

Applikations- und Systemmanagement mit WMI.NET

Schweizer Taschenmesser für Entwickler

WMI bietet dem Programmierer einen eleganten und einheitlichen Weg, um Daten über das lokale System oder komplette Netzwerk-Installationen einzuholen. Oft sind hier Informationen erhältlich, auf die man andernfalls nur umständlich und mit guten Kenntnissen des Windows-APIs Zugriff hat. Dieser Artikel zeigt, wie Sie mit den Klassen des .NET Framework auf WMI zugreifen und für alltägliche Aufgaben die Fähigkeiten von WMI ausnutzen.

„WMI? Ist das nicht etwas für Admins zum Skripten?“ – So oder zumindest ähnlich reagiert mancher, wenn er hört, dass es unter Windows eine Technologie dieses Namens gibt, mit der man Systemmanagement betreiben kann. Direkt im Anschluss folgt womöglich noch der Kommentar: „Systemmanagement mache ich nicht, ich programmiere normale Applikationen.“ Genau an solchen Meinungen liegt es vermutlich, dass die Windows Management Instrumentation (WMI) immer noch eine verhältnismäßig unbekannt Technologie ist. Ziel des Artikels ist es, WMI etwas näher unter die Lupe zu nehmen.

Ein Kollege hat WMI einmal als das Schweizer Taschenmesser des Programmierers bezeichnet. Im Wesentlichen wurde WMI für drei Aufgaben entwickelt:

- einheitliche Beschaffung von Informationen über im System vorhandene Komponenten und Abläufe,
- Systemkomponentenverwaltung,
- Verwaltung eigener Applikationen.

WMI bietet einen einheitlichen Weg zu vielen Arten von Informationen. Denkbar WMI-Anwendungen sind beispielsweise die Anzeige aller installierten Drucker in einer Dialogbox, das Beenden eines Prozesses auf einem Remote-System oder die Ermittlung der Mac-Adressen der lokalen Netzwerkkarten.

Die WMI-Technologie blickt mittlerweile auf eine mehrjährige Geschichte zurück und ist eine Implementierung des sogenannten Common Information Models (CIM). Das CIM wurde von einem Industrieverband namens Desktop Management Taskforce (DMTF) entwickelt und stellt das Kernelement der WBEM-Initiative (Web Based Enterprise Management) dar. Ziel der WBEM ist ein einheitliches, netzwerkübergreifendes Systemmanagement. In der Windows-Welt tauchte eine Implementierung des CIM in Gestalt der WMI Services bereits zu NT4-Zeiten auf und wurde ab Windows 2000 fester Bestandteil jeder Windows-Umgebung. Dort ist es als Systemdienst verfügbar und über verschiedene Schnittstellen programmierbar. Das .NET Framework kapselt diese Schnittstellen in einer Funktionsbibliothek. So ist ein komfortables Arbeiten mit WMI möglich.

WMI hinter den Kulissen

Das Common Information Model, auf dem WMI basiert, sieht alle verwaltbaren Elemente eines Systems als Objekte. Verwaltbare Elemente können hier sowohl physikalischer Natur (etwa eine Netzwerkkarte oder ein Drucker) als auch logischer Natur (zum Beispiel Netzwerkgaben

WMI – Spezifikationen und Ressourcen

- Distributed Management Task Force www.dmtf.org/
- Web Based Enterprise Management (WBEM)-Initiative www.dmtf.org/standards/standard_wbem.php
- WMI-Dokumentation http://msdn.microsoft.com/library/en-us/wmisdk/wmi/wmi_start_page.asp
- WMI-Downloads – Tools, Dokumentation, SDK <http://msdn.microsoft.com/downloads/list/wmi.asp>

oder Prozesse) sein. Beschreibungen aller bekannten Objekte werden in einer speziellen Notationssprache (MOF – Managed Object Format) vorgenommen und in kompilierter Form in einer Datenbank, dem CIM-beziehungswise WMI-Repository, gespeichert. CIM und WMI lassen sich als Abkürzungen oft austauschbar einsetzen. Wo dies der Fall ist, wird im Folgenden immer WMI verwendet.

Diese Beschreibungen werden als WMI-Klassen bezeichnet und können als eine Art Blaupause verwaltbarer Objekte verstanden werden. Innerhalb dieses Objektmodells kennt WMI auch das Konzept der Vererbung. Eine bestimmte Klasse (etwa *Win32_Process*) kann von einer ihr zugrunde liegenden Basisklasse abgeleitet sein (etwa *CIM_Process*).

An dieser Stelle möchte ich kurz einen Hinweis zu der verwendeten Terminologie anbringen: WMI und die dazugehörigen

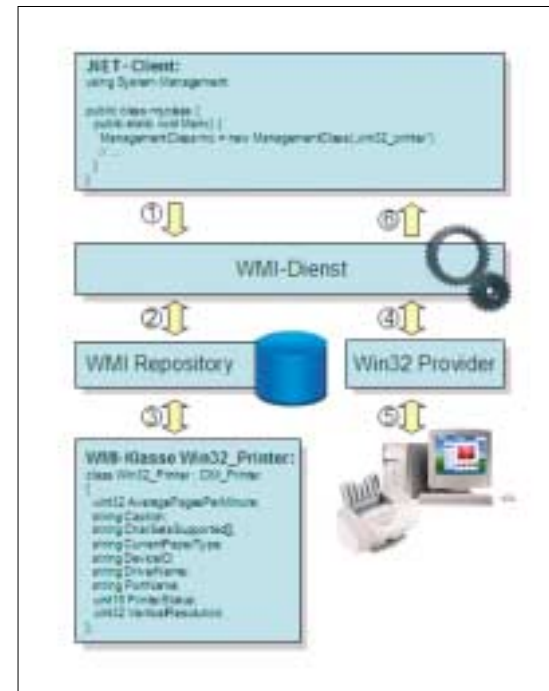


Abbildung 1 | Das Zusammenspiel von .NET-Client und WMI.

- 1 Der .NET-Client benutzt die Klassen in System.Management, um auf eine WMI-Klasse zuzugreifen.
- 2 Der WMI-Dienst nimmt die Anfrage entgegen und sucht nach der Definition der angefragten WMI-Klasse im Repository.
- 3 Die WMI-Klasse (hier vereinfacht *Win32_Printer*) ist als MOF definiert und gibt zusätzlich den Provider an, über den zugegriffen werden muss.
- 4 WMI übergibt die Kontrolle an den Provider und fordert ihn auf, Instanzobjekte passend zur tatsächlich vorhandenen Hardware zu liefern.
- 5 Der Provider ermittelt WMI-Instanzen.
- 6 WMI liefert das Ergebnis an den .NET-Client zur weiteren Verarbeitung.

.NET-Bibliotheken zu beschreiben, kann schnell verwirrend werden, da hier zwei eigenständige Objektmodelle vorliegen. Hier wie dort gibt es den Begriff der Klasse, der Instanz oder des Namensraumes. Um nicht zu viel Verwirrung zu stiften, werde ich daher im Folgenden von WMI-Klassen, WMI-Instanzen und WMI-Namensräumen sprechen, wenn ich mich auf die Management Instrumentation beziehe. Taucht einer dieser Begriffe ohne weitere Qualifizierung auf,

sind die entsprechenden Bestandteile des .NET Framework gemeint.

In einer WMI-Installation unter Windows XP finden sich mehr als 1000 WMI-Klassen im Repository. Sie decken ein sehr breites Spektrum an Objekten ab. Um in einem potenziell beliebig großen Objektmodell nicht den Überblick zu verlieren, kennt WMI, wie viele andere Objektmodelle auch, das Konzept der Namensräume, unter denen die einzelnen Klassen angeordnet

sind. Die Namensräume sind hierarchisch aufgebaut, beginnend mit *root*. Der in WMI wohl meistbenutzte Namensraum heißt *cimv2* (wird notiert als *root/cimv2*) und enthält neben den im CIM definierten Klassen zum Systemmanagement auch davon abgeleitete WMI-Klassen, welche speziell für das Windows-Management definiert wurden. Dieser Namensraum ist gleichzeitig der Standard-Namensraum und wird bei Fehlen einer anders lautenden Angabe automatisch verwendet. Die CIM-Klassen wie die Windows-Klassen lassen sich meist sehr einfach anhand ihres Namens unterscheiden. Erstere beginnen in der Regel mit *CIM_*, Letztere mit *WIN32_*. Der Unterschied zwischen Groß- und Kleinschreibung spielt bei der Notation von WMI-Klassennamen, WMI-Pfaden oder WMI-Abfragen keine Rolle.

Ist ein Objekt einmal als verwaltbare Klasse erfasst, kann im nächsten Schritt auf die Instanzen des Objekts zugegriffen werden, das heißt auf die tatsächlich im System vorhandenen Verkörperungen des mit der WMI-Klasse beschriebenen Objekts. Die Unterscheidung zwischen Klassen und Instanzen ist in WMI ein sehr wichtiges und weit gehendes Konzept. So sind beispielsweise auch WMI-Klassenbeschreibungen oder -Namensräume selbst wieder Instanzen anderer WMI-Klassen.

Informationsbeschaffung

Was jetzt noch fehlt, ist der eigentliche Zugriff. Bewaffnet mit Klassen und Instanzen kann man sich vielleicht durch ein Objektmodell bewegen; woher aber weiß eine Instanz der Klasse *Win32_Process*, dass ihr Name *notepad.exe*, ihr momentaner Speicherverbrauch 4,5 KB und ihr Threadcount 4 ist? Diese Verbindung wird in WMI mittels so genannter Provider oder Informationsanbieter hergestellt. Provider sind im Wesentlichen COM-Komponenten, die bestimmte Schnittstellen implementieren. Sie werden von WMI kontaktiert, sobald eine Information angefragt wird, für deren Beschaffung sie als zuständig hinterlegt wurden.

Abbildung 1 zeigt den generellen Ablauf. Die Zuordnung von WMI-Klassen zu einzelnen Providern wird wie die Klassendefinition selbst im Repository hinterlegt. Natürlich werden auch Provider durch WMI-Instanzen repräsentiert. Die dazu

SUMMARY

Autor
Dirk Primbs ist Technologieberater bei der Developer Plattform & Strategy Group der Microsoft Deutschland GmbH. Seine Schwerpunkte sind unter anderem verteilte Systeme und Technologien zum verteilten Systemmanagement. Er spricht regelmäßig auf Konferenzen und ist Referent beim Microsoft TechTalk. Sie erreichen ihn unter dirk@primbs.de

Eingesetzte Anwendungen
.NET Framework 1.0 oder 1.1, Visual Studio .NET 2002 oder 2003

Werkzeugkasten für WMI

Um WMI-Objekte zu finden, Abfragen zu testen oder einfache Eingriffe in das Repository durchzuführen, gibt es auf jeder WMI-Installation ein kleines Utility, das sich für kleine Experimente hervorragend eignet: `wbemtest.exe`.
Dieses sehr einfache Werkzeug bietet die Möglichkeit, nahezu jede WMI-Aktion zu testen, das Repository zu sichten oder WQL-Abfragen an WMI zu richten. Hauptvorteil des Tools ist seine universelle Verfügbarkeit auf Windows-Systemen mit WMI. Für weiter gehende Experimente gibt es eine Reihe ausgefeilterer Werkzeuge, allen voran das CIM-Studio. Download der WMI-Tools unter msdn.microsoft.com/downloads/list/wmi.asp.

gehörige Klassendefinition ist unter dem Namen `__Win32Provider` abgelegt.

Es gibt mehrere Arten von WMI-Providern, die wichtigsten sind so genannte Property-, Event- und Method-Provider. Property-Provider sind für die Beschaffung von Daten für WMI-Klassen zuständig. Method-Provider sind in der Lage, Methodenaufrufe von WMI-Klassen in Aktionen im System umzusetzen. Event-Provider schließlich spielen im Ereignissystem von WMI eine wichtige Rolle. Ein einzelner Provider kann mehrere dieser Aufgaben erfüllen.

Grundbausteine von WMI.NET

Alle .NET-Klassen, die mit WMI zu tun haben, finden sich in den Namensräumen `System.Management` und `System.Management.Instrumentation`; beide sind in der Bibliothek `System.Management.dll` zu finden. Während der erste Namensraum in der Hauptsache Funktionalitäten zur Abfrage von WMI bereithält, ist es mit dem zweiten möglich, so genannte instrumentierte Anwendungen zu schreiben, was in der WMI-Terminologie einem Provider entspricht.

Die zentralen Komponenten in WMI.NET sind – wie auch in WMI selbst – die WMI-Klasse und die dazu passenden WMI-Instanzen. Eine WMI-Klasse wird durch die .NET-Klasse `ManagementClass` repräsentiert, eine WMI-Instanz durch

WMI-Provider

WMI stellt eine Reihe vorinstallierter Provider (zum Beispiel Registry-Provider) zur Verfügung. Einzelne können wahlweise nachinstalliert werden. Insgesamt gibt es 22 Standard-Provider. Oft verwendet werden Event Log Provider, IP Route Provider, Performance Counter Provider, Ping Provider, Security Provider, Win32 Provider und der Windows Installer Provider. Der Win32 Provider ist für die überwiegende Mehrheit der Win32-Klassen zuständig und daher der meistbenutzte Provider. Einige Anwendungen haben eigene Provider, um Management-Aufgaben weiter zu vereinfachen. So hat etwa der IIS ein WMI-Zugriffsmodell und installiert einen eigenen Provider.

`ManagementObject`. `ManagementClass` kann der Einfachheit halber schon bei der Instanziierung mit dem Namen der gewünschten WMI-Klasse initialisiert werden; die Methode `GetInstances()` gibt dann eine Collection passender WMI-Instanzen zurück. Es kann also folgendes einfache Programm zur Ermittlung aller Prozesse im aktuellen System eingesetzt werden:

```
ManagementClass mgmtClass = new
ManagementClass("Win32_Process");
foreach (ManagementObject mgmtObj in
mgmtClass.GetInstances())
    Console.WriteLine(mgmtObj ["Name"]);
```

Spätestens bei diesem Dreizeiler dürfte klar werden, was das Schöne an WMI ist: Die Zugriffsmethodik bleibt bei jeder Art von Information und unabhängig von ihrer Quelle gleich. Angenommen, Sie möchten statt aller Prozesse im System die Netzwerkfreigaben ermitteln, dann müssen Sie einfach den WMI-Klassenamen in der ersten Zeile von `WIN32_Process` in `WIN32_Share` ändern. Oder es interessiert nicht der Name, sondern der aktuelle Threadcount aller Prozesse, dann verwenden Sie in Zeile drei nicht die `Name`-Eigenschaft, sondern die `ThreadCount`-Eigenschaft.

WMI im Netzwerk

Noch mächtiger wird WMI durch die Fähigkeit, über Rechnergrenzen hinweg

tätig zu werden. Um mit bigem Dreizeiler die Prozesse eines anderen Rechners zu ermitteln, muss er nur um eine WMI-Pfadangabe erweitert werden.

Alle Objekte innerhalb von WMI lassen sich über einen Objektpfad identifizieren, der in seiner absoluten Form neben dem Rechnernamen des Systems, auf dem die gefragte Objektinstanz zu finden ist, auch den Namensraum und den Namen der gewünschten WMI-Klasse enthält. Im Falle einer WMI-Instanz kommt noch der Wert der so genannten `Key`-Eigenschaft hinzu. WMI-Pfade können absolut (einschließlich kompletter Angabe des Rechners und des Namensraums) oder relativ (nur Klassenname und gegebenenfalls Wert der `Key`-Eigenschaft, Angabe bezieht sich auf den augenblicklich aktuellen Namensraum) sein. Ein vollständiger WMI-Objektpfad zum aktuellen Computersystem sähe so aus:

```
\\computername\root\cimv2:
Win32_ComputerSystem.Name="computername"
Win32_ComputerSystem.Name="computername"
```

In seiner relativen Form wird daraus

Welche Eigenschaft zur `Key`-Eigenschaft wird, ist in der MOF-Definition einer WMI-Klasse festgelegt. Diese Eigenschaft wird verwendet, um einzelne Objektinstanzen zu identifizieren. Im Falle von `Win32_ComputerSystem` ist dies die Eigenschaft `Name`. Bei einem Prozess hat `Handle` diese Funktion. Es kann auch mehr als eine `Key`-Eigenschaft definiert sein. Einzige Bedingung ist, dass spätere die Werte aller `Key`-Eigenschaften geeignet sein müssen, eine WMI-Instanz eindeutig zu identifizieren. Für WMI-Pfade findet sich in WMI.NET die Klasse `ManagementPath`. Sie können sie bei Instanziierung direkt mit einem der oben angegebenen Pfade initialisieren oder wahlweise auch alle Pfadbestandteile komfortabel als einzelne Eigenschaften der Klasse setzen.

In dem oben gezeigten Dreizeiler soll nun `ManagementPath` verwendet werden, um ein anderes Zielsystem für die WMI-Zugriffe zu definieren. Alles, was dazu getan werden muss, ist, die erste Zeile durch folgenden Code zu ersetzen:

```
ManagementPath path = new ManagementPath
("://computername/root/cimv2:Win32_Process");
ManagementClass mgmtClass =
new ManagementClass(path);
```

Um gezielt auf eine bestimmte WMI-Instanz zuzugreifen, kann ein WMI-Pfad auch direkt zur Erzeugung eines `ManagementObjects` verwendet werden:

```
ManagementObject mgmtObj = new
ManagementObject
("Win32_ComputerSystem.Name='computername'");
```

Hat das gewünschte WMI-Objekt mehr als eine `Key`-Eigenschaft, so werden die anderen Werte durch Kommatas getrennt:

```
Win32_environment.Name="Path",_UserName="SYSTEM"
```

`ManagementPath` ist nicht die einzige Klasse, die für die Darstellung von WMI-Pfaden verwendet werden kann. Es existiert eine weitere Möglichkeit: `ManagementScope`. Hier werden zusätzliche Angaben wie gegebenenfalls notwendige Anmeldeinformationen definiert und zusammen mit dem Objektpfad festgelegt. Im Minimalfall wird auch `ManagementScope` mit einem WMI-Pfad initialisiert.

Typisierte Management-Klassen

Das .NET Framework bietet dem Programmierer noch ein Werkzeug an, mit dem sich die Arbeit mit den WMI-Klassen deutlich vereinfachen lässt: typisierte Klassen für den Zugriff auf WMI-Objekte. Das Funktionsprinzip ist vergleichbar mit typisierten Datensets. Es wird einfach eine neue .NET-Klasse generiert, die sich von `ManagementClass` ableitet und alle Methoden und Eigenschaften der WMI-Klasse direkt implementiert.

Damit bekommt der Programmierer IntelliSense und Typensicherheit. Eine solche typisierte Klasse wird bei Eingabe mit dem Utility `mgmtclassgen.exe` auf der Kommandozeile erzeugt. Als Parameter wird dabei die gewünschte WMI-Klasse angegeben:

```
mgmtclassgen Win32_Process
```

Wahlweise kann eine C#-, VB.NET- oder J#-Datei erzeugt, in ein Projekt eingebunden und anschließend verwendet werden. Unser Dreizeiler zum Ermitteln von Prozessen wird auf diesem Weg sogar zum Zweizeiler:

```
foreach (process p in
process.GetInstances())
    Console.WriteLine("Name: {0},
ThreadCount: {1}", p.Name, p.ThreadCount);
```



Abbildung 2 | Die WMI-Erweiterung im Server Explorer.

Alternativ zu dem Kommandozeilen-Utility gibt es auf der Microsoft-Website (siehe Kästen *Spezifikationen und Ressourcen*) eine Erweiterung für den Server-Explorer, die typisierte WMI-Zugriffsklassen per Drag-and-Drop in das aktuelle Projekt einbindet (Abbildung 2).

WMI-Objekte managen

Für die Informationsbeschaffung ist WMI eindeutig eine große Arbeitserleichterung. Oft aber ist dies nur der erste Punkt auf der Wunschliste. Meist soll ein einmal gefundenes Objekt in irgendeiner Art und Weise geändert oder zu Aktionen veranlasst werden. Vielfach ist eine einfache Änderung einer Instanzeigenschaft der offensichtlichste Weg zum Ziel. Würde die zu ändernde Eigenschaft als `read-write`-Eigenschaft definiert, kann der Wert entweder direkt oder durch Aufruf von `setPropertyValue()` gesetzt werden.

Wichtig ist, dass nach erfolgten Änderungen ein Aufruf von `Put()` durchgeführt wird. Diese Methode sorgt dafür, dass Änderungen an WMI übermittelt und damit wirksam werden.

Folgendes Beispiel demonstriert die Vorgehensweise anhand einer Änderung der Systemvariablen `Path`:

```
ManagementObject mgmtObj =
new ManagementObject
("Win32_Environment.Name='Path',_UserName='SYSTEM'");
string path =
mgmtObj["VariableValue"].ToString()
+ "\\tools";
mgmtObj.SetPropertyValue
("VariableValue", path);
mgmtObj.Put();
```

Ist die gewünschte Eigenschaft `readonly`, oder soll weniger ein bestehendes Objekt geändert als vielmehr eine Aktion damit durchgeführt werden, so finden sich in vielen WMI-Klassen für diesen Zweck passende Methoden.

Die WMI-Klasse `Win32_Process` kennt zum Beispiel die Methode `Create()`, um neue Prozesse zu erzeugen:

```
ManagementClass mgmtClass =
new ManagementClass("Win32_Process");
mgmtClass.InvokeMethod("Create",
new Object[] {"notepad.exe"});
```

Mit WQL zum Ziel

Der eingangs gezeigte Dreizeiler zur Enumeration aller Systemprozesse hat bei aller Eleganz einen entscheidenden Nachteil: Er zwingt WMI dazu, jeden laufenden Prozess samt ihm zugeordneten Eigenschaften zu ermitteln, um dann doch nur einen einzigen Wert davon wirklich zu benutzen. Eine höchst ineffiziente Vorgehensweise, speziell wenn netzwerkweit gearbeitet wird. WMI löst dieses Problem, indem es einen Mechanismus bietet, die Anzahl der zu ermittelnden Werte bereits im Vorfeld zu limitieren. Zu diesem Zweck wurde durch die WBEM-Arbeitsgruppe eine Abfragesprache, die WMI Query Language (WQL), definiert, mit der genau solche Aggregationen möglich sind. Das Schöne daran: WQL ist eine SQL-ähnliche Sprache und damit schnell erlernbar. Eine einfache Abfrage, mit der die Festplatten eines Systems mit weniger als 3 GB freiem Speicherplatz herausgefiltert werden sollen, könnte zum Beispiel so aussehen:

```
SELECT DeviceID, FreeSpace FROM Win32_LogicalDisk WHERE FreeSpace < 3221225472
```

Die Vorteile liegen auf der Hand: WMI muss von vornherein deutlich weniger Informationen ermitteln (DeviceID und FreeSpace), und der Provider kennt die Kriterien der Auswahl im Vorfeld, was ihm Optimierungen ermöglicht. Außerdem müssen der

anfragenden Applikation insgesamt weniger Objektinstanzen übermittelt werden, was spätestens beim Zugriff über das Netzwerk wichtig ist.

In .NET wird für derartige Abfragen die Klasse *ManagementObjectSearcher* verwendet. Sie stellt das Bindeglied zwischen einer WQL-Abfrage und WMI dar. Um WQL-Abfragen wie die oben gezeigte zu definieren, stehen verschiedene Hilfsklassen zur Verfügung. Zum einen *WqlObjectQuery*, mit der nur komplette WQL-Abfragen gekapselt werden können, zum anderen eine davon abgeleitete Version namens *SelectQuery*, die typische WQL-Konstruktionen zur Instanzabfrage vereinfacht.

```
SelectQuery query =
    new SelectQuery
        ("Win32_Process", "Name = 'notepad.exe'");
ManagementObjectSearcher searcher =
    new ManagementObjectSearcher(query);
foreach (ManagementObject mgmtObj
    in searcher.Get()) {
    // ... Code ...
}
```

Auch *ManagementObjectSearcher* kann selbstverständlich mit einem Pfad dazu gebracht werden, seine Abfrage an ein beliebiges anderes WMI-System weiterzuleiten.

Objektbeziehungen

So durchgängig das bis zu diesem Punkt beschriebene Objektmodell schon ist – eine Kleinigkeit fehlt im Moment noch: die Abbildung von Objektbeziehungen. Gemeint ist die Möglichkeit, einzelne Objekte einander zuzuordnen zu können, um damit Abhängigkeiten zu modellieren. Ein Beispiel wäre etwa ein Prozessor, der zu einem bestimmten Computersystem gehört. In WMI gibt es zu diesem Zweck eine Konstruktion, die als Association bezeichnet wird. Associations sind Klassen, die Verbindungen zwischen zwei unabhängigen WMI-Objekten herstellen. So gibt es im WMI-Repository neben einer Klasse *Win32_Processor* und einer Klasse *Win32_ComputerSystem* auch eine Association-Klasse mit dem Namen *Win32_ComputerSystemProcessor*. Sie enthält eine Eigenschaft für jede der beiden Objektinstanzen, auf die sie verweist. Somit können relativ einfach zueinander in Beziehung stehende Instanzen ermittelt werden.

Weil es sich hier um eine verhältnismäßig häufig durchgeführte Aktion handelt, wurde WQL dafür um einen Befehl erweitert:

```
ASSOCIATORS OF (ObjectPath) [WHERE condition]
```

Die folgende Abfrage ermittelt die aktuellen Prozessoren im System:

```
ASSOCIATORS OF
(Win32_ComputerSystem.Name="mycomputer")
WHERE AssocClass=Win32_processor
```

Mit .NET sieht der Code folgendermaßen aus:

```
mgmtObj = new ManagementObject
    ("Win32_computersystem.name='mycomputer'");
foreach (ManagementObject relObj
    in mgmtObj.GetRelated("Win32_processor"))
    Console.WriteLine(relObj["Name"]);
```

Events – das System macht Meldung

WQL kann nicht nur für einfache Abfragen eingesetzt werden; ein ganz wesentlicher Zweck ist auch die Definition von Systemzuständen, zu denen WMI Events verschicken soll. Das WMI-Ereignismodell ist neben der normalen Abfrage von Informationen ein sehr mächtiges Feature der Management Instrumentation. WMI kann zu jedem Zustand, der sich mit WQL abfragen lässt, auch Ereignisse an die Client-Applikation liefern. Selbstverständlich kann auch hier mithilfe von Pfadangaben netzwerkweit gearbeitet werden.

Empfänger solcher Ereignisse werden als Event-Consumer bezeichnet. Sie können entweder dynamisch (nur für die Lebensdauer der Client-Applikation) oder statisch (als Komponentenregistrierung im

Repository) registriert werden. Ein Event in WMI ist – wie sollte es auch anders sein – ebenfalls eine Objektinstanz im Sinne von WMI. Passend dazu existiert für jedes mögliche Ereignis eine entsprechende WMI-Klasse. In den meisten Fällen wird auf eine der folgenden Event-Klassen zurückgegriffen:

- *__InstanceCreationEvent*
Eine Objektinstanz wurde erzeugt.
- *__InstanceModificationEvent*
Eine bestehende Instanz wurde verändert.
- *__InstanceDeletionEvent*
Eine Objektinstanz wurde entfernt.

Die WQL-Definition eines Ereignisses, das beim Start von *notepad.exe* ausgelöst wird, kann wie folgt aussehen:

```
SELECT * FROM __InstanceCreationEvent
WITHIN 5 WHERE TargetInstance ISA
'Win32_Process'
AND TargetInstance.Name = 'notepad.exe'
```

Im Idealfall hat sich in WMI für die angegebene Klasse ein Provider als Event-Provider registriert. Dann definiert *WITHIN* das Zeitintervall, in dem Events gruppiert weitergegeben werden. Ist kein solcher Event-Provider verfügbar, muss WMI das Eintreten des Ereignisses durch Vergleiche neuer Werte mit bestehenden Einträgen ermitteln. In diesem Fall gibt *WITHIN* die Zeitspanne an, in der dies zu geschehen hat. Es liegt auf der Hand, dass eine solche Vorgehensweise weniger performant als der Einsatz des Event-Providers ist. Hier müssen wirklich alle Werte immer wieder neu ermittelt werden, wohingegen ein Event-Provider deutlich selektiver vorgehen

Eigene Provider – instrumentierte Applikationen

Es ist mit den .NET-Klassen im Namensraum *System.Management.Instrumentation* möglich, so genannte „instrumentierte Applikationen“ zu erstellen. Gemeint sind hier im Grunde Property- und Event-Provider. Damit wird es möglich, eigene Applikationen auf einfache Weise durch WMI zu verwalten.

- Beispiele und Dokumentation:
- Instrumenting .NET Framework Applications
<http://msdn.microsoft.com/library/en-us/cpguide/html/cpconinstrumentingnetframeworkapplicationswithsystemmanagement.asp>
 - Monitoring in .NET Distributed Application Design
<http://msdn.microsoft.com/library/en-us/dnbdah/html/monitordotnet.asp>

Mit den derzeit in .NET verfügbaren Klassen können Property- und Event-Provider erzeugt werden. Um einen Method-Provider zu erstellen, ist Handarbeit mit COM-Schnittstellen notwendig.

kann. Das Schlüsselwort *targetinstance* ermöglicht es, sich im *WHERE*-Ausdruck auf die ermittelten ereignisauslösenden WMI-Instanzen zu beziehen. Das Schlüsselwort *ISA* schließlich kann zur Typeingrenzung verwendet werden.

In .NET ist der Event-Mechanismus von WMI mithilfe der Klasse *ManagementEventWatcher* verwendbar. Ähnlich wie *ManagementObjectSearcher* akzeptiert sie bei der Initialisierung neben Angabe von Scope beziehungsweise Pfad eine WQL-Query, um das zu empfangende Ereignis zu beschreiben. Auch hier gibt es einen spezialisierten .NET-Typ namens *WqlEventQuery*. Ereignisse können auf zwei Arten empfangen werden: synchron und asynchron. Während die erste Methode die Ausführung des Clients so lange stoppt, bis das gewünschte Ereignis eingetreten ist, gibt asynchrone Event-Bearbeitung die Kontrolle sofort wieder an den Client zurück.

Hier ein Beispiel für synchrone Ereignisbearbeitung:

```
WqlEventQuery query =
    new WqlEventQuery
        ("__InstanceCreationEvent",
         new TimeSpan(0,0,5),
         "targetinstance isa 'Win32_process'");
ManagementEventWatcher watcher =
    new ManagementEventWatcher(query);
ManagementBaseObject mbo =
    (ManagementBaseObject)watcher.
    WaitForNextEvent()["TargetInstance"];
Console.WriteLine
    ("Neuer Prozess: {0}", mbo["Name"]);
```

Am Anfang steht bei diesem Beispiel die bereits angesprochene *Event-Query*. Das *WqlEventQuery*-Objekt wird zunächst mit dem gewünschten Ereignistyp (*__instancecreationevent*), der Zeitspanne (siehe auch *WITHIN*) und auf eine Event-Bedingung (auslösende WMI-Instanz muss vom Typ *Win32_Process* sein) initialisiert. Diese Abfrage wird nun zur Initialisierung des *Event-Watchers* verwendet.

Ruft man anschließend am *Event-Watcher* die Methode *WaitForNextEvent()* auf, stoppt die Ausführung bis zum Eintreffen des gewünschten Ereignisses. *WaitForNextEvent()* liefert als Rückgabewert eine *WMI-Instanz* vom Typ *__instancecreationevent*. Die darin definierte Eigenschaft *targetinstance* referenziert die *WMI-Instanz*, die ereignisauslösend war. .NET liefert an dieser Stelle

den Typ *System.Object* zurück, was eine Typumwandlung erforderlich macht.

Stellen wir nun unser Beispiel von synchroner auf asynchrone Ereignisbearbeitung um. Zu diesem Zweck kann am *ManagementEventWatcher* dem Ereignis *EventArrived* ein Delegate zugeordnet werden. Ist dies geschehen, signalisiert man durch Aufruf der Methode *Start()* den Anfang und durch Aufruf von *Stop()* das Ende der Ereignisbearbeitung. Außerdem muss noch eine passende Methode definiert sein, in der eintreffende Events entgegengenommen werden:

```
static void Main() {
    ManagementEventWatcher watcher =
        new ManagementEventWatcher(query);
    watcher.EventArrived +=
        new EventArrivedEventHandler
            (watcher_EventArrived);
    watcher.Start();
    Console.ReadLine(); // Warten
    watcher.Stop();
}

public static void watcher_EventArrived
(object sender, EventArrivedEventArgs e) {
    ManagementBaseObject mo =
        (ManagementBaseObject)e.NewEvent
        ["targetinstance"];
    Console.WriteLine("Neuer Prozess: {0}",
        mo["Name"]);
}
```

Asynchrone Abläufe

Eine dem Ereignismechanismus ähnliche Vorgehensweise wird verwendet, wenn WMI-Abfragen lange dauern und daher ein asynchroner Empfang der Objektinstanzen wünschenswert wird. Für diesen Fall hält WMI.NET den *ManagementOperationsObserver* bereit. Er definiert eine Reihe von Ereignissen, an denen Delegates registriert werden können. Die wichtigsten sind hier *ObjectReady* (Aufruf bei einer neu empfangenen *WMI-Instanz*) und *Completed* (die gesamte Abfrage wurde abgearbeitet). Wurde ein *Observer* initialisiert, kann er beispielsweise dem *ManagementObjectSearcher* beim Aufruf von *Get()* als Parameter übergeben werden. So eingesetzt, liefert die *Get()*-Methode keine *Collection* von Objekten, sondern verwendet stattdessen die registrierten Ereignismethoden:

```
static void Main(string[] args) {
    SelectQuery query = new SelectQuery
        ("Win32_QuickFixEngineering");
    ManagementOperationsObserver observer =
        new ManagementOperationsObserver();
```

```
observer.ObjectReady += new
    ObjectReadyEventHandler
        (observer_ObjectReady);
observer.Completed += new
    CompletedEventHandler(observer_Completed);

ManagementObjectSearcher searcher =
    new ManagementObjectSearcher(query);
searcher.Get(observer);

Console.WriteLine
    ("Abwarten und Tee trinken...");
Console.ReadLine();
}

private static void observer_ObjectReady
(object sender, ObjectReadyEventArgs e) {
    Console.WriteLine("Installierter Hotfix:
    {0}", e.NewObject["Name"]);
}

private static void observer_Completed
(object sender, CompletedEventArgs e) {
    Console.WriteLine
        ("Operation abgeschlossen");
}
```

Der *ManagementOperationsObserver* kann an vielen Stellen des Objektmodells statt synchroner Aufrufe eingesetzt werden. So kann auch die Methode *GetInstances()* von *ManagementClass* wahlweise asynchron Ergebnisse liefern.

Fazit

Der Artikel hat in die Windows Management Instrumentation eingeführt und ihren Nutzen in alltäglichen Programmieraufgaben aufgezeigt. In der Literatur und in vielen einschlägigen Online-Foren werden Fragen immer häufiger mit dem Hinweis auf WMI beantwortet. Manche Fragen kann auch nur WMI auf einem befriedigenden Weg beantworten, etwa wenn es um die gesammelten Informationen in der Registry geht. (Wissen Sie, wo genau alle eingespielten Patches eingetragen werden? *Win32_QuickFixEngineering* weiß es.) In jedem Fall hat WMI den Vorteil einer einheitlichen Herangehensweise an ein breites Spektrum von Informationen.

Ein weiterer Aspekt, der den Rahmen dieses Artikels gesprengt hätte und daher nur angedeutet werden soll, ist auch nicht zu unterschätzen: die Instrumentierung eigener Applikationen. Mit .NET wird es sehr einfach, neben der Abfrage- auch die Anbieterseite von WMI zu implementieren und damit eigene Applikationen verwaltbar zu machen. Der Kasten *Eigene Provider – instrumentierte Applikationen* enthält Links zu weiterführenden Informationen. |||||