If during those two seconds, the process makes the first GUI call, the system gives five additional seconds to the process. If during those five seconds, the process shows a window, the system give another five seconds to the process to finish drawing the window.
ForceOffFeedback	Access type: Read/write Boolean indicating that the feedback cursor is forced off while the process is starting. The normal cursor is displayed.
RunInteractively	Access type: Read/write Boolean indicating whether the process is to be launched in the interactive winstation (true) or in the default service winstation (false). This property overrides the DesktopName property.
KillTimeout	Access type: Read/write Unsigned 32-bit integer specifying the number of seconds the WinMgmt service is to wait before killing the process. Zero (0) indicates the process is to not be killed. Killing the process is intended as a fail-safe to prevent a process from running indefinitely.

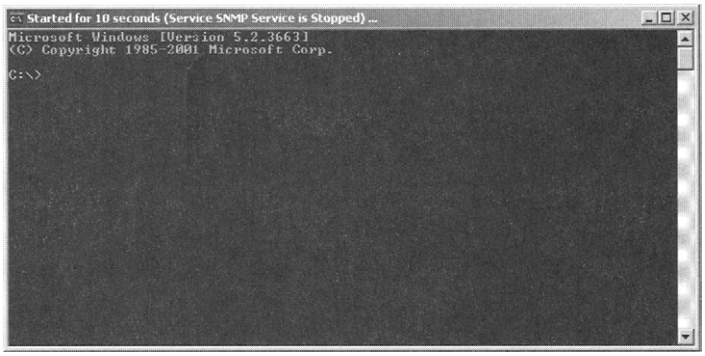

Figure 6.17 *The command-line permanent event consumer.*

perform the registration of the consumer instance in the CIM repository with MOFCOMP.exe. Table 6.6 summarizes the parameters available from the *Command-Line* event consumer.

As a result, Figure 6.17 shows the created command line window.

Of course, for the exercise we used the **CMD.EXE** executable, but any other Win32 program can be used.

6.3.4 Active Script event consumer

Similar to the *Command-Line* event consumer, the permanent *Active Script* event consumer allows starting a VBScript or a JScript. This event consumer is registered in the **Root\Subscription** namespace with the **SCRCONS.mof** file located in %SystemRoot%\System32\Wbem.

```
C:\>mofcomp -N:Root\CIMv2 %SystemRoot%\System32\Wbem\SCRCONS.MOF
Microsoft (R) 32-bit MOF Compiler Version 5.1.3590.0
Copyright (c) Microsoft Corp. 1997-2001. All rights reserved.
Parsing MOF file: J:\WINDOWS\System32\Wbem\SCRCONS.MOF
MOF file has been successfully parsed
Storing data in the repository...
Done!
```

As usual, to use the consumer, an instance of its class must be created. Sample 6.11 shows the MOF file used to create the consumer instance.

Sample 6.11 *A MOF file to associate the permanent active script event consumer with a JScript and any change notification coming from the Win32_Service instances (ActiveScriptConsumerInstanceReg.mof)*

```
1://---------------------------------------
2://  Sample for ActiveScriptEventConsumer
3://---------------------------------------
4:
```

```
 5:// Create the Event Filter
 6:instance of __EventFilter as $ef
 7:{
 8:  Name = "FilterForWIN32_Services";
 9:  Query = "SELECT * FROM __InstanceModificationEvent WITHIN 10 Where "
10:          "TargetInstance ISA 'Win32_Service'";
11:  QueryLanguage = "WQL";
12:};
13:
14:// create the consumer
15:instance of ActiveScriptEventConsumer as $ActiveScriptec
16:{
17:  Name = "ActiveScriptForSvc";
18:  ScriptingEngine = "JScript";
19:  ScriptText = "var ForReading = 1, ForWriting = 2, ForAppending = 8;"
20:               "var TristateFalse = 0, TristateTrue = -1, TristateUseDefault = -2;"
21:               "var strFileName = \"c:\\\\ActiveScriptEventConsumer.txt\";"
22:               "var objFileSystem = new ActiveXObject(\"Scripting.FileSystemObject\");"
23:               "var objfile = objFileSystem.OpenTextFile"
24:               "                  (strFileName, ForAppending, true, TristateFalse);"
25:               "objfile.WriteLine(new Date);"
26:               "objfile.WriteLine(\"Service %TargetInstance.DisplayName%"
27:               " is %TargetInstance.State%\");"
28:               "objfile.Close();";
29:  KillTimeout = 30;
30:};
31:
32:// bind the filter and the consumer
33:instance of __FilterToConsumerBinding
34:{
35:  Filter = $ef;
36:  Consumer = $ActiveScriptec;
37:};
```

Besides the traditional sections to define every instance type, the use of this consumer requires some particular attention. For security reasons, only an administrator may configure the active script consumer. Moreover, the started script runs in the LocalSystemSecurity context as a separate process from WMI. Next, the script is executed with the Windows Script Interfaces. This implies that the script environment is not Windows Script Host (WSH) and that no WSH feature is available. The script language is determined with a specific parameter (line 18). The script itself can be specified in two different ways. The first method, as shown in Sample 6.11, uses the *ScriptText* parameter. This parameter accepts the direct encoding of the script in the MOF file. Although this can be useful, some care must be taken to respect the MOF file syntax. Sample 6.11 uses a JScript already used in Chapter 1 for the remote script execution. The script simply creates a file with a UTC timestamp in it. The active script consumer MOF file starting the same script written in VBScript as shown in Sample 6.12.

Sample 6.12 *A MOF file to associate the permanent active script event consumer with a VBScript and any change notification coming from the Win32_Service instances (ActiveScriptConsumerInstanceReg2.mof)*

```
1://----------------------------------------
2://  Sample for ActiveScriptEventConsumer
3://----------------------------------------
4:
5:// Create the Event Filter
6:instance of __EventFilter as $ef
7:{
8:  Name = "FilterForWIN32_Services";
9:  Query = "SELECT * FROM __InstanceModificationEvent WITHIN 10 Where "
10:         "TargetInstance ISA 'Win32_Service'";
11:  QueryLanguage = "WQL";
12:};
13:
14:// create the consumer
15:instance of ActiveScriptEventConsumer as $ActiveScriptec
16:{
17:  Name = "ActiveScriptForSvc2";
18:  ScriptingEngine = "VBScript";
19:  ScriptText = "Option Explicit\n"
20:               "Const ForReading = 1\n"
21:               "Const ForWriting = 2\n"
22:               "Const ForAppending = 8\n"
23:               "Const TristateFalse = 0\n"
24:               "Const TristateTrue = -1\n"
25:               "Const TristateUseDefault = -2\n"
26:               "Dim strFileName\n"
27:               "Dim objFileSystem\n"
28:               "Dim objfile\n"
29:               "strFileName = \"c:\\ActiveScriptEventConsumer2.txt\"\n"
30:               "Set objFileSystem = CreateObject(\"Scripting.FileSystemObject\")\n"
31:               "Set objfile = objFileSystem.OpenTextFile"
32:               "                (strFileName, ForAppending, true, TristateFalse)\n"
33:               "objfile.WriteLine(Date & \" - \" & Time)\n"
34:               "objfile.WriteLine(\"Service %TargetInstance.DisplayName%"
35:               " is %TargetInstance.State%\")\n"
36:               "objfile.Close()\n";
37:  KillTimeout = 30;
38:};
39:
40:// bind the filter and the consumer
41:instance of __FilterToConsumerBinding
42:{
43:  Filter = $ef;
44:  Consumer = $ActiveScriptec;
45:};
```

Once included in a MOF file, some characters (such as the backslash and the quote) must be escaped to respect the MOF file syntax requirements (see Samples 6.11 and 6.12). If these modifications are not made, the MOF complier will complain during the MOF file parsing. When the JScript script is not included in a MOF file, it looks like this:

```
 1:var ForReading = 1, ForWriting = 2, ForAppending = 8;
 2:var TristateFalse = 0, TristateTrue = -1, TristateUseDefault = -2;
 3:
 4:var strFileName = "c:\\ActiveScriptEventConsumer.txt";
 5:var objFileSystem = new ActiveXObject("Scripting.FileSystemObject");
 6:var objfile = objFileSystem.OpenTextFile(strFileName, ForAppending, true, TristateFalse);
 7:
 8:objfile.WriteLine(new Date);
 9:
10:objfile.Close();
```

The second method consists of specifying the script file name itself. In this case, the script is no longer included in the MOF file but is stored on the file system as a standard JScript or VBScript. The ability to kill the script after a certain period (line 27 of Sample 6.11 and line 37 of Sample 6.12) is supported only if the script is started with the *ScriptText* parameter, not with the *ScriptFileName* parameter.

As WMI runs as a service, any script started by the permanent *Active Script* event consumer does not generate output. Running the WMI service **WinMgmt.exe** as an executable is not supported, but it allows any script to display popup messages by using the Msgbox function. But again, as mentioned before, no WSH function is available as the script is not started in the WSH context. Although the permanent active script consumer has some restrictions, such a consumer can be useful in some circumstances, especially when an administrator already has a script performing some tasks and wants to integrate the code with WMI quickly without having to rewrite the complete script logic on top of the WMI scripting API. In this case, it is a quick-and-dirty solution, but like any quick-and-dirty solution, it comes with restrictions (such as accessing remote systems from the started script and displaying output) and advantages (such as a quick WMI integration). Table 6.7 summarizes the parameters available from the *Active Script* event consumer.

6.3.5 The SMTP event consumer

The permanent *SMTP* event consumer was used in Chapter 2 when introducing the capabilities of WMI. Please refer to Chapter 2 for more information about this permanent event consumer provider.

6.3.6 The Forwarding consumer provider

The *Microsoft WMI Forwarding* consumer provider is available under Windows XP. However, before examining this consumer, it is necessary to gather more knowledge about WMI. Therefore, this consumer is discussed in

Table 6.7 *The Active Script Permanent Event Consumer Class Properties*

Name	Access type: Read-only Qualifiers: Key String that uniquely identifies the event consumer.
ScriptingEngine	Access type: Read-only Qualifiers: Not_Null String naming the scripting engine to use. For example: "VBScript" or "JScript". This property cannot be NULL.
ScriptText	Access type: Read-only Qualifiers: Template Template string containing the text of the script to execute. If NULL, the ScriptFileName property is used.
ScriptFileName	Access type: Read-only String naming the file from which the script text is read. Must be NULL if ScriptText is non-NULL.
KillTimeout	Access type: Read-only Unsigned 32-bit integer specifying the number of seconds after which the script will be terminated if it isn't already finished. If zero, the script will not be terminated. This property applies only to scripts specified in the ScriptText property.

Chapter 3 of the second book, *Leveraging Windows Management Instrumentation (WMI) Scripting* (ISBN 1555582990).

6.4 The permanent consumer registrations in the CIM repository

In using the permanent event consumers, we created several associations of an event filter with particular instances of the permanent event consumers. By using the WMI event registration tool, it is possible to see these instances (see Figure 6.18).

Because each of these instances is associated with the same event filter, changing the view to see the filters gives the permanent event consumers associated with the event filter (see Figure 6.19).

If you look using the **WMI CIM Studio**, you will see the class template of each permanent event consumer used to create the instances associated with the event filter (see Figure 6.20).

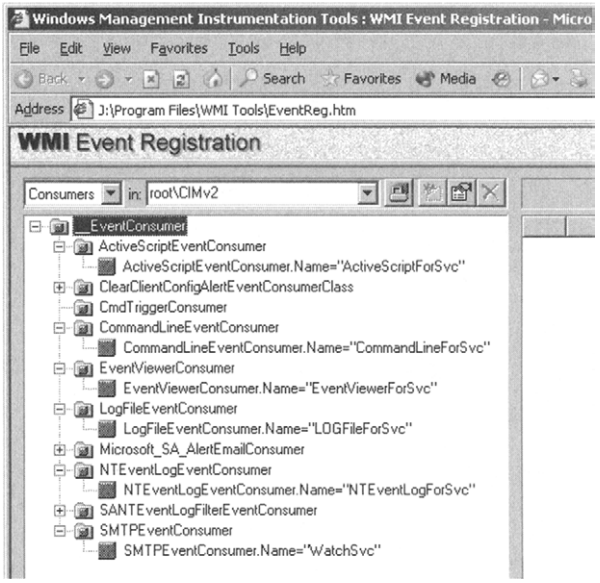

Figure 6.18 *The WMI event registration consumer instances.*

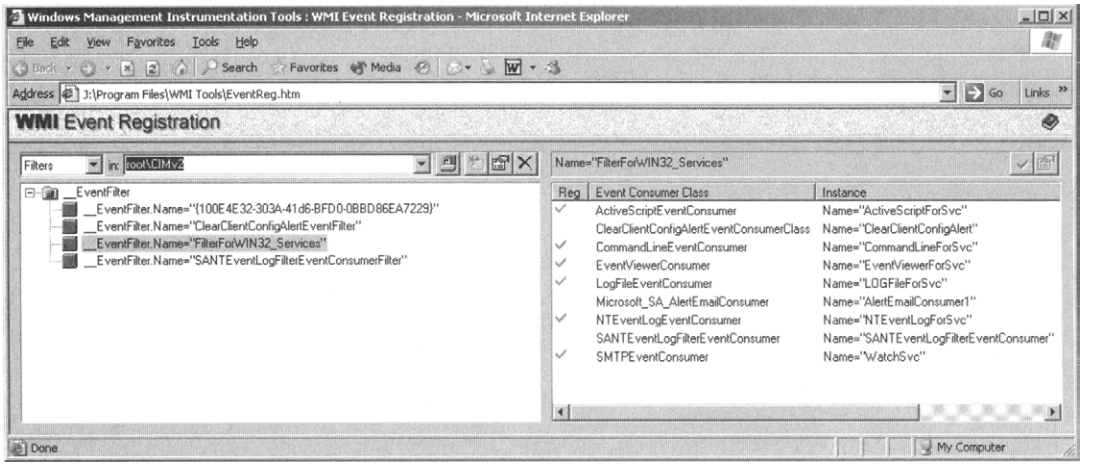

Figure 6.19 *The WMI event filter instances.*

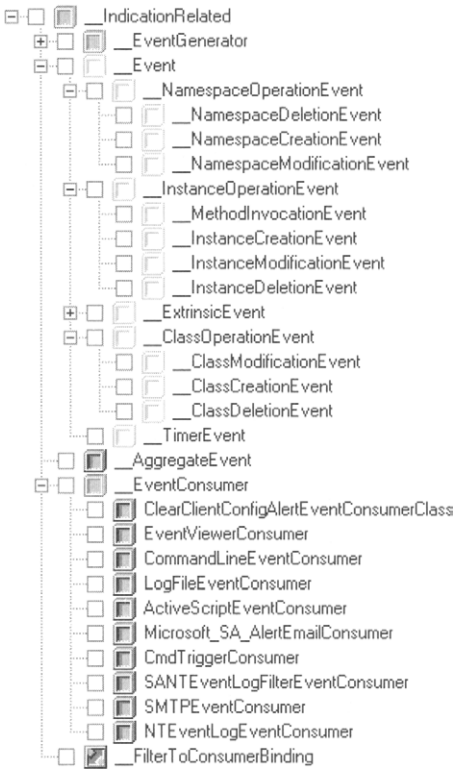

Figure 6.20 *Some WMI permanent event consumers classes.*

6.5 Scripting temporary event consumers

At this point, we have seen many things about the WMI event notifications, but we haven't learned how to script with WMI event notification. All the material we have examined is helpful to position WMI event scripting among all WMI features and to understand how this scripting technique works. Briefly, we have covered the following:

- **Temporary and the permanent subscriptions:** We know that the WMI event scripting uses only temporary subscription. This also helped us to understand the difference between a temporary event consumer and a permanent event consumer.
- **The typical event providers available:** It is possible to know the type of real-world entity that can be monitored without using the polling feature (**WITHIN** statement) of WQL and the type of event class that these event providers provide (extrinsic) or use (intrinsic).

- **The WQL event query:** As soon as we know which event class to use to track events related to instances, the formulation of the WQL event query is key. The query is used as a filter to determine events that are relevant for the consumer. We saw how to formulate WQL queries in Chapter 3.

With this background in mind, we are ready to dive in the WMI event scripting techniques.

When we ended the previous chapter with Sample 5.18 ("Looping in a script to detect a change") by using the *Refresh_* method of the **SWbemObject** and the For Next polling technique, we created a form of monitoring. Note that this technique performs synchronous monitoring of an instance (or collection of instances), but it does not represent a synchronous event-monitoring technique! Sample 5.18 polled an instance to show its current state regardless of whether there was a modification. The big difference with the WMI event notification is that the script is notified as soon as a modification occurs. In this case, the script receives an instance representing the event type.

In Chapter 4, we saw two major scripting techniques: synchronous scripting and asynchronous scripting. We will proceed exactly in the same way here. Let's start with the synchronous scripting technique first.

6.5.1 Synchronous events

In the case of a synchronous event notification, the script execution is in an idle state until a notification is received from WMI. To catch the event notification synchronously, we use the *ExecNotificationQuery* method of the **SWbemServices** object. This method requires a WQL event query in input. This tactic is implemented in Sample 6.13.

Sample 6.13 *Synchronous event notification*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
14:
15:    <script language="VBscript">
16:    <![CDATA[
..:
20:    ' ---------------------------------------------------------------------------
21:    Const cComputerName = "LocalHost"
22:    Const cWMINameSpace = "root/cimv2"
23:    Const cWMIQuery = "Select * from __InstanceModificationEvent Within 10
```

```
                                                    Where TargetInstance ISA 'Win32_Service'"
..:
29:    On Error Resume Next
30:
31:    objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
32:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
33:    Set objWMIServices = objWMILocator.ConnectServer(cComputerName, cWMINameSpace, "", "")
34:
35:    Set objWMIEvent = objWMIServices.ExecNotificationQuery (cWMIQuery)
36:
37:    WScript.Echo "Waiting for events ..."
38:
39:    Set objWMIEventInstance = objWMIEvent.NextEvent
40:    If Err.Number then
41:       WScript.Echo "0x" & Hex(Err.Number) & " - " & Err.Description & " (" & Err.Source & ")"
42:    Else
43:       WScript.Echo
44:       WScript.Echo FormatDateTime(Date, vbLongDate) & " at " & _
45:                    FormatDateTime(Time, vbLongTime) & ": '" & _
46:                    objWMIEventInstance.Path_.Class & "' has been triggered."
47:       WScript.Echo ">The '" & objWMIEventInstance.TargetInstance.DisplayName & _
48:                    "' instance is currently " & _
49:                    LCase (objWMIEventInstance.TargetInstance.State) & "."
50:       WScript.Echo ">Before modification the '" & _
51:                    objWMIEventInstance.PreviousInstance.DisplayName & _
52:                    "' instance was " & _
53:                    LCase (objWMIEventInstance.PreviousInstance.State) & "."
54:    End If
..:
61:    WScript.Echo "Finished."
62:
63:    ]]>
64:   </script>
65:  </job>
66:</package>
```

As in previous scripts, Sample 6.13 establishes the connection with WMI by creating an **SWbemServices** object (line 31 to 33). Next, the script executes the *ExecNotificationQuery* method exposed by the **SWbemServices** object (line 35). The *ExecNotificationQuery* method takes the WQL event query, defined at line 23, as a parameter. This WQL event query requests to watch modifications to any *Win32_Service* instances. Once the *ExecNotificationQuery* method executed, it immediately returns an **SWbemEventSource** object (line 35). This object is part of the WMI scripting object model in Figure 4.3 (object number 9). The **SWbemEventSource** object exposes only one method called *NextEvent*.

If no event occurs, the script waits indefinitely (line 39). Because the script is in an idle state until an event occurs, this technique refers to synchronous event notification. When the *NextEvent* method invocation returns successfully, it means that an event occurred. In such a case, an **SWbemObject** is returned (line 39). This **SWbemObject** represents an instance of the event itself. In Sample 6.13, it is an instance of the

__InstanceModificationEvent class (line 23). We know that the *__InstanceModificationEvent* class exposes two properties that encapsulate an **SWbemObject** representing the instance subject to the modification. It is a *Win32_Service* instance; one property (*PreviousInstance*) shows the *Win32_Service* instance in its previous state (lines 47 to 49), and the other property (*TargetInstance*) shows the *Win32_Service* instance in its new state (lines 50 to 53). We have these two properties available because the WQL query refers to the *__InstanceModificationEvent* intrinsic event class. Refer to Table 6.3 to get a list of the properties available from the intrinsic event classes. Below, you will find sample output when the Windows SNMP Service is stopped.

```
C:\>EventSyncConsumer.wsf
Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.

Waiting for events ...

Sunday, 01 July, 2002 at 10:36: '__InstanceModificationEvent' has been
triggered.
>The 'SNMP' instance is currently stopped.
>Before modification the 'SNMP' instance was running.
Finished.
```

The WQL event query influences the returned objects in two ways: the type of event instance contained in the **SWbemObject** returned from the *NextEvent* method and the type of instance subject to the event (and contained in the properties of the **SWbemObject** returned from the *NextEvent* method). Basically, the WQL event query has two variables: the event itself and the real-world object entity subject to the event.

By using Sample 6.13 as a base and with the knowledge acquired before, we can develop a script that is much more generic. This gives us the facility to formulate any WQL event query type from the command line and get the result displayed accordingly. Once done, we can use this script instead of using **WBEMTEST.exe** as temporary consumer. This generic script can be easily created by combining the XML WSH features to parse the command line and by reusing the DisplayProperties() function created in Chapter 4 (see Sample 4.30). The logic is implemented in Sample 6.14.

Sample 6.14 *A generic script for synchronous event notification*

```
1:<?xml version="1.0"?>
.:
8:<package>
9:  <job>
..:
```

```
13:    <runtime>
14:      <unnamed name="WQLQuery" helpstring="the WQL Event query to execute (between quotes)."
                required="true" type="string" />
15:      <named name="Machine" helpstring="determine the WMI system to connect to.
                (default=LocalHost)" required="false" type="string"/>
16:      <named name="User" helpstring="determine the UserID to perform the remote connection.
                (default=none)" required="false" type="string"/>
17:      <named name="Password" helpstring="determine the password to perform the remote
                connection. (default=none)" required="false" type="string"/>
18:      <named name="NameSpace" helpstring="determine the WMI namespace to connect to.
                (default=Root\CIMv2)" required="false" type="string"/>
19:    </runtime>
20:
21:    <script language="VBScript" src="..\Functions\TinyErrorHandler.vbs" />
22:    <script language="VBScript" src="..\Functions\DisplayInstanceProperties.vbs" />
23:
24:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
25:    <object progid="WbemScripting.SWbemDateTime" id="objWMIDateTime"/>
26:
27:    <script language="VBscript">
28:    <![CDATA[
..:
32:    ' ---------------------------------------------------------------------------------------
33:    Const cComputerName = "LocalHost"
34:    Const cWMINameSpace = "root/cimv2"
..:
48:    ' ---------------------------------------------------------------------------------------
49:    ' Parse the command line parameters
50:    If WScript.Arguments.Unnamed.Count < 1 Then
51:       WScript.Arguments.ShowUsage()
52:       WScript.Quit
53:    Else
54:       strWQLQuery = WScript.Arguments.Unnamed.Item(0)
55:    End If
56:
57:    strUserID = WScript.Arguments.Named("User")
58:    If Len(strUserID) = 0 Then strUserID = ""
59:
60:    strPassword = WScript.Arguments.Named("strPassword")
61:    If Len(strPassword) = 0 Then strPassword = ""
62:
63:    strComputerName = WScript.Arguments.Named("Machine")
64:    If Len(strComputerName) = 0 Then strComputerName = cComputerName
65:
66:    strWMINameSpace = WScript.Arguments.Named("NameSpace")
67:    If Len(strWMINameSpace) = 0 Then strWMINameSpace = cWMINameSpace
68:    strWMINameSpace = UCase (strWMINameSpace)
69:
70:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
71:    objWMILocator.Security_.Privileges.Add wbemPrivilegeSecurity
72:    Set objWMIServices = objWMILocator.ConnectServer(strComputerName, strWMINameSpace, _
73:                                                     strUserID, strPassword)
74:    If Err.Number Then ErrorHandler (Err)
75:
76:    Set objWMIEvent = objWMIServices.ExecNotificationQuery (strWQLQuery)
77:    If Err.Number Then ErrorHandler (Err)
78:
79:    WScript.Echo "Waiting for events ..."
80:
81:    Set objWMIEventInstance = objWMIEvent.NextEvent
```

```
 82:    If Err.Number then
 83:       WScript.Echo "0x" & Hex(Err.Number) & " - " & Err.Description & " (" & Err.Source & ")"
 84:    Else
 85:       WScript.Echo
 86:       WScript.Echo FormatDateTime(Date, vbLongDate) & " at " & _
 87:                    FormatDateTime(Time, vbLongTime) & ": '" & _
 88:                    objWMIEventInstance.Path_.Class & "' has been triggered."
 89:       DisplayProperties objWMIEventInstance, 2
 90:    End If
 ..:
 97:    WScript.Echo "Finished."
 98:
 99:    ]]>
100:   </script>
101:  </job>
102:</package>
```

Except that the script is generic, the logic of Sample 6.14 is not different from that of Sample 6.13. Because any type of WQL event query can be provided from the command line, the script grants a particular privilege required when examining the NT Event Log (line 71). Doing so, if such a query is provided on the command line, access is not denied. Of course, it is possible to add more parameters to include the security requirements from the command line, but as we focus on the WMI event notification scripting techniques, we will try to keep the script as simple as possible.

Lines 13 to 19 define the XML structure used to parse the command line. Lines 50 to 68 extract the information available from the command line. The second adaptation concerns the DisplayProperties() function call (line 89), which displays all properties related to the **SWbemObject** returned from the *NextEvent* method invocation (line 81). To execute the script in the same conditions as Sample 6.13, the following command line must be used:

```
 1:  C:\>GenericEventSyncConsumer.wsf "Select * From __InstanceModificationEvent Within 10
 2:                                    Where TargetInstance ISA 'Win32_Service'"
 3:  Microsoft (R) Windows Script Host Version 5.6
 4:  Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.
 5:
 6:  Waiting for events ...
 7:
 8:  Sunday, 17 June, 2002 at 15:33: '__InstanceModificationEvent' has been triggered.
 9:    PreviousInstance (wbemCimtypeObject)
10:      AcceptPause (wbemCimtypeBoolean) = False
11:      AcceptStop (wbemCimtypeBoolean) = True
12:      Caption (wbemCimtypeString) = SNMP Service
13:      CheckPoint (wbemCimtypeUint32) = 0
14:      CreationClassName (wbemCimtypeString) = Win32_Service
15:      Description (wbemCimtypeString) = Includes agents that monitor the activity ...
16:      DesktopInteract (wbemCimtypeBoolean) = False
17:      DisplayName (wbemCimtypeString) = SNMP Service
18:      ErrorControl (wbemCimtypeString) = Normal
19:      ExitCode (wbemCimtypeUint32) = 0
20:      InstallDate (wbemCimtypeDatetime) = (null)
```

```
21:       Name (wbemCimtypeString) = SNMP
22:       PathName (wbemCimtypeString) = J:\WINDOWS\System32\snmp.exe
23:       ProcessId (wbemCimtypeUint32) = 2208
24:       ServiceSpecificExitCode (wbemCimtypeUint32) = 0
25:       ServiceType (wbemCimtypeString) = Own Process
26:       Started (wbemCimtypeBoolean) = True
27:       StartMode (wbemCimtypeString) = Auto
28:       StartName (wbemCimtypeString) = LocalSystem
29:       State (wbemCimtypeString) = Running
30:       Status (wbemCimtypeString) = OK
31:       SystemCreationClassName (wbemCimtypeString) = Win32_ComputerSystem
32:       SystemName (wbemCimtypeString) = XP-DPEN6400
33:       TagId (wbemCimtypeUint32) = 0
34:       WaitHint (wbemCimtypeUint32) = 0
35:     SECURITY_DESCRIPTOR (wbemCimtypeUint8) = (null)
36:     TargetInstance (wbemCimtypeObject)
37:       AcceptPause (wbemCimtypeBoolean) = False
38:       AcceptStop (wbemCimtypeBoolean) = False
39:       Caption (wbemCimtypeString) = SNMP Service
40:       CheckPoint (wbemCimtypeUint32) = 0
41:       CreationClassName (wbemCimtypeString) = Win32_Service
42:       Description (wbemCimtypeString) = Includes agents that monitor the activity ...
43:       DesktopInteract (wbemCimtypeBoolean) = False
44:       DisplayName (wbemCimtypeString) = SNMP Service
45:       ErrorControl (wbemCimtypeString) = Normal
46:       ExitCode (wbemCimtypeUint32) = 0
47:       InstallDate (wbemCimtypeDatetime) = (null)
48:       Name (wbemCimtypeString) = SNMP
49:       PathName (wbemCimtypeString) = J:\WINDOWS\System32\snmp.exe
50:       ProcessId (wbemCimtypeUint32) = 0
51:       ServiceSpecificExitCode (wbemCimtypeUint32) = 0
52:       ServiceType (wbemCimtypeString) = Own Process
53:       Started (wbemCimtypeBoolean) = False
54:       StartMode (wbemCimtypeString) = Auto
55:       StartName (wbemCimtypeString) = LocalSystem
56:       State (wbemCimtypeString) = Stopped
57:       Status (wbemCimtypeString) = OK
58:       SystemCreationClassName (wbemCimtypeString) = Win32_ComputerSystem
59:       SystemName (wbemCimtypeString) = XP-DPEN6400
60:       TagId (wbemCimtypeUint32) = 0
61:       WaitHint (wbemCimtypeUint32) = 0
62:     TIME_CREATED (wbemCimtypeUint64) = (null)
63:   Finished.
```

In boldface, at the first indentation level (lines 9, 35, 36, and 62), we have the properties of the event instance itself (which is an **SWbemObject** returned from the *NextEvent* method invocation of the **SWbemEventSource** object). We can clearly see the *PreviousInstance* and the *TargetInstance* properties with its **wbemCimtypeObject** type (lines 9 and 36). This type specifies that another **SWbemObject** is encapsulated in these properties. Because the DisplayProperties() function executes recursively when an object is encapsulated in a property, the script displays the properties of the instance subject to the modification (the *Win32_Service* SNMP instance). *PreviousInstance* (lines 10 to 34) contains the instance before modification, and *TargetInstance* (lines 37 to 61) contains the object after modification.

Sample 6.14 displays all items related to an event independently of its nature. Now we have an easy tool to see what we obtain with each event type available from WMI.

6.5.2 Semisynchronous events

The major inconvenience of Samples 6.13 and 6.14 is that the scripts are idle until an event matching the WQL event query occurs. If we slightly modify the logic used in these scripts, it is possible to perform other tasks while we check to see whether a matching event occurred. Sample 6.15 is a modified version of the synchronous generic event consumer script (Sample 6.14).

Sample 6.15 *A generic script for semisynchronous event notification*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <runtime>
14:      <unnamed name="WQLQuery" helpstring="the WQL Event query to execute (between quotes)."
                                  required="true" type="string" />
15:      <unnamed name="EventWaitDelay" helpstring="The number of millseconds to wait for the
                                  event (-1=Infinite)." required="true" type="string" />
16:      <named name="Machine" helpstring="determine the WMI system to connect to.
                                  (default=LocalHost)" required="false" type="string"/>
17:      <named name="User" helpstring="determine the UserID to perform the remote connection.
                                  (default=none)" required="false" type="string"/>
18:      <named name="Password" helpstring="determine the password to perform the remote
                                  connection. (default=none)" required="false" type="string"/>
19:      <named name="NameSpace" helpstring="determine the WMI namespace to connect to.
                                   (default=Root\CIMv2)" required="false" type="string"/>
20:    </runtime>
21:
22:    <script language="VBScript" src="..\Functions\TinyErrorHandler.vbs" />
23:    <script language="VBScript" src="..\Functions\DisplayInstanceProperties.vbs" />
24:
25:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
26:    <object progid="WbemScripting.SWbemDateTime" id="objWMIDateTime" />
27:
28:    <script language="VBscript">
29:    <![CDATA[
..:
33:    ' ---------------------------------------------------------------------------------
34:    Const cComputerName = "LocalHost"
35:    Const cWMINameSpace = "root/cimv2"
..:
50:    ' ---------------------------------------------------------------------------------
51:    ' Parse the command line parameters
52:    If WScript.Arguments.Unnamed.Count < 2 Then
53:       WScript.Arguments.ShowUsage()
54:       WScript.Quit
55:    Else
56:       strWQLQuery = WScript.Arguments.Unnamed.Item(0)
```

```
57:       intEventWaitDelay = WScript.Arguments.Unnamed.Item(1)
58:    End If
59:
60:    strUserID = WScript.Arguments.Named("User")
61:    If Len(strUserID) = 0 Then strUserID = ""
62:
63:    strPassword = WScript.Arguments.Named("strPassword")
64:    If Len(strPassword) = 0 Then strPassword = ""
65:
66:    strComputerName = WScript.Arguments.Named("Machine")
67:    If Len(strComputerName) = 0 Then strComputerName = cComputerName
68:
69:    strWMINameSpace = WScript.Arguments.Named("NameSpace")
70:    If Len(strWMINameSpace) = 0 Then strWMINameSpace = cWMINameSpace
71:    strWMINameSpace = UCase (strWMINameSpace)
72:
73:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
74:    objWMILocator.Security_.Privileges.AddAsString "SeSecurityPrivilege", True
75:    Set objWMIServices = objWMILocator.ConnectServer(strComputerName, strWMINameSpace, _
76:                                                     strUserID, strPassword)
77:    If Err.Number Then ErrorHandler (Err)
78:
79:    objWMIServices.Security_.Privileges.AddAsString "SeSecurityPrivilege", True
80:    Set objWMIEvent = objWMIServices.ExecNotificationQuery (strWQLQuery)
81:    If Err.Number Then ErrorHandler (Err)
82:
83:    WScript.Echo "Waiting events for " & intEventWaitDelay & " ms ..."
84:
85:    Do
86:       Set objWMIEventInstance = objWMIEvent.NextEvent (intEventWaitDelay)
87:       If Err.Number then
88:          WScript.Echo "0x" & Hex(Err.Number) & "-" & Err.Description & " (" & Err.Source & ")"
89:          Exit Do
90:       Else
91:          WScript.Echo
92:          WScript.Echo FormatDateTime(Date, vbLongDate) & " at " & _
93:                       FormatDateTime(Time, vbLongTime) & ": '" & _
94:                       objWMIEventInstance.Path_.Class & "' has been triggered."
95:          DisplayProperties objWMIEventInstance, 2
96:       End If
97:
98:       WScript.Echo "Performing other tasks ..."
99:       WScript.Sleep (10000)
100:
101:      WScript.Echo "Waiting the next events for a new period of " & _
102:                   intEventWaitDelay & " ms ..."
103:
104:   Loop While True
105:
106:   Set objWMIEventInstance = Nothing
107:   Set objWMIEvent = Nothing
108:
109:   Set objWMIServices = Nothing
110:
111:   WScript.Echo "Finished."
112:
113:   ]]>
114:  </script>
115: </job>
116:</package>
```

Because we continue to use the *ExecNotificationQuery* synchronous method of the **SWbemServices** object (line 80) and because the script no longer waits until an event occurs, this technique is referred to as a semisynchronous technique. The trick resides in the way the *NextEvent* method of the **SWbemEventSource** object is called (line 86). This method accepts one parameter value defining the amount of time to wait for an event. Previously, no time was specified. Now the same method is called, but a time is specified. This causes the script to stay idle only for the specified period of time because the *NextEvent* method returns from invocation when the period of time expires. If no event occurred in this delay, a timeout error is returned (lines 87 to 89).

The second particularity of the coding is that the portion of code checking for events is enclosed in a "Do Loop" to keep the script from terminating immediately (lines 85 to 104). Because the script is executing a "Do Loop" while checking for events (line 86) during a limited period of time, it is possible to execute any other required tasks in between. If an event occurs, the script displays all objects related to the event with the help of the DisplayProperties() function (lines 91 to 95). If a timeout occurs (lines 87 to 89), the script terminates its execution.

There are two important remarks regarding this script:

1. If the provided WQL event query uses the **WITHIN** statement, it must be clear that the polling interval specified with the **WITHIN** statement must be shorter than the timeout period specified with the *NextEvent* method. Otherwise, a timeout will occur every time, as WMI will never return an event notification matching the WQL event query before the timeout expires. Of course, for any other queries not using the **WITHIN** statement, this is not an issue.

2. The script continues to run if it receives a WMI event notification in the period of time defined by the timeout. If no event occurs, the script terminates its execution (as a timeout occurs).

The last point could be annoying in some situations if we want to perform continuous semisynchronous monitoring regardless of whether a timeout has occurred. In this case, Sample 6.15 is modified, but only at line 89, to clear the error generated by the timeout. This modification is shown in Sample 6.16.

Sample 6.16 *A generic script for continuous semisynchronous event notification*

```
...:
...:
...:
 84:
 85:    Do
 86:       Set objWMIEventInstance = objWMIEvent.NextEvent (intEventWaitDelay)
 87:       If Err.Number then
 88:          WScript.Echo "0x" & Hex(Err.Number) & "-" & Err.Description & " (" & Err.Source & ")"
 89:          Err.Clear
 90:       Else
 91:          WScript.Echo
 92:          WScript.Echo FormatDateTime(Date, vbLongDate) & " at " & _
 93:                       FormatDateTime(Time, vbLongTime) & ": '" & _
 94:                       objWMIEventInstance.Path_.Class & "' has been triggered."
 95:          DisplayProperties objWMIEventInstance, 2
 96:       End If
 97:
 98:       WScript.Echo "Performing other tasks ..."
 99:       WScript.Sleep (10000)
100:
101:       WScript.Echo "Waiting the next events for a new period of " & _
102:                    intEventWaitDelay & " ms ..."
103:
104:    Loop While True
...:
...:
...:
```

Although this semisynchronous method works well, it is not applicable in all situations. If many tasks must be performed between the check of the WMI event notifications, it could be that the response time is not fast enough. In the same way, if event notifications of a different type must be handled, waiting a predefined timeout period can introduce some delays with regard to response time. The problem comes from the fact that all tasks executed in a semisynchronous event notification scripting logic are serialized by the nature of the coding logic used. To avoid this limitation, the asynchronous scripting technique must be used.

6.5.3 Asynchronous events

Asynchronous event notifications use the *ExecNotificationQueryAsync* method of the **SWbemServices** object. The first part is exactly the same as previous script samples, but the delivery of the instance representing the WMI event is made to a sink routine. In the previous chapter, we worked with the asynchronous scripting techniques. Asynchronous WMI event notification uses the same principle. This implies the use of the **SWbemSink** object and the use of some of the four sink routines: OnCompleted, OnObjectPut, OnObjectReady, and OnProgress. As we work in the context

of a WMI asynchronous event notification, only the sink routines OnCompleted, OnObjectReady, and OnProgress are used. The OnObjectReady is the sink routine receiving the instance representing the event. Sample 6.17 implements this logic. Basically, it combines some of the logic developed in the previous WMI event scripts and some of the logic used in Chapter 4 when we worked with other asynchronous scripting methods.

Sample 6.17 *A generic script for asynchronous event notification*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <runtime>
14:      <unnamed name="WQLQuery" helpstring="the WQL Event query to execute (between quotes)."
                                  required="true" type="string" />
15:      <named name="Machine" helpstring="determine the WMI system to connect to.
                                  (default=LocalHost)" required="false" type="string"/>
16:      <named name="User" helpstring="determine the UserID to perform the remote connection.
                                  (default=none)" required="false" type="string"/>
17:      <named name="Password" helpstring="determine the password to perform the remote
                                  connection. (default=none)" required="false" type="string"/>
18:      <named name="NameSpace" helpstring="determine the WMI namespace to connect to.
                                  (default=Root\CIMv2)" required="false" type="string"/>
19:    </runtime>
20:
21:    <script language="VBScript" src="..\Functions\TinyErrorHandler.vbs" />
22:    <script language="VBScript" src="..\Functions\DisplayInstanceProperties.vbs" />
23:    <script language="VBScript" src="..\Functions\PauseScript.vbs" />
24:
25:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
26:    <object progid="WbemScripting.SWbemDateTime" id="objWMIDateTime" />
27:
28:    <script language="VBscript">
29:    <![CDATA[
..:
33:    ' ---------------------------------------------------------------------
34:    Const cComputerName = "LocalHost"
35:    Const cWMINameSpace = "root/cimv2"
..:
49:    ' ---------------------------------------------------------------------------------
50:    ' Parse the command line parameters
51:    If WScript.Arguments.Unnamed.Count = 0 Then
52:       WScript.Arguments.ShowUsage()
53:       WScript.Quit
54:    Else
55:       strWQLQuery = WScript.Arguments.Unnamed.Item(0)
56:    End If
57:
58:    strUserID = WScript.Arguments.Named("User")
59:    If Len(strUserID) = 0 Then strUserID = ""
60:
61:    strPassword = WScript.Arguments.Named("strPassword")
62:    If Len(strPassword) = 0 Then strPassword = ""
63:
```

```
64:    strComputerName = WScript.Arguments.Named("Machine")
65:    If Len(strComputerName) = 0 Then strComputerName = cComputerName
66:
67:    strWMINameSpace = WScript.Arguments.Named("NameSpace")
68:    If Len(strWMINameSpace) = 0 Then strWMINameSpace = cWMINameSpace
69:    strWMINameSpace = UCase (strWMINameSpace)
70:
71:    Set objWMISink = WScript.CreateObject ("WbemScripting.SWbemSink", "SINK_")
72:
73:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
74:    objWMILocator.Security_.Privileges.AddAsString "SeSecurityPrivilege", True
75:    Set objWMIServices = objWMILocator.ConnectServer(strComputerName, strWMINameSpace, _
76:                                                     strUserID, strPassword)
77:    If Err.Number Then ErrorHandler (Err)
78:
79:    objWMIServices.ExecNotificationQueryAsync objWMISink, strWQLQuery,, wbemFlagSendStatus
80:    If Err.Number Then ErrorHandler (Err)
81:
82:    WScript.Echo "Waiting for events..."
83:
84:    PauseScript "Click on 'Ok' to terminate the script ..."
85:
86:    WScript.Echo vbCRLF & "Cancelling event subscription ..."
87:    objWMISink.Cancel
..:
92:    WScript.Echo "Finished."
93:
94:    ' ------------------------------------------------------------------------------------
95:    Sub SINK_OnCompleted (iHResult, objWBemErrorObject, objWBemAsyncContext)
96:
97:        Wscript.Echo
98:        Wscript.Echo "BEGIN - OnCompleted."
99:        Wscript.Echo "END   - OnCompleted."
100:
101:   End Sub
102:
103:   ' ------------------------------------------------------------------------------------
104:   Sub SINK_OnObjectReady (objWbemObject, objWbemAsyncContext)
105:
106:       Wscript.Echo
107:       Wscript.Echo "BEGIN - OnObjectReady."
108:       WScript.Echo FormatDateTime(Date, vbLongDate) & " at " & _
109:                    FormatDateTime(Time, vbLongTime) & ": '" & _
110:                    objWbemObject.Path_.Class & "' has been triggered."
111:
112:       DisplayProperties objWbemObject, 2
113:
114:       Wscript.Echo "END - OnObjectReady."
115:
116:   End Sub
117:
118:   ' ------------------------------------------------------------------------------------
119:   Sub SINK_OnProgress (iUpperBound, iCurrent, strMessage, objWbemAsyncContext)
120:
121:       Wscript.Echo
122:       Wscript.Echo "BEGIN - OnProgress."
123:       Wscript.Echo "END   - OnProgress."
124:
125:   End Sub
126:
```

```
127:    ]]>
128:   </script>
129:  </job>
130:</package>
```

The script is exactly the same as Sample 6.16 until line 71. At line 71 the **SWbemSink** object is created and initialized with a name corresponding to the first part of the sink routine name (SINK_). Next, the script performs the WMI connection as before. At line 79, the asynchronous method *ExecNotificationQueryAsync* is executed with the specified WQL event query and the corresponding **SWbemSink** object. As we have seen in previous chapters with asynchronous scripting, the script sample pauses (line 84) to avoid its termination. We have seen that a script is a temporary event consumer, so it is important to keep the script running to receive events. In a production environment, it is likely that some other tasks will be executed while receiving WMI events. Once the script starts and registers for events, the received event is displayed in the OnObjectReady sink (lines 104 to 116) by using the DisplayProperties() function, as before (line 112).

Figure 6.21 *The GenericEventAsyncConsumer.wsf script execution.*

The asynchronous event scripting technique is a powerful one. Of course, the script we discovered during this section is an academic script as it shows the mechanism that must be in place. Besides the event scripting technique, it is possible to code any type of management logic in a script. Let's see how we can use the power of the asynchronous event scripting technique to manage a Windows environment.

6.6 Going further with event scripting

Regarding the number of classes (and behind the scenes, the number of providers), it is almost impossible to give one sample for each class and each event type that exists with WMI. Although we review the WMI providers one by one in the second book, *Leveraging Windows Management Instrumentation (WMI) Scripting* (ISBN 1555582990), in this section we give three script samples that use most of the things we have discovered about WMI. In addition to showing other script samples of event scripting, they also provide a good synthesis of what we have already learned before discussing the WMI providers and their capabilities.

6.6.1 Monitoring, managing, and alerting script for the Windows services

The purpose of the script is to monitor a series of Windows services specified by the administrator to minimize the number of interventions when a Windows service is stopped. As soon as one of the specified services is stopped, the script restarts the service. If there is a problem, it is likely that the service won't be able to be restarted. In such a case, the script attempts to restart the service a certain number of times. For this, it maintains a sanity counter per service monitored. The script sends an alert once the number of trials has been reached. The alert is sent by e-mail, which is built on top of Collaboration Data Object (CDO) for Windows. Although, we don't focus on CDO in this book, we provide a brief explanation of how this works. The e-mail contains the status of the *PreviousInstance* and the *TargetInstance* of the concerned service. To make the WMI information easier to read, the script formats the message in HTML.

The script can monitor several services and maintains a sanity counter per service. To help achieve this, the script uses the notion of context. We used WMI contexts in the previous chapter when we examined asynchronous scripting. Samples 6.18 and 6.19 implement the logic described. Sample 6.18 is the initialization section of the script. Sample 6.19 is the sink routine handling the WMI asynchronous notifications.

Sample 6.18 *Monitoring, managing, and alerting script for the Windows services (Part I)*

```
 1:<?xml version="1.0"?>
 .:
 8:<package>
 9:  <job>
..:
13:    <runtime>
14:      <unnamed name="ServiceName" helpstring="the Windows Service list to monitor."
                  required="true" type="string" />
15:      <named name="Machine" helpstring="determine the WMI system to connect to."
                  required="false" type="string"/>
16:      <named name="User" helpstring="determine the UserID to perform the remote connection."
                  required="false" type="string"/>
17:      <named name="Password" helpstring="determine the password to perform the remote
                  connection. (default=none)" required="false" type="string"/>
18:    </runtime>
19:
20:    <script language="VBScript" src="..\Functions\TinyErrorHandler.vbs" />
21:    <script language="VBScript" src="..\Functions\PauseScript.vbs" />
22:    <script language="VBScript" src="..\Functions\LoopSvcStartupRetry.vbs" />
23:    <script language="VBScript" src="..\Functions\GenerateHTML.vbs" />
24:    <script language="VBScript" src="..\Functions\SendMessageExtendedFunction.vbs" />
25:
26:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
27:    <object progid="WbemScripting.SWbemNamedValueSet" id="objWMIInstanceSinkContext"/>
28:    <object progid="WbemScripting.SWbemDateTime" id="objWMIDateTime" />
29:
30:    <script language="VBscript">
31:    <![CDATA[
..:
35:    ' ---------------------------------------------------------------------------------
36:    Const cComputerName = "LocalHost"
37:    Const cWMINameSpace = "root/cimv2"
38:    Const cWMIClass = "Win32_Service"
39:    Const cWMIQuery = "Select * from __InstanceModificationEvent Within 10
                                Where TargetInstance ISA 'Win32_Service'"
40:
41:    Const cPauseBetweenRestart = 2
42:    Const cRestartLimit = 3
43:
44:    Const cTargetRecipient = Alain.Lissoir@LissWare.NET
45:    Const cSourceRecipient = WMISystem@LissWare.NET
46:
47:    Const cSMTPServer = "10.10.10.202"
48:    Const cSMTPPort = 25
49:    Const cSMTPAccountName = ""
50:    Const cSMTPSendEmailAddress = ""
51:    Const cSMTPAuthenticate = 0' 0=Anonymous, 1=Basic, 2=NTLM
52:    Const cSMTPUserName = ""
53:    Const cSMTPPassword = ""
54:    Const cSMTPSSL = False
55:    Const cSMTPSendUsing = 2            ' 1=Pickup, 2=Port, 3=Exchange WebDAV
..:
61:    Class clsMonitoredService
62:          Public strServiceName
```

```
63:         Public intServiceRetryCounter
64:    End Class
...:
78:    ' ------------------------------------------------------------------------------
79:    ' Parse the command line parameters
80:    If WScript.Arguments.Unnamed.Count = 0 Then
81:       WScript.Arguments.ShowUsage()
82:       WScript.Quit
83:    Else
84:       For intIndice = 0 To WScript.Arguments.Unnamed.Count - 1
85:           ReDim Preserve clsService(intIndice)
86:           Set clsService(intIndice) = New clsMonitoredService
87:           clsService(intIndice).strServiceName =
                                    Ucase (WScript.Arguments.Unnamed.Item(intIndice))
88:           clsService(intIndice).intServiceRetryCounter = 0
89:       Next
90:    End If
91:
92:    strUserID = WScript.Arguments.Named("User")
93:    If Len(strUserID) = 0 Then strUserID = ""
...:
101:   Set objWMISink = WScript.CreateObject ("WbemScripting.SWbemSink", "SINK_")
102:
103:   objWMILocator.Security_.AuthenticationLevel = wbemAuthenticationLevelDefault
104:   objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate
105:   Set objWMIServices = objWMILocator.ConnectServer(strComputerName, cWMINameSpace, _
106:                                                    strUserID, strPassword)
...:
109:   For intIndice = 0 To UBound (clsService)
110:
111:       Set objWMIInstance = objWMIServices.Get (cWMIClass & "='" & _
112:                                    clsService(intIndice).strServiceName & "'")
113:
114:       boolSvcStatus = LoopServiceStartupRetry (objWMIInstance, intIndice)
115:       If boolSvcStatus = False Then
116:          WScript.Quit
117:       End If
118:
119:       If Len(strWMIQuery) = 0 Then
120:          strWMIQuery = "TargetInstance.Name='" & clsService(intIndice).strServiceName & "'"
121:       Else
122:          strWMIQuery = strWMIQuery & " Or " & _
123:                        "TargetInstance.Name='" & _
124:                        clsService(intIndice).strServiceName & "'"
125:       End If
126:
127:       WScript.Echo "Adding '" & clsService(intIndice).strServiceName & _
128:                    "' to subscription to monitor '" & cWMIClass & "'." & vbCRLF
129:
130:       objWMIInstanceSinkContext.Add Cstr(clsService(intIndice).strServiceName), intIndice
131:   Next
132:
133:   strWMIQuery = cWMIQuery & " And TargetInstance.State='Stopped' And (" & strWMIQuery & ")"
134:
135:   objWMIServices.ExecNotificationQueryAsync objWMISink, _
136:                                            strWMIQuery, _
137:                                            , _
138:                                            , _
139:                                            , _
140:                                            objWMIInstanceSinkContext
```

```
...:
143:    WScript.Echo "Waiting for events..."
144:
145:    PauseScript "Click on 'Ok' to terminate the script ..."
146:
147:    WScript.Echo vbCRLF & "Cancelling event subscription ..."
148:    objWMISink.Cancel
...:
153:    WScript.Echo "Finished."
154:
...:
...:
...:
```

As we can see, the initialization part of the script is quite a bit more complex than in previous samples. Because the alert notification is made by e-mail, the script includes some parameters required to send e-mails (lines 44 to 55).

The script needs to specify at least one Windows service on the command line. Several Windows services can also be specified. For instance, the command line could be as follows:

```
C:\>ServiceMonitor.Wsf SNMP SNMPTRAP
```

With this command line, the script monitors the SNMP and the SNMPTRAP services. Therefore, the command line accepts a collection of services where at least one service must be given. This is why the script tests the presence of some unnamed parameters (line 80). The unnamed parameters represent a collection that is enumerated (lines 84 to 89) and stored in an array (line 87). Each service has its associated counter initialized to zero (line 88). This is why a VBScript class is used to associate the counter to the service name (lines 61 to 64 and line 86). Next, the script examines any optional parameters (lines 92 to 100) provided on the command line. As the script uses an asynchronous event notification, it creates an **SWbemSink** object (line 101). Once created, it connects to the system where the service list must be monitored (lines 111 and 112).

Because we may have several services, the script creates one event subscription combining all services to watch. The WQL query is constructed from the service list. To construct this WQL query, the script enumerates the collection of services (lines 109 to 131). During this loop, several operations are performed. Because the script monitors services that are stopped, the script ensures that all monitored services are already started. For this, it uses a function called LoopServiceStartupRetry() (line 114). We revisit this function later, but basically, the function starts a service when it is not started. If the startup of one of the monitored services fails, the script termi-

nates its execution (line 116). If a monitored service can't be started, there is no reason to perform its monitoring. The problem must be fixed first.

Next, the WQL event query is constructed for each service in the service list (lines 119 to 125). For further help in the sink routine, the script initializes an **SWbemNamedValueSet** object to add the notion of context to the subscription (line 130). This eases the retrieval of the counter for the corresponding service when an event occurs. We revisit the context usage later; for now, just note that the service name is used as a key in the **SWbemNamedValueSet** and that the corresponding index is associated with it.

Once the service list enumeration completes, the script finalizes the WQL query construction by adding the **Select** statement (defined in line 39) and the condition stating that the service must be stopped (line 133). Last but not least, the subscription is executed by invoking the *ExecNotificationQueryAsync* method of the **SWbemServices** object (lines 135 to 140). That's it for the initialization part.

Why use one subscription for all monitored services instead of using one subscription per service? Currently, the WQL event query used for the subscription is as follows:

```
Select * from __InstanceModificationEvent Within 10 Where
                                          TargetInstance ISA 'Win32_Service' And
                                          TargetInstance.State='Stopped' And
                                          (TargetInstance.Name='SNMP' Or
                                          TargetInstance.Name='SNMPTRAP')
```

Now, it is possible to create a subscription per service listed on the command line. In this case, for the SNMP and the SNMPTRAP services, the WQL event queries are as follows:

```
Select * from __InstanceModificationEvent Within 10 Where
                                          TargetInstance ISA 'Win32_Service' And
                                          TargetInstance.Name='SNMP' And
                                          TargetInstance.State='Stopped'"
```

and

```
Select * from __InstanceModificationEvent Within 10 Where
                                          TargetInstance ISA 'Win32_Service' And
                                          TargetInstance.Name='SNMPTRAP' And
                                          TargetInstance.State='Stopped'"
```

If the script uses this subscription type, the number of resources requested of the system is higher, as we end up with one subscription per service. Because the event instance to watch is the same for all services (*__InstanceModificationEvent*), it is not necessary to use different subscriptions. When the event nature is the same, it is always best to use the same

subscription if possible, which minimizes the number of resources requested.

Now, let's examine WMI asynchronous event management. Sample 6.19 shows the code logic. Like any other WMI event script, the event is delivered to the sink routine called OnObjectReady (lines 167 to 213).

Sample 6.19 *Monitoring, managing, and alerting script for the Windows services (Part II)*

```
...:
...:
...:
154:
155:    ' ------------------------------------------------------------------------------
156:    Sub SINK_OnCompleted (iHResult, objWBemErrorObject, objWBemAsyncContext)
...:
160:        Wscript.Echo
161:        Wscript.Echo "BEGIN - OnCompleted."
162:        Wscript.Echo "END   - OnCompleted."
163:
164:    End Sub
165:
166:    ' ------------------------------------------------------------------------------
167:    Sub SINK_OnObjectReady (objWbemObject, objWbemAsyncContext)
...:
175:        Wscript.Echo
176:        Wscript.Echo "BEGIN - OnObjectReady."
177:        WScript.Echo FormatDateTime(Date, vbLongDate) & " at " & _
178:                     FormatDateTime(Time, vbLongTime) & ": '" & _
179:                     objWbemObject.Path_.Class & "' has been triggered."
180:
181:        Select Case objWbemObject.Path_.Class
182:               Case "__InstanceModificationEvent"
183:                    Set objWMIInstance = objWbemObject
184:               Case "__AggregateEvent"
185:                    Set objWMIInstance = objWbemObject.Representative
186:               Case Else
187:                    Set objWMIInstance = Null
188:        End Select
189:
190:        If Not IsNull (objWMIInstance) Then
191:           boolSvcStatus = LoopServiceStartupRetry (objWMIInstance.TargetInstance, _
192:                           objWbemAsyncContext.Item (objWMIInstance.TargetInstance.Name).Value)
193:
194:           If boolSvcStatus =  False Then
195:              If SendMessage (cTargetRecipient, _
196:                              cSourceRecipient, _
197:                              objWMIInstance.TargetInstance.SystemName & " - " & _
198:                                 FormatDateTime(Date, vbLongDate) & _
199:                                 " at " & _
200:                                 FormatDateTime(Time, vbLongTime), _
201:                              GenerateHTML (objWMIInstance.PreviousInstance, _
202:                                            objWMIInstance.TargetInstance) , _
203:                              "") Then
204:                 WScript.Echo "Failed to send email to '" & cTargetRecipient & "' ..."
205:              End If
206:           End If
```

```
207:        End If
...:
211:        Wscript.Echo "END - OnObjectReady."
212:
213:    End Sub
214:
215:    ' ------------------------------------------------------------------------------
216:    Sub SINK_OnProgress (iUpperBound, iCurrent, strMessage, objWbemAsyncContext)
...:
220:        Wscript.Echo
221:        Wscript.Echo "BEGIN - OnProgress."
222:        Wscript.Echo "END   - OnProgress."
223:
224:    End Sub
225:
226:    ]]>
227:   </script>
228:  </job>
229:</package>
```

As soon as the event is retrieved in the sink routine, it displays the event class received (line 179). Currently, the WQL query uses the *__InstanceModificationEvent* intrinsic class (see Sample 6.18, line 39). The WQL query can be modified while continuing to use the *__InstanceModificationEvent* intrinsic class. For instance, the usage of the **GROUP** statement (see Chapter 3) in the query creates an aggregated event (using a class *__AggregateEvent*). This is why the script verifies the event class type provided through a **Select Case** statement to make sure that the examined instance always represents an instance modification event (lines 181 to 188).

Next, the script tries to restart the stopped service (lines 191 to 192). The sink routine tries to restart the service immediately because only a stopped service invokes the sink routine; this is because the WQL query clearly states that only stopped services can trigger an event (see Sample 6.18, line 133). To restart the service, we recognize the LoopServiceStartupRetry() function already used in the initialization of the script (see Sample 6.18, line 114). This function requires two parameters: the **SWbemObject** object that contains the instance of the service to be restarted and the index corresponding to the service instance examined in the sink routine. This is where the notion of context is used. The index is used to retrieve the service counter in the LoopServiceStartupRetry() function. Line 192 retrieves this index.

```
192:                objWbemAsyncContext.Item (objWMIInstance.TargetInstance.Name).Value)
```

The examined service name is contained in

```
objWMIInstance.TargetInstance.Name
```

Using the service name as a key in the **SWbemNamedValueSet** object retrieves the corresponding index value. Without the notion of context, it is necessary to perform a loop in the array containing the service list (see Sample 6.18, line 87) until the name of the examined service matches one in the service list. Using the notion of context is a much more elegant approach.

The LoopServiceStartupRetry() function contains the logic to continue to restart the service until the maximum number of retries is reached. Depending on the result, the function returns a value of True if it succeeds in restarting the service or of False if it fails. Let's take a look at Sample 6.20, which shows the LoopServiceStartupRetry() function.

Sample 6.20 *Restarting a service for a fixed number of times*

```
 .:
 .:
 6:' ------------------------------------------------------------------------------------------
 7:Function LoopServiceStartupRetry (objWMIInstance, intIndice)
..:
13:    clsService(intIndice).intServiceRetryCounter = 0
14:
15:    Do
16:         WScript.Echo "Service '" & objWMIInstance.DisplayName & _
17:                      "' is '" & UCase(objWMIInstance.State) & _
18:                      "' (Startup mode is '" & objWMIInstance.StartMode & "')."
19:
20:         boolSvcStatus = GetServiceStatus (objWMIInstance)
21:
22:         If boolSvcStatus Then
23:            LoopServiceStartupRetry = True
24:            Exit Do
25:         End If
26:
27:         If clsService(intIndice).intServiceRetryCounter >= cRestartLimit Then
28:            WScript.Echo "Retry limit reached (" & _
29:                         clsService(intIndice).intServiceRetryCounter & "/" & cRestartLimit & _
30:                         ") for service '" & objWMIInstance.DisplayName & "'."
31:            LoopServiceStartupRetry = False
32:            Exit Do
33:         Else
34:            clsService(intIndice).intServiceRetryCounter = _
35:                                          clsService(intIndice).intServiceRetryCounter + 1
36:
37:            ' Pause x milliseconds between status checkings and restart tentatives.
38:            WScript.Echo "Waiting " & cPauseBetweenRestart & _
39:                         " second(s) before rechecking status and retrying."
40:            WScript.Sleep (cPauseBetweenRestart * 1000)
41:
42:            WScript.Echo "Retry " & _
43:                         clsService(intIndice).intServiceRetryCounter & "/" & cRestartLimit & _
44:                         " for service '" & objWMIInstance.DisplayName & "."
45:         End If
46:
47:         WScript.Echo "Starting service '" & objWMIInstance.DisplayName & "' ..."
48:
```

```
49:         If objWMIInstance.StartService Then
50:            WScript.Echo "Service '" & objWMIInstance.DisplayName & _
51:                         "' startup failed!'"
52:         End If
53:
54:         ' Refresh the service instance since its restart.
55:         objWMIInstance.Refresh_
56:     Loop
57:
58:End Function
59:
60:' ---------------------------------------------------------------------------------------
61:Function GetServiceStatus (objWMIInstance)
62:
63:     Select Case UCase(objWMIInstance.State)
64:            Case "RUNNING"
65:                 GetServiceStatus = True
66:            Case "STOPPED"
67:                 GetServiceStatus = False
68:            Case "START PENDING"
69:                 GetServiceStatus = True
70:            Case "STOP PENDING"
71:                 ' Returns True because it is impossible to restart a
72:                 ' service while it is in 'STOP PENDING' state.
73:                 ' WMI trigger another event when the service will be in 'STOPPED' state.
74:                 GetServiceStatus = True
75:            Case Else
76:                 GetServiceStatus = False
77:     End Select
78:
79:End Function
```

This routine starts by resetting the counter to zero. We will see further that the removal of this line changes the script behavior. But first, we must understand how the LoopServiceStartupRetry() function works.

After displaying the current state of the service (lines 16 to 18), the function calls a subfunction called GetServiceStatus() (line 20). This function (lines 61 to 79) returns a Boolean value based on the service state. Basically, if the service is stopped, the function returns False (line 20). If the service is started, the function returns True. In this case, there is no need to restart the service, and we exit from the LoopServiceStartupRetry() function (lines 22 to 25). The LoopServiceStartupRetry() and GetServiceStatus() functions are not written for situations where the service is only stopped. This is why the GetServiceStatus() function returns True in most of the services states, even when the service is in a STOP PENDING state. This will avoid an attempt to restart the service while it is in a STOP PENDING state, as it is impossible to restart a service until the state is STOPPED. As soon as the service is in a STOPPED state, WMI triggers another event (based on the WQL query), and the script can process the stopped service accordingly. Although it is not needed to test all service states because the actual WQL query ensures that only STOPPED services are passed to the sink routine,

the function has been written for general purposes and not only in the context of this WQL query. As an exercise, you can change the submitted WQL event query by removing the statement that ensures that the service is stopped; you will see that the code behaves accordingly.

So, if the service is stopped, the LoopServiceStartupRetry() routine checks to see whether the maximum limit for the counter has been reached (line 27). If the limit was reached, the routine shows the current state of the counter and returns a False Boolean value to indicate that the service has not been restarted (lines 28 to 32). If the limit was not reached, the routine increments the counter by one, makes a pause, and displays the new counter value (lines 34 to 44).

Next, the service is restarted (line 49). If the startup fails, a message is displayed (lines 50 and 51). Because it is likely that the service instance has changed (due to the *StartService* method invocation), the routine refreshes the instance to get its new status (line 55) by using the *Refresh_* method of the **SWbemObject** object. This is a typical application of the *Refresh_* method. This is exactly the purpose for which it was designed. Keep in mind that this method is available only under Windows XP and Windows.NET Server. For instance, under Windows 2000, it is necessary to get a new instance of the service to know its new status.

Finally, the routine performs a loop to re-execute the process from the beginning (line 56). If the service instance has been restarted, the GetServiceStatus() returns a True Boolean value (line 20), which allows the routine to exit (lines 22 to 25). If not, the same logic is executed, and the counter is increased by one again (line 34). This is repeated until the limit is reached.

Now, let's revisit the counter itself. Because, the LoopServiceStartupRetry() function resets the counter every time the routine is called, there is no need to save the counter state. If there is no need to keep the counter state, there is no need to use the notion of context during the asynchronous sink calls. If the script resets the counter to zero each time the LoopServiceStartupRetry() is called, it limits the number of attempts to restart a service. In this case, a service can be stopped 10 times; as soon as it is restarted by the LoopServiceStartupRetry() function, there is no problem and no one will receive an alert. Keep in mind that an alert is sent only if the service restart fails in the LoopServiceStartupRetry() function.

If the line resetting the counter is commented out (line 13) each time a service is stopped, the counter starts at the value it had during the previous call. In this case, the number of times that a service can be stopped will be limited as the code remembers the value along the different calls. The script

will restart a service until the maximum counter value is reached. Once reached, the script does not start the service any more and an alert is sent. The script keeps the counter value by using the notion of context. Note that by removing a single line (line 13), we have totally changed the script logic; we limit the number of attempts to restart a service, but we also limit the number of times it can be detected as stopped.

To send an alert, the script sends an e-mail. The e-mail is an SMTP mail formatted in MIME. To perform this, the script uses CDO for Windows. Before sending the mail, the script must prepare the mail content. The script prepares an HTML body with the help of the function GenerateHTML() (lines 201 to 202 of Sample 6.19). The GenerateHTML() function is a quite complex string manipulation function that enumerates the **SWbemObject** properties and encapsulates them with their syntax and values between HTML tags. It is nothing more that an enhanced version of the DisplayProperties() function developed in Chapter 4. Note that we can use the *GetText_* method to generate an XML representation of an instance. Transforming this XML representation with the extensible styleheet language (XSL) offers another way to generate the HTML representation. Because the XML instance representation is available only under Windows XP and Windows.NET Server, using the GenerateHTML() function allows the creation of an HTML instance representation under Windows 2000 and previous platform versions. The returned result is a string that contains the HTML stream (lines 201 to 202).

Once the HTML body is ready, the script invokes the SendMessage() function shown in Sample 6.21. The function constructs and sends an e-mail with CDO for Windows. The first four parameters are mandatory: the To field, the From field, the Subject, and the HTML body. Only the attachment field is optional.

Sample 6.21 *Sending an e-mail with CDO for Windows*

```
..:
10:' ------------------------------------------------------------------------------
11:Function SendMessage (strTO, strFROM, strSubject, strHTMLBody, strAttachment)
..:
18:        strTO = Trim (strTO)
19:        strFrom = Trim (strFrom)
20:        strSubject = Trim (strSubject)
21:        strHTMLBody = Trim (strHTMLBody)
22:
23:        If Not (CBool(Len (strTO)) And _
24:                CBool(Len(strFrom)) And _
25:                CBool(Len(strSubject)) And _
26:                CBool(Len(strHTMLBody))) Then
27:           SendMessage = True
```

```
28:           Exit Function
29:        End If
30:
31:        Set objMessage = CreateObject("CDO.Message")
32:        Set objConfiguration = CreateObject ("CDO.Configuration")
33:
34:        objConfiguration.Fields
      ("http://schemas.microsoft.com/cdo/configuration/smtpserver") = cSMTPServer
35:        objConfiguration.Fields
      ("http://schemas.microsoft.com/cdo/configuration/smtpserverport") = cSMTPPort
36:        objConfiguration.Fields
      ("http://schemas.microsoft.com/cdo/configuration/smtpaccountname") = cSMTPAccountName
37:        objConfiguration.Fields
      ("http://schemas.microsoft.com/cdo/configuration/sendemailaddress") = cSMTPSendEmailAddress
38:        objConfiguration.Fields
      ("http://schemas.microsoft.com/cdo/configuration/smtpauthenticate") = cSMTPAuthenticate
39:        objConfiguration.Fields
      ("http://schemas.microsoft.com/cdo/configuration/sendusername") = cSMTPUserName
40:        objConfiguration.Fields
      ("http://schemas.microsoft.com/cdo/configuration/sendpassword") = cSMTPPassword
41:        objConfiguration.Fields
      ("http://schemas.microsoft.com/cdo/configuration/smtpusessl") = cSMTPSSL
42:        objConfiguration.Fields
      ("http://schemas.microsoft.com/cdo/configuration/sendusing") = cSMTPSendUsing
43:        objConfiguration.Fields.Update
44:
45:        objMessage.Configuration = objConfiguration
46:        objMessage.To = strTO
47:        objMessage.From = strFrom
48:        objMessage.Subject = strSubject
49:        objMessage.HTMLBody = strHTMLBody
50:
51:        If Len(Trim (strAttachment)) Then
52:           objMessage.AddAttachment strAttachment
53:        End If
54:
55:        objMessage.Send
56:        If Err.Number Then
57:           SendMessage = True
58:        Else
59:           SendMessage = False
60:        End If
...:
68:End Function
```

The function checks whether these parameters are specified (lines 23 to 29). Next, the miscellaneous parameters to send an SMTP mail are set (lines 34 to 43). The function uses a **CDO.Configuration** object created at line 32. Next, the configuration parameters are associated with the **CDO.Message** object created at line 31. The **CDO.Message** object represents the MIME message. The various fields required for an e-mail are set before it is sent (line 55). In case of failure, the script returns a True Boolean value (line 57).

This returned value is used in Sample 6.19 (line 195) to display an error message (line 204). Once the SendMessage() function completes (line 195),

the sink routine terminates, and the script returns to an idle state until another WMI event matching the WQL event query occurs.

When executed, the script displays the following output if a stopped service is restarted successfully:

```
 1:C:\>ServiceMonitor.wsf SNMP SNMPTRAP
 2:Microsoft (R) Windows Script Host Version 5.6
 3:Copyright (C) Microsoft Corporation 1996-2000. All rights reserved.
 4:
 5:Service 'SNMP' is 'STOPPED' (Startup mode is 'Manual').
 6:Waiting 2 second(s) before rechecking status and retrying.
 7:Retry 1/3 for service 'SNMP.
 8:Starting service 'SNMP' ...
 9:Service 'SNMP' is 'RUNNING' (Startup mode is 'Manual').
10:Adding 'SNMP' to subscription to monitor Win32_Service 'SNMP'.
11:
12:Service 'SNMP Trap Service' is 'RUNNING' (Startup mode is 'Manual').
13:Adding 'SNMPTRAP' to subscription to monitor Win32_Service 'SNMPTRAP'.
14:
15:Waiting for events...
16:
17:BEGIN - OnObjectReady.
18:Sunday, 01 July, 2002 at 16:02: '__InstanceModificationEvent' has been triggered.
19:Service 'SNMP' is 'STOPPED' (Startup mode is 'Manual').
20:Waiting 2 second(s) before rechecking status and retrying.
21:Retry 1/3 for service 'SNMP.
22:Starting service 'SNMP' ...
23:Service 'SNMP' is 'START PENDING' (Startup mode is 'Manual').
24:END - OnObjectReady.
25:
..:
..:
```

As we can see before monitoring the services, the script starts the services that are not yet started (lines 5 to 9). Once started (lines 8 and 9) or if already started (line 12), the service list is added in a WQL event query (lines 10 and 13). Next, the script waits for any events matching the event subscription (line 15). Once a service is stopped, the script processes the implemented logic in the LoopServiceStartupRetry() function and tries to start the service while counting the number of attempts (lines 17 to 24).

When the stopped service can't be restarted, the output looks like the following:

```
..:
..:
25:
26:BEGIN - OnObjectReady.
27:Sunday, 01 July, 2002 at 16:04: '__InstanceModificationEvent' has been triggere
28:Service 'SNMP' is 'STOPPED' (Startup mode is 'Disabled').
29:Waiting 2 second(s) before rechecking status and retrying.
30:Retry 1/3 for service 'SNMP.
31:Starting service 'SNMP' ...
32:Service 'SNMP' startup failed!'
33:Service 'SNMP' is 'STOPPED' (Startup mode is 'Disabled').
```

```
34:Waiting 2 second(s) before rechecking status and retrying.
35:Retry 2/3 for service 'SNMP.
36:Starting service 'SNMP' ...
37:Service 'SNMP' startup failed!'
38:Service 'SNMP' is 'STOPPED' (Startup mode is 'Disabled').
39:Waiting 2 second(s) before rechecking status and retrying.
40:Retry 3/3 for service 'SNMP.
41:Starting service 'SNMP' ...
42:Service 'SNMP' startup failed!'
43:Service 'SNMP' is 'STOPPED' (Startup mode is 'Disabled').
44:Retry limit reached (3/3) for service 'SNMP'.
45:END - OnObjectReady.
..:
..:
```

The process is exactly the same as before, but here the script retries up to three times to restart the service (lines 30, 35, and 40). Because, the service is disabled (lines 28, 33, 38, and 43), the service restart fails and an e-mail alert is sent. The received mail is shown in Figure 6.22, which shows the state of the service before (*PreviousInstance*) and after the (*TargetInstance*) modification.

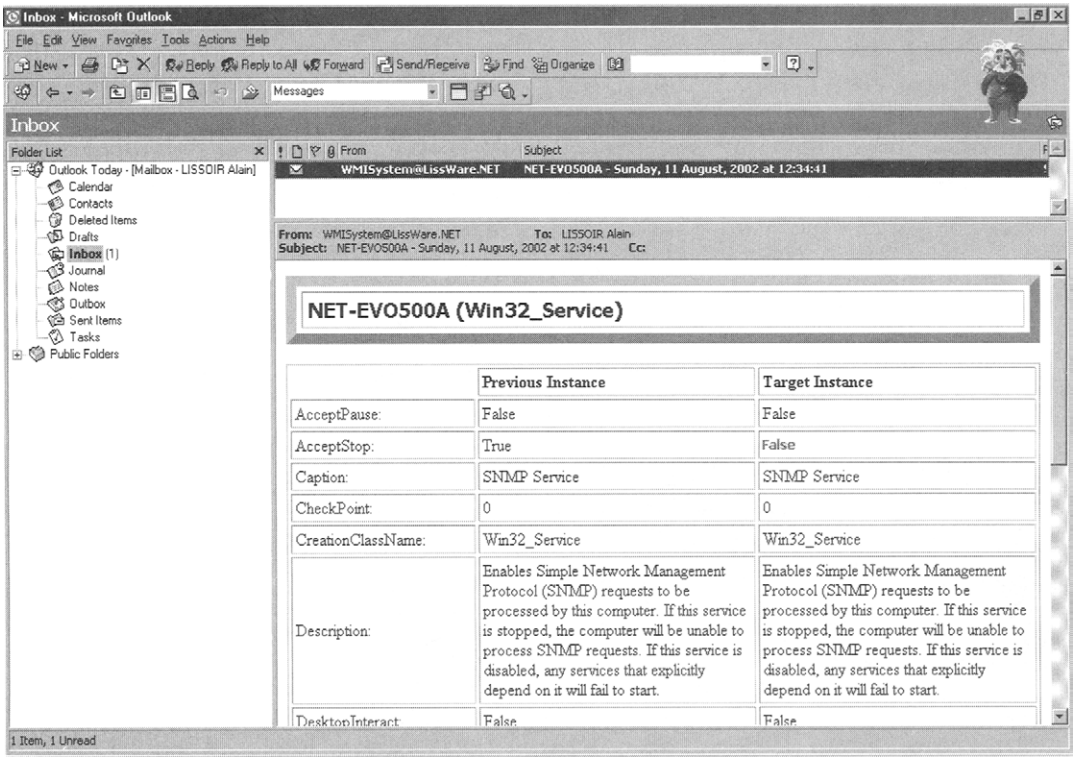

Figure 6.22 *The HTML e-mail viewed in Outlook.*

6.6.2 Calculating delay between two events

As we have seen throughout this chapter, when something occurs in a system, it is possible to receive a notification. But in some situations, it would be interesting to receive an alert when an executed operation takes longer than expected. For instance, if a server is executing some jobs in the background, there is no problem if the job takes 5 or 10 minutes. However, if the job takes more than 10 minutes, it would be nice to receive an alert. Two approaches can be taken here:

1. **Send an alert as soon as the job is completed and only if the execution time is longer than expected.** In this case, we wait for the job to complete and alert the administrator if it took longer than desired. This supposes that the job completes.
2. **Send an alert once the expected delay expires regardless of whether the job has completed.** In this case, we don't wait for the job to complete; the alert is sent as soon as the expected delay expires.

Although both methods are valid, they don't address the same type of problem. In some situations, the first approach would be good enough, and for some others, the second approach would be more suitable. As it is the easiest one, let's start with the first approach. In the next section, we illustrate the second approach by using the elements developed for the first.

A nice case of application for the first approach is related to the Knowledge Consistency Checker topology calculation time. When Active Directory is up and running, it starts a process called the Knowledge Consistency Checker (KCC) every 15 minutes. Among other things, this process calculates where connection objects that establish the link between the different DCs must be created. This allows Active Directory to replicate between DCs while respecting the rules defined by the architect. During its calculation, the KCC takes into consideration a certain number of elements defined by the architect, such as the DC locations, the number of domains, the number of sites created, the site links created, etc. In a large enterprise, when Active Directory deployment starts, the KCC calculation time is never an issue as the number of DCs and sites is quite small. If Active Directory design plans to have a huge number of Sites (more than 200 or more than 500 based on some design assumptions), the topology in place becomes bigger and bigger as the deployment moves forward. At a certain point, it is likely that the KCC calculation time will be an issue since it uses a lot of CPU time (close to 100 percent) during the calculation period. Although, some actions can be taken to avoid such a situation (mostly dur-

ing Active Directory design phase), there are always practical limits. Active Directory design is a complex matter and is not covered in this book. For more information on this subject, refer to *Mission-Critical Active Directory* by Mickey Balladelli and Jan de Clercq (ISBN 1555582400).

Back to our initial question: How to monitor the KCC calculation time? The KCC is a process, and a first solution could be the use of WMI to monitor the CPU utilization time made by this process. However, there is an easier solution. By changing a registry key in each Active Directory domain controller, it is possible to get an NT Event Log trace of the KCC's starting time and ending time. The timestamp difference between the two traces will determine the KCC calculation time. For this, the following system registry entry must be set to a value of 3:

Key:	HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\NTDS\Diagnostics
Value Name:	1 Knowledge Consistency Checker
Value Data:	KCC diagnostic levels.
Default:	0

Now, each time the KCC starts and stops its topology calculation, the NT Event Log will at least contain the two messages shown in Figure 6.23.

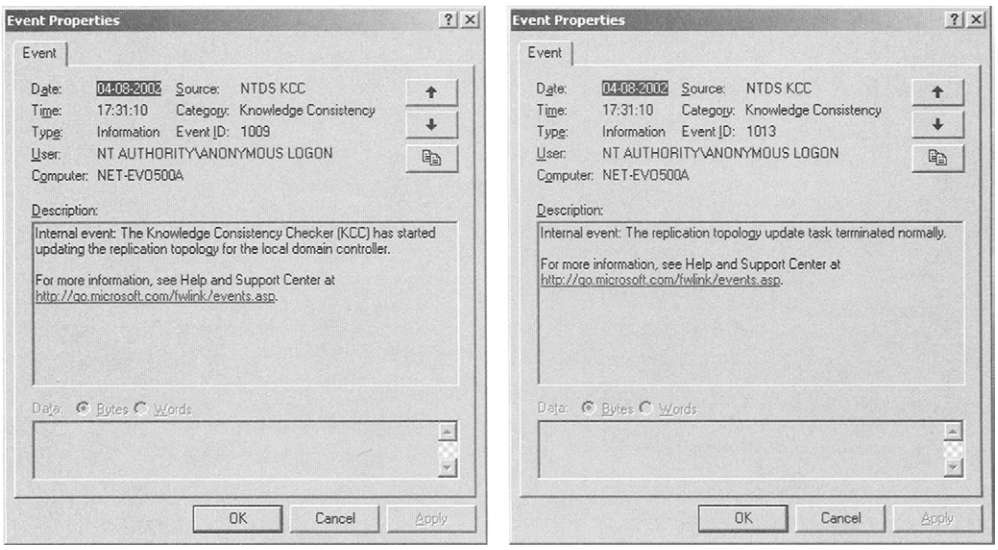

Figure 6.23 *The KCC start and stop events.*

The problem can be summarized as detecting the presence of two NT Event Log messages (1009 and 1013). When the second message appears (1013), the script logic calculates the amount of time between the two events. In order to make the script reusable in other circumstances, the event source name, the starting event code, the ending event code, and the maximum amount of time between the two events will be exposed as command-line parameters. If the amount of time is longer than expected, the script sends an e-mail alert as in the previous sample. Instead of generating an HTML stream for the e-mail body with the GenerateHTML() function seen before, the script will use the WMI XML representation discussed in the previous chapter. Sample 6.22 implements this logic in JScript.

Sample 6.22 *Sending alert when delay between two events reaches a fixed limit (Part I)*

```
1:<?xml version="1.0"?>
.:
8:<package>
9:  <job>
..:
13:    <runtime>
14:      <named name="SourceName" helpstring="Source event name. (i.e., NTDS KCC)"
                                  required="true" type="string"/>
15:      <named name="EventCodeBegin" helpstring="Event code begining. (i.e., 1009)"
                                  required="true" type="string"/>
16:      <named name="EventCodeEnd" helpstring="Event code ending. (i.e., 1013)"
                                  required="true" type="string"/>
17:      <named name="MaxDelayBetweenEvents" helpstring="Maximum delay between Begin event and
                                  Ending event. (in seconds)" required="true" type="string"/>
18:      <named name="Machine" helpstring="determine the WMI system to connect to.
                                  (default=LocalHost)" required="false" type="string"/>
19:      <named name="User" helpstring="determine the UserID to perform the remote connection.
                                  (default=none)" required="false" type="string"/>
20:      <named name="Password" helpstring="determine the password to perform the remote
                                  connection. (default=none)" required="false" type="string"/>
21:      <named name="NameSpace" helpstring="determine the WMI namespace to connect to.
                                  (default=Root\CIMv2)" required="false" type="string"/>
22:    </runtime>
23:
24:    <script language="VBScript" src="..\Functions\TinyErrorHandler.vbs" />
25:    <script language="VBScript" src="..\Functions\PauseScript.vbs" />
26:    <script language="VBScript" src="..\Functions\SendMessageExtendedFunction.vbs" />
27:    <script language="VBScript" src="..\Functions\SendAlertFunction.vbs" />
28:
29:    <object progid="WbemScripting.SWbemLocator" id="objWMILocator" reference="true"/>
30:    <object progid="WbemScripting.SWbemDateTime" id="objWMIDateTime" />
31:    <object progid="Microsoft.XMLDom" id="objXML" />
32:    <object progid="Microsoft.XMLDom" id="objXSL" />
33:
34:    <script language="Jscript">
35:    <![CDATA[
36:
37:    // ------------------------------------------------------------------------------
38:    var cComputerName = "LocalHost"
39:    var cWMINameSpace = "root/cimv2"
```

```
 40:
 41:    var cWMIQuery = "Select * from __InstanceCreationEvent
                                          Where TargetInstance ISA 'Win32_NTLogEvent'"
 42:
 43:    var cXSLFile = "PathLevel0Win32_NTLogEvent.XSL"
 44:
 45:    var cTargetRecipient = Alain.Lissoir@LissWare.NET
 46:    var cSourceRecipient = WMISystem@LissWare.NET
 47:
 48:    var cSMTPServer = "relay.LissWare.NET "
 ..
 76:    // ---------------------------------------------------------------------------
 77:    // Parse the command line parameters
 78:    if (WScript.Arguments.named.Count < 4)
 79:       {
 80:       WScript.Arguments.ShowUsage();
 81:       WScript.Quit();
 82:       }
 83:
 84:    strSourceName = WScript.Arguments.Named("SourceName");
 85:    intEventCodeBegin = new Number (WScript.Arguments.Named("EventCodeBegin")).valueOf();
 86:    intEventCodeEnd =
              new Number (WScript.Arguments.Named("EventCodeEnd")).valueOf();
 87:    intMaxDelayBetweenEvents =
              new Number (WScript.Arguments.Named("MaxDelayBetweenEvents")).valueOf();
 88:    strUserID = WScript.Arguments.Named("User");
...:
106:    strWMINameSpace = WScript.Arguments.Named("NameSpace");
...:
111:    strWMINameSpace = strWMINameSpace.toUpperCase();
112:
113:    objWMIEventSink = WScript.CreateObject ("WbemScripting.SWbemSink", "EventSINK_");
114:
115:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate;
116:    objWMILocator.Security_.Privileges.AddAsString ("SeSecurityPrivilege", true);
117:    try
118:      {
119:      objWMIServices = objWMILocator.ConnectServer(strComputerName, strWMINameSpace,
120:                                                   strUserID, strPassword);
121:      }
122:    catch (Err)
123:      {
124:      ErrorHandler (Err);
125:      }
126:
127:    strWMIQuery = cWMIQuery +
128:                  " And TargetInstance.SourceName='" + strSourceName + "'" +
129:                  " And (TargetInstance.EventCode='" + intEventCodeBegin + "' Or " +
130:                  "TargetInstance.EventCode='" + intEventCodeEnd + "')";
131:
132:    WScript.Echo ("Creating subscription to monitor delay between event log events '" +
133:                  intEventCodeBegin + "' and '"  + intEventCodeEnd + "' for event source '" +
134:                  strSourceName + "'.");
135:    WScript.Echo ();
136:
137:    try
138:      {
139:      objWMIServices.ExecNotificationQueryAsync (objWMIEventSink, strWMIQuery);
140:      }
```

```
141:    catch (Err)
142:      {
143:      ErrorHandler (Err);
144:      }
145:
146:    WScript.Echo ("Waiting for events...");
147:
148:    PauseScript ("Click on 'Ok' to terminate the script ...");
149:
150:    WScript.Echo ();
151:    WScript.Echo ("Cancelling event subscription ...");
152:    objWMIEventSink.cancel();
153:
154:    WScript.Echo ("Finished.");
...:
```

As usual, the first part of the script is the initialization part. This part respects a structure used along all samples we have seen before. Lines 13 to 22 contain the XML structure that defines the command-line parameters. Lines 24 to 27 include some external functions and define some constants used later in the script. Next, the script parses the command line parameters (lines 78 to 111) and creates the WMI connection (lines 115 and 125). Once the WQL event query constructed with the information coming from the command line (lines 127 to 130), the script processes the WMI event subscription (lines 137 and 144). If the command line used is

```
C:\>EventLogTimeDiffMonitor.wsf /SourceName:"NTDS KCC"
                                /EventCodeBegin:1009
                                /EventCodeEnd:1013
                                /MaxDelayBetweenEvents:600
```

The produced WQL event query at line 127 is

```
Select * from __InstanceCreationEvent Where
                        TargetInstance ISA 'Win32_NTLogEvent' And
                        TargetInstance.SourceName='NTDS KCC' And
                        (TargetInstance.EventCode='1009' Or TargetInstance.EventCode='1013')
```

Notice that there is no **WITHIN** statement because the *Win32_NTLogEvent* class is provided by the *NT Event Log* provider, which is an event provider. Next, the WQL event query ensures that the event source is the KCC and that only event code numbers 1009 and 1013 can trigger a WMI event. This means that we have one WQL query for both NT Event Log messages. As with Sample 6.19, many different approaches can be taken. It is also possible to perform a subscription for each message. In this case, it means that we create two subscriptions. Although technically valid, it is easier to combine the events in the same WQL event query as they relate to the same intrinsic event type. Moreover, this uses fewer system resources. If we had a situation in which two different types of event must be monitored (e.g., creation and modification of instances), then it is neces-

sary to create two different subscriptions. Once the WQL query is submitted, the script pauses at line 148.

When the WMI event matching the WQL event query occurs, the OnObjectReady sink routine is called. This sink routine is available in Sample 6.23.

Sample 6.23 *Sending alert when delay between two events reaches a fixed limit (Part II)*

```
...:
156:    // ---------------------------------------------------------------------------------------
157:    function EventSINK_OnObjectReady (objWbemObject, objWbemAsyncContext)
158:        {
...:
166:        WScript.Echo ();
167:        WScript.Echo ("EventBEGIN - OnObjectReady.");
168:        WScript.Echo (objDate.toLocaleDateString() + " at " +
169:                      objDate.toLocaleTimeString() + ": '" +
170:                      objWbemObject.Path_.Class + "' has been triggered.");
171:
172:        switch (objWbemObject.Path_.Class)
173:              {
174:              case "__InstanceCreationEvent":
175:                   objWMIInstance = objWbemObject.TargetInstance;
176:                   break;
177:              default:
178:                   objWMIInstance = null;
179:                   break;
180:              }
181:
182:        if (objWMIInstance != null)
183:           {
184:           objWMIDateTime.Value = objWMIInstance.TimeGenerated;
185:
186:           switch (objWMIInstance.EventCode)
187:                 {
188:                 case intEventCodeBegin:
189:                      strBeginEventCodeTime = objWMIDateTime.GetVarDate (false);
190:                      WScript.Echo ("Saving '" + strSourceName + "' event '" +
191:                                    intEventCodeBegin +
192:                                    "' startup time (" + strBeginEventCodeTime + ").");
193:                      break;
194:                 case intEventCodeEnd:
195:                      if (strBeginEventCodeTime != null)
196:                         {
197:                         strStopEventCodeTime = objWMIDateTime.GetVarDate (false);
198:                         WScript.Echo ("Saving '" + strSourceName + "' event '" +
199:                                       intEventCodeEnd +
200:                                       "' ending time (" + strStopEventCodeTime + ").");
201:
202:                         objBeginEventCodeTime = new Date(strBeginEventCodeTime);
203:                         objStopEventCodeTime = new Date (strStopEventCodeTime);
204:
205:                         intDelayBetweenEvents =
                                    (objStopEventCodeTime-objBeginEventCodeTime)/1000;
206:                         strBeginEventCodeTime = null;
```

```
207:
208:                          WScript.Echo ("Delay between events is " +
209:                                        intDelayBetweenEvents + " seconds.");
210:
211:                          if (intDelayBetweenEvents >= intMaxDelayBetweenEvents)
212:                             {
213:                             SendAlert (objWMIInstance,
214:                                        "'" + strSourceName + "' event interval (" +
215:                                            intEventCodeBegin + "/" + intEventCodeEnd +
216:                                            ") is " + intDelayBetweenEvents + " seconds. " +
217:                                            objDate.toLocaleDateString() + " at " +
218:                                            objDate.toLocaleTimeString());
219:                             }
220:                          }
221:                       break;
222:                   }
223:           }
224:
225:        WScript.Echo ("EventEND - OnObjectReady.");
226:
227:        }
228:
229:    ]]>
230:   </script>
231:  </job>
232:</package>
```

This sink routine is a bit more complex than previous sink routines. Although we do not use the notion of WMI context during the submissions (see Sample 6.22, line 139), there is a kind of context in presence. Why? The script must behave differently depending on whether the event received is the starting event or the ending event. For this reason the majority of the sink code is embraced in a **Switch Case** statement (lines 186 to 223). The **Switch Case** statement considers two cases: one for the starting event (lines 188 to 193) and one for the ending event (lines 194 to 221).

Let's start with the first event—the starting event. As soon as an event occurs, the sink routine saves the time when the event was generated in an **SWbemDateTime** object (line 184). Note that the script takes the time when the event is generated (line 184) and not the time when the WMI event is triggered in the script (line 169). This is important because if WMI has a lot of events in the queue, it is likely that the time when the event is submitted to the sink will be delayed in relation to the time when the event occurred. Because we are in the starting event case, this time is saved in a variable for further use (line 189). Once this is completed, the sink routine terminates normally.

After a while (from a few seconds up to a few minutes or hours), the ending event occurs. At this time, the sink routine branches to the other case of the **Switch Case** (lines 194 to 221).

This case is a bit more complex. First, the sink routine verifies whether the variable containing the starting time has been initialized (line 195). If not, this means that there was no starting event or the script missed the starting event (i.e., the script was started after the starting event). In such a case, there is no way to calculate the period of time between the two events. The triggered ending event is skipped. If the starting time is initialized, the ending time is captured with the help of an **SWbemDateTime** object and stored in a variable (line 197). Next, the sink routine calculates the difference between the two times in seconds with the help of two JScript **Date** objects (lines 202 to 205). This is where the saved time during the starting event is reused (line 202). Once the calculation completes, the starting time variable is reset (line 206).

The amount of time between the two events is evaluated against the maximum time expected (line 211). If the time is smaller than the maximum, the sink routine terminates silently. If the time is greater than or equal to the maximum, some more processing is made to send an alert. The alert creation is written with a VBScript function available in Sample 6.24.

Sample 6.24 *Sending alert when delay between two events reaches a fixed limit (Part III)*

```
 .:
 .:
 .:
 6:' ------------------------------------------------------------------------------
 7:
 8:Function SendAlert (objWMIInstance, strTitle)
..:
17:    Set objWMINamedValueSet = CreateObject ("WbemScripting.SWbemNamedValueSet")
18:
19:    objWMINamedValueSet.Add "LocalOnly", False
20:    objWMINamedValueSet.Add "PathLevel", 0
21:    objWMINamedValueSet.Add "IncludeQualifiers", False
22:    objWMINamedValueSet.Add "ExcludeSystemProperties", True
23:
24:    strXMLText = objWMIInstance.GetText_(wbemObjectTextFormatWMIDTD20, _
25:                                         , _
26:                                         objWMINamedValueSet)
..:
30:    objXML.Async = False
31:    objXML.LoadXML strXMLText
32:
33:    objXSL.Async = False
34:    objXSL.Load (cXSLFile)
35:
36:    strHTMLText = objXML.TransformNode(objXSL)
37:    WScript.Echo "Sending an email alert."
38:
39:    If SendMessage (cTargetRecipient, _
40:                    cSourceRecipient, _
41:                    strTitle, _
```

```
42:                     strHTMLText, _
43:                     "") Then
44:       WScript.Echo "Failed to send email alert to '" & cTargetRecipient & "' ..."
45:    End If
46:
47:End Function
```

The script sends an e-mail alert (lines 39 to 43). In the previous sample, we used a VBScript function to generate the HTML output to send the mail. However, in the previous chapter, we saw that it is possible to obtain an XML representation of a **SWbemObject** by invoking its *GetText_* method, which is precisely what this function is doing (only supported under Windows XP and Windows.NET Server). With the help of the **SWbemNamedValueSet** object, the script passes the parameters desired for the XML generation (lines 19 to 22). Next, the XML representation is retrieved (lines 24 to 26). Instead of using a VBScript function to generate the HTML output, the sink routine takes advantage of the XML features offered by WMI. By referencing the XSL file defined at line 43 in Sample 6.22 and using some object methods provided by the Microsoft DOMXML object model, it is possible to generate the desired HTML output (line 36). Of course, an XSL file must be created for this purpose. Here is the content of the XSL file:

```
1:<?xml version='1.0'?>
2:
3:<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
4: <xsl:template match="/">
5:
6:  <html>
7:   <TABLE ID="TBL" BORDER="10" WIDTH="100%" CELLSPACING="5" CELLPADDING="5">
8:    <TR>
9:     <TD>
10:      <B>
11:       <FONT color="black">
12:        <SPAN style="COLOR: black; FONT-FAMILY: Tahoma; FONT-SIZE: 14pt">
13:         <xsl:value-of select="/INSTANCE/PROPERTY[@NAME = 'ComputerName']/VALUE"/>
14:         (<xsl:value-of select="/INSTANCE/@CLASSNAME"/>)
15:        </SPAN>
16:       </FONT
17:      </B>
18:     </TD>
19:    </TR>
20:   </TABLE>
21:
22:   <BR></BR>
23:   <TABLE ID="TBL" BORDER="1" WIDTH="100%" CELLSPACING="2" CELLPADDING="2">
24:    <TR>
25:    <TD>
26:     <b>Properties</b>
27:    </TD>
28:    <TD>
29:     <b>Values</b>
30:    </TD>
```

```
31:    </TR>
32:    <xsl:for-each select="/INSTANCE/PROPERTY">
33:     <TR>
34:      <TD>
35:       <xsl:value-of select="@NAME"/>
36:       (<xsl:value-of select="@TYPE"/>)
37:      </TD>
38:      <TD>
39:       <xsl:value-of select="VALUE"/>
40:      </TD>
41:     </TR>
42:    </xsl:for-each>
43:   </TABLE>
44:
45:   <FONT color="red">
46:    <SPAN style="COLOR:red; FONT-FAMILY: Tahoma; FONT-SIZE: 10pt; FONT-WEIGHT: bold">
47:
48:    </SPAN>
49:   </FONT>
50:
51:   <FONT color="black">
52:    <SPAN style="COLOR:black; FONT-FAMILY: Tahoma; FONT-SIZE: 10pt">
53:
54:    </SPAN>
55:   </FONT>
56:
57:  </html>
58:
59: </xsl:template>
60:</xsl:stylesheet>
```

For the sake of completeness, the XML representation of an instance delivered during the ending event is as follows:

```
 1:<INSTANCE CLASSNAME="Win32_NTLogEvent">
 2: <PROPERTY NAME="Category" CLASSORIGIN="Win32_NTLogEvent" TYPE="uint16">
 3:  <VALUE>1</VALUE>
 4: </PROPERTY>
 5: <PROPERTY NAME="CategoryString" CLASSORIGIN="Win32_NTLogEvent" TYPE="string">
 6:  <VALUE>Knowledge Consistency Checker</VALUE>
 7: </PROPERTY>
 8: <PROPERTY NAME="ComputerName" CLASSORIGIN="Win32_NTLogEvent" TYPE="string">
 9:  <VALUE>XP-DPEN6400</VALUE>
10: </PROPERTY>
11: <PROPERTY.ARRAY NAME="Data" CLASSORIGIN="Win32_NTLogEvent" PROPAGATED="true" TYPE="uint8">
12: </PROPERTY.ARRAY>
13: <PROPERTY NAME="EventCode" CLASSORIGIN="Win32_NTLogEvent" TYPE="uint16">
14:  <VALUE>1013</VALUE>
15: </PROPERTY>
..:
..:
..:
33: <PROPERTY NAME="SourceName" CLASSORIGIN="Win32_NTLogEvent" TYPE="string">
34:  <VALUE>NTDS KCC</VALUE>
35: </PROPERTY>
36: <PROPERTY NAME="TimeGenerated" CLASSORIGIN="Win32_NTLogEvent" TYPE="datetime">
37:  <VALUE>20020623103934.000000+120</VALUE>
38: </PROPERTY>
```

```
39: <PROPERTY NAME="TimeWritten" CLASSORIGIN="Win32_NTLogEvent" TYPE="datetime">
40:  <VALUE>20020623103934.000000+120</VALUE>
41: </PROPERTY>
42: <PROPERTY NAME="Type" CLASSORIGIN="Win32_NTLogEvent" TYPE="string">
43:  <VALUE>information</VALUE>
44: </PROPERTY>
45: <PROPERTY NAME="User" CLASSORIGIN="Win32_NTLogEvent" TYPE="string">
46:  <VALUE>NT AUTHORITY\ANONYMOUS LOGON</VALUE>
47: </PROPERTY>
48:</INSTANCE>
```

The script loads the XML and XSL files in two DOM XML objects created in the XML header of the script (lines 30 and 31 in Sample 6.22). Once both XML files are loaded (lines 31 and 34 in Sample 6.24), the sink routine performs the XSL transformation to obtain the HTML representation (line 36). Next, the sink routine sends the mail as seen before in Sample 6.21. The mail received is almost the same as the one shown in Figure 6.22, but the output is adapted to match the requirements of an NT Event Log entry instance.

6.6.3 Calculating delay between one event and one nonevent

The previous script works perfectly as long as the ending event occurs. But what happens if the ending event never occurs? The answer is quite easy: The script will never send an alert. This can be unfortunate in some situations. In the previous scenario, if the process takes an extremely long time to complete, it is clear that the alert will be sent, but will it be too late? Moreover, if the process crashes after the starting event, the alert will never be sent, which is worse! This is why the script must also be able to support situations where the ending event doesn't occur in an expected delay. If we combine the previous script with some of the features we learned about the timer events (see Sample 6.2), it is possible to solve this problem using the same command-line parameters. This logic is implemented in Sample 6.25 and reuses, as a base, the code developed in the previous section (Samples 6.22 to 6.24).

Sample 6.25 *Sending an alert if the expired time from one event reaches a fixed limit (Part I)*

```
...:
...:
...:
119:    objWMIEventSink = WScript.CreateObject ("WbemScripting.SWbemSink", "EventSINK_");
120:    objWMINonEventSink = WScript.CreateObject ("WbemScripting.SWbemSink", "NonEventSINK_");
121:
122:    objWMILocator.Security_.ImpersonationLevel = wbemImpersonationLevelImpersonate;
123:    objWMILocator.Security_.Privileges.AddAsString ("SeSecurityPrivilege", true);
```

```
124:    try
125:      {
126:      objWMIServices = objWMILocator.ConnectServer(strComputerName, strWMINameSpace,
127:                                                   strUserID, strPassword);
128:      }
129:    catch (Err)
130:      {
131:      ErrorHandler (Err);
132:      }
133:
134:    strWMIQuery = cWMIQuery +
135:                  " And TargetInstance.SourceName='" + strSourceName + "'" +
136:                  " And (TargetInstance.EventCode='" + intEventCodeBegin + "' Or " +
137:                  "TargetInstance.EventCode='" + intEventCodeEnd + "')";
138:
139:    WScript.Echo ("Creating subscription to monitor delay between event log events '" +
140:                  intEventCodeBegin + "' and '"  + intEventCodeEnd + "' for event source '" +
141:                  strSourceName + "'.");
142:    WScript.Echo ();
143:
144:    try
145:      {
146:      objWMIServices.ExecNotificationQueryAsync (objWMIEventSink, strWMIQuery);
147:      }
148:    catch (Err)
149:      {
150:      ErrorHandler (Err);
151:      }
152:
153:    WScript.Echo ("Waiting for events...");
154:
155:    PauseScript ("Click on 'Ok' to terminate the script ...");
156:
157:    WScript.Echo ();
158:    WScript.Echo ("Cancelling event subscription ...");
159:    objWMINonEventSink.cancel();
160:    objWMIEventSink.cancel();
161:
162:    WScript.Echo ("Finished.");
...:
...:
...:
```

The startup section of the script is the same as before. The only addition is the creation of two **SWbemSink** objects instead of one. Here, we use two sink routines: a first sink routine to catch the starting and ending events as before (line 119) and a second sink routine to catch a timeout event (line 120). The sink routine catching the starting and ending events is referenced as before during the WQL query submission (line 146). The second **SWbemSink** object is used during the execution of the sink routine catching the starting and ending events. As it is the only change in the script startup section, let's take a look at the new sink routines. These routines are shown in Sample 6.26.

Sample 6.26 *Sending an alert if the expired time from one event reaches a fixed limit (Part II)*

```
...:
...:
...:
164:    // ---------------------------------------------------------------------------------------
165:    function EventSINK_OnObjectReady (objWbemObject, objWbemAsyncContext)
166:        {
...:
176:
177:        WScript.Echo ();
178:        WScript.Echo ("EventBEGIN - OnObjectReady.");
179:        WScript.Echo (objDate.toLocaleDateString() + " at " +
180:                      objDate.toLocaleTimeString() + ": '" +
181:                      objWbemObject.Path_.Class + "' has been triggered.");
182:
183:        switch (objWbemObject.Path_.Class)
184:              {
185:              case "__InstanceCreationEvent":
186:                   objWMIInstance = objWbemObject.TargetInstance;
187:                   break;
188:              default:
189:                   objWMIInstance = null;
190:                   break;
191:              }
192:
193:        if (objWMIInstance != null)
194:           {
195:           objWMIDateTime.Value = objWMIInstance.TimeGenerated;
196:
197:           switch (objWMIInstance.EventCode)
198:                 {
199:                 case intEventCodeBegin:
200:                      objWMIBeginEventInstance = objWMIInstance;
201:                      strBeginEventCodeTime = objWMIDateTime.GetVarDate (false);
202:                      WScript.Echo ("Saving '" + strSourceName + "' event '" +
203:                                    intEventCodeBegin +
204:                                    "' startup time (" + strBeginEventCodeTime + ").");
205:
206:                      objWMINonEventClass = objWMIServices.Get (cWMIClass);
207:
208:                      objWMINonEventInstance = objWMINonEventClass.SpawnInstance_();
209:                      objWMINonEventInstance.TimerID = cTimerID;
210:                      objWMINonEventInstance.SkipifPassed = true;
211:
212:                      objWMINonEventInstance.IntervalBetweenEvents =
                                                  (intMaxDelayBetweenEvents) * 1000;
213:                      objWMINonEventInstance.Put_
                             (wbemChangeFlagCreateOrUpdate | wbemFlagReturnWhenComplete);
214:
215:                      WScript.Echo ("'" + cTimerID + "' instance created.");
216:
217:                      objWMIServices.ExecNotificationQueryAsync (objWMINonEventSink,
218:                                                                 cWMINonEventQuery);
219:                      break;
220:                 case intEventCodeEnd:
221:                      if (strBeginEventCodeTime != null)
222:
```

```
223:                        strStopEventCodeTime = objWMIDateTime.GetVarDate (false);
224:                        WScript.Echo ("Saving '" + strSourceName + "' event '" +
225:                                    intEventCodeEnd +
226:                                    "' ending time (" + strStopEventCodeTime + ").");
227:
228:                        objBeginEventCodeTime = new Date(strBeginEventCodeTime);
229:                        objStopEventCodeTime = new Date (strStopEventCodeTime);
230:
231:                        intDelayBetweenEvents =
                                  (objStopEventCodeTime - objBeginEventCodeTime) / 1000;
232:                        strBeginEventCodeTime = null;
233:
234:                        WScript.Echo ("Delay between events is " +
235:                                    intDelayBetweenEvents + " seconds.");
236:
237:                        if (intDelayBetweenEvents >= intMaxDelayBetweenEvents)
238:                           {
239:                           SendAlert (objWMIInstance,
240:                                    "'" + strSourceName + "' event interval (" +
241:                                       intEventCodeBegin + "/" + intEventCodeEnd +
242:                                       ") is " + intDelayBetweenEvents + " seconds. " +
243:                                       objDate.toLocaleDateString() + " at " +
244:                                       objDate.toLocaleTimeString());
245:                           }
246:                        else
247:                           {
248:                           try
249:                             {
250:                             objWMINonEventInstance = objWMIServices.Get
                                                      (cWMIClass + "='" + cTimerID + "'");
251:                             objWMINonEventSink.Cancel();
252:                             objWMINonEventInstance.Delete_();
253:                             WScript.Echo ("'" + cTimerID + "' instance deleted.");
254:                             }
255:                           catch(Err) {}
256:                           }
257:                        }
258:                     break;
259:               }
260:          }
261:
262:       WScript.Echo ("EventEND - OnObjectReady.");
263:
264:       }
...:
...:
...:
```

The EventSINK_OnObjectReady() sink routine receiving the starting and ending events is exactly the same as the one presented in Sample 6.23. All previous comments about this sink remain valid. Of course, to address the nonevent problem described, some extra logic is added. This code is represented in boldface in lines 206 to 218 and lines 246 to 256. This extra logic is executed in two phases. The first phase executes when the starting event occurs (lines 199 to 219). This phase starts by saving the instance representing the starting event (line 200). Previously, only the event generation

time was saved (line 195 and 201), but now, because we may not have an ending event, it is useful to show the instance corresponding to the starting event when the expected period of time for the ending event has expired. This starting instance is used during the execution of the sink corresponding to the timeout event.

Next, the EventSINK_OnObjectReady() sink during a starting event creates a *__IntervalTimerInstruction* instance (lines 206 to 215). This piece of code simply reuses the code we discovered when examining the interval timer events. The code comes from Sample 6.2 (lines 35 to 40). Once the *__IntervalTimerInstruction* instance is created, the script proceeds to an asynchronous subscription for this interval timer event and references the second **SWbemSINK** object (line 217 and 218) created during the startup phase of the script (line 120). This sends all interval timer events to the dedicated timer sink. Note that this timer event is using a pulse corresponding to the maximum allowed time specified on the command line (line 212).

Now, the script is registered to receive two event types to two different sinks: (1) an event type corresponding to any starting or ending events and redirected to the EventSINK_OnObjectReady() sink and (2) an event type corresponding to the timer event and redirected to the NonEventSINK_OnObjectReady() sink. In such a situation, two cases must be considered:

1. **The ending event occurs before the timeout.** This situation corresponds to the second phase of the EventSINK_OnObjectReady() execution. This is the phase corresponding to the ending event. In the code, it corresponds to the second part of the **Switch Case** (lines 220 to 258 in Sample 6.26). This part of the script executes the same logic as that used in Sample 6.23 (lines 195 to 219). The extra code added to this part of the **Switch Case** is visible in boldface in Sample 6.26 (lines 246 to 256). Because the ending event occurs before the timer event, it is not necessary to keep the timer event running, as it is created to alert in case of nonevent of the ending event. This is why the lines 246 to 256 delete the created timer event. These lines of code are extracted from Sample 6.3 (lines 31 to 33) when we saw how to proceed to the deletion of an interval timer event.

   You can see that the ending event case continues to compare the expired time with the maximum allowed time (line 237). This allows the script to send an alert when the ending event occurs after the timeout.

2. **The timeout occurs before the ending event.** This situation occurs only when the ending event is not raised in the expected delay specified on the command line. In this case, the second sink, NonEventSINK_OnObjectReady(), is called (see Sample 6.27). Because the interval timer event has been triggered, it is no longer necessary to execute it, and the script deletes the interval timer event created during the starting event phase (lines 278 to 284). Because the interval timer occurred, it means that the maximum allowed time for the ending event has expired. In this case, the script sends an alert with the help of the SendAlert() function (lines 287 to 291). Notice that the SendAlert() function uses the starting instance as parameter. Doing so, the operator receives the timeout message with a view of the corresponding starting instance. When the ending event occurs, the EventSINK_OnOjbectReady() sink is invoked. Because the ending event is out of delay, the conditions at line 237 (see Sample 6.26) are True, and the script sends a second alert containing the ending event instance.

Sample 6.27 *Sending an alert if the expired time from one event reaches a fixed limit (Part III)*

```
...:
...:
...:
266:    // ---------------------------------------------------------------------------
267:    function NonEventSINK_OnObjectReady (objWbemObject, objWbemAsyncContext)
268:        {
...:
272:        WScript.Echo ();
273:        WScript.Echo ("NonEventBEGIN - OnObjectReady.");
274:        WScript.Echo (objDate.toLocaleDateString() + " at " +
275:                      objDate.toLocaleTimeString() + ": '" +
276:                      objWbemObject.Path_.Class + "' has been triggered.");
277:
278:        try
279:          {
280:          objWMINonEventInstance = objWMIServices.Get(cWMIClass + "='" + cTimerID + "'");
281:          objWMINonEventSink.Cancel();
282:          objWMINonEventInstance.Delete_();
283:          WScript.Echo ("'" + cTimerID + "' instance deleted.");
284:          }
285:        catch (Err) {}
286:
287:        SendAlert (objWMIBeginEventInstance,
288:                   "'" + strSourceName + "' event '" + intEventCodeEnd +
289:                       "' didn't occur within " + intMaxDelayBetweenEvents + " seconds. " +
290:                       objDate.toLocaleDateString() + " at " +
291:                       objDate.toLocaleTimeString());
292:
293:        WScript.Echo ("NonEventEND - OnObjectReady.");
```

```
294:
295:        }
296:
297:    ]]>
298:   </script>
299:  </job>
300:</package>
```

This script (from Samples 6.25 to 6.27) shows how the interaction of an intrinsic event (*__InstanceCreationEvent*) and an interval timer event (*__IntervalTimerInstruction*) can be combined to implement a powerful monitoring. In the code, we use only the *Win32_NTLogEvent* class, but this logic can be easily adapted to any class available in WMI. However, the properties and the conditions to trigger the desired event must be adapted.

6.7 Summary

This chapter examined WMI events by explaining the different event types available from WMI and the type of system classes they use. Beside the WMI scripting API used in a script managing WMI events, WMI events can be received only by formulating the right WQL event query. This is why the knowledge we acquired about WQL in Chapter 3 is very important. Without being able to formulate a WQL event query, it is impossible to work with the WMI events. In this chapter, we also saw that the event providers and the classes that they implement have an influence at three levels:

1. The way the WQL query is formulated
2. The retrieved properties from the event
3. The instance subject to the event

6.8 What's next?

A good knowledge of the provider capabilities is key in retrieving the expected information. Up to now, we have worked with a very small number of providers. Because the WMI scripting capabilities are closely related to the WMI providers' capabilities, it is important to discover the WMI providers available under Windows.NET Server and their capabilities. Gathering such knowledge enhances our scripting capabilities. This is the purpose of the second book, *Leveraging Windows Management Instrumentation (WMI) Scripting* (ISBN 1555582990).

6.9 Useful Internet URLs

Patch for Windows 2000 SP2: Q306274 (Corrected in SP3): WMI Inadvertently Cancels Intrinsic Event Queries:

```
http://support.microsoft.com/default.aspx?scid=kb;en-us;Q306274
```

```
http://www.microsoft.com/downloads/release.asp?releaseid=37814
```