.NET System Management Services

ALEXANDER GOLOMSHTOK

Apress™

`.NET System Management Services`

ISBN (pbk): 1-59059-058-9

Printed and bound in the United States of America 12345678910

Technical Reviewer: Yefim Nodelman
Editorial Directors: Dan Appleman, Gary Cornell, Jason Gilmore, Simon Hayes, Martin Streicher, Karen Watterson, John Zukowski
Managing Editor: Grace Wong
Project Manager: Nicole LeClerc
Copy Editor: Rebecca Rider
Compositor: Impressions Book and Journal Services, Inc.
Indexer: Valerie Perry
Cover Designer and Illustrator: Kurt Krames
Production Manager: Kari Brooks
Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`.

Outside the United States, fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

CHAPTER 1

.NET Framework and Windows Management Instrumentation

Over the last two decades, the remarkable evolution of the computer and networking technologies has changed our perception of computing forever. The days of monolithic mainframe installations and running operating systems, utilities, and applications from a single vendor are long gone, and simple, straightforward system architectures were forgotten long ago. The rapidly growing popularity of the distributed computing model has turned the vast majority of computing infrastructures into an extremely complex mix of dissimilar hardware devices that are interconnected by spiderwebs of local and wide area networks and are running software from thousands of different vendors. Besides elevating the complexity of the computer installations to a brand new level, this overwhelming technological progress has materialized the most horrible nightmares for many system administrators by making computer and network management painfully challenging.

The issue of system management is quite complex, even for a centralized homogenous computing environment where all software elements share common operational data and expose uniform management interfaces. A distributed, multivendor installation, however, is simply impossible to manage unless a uniform standard for presenting the operational and statistical data exists and a common protocol for managing the resources is available. That is why, over the years, numerous system engineers have attempted to standardize system management techniques.

In the late 1980s, the Simple Network Management Protocol (SNMP) was designed to address the issue of multi-vendor network management. However, the initial SNMP specification failed to address all the critical network management needs; as a result, SNMP needed a few enhancements. In 1991 the Remote Network Monitoring specification (RMON) was released to overcome SNMP's local area network management limitations. In 1993 an enhanced version of SNMP, widely known as SNMPv2, was released and was subsequently revised in 1995. SNMPv2 augmented the original SNMP specification to provide extended functionality and enhanced its performance. Finally, in 1998 SNMPv3 was issued, primarily to deal with the security capabilities of SNMP and to define the overall architecture for future enhancements. Today, SNMP remains the most popular standard for managing TCP/IP-based internets.

In 1994, another standard, the Desktop Management Interface (DMI), was developed in an attempt to deal with the consequences of the PC revolution. The first DMI specification (DMI v1.0) outlined the ground rules for hardware and software manufacturers; these rules allowed manageable networked desktop systems to be built. In April 1996, this specification was extended to offer remote manageability in networked environments. This extended DMI specification, known as DMI v2.0, was adopted as the industry-standard, and it included a set of operating system-independent and protocol-independent application programming interfaces (APIs) that provided a uniform desktop management framework.

Although SNMP, DMI, and other standards for management instrumentation are a major step forward in the field of system management, they still fail at completely solving the problem. Perhaps the main limitation of these standards is their fairly narrow specialization—each offers just a partial solution and does not provide the unified end-to-end view of the entire management domain. Individual elements or groups of elements are still managed in isolation, and there is little or no integration between management standards and techniques; hence we still need data duplication and specialized management front-ends.

The Birth of WBEM

In 1996 a few industry leaders—BMC Software, Cisco Systems, Compaq, Intel, and Microsoft—set out to address the limitations of the existing management standards and protocols by sponsoring a brand new initiative: Web-Based Enterprise Management (WBEM). The companies' main goal was to develop a uniform way to share management information and control management resources within an entire management domain, irrespective of the underlying platforms, networking technologies, and programming languages. In order to turn this vision into reality, three major design principles were employed:

The Common Information Model (CIM): This uniform, platform, and language independent model represents the managed elements in the enterprise. In addition to covering all major aspects and areas of system management, this model was designed to be open and extensible so that it could describe environment and platform-specific managed elements.

The easy and seamless integration of existing management standards such as SNMP and DMI: WBEM implementations were to allow management information to be translated between the formats utilized by the existing management tools and the CIM.

A standard method for accessing the management information from a variety of distributed managed nodes over different transports: Since the WBEM initiative came in the midst of the Internet revolution, the Web seemed like a natural transport vehicle for sharing and controlling the management information, hence the name Web-Based Enterprise Management.

In June of 1998, in order to achieve industry-wide acceptance and provide an open public forum for ongoing development of WBEM technologies, the founders of WBEM transferred the ownership of this initiative to an organization called the Distributed Management Task Force (DMTF). DMTF, which was founded in 1992 by a group of leading PC manufacturers, is still the industry consortium that leads the development and promotes the adoption of desktop, network, and Internet management standards.

DMTF leadership further accelerated the progress of the WBEM initiative and ensured the wide acceptance of its first standard—the CIM specification. In the next few years, DMTF, along with many participating companies, not only revised and enhanced the CIM specification, but also produced numerous other standards to describe and define uniform protocols for publishing and exchanging the management information. One of the standards that deserves special attention is the XML Encoding Specification, which allows CIM schemas to be encoded in XML. The first draft of this specification was proposed in October 1998, and it replaced the original WBEM data publishing standard, which was called the HyperMedia Management Protocol (HMMP).

Today, DMTF remains fully committed to promoting the WBEM technology as a premier vehicle for accessing the management information in the enterprise environment and lowering the total cost of ownership (TCO) associated with computer hardware and software configuration, deployment, and maintenance. Some software vendors, such as Microsoft and Sun Microsystems, already ship WBEM-compliant management frameworks, and extensive interest in this technology throughout the industry indicates that many more vendors are preparing to incorporate WBEM-based management solutions into their product offerings in the near future.

Introducing the Common Information Model

The *Common Information Model (CIM)* is the centerpiece of the WBEM technology. CIM is a well-defined, conceptual, object-oriented model, designed to serve as a framework that describes all aspects of a managed environment. The management information within CIM is organized into a hierarchy of classes that represent logical and physical elements in the enterprise. The object-oriented modeling approach provides several benefits such as data abstraction, inheritance, and polymorphism. Grouping related objects into classes based on common properties and behavior, for instance, allows the complexity of the model to decrease, while modularity and extensibility are promoted. Inheritance is the ability to construct classes from one or more other parent classes so that derived, or child, classes inherit the characteristics and behavior of the parent. Inheritance upholds the principles of generalization, specialization, and modular design. Finally, polymorphism is the ability of different classes within the same hierarchy to provide specialized responses to the same external message. This promotes extensibility and simplifies the development and maintenance of the management schema.

Management schemas are the primary building blocks of CIM. *CIM Schema* is a named collection of elements that describes a particular functional area within a managed environment, such as device configuration or performance management. CIM is organized as a system of interrelated schemas that cover every aspect of the managed enterprise.

To describe the individual entities within a managed environment, CIM utilizes the generalization technique—it factors common properties and behaviors of the managed elements into sets of classes so that each class reflects a single unit of management. A *CIM class* is essentially a template that describes a particular type of a managed element. These classes can contain properties, also known as data elements, that describe the state of the class instance, and methods that express the behavior of the class. Listing 1-1 presents a partial definition for one of the CIM core classes—`CIM_LogicalDevice`. This class is designed to serve as a high-level abstraction for a hardware element in a managed environment.

Listing 1-1. Managed Object Format Definition for `CIM_LogicalDevice`

```
 [Abstract, Description (
         "An abstraction or emulation of a hardware entity, that may "
         "or may not be realized in physical hardware...") ]
class CIM_LogicalDevice : CIM_LogicalElement  {
...
[Key, MaxLen (64),
       Description (
        "An address or other identifying information to uniquely "
        "name the LogicalDevice.") ]
   string DeviceID;
[Description (
        "LastErrorCode captures the last error code reported by "
        "the LogicalDevice.")]
   uint32 LastErrorCode;
[Description(
         "SetPowerState defines the desired power state for a "
         "LogicalDevice and when a device should be put into that "
         "state")]
   uint32 SetPowerState([IN] uint16 PowerState, [IN] datetime Time);
   [Description ("Requests a reset of the LogicalDevice")]
   uint32 Reset();
...
};
```

The notation used here is called *Managed Object Format (MOF),* which is a DMTF-defined language for specifying management schemas in WBEM. MOF may look a bit intimidating at first, but it is only presented here to provide a context for the discussion—the detailed overview of this language syntax is postponed until Chapter 6.

In this listing, `DeviceID` and `LastErrorCode` are properties that represent the identity and state of a class instance, while `SetPowerState` and `Reset` are methods that express the behavior of the class. Not all classes, however, have methods, especially those defined at the root of the CIM hierarchy. Typically, the root classes represent the highest level of abstraction where it may not always be possible to define any specialized behavioral characteristics.

Another interesting thing you should notice is that MOF definitions do not contain any implementation details for the methods of the class; rather, they specify the method signature. Because MOF is a declarative format for the management data rather than an implementation vehicle, the actual implementation is delegated to so-called data providers (this will be discussed in more detail in Chapter 7). Thus, CIM class specifications are somewhat similar to interface definitions, which are widely used in such frameworks as Component Object Model (COM) and Common Object Request Broker Architecture (CORBA) as well as in some programming languages like Java and C#. Obviously, you can draw a clear parallel between MOF and the Interface Definition Language (IDL).

Even though a class may have one or more methods defined, the implementation for these methods may not necessarily be available. This is because CIM defines a number of classes each of which represent a very high level of abstraction and exist just to serve as parents for more specialized classes. As a result, the actual method implementation is deferred to these respective subclasses. CIM specification requires that all implemented methods be marked with the Implemented qualifier in the MOF definition; this allows a class to indicate that a method is actually implemented by a data provider.

All class names must be unique within a particular schema, and the schema name acts as a distinguishing factor that helps differentiate classes with potentially conflicting names. By convention, the fully qualified name of a class always takes the form <schemaname>_<classname>, where the underscore serves as a delimiter between the name of the schema and the name of the class. Note that the underscore delimiter may not be a part of the schema name, although it is allowed as part of the class name. This convention limits the scope of the class name to the schema, significantly reducing the chances of name collisions between the classes from different schemas. The CIM classes defined by DMTF have the schema name of "CIM," thus the fully qualified name of the class, which represents an operating system process, is CIM_Process. Specific WBEM implementations may provide their own schemas; for instance, the Solaris WBEM SDK defines its own process class, called Solaris_Process, while Microsoft Windows Management Instrumentation (WMI) exposes the Win32_Process class.

As mentioned earlier, inheritance is another powerful concept of object-oriented design. It is used extensively within the CIM. Simply put, inheritance is a technique that allows class designers to construct a class from one or more other classes, while sharing properties, behavior and, sometimes, constraints.

The process of creating a new class using an existing class as a base is often referred to as *subclassing*. There are a few reasons for subclassing. The first, and perhaps the most obvious one is the ability to inherit some properties and standard methods, thus reducing the effort of building a new class. If you refer back to Listing 1-1, you will see that the CIM_LogicalDevice employs CIM_LogicalElement as its base class. Although this is not apparent in Listing 1-1, CIM_LogicalDevice inherits a few properties, such as Name, InstallDate, Description, and Caption.

Another reason why you would use subclassing is if you wanted to specialize some of the class semantics, moving from an abstract base class to a more specific subclass. In addition to adding new properties and methods, a specialized subclass may also redefine some of the existing characteristics of the base class. For instance, a subclass may restate the definition of a particular feature, such as a method description or an informational qualifier. In this case, the subclass overrides the respective characteristic of its base class. Alternatively, a subclass may provide a method or property implementation, different from that of a base class, while inheriting the method signature or property declaration. This process of altering the behavior of an arbitrary method or property is closely related to polymorphism, another fundamental concept of object-oriented design. Polymorphism implies that classes within a certain hierarchy are capable of responding to the same message in a manner specific to a particular class, using a potentially different implementation. This very powerful concept is widely used throughout all areas of the CIM.

Listing 1-2 demonstrates some of these concepts:

Listing 1-2. Inheritance and Method/Property Overriding

```
class CIM_Service : CIM_LogicalElement
{
  ...
   [ValueMap{"Automatic", "Manual"}] string StartMode;
  ...
   uint32 StartService();
   uint32 StopService();
};

class Win32_BaseService : CIM_Service
{
   [ValueMap{"Boot", "System", "Auto", "Manual", "Disabled"},
    Override("StartMode")]
   string StartMode = NULL;
  ...
   [Override("StartService"), Implemented] uint32 StartService();
   [Override("StopService"), Implemented] uint32 StopService();
   [Implemented] uint32 PauseService();
  ...
};
```

Here the base class CIM_Service declares a property, called `StartMode`, and states that the set of allowed values for this property is limited to `Automatic` and `Manual`. The subclass—Win32_BaseService—overrides two aspects of the `StartMode` property: first it adds a default value of `NULL` then it redefines the allowed set of values to `Boot`, `System`, `Auto`, `Manual`, and `Disabled`.

`PauseService`, which also appears in Listing 1-2, is a method specific to the `Win32_BaseService` and it does not exist within the scope of `CIM_Service` base class. You must realize that not every service can be paused; that is why the `CIM_Service` class, in an attempt to remain completely implementation-neutral, does not define any such function. The subclass, however, represents a Win32 service, which can be paused; therefore it declares the `PauseService` method and provides an implementation for it (notice the `Implemented` qualifier here). The subclass essentially extends the functionality of its base class.

The `Win32_BaseService` subclass also inherits two method definitions from its parent—`StartService` and `StopService`—and it provides its own implementation (again, notice the `Implemented` qualifier). Although the `CIM_Service` base class does not implement these methods (no `Implemented` qualifier within the `CIM_Service` definition), the `Win32_BaseService` implementation can still be considered polymorphic since it is specific to the subclass. It is conceivable that some other subclass of CIM_Service may provide its own implementation for these methods, different from that of Win32_BaseService.

Many object-oriented models and languages provide a facility called method overloading where several methods with the same name and different parameter types may coexist. Typically a language processor such as a compiler or an interpreter will select the correct method by inspecting the types of its parameters at the call site. Then a subclass may be able to alter the signature of a method inherited from a base class, thus providing an overloaded method definition. In CIM, however, method overloading is not supported, and a subclass, when overriding a method, must preserve its signature.

Another restriction that CIM imposes is single inheritance. Some object-oriented languages, such as C++, allow a subclass to inherit from more than one parent thus sharing more than one set of characteristics, properties, and methods; this is known as *multiple inheritance*. Although multiple inheritance is a very powerful feature, it typically creates too many problems because there is a great potential for conflicts. For instance, two base classes, A and B, may both implement a method Foo, although the A::Foo and B::Foo implementations may be completely different and not related in any way. Any attempt to create a subclass, C, that inherits from both A and B, will result in a naming conflict between the A::Foo and B::Foo methods. Although different languages and environments offer various solutions for disambiguating multiple inheritance name conflicts, there is no "silver bullet." The designers of CIM felt that the power of multiple inheritance did not justify the complexity associated with resolving these kinds of naming conflicts; thus CIM inheritance trees are single-rooted.

As we have already established, the CIM is a collection of related classes. Relationships between classes are usually expressed via special properties, called *references*. Essentially a reference is a pointer to an instance of a class; thus, in order to establish a relationship between arbitrary classes A and B, instances of either class may have references that point to a respective instance of another class. CIM relationships are modeled through *associations*—bidirectional semantic connections between related classes. In accordance with their bidirectional nature, all CIM associations are classes that contain references to other classes that are related through the association.

This design approach has several benefits. First, associations may have other properties in addition to references to the classes being associated. CIM designers, for instance, often use an example of a Marriage association between a Male and a Female property. A Marriage not only has links to both Male and Female, but it also includes other properties such as Date. Second, if you model associations as classes, you gain greater design flexibility because you can add an association without changing the interface of the related classes. Thus, adding a Marriage association between a Male and Female does not affect the definition of these two classes in any way (this is, perhaps, where there is a disconnect between the design of CIM and real life); the `Marriage` association simply ties them together via a pair of references. To enforce this design approach, CIM disallows nonassociation classes that have properties of reference data type.

As I mentioned earlier, classes are just templates for objects or instances; thus an arbitrary class may have one or more instances. For example, the `CIM_DataFile` class may have hundreds or thousands of instances within a given environment, each representing a single physical data file. To be able to deal with multiple instances of a class in a meaningful fashion, these instances have to be uniquely identifiable within a given system. Numerous instance identification schemes exist, but perhaps the most common scenarios are the following:

Globally Unique Identifiers (GUIDs): GUID-based object identity schemes are used extensively throughout the industry; the most widely known example is the Microsoft Component Object Model (COM). In a GUID-based model, each and every class instance is assigned an artificial identifier, unique across time and space. The benefits of this approach are twofold: GUIDs are guaranteed to be unique, which greatly reduces any chances of collisions; and GUIDs are also relatively cheap to generate. The problem is that GUIDs are intended for machine consumption and are not particularly human-friendly—in fact, they seem quite meaningless and are difficult to memorize.

Natural Keys: In a keyed object model, one or more properties of a class form a unique key that unambiguously identifies an object instance within a given environment. Our example in Listing 1-1, for instance, designates the `DeviceID` property of `CIM_LogicalDevice` class as a key (notice the `Key` qualifier) so that consumers of `CIM_LogicalDevice` instances (or rather instances of its subclasses) may refer to a particular device instance using the ID string. Obviously, using a keyed approach offers certain benefits for the users of the object model because the natural keys are more easily understood and memorized. Unlike GUIDs, however, the natural keys are only unique within a given scope—for instance, if a file is identified by its name or path, the object instance representing `C:\BOOT.INI` will only be unique within a single Windows system.

As I already implied, CIM is a keyed object model. Since CIM must be capable of handling the management data across multiple environments and possibly across multiple physical implementations, object keys alone may not be sufficient to uniquely identify an instance. There has to be a way to identify an environment or implementation or, in other words, provide a scope, within which the objects keys are unique. Thus, every object within CIM is uniquely identified by an object path, as outlined in Listing 1-3.

Listing 1-3. Object Naming

```
object_path ::= <namespace_path><model_path>

where:
   namespace_path ::= <namespace_type><namespace_handle>
   model_path ::= <object_name>.<key>=<value>,[<key>=<value>]

example:
   HTTP://CIMHOST/root/CIMV2:CIM_DataFile.Name="C:\BOOT.INI"

where:
   namespace_type = HTTP
   namespace_handle = CIMHOST/root/CIMV2
   object_name = CIM_DataFile
   key = Name
   value = C:\BOOT.INI
```

A *namespace path* is essentially a unique identifier for a namespace that hosts CIM objects, thus acting as a scope for these objects. Although a namespace path is implementation-specific, it typically provides at least two pieces of information: the type of implementation, or namespace, being referenced, and a handle for this namespace. The *namespace type* defines an access protocol used by the implementation to import, export, and update the management data. In a sense, the namespace type is similar to a protocol specification used in URLs, such as `http` or `ftp`. It is conceivable that a particular implementation may define several protocols or APIs for accessing the data, and in this case, each of these protocols must have an associated unique namespace type defined. A *namespace handle* defines an instance of a namespace within a given implementation. Although the details of the namespace handles are implementation-specific, these handles often include an identifier of a computer system ("CIMHOST" in our example) that hosts the namespace instance. If an implementation supports multiple namespace instances, a handle may also include an identifier for a particular instance within a given system ("root/CIMV2" in our example).

The purpose of the model path is to uniquely identify an object within its respective namespace. A *model path* consists of the name of the class to which the object belongs and one or more key-value pairs, such that each property designated as a key in the class definition is supplied with a value. Thus in our example, the object of type "`CIM_DataFile`" is uniquely identified within its respective namespace (`CIMHOST/root/CIMV2`) by the value of its `Name` property.

The entire CIM Schema is divided into three areas: the core model, the common model, and the extension schemas. The *core model* is a relatively small set of classes and associations that provides a foundation for describing any kind of managed environment and is applicable to all areas of system management. The core model serves as a basis from which to define more specialized extension schemas that are applicable to concrete management domains. In particular, the core model provides abstract base classes; these allow you to classify all managed objects within a system as either physical or logical. Thus, the `CIM_PhysicalElement` class is used as a base for modeling those managed objects that have some kind of physical representation, while the `CIM_LogicalElement` class is used to build abstractions, typically representing the state of a managed system or its capabilities. The core model also provides a set of abstract association classes that are used to model containment and dependency relationships between classes.

The *common model* is a set of classes that define various areas of system management while remaining completely implementation-neutral. This model is detailed enough to serve as a solid foundation for building the technology-specific extension models that are suitable for developing all kinds of system management applications. The common model describes the following aspects of a managed environment:

The systems schema: The systems schema addresses various top-level objects that make up the managed environment, such as computer systems and application systems.

The devices classes: This part of the common model is designed to represent the discrete logical elements of a system that possess basic capabilities, such as processing, input/output, and so on. Interestingly, CIM device classes are descendants of the `CIM_LogicalElement` class as opposite to `CIM_PhysicalElement` class. Although the system devices may appear as physical rather than logical elements, the reason for deriving the device classes from `CIM_LogicalElement` is the fact that management systems deal with the operating system view of the device rather than its physical incarnation.

The networks model: The networks model defines the classes that are necessary to model various aspects of a networked environment, including various network services, protocols, and network topologies.

The applications model: The intent of this model is to provide a basis for managing various software packages and applications. The applications model is fairly flexible and can easily be used to represent not only standalone desktop software components, but also complicated distributed application systems that may run on multiple platforms or be deployed via the Internet.

The physical model: The physical model is designed to reflect all aspects of the actual physical environment. As I already mentioned, the vast majority of managed elements may and should be modeled as logical elements because any changes within the physical environment are likely to be just the consequences of some events happening within the logical "world." For example, rotations of a physical disk platter come as a result of an I/O request that originates within a logical environment—the operating system. Also, the physical elements are unable to directly feed the management data into CIM and their state can only be determined through some controlling logical elements, such as device drivers. Yet another problem is that physical elements differ dramatically from environment to environment due to the difference in the underlying hardware technologies; thus they do not lend themselves easily to any generalized modeling techniques. For all these reasons, the aspects of a physical environment are not a direct concern of CIM designers.

Finally, the *extension schemas* are sets of classes that are specific to a particular CIM implementation and are dependent on the characteristics of a given managed environment. As part of their WBEM product offerings, various vendors typically provide extension schemas. Thus, the Microsoft WMI framework ships with a set of extension classes that represent the managed elements specific to Win32 platforms, and Solaris WBEM SDK exposes classes that are only relevant to the specific management aspects of Solaris systems.

Windows Management Instrumentation

In addition to being one of the founders and key contributors to the WBEM initiative, Microsoft was also one of the first software vendors to ship a fully WBEM-compliant instrumentation framework for its Windows platforms. The Microsoft WBEM implementation, called Windows Management Instrumentation (WMI), is the core of Windows management infrastructure. It was designed to facilitate maintenance and greatly reduce the total cost of ownership (TCO) of Windows-based enterprise systems.

WMI provides the following benefits to system managers, administrators, and developers:

A complete and consistent object-oriented model of the entire management domain: The Microsoft WMI implementation is fully CIMv2.0-compliant and as such, it fully supports all management data abstractions defined by the CIMv2.0 specifications of the core and common models. Additionally, WMI exposes a number of extension schemas that cover aspects of system management specific to Windows platforms.

A single point of access to all management information for the enterprise: WMI exposes a powerful COM-based API, which allows the developers of management applications to retrieve and modify virtually any kind of Windows configuration and status data.

An extensible architecture that allows for the seamless integration of existing management solutions and facilitates the instrumentation of third-party software products: WMI is equipped with a number of adapters that make the management data maintained by legacy systems, such as SNMP, accessible through standard WMI interfaces. Additionally, through WMI, the software vendors are afforded the flexibility of extending the management schema and exposing the specific management data for their software applications and hardware devices.

A robust event mechanism: WMI supports an event-driven programming model where management events, such as changes to system configuration or system state, can be intercepted, analyzed, and acted upon. Once registered as an event consumer, a management application can receive various management events, originating from a local or remote system.

A simple yet powerful query language: One of the most remarkable features of WMI is the WMI Query Language (WQL), which allows the developers of custom management applications to navigate through the WMI information model and retrieve the information in which they are interested. WQL is a subset of standard American National Standards Institute (ANSI) SQL (Structured Query Language) with some minor semantic changes that were necessary to accommodate the specifics of WMI.

WMI Overview and Architecture

Figure 1-1 presents a high-level view of the WMI Architecture.

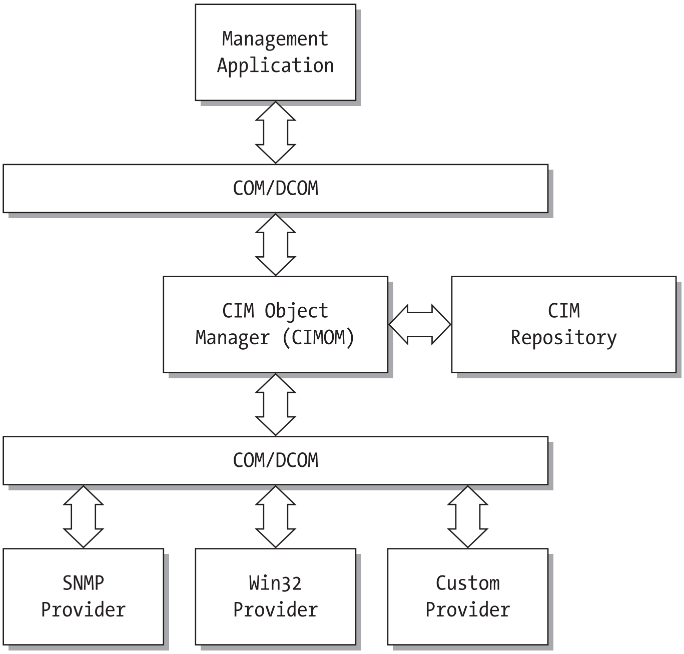


Figure 1-1. TheWMI Architecture

Conceptually, the key component of the WMI is the CIM Object Manager (CIMOM). In compliance with the main goal of WBEM—providing a uniform, centralized way of manipulating the management data—CIMOM acts as a single point of access to the entire universe of managed objects that are of interest to management applications. Although all client requests for management data go through CIMOM, it is not responsible for actually collecting this data. Instead, depending on the nature of the information requested, CIMOM may either retrieve the data from its repository (the CIM Repository) or if the data is not available in the repository, route the request to an appropriate data provider. In the latter case, the data provider may retrieve or generate the requested information on demand and return it back to CIMOM, which in turn will return the data to the client application that initiated the request.

The CIM Repository, managed by CIMOM, is a central storage area, intended primarily for storing the WMI schema information. In some cases, however, the repository may also hold the static instance data—mainly the instances of WMI classes that represent the system configuration information, which is not likely to change often. For example, an instance of the `Win32_Bios` class, which represents the system BIOS, is static and is not likely to change during the normal operations of a system; thus its data is stored in the CIM Repository and retrieved directly by CIMOM. The `Win32_Process` class, on the other hand, may have many transient instances—one for every process that the operating system creates. Generally, the large volumes and transient nature of the instance data make it unsuitable for storage in the CIM Repository, and in cases such as this one, the task of providing the data is delegated to the respective data provider. Providers that generate and return the management data on demand are often referred to as *dynamic providers.*

CIMOM Implementation

WMI packages the bulk of the CIMOM functionality as well as the repository management functions in a single executable file, WinMgmt.exe. This executable is installed into the `%SystemRoot%\System32\Wbem` directory and, on Windows NT/2000 platforms, it runs as a separate service process, which by default has an "Automatic" startup option. Since WinMgmt.exe depends on WinComn.dll (also installed in `%SystemRoot%\System32\Wbem`), which actually implements most of the WMI functionality, under Windows XP, WMI runs as a service host process. For such processes where the functionality is provided by a DLL, Windows XP supplies a generic svchost.exe service executable, which loads the DLL on startup and exposes its functionality via standard service interfaces. Finally, under Windows 98, WMI runs as a standard executable. It is possible, however, to configure WMI to start automatically, even on Windows 98, by doing the following:

1. Set the `HKLM\SOFTWARE\MICROSOFT\OLE\EnableDCOM` registry value to "Y."

2. Set the `HKLM\SOFTWARE\MICROSOFT\OLE\EnableRemoteConnect` registry value to "Y."

3. Add the `HKLM\SOFTWARE\MICROSOFT\WBEM\CIMOM\AutostartWin9X` registry key with value of "1" (which means that there will be an automatic start).

4. Add WinMgmt.exe to the system startup directory.

The WinMgmt.exe can also be run manually from the command prompt, in which case it may be used to perform certain maintenance tasks. Table 1-1 lists the command line switches available in WinMgmt.exe and explains their purpose.

Table 1-1. WinMgmt.exe Command-Line Options

COMMAND-LINE SWITCH	DESCRIPTION
`/exe`	This switch makes WinMgmt.exe run as a standard executable rather than as a service. The primary purpose of this switch is to facilitate the debugging of custom WMI data providers.
`/kill`	This switch shuts down all the WinMgmt.exe processes that are running on a local computer system, including the service processes started by the Service Control Manager as well as the processes started manually with the `/`exe switch.
`/regserver`	This switch registers WinMgmt.exe with the Service Control Manager as a Windows service. This is a standard switch that is normally implemented by all services.
`/unregserver`	This switch unregisters WinMgmt.exe as a Windows service. This is a standard switch that is normally implemented by all services.
`/backup <filename>`	This is a repository maintenance function that causes WinMgmt.exe to back up its CIM Repository to a file, named by the `<filename>` argument. Using this flag will lock the repository in exclusive mode and suspend all pending write requests until the backup operation is completed.
`/restore <filename>`	This is a repository maintenance function that allows for manual restoration of the CIM Repository from the file, named by the `<filename>` argument. When run with this flag, WinMgmt.exe will delete the existing repository file, lock the repository in exclusive mode (which may necessitate disconnecting all existing client connections to WMI), and load the contents of the backup file into the repository.
`/resyncperf <winmgmt service process id>`	This flag is only available on Windows 2000 and Windows XP and is used to invoke the AutoDiscovery/AutoPurge (ADAP) mechanism. ADAP is used to transfer the performance counters from registered performance libraries into WMI classes in the CIM Repository so that these counters may be accessed as WMI object properties.
`/clearadap`	This is another ADAP-related flag that effectively clears all ADAP status and configuration information.

As Figure 1-1 implies, WinMgmt.exe (CIMOM) communicates with the rest of the world through a set of well-defined COM interfaces. In fact, all of the WMI functionality is exposed through COM interfaces; this allows the developers to reap the advantages of component-based programming, such as language and location independence. This means that each of many different WMI objects inherits the interfaces, ultimately derived from IUnknown, and complies with COM-imposed rules for memory management, method call parameter manipulation, and thread handling.

WMI defines many different interfaces that are designed to deal with different aspects of its functionality; most of these are declared in the wbemcli.h header file. In addition to COM API, which is mainly used by C++ developers, WMI supplies a set of automation interfaces; these interfaces enable the scripting clients (for example, programs written in scripting languages such as VBScript, JavaScript, or Perl) to consume most of the WMI functionality. Most of these automation interfaces simply duplicate the functionality available through the primary COM API. For instance, automation interface ISWbemLocator, which is used to obtain a reference to SWSbemServices object (an entry point to WMI), reflects the functionality afforded by the IWbemLocator interface of the primary COM API. Both of these interfaces expose a single method, ConnectServer, which connects to WMI on a given computer system.

Much of the WMI configuration information is stored in the Windows Registry under the key HKLM\SOFTWARE\MICROSOFT\WBEM\CIMOM. Coincidentally, WMI provides the Win32_WMISetting class as part of its schema so that each property of this class maps to a respective configuration value in the system registry. Listing 1-4 shows the MOF definition of the Win32_WMISetting class.

Listing 1-4. Win32_WMISetting Class Definition

```
class Win32_WMISetting : CIM_Setting
{
  string ASPScriptDefaultNamespace  ;
  boolean ASPScriptEnabled  ;
  string AutorecoverMofs[]  ;
  uint32 AutoStartWin9X
  uint32 BackupInterval  ;
  datetime BackupLastTime  ;
  string BuildVersion  ;
  string DatabaseDirectory  ;
  uint32 DatabaseMaxSize  ;
  boolean EnableAnonWin9xConnections  ;
  boolean EnableEvents  ;
  boolean EnableStartupHeapPreallocation  ;
  uint32 HighThresholdOnClientObjects  ;
  uint32 HighThresholdOnEvents  ;
```

```
    string InstallationDirectory  ;
    uint32 LastStartupHeapPreallocation  ;
    string LoggingDirectory  ;
    uint32 LoggingLevel  ;
    uint32 LowThresholdOnClientObjects  ;
    uint32 LowThresholdOnEvents  ;
    uint32 MaxLogFileSize  ;
    uint32 MaxWaitOnClientObjects  ;
    uint32 MaxWaitOnEvents  ;
    string MofSelfInstallDirectory  ;
};
```

Table 1-2 provides an explanation of every property of Win32_WMISetting and indicates the respective registry subkey (relative to HKLM\SOFTWARE\MICROSOFT\WBEM) to which a property maps.

Table 1-2. Win32_WMISetting *Properties*

WIN32_WMISETTING PROPERTY	REGISTRY MAPPING	DESCRIPTION
ASPScriptDefaultNamespace	Scripting\Default Namespace	Contains the default namespace, used by the scripting API calls in case the caller does not explicitly provide the namespace. Usually set to “root\CIMV2”.
ASPScriptEnabled	Scripting\Enable for ASP	Determines whether WMI scripting API can be used by Active Server Pages (ASP) scripts. This property is only applicable to Windows NT systems, since under Windows 2000 and later, WMI scripting for ASP is always enabled.
AutorecoverMofs	CIMOM\Autorecover MOFs	Ordered list of MOF file names, used to initialize or recover the WMI Repository.
AutostartWin9X	CIMOM\AutostartWin9X	Determines how Windows 98 systems should start WinMgmt.exe: 0—Do not start 1—Autostart 2—Start on reboot

(continued)

Table 1-2. `Win32_WMISetting` *Properties (continued)*

WIN32_WMISETTING PROPERTY	REGISTRY MAPPING	DESCRIPTION
`BackupInterval`	`CIMOM\Backup Interval Threshold`	Time interval (in minutes) between the backups of the WMI Repository.
`BackupLastTime`	No registry mapping	Date and time of the last backup of the WMI Repository.
`BuildVersion`	`Build`	Version number of the WMI service installed on the system.
`DatabaseDirectory`	`CIMOM\Repository Directory`	Directory path of the WMI Repository.
`DatabaseMaxSize`	`CIMOM\Max DB Size`	Maximum allowed size (in KB) of the WMI Repository.
`EnableAnonWin9xConnections`	`CIMOM\EnableAnonConnections`	Determines whether remote access may bypass security checking. This property is only applicable on Windows 98 systems.
`EnableEvents`	`CIMOM\EnableEvents`	Determines whether the WMI event subsystem should be enabled.
`EnableStartupHeapPreallocation`	`CIMOM\EnableStartupHeapPreallocation`	If set to `TRUE`, forces WMI to preallocate a memory heap on startup with the size of `LastStartupHeapPreallocation`.

(continued)

Table 1-2. `Win32_WMISetting` *Properties (continued)*

WIN32_WMISETTING PROPERTY	REGISTRY MAPPING	DESCRIPTION
`HighThresholdOnClientObjects` differences in	`CIMOM\High Threshold On Client Objects`	To reconcile the processing speed between the providers and the clients, WMI queues objects before handing them out to consumers. If the size of the queue reaches this threshold, WMI stops accepting the objects from providers and returns an "out of memory" error to the clients.
`HighThresholdOnEvents` differences in	`CIMOM\High Threshold On Events`	To reconcile the processing speed between the providers and the clients, WMI queues events before delivering them to consumers. If the size of the queue reaches this threshold, WMI stops accepting the events from providers and returns an "out of memory" error to the clients.
`InstallationDirectory`	`Installation Directory`	Directory path of the WMI software installation. By default, set to `%SystemRoot%\System32\Wbem`.
`LastStartupHeapPreallocation`	`CIMOM\LastStartupHeapPreallocation`	Size of the memory heap (in bytes) that was preallocated at startup.
`LoggingDirectory`	`CIMOM\Logging Directory`	Directory path to the location of WMI system logs.
`LoggingLevel`	`CIMOM\Logging`	Level of WMI event logging: 0—Off 1—Error logging on 2—Verbose error logging on

(continued)

Table 1-2. `Win32_WMISetting` *Properties (continued)*

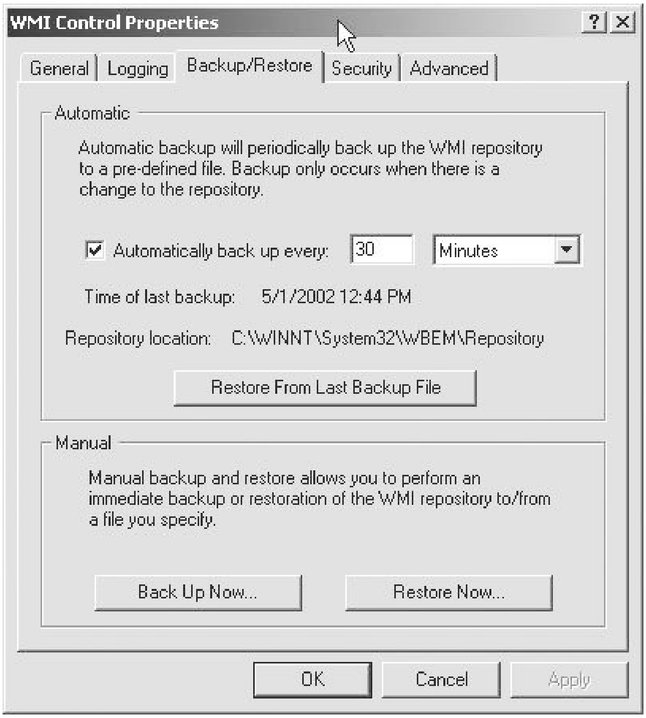
WIN32_WMISETTING PROPERTY	REGISTRY MAPPING	DESCRIPTION
LowThresholdOnClientObjects	CIMOM\Low Threshold On Client Objects	To reconcile the differences in processing speed between the providers and the clients, WMI queues objects before handing them out to consumers. If the size of the queue reaches this threshold, WMI slows down the creation of the objects to accommodate the client's speed.
LowThresholdOnEvents	CIMOM\Low Threshold On Events	To reconcile the differences in processing speed between the providers and the clients, WMI queues events before delivering them out to consumers. If the size of the queue reaches this threshold, WMI slows down the delivery of the events to accommodate the client's speed.
MaxLogFileSize	CIMOM\Log File Max Size	Maximum size of the log file produced by the WMI service.
MaxWaitOnClientObjects	CIMOM\Max Wait On Client Objects	Length of time a new object waits to be used by the client before it is discarded by WMI.
MaxWaitOnEvents	CIMOM\Max Wait On Events	Length of time an event, sent to the client, is queued before it is discarded by WMI.
MofSelfInstallDirectory	MOF Self-Install Directory	Directory path to extension MOF files. WMI automatically compiles all the MOF files that are placed into a directory that is designated by this property, and depending on the outcome of the compilation, moves the files into a "good" or "bad" subdirectory.

The Win32_WMISetting class offers a convenient way to access and modify WMI configuration settings and, for reasons that are obvious, using this class is superior to accessing the WMI-related portion of the registry directly. First, there is no guarantee that future releases of WMI will still keep the configuration data in the system registry, so any management application that relies on the presence of certain registry entries may easily break. Using the interface of the Win32_WMISetting class, on the other hand, ensures the data location transparency. Since the interface is immutable, only the underlying implementation of the Win32_Setting class would have to change in order to accommodate changes to the storage location of WMI configuration parameters. For example, you may have noticed that one of the properties of Win32_WMISetting class, BackupLastTime, does not have the corresponding registry entry; instead WMI dynamically determines its value by examining the timestamp of the repository backup file. The advantage of using the Win32_WMISetting class is clear—an application interested in retrieving the value of BackupLastTime property does not have to be concerned with the implementation details and may simply read the object property. Finally, accessing the registry directly and especially modifying the keys and values is notoriously error-prone and may result in data corruption.

Using WMI-provided interfaces is not only a more elegant approach, but it is also much safer and leaves fewer chances for fatal mistakes. As you will see shortly, reading and manipulating the management data through WMI is not too difficult—in fact, it may even be simpler than programmatically accessing the registry, so writing a small management utility to query and set the WMI configuration data is fairly trivial.

If you are less adventurous and do not find building WMI management utilities too exciting, you can use Microsoft's Management Console (MMC) WMI Control Properties snap-in (see Figure 1-2)—this nice little graphical interface allows you to view and modify most of the WMI-related registry settings.

This snap-in features five tabs, each designed to administer different aspects of WMI behavior.



Automatic backup will periodically back up the WMI repository to a pre-defined file. Backup only occurs when there is a change to the repository.

☑ Automatically back up every: 30 Minutes

Time of last backup: 5/1/2002 12:44 PM

Repository location: C:\WINNT\System32\WBEM\Repository

Manual backup and restore allows you to perform an immediate backup or restoration of the WMI repository to/from a file you specify.

Figure 1-2. The WMI Control Properties MMC snap-in

The General tab: This tab just houses a login prompt that allows you to connect to WMI using the identity of an arbitrary user.

The Logging tab: This tab is a bit more interesting because it controls the logging features of WMI, such as the logging level, the location, and the maximum allowed size of the log file. These controls map to the `LoggingLevel`, `LoggingDirectory`, and `MaxLogFileSize` properties of the `Win32_WMISetting` class.

The Backup/Restore tab: This tab, shown in Figure 1-2, controls the repository backups. By default, the CIM repository is automatically backed up every 30 minutes. Using this tab, you can perform a manual backup or restore the repository.

The Security tab: This tab lets you set access-level permissions for each WMI namespace. The interface for setting up the permission is similar to the one you use to administer file and directory structure permissions in Windows.

The Advanced tab: This last tab, for some mysterious reason labeled "Advanced," allows you to change the value of the `ASPScriptDefaultNamespace` property of the `Win32_WMISetting`.

The WMI Control MMC snap-in is usually all you need to control most of the WMI configuration parameters; however, there are some important aspects of the WMI behavior that it does not address. For instance, you cannot view or change some properties, such as the size of the heap and various thresholds, with the MMC snap-in; if you are interested in these properties, you will have to use either your registry editor or a custom management utility to alter them.

WMI Repository

As Table 1-2 shows, the location of the CIM Repository is either identified by the DatabaseDirectory property of `Win32_WMISetting` class or pointed to by a key `HKLM\SOFTWARE\MICROSOFT\WBEM\CIMOM\Repository` Directory in the system registry. The repository location is usually set to `%SystemRoot%\System32\WBEM\Repository`. The repository itself is implemented as a single file named `CIM.REP`. As I already mentioned, it contains just the WMI schema information as well as some static instance data, so that the size of the repository file is not very large (usually just a few megabytes).

If you wish to extend the WMI schema you will need to provide the MOF definitions for your classes in a form of text files with the .mof extension. You can then load these files into the CIM Repository using either the command line utility mofcomp.exe (MOF compiler) or the WMI COM API. You also have the option of placing the MOF files for your schema extensions into a specially designated self-install directory, pointed to by the `MofSelfInstallDirectory` property of the `Win32_WMISetting` or the registry key `HKLM\SOFTWARE\MICROSOFT\WBEM\MOF Self-Install Directory`. By default, the location of this self-install directory is set to `%SystemRoot%\System32\Wbem\Mof`. Each MOF file placed into this directory is automatically compiled by WMI. If the compilation yields no errors, the file is acknowledged as "good" and moved to the `good` subdirectory of the self-install directory. If the compilation fails, the file is moved to the `bad` subdirectory.

If the repository becomes corrupted, you may have to restore it from the latest backup. However, this backup may not include the definition for those schema extensions that were loaded into the repository after it was last backed up. If this is the case, you may have to manually recompile and load the extension MOFs into the repository. To facilitate the restoration of the extension schemas, the WMI MOF compiler allows you to specify a special preprocessor command—`#pragma autorecover` in the source MOF file. When it encounters this pragma, the MOF compiler will add the fully qualified name of the MOF file to the autorecover list that was pointed to by the `AutorecoverMofs` property of the `Win32_WMISetting` or the registry key `HKLM\SOFTWARE\MICROSOFT\WBEM\CIMOM\Autorecover MOFs`. Then every time the repository is recovered, WMI will compile all of the MOF files on the autorecover list and load them into the repository, thus ensuring that the repository contains all the latest extension definitions.

When talking about WMI class and object naming conventions, I mentioned that a particular WBEM implementation might have multiple namespace instances, uniquely identified by their respective namespace handles. If you closely examine the WMI repository, you will see that WMI utilizes multiple namespace instances that are organized into a hierarchical structure. WMI documentation simply refers to these namespace instances as namespaces, so in order to avoid confusion, I will use the same terminology from now on. Multiple namespaces within WMI serve the purpose of logically grouping related classes and they do nothing more than just provide the scope for name resolution. A typical WMI installation has the following namespaces:

`root`: This namespace is at the very top of the WMI namespace hierarchy and its primary purpose is to contain other namespaces.

`root\DEFAULT`: The `DEFAULT` namespace holds most of the system classes, most of which are of little interest to a typical application developer.

`root\CIMV2`: The `CIMV2` namespace contains the classes and instances of classes that represent the elements in the managed environment. The name `CIMV2` stands for Common Information Model Version 2.0, which implies that this namespace contains all of the classes specified by CIMv2.0 as well as any additional Win32 specific classes and instances. A typical management application will primarily be concerned with the contents of the `root\CIMV2` namespace.

A particular WMI installation may also supply some additional namespaces; the extensions schemas, for instance, are often placed in a separate namespace to avoid potential naming conflicts. Thus Microsoft Internet Explorer may provide some WMI classes, such as `MicrosoftIE_InternetExplorer`, which are usually placed into the `Applications\MicrosoftIE` namespace within the `root\CIMV2` namespace. You may also see `ms_409` or other similarly named namespaces within the `root\CIMV2` namespace or within any of the extension namespaces. These namespaces are used to segregate the localized versions of WMI classes; in fact, the name "ms_409" is just a locale identifier. Microsoft locale identifiers take the form of "MS_XXXX" where "XXXX" is a hexadecimal locale string or Locale Identifier (LCID), so that "ms_409" stands for "American English".

Each WMI namespace contains a `__NAMESPACE` system class (notice the double underscore, which is the class naming convention for system classes); this is so that each instance of this class describes a single subordinate namespace. For example, the root namespace will contain at least two instances of the `__NAMESPACE` class: one for `root\DEFAULT` and one for `root\CIMV2`. The `__NAMESPACE` class is very simple and contains just a few useful properties, such as the name of the respective namespace, its full and relative paths, and the name of the computer system on which it resides. Nevertheless, this class is very convenient because it allows the developer to quickly determine the count and the names of all subordinate namespaces by simply enumerating the instances of the `__NAMESPACE` class.

If you closely inspect the objects in the WMI repository, you will see that every object, regardless of its class, has a number of system properties, which, by convention, are prefixed with double underscores. None of the MOF examples presented so far included these properties; in fact, none of the MOF files distributed with WMI include these properties either. Instead, the system properties are automatically added by WMI as part of the object creation process. The main purpose of these system properties is to identify a particular object and determine its place within the WMI class hierarchy. Table 1-3 shows all available system properties.

Table 1-3. WMI System Properties

PROPERTY	DESCRIPTION
`__CLASS`	The name of the class the object belongs to. This is a read-only property. Example: for an instance of `Win32_NTEventlogFile`, `__CLASS` is set to `Win32_NTEventlogFile`.
`__DERIVATION`	A list of class names, showing the inheritance hierarchy of a given class. The first element is the immediate superclass; the next one is its parent, and so on. This is a read-only property. Example: for an instance of `Win32_NTEventlogFile`, `__DERIVATION` is set to (`CIM_DataFile`, `CIM_LogicalFile`, `CIM_LogicalElement`, `CIM_ManagedSystemElement`).
`__DYNASTY`	The name of the top-level class from which the class is ultimately derived. This is a read-only property. Example: for an instance of `Win32_NTEventlogFile`, `__DYNASTY` is set to `CIM_ManagedSystemElement`.
`__GENUS`	A numeric value that is used to distinguish between classes and instances of a class. The value of '1' represents class and '2' represents instance. This is a read-only property. Example: for an instance of `Win32_NTEventlogFIle`, `__GENUS` is set to '2'.
`__NAMESPACE`	The name of the namespace to which the class belongs. This is a read-only property. Example: for an instance of `Win32_NTEventlogFile`, `__NAMESPACE` is set to `root\CIMV2`.
`__PATH`	The full path to the class or instance. This is a read-only property. Example: for an instance of the `Win32_NTEventlogFile` class, `__PATH` is set to `\\machine1\root\CIMV2:Win32_NTEventlogFile`.
`__PROPERTY_COUNT`	A number reflecting the total count of non-system properties, defined for the class. This is a read only property. Example: for an instance of the `Win32_NTEventlogFile` class, `__PROPERTY_COUNT` is set to 39.
`__RELPATH`	A path to the class or instance, relative to the namespace. This is a read-only property. Example: for an instance of the `Win32_NTEventlogFile` class, `__RELPATH` is set to `Win32_NTEventlogFile`.
`__SERVER`	The name of the server that supplies the class or instance. This is a read-only property. Example: for an instance of the `Win32_NTEventlogFile`, `__SERVER` is set to `machine1`.
`__SUPERCLASS`	The name of the immediate superclass for a class or instance. This is a read only property. Example: for an instance of `Win32_NTEventlogFile`, `__SUPERCLASS` is set to `CIM_DataFile`.

WMI Data Providers

As powerful and sophisticated as the CIM Object Manager looks, it is just a middleman whose primary responsibility is to communicate with data providers on behalf of client applications. Thus the task of manipulating the actual managed elements represented via WMI classes, objects, properties, and events, such as retrieving or updating the management information, is left to the data providers. For instance, if a client application asks CIMOM to retrieve a value of the `DatabaseDirectory` property of the `Win32_WMISetting` object, CIMOM delegates the handling of this request to a standard WMI registry provider, which, when the request is acknowledged, reads and returns the value of the `HKLM\SOFTWARE\MICROSOFT\WBEM\CIMOM\Repository Directory` registry entry.

Not all providers are created equal, and depending on the type of functionality exposed and the types of requests serviced, the providers can be categorized as follows:

Instance providers: The instance providers are, perhaps, the most common type of WMI providers; their primary purpose is to supply instances of a given class and support such operations as instance retrieval, enumeration, modification, and deletion, as well as query processing. For instance, the Event Log Provider (which provides access to the Windows Event Log data and event notifications) acts as an instance provider since it supplies the instances of the `Win32_NTEventlogFile` class, which represent the system event log.

Class providers: The only purpose of class providers is to provide applications with class definitions. You only need this if you are going to dynamically generate class definitions and they are affected by factors outside of WMI. In most of the cases, these class definitions are static, and once they are placed into the CIM repository, they never or rarely change; that is why class providers are rare. Yet another reason to avoid class providers is that they have an adverse effect on the performance of WMI; this is because in order to retrieve a definition for a class, CIMOM has to contact the provider rather than just read the repository. Sometimes, however, using class providers is unavoidable; they have to be used despite the performance penalty that they incur. One example of this is when you have to use the Active Directory Services provider dsprov.dll, which enables WMI applications to interoperate with Microsoft Active Directory Service Interfaces (ADSI). Due to the dynamic nature of the information housed by Active Directory, storing the data in the CIM Repository is not practical.

Property providers: As the name implies, the property providers retrieve and modify the values of instance properties. As opposite to instance providers, property providers allow the client to manipulate the values of individual properties rather than modifying the entire instance. The NT Event Log provider is also a property provider because it supports the operations on individual properties of `Win32_NTEventlogFile` instances.

Method providers: The method providers implement the methods of a given class or a collection of classes. For example, the NT Event Log provider is a method provider because it implements the methods of the `Win32_NTEventlogFile` class (and other related classes) such as `Compress` or `Uncompress`.

Event providers: The event providers are responsible for delivering the event notifications that originate from their respective data sources to WMI CIMOM, which forwards these events to the interested applications. The Event Log provider, which is also an event provider, supports the `Win32_NTLogEvent` class that is used to represent the Windows events.

Event consumer providers: The event consumer providers are used to support the permanent event consumer architecture within WMI. Permanent consumer architecture enables the developers to implement permanent event consumers—also known as custom event sinks that are automatically invoked by WMI every time an event of interest is triggered. Thus, the primary responsibility of an event consumer provider is dispatching an event to be handled by the proper consumer sink. One example of an event consumer provider that comes with WMI distribution is the WMI Event Viewer.

In addition to being classified in one of the categories just mentioned, providers may be categorized as push or pull providers, based on the nature of their interactions with the rest of the WMI infrastructure. *Push providers* typically manage the data that is fairly static and does not change frequently. At initialization time, a push provider simply stores its data in the CIM Repository so that each client request can be serviced directly by CIMOM without incurring the overhead that is the result of communicating with the provider. Not only does this push approach significantly simplify the provider development (because providers are not required to implement their own data retrieval or event notification services), but it is also very efficient since CIMOM is optimized for retrieving the data from the repository.

Unfortunately, storing large amounts of frequently changing data in the repository is not practical; therefore, the vast majority of data providers are implemented as pull providers. *Pull providers* respond to requests from CIMOM by either dynamically generating the data or by retrieving the data from some kind of local cache that is maintained by the provider itself. Although this model supports handling large amounts of dynamic data, it lacks the efficiency and significantly complicates the programming of WMI providers.

There are two parts to provider implementation: a section of the WMI schema that describes the managed elements that are supported by the provider; and the provider DLLs that contain the implementation code. The Windows Event Log provider, for example, is implemented as a single DLL, ntevt.dll, and is usually installed in `%SystemRoot%\System32\Wbem`. The WMI classes supported by this provider, such as `Win32_NTEventlogFile`, `Win32_NTLogEvent`, and many more, are defined in `ntevt.mof` file, which can also be found in the `%SystemRoot%\System32\Wbem` directory.

As has already been mentioned, CIMOM communicates with providers through a set of well-defined COM interfaces. WMI requires providers to implement different interfaces based on the provider type. Push providers, for instance, are only required to implement the `IWbemProviderInit` provider initialization interface that is used by CIMOM whenever a provider is loaded. It is the responsibility of a push provider to ensure that the CIM Repository is updated with proper data when its initialization interface is invoked. When the initialization is complete and the repository contains the updated copy of the provider's data, WMI takes over and services all client requests itself, without invoking the provider.

The situation is different, however, for pull providers, which are required to implement a slew of other interfaces, depending on whether they act as instance, property, method, or event providers. A property provider, for instance, is obliged to implement the `IWbemPropertyProvider` interface, which exposes methods for getting and setting the values of object properties. The methods of this interface are invoked by CIMOM whenever a client issues a property retrieval or modification request. Just like the CIMOM COM interfaces described earlier, most of the provider interfaces are declared in the `wbemcli.h` header file.

Since COM is the sole communication vehicle between CIMOM and the data providers, the providers are registered with COM just like any other COM objects. However, in order for WMI to dispatch a client request to a proper provider, it has to maintain its own provider registration database. Thus, each WMI provider is described by a static instance of the `__Win32Provider` system class, which resides in the CIM Repository. Instances of this system class contain just the basic provider identification information, such as the Class Identifier (CLSID) of the COM object and the provider name.

To identify the provider as instance, property, method, or event provider, WMI maintains a collection of class instances that are derived from

__ProviderRegistration. An instance provider, for example, is represented by a static instance of the __InstanceProviderRegistration class; a method provider is represented by __MethodProviderRegistration, and so on. Each of the subclasses of the __ProviderRegistration class has at least one property—a reference to a respective instance of the __Win32Provider class—although some subclasses may have other type-specific properties that indicate whether a provider supports certain functionality. Thus, if WMI knows the name and the type of the required provider, it may quickly look up an appropriate instance of the __ProviderRegistration subclass, determine its capabilities, and follow its provider reference to retrieve the corresponding instance of the __Win32Provider class that contains all the information it needs to invoke the methods of this provider.

Provider-based architecture is, perhaps, the key to the extensibility of WMI. The well-defined interfaces that WMI uses to communicate with providers allow third-party vendors to instrument their applications by providing extension schemas and custom provider DLLs. WMI distribution comes with a number of built-in providers that usually cover the management needs of a rather sophisticated computing environment sufficiently. The following are examples of some of these providers:

NT Event Log Provider: This supplies Windows Event Log data and notifications of Windows events.

Performance Counters Provider: This enables management applications to access the performance counters' raw data.

Registry provider: This provides access to the data stored in the system registry.

Win32 Driver Model (WDM) provider: This provides access to data and events that are maintained by Windows device drivers and conforms to WMI interface specifications.

Win32 provider: This acts as an interface to a Win32 subsystem.

SNMP provider: This enables WMI applications to interoperate with SNMP.

Of these providers, the SNMP provider is especially interesting because it is actually an adapter that allows data and events, maintained by a legacy management system, to be incorporated into WMI.

The main challenge of making WMI interoperate with SNMP is figuring out how to represent the SNMP data in a CIM-compliant fashion. SNMP maintains its own database of managed objects, called Management Information Base (MIB). MIB is a collection of files that describe the management data using the Abstract Syntax Notation (ASN) language, which is quite different from MOF. To address this difference, the developers of an SNMP provider offer two possible approaches. The first is to use the SNMP information module compiler, which transforms the MIBs into a format understood by WMI so that the output of the compilation can be loaded into the CIM Repository. Once the SNMP-managed objects are defined in WMI, SNMP instance and event providers map the WMI requests into SNMP operations and present the SNMP traps as WMI events. The second approach (which is a bit less efficient) is to make use of the SNMP class provider. This provider generates dynamic SNMP class definitions on request from WMI. With this approach, efficiency is sacrificed to increase flexibility because none of the SNMP MIBs need to be manually compiled and loaded into CIM.

Regardless of the integration approach chosen, the SNMP provider makes SNMP-managed objects appear as an integral part of the WMI universe, thus fulfilling the promise of WBEM founders—seamless integration with legacy management standards and protocols.

Introducing System.Management Namespace

From the day WMI was introduced, the developers that wished to access Windows management resources and services had a choice. They could either use the native WMI COM interfaces or they could use its scripting API. Unfortunately, neither of these two options was completely problem-free. Though the COM API offered virtually unlimited power, high performance, and access to even the most obscure features of WMI, it was, after all, just another COM API. As such, it remained completely inaccessible to millions of those poor developers and system administrators who never managed to overcome the COM's steep learning curve.

The scripting API partly solved this problem bringing the joy of WMI to an audience that was much wider than just a bunch of skilled C++ programmers. However, even the scripting, despite its simplicity and adequate power, did not appear to be a complete solution to the problem. First, it was not fast enough. This was because the dispatch interfaces that were necessary for scripting clients did not offer the same speed as the native COM API. Second, the scripting API lacked power and covered only a limited subset of WMI functionality. In fact, certain things, such as provider programming, remained outside the realm of script developers and still required the use of native COM interfaces.

The rollout of the Microsoft .NET Platform took the world of software development by storm. It is rapidly (and hopefully forever) changing the Windows programming paradigm. Programmatic access to WMI was one of the million things that has been radically affected by .NET. Using .NET you can completely replace both the native COM and the scripting APIs. In the true spirit of .NET, the new WMI interface combines the best features of the older APIs; it merges the performance and unlimited power of the native COM API with accessibility and simplicity of the scripting interface.

There are two parts to the .NET Platform: the .NET Framework and the Framework Class Library (FCL). While the Framework, which supports such modern programming concepts as automatic garbage collection, seamless language interoperability, and much more, is definitely the enabling technology behind .NET, it remains fairly transparent to a casual developer.

The FCL, on the other hand, is something that a programmer will immediately appreciate; it exposes thousands of classes or types that address nearly every aspect of Windows programming. There are types that deal with GUI and Web Interface development, types that are designed to make working with structured and unstructured data easier, and, most importantly, there are types that are dedicated solely to interfacing with WMI. The entire FCL is structured so that functionally related types are organized into a hierarchy of namespaces. For instance, types that are used to build Windows GUI applications are grouped into the `System.Windows.Forms` namespace. The namespace that holds all the types that you need to interact with WMI is called `System.Management`. The most important types, contained in the `System.Management` namespace, are the following:

`ManagementObject`**:** This is a fundamental abstraction that is used to represent a single instance of a managed element in the enterprise.

`ManagementClass`**:** This is a type that corresponds to a class of a managed object as defined by WMI.

`ManagementObjectSearcher`**:** This type is used to retrieve collections of `ManagementObject` or `ManagementClass` objects that satisfy particular criteria specified by a WQL query or enumeration.

`ManagementQuery`**:** This type is used as a basis for building queries, used to retrieve collections of ManagementObject or ManagementClass objects.

`ManagementEventWatcher`**:** This is a type that allows you to subscribe for WMI event notifications.

Nested inside the `System.Management` namespace, there is a `System.Management.Instrumentation` namespace that contains types that are used primarily when you instrument .NET applications. These types allow application developers to use WMI to expose the management data relevant to their applications as well as their application-originated events; this process makes these events and data accessible to a wide variety of management clients.

Because the .NET model used to expose the management applications is mainly declarative, you will not need to use much coding. The `System.Management.Instrumentation` namespace includes a number of attribute types; developers can use these to describe their managed objects and classes to WMI using simple declarations. In addition to these attributes, this namespace defines a number of schema types that are designed to serve as base types for custom managed classes. These schema types already include all necessary attribution so that WMI immediately recognizes any custom type that uses a schema type as its parent.

As the next few chapters will show, the .NET System.Management types offer enough building blocks to solve nearly any system management problem, and backed by the power and flexibility of the .NET Framework, these types are likely to become the ultimate platform that will be used to develop future management applications. While the Framework addresses such general programming issues as automatic memory management, language interoperability, security, and distribution and maintenance ease, the `System.Management` namespace of the FCL brings the following benefits to the developers of management applications:

Consistent object-oriented programming model: As is most of the FCL, the `System.Management` namespace is organized in a very consistent fashion and exposes a well designed object model, which is both natural and easy to understand. The types are well thought-out logical abstractions that are not affected by the peculiarities of the WMI inner workings; as a result they are fairly self-describing and comprehensible. The overall programming paradigm is consistent with the rest of the .NET programming model. This consistency makes it so that .NET developers do not need to learn new skills in order to start programming management applications.

Relative simplicity: Unlike COM programming, in .NET, developers no longer have to take care of the low-level plumbing, such as memory management via reference counting. Instead, they can concentrate on the problem domain. The .NET programming model not only greatly minimizes the amount of boilerplate code for which the developers are responsible, but it also reduces the level of complexity to match that of the legacy WMI scripting API.

Uncompromised performance: When compared to the WMI scripting interface, the .NET System.Management types offer a significant performance enhancement. This is because you no longer a need to use the inefficient dispatch interfaces in order to access the WMI functionality. However, this does not mean that the System.Management types completely solve the performance problems associated with WMI. Unfortunately, some of these performance problems have little to do with the API used by the client applications. The dynamic class providers, for example, are inherently slow and no matter how fast a client application may be able to process the data, the overall efficiency is still hampered by the necessity to generate the class definitions on the fly. Nevertheless, .NET System.Management types offer a significant performance improvement over the less efficient scripting clients.

Tight integration with .NET: The types of `System.Management` namespace are an integral part of FCL and as a result, they comply with all the .NET programming principles and interoperate seamlessly with the types in other namespaces. Typically, even the simplest management application still has to possess some basic user interface capabilities and may request the operating system or I/O subsystem services. Thus, the FCL offers a complete end-to-end solution to the entire universe of programming problems by making thousands of uniformly designed types available. These types adhere to consistent naming conventions, all use the same error handling protocol, feature the same event dispatching and notification mechanism, and most importantly, are designed to work together.

So with a complete arsenal of powerful tools at their disposal, programmers can build and deploy large-scale enterprise management systems.

As you can see, the .NET System.Management types represent an important step toward turning Microsoft Windows into the number one enterprise-computing platform. However, so far I have still yet to answer a couple of questions: How exactly do these types interact with WMI? Also, did Microsoft intend to replace some or all of the WMI infrastructure with .NET-compliant implementation, or was the `System.Management` namespace designed to work in concert with the existing WMI components? The best way to answer these questions is by taking a closer look at how the System.Management types are implemented.

As I already mentioned, the main COM interface that the client and provider applications use to access WMI is `IWbemServices`. This interface exposes a slew of methods that let you issue queries so that you can retrieve collections of classes and instances, create and delete instances, update instance properties, and execute methods. Thus, perhaps the first thing you would want to do with any WMI application is make sure that it gets hold of an object that implements the `IWbemServices` interface. Listing 1-5 shows a common coding pattern that you can use to retrieve an `IWbemServices` interface pointer. Note that for the sake of simplicity, this example does not implement the proper error checking and does not include the code necessary to initialize the variables used in the method calls.

Listing 1-5. Retrieving an `IWbemServices` *Interface Pointer*

```
IWbemLocator  *pIWbemLocator = NULL;
IWbemServices *pIWbemServices = NULL;

CoCreateInstance(CLSID_WbemLocator,
                 NULL,
                 CLSCTX_INPROC_SERVER,
                 IID_IWbemLocator,
                 (LPVOID *) &pIWbemLocator);
...
pIWbemLocator->ConnectServer(pNamespace,
                             NULL,
                             NULL,
                             0L,
                             0L,
                             NULL,
                             NULL,
                             &pIWbemServices);
```

Here the client application first creates an instance of `WbemLocator` class, which you then use to retrieve the `IWbemServices` interface pointer. The `WbemLocator` class implements the `IWbemLocator` interface, which has a single method, `ConnectServer`. The purpose of this method is to establish a connection to WMI services on a particular host. This method takes several parameters, namely the full path of the namespace to which the client application wishes to connect (`pNamespace`), the address of the memory location that will hold the `IWbemServices` interface pointer (`pIWbemServices`), and a few security-related arguments, which our example ignores completely. When the `ConnectServer` method finishes successfully, the `pIWbemServices` variable will contain a valid `IWbemServices` interface pointer that you can use as a main point of access to WMI services and resources.

Now let's see how a typical .NET application accomplishes the same task of connecting to WMI.The type in the System.Management namespace that is responsible for connecting to WMI is `ManagementScope`, and Listing 1-6 shows the disassembly of the class definition and one of its methods, called InitializeGuts. The disassembly listing, which contains Microsoft Intermediate Language (MSIL) instructions, was produced using ILDASM.EXE utility, distributed as part of the .NET Framework SDK.. The MSIL instruction sequences may look quite intimidating at first; however, given the limited amount of documentation that comes with .NET, disassembling various parts of FCL is an excellent source of in-depth information, pertinent to the implementation of certain .NET features. Throughout the course of this book, I will occasionally resort to using such disassembly listings to help you better understand the underpinnings of the System.Management types. The disassembly listing, presented here, is not a complete implementation of the ManagementScope type or its InitializeGuts method—for the sake of simplicity, only the code fragments relevant to the WMI connection establishment process are shown.

Listing 1-6. Disassembly of the InitializeGuts *Method of the* ManagementScope *Type*

```
.class public auto ansi beforefieldinit ManagementScope
         extends [mscorlib]System.Object
         implements [mscorlib]System.ICloneable
{
   ...
    .field private class System.Management.IWbemServices wbemServices
   ...
    .method private hidebysig instance void
            InitializeGuts() cil managed
    {
      // Code size       268 (0x10c)
      .maxstack  9
      .locals init (
                class System.Management.IWbemLocator V_0,
                string V_1,
                bool V_2,
                class System.Management.SecurityHandler V_3,
                int32 V_4,
                class [mscorlib]System.Exception V_5)

      newobj     instance void System.Management.WbemLocator::.ctor()
      stloc.0
  ...
      ldflda     class System.Management.IWbemServices
                System.Management.ManagementScope::wbemServices
      callvirt   instance int32
        System.Management.IWbemLocator::ConnectServer_(
                      string,
                     string,
                     string,
                     string,
                     int32,
                     string,
                     class System.Management.IWbemContext,
                     class System.Management.IWbemServices&)
   ...
     }
    ...
}
```

The first important thing you should notice about Listing 1-6 is that there is a private member variable wbemServices of type System.Management.IWbemServices declared using the .field directive at the beginning of the class definition. This variable has been designed to hold what appears to be a reference to an instance of IWbemServices type. Also, at the very beginning of the InitializeGuts method, notice that a local variable of type System.Management.IwbemLocator is declared. This declaration marks a storage location that will hold a reference to the IWbemLocator instance. The first executable instruction of the InitializeGuts method, newobj, creates an instance of the System.Management.IWbemLocator type and invokes its constructor. When this operation finishes, the operands stack will hold a reference to a newly created instance of the IWbemLocator type. The method will now execute its next instruction, stloc.0. This pops the reference off the stack and stores it at the location that is designated by the first local variable declaration—V_0 of type System.Management.IWbemLocator. Then the code loads the wbemServices member variable's address on the stack using the ldflda instruction and calls the ConnectServer method of the IWbemServices type, which passes this address as one of the parameters. When the ConnectServer method returns, the wbemServices variable contains a valid reference to IWbemServices type.

The code described here is very similar to the native COM API code, shown previously in Listing 1-5; in fact, the interface usage pattern is the same! Now, the only thing that remains unclear is how the IWbemLocator and IWbemServices types are implemented in the System.Management namespace. Listing 1-7 should clear this up by showing the complete class declarations for System.Management.IWbemLocator and System.Management.IWbemServices types.

Listing 1-7. IWbemLocator *and* IWbemServices *Class Declarations*

```
.class interface private abstract auto ansi import IWbemLocator
{
  .custom instance void [mscorlib]
  System.Runtime.InteropServices.TypeLibTypeAttribute::.ctor(int16) =
  ( 01 00 00 02 00 00 )
  .custom instance void [mscorlib]
  System.Runtime.InteropServices.GuidAttribute::.ctor(string) =
  ( 01 00 24 44 43 31 32 41 36 38 37 2D 37 33 37 46 // ..$DC12A687-737F
    2D 31 31 43 46 2D 38 38 34 44 2D 30 30 41 41 30 // -11CF-884D-00AA0
    30 34 42 32 45 32 34 00 00 )                    // 04B2E24..
  .custom instance void [mscorlib]
  System.Runtime.InteropServices.InterfaceTypeAttribute::.ctor(int16) =
  ( 01 00 01 00 00 00 )
}

.class interface private abstract auto ansi import IWbemServices
{
  .custom instance void [mscorlib]
  System.Runtime.InteropServices.InterfaceTypeAttribute::.ctor(int16) =
  ( 01 00 01 00 00 00 )
  .custom instance void [mscorlib]
  System.Runtime.InteropServices.TypeLibTypeAttribute::.ctor(int16) =
  ( 01 00 00 02 00 00 )
  .custom instance void [mscorlib]
  System.Runtime.InteropServices.GuidAttribute::.ctor(string) =
  ( 01 00 24 39 35 35 36 44 43 39 39 2D 38 32 38 43 // ..$9556DC99-828C
    2D 31 31 43 46 2D 41 33 37 45 2D 30 30 41 41 30 // -11CF-A37E-00AA0
    30 33 32 34 30 43 37 00 00 )                    // 03240C7..
}
```

Interestingly, the declarations of both types happen to carry the import flag, which indicates that these types are not implemented in managed code. When the .NET runtime encounters a type marked with the import flag, it identifies a type as COM server and invokes its COM interoperability mechanism. Obviously, COM objects are quite different from the types implemented in managed code—they are allocated from the unmanaged heap and require explicit memory management. Thus, to accommodate a COM server, .NET runtime creates *Runtime Callable Wrappers (RCW)* for each instance of a COM object to be consumed by the managed code. An RCW is a managed object, allocated from the garbage-collected heap, that caches the actual reference-counted COM interface pointer so that .NET code treats it just like any other managed type. To the COM server, however, RCW looks like a conventional well-behaved COM client that adheres to all COM rules and restrictions.

In short, each time an instance of `IWbemLocator` or `IWbemServices` types is requested, the runtime silently creates an underlying COM object, allocates the corresponding RCW, and returns the reference to this RCW back to the requestor.

The last question to answer is how the runtime knows which COM object to allocate when the managed code requests an instance of `IWbemLocator` or `IWbemServices` type. As you may have noticed, the declarations of both of these classes are decorated with several attributes—`System.Runtime.InteropServices.GuidAttribute` is one of these. You can then see that the constructor for this attribute takes a string parameter, which specifies the GUID of the COM server to be created. To no surprise, the inspection of the Windows registry shows that the COM objects, used here, are the same COM objects that are utilized when programming native COM API WMI applications.

As it turns out, the types of the `System.Management` namespace are by no means a complete reimplementation of the WMI access API. Instead, the entire .NET system management class library is just a clean, managed, object-oriented wrapper that is implemented on top of the existing COM WMI-access API.

Summary

This chapter has provided a comprehensive overview of the latest trends in enterprise system management, defined the main objectives of an ideal management system, and introduced a leading-edge management technology—Microsoft WMI. Although I did not intend to supply an exhaustive technical overview of all components that constitute WMI, I hope that the material presented in this chapter was enough to help you understand its most fundamental concepts. These are essential for grasping the material that I will present in the rest of this book. Armed with this knowledge, you should now be ready to delve into the intricacies of .NET WMI programming.