

Start to Finish Guide to MOF Editing

The Definitive Guide to Systems Management Server Hardware Inventory Customization

```
SMS_DEF.Mof change detected $$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:37.018
2005 Eastern Daylight Time><thread=2868 (0xB34)>~Connected to SQL; waiting for Hinv action
ID...$$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:37.018 2005 Eastern Daylight
Time><thread=2868 (0xB34)>~Done with wait for Hinv action
ID.$$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:37.018 2005 Eastern Daylight
Time><thread=2868 (0xB34)>Start of cimv2\sms-to-policy conversion
$$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:37.018 2005 Eastern Daylight
Time><thread=2868 (0xB34)>Resetting SMS_Report qualifier to FALSE on all
classes and properties in cimv2\sms namespace $$<SMS_INVENTORY_DATA_LOADER>
<Thu Aug 04 08:40:37.018 2005 Eastern Daylight Time><thread=2868 (0xB34)>
Running MOFCOMP on D:\SMS\inboxes\clifiles.src\hinv\sms_def.mof
$$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:37.128 2005 Eastern Daylight
Time><thread=2868 (0xB34)>MOF backed up to D:\SMS\data\hinvarhive\
sms_def.mof.bak~ $$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:37.784 2005 Eastern
Daylight Time><thread=2868 (0xB34)>End of cimv2\sms-to-policy conversion;
returning 0x0 $$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:47.097 2005
Eastern Daylight Time><thread=2868 (0xB34)>SMS_DEF.Mof change detected
$$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:37.018 2005 Eastern Daylight
Time><thread=2868 (0xB34)>~Connected to SQL; waiting for Hinv action ID...
$$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:37.018 2005 Eastern
Daylight Time><thread=2868 (0xB34)>~Done with wait for Hinv action ID.
$$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:37.018 2005 Eastern
Daylight Time><thread=2868 (0xB34)>Start of cimv2\sms-to-policy conversion\
$$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:37.018 2005 Eastern Daylight
Time><thread=2868 (0xB34)>Resetting SMS_Report qualifier to FALSE on all
classes and properties in cimv2\sms namespace $$<SMS_INVENTORY_DATA_LOADER>
<Thu Aug 04 08:40:37.018 2005 Eastern Daylight Time><thread=2868 (0xB34)>
Running MOFCOMP on D:\SMS\inboxes\clifiles.src\hinv\sms_def.mof
$$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:37.128 2005 Eastern Daylight
Time><thread=2868 (0xB34)>MOF backed up to D:\SMS\data\hinvarhive\
sms_def.mof.bak~ $$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:37.784 2005 Eastern
Daylight Time><thread=2868 (0xB34)>End of cimv2\sms-to-policy conversion;
returning 0x0 $$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:47.097 2005
Eastern Daylight Time><thread=2868 (0xB34)>lasses and properties in cimv2\sms namespace
$$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:37.018 2005 Eastern Daylight
Time><thread=2868 (0xB34)>Running MOFCOMP on D:\SMS\inboxes\clifiles.src
\hinv\sms_def.mof$$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:37.128 2005 Eastern
Daylight Time><thread=2868 (0xB34)>MOF backed up to D:\SMS\data\hinvarhive\
sms_def.mof.bak~ $$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:37.784 2005
Eastern Daylight Time><thread=2868 (0xB34)>End of cimv2\sms-to-policy conversion;
returning 0x0 $$<SMS_INVENTORY_DATA_LOADER><Thu Aug 04 08:40:47.097 2005
Eastern Daylight Time><thread=2868 (0xB34)>
```

I would like to dedicate this book to
my beautiful wife Christine who
always encourages me to believe in myself.

CHAPTER 1: BOOT CAMP.....	1
WHAT IS WMI?	1
HOW DOES HARDWARE INVENTORY WORK?	3
WHAT ARE MOF FILES?.....	6
CHAPTER SUMMARY	9
CHAPTER 2: INTRODUCING THE SMS_DEF.MOF	11
THE MENU TO HARDWARE INVENTORY	11
A LOOK INSIDE THE SMS_DEF.MOF.....	13
DECLARATIONS.....	13
CLASS LEVEL REPORTING PROPERTIES	14
FIELD LEVEL REPORTING PROPERTIES.....	14
SPECIAL ORDERS	16
CHAPTER SUMMARY	21
CHAPTER 3: SMS_DEF.MOF SYNTAX	23
COMPILING WITH MOFCOMP	23
COMMENTS	25
DEFINITIONS	27
NAMESPACES	27
DATA CLASSES	28
REPORTING CLASSES	29
INSTANCES	29
PROVIDERS	30
SMS_DEF.MOF STRUCTURE RECAP	30
BASIC MOF MODIFICATION STEPS	31
CHAPTER SUMMARY	36
CHAPTER 4: INTRODUCTION TO WMI MANIPULATION	37
THE BASIC STRUCTURE OF WMI	37
WMI DATA CLASSES.....	39
KEY FIELDS	40
USING TOOLS TO ACCESS WMI.....	41
WBEMTEST.....	41
CIM STUDIO	46
SCRIPTOMATIC VERSION 2.....	47
WMI CODE CREATOR VERSION 1	50
WMI MANIPULATION METHODS	52
CHAPTER SUMMARY	55
CHAPTER 5: REPORTING ON AN EXISTING CLASS.....	56
STRUCTURE OF A REPORTING CLASS	58
CHAPTER SUMMARY	68
CHAPTER 6: REGISTRY PROPERTY PROVIDER	70
WHEN TO USE THE REGISTRY PROPERTY PROVIDER	70
CHAPTER SUMMARY	79
CHAPTER 7: THE REGISTRY INSTANCE PROVIDER.....	80
WHEN TO USE THE REGISTRY INSTANCE PROVIDER	80

CHAPTER SUMMARY	87
CHAPTER 8: USING THE VIEW PROVIDER.....	88
ACCESSING NAMESPACES OTHER THAN <i>ROOT\CIMV2</i>	88
USING WQL QUERIES TO FILTER WMI INFORMATION	95
CHAPTER SUMMARY	96
CHAPTER 9: STATIC HARDWARE INVENTORY EXTENSIONS	97
STATIC MOF FILES	97
NOIDMIF AND IDMIF FILES IN GENERAL	99
NOIDMIF FILES	101
IDMIF FILES	104
CHAPTER SUMMARY	106
CHAPTER 10: SCRIPTED HARDWARE INVENTORY EXTENSIONS	107
STATIC FILE SCRIPTED EXTENSIONS	107
SCRIPTS THAT WRITE DIRECTLY TO WMI	117
CHAPTER SUMMARY	119
CHAPTER 11: IT'S BETTER TO TEST NOW THAN BE TESTY LATER	120
VERIFY THE MOF SYNTAX WITH MOFCOMP	122
COMPILE THE MOF ON A TEST MACHINE	125
USE WBEMTEST TO CHECK FOR THE CLASS	126
INITIATE A HARDWARE INVENTORY	128
VERIFY THE HARDWARE INVENTORY PROCESS	129
VERIFY THE DATA ON THE SMS SITE SERVER	130
CHAPTER SUMMARY	132
CHAPTER 12: PULLING THE TRIGGER.....	133
BACK UP THE ORIGINAL SMS_DEF.MOF	133
REPLACE THE ORIGINAL SMS_DEF.MOF ON THE SITE SERVER	133
UPDATE THE SITE HIERARCHY	133
SMS_DEF.MOF UPDATES: BEHIND THE SCENES	134
SMS_DEF.MOF BACK UPS: GOOD AND BAD	135
USING A MINI-MOF	136
CHAPTER SUMMARY	137
CHAPTER 13: CLEANING UP	138
CLEAN UP THE SMS_DEF.MOF ON THE SITE SERVER	139
REMOVE THE UNNECESSARY WMI CLASSES FROM YOUR CLIENTS	139
REMOVE THE CLASS DATA FROM THE DATABASE	141
DELGRP	141
SMS EXPERT'S SITE SWEEPER UTILITY	146
CHAPTER SUMMARY	149
CHAPTER 14: TROUBLESHOOTING AND TIPS.....	150
LOG TAG	150
MOF EDITING ERRORS	155
MOFCOMP ERRORS	157
SQL DATABASE ERRORS	158
TROUBLESHOOTING WMI	159

WMI DIAG	162
SOME TIPS AND OTHER SILLY MOF TRICKS.....	162
CREATE CUSTOM REMINDERS	162
BE #PRAGMATIC.....	163
BYPASS MOFCOMP	164
THE MIN MONSTERMOF.....	165
MORE TIPS AND TRICKS.....	166
CHAPTER SUMMARY	168
CHAPTER 15: MOF EDITING IN 15 MINUTES.....	170
INTRODUCING THE SMS_DEF.MOF.....	170
WINDOWS MANAGEMENT INSTRUMENTATION	171
HARDWARE INVENTORY PROCESS	172
MODIFYING THE MOF	173
REPORTING ON AN EXISTING DATA CLASS	174
REGISTRY PROVIDERS.....	175
THE VIEW PROVIDER	178
STATIC HARDWARE INVENTORY EXTENSIONS.....	179
SCRIPTED HARDWARE INVENTORY EXTENSIONS	181
COMMENTS	182
COMPILING THE MOF.....	183
DISTRIBUTING MOF UPDATES.....	184
CHAPTER SUMMARY	185

Foreword

Hardware inventory sometimes seems to me to be a "forgotten" feature of SMS. It can be argued that the big gun functionality of SMS lies in its software distribution and software update ability. Certainly there is a lot of discussion around distribution and update, and certainly a greater amount of support. Nevertheless, hardware inventory is still a workhorse feature of SMS. With it you can collect a wealth of information about the computers you manage with SMS, especially if you take the time and effort to learn how to customize it. Interestingly, as Jeff points out in his introduction, hardware inventory customization is indeed a mysterious world. With the demise of the SMS Resource Kit and the MOF Editor, content about how to customize hardware inventory became harder to find. Community forums like myITforum and third party developers like SMS Expert provide a valuable source of field experience to be sure, but, speaking as an author, there is nothing like a good book to curl up with in front of that SMS Administrator's Console. Jeff Gilbert has mined several excellent resources, including his own experience, to collect that useful information into just such a book. This work demystifies hardware inventory customization with practical examples and step-by-step guidance. And it's easy to read. Jeff is a talented writer (or he wouldn't have been recruited to write about SMS for Microsoft). I recommend that every SMS administrator keep this one in your tool bag or bookmarked on your favorites list.

Steve Kaczmarek

Author, Microsoft Systems Management Server 2003 Administrator's Companion (MS Press, 2003)

Introduction

In 2002, Michael Schultz’s eye-opening MOF editing guide took the SMS world by storm. To this solid base, I’ve added my own insights into the mysterious world of SMS hardware inventory customization.

This current version of the guide is the culmination of the hard work of many people. In particular, my thanks go out to Michael Schultz, Garth Jones, any previous and future “MOF Masters” out there and last, but certainly not least, Microsoft, myITforum, Inc., and the rest of the SMS community at large who all strive for success one SMS administrator at a time.

The ultimate aim of this current incarnation of the MOF Editing Guide is to set up SMS administrators for success in modifying SMS hardware inventory. The guide begins with getting to know the basics of WMI and the hardware inventory process itself. Next, I’ve explained each of the various methods for modifying hardware inventory in as much detail as I could fit between the margins of the printed page. The end result is an informative guide to modifying hardware inventory that I’ve also tried to make fun to read.

When learning how to modify SMS hardware inventory, you may have days when you will wish that you had taken up a different vocation. You may also find yourself alienated by loved ones threatening to leave if you don’t stop the recurring late night primal scream therapy sessions with your laptop.

Do not be discouraged by any early, failed attempts at modifying hardware inventory. With the help of this guide, modifying the SMS_def.mof will be as easy as sending an email to an old friend—only, in this case, instead of asking “Anything new going on back home?” you’re asking “So, anything new with the Win32_Desktop class?”

“I am always doing that which I can not do, in order that I may learn how to do it.”

–Pablo Picasso

Chapter 1: Boot Camp

Introduction to WMI, SMS Hardware Inventory, and MOF Files

Before spending hours writing about how to instantiate a data and reporting class utilizing the property provider to enable SMS to collect registry data from the WMI repository of your clients...

I should probably go over what in the heck all of that means. Right?

Don't worry. By the time you're finished reading this book, you too will be able to say confusing phrases in front of your boss and coworkers. In fact, you'll not only be able to say them, but you'll actually understand what they mean!

There are three topics to discuss before we get this party started:

- What is WMI?
- How does the SMS hardware inventory work?
- What are MOF files?

What is WMI?

Founded in 1992, the Distributed Management Task Force (DMTF) is the industry organization leading the development of management standards and integration technology for enterprise and Internet environments. In 1996, the DMTF created the Web-Based Enterprise Management (WBEM) initiative. This initiative is a common interface for applications and operating systems to access key enterprise data from their client systems and hardware components. The Microsoft implementation of WBEM is Windows Management Instrumentation, better known as WMI.

WMI is preinstalled in Windows Server 2003, Windows XP, Windows ME, and Windows 2000. For Windows NT 4.0 SP4 and later, WMI is available through "Add/Remove Windows Components" in Control Panel as WBEM option install. WMI 1.5 is available for Windows 95/98 as well as a later, more comprehensive,

version for Windows NT 4.0 is available as an Internet download from the Windows download site: Windows Management Instrumentation (WMI) CORE 1.5 (Windows 95/98/NT 4.0):

<http://www.microsoft.com/downloads/details.aspx?FamilyID=afe41f46-e213-4cbf-9c5b-fbf236e0e875&DisplayLang=en>



WBEM: “web-um” Web-Based Enterprise Initiative - a common interface for applications and operating systems to access data from their client systems and hardware components.

WMI: “doubleyou-em-eye”. Windows Management Instrumentation - Microsoft’s implementation of the WBEM initiative.

On computers that are running either Windows XP or the Windows .NET Framework, the build version for WMI is the same as the build version for the operating system. The version of WMI also reflects the version of the operating system. For example, the version of WMI that is included with Windows XP is 5.1.2600.0.

Be aware that some Windows 95/98 clients may have issues being upgraded to WMI 1.5 if their DCOM has not been upgraded to version 1.3 first. DCOM for Windows 95/98 can be downloaded here: <http://www.microsoft.com/com/default.msp>

So what is WMI good for? WMI holds the answer to any and all questions ... about a computer, that is. It resides in the bowels of a Windows based system and is constantly alert to the condition and nature of its surroundings.

If you’re looking for information such as: BIOS version, CPU speed, performance monitoring data, manufacturer of the CD-ROM, or even the location and path of every single shortcut on a particular system, the system’s WMI repository is the place to go.

What about my RAM? Can I find information on that in the WMI? Certainly. You can find how out much RAM you have, how many physical chips are present, and in which physical slots those chips can be found. As I said, WMI is meant to be a one-stop shop for any and all information about a computer system.

Some manufacturers create their own applications that access hardware data, such as the HP Client Management Interface (HP CMI). But, what if you have HP/Compaq and IBM machines in-house? Now you will need *two* tools to access the data. You can see where this is going.

In a perfect world, every computer in your company would be identical, from the operating system all the way down to the smallest jumper. However, in real life, the variety of systems would easily rival Baskin Robbins’ 31 flavors.

Older devices and components may not be registered correctly for WMI to access the data. However, most new devices provide an abundance of information.

How exciting would the world be if every car was the same color and same shape? That wouldn't be very exciting right? What Microsoft did was to develop the internal combustion engine in WMI. So, when creating a new application, software developers apply Microsoft's basic designs to run their "car", but they are free to make it look any way that they want. In other words, Microsoft created WMI as a tool to enable data storage and extraction from within a framework that is machine independent.

So WMI will let me access my hardware data, no matter who made the machine? Right. The manufacturers of your CD-ROMs, hard drives, motherboards, video cards, and most other components, have agreed to abide by the initiative created by the DMTF. If they have placed the data in the correct location, in the proper way, WMI will be able to access it.



WMI is meant to be ahead of its time. Often there are classes in WMI that hardware inventory can collect, but the hardware manufacturers do not have the data in place to be collected.

So if I want this hardware data, I have to access WMI to get it. Are there any tools I can use to access WMI? There are many tools to access WMI. Among the most common are WBEMTEST.EXE (located in your wbem directory as %WINDIR%\system32\wbem directory) and [CIM Studio](#). CIM Studio comes with the WMI Administrative Tools, which can be downloaded from MSDN: <http://www.microsoft.com/downloads/details.aspx?FamilyId=6430F853-1120-48DB-8CC5-F2ABDC3ED314&displaylang=en>. However, there are many applications that have hooks into WMI which enables them to access the data contained therein.

And if that was not enough WMI information to get you started, don't worry. I'll go into a lot more about WMI, how it relates to SMS, and how to manipulate it to serve the forces of good in Chapter 4.

How Does Hardware Inventory Work?

SMS is an excellent tool for gathering hardware data from client machines. Instead of having a unique method of extracting data from each different system, the hardware manufacturers have stored all of their data in such a way that it can be accessed by WMI and SMS hardware inventory.

Enabling hardware inventory causes SMS to install the appropriate agent components on Legacy Clients, and enables, or turns on, the pre-installed inventory agent for Advanced Clients. Once installed or enabled, the hardware inventory agent starts collecting data on a schedule you specify. Technically, the agents involved are the hardware inventory agent for Legacy Clients and the inventory agent for Advanced Clients.



I'm going to go out on a limb here and assume that you know how to enable hardware inventory and set the schedule for it to run since you're reading an advanced inventory book 😊

Maybe a quick explanation of the flow of things will help you out here.

The inventory process for Legacy Clients and Advanced Clients is fairly similar, but different enough to require separate explanations.

Legacy Clients. Ahhh, old friend how I miss your simplicity ... but I digress. Legacy Clients follow the procedure below to get all their juicy bits into the SMS database:

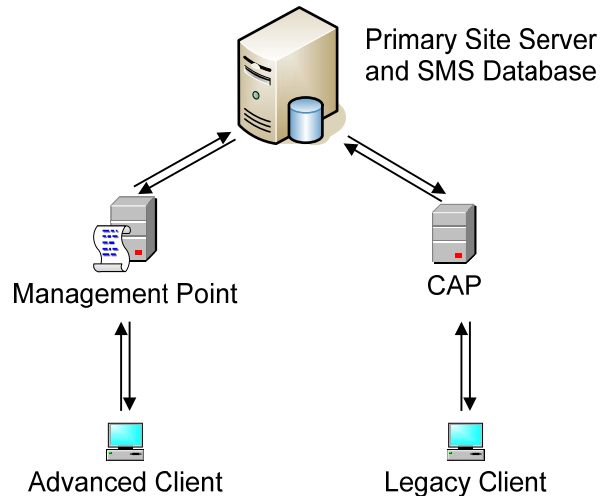
- The hardware inventory agent component queries the local system WMI for **X**, **Y**, and **Z** data, which it then places into a .MIF file
- This file is copied to the client access point (CAP).
- The CAP then copies the .MIF file to the SMS Site Server.
- The data loader service on the site server parses the .MIF file and enters the data into the appropriate tables in the SQL database.
- The workstation's hardware inventory data can now be viewed and queried in the SMS database and resource explorer.

Advanced Clients. The Advanced Client is a little pickier. A stickler for administration, it requires an official inventory policy from management to follow for conducting inventories.

- The SMS policy provider creates an Advanced Client policy for conducting hardware inventory based on the SMS_def.mof stored on the site server.
- Advanced Clients retrieve this policy from their management point at their next scheduled client policy refresh interval (by default, this is once per hour).
- The Advanced Client dutifully conducts hardware inventory by poking around the local system WMI according to the policy instructions. It then forwards the all important **X**, **Y**, and **Z** data back to the management point in .XML format.
- The management point completes the loop by sending the inventory results back to the SMS site server which squirrels away the data in the SQL database.
- The workstation's hardware inventory data can now be viewed, and queried in the SMS database, and resource explorer just like the Legacy Client.

Just to reinforce this process, here's a simple diagram in Figure 1.1.

Figure 1.1



The initial inventory for both client types is a full hardware inventory. This establishes a baseline for future inventories. After the initial hardware inventory, the subsequent inventories usually only send changes, or deltas, to their respective supervisors—the client access point for Legacy Clients and the management point for Advanced Clients.

This *delta inventory* is a good idea. I mean, how many times do you need a client to report that the same hardware is installed? Not to mention the reduction in redundant data packets traveling around your network and slowing down your ability to surf the Internet to check on your favorite football team's stats.

So what happens if a client isn't connected to the network during the next hardware inventory cycle? The Advanced Client is designed to provide inventory data when the client has a non-continuous network connection. This means that hardware inventory still goes on as scheduled, but the actual inventory report isn't sent up until the client's network connection has been restored.

Now, you will notice above that the hardware inventory component queries WMI for **X**, **Y**, and **Z** data to place into an inventory results file (either .MIF or .XML). The question of the week: What if you, as an SMS administrator, actually want **W**, **Q**, **R**, **X**, **L**, **M**, **P**, and a little bit of **Ω**?

The answer: SMS uses a .MOF file to search for those needles in the haystack. This file is named SMS_def.mof and is found in the %WINDIR%\MS\SMS\Sitefile*<site code>*\Hinv folder on Legacy Clients. For Advanced Clients, the SMS_def.mof file is stored in the SMS\INBOXES\CLIFILES.SRC\HINV directory on the site server.

What Are MOF Files?



A .MOF (Managed Object Format) file is simply a text file with the data contained within stored in a fairly simple to understand format.

MOF: ‘mawf’ Managed Object Format – The MOF file is a set of instructions and locations from where data should be extracted from WMI.

So, long story short, hardware inventory is configured by using one of these funky files—the SMS_def.mof. This file is just one of the many files copied to the SMS 2003 Legacy Client when a client install occurs. The Advanced Client receives a policy generated by the site server’s policy provider based on the SMS_def.mof stored on the primary site server. Think of the data inside the SMS_def.mof file as a roadmap for the hardware inventory to follow as it drives around your systems.

This fancy text file includes the WMI location of the attributes you’re after with hardware inventory and gives the inventory agent instructions on how to get to it. SMS can read this file because the text inside is structured in such a way that when it is compiled (either by the Legacy Client itself or the policy provider for Advanced Clients) the data contained within is added to the local WMI.

What do you mean “compile”? As stated earlier, a MOF file is simply a text file with a .MOF extension. Having the .MOF extension on the file signifies that there is data inside that can be added to WMI. A tool which is present on every Windows computer, called MOFCOMP.EXE, can be used to compile this MOF file into a machine readable format, and add the data to the WMI repository of SMS client systems.

In the following example, the data inside TESTFILE.MOF will be added to the local WMI on the workstation:

```
C:\winnt\system32\wbem\MOFCOMP testfile.MOF
```

Can I put just anything in the MOF file and add it to WMI? Sorry, no. If it was that simple, there’d be no need for this book. The data in the MOF file must be structured exactly right for MOFCOMP to know how and where to put it in the WMI. The syntax and details of this will be discussed later.

So the SMS_def.mof file tells the hardware inventory what to collect? Almost. The data *inside* the file is what is *indirectly* used to guide the hardware inventory. The hardware inventory queries WMI for the data it should collect; and then **she** queries WMI *again* to actually collect that data.

What do you mean “she”? Well, the hardware inventory *has* to be female. She makes sure to ask for directions.

That was poor. I know. Sorry.

Seriously, though, the hardware inventory is *unaware* of what data it is supposed to collect until that data is entered into the WMI by compiling the SMS_def.mof.

On SMS 2.0 and SMS 2003 Legacy Client, the hardware inventory first checks to see if the SMS_def.mof file that was copied from the site server to the client access point is newer than the *current* SMS_def.mof file on the local system. During client installation, because there is no current file, the SMS_def.mof is automatically compiled; then copied into the `%WINDIR%\ms\SMS\cli\files\hinw` directory, thereby becoming *current*. The SMS_def.mof file is only compiled if the version from the site server is newer than the current version on the local machine.

When an SMS administrator is interested in changing what the hardware inventory collects from their clients, they must change the SMS_def.mof file, copy it to their site server, and then this becomes the *new* SMS_def.mof file. This new SMS_def.mof is copied to the client access points. During their client refresh interval the Legacy Clients will recognize that there is a new SMS_def.mof file located on the client access point and download and compile it.

For Advanced Clients, when a modified SMS_def.mof is placed on the site server, it is automatically compiled, and if it is syntactically correct, a new inventory policy is created. This new policy is sent to the management points to update the client inventory policy at their next policy request interval. This works fine as long as there hasn't been any major changes introduced in your modified SMS_def.mof file.

What do you mean major changes? By major changes I mean the introduction of a new provider, or an additional class not in the system's WMI repository by default. You will need to follow the steps below in order to compile the new inventory information from the modified SMS_def.mof for the Advanced Client.



If these steps aren't followed correctly, the Advanced Client will not be able to report new inventory information based on your modified SMS_def.mof file.

In order to update the advanced client's WMI repository to reflect the new classes you are after, you need to manually compile the SMS_def.mof *on each and every Advanced Client in your hierarchy*.

- Copy the new SMS_def.mof to the client's temp directory (c:\temp)
- Execute `MOFCOMP c:\temp\SMS_def.mof`

Additionally, you will need to create an SMS program that performs these tasks and advertise it to run on a schedule for current or future Advanced Clients that have missed these steps.

I really want all of the data from my client machines. Do I still need the SMS_def.mof file if I just want everything? Well, first off, yes. Remember, that hardware inventory doesn't know what it's supposed to collect until the SMS_def.mof file is compiled and places the roadmap into WMI.

Secondly, as previously noted, WMI contains information about almost *everything* on the client. That means you can get information on how fast the fans are spinning inside the client machines, how many volts are being used by the power supply, and how hot the processor was at the time of the inventory. That's a heckuva lot of data!

Think about WMI as a big yard sale. Now, you *could* pick up everything there for \$17.25 and fill your basement with junk, or you can pick and choose which items you will actually use. It's your call, but it is usually best to be selective about what data you collect in order to put as little strain on your client machines, network, and site server as possible.

Strain on my clients, network, and site server? How much of a strain? Okay, perhaps strain is too strong of a word. The impact of the hardware inventory on typical business machines in this day and age is almost negligible, as is the impact on the site server and network. Maybe the greater concern would be the enormous haystack of useless data one would be required to wade through to find the needle of good information.



Chapter Summary

WMI

- In 1996, the [Distributed Management Task Force](#) created the Web-Based Enterprise Management initiative (WBEM). The Microsoft implementation of this initiative is Windows Management Instrumentation or WMI.
- **WMI is a management technology built into Windows that allows you to access system resource information** including hard drives, operating system information, services, registry settings, and pretty much whatever else you want to know about a computer.



Hardware Inventory

The inventory process for Legacy Clients and Advanced Clients is fairly similar, but different enough to require separate explanations.

- **Legacy Clients.** Legacy Clients maintain their own local copy of the **SMS_def.mof file**. They report hardware inventory information to a client access point which forwards it to the SMS Site Server and eventually to the SMS database.
- **Advanced Clients.** Advanced Clients receive an inventory policy from a management point based on the contents of the **SMS_def.mof file**. Using this policy, the Advanced Client conducts hardware inventory and reports the resulting data back to the management point. The management point sends inventory report to the primary SMS site server and SQL database tables.

The initial (and complete) inventory for both client types establishes a baseline for future inventories. After the initial hardware inventory, the subsequent inventories usually only send changes—or deltas.

Managed Object Format (MOF) files are text files and should be viewed with NOTEPAD.EXE. SMS can read these files because the text inside is structured in such a way that when it is compiled (either by the Legacy Client itself or the policy provider for Advanced Clients) the data contained within is added to the local system WMI.

SMS clients use the SMS_def.mof file to determine what information will be collected during the next hardware inventory in different ways.

- Legacy Clients *store a local copy* of the SMS_def.mof file in the %WINDIR%\ms\SMS\clifiles\binv folder.
- Advanced Clients *retrieve an inventory policy* based on the SMS_def.mof stored on the site server in the SMS\inboxes\clifiles.srv\binv share.

Whenever the SMS_def.mof file is changed, SMS loads its contents into the SMS database so that Advanced Clients can request them as policy from the management point. The SMS_def.mof is also downloaded to client access points so that Legacy Clients can acquire it.

Introducing a new provider, or additional class not in the system's WMI by default, requires that the SMS_def.mof be manually re-compiled for the Advanced Client.

To manually re-compile the SMS_def.mof on Advanced Clients to enter new information into the local system's WMI follow these steps:



- Copy SMS_def.mof to the system's temporary directory (c:\temp).
- Execute *MOFCOMP c:\temp\SMS_def.mof*.
- Create an SMS program that performs these tasks and advertise it to run on a schedule for current or future Advanced Clients that have missed these steps.



WBEM: “web-um” Web-Based Enterprise Initiative - a common interface for applications and operating systems to access data from their client systems and hardware components.

WMI: “doubleyou-em-eye”. Windows Management Instrumentation - Microsoft's implementation of the WBEM initiative.

MOF: “mawf” Managed Object Format – The MOF file is a set of instructions and locations from where data should be extracted from WMI.

Chapter 2: Introducing the SMS_def.mof

What is the SMS_def.mof file, and why do I care?

Modifying the SMS_def.mof file, A.K.A. “the MOF”, is a long and tedious process, often requiring hours of preparation and implementation for even the slightest modification. In fact, it’s pretty dangerous. So, before changing the MOF, I would update your resume.

I’m just joking! Making changes to the MOF is surprisingly simple, as long as you follow the proper steps. After a bit of practice, an experienced admin should be able to modify or even add new classes to the MOF and deploy it site-wide very quickly.

Chapter 1 offered a brief introduction to WMI, SMS hardware inventory, and MOF files in general. In this chapter, we’ll cover the role of the MOF file and its relationship to SMS hardware inventory in depth. As our examination of the MOF becomes more detailed, so will the analogies.

The Menu to Hardware Inventory

The SMS_def.mof file is more than just a roadmap for hardware inventory. It’s more like placing an order at your favorite restaurant. There is an entire menu of items to select from, but there are a limited number of those items you actually want. The MOF file is the order you’re placing at W.M.I.Fridays. Except, instead of ordering extreme fajitas or smoked chicken, you’re ordering Win32_Processor and Win32_VideoController data.

When you enter W.M.I. Fridays, you’re given a menu and asked to select the items you want. By default, if you do not select the items, that means you don’t want them. Also, just like a real restaurant, you can even request “special orders”.

I think I’ll order today and I’ll have...Win32_Fan for an appetizer, Win32_VideoController for the main course, and for desert...the scrumptious Win32_SystemEnclosure! I do *not* want Win32_Refrigeration because it gives me frightful headaches, and Win32_SerialPort gives me heartburn.

When I'm done with my order, I turn it in to W.M.I. Fridays. In other words, when you're done editing your SMS_def.mof file, the information is entered into WMI on the client machine.

How does the information in the SMS_def.mof file get entered into the client's WMI? If you recall from Chapter 1 when the MOF file is compiled the information contained within gets added to the local system WMI. Also remember that when a new MOF needs to be deployed you must use the appropriate method for each type of SMS client. This is done to make sure that hardware inventory is collecting exactly what the administrator wants to collect.

If I don't change anything in the SMS_def.mof file, am I turning in a blank order? Actually, no. You see, W.M.I. Fridays is owned by Microsoft Foods. They selected some of the more useful items, and added them to *everyone's* order, so it automatically has those selected on the default menu. To keep from getting the items, you have to make it a point to remove them from your order.

In other words, they reasoned that there were certain items from WMI that *most* administrators will want to collect, and it would be easier to "pre-order" them when SMS is installed, instead of making a book, such as this one, required reading before getting any useful data.

So in my default SMS_def.mof file, I already have things ordered for me? That's not cool. I must disagree. Most of the items that Microsoft selected are extremely valuable. Without changing the SMS_def.mof file at all, your site will contain the processor speed, manufacturer, CD-ROM type, hard disk size, and many other useful pieces of information from each of your clients. If you ask me, Microsoft could have selected *more* items to "pre-order".

As you can imagine, the SMS_def.mof file can be extremely valuable to an administrator tasked to collect even the most obscure data from their client machines. The SMS_def.mof file allows the administrator to:

- Order an item from WMI.
- Not order an item from WMI.
- Special request an item from WMI.

Unfortunately, the process of editing the SMS_def.mof file cannot be all fun and games, but I'll do my best to entertain you during the most boring parts. Starting ... now.

A Look Inside the SMS_def.mof

When opened in notepad, the SMS_def.mof file can appear formidable. If you examine every character, every word, or even every phrase, it can seem like a swarm of bees that should not be disturbed. “Leave it alone” is what you may tell yourself the first time you look at it. But don’t be afraid. The MOF is more like a litter of kittens. Not much to be scared of, but you still need to be careful.



Notepad is the preferred editor to avoid adding hidden characters into the file as some text editing programs may do.

There are two pieces of the SMS_def.mof file; the tiny top 1% of the iceberg, and the bottom 99% that actually should concern you most days. The top 1% declares namespaces and providers. However, if you scroll through the rest of the file, you will see the real meat of the MOF.

The bottom section consists of blocks of code which are called reporting classes. These reporting classes are entered into WMI when the MOF is compiled, and ultimately read by the hardware inventory process to determine what information is to be collected and sent back to the SMS database.

Declarations

In the below example the text in red is the iceberg—the declarations, the black text is a partial example of a reporting class. Notice the top portion defines the namespaces where the data resides, and the providers used to access it. The reporting classes contain information within those namespaces that the provider uses to get the data.

```
//=====
///SMS_def.mof - Maps SMS inventorable set to that provided by
//the WBEM CIMV2 Win32 Provider - version 1085
///Copyright (c) Microsoft Corporation, All Rights Reserved
//=====

//=====
//Create namespaces used by the Inventory Agent
//=====

#pragma namespace ("\\.\root\CIMV2")
instance of __Namespace
{
    Name = "SMS" ;
};

#pragma namespace ("\\.\root\CIMV2\SMS")
class SMS_Class_Template
{
};
```

```

//=====
// Register the view provider in the SMS namespace
// Refer to WMI SDK documentation for use
//=====

#pragma namespace ("\\\\.\\root\\CIMV2")
instance of __Win32Provider as $ViewProv
{
    Name = "MS_VIEW_INSTANCE_PROVIDER";
    Clsid = "{AA70DDF4-E11C-11D1-ABB0-00C04FD9159E}";
    ImpersonationLevel = 1;
    PerUserInitialization = "True";

};
.
.
.
.
[ SMS_Report (TRUE),
  SMS_Group_Name ("BIOS"),
  SMS_Class_ID ("MICROSOFT|PC_BIOS|1.0") ]

class Win32_BIOS : SMS_Class_Template
{
    [SMS_Report (FALSE) ]
    uint16 BiosCharacteristics[];
    [SMS_Report (TRUE) ]
    string BuildNumber;
};

```

Class Level Reporting Properties

Each block of code represents a menu item that you can order from W.M.I. Fridays. At the very top of each block is what is called a **class level reporting property**. This line looks like either “[SMS_Report(FALSE)” or “[SMS_Report(TRUE)” and it tells hardware inventory if that block of code should be reported to SMS or not—would you like the PC_BIOS class for dessert?

Field Level Reporting Properties

You may also notice that inside each reporting class (block of code), there are similar lines that say [SMS_Report(TRUE)] or [SMS_Report(FALSE)]. Unlike the class level reporting property, these lines represent the individual bits of data inside a class. These lines are called **field level reporting properties**.

Below is an example of the Win32_SystemEnclosure reporting class. You'll notice that not only is the class enabled, but only the SerialNumber field is set to report. For now, don't worry about the syntax and just note how the class and its fields are enabled.

```
[SMS_Report(True),                ← Class Level Reporting Property
SMS_Group_Name("System Enclosure"),
SMS_Class_ID("MICROSOFT[System_Enclosure|1.0"])]
class Win32_SystemEnclosure : SMS_Class_Template
{
  [SMS_Report(False), key]
  string    Tag;
  [SMS_Report(False)]
  string    Caption;
  [SMS_Report(False)]
  string    Description;
  [SMS_Report(False)]
  string    Manufacturer;
  [SMS_Report(False)]
  string    Name;
  [SMS_Report(True)]              ← Field Level Reporting Property
  string    SerialNumber;
```

Just like ordering in a restaurant, you can order a burger without the pickle. You can even order it without the bun so you're left with just a beef patty. If that's the way you want it, you can have it!

If you want to inventory the Win32_SystemEnclosure class, but you *only* want the serial number information, just change the class level reporting property from FALSE to TRUE and set all of the fields under that class to FALSE except for the SerialNumber field. Voila! The only data that SMS will see from the Win32_SystemEnclosure Class will be the serial number information.

If I want other fields under that class, I just keep changing FALSE to TRUE?

Yes. The only thing that needs to be done for SMS to start reporting on a field is to change FALSE to TRUE on the field level reporting property line.

Many classes in the default SMS_def.mof file are already enabled at the class level, but may not have all of their fields enabled. Take a few minutes to scroll through all the reporting classes in the default SMS_def.mof and see if there are any additional fields you would like to enable to get them reporting to SMS.

What happens if I change all of the field level reporting lines to TRUE, but change the class level line to FALSE? No data from that class will report to SMS. Even if you only want the beef patty, you still have to order the burger first.

The same thing will happen if you change the class level reporting property to `TRUE` and all of the field level reporting properties to `FALSE`. If you order the burger without the pickle, bun, condiments, or beef patty, what is left?

Therefore, if you want *any* of the data inside a reporting class, you first have to verify that the class level reporting property is set to `TRUE`, and then enable any of the desired field level reporting lines inside that class.

Let's say at Schultz, Inc. I'm a freak about fans, so I want to know how fast the fans are spinning on my workstations. Since this is not typical information most administrators are interested in, it's not included by default in the `SMS_def.mof`, however it *is* included as a standard Win32 WMI class

Because this is a standard WMI class, the only thing I have to do is create a reporting class in the MOF that says `TRUE` I want to see the `Win32_Fan` class, and `TRUE` I want to see the `Desired Speed` field in that class. (Again, the exact syntax for this type of addition will be explained later).

If added correctly, I will now have the fan speed for all of my workstations reported to SMS, and displayed for the systems in Resource Explorer. WOOHOO!



In the above example, `Win32_Fan` is an existing class in systems with WMI 1.5 installed. Also, this class may not be populated with data if the hardware manufacturers have not provided the information in the proper format and locations. The example is used more for its simplicity than anything else.

Okay, the `SMS_def.mof` is used to order, or not order, items from the menu. What are these “special orders” you talked about before? WMI is able to provide just about anything your heart desires. However, you may come across a situation where you need to query the registry or even create a new WMI class that just didn't make it to the menu. To make these special orders you will need to create your very own data classes as well.

Special Orders

Occasionally, the information that you need to retrieve from a system is not found in an existing data class. Sometimes you will need to find information found in registry keys, such as application versions and system settings. However, because registry information is not normally collected by mere mortal SMS admins in either the hardware or software inventories, this becomes a more difficult request. Don't worry, by the time you finish this book you'll be able to do it in your sleep.

Thankfully, Microsoft provided the tools necessary to extract this information using WMI.

What are these tools? Well, remember the top 1% of the MOF “iceberg” that I mentioned earlier—the declarations? That is where the tools for the default SMS_def.mof file are located.

Example: At CDS Lettering, Inc., we put the User ID of the primary user of each machine somewhere in the registry. Because gathering registry keys is not standard (and there is obviously no existing class for every registry key or combination of keys) I have to add three items to the MOF.

- The Provider - What tool should I use to get a registry key?
- The data class - Where is this registry key?
- The reporting class - What does SMS want to see?



The registry providers are in the default SMS 2003 SMS_def.mof so you do not need to register a new provider to query registry keys with SMS 2003. However, there may come a time when you will want to register a different provider not registered by default, so pay attention. If you are still running an SMS 2.0 site, you absolutely must declare the registry providers.

Providers

The "tools" used to query for inventory information are called providers. In his hardware inventory training class Scott Stephen had an excellent analogy about the provider being a butler who shops at WMI-Mart. The shopping list used by the butler is the SMS_def.mof file.



Provider: A WMI Provider is a COM object that is an intermediary between WMI and a managed object.

Continuing with Scott’s analogy, there are actually three different types of butlers.

The first butler (or provider) we’ll discuss is able to get you just about anything you want from WMI-Mart, as long as you tell him exactly what it is you want, and exactly where he can get it. This is the **Property Provider**.



Property Provider: A Property Provider retrieves and modifies individual property values for instances of a given class that is stored in the WMI repository.

If you were instructing the Property Provider to shop for you, you would tell it:

"Get me the DVD movie 'Hannibal' from the Electronics area, aisle 7, shelf 1, column 9. Also get me a Hanes T-Shirt, white, large, style A, from Men's Clothing, aisle 4, shelf 3, column 6, and finally a bag of WEGE pretzels, broken, 18 oz, from the Food area, aisle 2, shelf 4, column 2."

The second butler (or provider) is less picky, but is also limited in certain ways. This butler doesn't need to know exactly what you want, but it is only able to shop in one area for one type of item. This is the **Instance Provider**.



Instance Provider: An Instance Provider supplies instances of one or more given classes.

In instructing the Instance Provider to shop for you, you would tell it:

"Go to the Electronics area and get me the name, price, producer, and director for every movie in aisle 7."

The third butler (or provider) is able to get you just about anything you want from WMI-Mart's mall or parking lot as long as it exists within WMI, and not just the SMS portion of WMI. This is the **View Provider**.



View Provider: The View Provider is an instance and method provider that creates new classes based on instances of other classes. You can use the View provider to take properties from different source classes, different namespaces, or different computers and combine the properties as a single class. We'll talk more about the View Provider in greater detail in Chapter 8.

Obviously, each butler (or provider) will be useful in different situations, depending upon your needs. With SMS 2003 all the butlers (providers) are now registered within the default SMS_def.mof. However, in SMS 2.0 since it's not typical to "shop" for registry keys and in the standard SMS_def.mof, the providers are not found in the file by default. *Each type of provider only needs to be defined once at the top of the SMS_def.mof.*



For SMS 2.0, I recommend downloading the Monster MOF freely available on www.SMSExpert.com. This MOF file includes all the providers and it a good starting point for Administrators new to SMS_def.mof editing.

For this next example, I'll be using the **Registry Property Provider** – since I know the exact key that I want, and I know where it is located. If it's not already there, to put the registry property provider into my SMS_def.mof, I'd simply add the following lines:

```
#pragma namespace("\\\\.\\root\\CIMV2")
instance of __Win32Provider as $PropProv
{
```

```

Name = "RegPropProv";
Clsid = "{72967901-68EC-11d0-B729-00AA0062CBB7}";
};

instance of __PropertyProviderRegistration
{
    Provider = $PropProv;
    SupportsPut = TRUE;
    SupportsGet = TRUE;
};

```



Just for kicks I'll show you the Registry Instance Provider. Remember, they are 2 separate entities.

```

instance of __Win32Provider as $InstProv
{
    Name = "RegProv";
    Clsid = "{fe9af5c0-d3b6-11ce-a5b6-00aa00680c3f}";
};

instance of __InstanceProviderRegistration
{
    Provider = $InstProv;
    SupportsPut = TRUE;
    SupportsGet = TRUE;
    SupportsDelete = TRUE;
    SupportsEnumeration = TRUE;
};

```



Don't worry about trying to learn the code that I pasted above. It is only there to show you what two of the tools mentioned look like in case a fire breaks out. I'll show you the View Provider later in Chapter 8. It can be a little tricky and confusing to introduce it at this point, and I want you to understand these first important concepts clearly.

Data Classes

The key I'm looking for is located in: *HKEY_Local_Machine\Software\Schultz* and the Value name is UserID. To declare this information in the SMS_def.mof, I will create a data class in the *root\CIMV2* namespace.

```
#pragma namespace("\\\\.\root\CIMV2")

[DYNPROPS]
class SchultzID
{
    [key] string    KeyName="";
    string        UserID;
};
[DYNPROPS]
instance of SchultzID
{
    KeyName="The Schultz User ID";

[PropertyContext("local|HKEY_LOCAL_MACHINE\SOFTWARE\Schultz|U
serID"),
    Dynamic, Provider("RegPropProv")] UserID;
};
```

Notice that two items were declared. First, a data class was created - called SchultzID. Second, an Instance of that class was created that tells WMI exactly where the registry key and its value are located, and finally, where to put the information.

Reporting Classes

The data class has been defined, so now I just need to make sure this data is collected by SMS. To do this, create a reporting class must be created to allow SMS to collect the "SchultzID" data class. Notice that the reporting class is defined in the *root\CIMV2\SMS* namespace:

```
#pragma namespace("\\\\.\root\CIMV2\SMS")

[SMS_Report(TRUE),
SMS_Group_Name("My Schultz ID"),
SMS_Class_ID("SchultzID")]
class SchultzID : SMS_Class_Template
{
    [SMS_Report(TRUE),key]
    string    KeyName;
[SMS_Report(TRUE)]
    string    SchultzID;
};
```

Put those three sections together, and when the MOF is compiled, the WMI angels fly down from the heavens and, magically, the class is created. The next time a hardware inventory is run, the client will report this data to SMS!



Chapter Summary

This chapter was just to introduce some of the core concepts of viewing and editing the MOF. Don't get flustered if you don't understand it all, as I'll be covering this and more in much greater detail in the rest of the book.

Modifying the SMS_def.mof is not terribly complicated if you follow the correct procedure.

The SMS_def.mof allows administrators to collect information stored in WMI, add to default inventory information, and report that information back to the SMS site server.

Notepad is the preferred application for modifying the SMS_def.mof.

What needs to be defined in the MOF:

- **The provider** - What tool to use to get the data?
- **The data class** - Where is the data?
- **The reporting class** - What does SMS want to know about the data?

The SMS_def.mof file is divided into two major sections:

- Top—declares providers and namespaces
- Bottom—reporting classes
 - Class Level Reporting Property
 - Field Level Reporting Properties

Both the class level reporting property and at least one field level reporting property must be set to TRUE in order for any data to be inventoried for that particular class.

There are three types of WMI Providers that SMS deals with on a normal basis:

1. **Property Provider.** A Property Provider retrieves and modifies individual property values for instances of a given class that is stored in the WMI repository.
2. **Instance Provider.** An Instance Provider supplies instances of one or more given classes.



3. **View Provider.** The View Provider is an instance and method provider that creates new classes based on instances of other classes. You can use the View provider to take properties from different source classes, different namespaces, or different computers and combine the properties as a single class.

Each type of provider needs to be defined only once in the SMS_def.mof.

Before you create a class, or register a Provider, check the SMS_def.mof file to see if the class or provider already exists, but is simply disabled.



Provider: A WMI Provider is a COM object that is an intermediary between WMI and a managed object.

Property Provider: A Property Provider retrieves and modifies individual property values for instances of a given class that is stored in the WMI repository.

Instance Provider: An Instance Provider supplies instances of one or more given classes.

View Provider: The View Provider is an instance and method provider that creates new classes based on instances of other classes. You can use the View provider to take properties from different source classes, different namespaces, or different computers and combine the properties as a single class.

Chapter 3: SMS_def.mof Syntax

Syntax, my lad. It has been restored to the highest place in the republic.

—John Steinbeck

The next step toward becoming a master MOF editor is to learn the proper syntax. For simplicity, I'm going to avoid covering every single item in the MOF. Instead I'm going to discuss the basic structure of the MOF, individual classes, and some of the most common elements used.

Compiling With MOFCOMP

I briefly touched on compiling in Chapter 2 but I'd rather not progress further without giving you a better description. Although it may work against Stormtroopers, waving my hand and uttering, "These aren't the definitions you're looking for" won't work against someone as smart as yourself.

As I stated before, a MOF file is simply a specially formatted text file. A program that comes with WMI, MOFCOMP ([Managed Object Format \(MOF\) compiler](#)) is used to edit, delete, and insert classes in the local WMI repository based upon what is defined in the MOF that has been compiled. When this executable is run with a MOF file as a parameter, the program will read through the MOF line by line, top to bottom, and perform whatever action is specified.

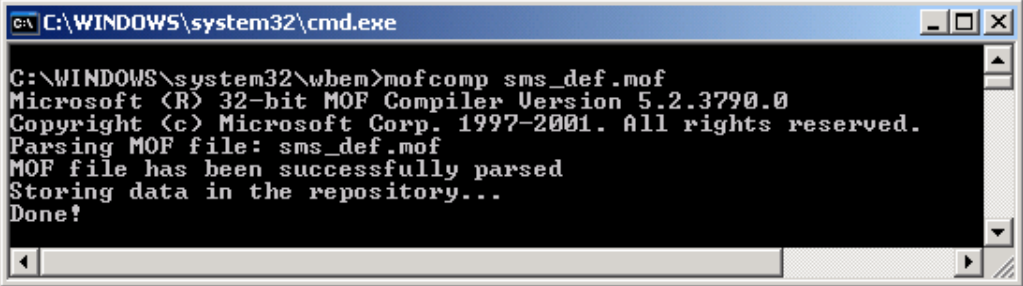
Unlike application compilers such as C++, Pascal or Visual Basic that take text from a file and convert that text into machine code, MOFCOMP simply reads the file and follows the instructions. In this way it is similar to a batch file. This could mean that MOFCOMP must create a namespace, edit a class, or even delete a class from the local WMI.

It is important to understand that MOFCOMP does only what it is instructed to do by the MOF file. If you were to tell MOFCOMP to put a steak on the grill and take it off after 10 minutes, you're going to have a pretty rare steak unless you tell it to light the grill first.

Because of this, it is very important to put your information into the MOF with the understanding that the items closer to the top of the file will be compiled before those closer to the bottom.

Figure 3.1 shows MOFCOMP in action ... and you thought this was going to be a boring book!

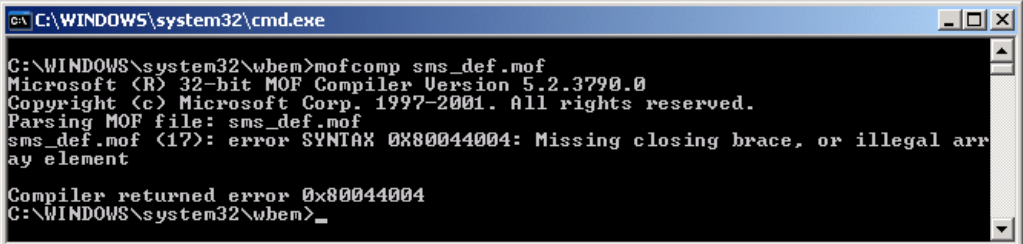
Figure 3.1



```
C:\WINDOWS\system32\cmd.exe
C:\WINDOWS\system32\wben>mofcomp sms_def.mof
Microsoft (R) 32-bit MOF Compiler Version 5.2.3790.0
Copyright (c) Microsoft Corp. 1997-2001. All rights reserved.
Parsing MOF file: sms_def.mof
MOF file has been successfully parsed
Storing data in the repository...
Done!
```

If the compiler runs into an error, or a class that is not properly structured, it will cease the compilation process and display an error message with the line on which the error occurred, as seen in Figure 3.2:

Figure 3.2



```
C:\WINDOWS\system32\cmd.exe
C:\WINDOWS\system32\wben>mofcomp sms_def.mof
Microsoft (R) 32-bit MOF Compiler Version 5.2.3790.0
Copyright (c) Microsoft Corp. 1997-2001. All rights reserved.
Parsing MOF file: sms_def.mof
sms_def.mof (17): error SYNTAX 0x80044004: Missing closing brace, or illegal array element
Compiler returned error 0x80044004
C:\WINDOWS\system32\wben>_
```

Because changes are almost instantaneous, every complete class or instruction prior to the error will have had its intended effect on the WMI repository. Everything listed after the error inducing line will not be seen by the compiler, as it has ceased the compilation process.

Think of the compiler as a finicky moviegoer going to a new show. If *“The MOFman Prophecies”* is a good movie, the compiler will stay for the whole thing, even through the credits. If the film is poor in the slightest, the compiler is unforgiving, and will walk out immediately, knowing only the parts it watched.

When a hardware inventory is run, MOFCOMP goes into action just as if you had manually requested it to run. The logs for the hardware inventory, including status

messages about the success or failure of the MOF compilation, can be found in the locations below:

- For the SMS 2.0 (or Legacy) Client: %windir%\ms\SMS\logs\himv.log.
- For the SMS 2003 Advance Client: %windir%\system32\ccm\logs\inventoryagent.log

Comments

One of the most common phrases uttered by administrators editing the MOF is “[Dadgummit], where did I put that?” Due to the size of the MOF, it is extremely easy to “lose” classes and fields, or to forget where a change was made. Quite often, if two administrators are working together, changes may be made by one of which the other is not aware.

Because of this, commenting is extremely important. Let me repeat that—commenting is *extremely* important. Where is that new class you just copied in from www.SMSexpert.com or from www.myITforum.com? Who added that field to my class? Did I add this, or was it in the default SMS_def.mof file? These are questions to which you absolutely want answers.

Adding comments to a MOF file is very easy. There are two ways to comment text. The first is with two forward slashes (//). These two characters together tell the compiler to ignore everything on the line after the slashes.

These slashes could be used at the beginning of a line:

// Start of new class entry (10/08/2005)

In the middle of a line of code:

```
#pragma namespace ("\\.\root\CIMV2") //Hello ☺
```

Or they can be used to create a block of comments to help structure your MOF:

```
//=====
// START of MOF additions – M3
//
// 08/04/2005 - Gawd I hope this works!
// 08/08/2005 - Added System Enclosure Class
//=====
```

The second way to comment in the MOF is to use /* and */ to surround the text you want the compiler to ignore. This method is primarily used to “remove” classes

without actually deleting the code, but it can be used for large blocks of comments or even simple comments as well.

For example, commenting a line:

```
/* Start of new class entry (10/08/2005) */
```

In the middle of a line of code:

```
#pragma namespace ("\\.\root\CIMV2") /*Hello again ☺ */
```

Or they can be used to “remove” a class from the MOF:

```
/*  
Here I'm defining a class  
More class stuff...please don't comment me!  
I want to be added to WMI!  
Please?  
Sugar on top?  
End Class definition  
*/
```

As you can see, commenting your MOF is a simple task and the benefits far outweigh the time it takes to create them.

There are three rules that I follow when creating comments:

1. At the end of the stock SMS_def.mof file, create a large block of comments to signify the end of the default section and the beginning of your own. Inside this large block of comments make a summary of all the changes you are adding, why you are adding them, who added them, and the date they were added.
2. Before each new class is defined include a block of comments stating the name of the class, what data it is to retrieve, any relevant Q articles, and any modifications to the class with their respective dates.
3. Create a comment before any line that has been modified or altered from the original, or if you want to remember why a class or field isn't enabled for inventory.

Although it sounds like a tremendous amount of work, when you go back to your MOF two months after making modifications, you'll be very thankful that you made these comments.

Definitions

Before going further into the syntax, let's cover a few of the terms I'll be using in the rest of the book. This is another section of the book that is pretty important, so for those readers skimming the pages it's time to downshift and actually read through this section thoroughly. I have already mentioned some of these terms in the previous chapters but it can't hurt to review.

Namespaces

Namespaces are containers used to hold class and instance data. Adding a class to WMI is not like leisurely tossing dirty socks into a hamper with the rest of the laundry. Rather, adding a class is more like gently folding each piece of clothing and placing it in the appropriate dresser drawer.

Each class must be placed in the appropriate drawer (or namespace) to be found when hardware inventory comes looking for it. If you have placed your favorite pink polka-dot socks in with your boxers, hardware inventory will not be able to find them when the time comes to review your wardrobe.

A namespace is structured much like a file directory. Data and reporting classes are defined inside a namespace just like files in a directory. This namespace (or directory) can also have other namespaces (sub-directories) underneath it.

You may recognize the following line from your SMS_def.mof file:

```
#pragma namespace ("\\\\.\\root\\CIMV2")
```

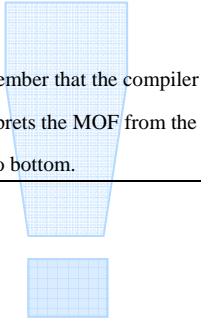
This line can be interpreted as, "From this point forward, place all classes into the `root\CIMV2` namespace until told otherwise." Prior to defining reporting classes, data classes and Instances, this line is necessary to tell the compiler to put this information in the proper location (namespace).

Another line often seen is:

```
#pragma namespace ("\\\\.\\root\\CIMV2\\SMS")
```

This line simply changes the namespace to be used to `root\CIMV2\SMS`. Often in a heavily modified SMS_def.mof you will see the default namespace switched back and forth between the two namespaces.

How many namespaces do I have to deal with? The two namespaces mentioned above are the primary namespaces used when you're making MOF modifications. There are many more namespaces that you can spend your free time perusing for juicy information to inventory, but for now let's stick to the main ones.



Remember that the compiler interprets the MOF from the top to bottom.

Reporting classes must be placed into the `root\CIMV2\SMS` namespaces, while all data classes and instances are usually in the `root\CIMV2` namespace.

Why is that a rule? The reason for this is because of the way the hardware inventory operates. When hardware inventory runs, it checks the `root\CIMV2\SMS` namespace to find the reporting classes, which tell it which data class information to pull from the data classes. The data class information in `root\CIMV2` either provides the hardware inventory with data directly, or tells it where the data can be retrieved.

How does the hardware inventory know which reporting class goes with which data class? Each data class and corresponding reporting class share the same class name.

Do I have to use that namespace line before every class I create? No. You only need to use the `#pragma namespace` line when you want to *change* the namespace to be used. If you have a large amount of data that needs to go into the same namespace, just add the line once prior to the modifications.

Great! Now what are these data classes and reporting classes again? Well, just read on...

Data Classes

A data class is the rainbow that leads you to the pot of gold. It either contains the data itself, or it points you to where you can get it.

If you are surfing the web for information, the data class would be the keywords you're searching for. If you're looking for how old Mel Gibson is, you would enter "Mel Gibson Age" into a search engine and you would be provided with either the answer to your question, or you would be directed to a web page that would give you the answer.

A pseudo-data class would look like this:

```
MelGibsonClass
MelsAge = 49;
MelsMiddleName = Florence;
MelsFavoriteCars → Go to www.favoritecars.com/melgibson.html
End MelGibsonClass
```

Again, notice in the example above that the data class can either contain the data directly, or it can point you to a location where you can find the data you're looking for, as it does with `MelsFavoriteCars`.

Reporting Classes

The reporting class lets you pick and choose what classes and fields in WMI you want to see reported to your SMS site. Using the same analogy above, the reporting class is a filter of the data. Your data class can include every bit of information about Mel Gibson, but if you only want his age, the reporting class will tell the hardware inventory to only collect the **MelsAge** field from the **MelGibsonClass** data class.

As I mentioned earlier, there are hundreds of other classes and thousands of fields that could be added to the SMS_def.mof file. The reporting class enables you to define what information you want to know about your SMS clients, and allows you to filter out data that is either of no use to you, or hindering your use of relevant data.

For example, if you're bitten by a spider and want to know if it is poisonous, you can find a plethora of information on the Internet about that spider. However, you don't care what its Latin name is, whether it is indigenous to Canada, or how its migration north is relational to the Mezzo-American jumping bean population.

What you want to see is:

Spider Name: Brown Recluse

Looks Like: Brown and Fuzzy

Poisonous: If you're able to read this page, you weren't bitten by one of these.

The reporting class allows you to filter out the junk and provides you with just the "good stuff."



Remember, the namespace is changed to `root\CIMV2\SMS` prior to the reporting class definition.

Instances

WMI was created with many object-oriented principles. One of these principles is the process of defining a class in generic terms, and then creating an instance of that class. Defining a class is like creating a mold. When you use that mold to create objects, you're creating instances.

When you built sandcastles as a child, you may have used a bucket as a mold. When the mold is removed, the wet sand retains the shape of the bucket. If you were to use that same mold while making gelatin for a party, the gelatin would have the same shape as the sandcastle, and probably have the same texture if you forgot to wash it out.

In short, instances of a particular data class will have the same shape, or fields, but they may be comprised of largely different data.

Providers

If data classes are the keywords and reporting classes are the filters, then a provider would be the search engine used to get the job done. You don't know how a search engine like www.google.com works, but you know if you type in the right keywords, you'll get the information you want.

Providers were created by Microsoft and others to perform certain tasks. When you want to write a letter, they are the pen. When you want to turn a screw, they are the screwdriver. When you want to drive a car, they are the engine. If you want a data class to return data, a provider must be used.

Basically what you need to understand about providers at this point is this: the provider is the middle man between WMI and SMS.

Providers are described in more detail later in this and other chapters. You didn't really think I'd just leave you hanging like that, did you?

SMS_def.mof Structure Recap

In the first two chapters, the SMS_def.mof file was described as having two main sections. The smaller top section contains “stuff” that was to be largely ignored, with the significantly larger bottom section containing reporting classes.

The top section consists of lines to create the *root\CIMV2\SMS* namespace (if it doesn't already exist), and lines to register a few providers. Again, *providers* are the “tools” that are used by the hardware inventory to retrieve the requested data. Since these lines do not need to be modified, they can, once again, be ignored.

If you scroll through the rest of the MOF, you'll see the reporting classes with fields that can be enabled by simply changing False to True. Unfortunately, this is when things start to get complicated.

You'll recall that reporting classes are used to retrieve data from data classes. If the reporting class for an existing data class is not in the default SMS_def.mof file then you can create a reporting class in the *root\CIMV2\SMS* namespace so that the hardware inventory will collect the information you're after. MOF editing can become complicated when the data you are looking for is not already in a default class and you must create your own. Aren't you lucky that this guide exists?

Basic MOF Modification Steps

Step 1: At the end of the stock SMS_def.mof, create a big block of comments to signify the end of the default section, and the beginning of your own. This will help you to avoid accidentally deleting or modifying anything that you did not create. Speaking of deleting things, it's almost never a good idea to completely delete default sections of the MOF. While the sections may not seem useful to you, those data classes were entered into WMI for some reason and deleting them may adversely affect the client system adversely.

So, following step 1, your initial comments should look something similar to the below excerpt:

```
//=====
// THIS IS MY STUFF
// Written by: <insert name, if you dare>
// Date:
//=====
```

Step 2: Open your toolbox and pull out the tool you intend to use. In other words, register the provider you intend to use to retrieve the data if it's not already listed in your SMS_def.mof. This step is primarily for SMS 2.0. Check the declarations (top section) of your SMS_def.mof to see if the provider you're trying to use is already registered in any case.

Step 3: Define the data class that the provider will query to retrieve the data.

Step 4: Define the reporting class that will tell the hardware inventory agent which fields in the data class you would like to see reported.

The basic premise is fairly simple: register the tool you want to use, define the data class that the tool will query, and define the reporting class so that the proper fields are collected from the data class.

Do I always have to define the data class before the reporting class? Nope. Step 3 and Step 4 are interchangeable. The reporting class *can* be defined prior to the data class because when the MOF is compiled, the classes are simply created in their respective namespaces in the local WMI. As long as they share the same class name (which they must), the order in which they are created does not matter.

I prefer to define the data class prior to creating the reporting class. Logically, it makes sense to me to define what the data class should look like before deciding what pieces of that class I actually want reporting to SMS.

Following the steps above, a pseudo-MOF addition would look like this:

```
//My Block of Code
#pragma namespace("\\\\.\\Root\\CIMV2")
<providers are registered>
<data class "TravelMode" is defined>

#pragma namespace ("\\.\\root\\CIMV2\\SMS")
<reporting class for "TravelMode" is defined>
```

In English, this says: "Inside the *root\CIMV2* namespace, register a provider and define a class called "TravelMode". Switch to the *root\CIMV2\SMS* namespace and define a reporting class for the "TravelMode" class."

As you can see, providers are registered and data classes are defined in *root\CIMV2*, and reporting classes are defined in *root\CIMV2\SMS*. To tell the compiler where the classes should be defined, the *#pragma namespace* line is used.

The code above is a good example of what I like to call a “block of code”. All relevant information pertaining to the TravelMode class is located in one area of the MOF. Should I want to remove this class, I can either comment this section with */** before it and **/* after it, or just delete the entire section. Either way, the class would no longer be seen by MOFCOMP when the SMS_def.mof file is next compiled.

What if I want to pull data from somewhere besides root\CIMV2?

Ahh, the hundred dollar question! There are many namespaces other than *root\CIMV2* and *root\CIMV2\SMS*. Normally, SMS is pretty happy with collecting data from those two namespaces, but if you’re like me, and I know I am, then you’ll be interested in extending your inventory capabilities beyond these standard locations.

If you’re still running SMS 2.0 then you’re going to have to do it the hard way. In this case, the hard way being the use of the WMI view provider to mirror the information to the *root\CIMV2* namespace so it can be accessed by the hardware inventory.



The View Provider is explained in Chapter 8.

The SMS 2003 Advanced Client’s inventory agent *can* access namespaces other than *root\CIMV2* by using a reporting class qualifier. When extending your MOF for a class not included in the normal *ROOT\CIMV2* namespace, just slap the name of the namespace you want to get to under the “normal” reporting class block of code before all that TRUE and FALSE stuff, as in the example below that inventories Internet Explorer information. The namespace qualifier is highlighted in red below.

```
#pragma namespace ("\\\\.\\root\\CIMV2\\SMS")

[ SMS_Report (TRUE), SMS_Group_Name ("Microsoft IE Summary"),
  SMS_Class_ID ("MICROSOFT|IE_SUMMARY|1.0"),
  Namespace("\\\\.\\root\\CIMV2\\Applications\\MicrosoftIE")]
class MicrosoftIE_Summary : SMS_Class_Template
{
  [SMS_Report (TRUE)] string Build;
  [SMS_Report (TRUE)] string IEAKInstall;
  [SMS_Report (TRUE)] uint32 CipherStrength;
  [SMS_Report (TRUE)] string Version;
  [SMS_Report (TRUE),Key] string Name;
};
```

Big ups to Kan Mongwa for opening my eyes to this technique and pointing out the *root\CIMV2\Applications\MicrosoftIE* namespace.

This is also how you can get to the yummy morsels of information about those applications that have their own WMI Providers, such as Microsoft Exchange, SQL Server, and IIS.

Why does your MOF syntax appear different from my default SMS_def.mof?

To MOFCOMP there is no difference in the format I'll use, and that of the standard, Microsoft format shown. The two examples below illustrate what I mean. First, I'll show you the default format of the SMS_def.mof:

```
[ SMS_Report (TRUE),
  SMS_Group_Name ("Physical_Memory"),
  SMS_Class_ID ("MICROSOFT|Physical_Memory|1.0") ]
class Win32_PhysicalMemory : SMS_Class_Template
{
  [SMS_Report (TRUE) ]
  string BankLabel;
  [SMS_Report (TRUE) , SMS_Units("Megabytes") ]
  uint64 Capacity;
  [SMS_Report (TRUE) ]
  string Caption;
  [SMS_Report (TRUE)]
  string DeviceLocator[];
  .
  .
  .
};
```

... and then our customized format:


```
[ SMS_Report (TRUE), SMS_Group_Name ("Physical_Memory"),
  SMS_Class_ID ("MICROSOFT|Physical_Memory|1.0") ]
class Win32_PhysicalMemory : SMS_Class_Template
{
  [SMS_Report (TRUE)] string   BankLabel;
  [SMS_Report (TRUE) , SMS_Units("Megabytes")] uint64 Capacity;
  [SMS_Report (TRUE)] string   Caption;
  [SMS_Report (TRUE)] string   DeviceLocator[];
  .
  .
  .
};
```

I use this format to help differentiate between the original MOF and our customized sections, and because it makes the MOF shorter and easier to read. You can use whatever format you wish.

Hey, what's that crazy "SMS_Units("Megabytes")" section in your example all about?

Bonus points to you for seeing that! In order to format WMI information to make it easier to understand and view we can use the SMS_Units("Megabytes") line to convert that uint64 (unsigned 64-bit integer) value into megabytes.

This information is actually located in the default SMS_def.mof. Below are the possible data conversions you can make from the raw WMI inventory data into something that SMS can either read, or just to make it easier for us humans to see what is going on:

- Kilobytes, divides integer value by 1024
- Megabytes, divides int value by (1024 * 1024)
- HexString, converts int value to hex characters, (i.e: hex value 0A1 converted to string "0xA1")
- DecimalString, converts int value to decimal string (i.e: value 123 converted to string "123")
- Seconds, divides int value by 1000
- DateString, converts value to interval string (i.e: DateTime value "00000008061924.000000:000" turns into string "'8 Days 08:15:55 Hours")

Confusing? Here's a quick example. Figure 3.3 is the resource explorer view for the Win32_PhysicalMemory class that inventories physical RAM chips. Without putting in the SMS_Units("Megabytes") line for the uint64 data type field Capacity:

Figure 3.3

BankLabel	Capacity	Caption	CreationClassName	Devic...	FormFactor	MemoryT...	PositionInRow	Speed	Tag
	536,870,912	Physical Memory	Win32_PhysicalMemory	DIMM_1	8	17	1	400	Physical Memory 0
	536,870,912	Physical Memory	Win32_PhysicalMemory	DIMM_2	8	17	1	400	Physical Memory 1
	536,870,912	Physical Memory	Win32_PhysicalMemory	DIMM_3	8	17	1	400	Physical Memory 2
	536,870,912	Physical Memory	Win32_PhysicalMemory	DIMM_4	8	17	1	400	Physical Memory 3

Adding SMS_Units("Megabytes") to Capacity field like so:

```
[SMS_Report (TRUE) , SMS_Units("Megabytes")] uint64 Capacity;
```

Gives you the resulting Capacity information displayed in Figure 3.4:

Figure 3.4

BankLa...	Capacity	Caption	CreationClassName	Devic...	FormFactor	MemoryT...	PositionInRow	Speed	Tag
	512	Physical Memory	Win32_PhysicalMemory	DIMM_1	8	17	1	400	Physical Memory 0
	512	Physical Memory	Win32_PhysicalMemory	DIMM_2	8	17	1	400	Physical Memory 1
	512	Physical Memory	Win32_PhysicalMemory	DIMM_3	8	17	1	400	Physical Memory 2
	512	Physical Memory	Win32_PhysicalMemory	DIMM_4	8	17	1	400	Physical Memory 3

512 MB is a lot easier to understand than 536,870,912 Bytes, right?



Chapter Summary

MOFCOMP.EXE, which comes with WMI, is used to edit, delete and insert classes into the local WMI repository based upon information stored in the MOF file.

If there is a syntax error in the MOF file, MOFCOMP will stop compiling at the point that it encounters the error in the MOF.

Compiler status messages can be found in the `hinw.log` for legacy clients or the `inventory.log` for advanced clients.

There are two ways to add comments in your MOF:

1. Place two `/`'s (`//`) in front of your comments on a single line
2. Place `/*` in front of the start of a block of comments and end your comments with `*/`.

Data stored in namespaces other than `root\CIMV2` or `root\CIMV2\SMS` can be accessed by using a view provider for legacy clients or a namespace qualifier for Advanced Clients.

Namespace: namespaces are containers used to hold class and instance data.

Reporting Class: reporting classes define the information within data classes targeted for inventory.

Data Class: data classes either contain the inventory data you're after or point to its location.

Instance: Instances are generic class definitions.

Provider: Providers are the "tools" used to access information stored in WMI.



Chapter 4: Introduction to WMI Manipulation

There are things known and there are things unknown, and in between are the doors.

– Jim Morrison

So far we've covered a lot of the basics of hardware inventory, MOF files, SMS_def.mof syntax, and compiling. One thing that always seems to pop up with these is WMI. Understanding the relationship between WMI and SMS hardware inventory is *critical* to your success as a MOF editor.

Grasping the concepts of how WMI is structured, and how it is accessed and manipulated by SMS is the door to the Kingdom of the MOF Master ...and I've left it open for you.

The Basic Structure of WMI

Remember from Chapter 1 when I first introduced you to WMI? I said that WMI, Windows Management Instrumentation, was the Windows implementation of WBEM or Web-Based Enterprise Management initiative. Well, what the heck does that mean you've probably been wondering.

WMI (Windows Management Instrumentation) is basically a tool used to gain access to and/or modify the data stored in Common Information Model Version 2 (CIMV2) repository on a system. This makes the CIMV2 Repository the "home" for all that interesting data. WMI is like a kind of address book that the SMS_def.mof uses to navigate around the neighborhood searching for those objects that you've chosen to inventory.

A complete explanation of CIMV2 is beyond the scope of this book, so bear with me if you've noticed the over-simplification here. Instead, I'll be focusing on the elements of WMI that are important to hardware inventory and the SMS_def.mof.

WMI is organized into namespaces (classes and instances) that contain more namespaces (subclasses and instances). Think of the WMI repository as a file cabinet,

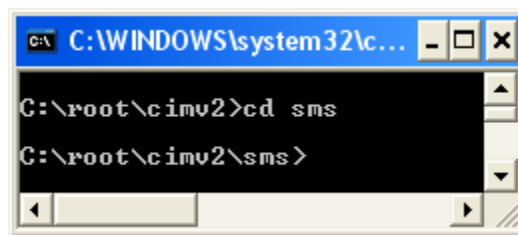
or a computer's file and folder structure. There are root folders which, in turn, contain subfolders full of significant information. You may remember seeing a line such as the one below a few pages back:

```
#pragma namespace ("\\.\root\CIMV2\SMS")
```

Does that line make a little more sense now? Remember, the *#pragma namespace* part tells SMS to switch namespaces. The *root\CIMV2* namespace is where most of the data is located (or instances of classes are to be technical), and the *root\CIMV2\SMS* namespace is where SMS stores everything it wants reported in hardware inventory—the reporting classes and instances. So in the *#pragma namespace* example above, MOFCOMP is being told to switch to the SMS namespace in order to store some vital piece of information.

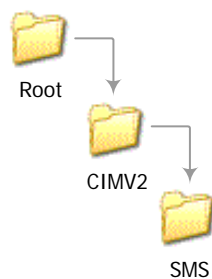
Continuing the computer directory analogy, and to go back in time for some people, I'll compare this to a simple DOS command shown in Figure 4.1:

Figure 4.1



Didn't help? OK, how about a picture depicting the namespace organization such as that displayed in Figure 4.2?

Figure 4.2



Think of the root as...well ... the root, the CIMV2 as the CIMV2 repository containing all the significant information about the system, or data classes, and the SMS folder as containing everything you want SMS to report on, the reporting classes.

OK show of hands, who likes the folder structure view better? All right, I'll use that from now on when I want to illustrate the WMI structure. I'm still partial to the command line, but I know I'm part of a dying breed.

You may notice that I've only shown the CIMV2 and SMS namespaces above. Does that mean these are the only two namespaces SMS is capable of peering into? No way! Sometimes the data you want to see is in a class not included as a root class within CIMV2. Remember, the `root\CIMV2\SMS` namespace is used to store the information you've tagged for reporting, but there are other sub classes that need to be considered for inventorying as well.

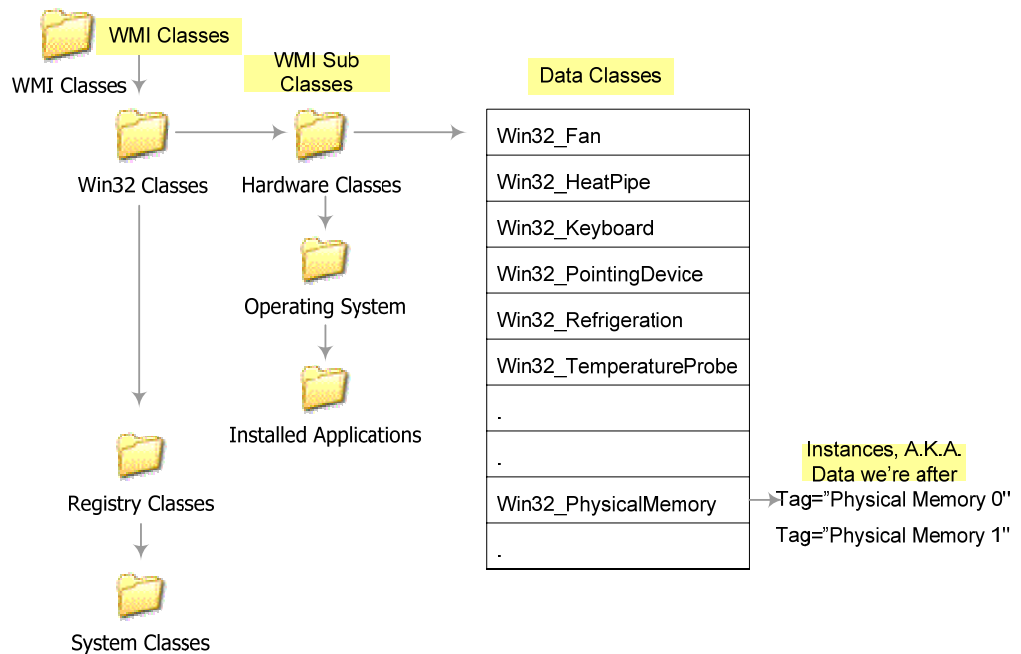
Generally speaking, you will use the "normal" CIMV2 Win32 classes for 99% of your SMS_def.mof editing, but I wanted to make you aware of the importance of the different namespaces within WMI. We'll talk more about this later; I just didn't want you to get tunnel vision on the `root\CIMV2` and `root\CIMV2\SMS` namespaces at this point.

WMI Data Classes

Data classes are the meat and potatoes of hardware inventory and for modifying the SMS_def.mof. Sometimes you may prefer fish, or registry information, but for the most part you'll spend your time making MOF edits focused on data classes.

Remember, WMI namespaces are full of classes, which in turn, are full of subclasses, and both contain instances...that's about as clear as mud, right? I remember reading something like that once when I first started learning about MOF editing so I'll illustrate this here in Figure 4.3—a picture is worth a thousand words, right?

Figure 4.3



Now when you want to see how many RAM chips are in a system you can expertly expound upon accessing the CIMV2's Win32 Hardware Classes in order to retrieve the instances contained within the Win32_PhysicalMemory data class! See, not only is this book informative, it offers you the opportunity to speak in a different language!

If this still makes no sense to you, remember where this picture is and come back to it later. I've purposely used the Win32_PhysicalMemory data class as an example because you will be seeing it again later on.

Obviously, that diagram barely scratched the surface of WMI classes, but rather than make the rest of this book one big picture full of fun arrows and pretty boxes, I'll just refer you to the below links if your thirst for WMI classes has yet to be quenched:

A list of all WMI classes:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/wmi_classes.asp

A list of all Win32 classes:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/win32_classes.asp

A list of all Win32 hardware classes:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/computer_system_hardware_classes.asp

Go ahead and check those websites out. It's good to be familiar with these sites, because if you click on a class of your choice you'll notice that the class is broken down for you. Each property is meticulously defined, including its name, the type of data for each of its properties, and even a breakdown of what those crazy values mean that the data represents. You'll also notice one or two properties with a line that says "*Qualifiers: Key*".

Key Fields

Hear me now, believe me later, if you do not properly identify the key fields for your reporting classes you are asking for serious trouble and probable loss of sleep.

Key fields are used by WMI and SMS to identify individual objects within arrays among other things. Think of the physical memory class again. The odds are pretty good nowadays that a system is going to have more than one RAM chip. In order to differentiate the queried data between the chips the key field, or fields, for the class are referenced.

When you enable a new reporting class in your SMS_def.mof and at least one client conducts a hardware inventory, the data and history tables—among other things—are created in your SMS database to store the newly inventoried information. The key fields are identified at that time for the new class table in the SQL Database. These key identifiers for the table cannot be updated by modifying your SMS_def.mof in the future. Let me say that again for effect: *these key identifiers for the table cannot be updated by modifying your SMS_def.mof in the future.*

In case you skipped checking out the websites earlier, here's a link to the Win32_PhysicalMemory class so that you can see those key qualifier lines for yourself:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/win32_physicalmemory.asp

If you clicked on that link then now you can also see what I meant by the value data too. Notice that a FormFactor value of 8 means DIMM. This comes in handy when you're perusing through your SMS site data at a physical memory report and your boss asks you what they heck 8 means there!

If you are creating your *own* class from scratch, you can choose whichever field you think is most likely to be unique as your key field. If you think there are two fields that must be unique to identify the differences between two objects, then make a couple of the fields key fields. This is called a compound key.

Using Tools to Access WMI

There have been many days when I've wished I could just take a stick and bang inventory data out of my SMS Server, but until Microsoft Piñata Server 2027 comes out we'll all have to do it by utilizing applications or scripts that access WMI information and return it to you in some meaningful format.

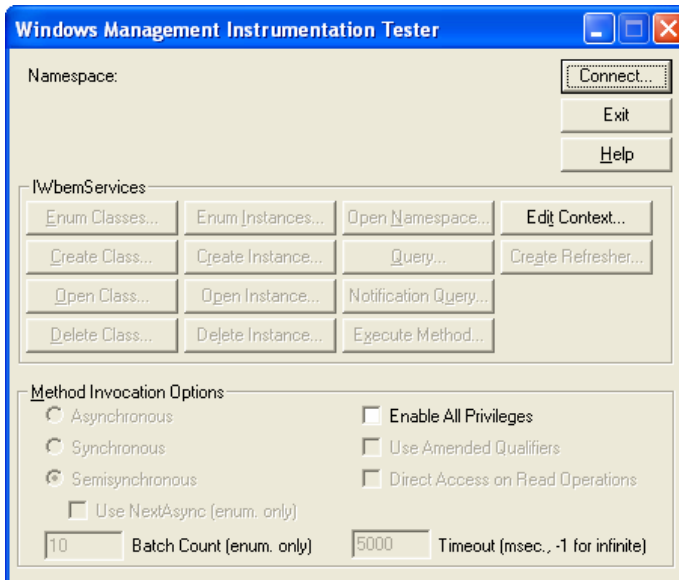
It seems that everyone has a favorite for taking a stab at WMI programmatically. I think the easiest, and since it's installed with WMI, the most obvious to use is the dexterous WBEMTEST.EXE, or the Windows Management Instrumentation Tester to be precise.

WBEMTEST

I'll mention a few other methods to access WMI information, but since you already have access to WBEMTEST by default on WMI enabled systems I'll go into deeper detail on it than the others.

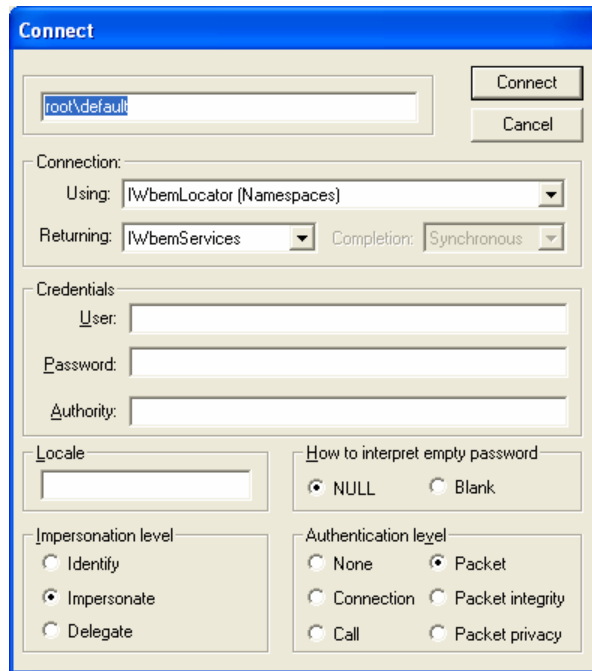
Accessing WBEMTEST is simple. Just click on the Start button, click Run, type in WBEMTEST and hit OK. Once all that is accomplished, you are rewarded with the following eye-catching screen in Figure 4.4:

Figure 4.4



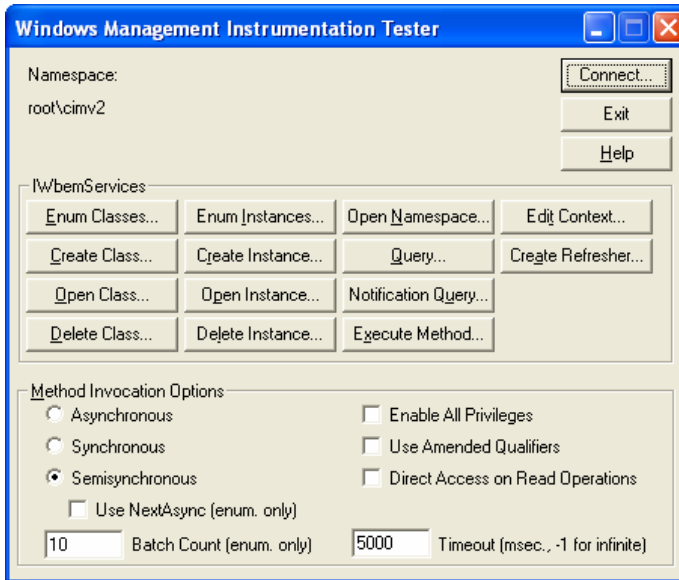
Next, you'll want to click on Connect to get the party started. Which gives you the resulting screen displayed in Figure 4.5:

Figure 4.5



Now, remember the namespaces SMS likes to look in? Type over *root\default* with *root\CIMV2* and click Connect again.

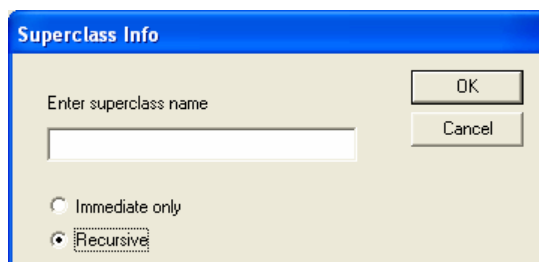
Figure 4.6



Now you're back from where you started! Just kidding. Notice now that there is actually a namespace listed under Namespace (*root\CIMV2*) and the IWbemServices buttons are enabled.

Let's play with the buttons. Click on *Enum Classes* and check the recursive radio button as so:

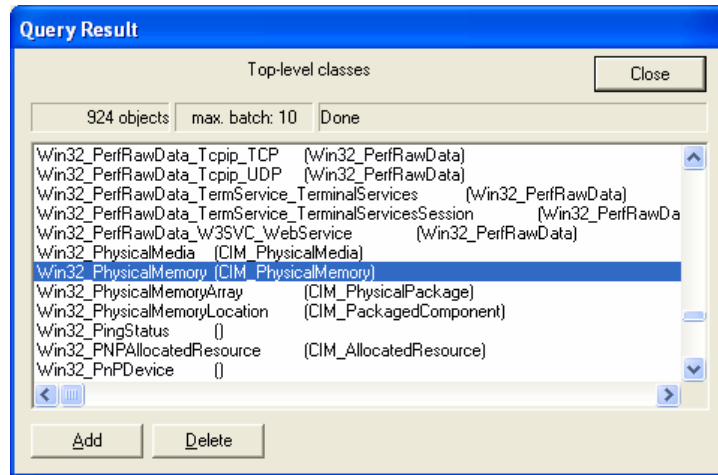
Figure 4.7



Click OK and Whammo! Here comes the WMI data from your system right to your desktop.

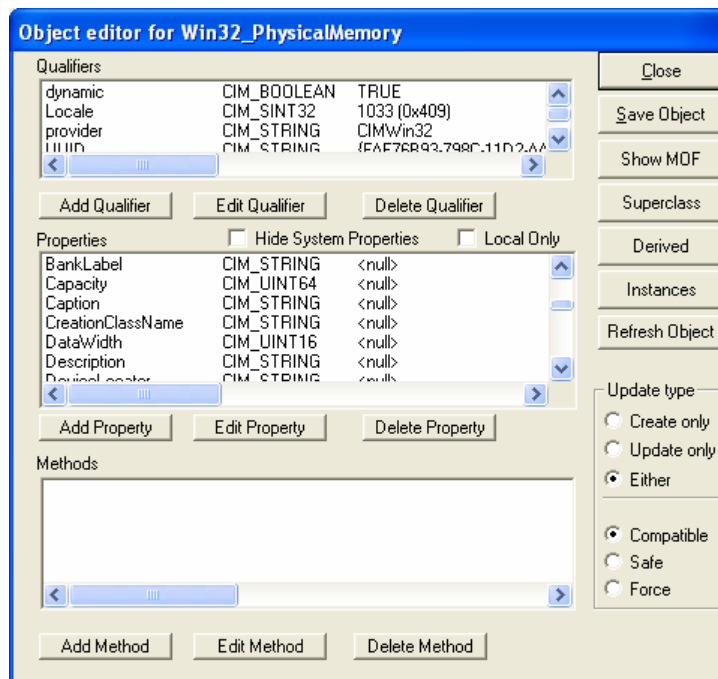
I'm planning to be more spontaneous in the future, but for right now I'm going to stick with the *Win32_PhysicalMemory* class. To see the *Win32_PhysicalMemory* information via WMI simply scroll down the list of classes until you see the physical memory section:

Figure 4.8



Double clicking on the Win32_PhysicalMemory Class name yields you the next resulting screen shown in Figure 4.9:

Figure 4.9



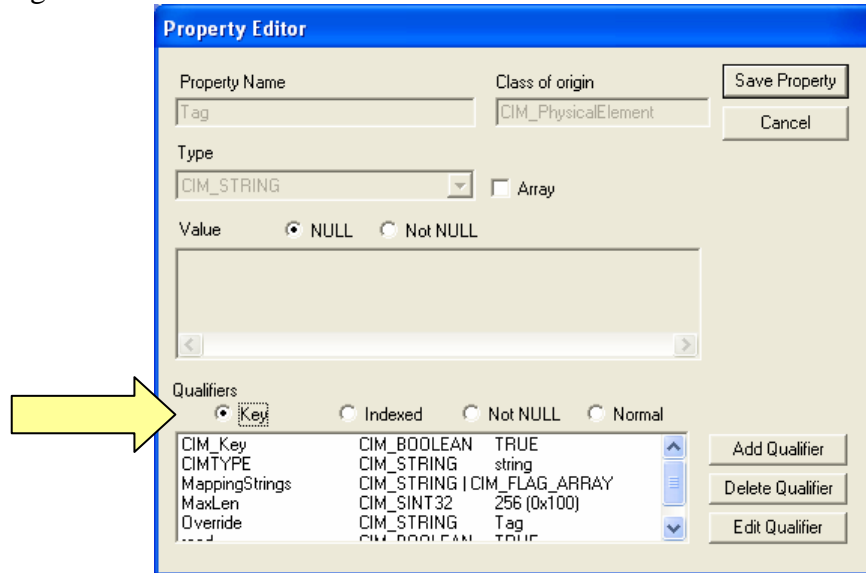
From here you can scroll down the list of properties and see the data type for each field within the class.



These fields are also defined on the web page for the individual class definitions on MSDN's website.

One thing you'll notice about these properties, though, is that nowhere does it say which fields are 'key' fields. To see which properties have the key qualifier using WBEMTEST, you have to double click on each property and look for a resulting screen as Figure 4.10 illustrates:

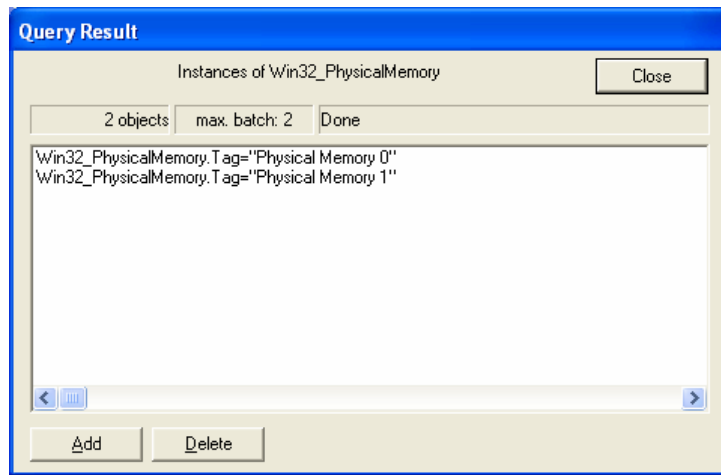
Figure 4.10



This can be quite tedious to do for classes with many, many properties to check, so use this as a starting point and then check, *and double check*, the field information for the key fields from the class website.

Now we have seen the class and the reporting fields, let's see what is actually stored on the system for the class. Click on the 'Instances' button on the right side of the Win32_PhysicalMemory object editor screen and you'll be presented with Figure 4.11 (or something similar, of course):

Figure 4.11



From these results you can tell there are two physical memory chips installed on my current machine. So I'm getting something anyway. Don't be despondent if a ton of cool information doesn't immediately present itself to you. The information stored in WMI is hardware vendor specific. In other words, if the RAM manufacturer decided not to include this information it won't be there. Conversely, running this same query on another machine may yield more information than you even want.

If you're really bored and decide to count the fields in the Win32_PhysicalMemory Class you'll see that there are 30 different reporting fields possible. You can use whatever tool your heart desires to view the information stored in those fields and then include only those fields you want SMS to report on in your SMS_def.mof modification. Remember, if you want, you can add them all, and just set the report qualifier to FALSE for the ones you're not interested in.

The main idea here is to verify that this class actually holds the information you're looking for in general. I like to use this method to find the data types and fields for making my own reporting classes.

So now we've used WBEMTEST to connect to the *root\CIMV2* namespace, found the Win32_PhysicalMemory Class, viewed the data type information for the fields and enumerated the instances of that class on a local machine. These same actions can be made with the other programs we'll talk about next.

CIM Studio

CIM Studio is more complex than WBEMTEST, and designed with developers in mind. It's really a set of four web page files (studio.htm, studiobanner.htm, calssnav.htm, and editor.htm) that interact with the CIM repository on a system via WMI.

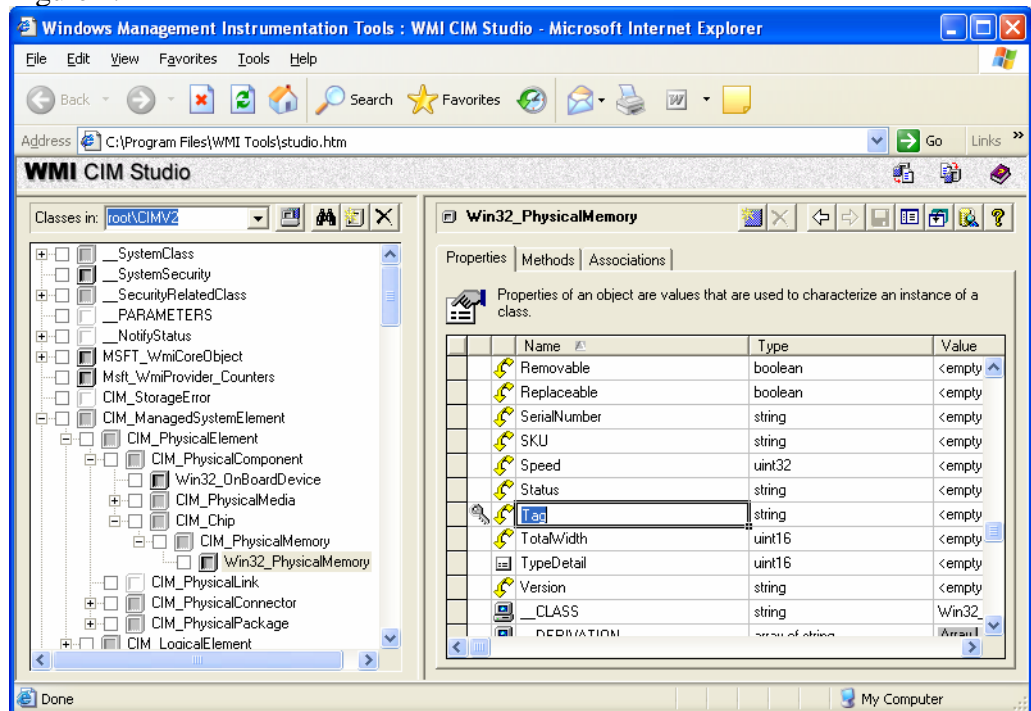
For those who desire a deeper look into WMI, download the WMI Administrative Tools containing the CIM studio:

<http://www.microsoft.com/downloads/details.aspx?FamilyID=6430f853-1120-48db-8cc5-f2abdc3ed314&DisplayLang=en>

Using CIM Studio you can display a tree view of the CIMV2 repository of a system, view details on classes and instances, and edit that information.

By drilling down the tree view of the CIMV2 repository on the left pane of CIM Studio you can select the class you want to view. Clicking on the class gives you a detailed properties view in the results pane on the right side. Something worth noting here is that when you view the properties of the class, the key fields are identified by an image of a key in the results as seen in Figure 4.12:

Figure 4.12



In my opinion, CIM Studio is great for advanced scripters or developers, but not as user friendly as some of the other options I'll talk about next.

Scriptomatic Version 2

For those who are into a little more glamour and pizzazz, there's the Microsoft Scripting Guys' GUI answer to WBEMTEST—The Scriptomatic (version 2) which is an .hta application and always a crowd favorite. Not only does the Scriptomatic allow you to view WMI information, it also allows you to create scripts to access the

information in several different scripting languages with varying output formats. This nifty tool even allows you to query remote machines.

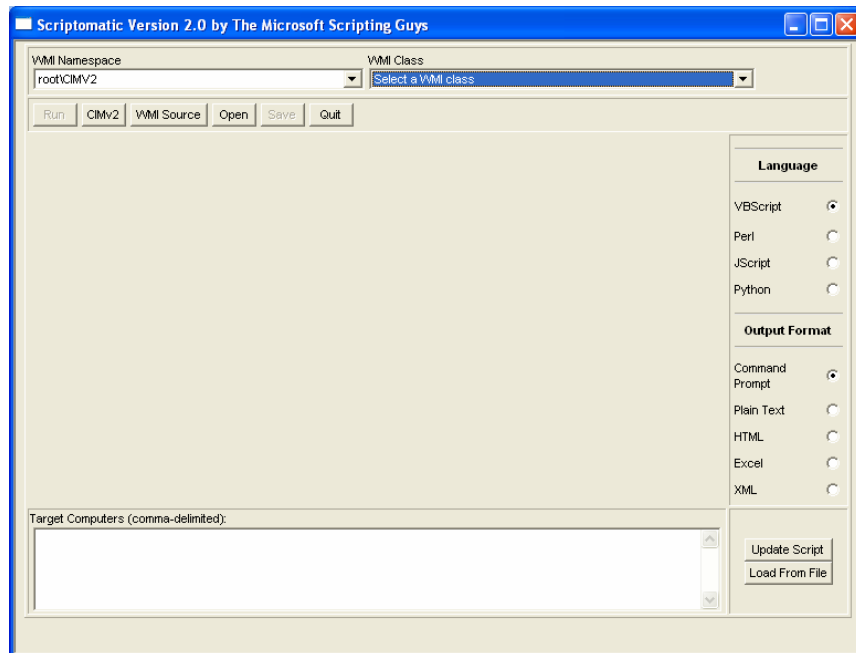
To download the Scriptomatic, just navigate to the link below:

<http://www.microsoft.com/downloads/details.aspx?FamilyID=09dfc342-648b-4119-b7eb-783b0f7d1178&DisplayLang=en>

I won't go into as much detail on the Scriptomatic as I did on WBEMTEST for a couple of reasons. First, it's much more intuitive and easy to understand if you're interested enough to download and play with it. Second, I just want to familiarize you with the different tools available so we can get on with the actual MOF editing in the next chapters—this book would never end if I described each tool in painful detail.

By double clicking on the downloaded ScriptomaticV2.hta file you are presented with something similar to Figure 4.13. Just by looking, you get a hint as to how easy it is to work with it.

Figure 4.13

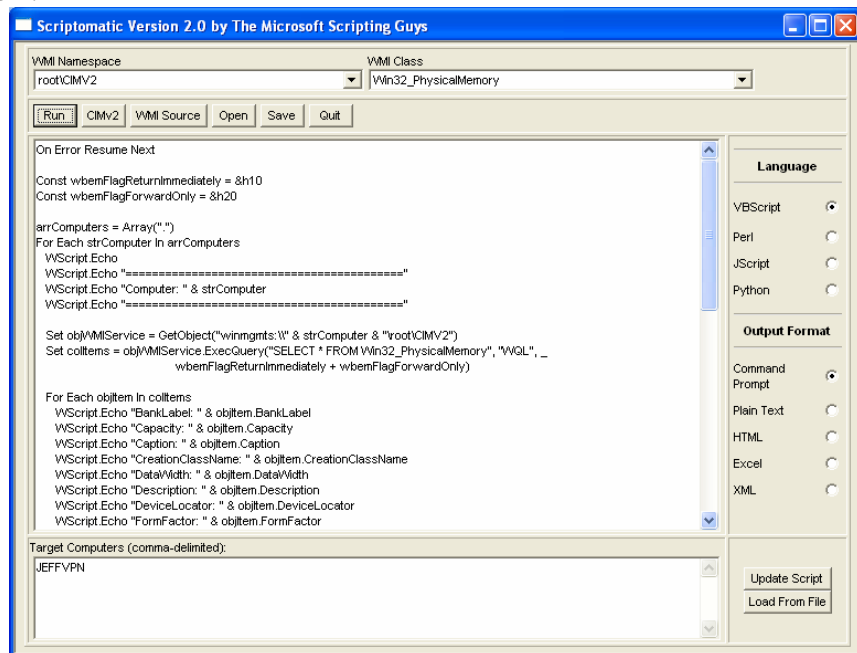


Let's take a quick look at all those handy options the ScriptomaticV2 offers you. This tool is aimed mainly at IT Professionals who want a quick and easy way to write WMI-based scripts.

Take notice of the great variety of scripting languages and output formats available to you. It would take considerably longer to learn all the scripting languages available here than the second or two it takes to decide which options you want to view the WMI information with here.

So by accepting the default WMI Namespace of *root\CIMV2* and selecting our favorite WMI Class—*Win32_PhysicalMemory* again—we get something like Figure 4.14. We don't see any WMI information; we see a script! Because I've left the default scripting language set to VBScript, it's a VB Script written on the fly to access WMI. Hey, that rhymed! Anyway, the scripting capability alone can make this little utility worthy of future study:

Figure 4.14



You can play with the other buttons at your leisure, but right now I'll just accept the default to execute the script with output going to the command prompt. You may notice some familiar information here in Figure 4.15:

Figure 4.15

```

C:\WINDOWS\system32\cmd.exe
Description: Physical Memory
DeviceLocator: DIMM_A
FormFactor: 8
HotSwappable:

InterleaveDataDepth: 0
InterleavePosition: 0
Manufacturer:
MemoryType: 0
Model:
Name: Physical Memory
OtherIdentifyingInfo:
PartNumber:
PositionInRow: 1
PoweredOn:
Removable:
Replaceable:
SerialNumber:
SKU:
Speed: 266
Status:
Tag: Physical Memory 0
TotalWidth: 64
TypeDetail: 128
Version:

```

You see all the fields and whatever information the Scriptomatic was able to get via WMI from your system. You could even change the name of the queried system from the local system to poke around your cubicle farm and check everyone's RAM setup if you get bored.

The Scriptomatic is easy to use, extremely powerful, and versatile. It allows you to quickly query a system, locally or remotely, and makes WMI scripting as easy as selecting a class and pressing run. Download it, run it, and have fun with it. You won't be able to stop for a while so I recommend setting aside a good bit of time before running it for the first time. If you are interested in writing WMI scripts this is also an excellent learning tool.

WMI Code Creator Version 1

Another good utility for accessing WMI is called The WMI Code Creator Tool, Version 1, is basically the Scriptomatic on steroids. This new tool allows you to generate scripts like the Scriptomatic, but the languages it uses are VBScript, C#, and VB .NET.

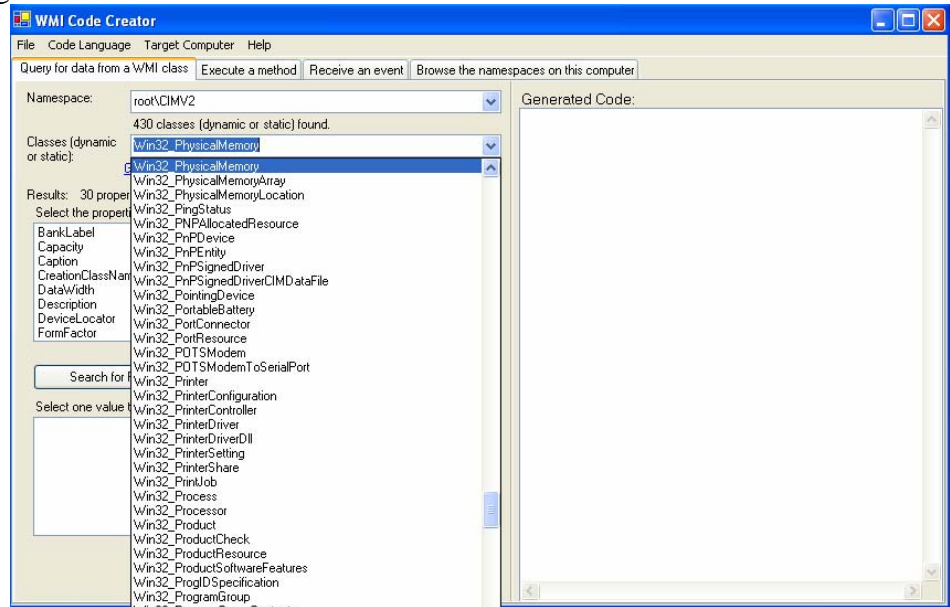
It is a relatively small download for such a powerful utility and it can be found at: <http://www.microsoft.com/downloads/details.aspx?familyid=2CC30A64-EA15-4661-8DA4-55BBC145C30E&displaylang=en>

Not only can the tool be run against a local system or remote system, it even allows you to target a *group* of remote systems to run on simultaneously! Finally! A way to inventory the entire cubicle farm with one click!

Again, just like the Scriptomatic, the WMI Code Creator Tool is an excellent learning tool for writing WMI scripts, this time in C# and VB .NET, as well as regular VBScripting.

One extremely handy feature is that the WMI Code Creator Tool allows you to browse namespaces rather than having to know where they are as WBEMTEST requires as is demonstrated in Figure 4.16:

Figure 4.16



Just select the class you want to query, click the Execute Code button and Viola! You get a Malaysian Leaf Frog ... oh wait, that's not it. You get the Win32_PhysicalMemory information from your system, of course. This time, as you may notice in Figure 4.17, the information has been retrieved via VB .NET instead of regular VBScript:

Figure 4.17

```

C:\WINDOWS\system32\cmd.exe
Microsoft (R) Visual Basic .NET Compiler version 7.10.6001.4
for Microsoft (R) .NET Framework version 1.1.4322.2032
Copyright (C) Microsoft Corporation 1987-2002. All rights reserved.

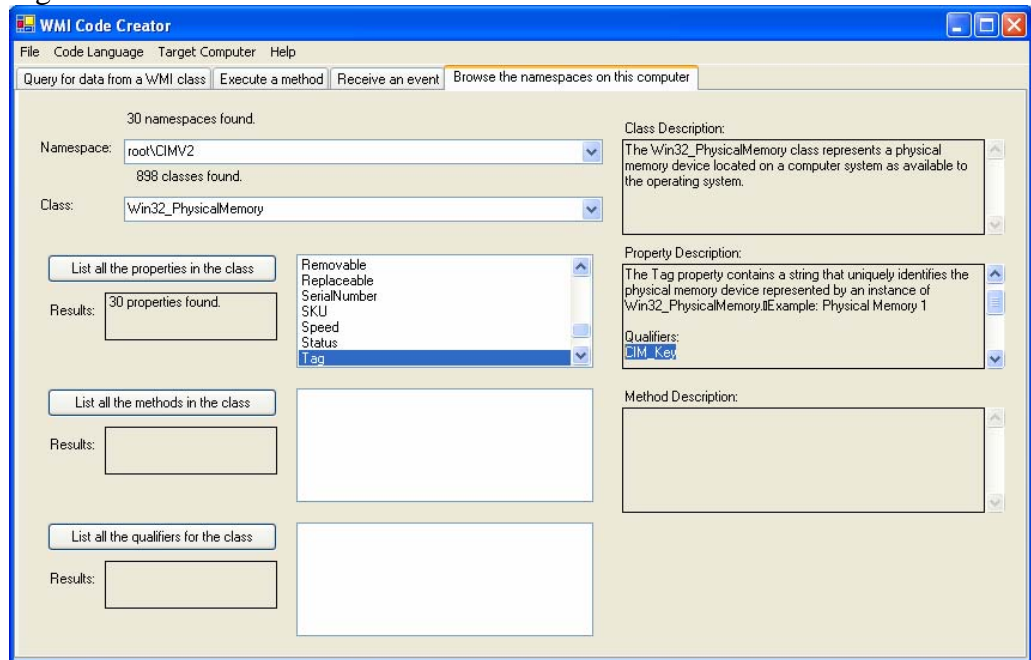
-----
Win32_PhysicalMemory instance
-----
BankLabel:
Capacity: 536870912
Caption: Physical Memory
CreationClassName: Win32_PhysicalMemory
DataWidth: 64
Description: Physical Memory
DeviceLocator: DIMM_A
FormFactor: 8
HotSwappable:
InstallDate:
InterleaveDataDepth: 0
InterleavePosition: 0
Manufacturer:
MemoryType: 0
Model:
Name: Physical Memory
OtherIdentifyingInfo:
PartNumber:

```

Another handy feature of the WMI Code Creator is that it allows you to query the fields in a class and get a detailed report on it by simply clicking on the *Browse the namespaces on this computer tab*.

You can retrieve detailed class and property definitions and values, including qualifiers such as key as shown next. It's still not a replacement for checking the website in my opinion, but awfully close. Most of the information retrieved this way is extremely similar to that found on the MSDN class website.

Figure 4.18



Much like the other tools, if you're interested in accessing WMI information programmatically this will be a fun one to play with. This may just be my new favorite tool.

WMI Manipulation Methods

The time has come to progress beyond rudimentary examples and explore the various methods used in the MOF to open the doors that extract data from the client systems.

This book will cover six methods to manipulate the WMI on a managed system for SMS hardware inventory purposes. Some of these methods can be used in more than one situation, but if you stop to think about it, and analyze what you are trying to do, eventually you will decide upon the correct method—use the power of the MOF Luke.

In the next chapters I'll go into each one with a scenario and suggested use situation to try to help you decide how to go about using the various methods.



As was mentioned earlier, this is not a complete guide on WMI or a guide to advanced MOF scripting. It is meant to be a comprehensive guide to the SMS_def.mof file and its use by SMS to extract data from client systems.

Two men approached a deep, and raging river. The first man begins construction of a raft and collects dead logs, branches, and vines. After two hours of intense labor, the raft is finally ready. With his makeshift paddle in hand and his heart intent on reaching the opposite shore, he pushes the raft into the water.

After an hour of paddling through rough rapids, humongous boulders, and sharp rocks, the man is almost to the riverbank. Finally he sets foot on the soft soil and raises his arms in triumph!

The second man, having walked upstream and crossed the river using the bridge, has been in the pub for two hours.

The moral of the story is: *Sometimes it's easier to follow the **best** path rather than to just head straight toward your goal.*

This is quite often the key to successful MOF and WMI modification. First, analyze your goal, and then determine the best method to reach that goal.

The following are the six methods we will be covering:

1. Reporting on an existing class
2. Pulling registry keys using the Registry **Property** Provider
3. Pulling registry keys using the Registry **Instance** Provider
4. Pulling data using the **View** Provider
5. Static hardware inventory extensions.
6. Scripted hardware inventory extensions

Each of the six methods is best suited to obtain client data in a different manner. Although two methods *may* be able to get the same data, it is better to use the proper method for the job. A brick may drive a nail, but why not use a hammer?

It would be nice if every bit of data from a client could be collected, but sometimes there is data that cannot be retrieved using the existing providers. For example, registry trees with unpredictable key names and structures. This is also the case when the keys have values that the registry providers simply cannot collect, such as the HAL.



In these scenarios, you must write a script that collects the data and writes it to a static MOF (Method 6), directly into a WMI class, or into a MIF to be collected by the hardware inventory (Chapter 10 describes this process).

This is what you've been waiting on—actually getting to the SMS_def.mof modification process. The next six chapters will walk you through using the different methods from start to finish...enjoy!



Chapter Summary

WMI is the Microsoft implementation of WBEM and is used to gain access to the data stored in the Common Information Model Version 2 (CIMV2) repository on a system.

The **CIMV2 repository** on a system can be compared to the windows file system; folders within folders and each containing data relevant to that folder. In other words, namespaces within namespaces and each contain their corresponding instances.

The **#pragma namespace** line is used to navigate throughout the various namespaces within the CIMV2 repository.

Data classes contain the actual WMI data that SMS queries for.

Reporting classes contain the specific data retrieved from data classes that SMS inventories.

Key fields are used by WMI and SMS to identify individual objects within arrays. Key fields are set within the SMS database the first time a new class is added via hardware inventory and cannot be easily changed thereafter.

Using **tools to access WMI** information you can see what data is there and check the success or failure of your MOF modifications. These tools include:

WBEMTEST

Built-in WMI component

CIM Studio

<http://www.microsoft.com/downloads/details.aspx?FamilyID=6430f853-1120-48db-8cc5-f2abdc3ed314&DisplayLang=en>

Scriptomatic version 2

<http://www.microsoft.com/downloads/details.aspx?FamilyID=09dfc342-648b-4119-b7eb-783b0f7d1178&DisplayLang=en>

WMI Code Creator

<http://www.microsoft.com/downloads/details.aspx?familyid=2CC30A64-EA15-4661-8DA4-55BBC145C30E&displaylang=en>

Chapter 5: Reporting on an Existing Class

Even when you know where you are going, the journey is still often filled with surprises.

–Steven R. Webber

The easiest addition to any MOF file is to find an existing WMI data class and add a new reporting class for it to the end of the standard SMS_def.mof file. There are HUNDREDS of classes that are not included in the standard SMS_def.mof file to choose from.

Now don't go thinking you should just skim through this chapter because I said it was the easiest. Even though it's fairly straightforward, there are still some pretty big pitfalls you need to avoid.

Here is a useful example of using the SMS_def.mof to query an existing Win32 class. By adding a reporting class in your MOF for our good buddy, the Win32_PhysicalMemory class, you are able to retrieve physical memory device information from your client systems. This information will allow you to see how many physical RAM chips are installed on a system and in what slots.

Because Win32_PhysicalMemory is an existing Win32 class, the data class is already defined in WMI. In order for SMS to get the data, you just need to create the reporting class in the `root\CIMV2\SMS` namespace to tell hardware inventory what to collect.

To add the Win32_PhysicalMemory reporting class to your default SMS_def.mof just paste the following example *at the very end* of your SMS_def.mof file. Take note of the compound key created by using both the Tag and CreationClassName fields:

```
//-----
// Start of Physical Memory reporting class
//-----
#pragma namespace ("\\.\root\CIMV2\SMS")

[SMS_Report(TRUE),SMS_Group_Name("Win32_PhysicalMemory"),
SMS_Class_ID("MICROSOFT|Win32_PhysicalMemory|1.0")]
class Win32_PhysicalMemory : SMS_Class_Template
{
[SMS_Report (TRUE)] string   BankLabel;
[SMS_Report (TRUE), SMS_Units("Megabytes")] uint64   Capacity;
[SMS_Report (TRUE)] string   Caption;
[SMS_Report (TRUE)] string   DeviceLocator[];
[SMS_Report (TRUE)] uint16   FormFactor;
[SMS_Report (TRUE)] string   Manufacturer;
[SMS_Report (TRUE)] uint16   MemoryType;
[SMS_Report (TRUE)] uint32   PositionInRow;
[SMS_Report (TRUE)] uint32   Speed;
[SMS_Report (TRUE),Key] string   Tag;
[SMS_Report (TRUE),Key] string   CreationClassName;
};
//-----
// End of Physical Memory reporting class
//-----
```



It's a good idea to always add your commented additions to the end of the standard SMS_def.mof file so you can find them later. Even though sitting around skimming through the SMS_def.mof looking for a particular class may make you look busy for your boss, it's really not the most efficient way to go.

Copy, paste, save, and you're all set with a new reporting class in your MOF. When the SMS_def.mof is compiled this class will be added to the local WMI of your SMS clients.

Wait a second. What exactly did that do? The example above is a prime example of a reporting class that was not in the default SMS_def.mof file, but could add useful information. If you look at the field names, you'll see Capacity, Caption, Manufacturer, Memory Type, etc. This information may prove valuable to you as an SMS administrator.

Okay, I know what a reporting class is and what it does, but how did you even create that class? Reporting classes look **very** similar. They all have the same header structure followed by a list of field names set either to "TRUE" or "FALSE". Therefore, if I find a class I want to use on the Microsoft website, or in the WMI repository itself, I can create my own class by following the same structure of an existing class.

Structure of a Reporting Class

Below is an example of a “trimmed down” physical memory reporting class:

```

LINE 1 //-----
LINE 2 // Start of Physical Memory reporting class
LINE 3 //-----
LINE 4 #pragma namespace ("\\.\root\CIMV2\SMS")

LINE 5
[SMS_Report(TRUE),SMS_Group_Name("Win32_PhysicalMemory"),
LINE 6 SMS_Class_ID("MICROSOFT|Win32_PhysicalMemory|1.0")]
LINE 7 class Win32_PhysicalMemory : SMS_Class_Template
LINE 8 {
LINE 9 [SMS_Report (TRUE)] string DeviceLocator[];
LINE 10 [SMS_Report (TRUE),Key] string Tag;
LINE 11 };

```

Lines 1-3:

```

LINE 1 //-----
LINE 2 // Start of Physical Memory reporting class
LINE 3 //-----

```

Comments, comments, comments. Whenever you modify the SMS_def.mof you need to comment the beginning and ending appropriately. Just scrolling through the MOF you’ll notice everything pretty much looks the same. Finding this section a month from now would be difficult without the comments telling you what you did. These comments are for example only; usually you’ll want to add more descriptive information. Maybe you will want to know the exact date you started this modification, who initiated it, and who was the last person to modify it. Adjust your commenting policy to fit whatever your personal situation is and requirements from your employer if applicable.

Line 4:

```
#pragma namespace ("\\.\root\CIMV2\SMS")
```

This line declares which namespace the following data should be added to. It only needs to be done once to change the “destination”. For example, if you change to Channel 6 on your TV, until you change the channel, you’ll always be watching Channel 6. Until you change the namespace, all data will continue to go into the last namespace chosen. In most cases, if the last class in the MOF was a reporting class,

you do not need this line, as the “Channel” is already set to the proper namespace:
`root\CIMV2\SMS.`

Lines 5-6:

```
LINE 5 [SMS_Report(TRUE),SMS_Group_Name("Win32_PhysicalMemory"),
LINE 6 SMS_Class_ID("MICROSOFT|Win32_PhysicalMemory|1.0")]
```

The next line begins the Header section of the class declaration, enclosed in brackets [and]. Inside the Header are typically three properties:

- The Class Level Reporting Property
- The SMS Group Name
- The SMS Class ID

Line 5:

```
[SMS_Report(TRUE),
```

The class level reporting property. This line determines whether the hardware inventory will collect *any* information on the class at all. If it's TRUE, then the class will be collected by the hardware inventory process. If it's set to FALSE, then the entire class will be excluded.

Line 5:

```
SMS_Group_Name("Win32_PhysicalMemory"),
```

This is the name of the group that will appear in the resource explorer for your systems, and when you are looking to do queries on this class. The name of the group can be any name you choose, but I recommend sticking close to the actual name of the class to keep things from getting confusing. If your class name was “BoogaBooga” and your Group Name was set as “My Company User Information”, you're asking for trouble.

The group name is also used when SMS initially creates the table in the SQL database. If you have a class created with an SMS_GROUP_NAME of "Mikes Registry Information" with 3 string fields, you'll have an SMS table created called

"Mikes_Registry_Inform_DATA" (21 characters of your group name will be used, including spaces converted to underscores) with 3 string fields.



It only takes one machine with one new class in its repository to add two tables, two stored procedures, and three entries into the WMI on your SMS site server. Be wary when you experiment!

Line 6:

```
SMS_Class_ID("MICROSOFT|Win32_PhysicalMemory|1.0")]
```

This line can be viewed as the “main” line for the entire class. The class ID is the only property in the class that **must** be unique. If you created three classes with the Group Name of “User Information” that would be acceptable (not recommended, but acceptable). However, the Class ID for each of these classes *must be unique*. One class may have a Class ID of “User Information” and one may have “User Data” and one may have “Microsoft|User_Info|1.0”. However, the only requirement is that they be unique.

You may want to change the Company name and version number (“Microsoft|” and “|1.0”) to keep track of which reporting classes were default from Microsoft, and which you added yourself.

Do I have to include the “Microsoft |” and “|1.0” around my class ID? No. This is a misconception that hasn’t been properly clarified. Having “Microsoft|” before a class ID is not significant, and could be replaced with “Christmas|”, or even removed altogether. The class ID only has to be unique.

Many administrators will use the “Microsoft|” and “|1.0” when they are creating reporting classes for existing Win32 data classes. However, I like to change the company name section from these items to keep track of which reporting classes were default from Microsoft and which ones I added myself. An example of doing that would be:

```
SMS_Class_ID("SMSExpert|Win32_PhysicalMemory|1.0")]
```

That way I know I added this reporting class. I could even update the “1.0” part to “2.0” whenever I modified this reporting class in the future.

Line 7:

```
class Win32_PhysicalMemory : SMS_Class_Template
```

Line 7 is the beginning of the actual reporting class declaration. The name of the reporting class *must* be identical to the name of the data class that it is reporting on. If you are creating a reporting class for an existing data class, there is no room for error. However, if you are creating a reporting class for a data class that *you* created, then you can change the name of both the reporting class and data class to whatever you want.

If your data class is Win32_Christmas, then *your* line 7 would be:

```
class Win32_Christmas : SMS_Class_Template
```

Lines 8 and 11: { and };

As silly as it sounds, these two lines are very important and can be the hardest to troubleshoot if you miss them. Before the “innards” of a class can be defined, you have to put a left curly brace “{”, and after you are finished you must put a right curly brace with a semicolon “};”. These mark the beginning and end of the entire class declaration.

Lines 9 and 10:

```
[SMS_Report (TRUE)] string DeviceLocator;  
[SMS_Report (TRUE),Key] string Tag;
```

Remember from way back in Chapter 2 where I said these were field level reporting properties? Surprise, they’re still field level reporting properties! One of these is special though; notice that line 10 has another word in there after TRUE. This is a “key” thing of which to take notice when you’re defining a reporting class.

Before I continue, I need to quickly explain what these lines do.

Line 9:

```
[SMS_Report (TRUE)] string DeviceLocator;
```

This is the field level reporting property line for the WMI field called DeviceLocator within the Win32_PhysicalMemory class. This line tells the SMS Inventory Agent to pay attention to this field and record its information in the SMS database. If this was set to FALSE I’m betting you can tell me what happens, since I’ve pretty much beaten this dead horse enough already!

Line 10:

```
[SMS_Report (TRUE),Key] string Tag;
```

A line such as line 10 is typically, but not always, found only once in a reporting class. Not only does it state that this field should be reported, but it also states that the field is a *key field*. This key field is important when creating reporting classes for data classes with multiple instances.

If a data class has multiple fields with the key qualifier, then all those key fields taken together form what is called a compound key. The Win32_PhysicalMemory class is a prime example of a compound key in action. Take another look at the Win32_PhysicalMemory reporting class and see if you can identify the compound key:

```
//-----
// Start of Physical Memory reporting class
//-----
#pragma namespace ("\\\\.\\root\\CIMV2\\SMS")

[SMS_Report(TRUE),SMS_Group_Name("Win32_PhysicalMemory"),
SMS_Class_ID("MICROSOFT|Win32_PhysicalMemory|1.0")]
class Win32_PhysicalMemory : SMS_Class_Template
{
[SMS_Report (TRUE)] string BankLabel;
[SMS_Report (TRUE), SMS_Units("Megabytes")] uint64 Capacity;
[SMS_Report (TRUE)] string Caption;
[SMS_Report (TRUE)] string DeviceLocator[];
[SMS_Report (TRUE)] uint16 FormFactor;
[SMS_Report (TRUE)] string Manufacturer;
[SMS_Report (TRUE)] uint16 MemoryType;
[SMS_Report (TRUE)] uint32 PositionInRow;
[SMS_Report (TRUE)] uint32 Speed;
[SMS_Report (TRUE),Key] string Tag;
[SMS_Report (TRUE),Key] string CreationClassName;
};
//-----
// End of Physical Memory reporting class
//-----
```

Now you see it, right? Both the Tag and CreationClassName fields have “Key” in their description. These two fields form the compound key for the Win32_PhysicalMemory Class.

If you are attempting to inventory the above mentioned Win32_PhysicalMemory class to see how many memory chips your system contains, and create the MOF

modifications using only one key tag, you will never get proper information in your data table for the class.

Hey! I just looked up the Win32_PhysicalMemory Class from that link you gave me and there are a lot more data fields than the ones you showed in your example. What gives?

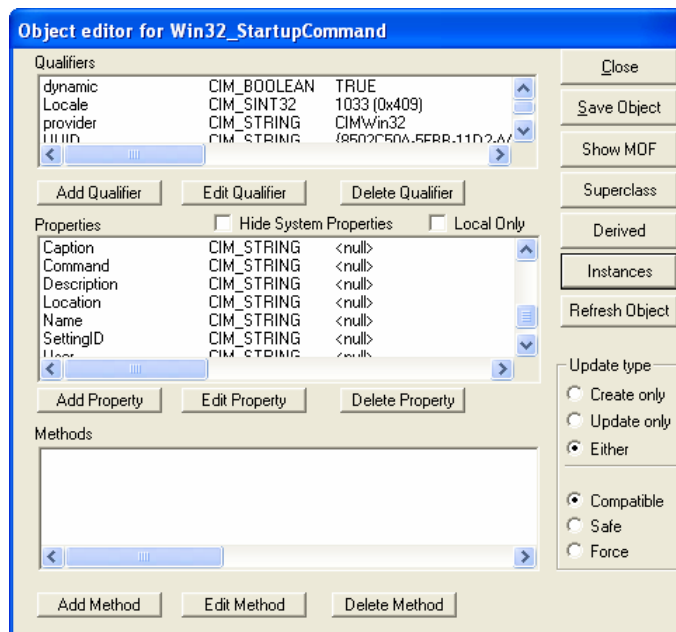
Would I ask you a rhetorical question? OK, OK I'll get serious now. Just because there are many possible fields for each class doesn't mean that you have to list them all in your SMS_def.mof. You only need to ensure you have all the key fields and the fields that you find useful to your purposes.

Well, since you look bored sitting there listening to me talk about the Win32_PhysicalMemory Class—and the guy in the back is actually snoring—let's go through the process of reporting on an existing class together.

Someone pick a WMI Class for us to report on ... you sir, with the red shirt on, you had your hand up first. What's that? Win32_StartupCommand? All right, that's a good one because that isn't in the default SMS_def.mof.

First, let's see what the selling points of this class are, and if we are sure we want to add it to our SMS_def.mof. First step, open up WBEMTEST and see what's in there.

Figure 5.1



Hmm that looks interesting! Viewing instances for this class shows a whole bunch of gobbly-goop that looks like program names and executable paths. Now I'm curious

enough to visit the MSDN website to see the particulars of this class. So let's go see what's at:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/win32_startupcommand.asp

Just in case you're not near a computer right now, here's what the site has to say about it:

The Win32_StartupCommand WMI class represents a command that runs automatically when a user logs onto the computer system.

The following syntax is simplified from MOF code and includes all inherited properties.

```
class Win32_StartupCommand : CIM_Setting
{
string Caption;
string Command;
string Description;
string Location;
string Name;
string SettingID;
string User;
string UserSID[];
};
```

So this class is where we look for commands that run automatically when a user logs onto a system. Maybe this would be a good place to look for adware or spyware. Even if you wouldn't want this in your MOF, the practicality of it isn't important right now; what is important is that we illustrate this method.

Look at that part in bold. Looks pretty familiar, doesn't it? You've almost got a MOF modification for this class already. Another thing about that part in bold is that it looks like there's a new field in there that wasn't in our WBEMTEST display, the 'string UserSID[];' part. Don't ask me about that. I just follow what is posted on MSDN's site, and it hasn't failed me yet.

The UserSID field is used to match up an SID to a user for whom startup commands run. This is how you determine who the user is for whom the program is set to start.

How'd I figure that out so quickly? Maybe you didn't check out the UserSID field on the class from the MSDN website, so no extra credit for you! This is a perfect example of why I cannot stress enough the importance of checking that site for any class you want to add. Had we gone straight from WBEMTEST's displayed information we wouldn't have had that field in the MOF and we wouldn't know what was going on with some bizarre side effect caused by trying to match up a startup command with a deleted user.



Maybe you also noticed the [] symbol following the UserSID field. The [] symbol means the data from that field is stored as an array. Multiple users = an array of SIDS.

Counting the UserSID field, there are eight reporting fields we can use for the Win32_StartupCommand Class. So, in this case, we won't be picky about the fields we want, and we'll just use them all.

The first step would be to copy an existing reporting class and paste it into a new text document. Then modify that reporting class to contain data for this new class going off the MSDN website and what we learned about it with WBEMTEST.



Copying an existing reporting class helps to maintain the format and syntax without having to memorize everything.

Any reporting class will do for the most part, just pick one at random and play the 'which words don't belong' game.

The end result after copying and modifying one of the existing reporting class structures to accommodate the Win32_StartupCommand Class is shown below: (the parts I've changed to oblige the Win32_StartupCommand Class are in bold red):

```
//-----
// Start of Startup Command reporting class
//-----
#pragma namespace ("\\\\.\\root\\CIMV2\\SMS")

[SMS_Report(TRUE),SMS_Group_Name("Win32_StartupCommand"),
SMS_Class_ID("MICROSOFT|Win32_StartupCommand|1.0")]
class Win32_StartupCommand : SMS_Class_Template
{
[SMS_Report (TRUE)] string Caption;
[SMS_Report (TRUE),Key] string Command;
[SMS_Report (TRUE)] string Description;
[SMS_Report (TRUE),Key] string Location;
[SMS_Report (TRUE),Key] string Name;
[SMS_Report (TRUE)] string SettingID;
[SMS_Report (TRUE),Key] string User;
[SMS_Report (TRUE)] string UserSID[];
;
//-----
// End of StartupCommand reporting class
//-----
```


Whoa! Look at all those key fields! This is another example of a compound key for a WMI Class. Lucky us! We wouldn't have known this without dutifully checking the MSDN site for this class's properties.

Another bad thing could have happened. We could have decided we didn't want to use some of these fields and, unknown to us, they turned out to be key fields. Have I made the point enough for you to use that MSDN site?



When viewing the class information on the web, it may note at the bottom of the page that there are prerequisite MOF files that must be compiled on the clients to obtain the information successfully. With WMI 1.5, this should not be necessary for most classes, but it is still a potential issue.

What other data types other than string can you use?

There are quite a few different values that can be used in the MOF:

- Uint8, Uint16, Uint32, Uint64 = 8, 16, 32 and 64 bit Unsigned Integers
- String = String of Characters
- Boolean = True or False, Yes or No, 1 or 0
- Sint8, Sint16, Sint32, Sint64 = 8, 16, 32, and 64 bit Signed Integers
- Real8, Real16, Real32, Real64 = 8, 16, 32, and 64 bit Real Numbers
- DateTime = Date and Time value

Why do I sometimes see an additional line in the Header with "SMS_rxpl.dll"?

You're referring to the line: ResID(600),ResDLL("SMS_RXPL.dll"). This line exists in the reporting classes that already are included in the SMS_def.mof file.

The purpose of the line is to identify this class to the hardware inventory based upon its ResID in SMS_rxpl.dll. This enables the fields from that reporting class to be labeled appropriately, with spaces.

For example, if you create a reporting class without this line, all of the fields for that class will be labeled with their field name; VideoAdapterConfiguration will be labeled "VideoAdapterConfiguration". However, if you identify this reporting class with its appropriate ResID in the SMS_RXPL.dll, VideoAdapterConfiguration will be labeled as "Video Adapter Configuration". A very minor difference, but sometimes it is the little things that make it easier to read!



SMS_rxpl.dll can be explored using Microsoft Visual Studio to identify the other ResIDs.

As you can see, it's very easy to take an existing data class and create your own reporting class. Just change a few items in a reporting class, cut and paste a few lines, and in minutes, you'll have this data reporting to SMS!. Don't ever tell anyone it's that easy or we can all say good-bye to our future raises!

Either that or just convince them that you found the data using one of the harder methods we'll talk about next! That didn't scare you, did it?



Chapter Summary

Reporting on existing data class is the easiest MOF modification method. This method simply involves creating a new reporting class to tell SMS to report it during hardware inventory.

Because reporting classes are very similar to each other, **the easiest way to create a custom reporting class is to copy and paste an existing reporting class and modify it** to fit your specific needs.

Use comments to mark the beginning and end of any new custom MOF modification. This allows you to see what you have added and why.

Changing namespaces only needs to be done when you switch from a data class to a reporting class. **As long as the last namespace change was `#pragma namespace ("| | | . | | root | | CIMV2 | | SMS")` you can add as many new reporting classes as you choose without changing the namespace again.** If you need to add an additional data class after previous reporting classes, be sure to change the namespace back to CIMV2 by using the `#pragma namespace ("^\\|\\| . \\root\\|\\CIMV2")` command.

Class level reporting properties determine whether hardware inventory will collect information for the class.

SMS Group Names are the names for classes as they appear in Resource explorer for client systems, as well as the table names for the class in the SQL database.

Class ID's must be unique for each class created. You may want to change the company name and version number of class ID's ("Microsoft|" and "|1.0") to keep track of which reporting classes were default from Microsoft, and which you added yourself.

Reporting class names must be identical to their corresponding data classes.

Compound keys are created by identifying two or more field level properties as Key fields.

There are quite a few different data type values that can be used in the MOF:

- Uint8, Uint16, Uint32, Uint64 = 8, 16, 32 and 64 bit Unsigned Integers
- String = String of Characters
- Boolean = True or False, Yes or No, 1 or 0
- Sint8, Sint16, Sint32, Sint64 = 8, 16, 32, and 64 bit Signed Integers
- Real8, Real16, Real32, Real64 = 8, 16, 32, and 64 bit Real Numbers
- DateTime = Date and Time value

The **SMS_rxpl.dll** file can be explored using Microsoft Visual Studio to identify the other ResIDs. These ResIDs are used by SMS to enable spaces in data classes.

Chapter 6: Registry Property Provider

You cannot put the same shoe on every foot.

-Publius Syrus (c. 42 B.C.)

Have you ever found yourself perusing MOF examples on the web and finding two that inventory similar registry information but look entirely different? It could be because one works and one doesn't, however it's much more likely that they are using two different providers to get to the same information.

The Registry Property Provider (RPP) and the Registry Instance Provider (RIP) are commonly confused. While they both can be used to pull *most* single registry keys, they each have their own unique purpose and capabilities.

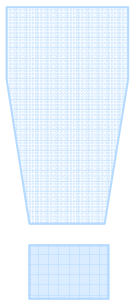
This chapter is about the Registry Property Provider (RPP). I've decided to make the different registry providers completely separate chapters just to help make a distinction between the two, and to help you better understand them

In a short summary, the Registry Property Provider excels at grabbing a variety of keys from a variety of *known* locations and adding them all to one reporting class if necessary. The Registry Instance Provider excels at grabbing a variety of *unknown* keys from a *single location*.

Take a moment and repeat that last line a couple of times to yourself just to let it sink in ... trust me, it will help.

When to Use the Registry Property Provider

The beauty of the Registry Property Provider is that it looks through *different* registry locations, for *specific* keys, and returns all that data into *one common data class*.

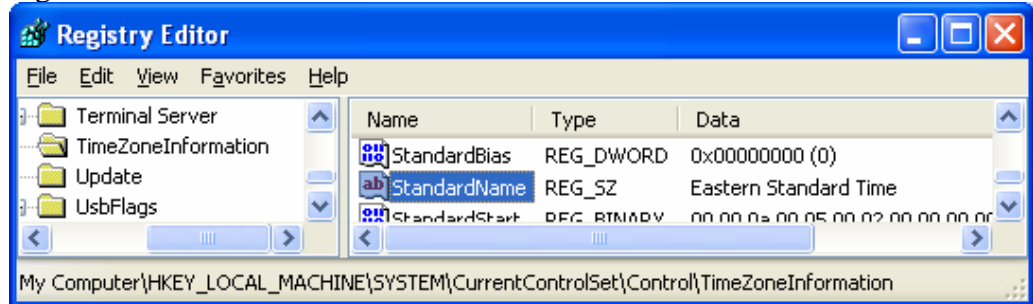


Here's an easy example: Let's suppose you want to use the RPP to find the time zone names for each of your SMS clients. You know exactly where this information is in the registry, and you know the exact name of the key this information resides in. In case you forgot that you knew that, here is a picture to help refresh your memory:



Yes, the Time Zone information is already in the standard SMS 2003 SP1 SMS_def.mof. I'm just using this as an example for the RPP here. Bear with me.

Figure 6.1



Because you know the exact location in the registry, and the exact key name where the data resides, this makes a perfect scenario for using the Registry Property Provider.

For SMS 2003, we don't have to define the Registry Property Provider itself because it is already defined at the top of the standard SMS_def.mof. For SMS 2.0 you'll need to copy and paste the below into your SMS_def.mof to register the provider before using it:



```
// Registry property provider
instance of __Win32Provider as $PropProv
{
    Name = "RegPropProv" ;
    CLSID = "{72967901-68EC-11d0-B729-00AA0062CBB7}";
    ImpersonationLevel = 1;
    PerUserInitialization = "False";
};

instance of __PropertyProviderRegistration
{
    Provider = $PropProv;
    SupportsPut = True;
    SupportsGet = True;
};
```

For those wondering what is actually done in those 15 lines, here you go:

The namespace is changed to *root\CIMV2*. An instance of `__Win32Provider` is declared with an ID of `$PropProv`, a Name of “RegPropProv”, and using the class ID for the Property Provider. Then, the provider is actually registered by creating an instance of the `__PropertyProviderRegistration` class, using the ID of the provider in the instance above it, `$PropProv`, and with the properties of Put and Get set to TRUE, meaning that data can be added or extracted from the class using this provider.

Wow, that was a mouthful huh? Let’s move on.

Using the RPP you would define the `TimeZoneInfo` data class and declare an instance of the data class to store the time zone information. Let’s take a look at all three steps in more detail.

Before we can get to the “good stuff” we need to make sure we’re in the correct namespace for a data class which would entail the below first line of the MOF edit:

```
#pragma namespace("\\\\.\\root\CIMV2")
```

Now, the “good stuff”. We need to define the `TimeZoneInfo` data class. To do so we use the below lines:

```
[DYNPROPS]
class TimeZoneInfo
{
    [key] string KeyName;
    string Name;
};
```

Notice the `[DYNPROPS]` line? That tells MOFCOMP that we’re about to go after some dynamic data maintained by a provider. It’s patient, it will wait for us to name the provider in the next few lines. The next lines define the name of the class, the property names and data types for each.

```
[DYNPROPS]
instance of TimeZoneInfo
{
    KeyName = "TimeZoneInfo";
    [PropertyContext("local\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\TimeZoneInformation\StandardName"),
    Dynamic,Provider("RegPropProv")] Name;
};
```

Be careful of the effects of word wrap in these examples, sometimes there just isn’t room between the margins to display it all out right—that registry key should extend all the way to the end here.

There's the [DYNPROPS] line again, denoting the dynamic data we're about to define—the instance of the `TimeZoneInfo` class. The `KeyName` is just some static information that serves as a `Key` field for the class. Naming this `KeyName` is a personal choice; you could name it "MyFlipFlop" and it would serve the same purpose. Just remember if you call it `MyFlipFlop` here you have to make sure it is defined as `MyFlipFlop` in the class definition above as well.

Next come the `PropertyContext` line(s). This is the line that puts the registry in Registry Property Provider. Just define the registry branch you want to query with a | symbol in front of the particular key you're after. Directly after the registry key information you'll notice a line like: `Dynamic,Provider("RegPropProv") [name];`. This tells MOFCOMP that this is a dynamic registry location and the way to get to it is to use the RPP or `RegPropProv` as the `SMS_def.mof` calls it. Right after the provider information you give the registry data stored in that location a name that you want to see in your resource explorer.

Here I'm only touching on one particular key. You could continue on with as many keys as you wanted anywhere in the registry as long as you define the correct path and key name. Just add new `PropertyContext` lines completed as above with paths for each registry key you want to peer into, and define a name to make this key your own. Just remember that when you do this, all of this data will appear in the `TimeZoneInfo` class in Resource Explorer.

On to the reporting class ...

```
#pragma namespace("\\\\.\\root\\CIMV2\\SMS")

[SMS_Report(TRUE),SMS_Group_Name("TimeZoneInfo"),SMS_Class_ID("MICROSOFT|TimeZoneInfo|1.0")]
class TimeZoneInfo : SMS_Class_Template
{
    [SMS_Report(TRUE),key] string KeyName;
    [SMS_Report(TRUE)] string Name;
};
```

I'm sure this looks familiar to you at this point. It's the reporting class for our `TimeZoneInfo` data class. Switch to the `root\CIMV2\SMS` namespace to declare the reporting class, name your resource explorer group name after the class name—*exactly*, and then define the reporting properties for the fields you've already defined, a.k.a..the names you've given the data from the `PropertyContext` lines.

Remember, this new `TimeZoneInfo` data class is not present by default in the WMI repositories of your clients. For SMS 2.0 just update the `SMS_def.mof` on the site server and the clients will get the new information. For SMS 2003 clients, you'll have

to compile this new MOF addition on each of them locally to get the class into WMI before hardware inventory will report the time zone information to SMS.

Add in the comment lines so that everyone who looks at your SMS_def.mof file when you're finished—including yourself a month from now—will understand your fascination with discovering the names of the time zone your systems are using. Your MOF edit will look something like this:

```
//-----
//Start of Time Zone Information
//-----
#pragma namespace("\\\\.\\root\\CIMV2")

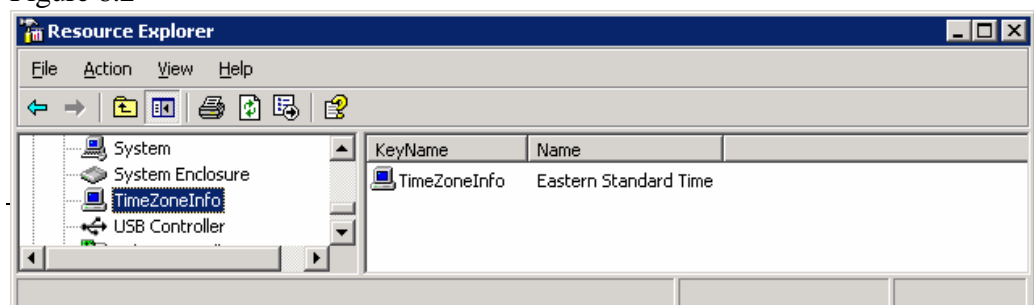
[DYNPROPS]
class TimeZoneInfo
{
    [key] string KeyName;
    string Name;
};

[DYNPROPS]
instance of TimeZoneInfo
{
    KeyName = "TimeZoneInfo";
    [PropertyContext("local|HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Control\\TimeZoneInformation|StandardName"),
    Dynamic,Provider("RegPropProv")] Name;
};

#pragma namespace("\\\\.\\root\\CIMV2\\SMS")
[SMS_Report(TRUE),SMS_Group_Name("TimeZoneInfo"),SMS_Class_ID("MICROSOFT|TimeZoneInfo|1.0")]
class TimeZoneInfo : SMS_Class_Template
{
    [SMS_Report(TRUE),key] string KeyName;
    [SMS_Report(TRUE)] string Name;
};
//-----
//End of Time Zone Information
//-----
```

Using this MOF edit your resulting SMS console will display some nifty time zone information for your systems similar to Figure 6.2:

Figure 6.2



Notice in Figure 6.2 how the names you've chosen appear in resource explorer. The KeyName field appears right beside our lonely PropertyContext line we called Name.

I've only queried one registry key here for simplicity sake, but the real beauty of the RPP is that it is capable of getting multiple instances of data from the same explicitly defined registry branch locations to drag into a single data class for reporting.

All right, I can understand if that didn't make too much sense. How about a more complicated example?

There is an excellent example of the RPP in action within the default SMS 2003 SMS_def.mof. This Microsoft supplied example comes in the form of SMS Client State. This class collects the SMS client component name, pending time, pending version, state, and version as instances of SMS Client State. Instead of creating a new class for every separate component, all the data pertaining to the SMS Client State can be found in a single location in the database.

This is an extremely long MOF edit so I'll only show you a glimpse of it. All you really need to know is that a data class is declared—SMS Client State—and each of the components are declared as instances of SMS Client State. They have put the reporting class first, and then declared the many instances of the class below in this example.



Remember, the order in which the reporting class and data class are created is not important. I recommend always declaring the data class, instances of the data class, and then reporting class for simplicity sake. In reality though, MOFCOMP doesn't care in what order these classes are created, and hardware inventory only cares that the class exists when it is asked to query a reporting class to see the instances of the data that is stored there.

Because this is such a long MOF edit I've taken out all but the first two instances of SMS Client State, but you should be able to pick out the pattern in the excerpt below and recognize the syntax for the RPP by now.

```
//-----
// SMS Client State
//-----

#pragma namespace ("\\\\.\\root\\CIMV2\\SMS")

// Declare the SMS delta/reporting class for standard client components
[ SMS_Report (FALSE),
  SMS_Group_Name ("SMS Client State"),
```

```

SMS_Class_ID ("MICROSOFT|SMS_CLIENT_STATE|1.0") ]

class Win32Reg_SMSClientState : SMS_Class_Template
{
    [SMS_Report(TRUE),key]
    string Component;
    [SMS_Report(TRUE)]
    string State;
    [SMS_Report(TRUE)]
    string Version;
    [SMS_Report(TRUE)]
    string PendingVersion;
    [SMS_Report(TRUE)]
    string PendingTime;
};

#pragma namespace ("\\\\.\\root\\CIMV2")

// Declare the class for client component registry properties

[DYNPROPS]
class Win32Reg_SMSClientState
{
    [key]
    string Component = "";

    string State;
    string Version;
    string PendingVersion;
    string PendingTime;
};

// Declare the instances, one for each client component

[DYNPROPS]
instance of Win32Reg_SMSClientState
{
    Component="SMS Client Base Components";

    [PropertyContext("local|HKEY_LOCAL_MACHINE\\Software\\Microsoft\\
SMS\\Client\\Client Components\\SMS Client Base
Components\\Installation Properties|SMS Client Installation State"),
    Dynamic, Provider("RegPropProv")]
    State;
    [PropertyContext("local|HKEY_LOCAL_MACHINE\\Software\\Microsoft\\
SMS\\Client\\Client Components\\SMS Client Base
Components\\Installation Properties|Installed Version"),
    Dynamic, Provider("RegPropProv")]
    Version;
};

```

```
[PropertyContext("local|HKEY_LOCAL_MACHINE\\Software\\Microsoft\\
SMS\\Client\\Client Components\\SMS Client Base
Components\\Installation Properties|Pending Operation Version"),
    Dynamic, Provider("RegPropProv")]
    PendingVersion;
```

```
[PropertyContext("local|HKEY_LOCAL_MACHINE\\Software\\Microsoft\\
SMS\\Client\\Client Components\\SMS Client Base
Components\\Installation Properties|Pending Operation Time"),
    Dynamic, Provider("RegPropProv")]
    PendingTime;
};
```

```
[DYNPROPS]
instance of Win32Reg_SMSClientState
{
    Component="Available Programs Manager Win32";
```

```
[PropertyContext("local|HKEY_LOCAL_MACHINE\\Software\\Microsoft\\
SMS\\Client\\Client Components\\Available Programs Manager
Win32\\Installation Properties|SMS Client Installation State"),
    Dynamic, Provider("RegPropProv")]
    State;
```

```
[PropertyContext("local|HKEY_LOCAL_MACHINE\\Software\\Microsoft\\
SMS\\Client\\Client Components\\Available Programs Manager
Win32\\Installation Properties|Installed Version"),
    Dynamic, Provider("RegPropProv")]
    Version;
```

```
[PropertyContext("local|HKEY_LOCAL_MACHINE\\Software\\Microsoft\\
SMS\\Client\\Client Components\\Available Programs Manager
Win32\\Installation Properties|Pending Operation Version"),
    Dynamic, Provider("RegPropProv")]
    PendingVersion;
```

```
[PropertyContext("local|HKEY_LOCAL_MACHINE\\Software\\Microsoft\\
SMS\\Client\\Client Components\\Available Programs Manager
Win32\\Installation Properties|Pending Operation Time"),
    Dynamic, Provider("RegPropProv")]
    PendingTime;
```

```
};
.
.
```

Viewed from resource explorer, the complete data class (with all the instances) looks like Figure 6.3:

Figure 6.3

The screenshot shows the Resource Explorer window with the 'SMS Client State' class expanded. The main pane displays a table of instances with the following columns: Component, PendingTime, PendingVersion, State, and Version.

Component	PendingTime	PendingVersion	State	Version
Available Programs Manager Win32			Installed	2.00.1493.3010
Hardware Inventory Agent			Installed	2.00.1493.3010
License Metering				
NT Event To SNMP Trap Translator				
Remote Control			Installed	2.00.1493.3010
SMS Client Base Components			Installed	2.00.1493.3010
Software Distribution			Installed	2.00.1493.3010
Software Inventory Agent			Installed	2.00.1493.3010
Windows Management		1085.0005	Installed	1085.0005
SMS 1.x Client Migration (if required)	n/a	n/a		n/a
SMS 2.0 Client Upgrade (TRUE is Disabled)	n/a	n/a		n/a
SMS 2.0 Pre-SP2 Client Upgrade (TRUE is Disabled)	n/a	n/a		n/a

See how each instance is listed under the main SMS Client State class? The same data class reporting fields are utilized for each instance of the class.



Chapter Summary

The **Registry Property Provider (RPP)** excels at grabbing a variety of keys from a variety of known locations and adding them all to one reporting class.

The **RPP is already registered in the default SMS_def.mof for SMS 2003**. For SMS 2.0 sites you need to add in the registration information yourself before using it.

The **RPP's PropertyContext line** identifies the exact registry key to search in by using the | symbol at the end of the registry path:

```
[PropertyContext("local|HKEY_LOCAL_MACHINE\\Software\\Microsoft\\SMS\\Client\\Client Components\\Available Programs Manager Win32\\Installation Properties|Pending Operation Version")
```

When adding new data classes with the RPP you need to compile the MOF modification on Advanced Clients to get the new data class information into their WMI repositories.

The order in which the reporting class and data class are created is not important. I recommend always declaring the data class, instances of the data class, and then reporting class for simplicity sake.

Chapter 7: The Registry Instance Provider

Life is like playing a violin solo in public and learning the instrument as one goes along.

–Samuel Butler

The Registry Instance Provider (RIP) doesn't need to know the exact names of the keys it is pulling like the RPP does. It simply needs to know the format of the key structure. As long as you point it in the general direction, this provider will query everything under the main registry key you send it to looking for subkey names that you've specified as inventory properties.

When to Use the Registry Instance Provider

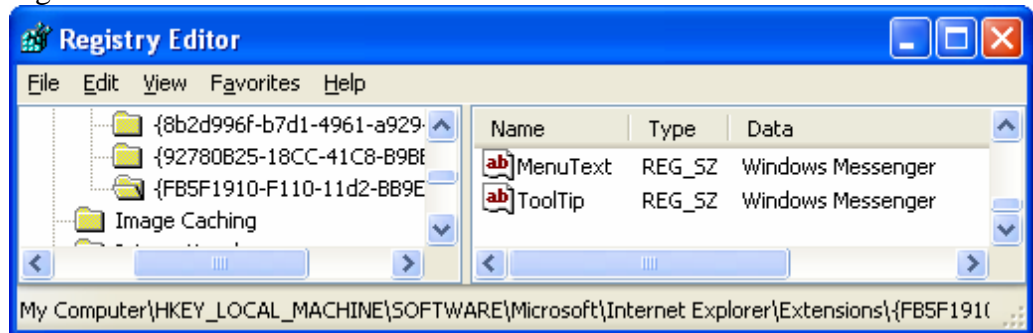
The best use for the RIP is for inventorying various subkeys under one main registry key that all have similar data values that you want to inventory without creating more than one reporting class.

Think of the RIP as a nosy neighbor peeking through windows. If you tell it to look in a kitchen window and report back on all the knives it can see, it will. The RPP, in comparison, would only tell you about knives from a specific drawer.

What if you wanted to see all the installed add-ons to Internet Explorer on your systems to check for spyware or adware? You would have to use the RIP to recourse through the *Internet Explorer\Extensions* Keys to find that information. You know that the main area in the registry you're going to look in is: *HKLM\SOFTWARE\Microsoft\Internet Explorer\Extensions*, and you know that each of the subkeys under there is going to contain some of the same values. So, if you want those lower-level values you're going to have to use the Registry Instance Provider.

Check out the example below and tell me if you would want to use the Registry Property Provider to query for each of those “{FBSF1910-F110-11d2-BB9E...}” subkeys under Extensions. Oh, and good luck guessing what those crazy keys even are, not to mention not making a typo for any of them.

Figure 7.1



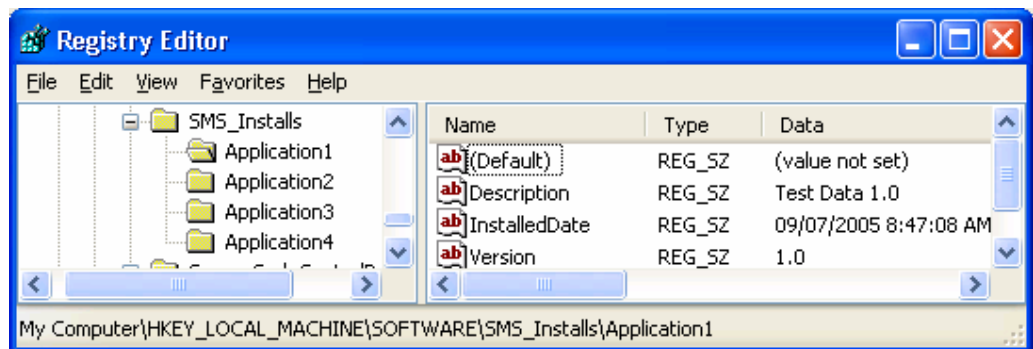
Wouldn't it be easier to just say, "Hey Registry Instance Provider guy, go check out all the subkeys under *HKLM\SOFTWARE\Microsoft\Internet Explorer\Extensions*, and look for any subkeys containing a MenuText value under there. Thanks, I'm going to go get a cup of coffee and brag to my security people now."

As another example, suppose you as the SMS administrator, decided to add a handy registry entry to help you keep track of software that you have installed on systems. You have created your own *HKLM\SOFTWARE\SMS_Installs* registry key and for each application installation you have added values for storing data on the Description, InstalledDate, and Version.

All you have to do is identify the main or parent registry key all the goodies are in to the RIP, and like a nosy neighbor, it will report back every detail about your kitchen, or every SMS installed application it can find there.

Sure sounds easy you say ... must be complicated to do. Nope. Sticking with the above example about SMS installed applications, if you had installed four applications in which you had added the applicable registry keys for each installation, your custom registry additions would appear like Figure 7.2 in the client registry:

Figure 7.2



The main confusion some have about the RIP is how it knows where to look under the main key for the properties you're after. By qualifying as *Key* the sub key containing the values we're after, the RIP knows to look.

So all we have to do is instruct the RIP to go find the parent key that all the SMS installed application information has been stored in, or:

```
HKLM\SOFTWARE\SMS_Installs.
```

By looking at the registry screenshot above, we know that the application name will be our key field since it's the core key for each installed application. The RIP will look under the `HKLM\SOFTWARE\SMS_Installs` registry key. If it finds any of the values you've specified as reporting fields (in the right pane of regedit) it will decide that the parent key (in the left pane of regedit) to the field value keys must fall under the category of `ApplicationName`.

How does the RIP know to look for a key named ApplicationName?

The RIP doesn't actually look for a specific key name, it just looks for the key under the parent key you've specified and trusts that you know what you're doing when you identify that key as `ApplicationName`.

The MOF edit itself is fairly straightforward as well. Just identify the registry provider you're going to use, the RIP, and the parent key you're after:

```
#pragma namespace("\\\\.\\root\\CIMV2")
[ dynamic, provider("RegProv"),
  ClassContext("local|HKEY_LOCAL_MACHINE\SOFTWARE\SMS_Installs")]
```



Just to reinforce how the RPP and RIP are distinct, observe how the `ClassContext` line shows the single, parent key from which the RIP will pull information. Yet it will recurse through the entire `SMS_Installs` registry key looking for those named values.

Next, declare the data class and the values you want as `PropertyContext` lines.

```
class SMS_Installs
{
  [Key] string ApplicationName;
  [PropertyContext("Description")] string Description;
  [PropertyContext("InstalledDate")] string InstalledDate;
  [PropertyContext("Version")] string Version;
};
```

Finally, add in the reporting class.

```
#pragma namespace("\\\\.\\root\\CIMV2\\SMS")

[SMS_Report(TRUE), SMS_Group_Name("SMS_Installs"),
SMS_Class_ID("SMSExpert|SMS_Installs|1.0")]
class SMS_Installs : SMS_Class_Template
{
    [SMS_Report(TRUE),Key] string ApplicationName;
    [SMS_Report(TRUE)] string Description;
    [SMS_Report(TRUE)] string InstalledDate;
    [SMS_Report(TRUE)] string Version;
};
```

After throwing in some comments and putting it all together, your RIP MOF edit should look something like this:

```
//-----
// Start of SMS_Installs
//-----
#pragma namespace("\\\\.\\root\\CIMV2")

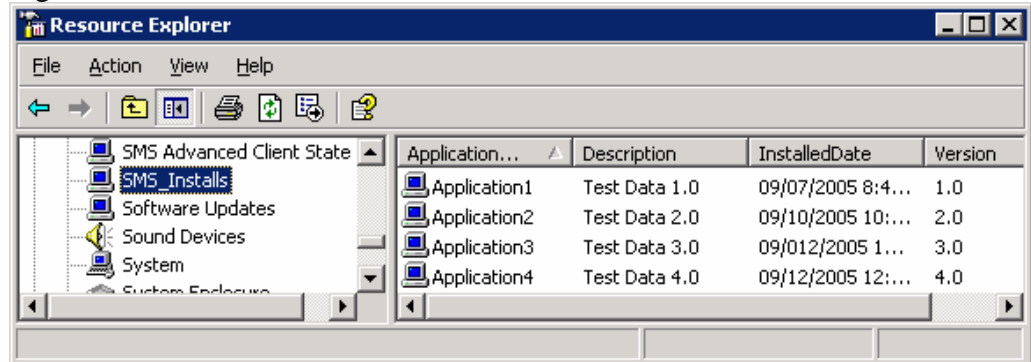
[ dynamic, provider("RegProv"),
ClassContext("local|HKEY_LOCAL_MACHINE\\SOFTWARE\\SMS_Installs"
)
]

class SMS_Installs
{
    [Key] string ApplicationName;
    [PropertyContext("Description")] string Description;
    [PropertyContext("InstalledDate")] string InstalledDate;
    [PropertyContext("Version")] string Version;
};

#pragma namespace("\\\\.\\root\\CIMV2\\SMS")
[SMS_Report(TRUE), SMS_Group_Name("SMS_Installs"),
SMS_Class_ID("SMSExpert|SMS_Installs|1.0")]
class SMS_Installs : SMS_Class_Template
{
    [SMS_Report(TRUE),Key] string ApplicationName;
    [SMS_Report(TRUE)] string Description;
    [SMS_Report(TRUE)] string InstalledDate;
    [SMS_Report(TRUE)] string Version;
};
//-----
// End of SMS_Installs
//-----
```

Once you have compiled the MOF edit locally on a test system and received the new inventory policy from the MP (or the new SMS_def.mof from the CAP if you're using SMS 2.0) and run a hardware inventory, you should see something similar to Figure 7.3 in resource explorer for the system after a few minutes:

Figure 7.3



Another good example of the RIP in action can be found in the default SMS 2003 SMS_def.mof in the form of the Add/Remove programs section:

```
//-----
// Add Remove Programs
//-----

#pragma namespace ("\\\\.\\root\\CIMV2")

[ dynamic,
  provider("RegProv"),

  ClassContext("local|HKEY_LOCAL_MACHINE\\Software\\Microsoft\\Win
dows\\CurrentVersion\\Uninstall")
]
class Win32Reg_AddRemovePrograms
{
  [key]
  string  ProdID;
  [PropertyContext("DisplayName")]
  string  DisplayName;
  [PropertyContext("InstallDate")]
  string  InstallDate;
  [PropertyContext("Publisher") ]
  string  Publisher;
  [PropertyContext("DisplayVersion")]
  string  Version;
};

#pragma namespace ("\\\\.\\root\\CIMV2\\SMS")
```

```
[ SMS_Report (TRUE),
  SMS_Group_Name ("Add Remove Programs"),
  SMS_Class_ID ("MICROSOFT|ADD_REMOVE_PROGRAMS|1.0") ]

class Win32Reg_AddRemovePrograms : SMS_Class_Template
{
  [SMS_Report (TRUE), key ]
  string ProdID;
  [SMS_Report (TRUE) ]
  string DisplayName;
  [SMS_Report (TRUE) ]
  string InstallDate;
  [SMS_Report (TRUE) ]
  string Publisher;
  [SMS_Report (TRUE) ]
  string Version;
};
```

Again, the **Registry Property Provider** excels at grabbing a variety of keys from a variety of known locations. The **Registry Instance Provider** excels at grabbing a variety of unknown keys from a single location.

These examples assume that the Registry Instance Provider has been registered elsewhere in your SMS_def.mof. If you are running SMS 2003 this is done for you by default. For SMS 2.0 you need to ensure that the RIP is registered either above this MOF edit or as part of it.

Remember, these providers only need to be registered in your SMS_def.mof once! To register the Registry Instance Provider for SMS 2.0 cut and paste the below lines into your SMS_def.mof :



```
#pragma namespace("\\\\.\\root\\CIMV2")
instance of __Win32Provider as $Instprov
{
  Name      = "RegProv" ;
  CLSID     = "{fe9af5c0-d3b6-11ce-a5b6-00aa00680c3f}" ;
};
instance of __InstanceProviderRegistration
{
  Provider      =$InstProv;
  SupportsPut   =TRUE;
  SupportsGet   =TRUE;
  SupportsDelete =FALSE;
  SupportsEnumeration =TRUE;
};
```

Notice that the provider is registered in the *root\CIMV2* namespace and that the name of the provider is *RegProv*. This is how any class that uses the provider will refer to it. It is best not to change this name, as many examples you will find on the Internet and in SMS communities will be using the same name for the providers.

If your name is Ed and someone yells “Chris” in a crowd, you’re not going to turn your head. Neither will the Registry Instance Provider. If you name your provider Ed and copy someone else’s class into your MOF that calls that same provider, Chris, the Class will not function correctly.



Chapter Summary

The Registry Instance Provider (RIP) doesn't need to know the exact names of the keys it is querying. Just give it a "main key" to search under and it will find any subkey containing the values you named as reporting fields. The parent key of the values found will be assumed to be your key field, and the resulting data will appear associated with that key.

The RIP does not explicitly define an ending registry key in the MOF edit like the Registry Property Provider does.

The RIP is already registered in the default SMS_def.mof for SMS 2003 as RegProv. For SMS 2.0 sites you need to add in the registration information yourself before using it.

The best use for the RIP is for inventorying various keys that all have similar data values that you want to inventory without creating more than one reporting class.

Chapter 8: Using the View Provider

Using MOF Extensions with Namespaces Other Than root\CIMV2 for fun and profit.

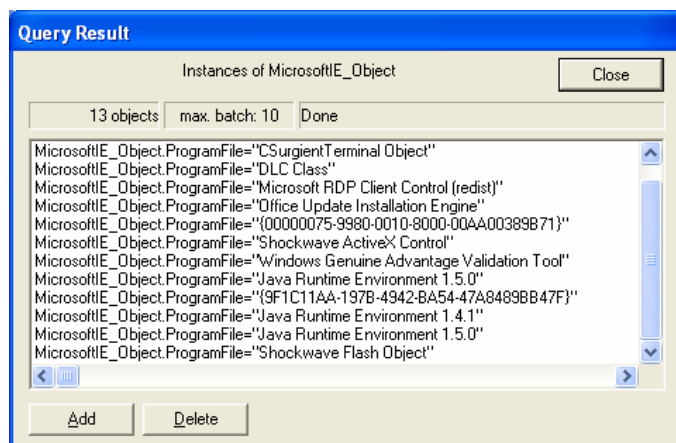
Way back in Chapter 3 I briefly touched on the View Provider and it's usefulness for SMS 2.0 clients to access namespaces other than *root\CIMV2* and *root\CIMV2\SMS*. This provider is not quite as easy to use as the others, but can be extremely useful when extending your inventory.

The View Provider can create instances of new classes based on instances of existing classes. In other words, you can trick the SMS inventory into believing that instances found in the data class you've created are there when in reality, they are somewhere else. This can come in handy in a few situations that I'll go over next.

Accessing Namespaces Other Than *root\CIMV2*

Using WBEMTEST, you can see the instances stored in the *root\CIMV2\applications\microsoftIE* namespace on my test system:

Figure 8.1



Well, suppose you wanted to inventory this information and you were running an SMS 2.0 site. How would you get the data, since the hardware inventory agent for SMS 2.0 doesn't see namespaces other than the *root\CIMV2* and *root\CIMV2\SMS*, and this data is in the *root\CIMV2\applications\MicrosoftIE* namespace?

The View Provider is the answer, of course.



This example is mainly for SMS 2.0 and educational purposes only. If you are using SMS 2003 there are easier ways to do this which I'll go over later. Even with SMS 2003 the View Provider can be useful so don't sleep through this example!

Checking out the class with WBEMTEST yields some information we need for our MOF edit, namely, what are the properties of the class that we can query for and should include in our edit. Double clicking on the properties and checking qualifiers tells us that the *ProgramFile* property is the key property for this class so let's make sure to make note of that.

For SMS 2.0 you need to register the View Provider just like the registry providers:

```
#pragma namespace ("\\\\.\\root\\CIMV2")

instance of __Win32Provider as $ViewProv
{
    Name = "MS_VIEW_INSTANCE_PROVIDER";
    ClsId = "{AA70DDF4-E11C-11D1-ABB0-00C04FD9159E}";
    ImpersonationLevel = 1;
    PerUserInitialization = "True";
};

instance of __InstanceProviderRegistration
{
    Provider = $ViewProv;
    SupportsPut = True;
    SupportsGet = True;
    SupportsDelete = True;
    SupportsEnumeration = True;
    QuerySupportLevels = {"WQL:UnarySelect"};
};

instance of __MethodProviderRegistration
{
    Provider = $ViewProv;
};
```



So after using WBEMTEST and making copious notes, we've decided to go after the *Caption*, *CodeBase*, *Description*, *ProgramFile(key)*, *SettingID*, and *Status* properties of the *MicrosoftIE_Objects* class found in the *root\CIMV2\applications\MicrosoftIE* namespace.

The resulting MOF edit using the View Provider would look like this:

```
#pragma namespace("\\\\.\\root\\CIMV2")

[Union,
ViewSources{"select * from MicrosoftIE_Object"},
ViewSpaces{"\\.\\root\\CIMV2\\applications\\MicrosoftIE"},
dynamic, provider("MS_VIEW_INSTANCE_PROVIDER")]
class MicrosoftIE_Objects

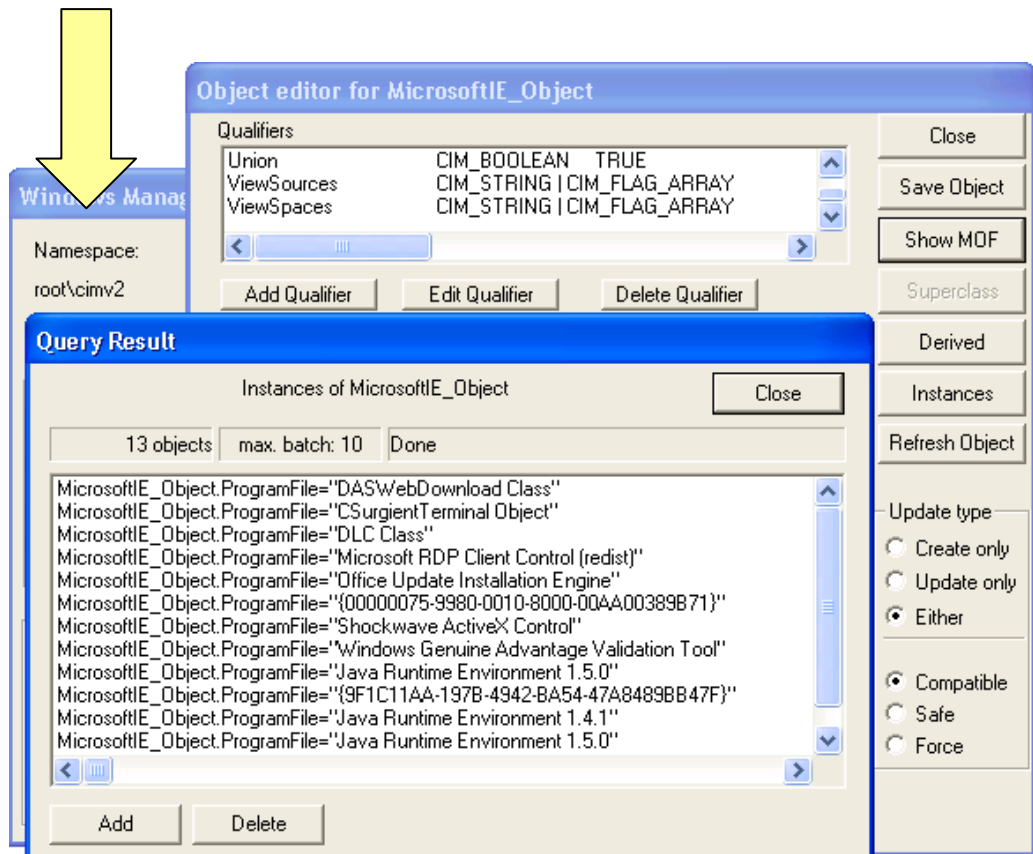
{
[PropertySources{"Caption"}] string Caption;
[PropertySources{"CodeBase"}] string CodeBase;
[PropertySources{"Description"}] string Description;
[PropertySources{"ProgramFile"},Key] string ProgramFile;
[PropertySources{"SettingID"}] string SettingID;
[PropertySources{"Status"}] string Status;
};

//reporting class
#pragma namespace("\\\\.\\root\\CIMV2\\SMS")
[SMS_Report(TRUE), SMS_Group_Name("MSFT_IEObjects"),
SMS_Class_ID("SMSExpert|MSFT_IEObjects|1.0") ]
class MicrosoftIE_Objects : SMS_Class_Template
{
[SMS_Report(TRUE)] string Caption;
[SMS_Report(TRUE)] string CodeBase;
[SMS_Report(TRUE)] string Description;
[SMS_Report(TRUE),Key] string ProgramFile;
[SMS_Report(TRUE)] string SettingID;
[SMS_Report(TRUE)] string Status;
};
```

If you take the above and compile it on a test system, like I did, and then use WBEMTEST to query the *root\CIMV2* namespace, you will stumble upon the *MicrosoftIE_Object* class that has mysteriously appeared, and containing all the information thought to only previously reside in the *root\CIMV2\applications\MicrosoftIE* namespace!

The resulting new class can now be inventoried, as usual, using the reporting class just like any other inventoried *root\CIMV2\SMS* class.

Figure 8.2



So ... got it? Ready to move on? I wouldn't do that to you. Let's go through something probably a little closer to your heart if you're an SMS 2.0 administrator wanting to know about Internet Explorer. Let's go through getting Internet Explorer version information using the View Provider.

You should know that Internet Explorer information is stored in the *root\CIMV2\applications\MicrosoftIE* namespace. The actual version information can be found in the MicrosoftIE Summary class. You also now have the general idea of how to get the class information and key fields so I'm just going to go through the motions quickly and then break down the actual MOF edit.

The major properties from the MicrosoftIE_Summary class we're going to go after for this example are: *Build*, *IEAKInstall*, *CipherStrength*, *Version*, and the key field *Name*.

To pull Internet Explorer version information using the View Provider your MOF edit would look like so:

```
#pragma namespace("\\\\.\\root\\CIMV2")

[Union,
ViewSources{"select * from MicrosoftIE_Summary"},
ViewSpaces{"\\.\\.\\root\\CIMV2\\applications\\MicrosoftIE"}, Dynamic
:
ToInstance, provider("MS_VIEW_INSTANCE_PROVIDER")]
class MicrosoftIE_Summary

{
[PropertySources{"Build"}] string Build;
[PropertySources{"IEAKInstall"}] string IEAKInstall;
[PropertySources{"CipherStrength"}] uint32 CipherStrength;
[PropertySources{"Version"}] string Version;
[PropertySources{"Name"},Key] string Name;
};

//reporting class
#pragma namespace("\\\\.\\root\\CIMV2\\SMS")
[SMS_Report(TRUE), SMS_Group_Name("MSFT_Summary"),
SMS_Class_ID("SMSExpert|MSFT_Summary|1.0") ]
class MicrosoftIE_Summary : SMS_Class_Template
{
[SMS_Report(TRUE)] string Build;
[SMS_Report(TRUE)] string IEAKInstall;
[SMS_Report(TRUE)] uint32 CipherStrength;
[SMS_Report(TRUE)] string Version;
[SMS_Report(TRUE),Key] string Name;
};
```



Make sure you check the property type for each of these! Putting string when the property is of type uint32 will give you about 10 minutes of confusion ... trust me, I just did it ☺

Another thing you may want to think about is querying only for the information you really need. Instead of:

```
ViewSources{"select * from MicrosoftIE_Summary"},
```

You could just as easily query for only the properties you need from the class:

```
ViewSources{"select Build,IEAKInstall,CipherStrength,Version,Name from MicrosoftIE_Summary"},
```

This is a basic use of WQL filtering for hardware inventory, and I'll talk about that some more in a minute.

For SMS 2003 Advanced Clients you don't always have to use a View Provider. All you need is a namespace qualifier in your reporting class to actually peer into the nonstandard namespace with your MOF edit as so:

```
// START - IE Objects
#pragma namespace("\\\\.\\root\\CIMV2\\SMS")

[ SMS_Report (TRUE), SMS_Group_Name ("Microsoft IE Object"),
SMS_Class_ID ("MICROSOFT|IE_Object|1.0"),
Namespace("\\\\.\\root\\CIMV2\\Applications\\MicrosoftIE")]

class MicrosoftIE_Object : SMS_Class_Template
{
    [SMS_Report (TRUE)] string    Caption;
    [SMS_Report (TRUE)] string    CodeBase;
    [SMS_Report (TRUE)] uint32    Description;
    [SMS_Report (TRUE),Key ] string    ProgramFile;
    [SMS_Report (FALSE)] string    SettingID;
    [SMS_Report (TRUE)] string    Status;
};
//END - IE Objects#pragma namespace ("\\.\\root\\CIMV2\\SMS")
```

To get the Internet Explorer version information using the namespace qualifier to peer into the *root\CIMV2\applications\MicrosoftIE* namespace just use the below edit:

```
// BEGIN - IE Summary
[ SMS_Report (TRUE), SMS_Group_Name ("Microsoft IE Summary"),
SMS_Class_ID ("SMSExpert|IE_SUMMARY|1.0"),
Namespace("\\\\.\\root\\CIMV2\\Applications\\MicrosoftIE")]

class MicrosoftIE_Summary : SMS_Class_Template
{
    [SMS_Report (TRUE)] string    Build;
    [SMS_Report (TRUE)] string    IEAKInstall;
    [SMS_Report (TRUE)] uint32    CipherStrength;
    [SMS_Report (TRUE)] string    Version;
    [SMS_Report (TRUE),Key] string    Name;
};
// END - IE Summary
```

SMS 2003 can use this namespace trick to get inside hardware vendor-specific WMI namespaces. As a matter of fact, SQL Server, Exchange, Office, IE and many other major software packages have their own namespaces that you can use this technique to get into. If you're using SMS 2.0 you can still do it by using a View Provider.

Below is an example of using the View Provider to get inside the Microsoft SQL Server namespace (*root\MicrosoftSQLServer*) taken from the Microsoft website:

http://www.microsoft.com/technet/prodtechnol/SMS/SMS2003/opsguide/ops_6ewj.msp

```
#pragma namespace("\\\\.\\Root\\CIMV2")
instance of __Win32Provider as $DataProv
{
    Name = "MS_VIEW_INSTANCE_PROVIDER";
    ClsId = "{AA70DDF4-E11C-11D1-ABB0-00C04FD9159E}";
    ImpersonationLevel = 1;
    PerUserInitialization = "True";
};
instance of __InstanceProviderRegistration
{
    Provider = $DataProv;
    SupportsPut = True;
    SupportsGet = True;
    SupportsDelete = True;
    SupportsEnumeration = True;
    QuerySupportLevels = {"WQL:UnarySelect"};
};
[union, ViewSources{"Select * from MSSQL_Database"},
ViewSpaces{"\\.\\.root\\MicrosoftSQLServer"}, Dynamic :
ToInstance, provider("MS_VIEW_INSTANCE_PROVIDER")]
class SQL_Databases
{
    [PropertySources("Size") ] sint32 Size;
    [PropertySources("SQLServerName"), key ] string SQLServerName;
    [PropertySources("Name"), key ] string Name;
    [PropertySources("SpaceAvailable") ] sint32 SpaceAvailable;
};
//Also, add the following MOF to SMS_def.mof :
#pragma namespace("\\\\.\\root\\CIMV2\\SMS")
[SMS_Report(TRUE),
SMS_Group_Name("SQL Database"),
SMS_Class_ID("MICROSOFT|SQLDatabase|1.0")]
class SQL_Databases : SMS_Class_Template
{
    [SMS_Report(TRUE),key]
    string SQLServerName;
    [SMS_Report(TRUE),key]
    string Name;
    [SMS_Report(TRUE)]
    sint32 Size;
    [SMS_Report(TRUE)]
    sint32 SpaceAvailable;
};
```

Hopefully you can now pick out the parts of that edit—the View Provider registration, the data class (including the WQL select statement) and the reporting class.

Using WQL Queries to Filter WMI Information

WMI filtering using WQL queries is a good idea in general. Why search through and mirror to *root\CIMV2* a complete namespace when all you need is one or two properties? A really good example of WMI filtering using the View Provider can be found on Eric Holtz's blog (slightly modified below):

<http://myitforum.techtarget.com/blog/eholtz/articles/3059.aspx>

Here, he is using a WQL filter to tell the inventory agent not to report back on domain accounts—which the *Win32_Account* class will do by default unless told otherwise. Imagine every SMS client you have returning every domain account during hardware inventory and you can see why this is not a good idea.

```
//----- Start LocalAccount.MOF
#pragma namespace("\\\\.\\root\CIMV2")
[Union,
ViewSources{"select * from Win32_Account where LocalAccount=True
and SIDType=1"},
ViewSpaces{"\\.\\root\CIMV2"},
dynamic, provider("MS_VIEW_INSTANCE_PROVIDER")]
class Win32_LocalUserAccount
{
[key, PropertySources{"Domain"}]
string Domain;
[key, PropertySources{"Name"}]
string Name;
};
#pragma namespace("\\\\.\\root\CIMV2\SMS")
[ SMS_Report(TRUE),
SMS_Group_Name("Local User Account"),
SMS_Class_ID("MICROSOFT|LOCAL_USER_ACCOUNT|1.0") ]

class Win32_LocalUserAccount: SMS_Class_Template
{
[key, SMS_Report(TRUE)]
string Domain;
[key, SMS_Report(TRUE)]
string Name;
};
//----- End SMS_def.mof addition
```



Chapter Summary

The View Provider can create instances of new classes based on instances of existing classes by mirroring data residing in different classes to the *root\CIMV2* namespace.

The View Provider is already registered in the default SMS_def.mof for SMS 2003 as MS_VIEW_INSTANCE_PROVIDER. For SMS 2.0 sites you need to add in the registration information yourself before using it.

The View Provider is used mainly for SMS 2.0 to inventory namespaces other than *root\CIMV2* and *root\CIMV2\SMS*. SMS 2003 sites can use the namespace qualifier to do this much easier.

The View Provider can be used to perform WQL filtering similar to SQL statements.

Chapter 9: Static Hardware Inventory Extensions

To do great work, a man must be very idle as well as very industrious.

—Samuel Butler

Extending hardware inventory for static data is fairly straightforward and depending on your personal level of scripting experience, relatively easy—at least in theory. Static MOF extensions are simply static data in MOF format that is added to the SMS hardware inventory. There are many ways to do this and I'm going to talk about three of them here:

- Static MOF files
- NOIDMIF files
- IDMIF files

Static MOF Files

Static MOF files are simply 'mini-MOF' files compiled on a machine to store system-specific information. These files are an easy way to maintain static data, but you won't be able to retrieve dynamic data, or data that changes on a regular basis with static MOF files.

To utilize the special powers of the static MOF you must edit the 'mini-MOF' to reflect the information you want and then compile it on the local system to store the data in the system's WMI repository. If the data changes, you'll have to go in and manually edit the file and re-compile the MOF to overwrite the old information stored in WMI.

Static MOF files are really easy to create. Their format is exactly the same as a 'normal' MOF edit except that they contain instance data. All you really need to remember is that you must create a data class and then tell SMS inventory what data, or instances, go with the data class for that particular machine.

In the below example from the Microsoft website, you can see that the data class called `Static_MOF` is created (you're allowed to be more imaginative, of course) and then instances of the `Static_MOF` class are created. This would be your 'mini-MOF':

```
#pragma namespace ("\\.\root\CIMV2")
class Static_MOF
{
    [key]
    string user;
    string office;
    string phone_number;
};
instance of Static_MOF
{
    user = "John Smith";
    office = "Building 4, Room 26";
    phone_number = "(425) 707-9791";
};
instance of Static_MOF
{
    user = "Denise Smith";
    office = "Building 4, Room 26";
    phone_number = "(425) 707-9790";
};
```

This static MOF would need to be compiled locally using `MOFCOMP`. Once you've done this, you just treat it like any other MOF edit and create the reporting class in your `SMS_def.mof` stored on the site server:

```
#pragma namespace ("\\.\root\CIMV2\SMS")
[ SMS_Report (TRUE),
  SMS_Group_Name ("Static AssetInfo MOF"),
  SMS_Class_ID ("MICROSOFT|Static_MOF|1.0")]
class Static_MOF : SMS_Class_Template
{
    [SMS_Report(TRUE), key]
    string user;
    [SMS_Report(TRUE)]
    string office;
    [SMS_Report(TRUE)]
    string phone_number;
};
```



Because you must manually compile the static MOF using MOFCOMP every time something changes, this is not the best idea for information that changes a lot.

Sure, this is kind of a pain, but I'll bet there are a few systems out there that you would like to have this capability of manually adding in static inventory information, right? How about serial numbers for specific software or license numbers? Use your imagination and I'm sure you can come up with some scenario that will work in your situation.

NOIDMIF and IDMIF Files in General

Management Information Format (MIF) files are used to collect inventory information by SMS. By creating custom MIF files you can extend hardware inventory in ways similar to the static MOF example. These files are indeed static, but interact with SMS in different ways.

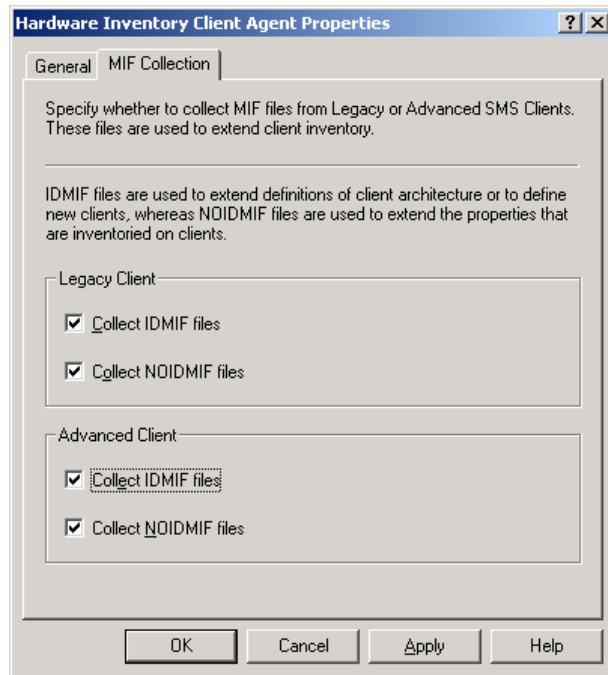
The main difference between NOIDMIF files and IDMIF files is that NOIDMIF files are information relating to an existing client system—they have an ID so none is needed, thus the name **NOIDMIF**.

IDMIF files contain data for a non-client system or for an object not already in the database. These files can pertain to stand-alone systems for which you want some kind of inventory data, or other things, like printers, people, etc. Because the data is not associated with a known entity, the IDMIF must contain a unique identifier to tell SMS to exactly what this information is referring. This is what puts the ID in **IDMIF**.

To enable MIF file collection, open the administrator console and browse to Systems Management Server\Site Database\Site Hierarchy\[Site Code]\Client Agents. Right click on **Hardware Inventory Client Agent** and then properties, and finally, click on the **MIF Collection** tab.

On the MIF Collection tab you will see the various check boxes controlling MIF collection properties for both Legacy Clients and Advanced Clients. Simply check the box that corresponds to your MIF collecting wishes and you're site will be all set to collect these little critters from your clients at their next hardware inventory cycle.

Figure 9.1

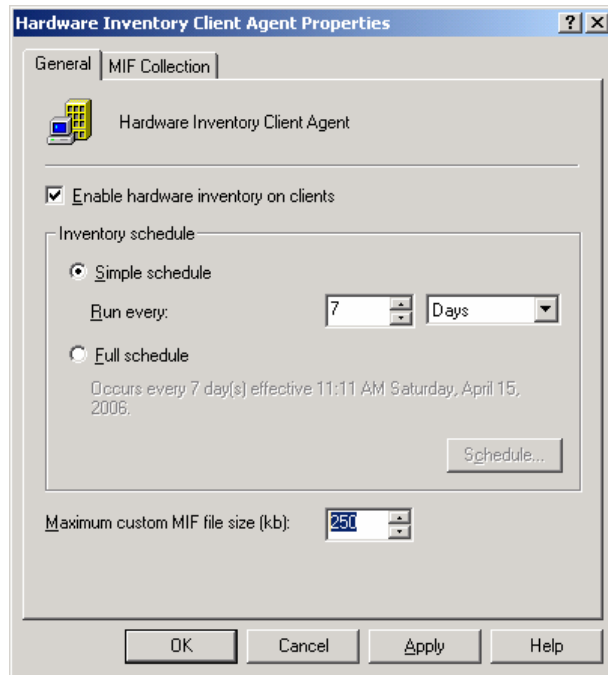


Just so you know at this point—these MIF files aren't really "collected" as the action implies. They are simply read by the inventory agents and their data is added to the system's hardware inventory sent to the site server.

Going back a tab to the General tab you will notice some more MIF information at the bottom. Regardless of whether the MIF file is a NOIDMIF or IDMIF, the maximum custom MIF file size processed by SMS is 250kb by default. This can be adjusted if you feel your inventory MIF files will be larger than this by adjusting the MIF size value in the hardware inventory client agent properties. This is an important factor to note. If your MIF file is too large or syntactically incorrect, it will be moved to a BADMIFS folder and ignored by hardware inventory.

The maximum custom MIF size can be set in kb's by changing the bottom text box on the General tab as in Figure 9.2.

Figure 9.2



These inventory extensions *do not* require modifying the SMS_def.mof. Everyone see that? I'm using this section to acquaint you with other methods for collecting inventory information, just so you will know about some other options. Let's face it, sometimes making an actual SMS_def.mof modification isn't the best way to go.

NOIDMIF Files

NOIDMIF files are stored locally on client systems. They do not need to be compiled by MOFCOMP, and do not interact with the local system WMI at all. The NOIDMIF file is read by hardware inventory and processed alongside the "normal" hardware inventory data. The information from the NOIDMIF is then added to the hardware inventory information of the client system where it was found.

SMS won't go playing hide and seek with your systems to find these MIF files; they must be stored in a particular folder created just for them. NOIDMIF files must be stored in a special NOIDMIF directory:

- Advanced Clients: %Windir%\System32\CCM\Inventory\Noidmifs
- Legacy Clients: %Windir%\MS\SMS\Noidmifs

So what happens if I delete the NOIDMIF file from the clients?

If the NOIDMIF file is deleted, when the next hardware inventory is run, the inventory data is deleted from the database data table but remains in the database history table for the class previously created by the NOIDMIF file.



If you do not want your NOIDMIF to create a history table for this reason, you can give your NOIDMIF an .NHM extension instead of .MIF. An .NHM or No History MIF file will only populate the data table for your class with information.

So that should suffice for background here, considering I'm going a little off-topic. I'll just run you through creating and using a NOIDMIF file real quickly ... this ought to be about as quick as boiling an egg over an open fire in Alaska while ice fishing, but here we go.

The first thing you need to know about using a NOIDMIF file is the format of the file itself. I'll try to make this easy so I don't confuse you here and list the key parts and what they're used for—these are my own definitions, in case you are wondering if Webster took the day off:



Component: There can be only one component per NOIDMIF. Start your NOIDMIF with *Start Component* and end the NOIDMIF with *End Component*.

Group: The name this information will appear under in a Resource explorer view of a client system. This will also become the table name in the SMS database except that the spaces will be replaced with underscores.

Attribute: These are the data properties that you are inventorying and are associated with the group name.

There is an excellent example of using a NOIDMIF file in the SMS 2003 Operations Guide. Rather than try to improve upon the hard work of the Microsoft technical writers, I'll borrow from their example and try to expound upon the key areas I think are important.

The example is called “Creating a Class by Using a NOIDMIF File”. I'll summarize the scenario for you by just detailing the high points and objective of the example as best I can. Basically, what is going on here is that you, as an SMS administrator for Wide World Importers, need to be able to store computer asset numbers in your database so that they are available for queries and asset management.

To do this, you will use a NOIDMIF file. Here is the resulting NOIDMIF file that is created to accomplish this task:

```

Start Component
  Name = "System Information"    Start Group
  Name = "Wide World Asset Numbers"    ID = 1
  Class = "WideWorldAssetNumbers"    Key = 1
  Start Attribute
    Name = "Computer Asset Number"    ID = 1
    Type = String(10)
    Value = "414207"    End Attribute
  End Group
End Component

```

Here's how to create that NOIDMIF file using a text editor, as taken from the SMS 2003 Operations Guide:

1. Type the following line to begin the NOIDMIF file:

```
Start Component
```

2. You must always add a component and name the component when you create a NOIDMIF file.

3. Type the following line to name the component:

```
Name = "System Information"
```

4. By using a general name such as **System Information**, this component becomes more flexible. You can then use it to add any information you want to maintain for this client by adding new groups to the existing NOIDMIF file.

5. Type the following line to add the Display Name for the new **Wide World Importers Asset Numbers** class:

```
Start Group
Name = "Wide World Importers Asset Numbers"
```

6. The **Name** property is the string that administrators see in Resource explorer to refer to this class. Wide World Importers Asset Numbers is a DMTF group class. When SMS first loads this group, it creates a WMI class called SMS_G_wide_world_asset_numbers.

7. After you add properties, even if you add only a single property, you need to add a group to contain your new properties.

8. Type the following line to give the **Wide World Importers Asset Numbers** class a group ID number:

```
ID = 1
```

9. Use any method to determine the unique ID number for each group and property, if the ID number is unique for groups within this component.

10. Type the following line to add the **wideWorldImportersAssetNumbers** class:

```
Class = "wideWorldImportersAssetNumbers"
```

11. The class information is used for processing and is never seen by administrators.

12. Type in the following line to add the key property:

```
Key = 1
```

13. This entry indicates that the first property listed is the key. Key properties are unique properties that identify instances of a certain class. Whenever you have more than one instance of a class, you must include at least one key property, or the subsequent instances of the class will overwrite the previous instances. If no key properties are defined for a NOIDMIF file on a client running a 32-bit operating system, all the properties are designated as key by the inventory process. This does not occur for IDMIF files or for NOIDMIF files on clients running 16-bit operating systems.

14. Type the following lines to add the first property:

```
Start Attribute
Name = "Computer Asset Number"ID = 1
Type = String(10)
Value = "414207"End Attribute
```

15. You must set an ID number for this property, name the property, and then specify a data type. The ID number you choose must be unique within the group. Only three data types are recognized by the system: integer, string, and specially formatted **DateTime** string. You must also specify a valid value for the data type you selected.

There you go, everything you've ever wanted to know about the NOIDMIF. I could go on and on about the NOIDMIF, but this book only has room for one ASCII type file star and that's the SMS_def.mof. Just know that if you ever want to get information into your SMS database for your client systems without modifying the SMS_def.mof this one of the ways to do it.

IDMIF Files

Remember that you use IDMIF files to add non-client systems, or other items such as printers to the SMS site database. Without getting too technical, let me say that IDMIF files are identical to NOIDMIF files except that you have to provide an architecture name and unique ID in a header at the top of the file.

This IDMIF header is really just a couple of comments. These are important comments, though. They must be formatted as so (including the <>'s):

```
//Architecture<ArchitectureName>
//UniqueID<999>
```

Of course, your architecture name and unique ID number would be different, but you get the idea.



There can be many instances of a class in your database—the unique ID is basically the key for this instance. Without a key, each successive instance will overwrite the previous instance(s) in the database.

You must have a top-level group with the same class as the architecture name, and that group must have at least one property. Remember that each instance of a class must have one key property to differentiate the instances of the class. That wasn't confusing, was it? Read this over a couple of times and it will sound just like the key properties of data classes in our regular MOF editing experiences together.

IDMIF files must be stored in a special IDMIF directory:

- Advanced Clients: *%Windir%\System32\CCM\Inventory\idmifs*
- Legacy Clients: *%Windir%\MS\SMS\idmifs*

The following is an example of a simple IDMIF file from the SMS 2003 Operations Guide:

```
//Architecture<Widget>
//UniqueId<414207>
Start Component
  Name = "System Information"  Start Group
  Name = "Widget Group"      ID = 1
  Class = "Widget"          Key = 1
  Start Attribute
    Name = "Widget Asset Number"  ID = 1
    Type = String(10)
    Value = "414207"  End Attribute
  End Group
End Component
```




Chapter Summary

Static hardware inventory extension methods are only as current as their creation date/time. To retrieve current data, they must be recreated, and also re-compiled, on a regular basis unless the data does not change.

Management Information Format (MIF) files are used to collect inventory information by SMS in a different way than MOF files. **MIF files do not require compilation or reporting classes added to the SMS_def.mof.** They are read by hardware inventory and their data is added to the inventory report.

NOIDMIF files are SMS client machine specific and are stored in the following locations:

- Advanced Clients: `%Windir%\System32\CCM\Inventory\Noidmifs`
- Legacy Clients: `%Windir%\MS\SMS\Noidmifs`

IDMIF files are not associated with a specific SMS client machine and you have to provide an architecture name and unique ID in a header at the top of the file.

IDMIF files must be stored in a special IDMIF directory:

- Advanced Clients: `%Windir%\System32\CCM\Inventory\idmifs`
- Legacy Clients: `%Windir%\MS\SMS\idmifs`


Maximum MIF file size by default in SMS is 250KB.



Component: There can be only one component per NOIDMIF. Start your NOIDMIF with *Start Component* and end the NOIDMIF with *End Component*.

Group: The name this information will appear under in a resource explorer view of a client system. This will also become the table name in the SMS database except that the spaces will be replaced with underscores.

Attribute: These are the data properties that you are inventorying, and are associated with the group name.



Chapter
10

Chapter 10: Scripted Hardware Inventory Extensions

It's always been and always will be the same in the world: The horse does the work, and the coachman is tipped. —Anonymous.

Scripted extensions can be used to add information to hardware inventory by writing MOF, NOIDMIF or IDMIF files. Scripts can even be used to write information directly into a system's WMI. The easiest way to do this is by using a scripting language to write static files. I'll go through that first, and then give examples of using scripts to actually read system information and write it to WMI for more advanced inventory capabilities.

Static File Scripted Extensions

For simple, static MOF, NOIDMIF, and IDMIF files you can use any scripting language you prefer that is capable of writing a text file. Learning how to do this is beyond the scope of this book, but I'll give you some examples of scripts here—after that, you are on your own!

The general idea for writing these static files from scripts is to just format the static data to be syntactically correct for the type of file you're writing—MOF, NOIDMIF, or IDMIF. Your script writes the file, as well as the data to be reported. If you can write a .BAT or .VBS script to create a text file, a little playing around with the static inventory examples throughout this book should get you going in no time.

I don't think I need to go into how to write a static MOF file; just use a script to write a file that looks like a 'mini-MOF' and compile it. Creating NOIDMIF files, however, is always challenging because of the format and required field structure involved. Finding good examples of them is also pretty hard for most people.

So, just as a quick, and hopefully fun, example I'm going to show you how to use a script to add flip flop slang name information to your SMS database by writing a NOIDMIF file.

Of course, this will probably never be a requirement for you, but I think using a silly example will help you to understand the concept a little easier than going through the Win32_PhysicalMemory class again, right? Someone tell me I'm right; my wife thinks I've gone crazy at this point.

This example .VBS script will create a NOIDMIF file to get flip flop slang names from a few different countries into your SMS database. Just copy and paste the below script into notepad, name it flipflop.VBS and run it. The NOIDMIF, which is named NOIDMIF_Format.MIF, will be created in the same folder the script is run from:

```
Set fso = CreateObject("Scripting.FileSystemObject")
Set f1 = fso.createTextFile("NOIDMIF_Format.MIF",2,False)
f1.WriteLine "Start Component"
f1.WriteLine " Name = " & Chr(34) & "NOIDMIF Example" & Chr(34)
f1.WriteLine " Start Group"
f1.WriteLine "  Name = " & Chr(34) & "Flip Flop Information" & Chr(34)
f1.WriteLine "  ID = 1"
f1.WriteLine "  Class = " & Chr(34)&"SMSExpert|Flip Flop Information|1.0"&Chr(34)
f1.WriteLine "  Start Attribute"
f1.WriteLine "    Name = " & Chr(34) & "New Zealand Name" & Chr(34)
f1.WriteLine "    ID = 1"
f1.WriteLine "    ACCESS = READ-ONLY"
f1.WriteLine "    Storage = Specific"
f1.WriteLine "    Type = String"
f1.WriteLine "    Value = " & Chr(34) & "jandals (Japanese Sandals)" & Chr(34)
f1.WriteLine "    End Attribute"
f1.WriteLine "  Start Attribute"
f1.WriteLine "    Name = " & Chr(34) & "Australian Name" & Chr(34)
f1.WriteLine "    ID = 2"
f1.WriteLine "    ACCESS = READ-ONLY"
f1.WriteLine "    Storage = Specific"
f1.WriteLine "    Type = String"
f1.WriteLine "    Value = " & Chr(34) & "thongs" & Chr(34)
f1.WriteLine "    End Attribute"
f1.WriteLine "  Start Attribute"
f1.WriteLine "    Name = " & Chr(34) & "South African Name" & Chr(34)
f1.WriteLine "    ID = 3"
f1.WriteLine "    ACCESS = READ-ONLY"
f1.WriteLine "    Storage = Specific"
f1.WriteLine "    Type = String"
f1.WriteLine "    Value = " & Chr(34) & "thongs or slops" & Chr(34)
f1.WriteLine "    End Attribute"
f1.WriteLine " End Group"
f1.WriteLine "End Component"
f1.close
```

The result of running the above script is a NOIDMIF that looks like this:

```

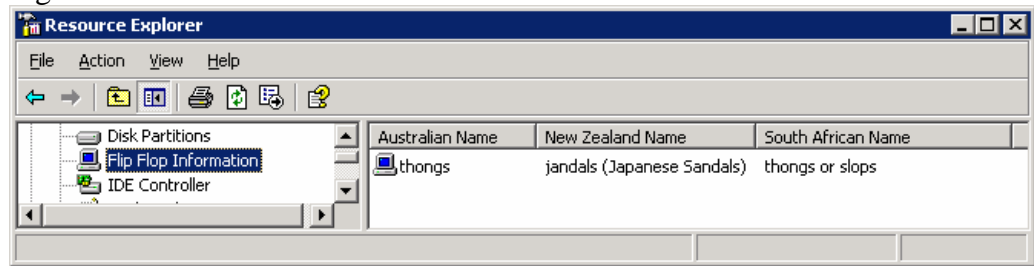
Start Component
  Name = "NOIDMIF Example"
  Start Group
    Name = "Flip Flop Information"
    ID = 1
    Class = "SMSEXPERT|Flip Flop Information|1.0"
    Start Attribute
      Name = "New Zealand Name"
      ID = 1
      ACCESS = READ-ONLY
      Storage = Specific
      Type = String
      Value = "jandals (Japanese Sandals)"
    End Attribute
    Start Attribute
      Name = "Australian Name"
      ID = 2
      ACCESS = READ-ONLY
      Storage = Specific
      Type = String
      Value = "thongs"
    End Attribute
    Start Attribute
      Name = "South African Name"
      ID = 3
      ACCESS = READ-ONLY
      Storage = Specific
      Type = String
      Value = "thongs or slops"
    End Attribute
  End Group
End Component

```

If you were to drop this flip flop NOIDMIF into your SMS client's noidmifs folder and run a hardware inventory, the flip flop NOIDMIF file will be read and processed by SMS as inventory information associated with that particular system.

You'll be rewarded with all the flip flop slang information you'll ever need in your SMS database!

Figure 10.1



Simply remove the flip flop NOIDMIF from your client's noidmifs folder (`%WINDIR%\System32\CCM\Inventory\Noidmifs` for Advanced Clients) and this information will disappear from the resource explorer view for the system and your SMS database's data tables, but not the history tables. *The data will disappear, that is, not the tables themselves.* You'll still be stuck with flip flop information data and history tables, as well as their related views, until you remove them from your database



Special note here in case you've been playing with my flip flop NOIDMIF example and are now trying to find some way to get rid of those silly tables. **DO NOT** just go into SQL and manually delete those tables! You can use the Microsoft DELGRP.EXE utility to remove those tables that were created when SMS processed the NOIDMIF. DELGRP.EXE and SMS Expert's Site Sweeper utility—which I prefer—will be discussed more in Chapter 13, Cleaning Up.

Scripting to pull data from a system is a little more complicated. Using a scripting language you can harness the power of the language you've chosen to query the system and then write custom inventory data to the local system's WMI repository.

Usually you will find most examples to do this in visual basic script (.VBS). The inventory scripts found at www.SMSExpert.com are .VBS scripts, for example. Visual basic scripts are probably the easiest way to tie into WMI as well, for either querying for information and creating .MIF files, or for writing straight to WMI itself.

An excellent example of using a .VBS script to write a NOIDMIF file from reading system information is shown below. This example comes from Michael Leonard and uses visual basic scripting to read DHCP & DNS/WINS information from SMS 2003 Advanced Clients. Michael is an enterprise admin and needed a way to extract this information from his client systems to ensure they were being configured properly. He created the .VBS script to extract the information and then ran an advertised program weekly to collect and update the data.

This appears to be a pretty long script when formatted for this book, but trust me; it's an excellent script, as well as an excellent example for this section. Thank you, Michael!

```

' VBS - Read DHCP & DNS/WINS config info and put to MIF file for SMS
' for W2K & better workstations/servers
' 04/27/2005 MA.Leonard, State of Nebraska, Dept of Information Technology

Dim WSHShell, wDir, FSO, ObjExec, flenm
Dim strFromProc, rtncd

Set WSHShell = CreateObject("WScript.Shell")
Set FSO = CreateObject("Scripting.FileSystemObject")

' NOIDMIF files must be stored in the following folder
' on Advanced Clients: %Windir%\System32\CCM\Inventory\Noidmifs
' on Legacy Clients: %Windir%\MS\SMS\Noidmifs
' make sure users have write/modify permission to the folder

wDir = WSHShell.ExpandEnvironmentStrings("%Windir%")
Const adclnt = "\\System32\CCM\Inventory\Noidmifs"
Const lgclnt = "\\MS\SMS\Noidmifs"

function CkDir()
' Check for SMS subdir/folder, if not there terminate
if FSO.FolderExists(wDir & adclnt) Then
Set flenm = FSO.CreateTextFile(wDir & adclnt & "\SMSipcfg.mif", True)
CkDir = True
else
If FSO.FolderExists(wDir & lgclnt) Then
Set flenm = FSO.CreateTextFile(wDir & lgclnt & "\SMSipcfg.mif", True)
CkDir = True
else
CkDir = False
end if
end if
end function

Sub procDhcp()
if currln = "" then
fndpos = 0
fndpos = InStr(1,strFromProc,"DHCP enabled",1)
if fndpos > 0 then
fndpos = 0
fndpos = InStr(1,strFromProc,":",1)
if fndpos > 0 then
tmpval = ""
tmpval = Trim(Mid(strFromProc,fndpos + 1))
wrkfld = " Start Attribute" & VBCrLf
wrkfld = wrkfld & " Name = ""DHCP Value"" & VBCrLf
wrkfld = wrkfld & " ID = 1" & VBCrLf
wrkfld = wrkfld & " Type = String(3)" & VBCrLf
wrkfld = wrkfld & " Value = " & """" & tmpval & """" & VBCrLf

```

```

    wrkfld = wrkfld & " End Attribute" & VBCrLf
    flnm.Write wrkfld
    currln = "dhcp"
    prevln = ""
  end if
end if
end if
end sub

Sub procIpAddr()
if currln = "" then
  fndpos = 0
  fndpos = InStr(1,strFromProc,"IP address",1)
  if fndpos > 0 then
    fndpos = 0
    fndpos = InStr(1,strFromProc,":",1)
    if fndpos > 0 then
      tmpval = ""
      tmpval = Trim(Mid(strFromProc,fndpos + 1))
      wrkfld = " Start Attribute" & VBCrLf
      wrkfld = wrkfld & " Name = ""IP Address"" & VBCrLf
      wrkfld = wrkfld & " ID = 2" & VBCrLf
      wrkfld = wrkfld & " Type = String(15)" & VBCrLf
      wrkfld = wrkfld & " Value = " & """" & tmpval & """" & VBCrLf
      wrkfld = wrkfld & " End Attribute" & VBCrLf
      flnm.Write wrkfld
      currln = "ipaddr"
      prevln = ""
    end if
  end if
end if
end sub

Sub procDns()
if currln = "" then
  fndpos = 0
  fndpos = InStr(1,strFromProc,"DNS",1)
  if fndpos > 0 then
    fndpos = 0
    fndpos = InStr(1,strFromProc,"Static",1)
    if fndpos > 0 then
      tmpval = "Yes"
    else
      tmpval = "No"
    end if
    wrkfld = " Start Attribute" & VBCrLf
    wrkfld = wrkfld & " Name = ""Static DNS"" & VBCrLf
    wrkfld = wrkfld & " ID = 3" & VBCrLf
    wrkfld = wrkfld & " Type = String(3)" & VBCrLf
  end if
end if
end sub

```

```

wrkfld = wrkfld & " Value = " & """" & tmpval & """" & VBCrLf
wrkfld = wrkfld & " End Attribute" & VBCrLf
flenm.Write wrkfld

fndpos = 0
fndpos = InStr(1,strFromProc,":",1)
if fndpos > 0 then
    tmpval = ""
    tmpval = Trim(Mid(strFromProc,fndpos + 1))
    wrkfld = " Start Attribute" & VBCrLf
    wrkfld = wrkfld & " Name = ""Primary DNS Value"" & VBCrLf
    wrkfld = wrkfld & " ID = 4" & VBCrLf
    wrkfld = wrkfld & " Type = String(15)" & VBCrLf
    wrkfld = wrkfld & " Value = " & """" & tmpval & """" & VBCrLf
    wrkfld = wrkfld & " End Attribute" & VBCrLf
    flenm.Write wrkfld
    currln = "dns"
    prevln = "dns"
end if
else
    if prevln = "dns" then
        tmpval = ""
        tmpval = Trim(Mid(strFromProc,1,15))
        if tmpval = "" then
            tmpval = Trim(strFromProc)
            wrkfld = " Start Attribute" & VBCrLf
            wrkfld = wrkfld & " Name = ""Secondary DNS Value"" & VBCrLf
            wrkfld = wrkfld & " ID = 5" & VBCrLf
            wrkfld = wrkfld & " Type = String(15)" & VBCrLf
            wrkfld = wrkfld & " Value = " & """" & tmpval & """" & VBCrLf
            wrkfld = wrkfld & " End Attribute" & VBCrLf
            flenm.Write wrkfld
            currln = "dns"
            prevln = "dns"
        end if
    end if
end if
end if
end sub

Sub procWins()
if currln = "" then
    fndpos = 0
    fndpos = InStr(1,strFromProc,"WINS",1)
    if fndpos > 0 then
        fndpos = 0
        fndpos = InStr(1,strFromProc,"Static",1)
        if fndpos > 0 then
            tmpval = "Yes"

```



```

else
    tmpval = "No"
end if
wrkfld = " Start Attribute" & VBCrLf
wrkfld = wrkfld & " Name = ""Static WINS"" & VBCrLf
wrkfld = wrkfld & " ID = 6" & VBCrLf
wrkfld = wrkfld & " Type = String(3)" & VBCrLf
wrkfld = wrkfld & " Value = " & """" & tmpval & """" & VBCrLf
wrkfld = wrkfld & " End Attribute" & VBCrLf
flenm.Write wrkfld

fndpos = 0
fndpos = InStr(1,strFromProc,":",1)
if fndpos > 0 then
    tmpval = ""
    tmpval = Trim(Mid(strFromProc,fndpos + 1))
    wrkfld = " Start Attribute" & VBCrLf
    wrkfld = wrkfld & " Name = ""Primary WINS Value"" & VBCrLf
    wrkfld = wrkfld & " ID = 7" & VBCrLf
    wrkfld = wrkfld & " Type = String(15)" & VBCrLf
    wrkfld = wrkfld & " Value = " & """" & tmpval & """" & VBCrLf
    wrkfld = wrkfld & " End Attribute" & VBCrLf
    flenm.Write wrkfld
    currln = "wins"
    prevln = "wins"
end if
else
    if prevln = "wins" then
        tmpval = ""
        tmpval = Trim(Mid(strFromProc,1,15))
        if tmpval = "" then
            tmpval = Trim(strFromProc)
            wrkfld = " Start Attribute" & VBCrLf
            wrkfld = wrkfld & " Name = ""Secondary WINS Value"" & VBCrLf
            wrkfld = wrkfld & " ID = 8" & VBCrLf
            wrkfld = wrkfld & " Type = String(15)" & VBCrLf
            wrkfld = wrkfld & " Value = " & """" & tmpval & """" & VBCrLf
            wrkfld = wrkfld & " End Attribute" & VBCrLf
            flenm.Write wrkfld
            currln = "wins"
            prevln = "wins"
        end if
    end if
end if
end if
end sub

Sub endgroup()

```

```

wrkfld = " End Group" & VBCrLf & VBCrLf
flenm.Write wrkfld
grpopen = 0
end sub
' *****
' This is the main proc
' use NetShell to get only active adapters
' and have report piped into I/O object for reading
,

rtncd = CkDir()
if rtncd then
Set ObjExec = WSHShell.Exec("Netsh interface ip show config")
grpopen = 0
adptnum = 0
Do
strFromProc = ObjExec.StdOut.ReadLine()
if strFromProc <> "" then
tmpval = ""
tmpval = LCase(Left(Trim(strFromProc),6))
if tmpval = "config" then
if grpopen = 1 then endgroup()
fndpos = 1
fndpos = InStr(1,strFromProc,"local",1)
if fndpos > 0 then
adptnum = adptnum + 1
grpopen = 1
if adptnum = 1 then
wrkfld = "Start Component" & VBCrLf
wrkfld = wrkfld & "Name = ""System Network Information"" & VBCrLf
flenm.Write wrkfld
end if
wrkfld = " Start Group" & VBCrLf
wrkfld = wrkfld & " Name = ""Network IP Config"" & VBCrLf
wrkfld = wrkfld & " ID = " & adptnum & VBCrLf
wrkfld = wrkfld & " Class = ""networkipconfig"" & VBCrLf
wrkfld = wrkfld & " Key = 1" & VBCrLf
flenm.Write wrkfld
adptsw = 1
prevln = ""
else
if grpopen = 1 then endgroup()
adptsw = 0
end if
else
if adptsw = 1 then
currln = ""
procDhcp()
procIpAddr()
procDns()

```

```

procWins()
else
' nop
end if
end if
end if
end if
Loop While Not ObjExec.Stdout.atEndOfStream
if grpopen = 1 then endgroup()
if adptnum > 0 then
wrkfld = "End Component" & VBCrLf
flenm.Write wrkfld
end if
flenm.Close
else
' WScript.Echo "No SMS client folder found"
end if

```



The above script will process for multiple adapters and for more than two DNS, two WINS, and one IP entry, but will not create the MIF file key fields properly. This was by design, and it causes a MIF processing error at inventory time. These errors were captured in some error reports that Michael had created to catch multiple active adapters and more than their normal two DNS and WINS entries.

Running the script will create a NOIDMIF file named SMSipcfg.mif, appearing similar to the one shown below for a system using DHCP without static DNS or WINS entries:

```

Start Component
Name = "System Network Information"
Start Group
Name = "Network IP Config"
ID = 1
Class = "networkipconfig"
Key = 1
Start Attribute
Name = "DHCP Value"
ID = 1
Type = String(3)
Value = "Yes"
End Attribute
Start Attribute
Name = "Static DNS"
ID = 3
Type = String(3)
Value = "No"
End Attribute
Start Attribute

```

```

Name = "Static WINS"
ID = 6
Type = String(3)
Value = "No"
End Attribute
End Group
End Component

```

Scripts That Write Directly to WMI

Creating scripts that write directly to WMI is considerably more complicated than writing a static MOF file by scripting. This isn't a book about scripting so I won't go into much more detail on it here. There is a good example of a script that writes directly to WMI in the [Systems Management Server 2003 Operations Guide](http://www.microsoft.com/technet/prodtechnol/sms/sms2003/opsguide/ops_9p7.p.mspx?mfr=true) (http://www.microsoft.com/technet/prodtechnol/sms/sms2003/opsguide/ops_9p7.p.mspx?mfr=true) that I'll share with you here. For more information about how to accomplish this scripting feat consult the SMS 2003 Operations Guide and the [SMS 2003 Software Development Kit](http://www.microsoft.com/downloads/details.aspx?FamilyID=58833cd1-6dbb-45bb-bb77-163446068ef6&DisplayLang=en) (<http://www.microsoft.com/downloads/details.aspx?FamilyID=58833cd1-6dbb-45bb-bb77-163446068ef6&DisplayLang=en>).

```

Set Loc = CreateObject("WbemScripting.SWbemLocator")
Set WbemServices = Loc.ConnectServer(, "root\CIMV2")
On Error Resume Next
Set WbemObject = WbemServices.Get("SMS_AssetWizard_1")
' If this call failed, we need to make the SMS_AssetWizard_1 data class
If Err Then
'Retrieve blank class
Set WbemObject = WbemServices.Get
'Set class name
WbemObject.Path_.Class = "SMS_AssetWizard_1" 'Add Properties (8 =
CIM_STRING, 11 = CIM_BOOLEAN)
WbemObject.Properties_.Add "Type", 19
WbemObject.Properties_.Add "ContactFullName", 8
WbemObject.Properties_.Add "ContactEmail", 8
WbemObject.Properties_.Add "ContactPhone", 8
WbemObject.Properties_.Add "ContactLocation", 8
WbemObject.Properties_.Add "SysLocationSite", 8
WbemObject.Properties_.Add "SysLocationBuilding", 8
WbemObject.Properties_.Add "SysLocationRoom", 8
WbemObject.Properties_.Add "SysUnitManufacturer", 8
WbemObject.Properties_.Add "SysUnitModel", 8
WbemObject.Properties_.Add "SysUnitAssetNumber", 8
WbemObject.Properties_.Add "SysUnitIsLaptop", 11
' Add key qualifier to Type property
WbemObject.Properties_("Type").Qualifiers_.Add "key", True
WbemObject.Put_
End if
On Error Goto 0
Set WbemServices = Loc.ConnectServer(, "root\CIMV2")
Set WbemObject = WbemServices.Get("SMS_AssetWizard_1").SpawnInstance_
' Store property values (the data!)
WbemObject.Type = 0
WbemObject.ContactFullName = "John Smith"
WbemObject.ContactEmail = "JSmith"
WbemObject.ContactPhone = "(425) 707-9791"
WbemObject.ContactLocation = "Redmond"
WbemObject.SysLocationSite = "Campus"
WbemObject.SysLocationBuilding = "24"

```

```
WbemObject.SysLocationRoom = "1168"  
WbemObject.SysUnitManufacturer = "Dell"  
WbemObject.SysUnitModel = "GX1"  
WbemObject.SysUnitAssetNumber = "357701"  
WbemObject.SysUnitIsLaptop = False  
'WMI will overwrite the existing instance'  
WbemObject.Put_
```



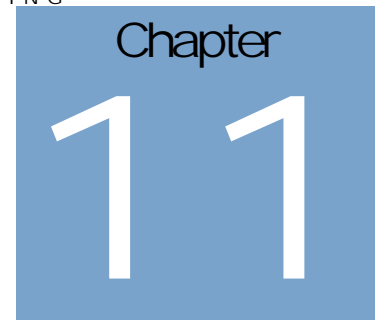
Chapter Summary

Scripted extensions can be used to add information to hardware inventory by writing MOF, NOIDMIF, IDMIIF files or directly to client WMI repositories.

Using a scripting language you can harness the power of the language you've chosen to query the system and then write custom inventory files or data directly to the local system's WMI repository.

Scripted hardware inventory extension methods basically create static inventory files or WMI additions. **These inventory additions are only as current as their creation date/time.** To retrieve current data, they must be re-run or their output files re-compiled on a regular basis unless the data does not change.

Scripted inventory extensions are primarily used when there is no other available method to retrieve the inventory data. Finding data stored in the *HKLM\CURRENT_USER* registry key is a good example.



Chapter 11: It's Better To Test Now Than Be Testy Later

It wasn't raining when Noah built the ark.

—Howard Ruff

Have you ever watched a pilot walk around an aircraft touching and checking all the moving parts before take-off? Kind of makes sense that the pilot would want to make sure everything was in working order before taking off and being 10,000 feet in the air before realizing that the plane was missing a part right?

Well, modifying SMS hardware inventory is really no different. No, you won't fall 10,000 feet if something goes wrong, but hiding under your office desk until everyone else goes home is a real possibility if things go terribly wrong!

Just like a pilot checking to ensure both wings are still attached to the aircraft, you should be checking and testing all of your changes before committing them to your production site server for inventories.

The best way to do this is with a lab setup separate from your production network. It doesn't have to be anything fancy, just a basic, small network that mimics your production network. This way, when you modify your MOF or run scripts to create or collect hardware inventory information you are not exposing your production network to any risks associated with your attempts to alter the natural order of things.

Remember that when you compile MOF edits you are modifying the WMI repositories on your clients and creating tables, views, and stored procedures for those modifications in the SMS database. This phase is called testing for a reason; don't take the risk of introducing bad data into your production environment! These mistakes can all be cleaned up, but it's much easier not to have to deal with the mess on your production network.

I can't get my boss to buy me a box of floppy disks so how am I going to get him to buy me the hardware and software to run a separate lab so I can do testing?

I'm a realist. I know that a lot of organizations out there won't support paying for servers and software so you can enjoy the benefits of having your own play lab.

So here's what I suggest: get a hold of some relatively inexpensive software that allows you to have your own virtual lab. I use Microsoft Virtual PC (<http://www.microsoft.com/windows/virtualpc/default.mspx>) and it hasn't done me wrong yet.

Using this software, all you need is enough RAM in your workstation to run the various operating systems and applications you'll need for your lab network. This virtual network doesn't even necessarily have to be accessible from your production network or the Internet at all. But you should still use licensed software though.

So one way or another you've got your lab right? No? OK, you can't get the assets you need for a "real" lab, and you can't even get the boss to buy Virtual PC for you, so what do you do now?

Well, now you need to draw straws with your co-workers to decide who is going to sacrifice their system to the MOF testing gods.

Maybe that isn't such a good idea. The next best option is to have a dedicated testing machine upon which to practice your inventory white magic. Choosing to run tests on one's own workstation has also been known to happen.

I didn't believe in all this testing stuff at one time and used my workstation to test my edits. A few WMI repository rebuilds later and I was praising Virtual PC.

Regardless of the method you choose, we're going to call this guinea pig—virtual or not-- the test system from now on. Remember, *always, always, always* test your inventory modifications before reaching out and touching every SMS client you have.

You may create some glorious modification that performs supernatural inventory feats, but you won't know those feats come at the price of hardware inventory taking five minutes to run on a workstation or creating a 2GB MIF file unless you test first. You must test your modifications to ensure that you are accomplishing your goals at a price you can afford.



Remember that class changes always have to be made in the primary site server's SMS_def.mof. If you're not using a lab to test your modifications, you need to be doubly sure you are confident in what you are doing.

The most common SMS inventory modification is to modify the SMS_def.mof itself. Whether adding a reporting class for an existing class or utilizing one of the other

various methods I've discussed earlier, sometime, sooner or later, you *will* modify your SMS_def.mof. Why else are you reading this book?

There are basically six checkpoints to ensure that your SMS_def.mof modifications are not going to land you at 10,000 feet looking for that left flap. I like to call these things the six checkpoints to SMS_def.mof:

- ☑ Verify the MOF syntax with MOFCOMP.
- ☑ Compile the MOF on a test machine.
- ☑ Use WBEMTEST to check for the reporting class in the correct namespace.
- ☑ Initiate a hardware inventory.
- ☑ Verify the hardware inventory process.
- ☑ Verify the data on the SMS site server.



Verify the MOF Syntax With MOFCOMP

Please do this. I don't know how many emails I've gotten and a simple MOFCOMP check has shown the problem to be invalid syntax. Syntax is a killer. The way to use MOFCOMP to verify your SMS_def.mof or static MOF syntax is to go to a command prompt and type in *MOFCOMP -check your:MOF.MOF*.

I'm going to use our old buddy from Chapter 7, the SMS_Installs RIP example, to illustrate this for you.

Let's pretend I've just created the SMSinstalls.MOF file in notepad. Now I want to test it to ensure that it is free of syntax errors before adding this information to the WMI repository of my test system.

My "mini-MOF" looks like so:

```
//-----
// Start of SMS_Installs
//-----

#pragma namespace("\\\\.\root\CIMV2")

[ dynamic, provider("RegProv"),
ClassContext("local\HKEY_LOCAL_MACHINE\SOFTWARE\SMS_Installs")
]
```

```

class SMS_Installs
{
  [Key] string ApplicationName;
  [PropertyContext("Description")] string Description;
  [PropertyContext("InstalledDate")] string InstalledDate;
  [PropertyContext("Version")] string Version;
};

#pragma namespace("\\\\.\\root\\CIMV2\\SMS")

[SMS_Report(TRUE), SMS_Group_Name("SMS_Installs"),
SMS_Class_ID("SMSExpert|SMS_Installs|1.0")]
class SMS_Installs : SMS_Class_Template
{
  [SMS_Report(TRUE),Key] string ApplicationName;
  [SMS_Report(TRUE)] string Description;
  [SMS_Report(TRUE)] string InstalledDate;
  [SMS_Report(TRUE)] string Version;
};
//-----
// End of SMS_Installs
//-----

```

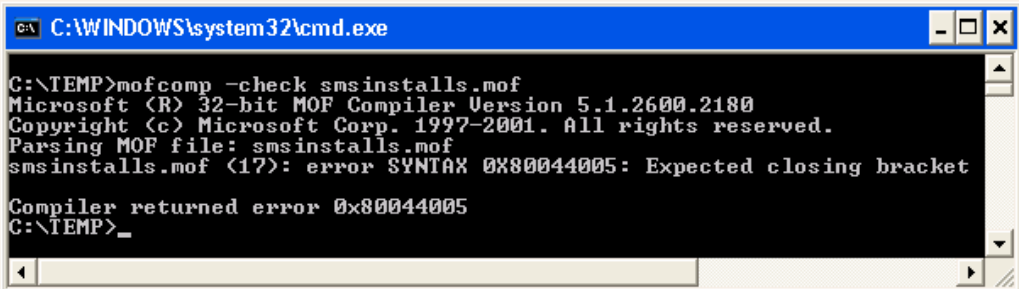
The actual command I would use here is:

```
MOFcomp -check SMSinstalls.MOF
```

Whenever you modify the SMS_def.mof you should run this check as soon as you *think* you have everything right. MOFCOMP is real good at exposing typos and forgotten symbols like ; and].

If you do happen to forget something, MOFCOMP will thoughtfully remind you and even give you a hint as to what you have done wrong, as in Figure 11.1:

Figure 11.1



```

C:\WINDOWS\system32\cmd.exe
C:\TEMP>mofcomp -check smsinstalls.mof
Microsoft (R) 32-bit MOF Compiler Version 5.1.2600.2180
Copyright (c) Microsoft Corp. 1997-2001. All rights reserved.
Parsing MOF file: smsinstalls.mof
smsinstalls.mof (17): error SYNTAX 0x80044005: Expected closing bracket
Compiler returned error 0x80044005
C:\TEMP>_

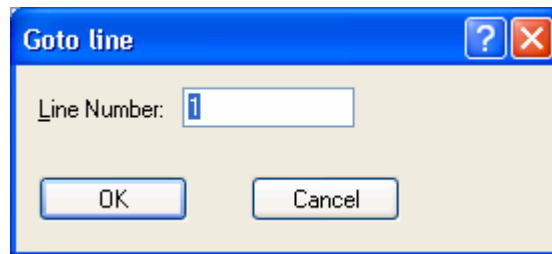
```

Hmm, that looks important, and MOFCOMP doesn't look happy for some reason. See that number in the parenthesis on the fifth line? That is MOFCOMP's subtle way of saying, "Hey man, you messed up on line 17 and I'm taking my ball and going home now".

So let's go check out line 17 because it appears there is a missing expected closing bracket. Time to open up that peculiar MOF file and see what's going on.

Here's a neat trick. If you are running Windows 2000 or above—and I hope you are by now—then all you have to do is hit <Ctrl-G> in Notepad (with word wrap turned off). You should see something like Figure 11.2:

Figure 11.2



Just type in 17 (since that's the line MOFCOMP told us was messed up) and you are there. Checking out line 17 from the `C:\Temp\SMSinstalls.MOF` we see this:

```
[PropertyContext("Version")] string Version;
```

Hmm, this line looks good to me. Maybe this MOFCOMP thing isn't all it's cracked up to be. But wait a minute, let's look at line 16:

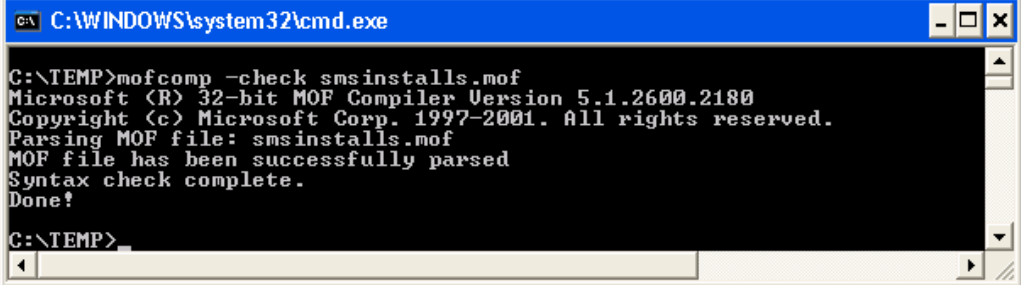
```
[PropertyContext("InstalledDate")] string InstalledDate
```

Something looks different about this line, right? Maybe something like, oh I don't know, *a closing bracket*? Remember that MOFCOMP will continue to process the MOF file until it finds an error and then it will stop ... right then, right there. When it ran through this MOF edit, it was happily processing until it got to line 17, where it noticed that it should not continue on to the next line when there wasn't a semi-colon at the end of the preceding line.

In other words, MOFCOMP didn't notice a problem until it got to line 17 when the error was really on line 16. Did that make sense? You have to keep looking and thinking "outside the MOF" to find something out of the ordinary when things go wrong. This doesn't always happen, but in case it does, don't get wrapped around the handle because line 17 looks perfectly fine to you!

So now we've exposed and corrected the syntax error on line 16 by adding the semi-colon at the end of the line. It's time to re-check the MOF with MOFCOMP:

Figure 11.3



```

C:\WINDOWS\system32\cmd.exe

C:\TEMP>mofcomp -check smsinstalls.mof
Microsoft (R) 32-bit MOF Compiler Version 5.1.2600.2180
Copyright (c) Microsoft Corp. 1997-2001. All rights reserved.
Parsing MOF file: smsinstalls.mof
MOF file has been successfully parsed
Syntax check complete.
Done!

C:\TEMP>

```

In a perfect world the last line in compilation would always be *Done!* like it is in Figure 11.3. This tells us that our MOF edit has successfully passed MOFCOMP's syntax check. When MOFCOMP is happy, everyone is happy. With the syntax check successful, it's time to check block number 1.

- Verify the MOF syntax with MOFCOMP

Compile the MOF on a Test Machine

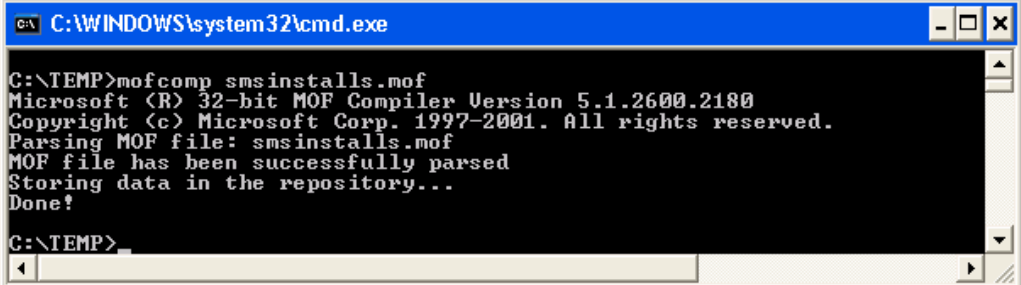
Once you've verified the MOF syntax it's a simple matter of running MOFCOMP again to compile the MOF on the system you've chosen as your guinea pig—er, test system. Compiling, as you may remember, is simply adding the contents of the MOF file to the system's local WMI repository so that you can inventory it later with SMS.

To do this you simply take the *-check* out of the command line that we used before. Your command line should now be:

```
MOFCOMP SMSinstalls.MOF
```

Your command prompt window should now look something like Figure 11.4:

Figure 11.4



```

C:\WINDOWS\system32\cmd.exe

C:\TEMP>mofcomp smsinstalls.mof
Microsoft (R) 32-bit MOF Compiler Version 5.1.2600.2180
Copyright (c) Microsoft Corp. 1997-2001. All rights reserved.
Parsing MOF file: smsinstalls.mof
MOF file has been successfully parsed
Storing data in the repository...
Done!

C:\TEMP>

```

We must still be in a perfect world. MOFCOMP says *Done!* The data from the SMSinstalls.MOF file should now be successfully stored in the test system's WMI repository—should be, we'll verify it in a second.

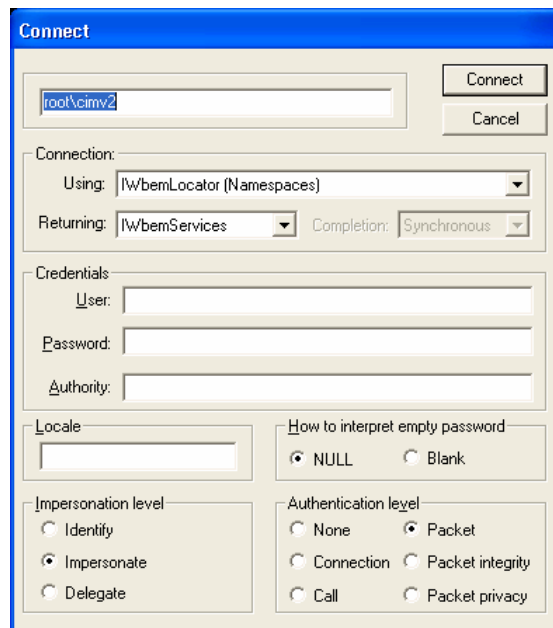
- Compile the MOF on a test machine.

Use WBEMTEST to Check for the Class

Now everything is good in the world, right? Yep, and I've got a nice mountain view duplex to sell you in Florida too! When it comes to modifying your MOF you must think like the X-files—trust no one.

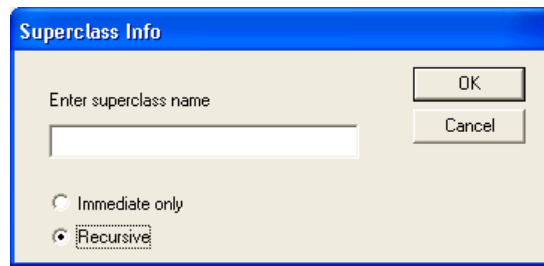
Time to use our old friend from Chapter 3, WBEMTEST. Just hit start/run/type in WBEMTEST and hit OK. Connect to the *root\CIMV2* namespace—that's where we put the data class for the SMS Installs information.

Figure 11.5



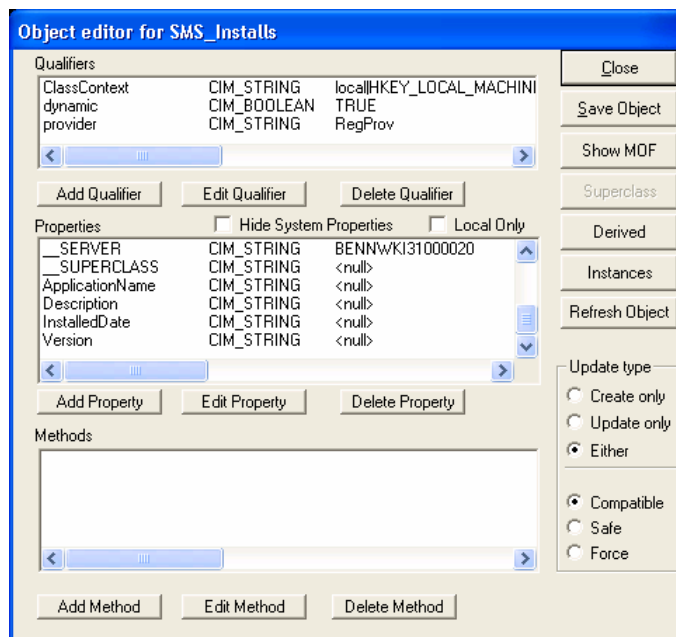
Hit connect and let's recursively search for the class we've created.

Figure 11.6



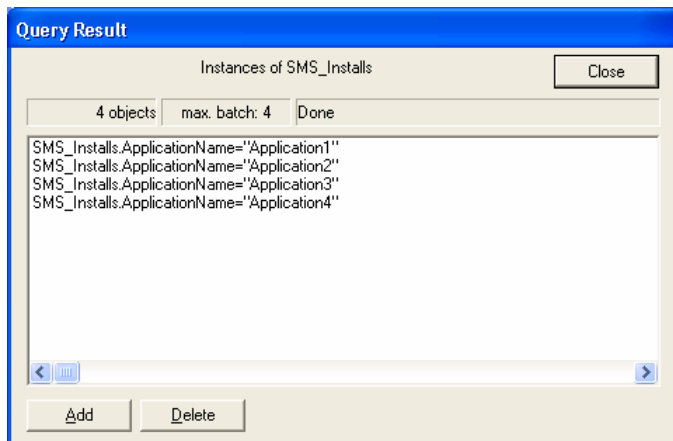
Once we've connected, let's look for the SMS_Installs class we created.

Figure 11.7



Great! The class is there along with all the fields we wanted. But let's see if it's doing anything besides sitting there and looking pretty. Time to get the instances from this class by clicking on the, you guessed it, instances button!

Figure 11.8



Hurray! Our MOF edit worked! There are four instances of SMS_Installs in my test system's WMI which correspond to the four entries in the registry of the test machine. Now we are just waiting on a hardware inventory cycle to come grab them up and add them to the database. Things are looking good so far.

Of course, this will never happen unless we add our SMS_Installs MOF addition to our SMS_def.mof on the site server. Remember, we've compiled the MOF edit on our test system, but SMS has no clue that we're even interested in this class at this point.

Let's go do that now—add the entire SMS_Installs.MOF to the bottom of the SMS_def.mof stored on the site server. Remember, it's going to take a few minutes for the modified SMS_def.mof to get through the syntax check and file to policy conversion process for Advanced Clients.

Whether you have Advanced Clients or Legacy Clients, make sure you wait a few minutes (5-10) for the new SMS_def.mof information to get to the management point or client access point respectively.

Moving on, you know what to do now:

- Use WBEMTEST to check for the reporting class in the correct namespace.

Initiate a Hardware Inventory

So we've created the MOF edit, checked its syntax, compiled it on our test system, and verified that the class is in the correct namespace. Time now to initiate a hardware inventory and see what this baby really does! I'll run through the steps quickly for both the Advanced Client and Legacy Clients here.

Advanced Clients:

From your test system, open the Systems Management applet in control panel, click on the actions tab and select Machine Policy Retrieval & Evaluation Cycle.

Wait a minute; this section is called Initiate a hardware inventory and you're telling me to select Machine Policy Retrieval & Evaluation Cycle?

OK, hold on, someone was sleeping through Chapter 1. SMS Advanced Client hardware inventory is controlled by a hardware inventory policy *based on the contents of* the SMS_def.mof not the SMS_def.mof itself.

After initiating a machine policy retrieval and evaluation cycle, wait a few moments and then highlight the hardware inventory cycle and click the Initiate Action button.

Legacy Clients:

To force a hardware inventory on a Legacy Client, open the Control Panel; double-click on System Management; click on the third tab; click on the hardware inventory agent; click on the Start Component button.

- Initiate a Hardware Inventory.

Verify the Hardware Inventory Process

So you've hit the appropriate buttons to initiate the hardware inventory, but how do you know if it's really doing anything? Logs. Logs are your friends. They're not much fun at parties, but they sure are useful at work!



If you are using SMS Trace to view the inventory logs you can see the inventory actions as they are occurring in real time. I'm a big fan of SMS Trace, but you can use notepad or any log viewer that you so desire.

The latest download location for SMS Trace is the download page for the Systems Management Server 2003 Toolkit 2 and can be found at:

<http://www.microsoft.com/SMServer/downloads/2003/tools/toolkit.msp>

Advanced Client:

To follow along with all the hardware inventory fun for the Advanced Client you need to open up the InventoryAgent.log file. This file is located at:

`%WINDIR%\system32\CCM\Logs\InventoryAgent.log`

With SMS Trace you can watch the inventory progress. If things go bad for your new class, SMS Trace very thoughtfully highlights the error in red. If things go well, you'll have to use the search function to search for your class name to ensure it was picked up. Regardless of the tool you use to search for it, you're looking for a line that says something like this:

```
Collection: Namespace = \\.\root\CIMV2; Query = SELECT __CLASS, __PATH,
__RELPATH, ApplicationName, Description, InstalledDate, Version FROM SMS_Installs;
Timeout = 600 secs.
```

That tells you that the class has made it to the big time and is being inventoried successfully.

Legacy Clients:

The Legacy Client log file for your particular attention in hardware inventory actions is the `hinv.log` file. This file is located at:

```
%WINDIR%\ms\SMS\logs\hinv.log
```

The process is basically the same here as it was with the Advanced Client. Just look for a line that shows the new class being processed.

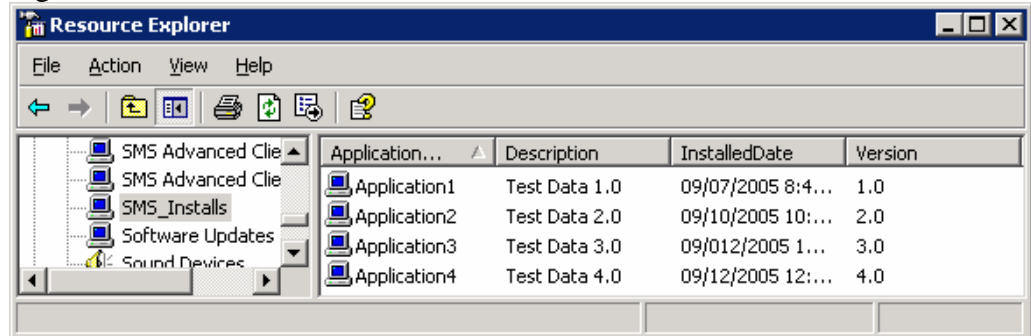
Should there be any issues with either the Advanced Client or Legacy Client processing the new class, these log files will more than likely contain the first clues to correcting the problem. This isn't the troubleshooting chapter, so for now, we'll just stay in our perfect world and move along.

- Verify the hardware inventory process.

Verify the Data on the SMS Site Server

Verifying the data on the SMS site server is pretty easy. Just open resource explorer for your test client and look for the class's group name under the hardware section. You should see something like Figure 11.9:

Figure 11.9



This is the point where you will get all giddy, jump up and down in excitement and attempt to explain your feat to your family and co-workers. They will all give you the “what look”. Do not be discouraged; this happens to us all, and there are always people like me around who will say how cool you are and stuff.

- Verify the data on the SMS site server.

And so we come to the end of another exciting chapter. Just remember, there is a lot of waiting involved in making SMS_def.mof modifications. Don't try to rush things! Go get a cup of coffee or talk about last night's football game with your co-workers and let SMS do the driving for a while between steps.



Chapter Summary

You should be checking and testing all of your changes before committing them to your site server for production inventories.

The best way to do this is with a lab setup separate from your production network. It doesn't have to be anything fancy, just a basic, small network that mimics your production network. **If you can't get a physical lab, consider creating a virtual one** by using Microsoft Virtual PC :

<http://www.microsoft.com/windows/virtualpc/default.mspx> or by dedicating a specific test computer or computers to test your inventory extensions.

The way to use MOFCOMP to verify your MOF syntax is to go to a command prompt and type in ***MOFCOMP -check yourMOF.MOF***. **Take the *-check* out of the command line to commit your changes to the WMI repository.** Your command line would then be: *MOFCOMP yourMOF.MOF*

SMS Trace allows you to view the inventory logs and can see the inventory actions as they are occurring in real time. The latest download location for SMS Trace is the download page for the Systems Management Server 2003 Toolkit 2 and can be found at: <http://www.microsoft.com/SMServer/downloads/2003/tools/toolkit.mspx>

The Advanced Client inventory log is named InventoryAgent.log and is located at: *%WINDIR%\system32\CCM\Logs\InventoryAgent.log*

The Legacy Clients inventory log is named hin.v.log and is located at: *%WINDIR%\ms\SMS\logs\hin.v.log*

- Verify the MOF syntax with MOFCOMP.
- Compile the MOF on a test machine.
- Use WBEMTEST to check for the reporting class in the correct namespace.
- Initiate a hardware inventory.
- Verify the hardware inventory process.
- Verify the data on the SMS site server.



Chapter 12: Pulling the Trigger

"It's kind of fun to do the impossible." —Walt Disney

With a MOF edit successfully tested, the only thing left to do is to move the modification to the production network. We've already covered this a little, but there are some important things to know when it comes to distributing the SMS_def.mof and getting the WMI repositories of your clients updated correctly—especially in the case of Advanced Clients

Back Up the Original SMS_def.mof

Whenever you want to make a change to the SMS_def.mof, a good practice to get into is to back-up the old SMS_def.mof just in case things don't go your way. SMS 2003 SP1 does this automatically for you, but it's still a good habit to get into. This is where that funny SMS_def.mof.bak file comes from in the `SMS\data\bin\archive` folder.

Replace the Original SMS_def.mof on the Site Server

Once you've backed up the old SMS_def.mof, save the new, tested SMS_def.mof to the `SMS\inboxes\clifiles.srv\bin` folder on the primary site server to get the new MOF distributed to the first primary site.

Update the Site Hierarchy

Once you've finished distributing the SMS_def.mof to the first site, you'll need to copy the new SMS_def.mof file to *all* your primary site servers. Each site maintains its own version of the SMS_def.mof. If you update the SMS_def.mof on one site in your hierarchy you need to remember to distribute it to all the other sites as well if you expect to retrieve the new information from their clients.

What if I don't want to collect this information from my child sites? Do I still need to get the new SMS_def.mof on my lower level site servers?

Yes. If you have created collections at the parent site utilizing information from your new MOF extensions as criteria, when these collections are propagated throughout

your site any site server without knowledge of the modified SMS_def.mof will not be able to create those collections. It will cause a ton of status messages saying so. Even if you really don't want to use the inventory extensions you've created at lower sites, it's still a good idea to get that modified SMS_def.mof distributed correctly.

SMS_def.mof Updates: Behind the Scenes

Once you've updated the SMS_def.mof on the site server, SMS will automatically compile it and distribute it via its magical powers to the management point as a new inventory policy for Advanced Clients and to the client access points for download by Legacy Clients.

I suppose I owe you a little better explanation than “magical powers”. OK, here goes.

When you place a new SMS_def.mof on the site server, many things go into action behind the scenes. I'm going to go through a quick explanation of it here; this is all you need to know if things go right. For a deeper explanation for when things go wrong, check out the troubleshooting chapter.

Legacy Clients:

The master copy is saved on the site server in the SMS\inboxes\clifiles.sr\binv folder. This SMS_def.mof is, in turn, copied to the CAP by inbox manager. Legacy Clients download the new file at their next client refresh cycle—every 23 hours. When the next scheduled inventory is run, the clients will compare this new MOF to their existing local copy of the MOF, decide the one from the server is newer, and compile it. Once it has been compiled, the client will kick off a *full* hardware inventory. Once the inventory is complete, the client will send its inventory results in .MIF format back to the CAP, which then forwards it on to the primary site server and the SMS database.



Do not copy the new SMS_def.mof directly to a Legacy Client or CAP. These files will be overwritten by the SMS Site Server's copy.

Advanced Clients:

The new SMS_def.mof is saved on the site server in the SMS\inboxes\clifiles.sr\binv folder. This file is compiled by data loader to get the new information into the SMS database. The new MOF file is then copied to the CAP for Legacy Client support.

Policy provider creates the inventory policies for Advanced Clients and sends them to the management point for download by the clients at their next machine policy refresh interval. This interval is set to once an hour by default. The Advanced Client

downloads and evaluates the policy, which gets all the new goodies from the modified MOF into the WMI repository and executes a normal hardware inventory cycle.

Unlike the Legacy Client, the Advanced Client only performs a *delta* hardware inventory at this point. The inventory information is sent back to the management point in .XML format. From there it finds its way back to the primary site server and the SMS database.



There is a built-in two-minute interval between the time the Advanced Client downloads and evaluates a new hardware inventory policy.

I kind of skipped over an important point here that I think I'll just go ahead and get out of the way now.

When you copy the new SMS_def.mof to the site server, the server itself compiles the MOF with MOFCOMP to ensure that the file is syntactically correct. If all the data within the file is correct, then things progress as described above. If not, the file is moved to the SMS\data\himarchive folder.

Yes, this is where I said SMS 2003 SP1 automatically backs up your SMS_def.mof file. It's also where SMS will place your SMS_def.mof if it contains syntax errors. The only difference is that, instead of naming your file SMS_def.mof.bak, it will call it SMS_def.mof.bad.bak.

SMS_def.mof Back Ups: Good and Bad

SMS will keep up to six copies of your old SMS_def.mof files—good and bad—in the SMS\data\himarchive folder. The first good backup gets named SMS_def.mof.bak, and then the successive backups are named SMS_def.mof.bk0 through SMS_def.mof.bk4.

Bad SMS_def.mof files receive basically the same treatment. These files are named SMS_def.mof.bad.bak, and SMS_def.mof.bad.bk0 through SMS_def.mof.bad.bk4. For the next SMS_def.mof backups (good and bad) the process just starts again from the beginning overwriting the oldest record first.

The above scenario works fine if you are only making changes to existing classes or providers. Otherwise you're going to have to compile the MOF containing any new data classes/providers on your Advanced Clients. Advanced Clients require the local compilation for both new classes and providers, as opposed to Legacy Clients, which only require this when a new provider needs to be registered.

There are a couple of options here. You can update the SMS_def.mof on the site server and locally compile that copy on the clients. Only caveat here is that SMS will attempt to inventory data classes that aren't in existence on clients that haven't

compiled the new SMS_def.mof yet. Technically, this option won't hurt anything—if SMS says to inventory a non-existent class on your clients, you will just get errors in your hinvt.log or InventoryAgent.log respectively saying that the class doesn't exist.

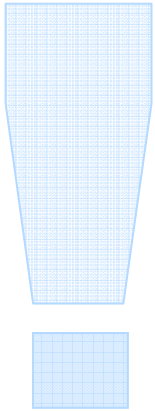
Using a Mini-MOF

Another option is to create the “mini-MOF” containing only the new data class and/or provider information and compile it locally on the clients before adding the reporting class to the SMS_def.mof on the site server. Either way, your clients will have to compile a MOF file to get the data you are after into the database. That MOF file will have to be available for new clients that join your site so that it can be compiled by them as well.

How do you accomplish this? SMS's built-in software distribution, of course. Just create a package with a program consisting of your custom MOF edits in one big “min-MOF”, and a command line of `MOFCOMP <yourMOFEdits>.MOF`. Remember, if you want to use the `MOFCOMP -autorecover` option, then the file will need to be copied locally to the client in a *safe and stable* location before compilation occurs. Advertise this program on a recurring schedule according to your personal situation to pick up new clients and update older ones who may have missed your latest and greatest modification.

This technique also works for advertising scripts that populate the client WMI repositories for inventory.

If you're interested in doing this and don't think that's enough information do not fret my friend. I'll talk a lot more about this technique in Chapter 14, Troubleshooting and Tips.





Chapter Summary

Always back up your SMS_def.mof before making changes. SMS 2003 SP1 does this automatically for you by renaming your old SMS_def.mof to SMS_def.mof.bak and storing it in the SMS\data\hinvararchive folder.

Each SMS Site maintains its own version of the SMS_def.mof. You should copy a modified SMS_def.mof to each primary site within your hierarchy to maintain standard inventory processes and avoid odd collection membership issues later on.

Legacy Clients download the modified SMS_def.mof file from their CAP during their client refresh cycle (every 23 hours by default) and perform a **full hardware inventory**.

Advanced Clients receive a new inventory policy from their Management Point during their next policy refresh interval (once an hour by default) and perform a **delta hardware inventory**.

There is a built-in two-minute interval between the time the Advanced Client downloads and evaluates a new hardware inventory policy

Advanced Clients require the local compilation for both new classes and providers as opposed to Legacy Clients, which only require this when a new provider needs to be registered.

Chapter 13: Cleaning up

The greatest mistake you can make in life is to be continuously fearing you will make one.
—Elbert Hubbard

Hi, my name is Jeff, and I have a bad habit. I have a corner of my desk where I pile up “stuff”. I used to call this my inbox, but “stuff” seems to fit this particular location better now. This stuff is all useful material that I’ve printed out from every corner of the Internet and beyond. Some day I’m sure I’ll need to reference it for one reason or another.

My problem is, every now and then I’ll need to do something, and, realizing that the information I need is in the pile, start the laborious process of sorting through it to find the bits and pieces I need. Sure, by my third cup of coffee I’ve usually found what I’m looking for—or worse, found three copies of it! This shames me into doing something my wife wishes I would do more often—clean up!

The same thing can happen to your SMS site if you’re not careful. No matter how dedicated you are to your SMS site, things happen. Data classes become unneeded on your clients. Tables, views and stored procedures left behind in the SQL database by some long forgotten SMS admin are discovered. You play with the flip-flop MIF example from Chapter 10 and forgot to clean up the database. You get the picture, right?

There are a couple of steps to cleaning up undesired data from your SMS site. Remember, you don’t just need to clean up your database; your clients are in need of a good scrubbing as well.

- Clean up the SMS_def.mof on the site server.
- Remove the unnecessary WMI classes from your server and clients.
- Remove the class data from the database.

Clean up the SMS_def.mof on the Site Server

The first thing that needs to be done is to clean up the master SMS_def.mof for the site. This makes sense, right? I mean, it's kind of pointless to start with cleaning up the database tables when the next time a client runs hardware inventory those tables will just be recreated. Don't forget to get the newly modified SMS_def.mof copied to all the sites in your hierarchy that use that old version.

So, open up your SMS_def.mof from the SMS\inboxes\clifiles.srv\himv folder. If you just want to stop collecting class data then just turn the class reporting qualifier from TRUE to FALSE and SMS will not inventory the class anymore.

You could also get crazy and just delete the offending class completely from your SMS_def.mof. I say get crazy, because as well all know, as soon as you delete the class you're going to be told by management that now they would like to see that "widget serial number data" you said you had started inventorying six months ago! Now you're going to have to go to your own version of "the pile" and remember what that class looked like and go through the entire process again.

A better option is just to comment out the class like we talked about in Chapter 3. Just put a /* at the top of the section and a */ after the end of it in case you ever need it again.

Now clients conducting hardware inventories during your cleanup proceedings won't sabotage your work by submitting unneeded inventory data to your database. However, this still leaves that WMI and reporting class data on all your client machines...that's not cool.



Something else to think about here. If you're collecting inventory data for the class you're planning on deleting using MIF files instead of the SMS_def.mof, ensure that you delete those files from the appropriate client directory and/or delete the advertisement causing a script to run that creates them.

Remove the Unnecessary WMI Classes From Your Clients

Here's where my consolidated "mini-MOF" idea for MOF customizations comes into its own. Remember, I use this "mini-MOF" to regularly maintain the custom WMI data and reporting classes on my Advanced Clients by scheduling it to be compiled via SMS software distribution on a regular basis.

To delete the unneeded classes from the local system WMI you're going to have to compile a modified MOF containing the now famous `#pragma deleteclass` line. Just open up your "mini-MOF" and comment out the entire section for your targeted class—or delete the class totally and roll the dice that you'll never need it again. Make sure you use the `#pragma deleteclass` command on both the data and reporting classes.

Place these four magic lines, either just above the commented section or where the old class information used to be if you deleted it. I'm using the custom `TimeZoneInfo` class for this example:

```
#pragma namespace("\\\\.\\root\CIMV2")
#pragma deleteclass("TimeZoneInfo",NOFAIL)
#pragma namespace("\\\\.\\ROOT\CIMV2\SMS")
#pragma deleteclass("TimeZoneInfo",NOFAIL)
/*
...TimeZoneInfo class information that I used to like, but now I don't...
*/
```



Do not delete a data class unless you created it! SMS and WMI work in mysterious ways sometimes, and deleting a class may result in your having to rebuild the entire WMI repository to get the system to function correctly again.

Now, just update the distribution points for your scheduled MOF compilation program, and sit back and enjoy the show as the classes are magically deleted from your clients' WMI repositories.

If you are running Legacy Clients, just make these changes on the master `SMS_def.mof` on the site server and those clients will delete the classes during their normal inventory, since they'll be compiling the full `SMS_def.mof` when it changes anyway.

This is the ticket if you have deployed the modifications site-wide. If you only want to delete the class from a test system, just open `WBEMTEST`, search for your unwanted class and hit delete (take it out of the `root\CIMV2` and `root\CIMV2\SMS` namespaces).

Now that the `SMS_def.mof` has been cleaned up and the client's WMI repositories have been cleansed, it's time to go after that useless data cluttering up the database.

Don't forget to clean this class out of your site server's WMI repository as well. If you don't, you may still find these "ghost" classes as attributes for building queries!

Remove the Class Data From the Database

I'm going to show you the “hard” way to do this first, and then the “easy” way. The hard way in this case is manually deleting the tables yourself using the Microsoft SMS 2003 Toolkit 2 tool DELGRP.EXE. This tool can be found at:

(<http://www.microsoft.com/SMServer/downloads/2003/tools/toolkit.mspix>)

DELGRP

DO NOT just go into the SQL database and manually delete the TimeZoneInfo data and history tables! Doing so will make SQL very angry with you and basically ruin your whole day.

Remember that when you modify the SMS_def.mof and client configurations to report on new data, many different items are created in your SMS database. These items include (I'll give examples for the Flip Flop Information class that I'm about to delete):

- **Data and history tables**
 - Flip_Flop_Information_Data
 - Flip_Flop_Information_History
- **Views for both tables**
 - v_GS_Flip_Flop_Information
 - v_HS_Flip_Flop_Information
- **Stored procedures for both tables**
 - d Flip_Flop_Information_DATA
 - p Flip_Flop_Information_DATA

We need to whack all of those back to where they came from—our imagination in this case. So we need to get to work with DELGRP.

To use DELGRP, simply copy the file to your site server and open up a command window focused on the directory where you copied DELGRP. Follow the procedure below for each data class that you want to delete (I'll give you the syntax in a minute):

1. Stop the SMS Executive Service

2. Delete the class by referencing the exact class name 'i.e.' "Microsoft|Class Name|1.0" (don't forget the quotes!)
3. Restart the SMS Executive Service

An easy way to expedite this process is to write a quickie .BAT file to do the work for you. If you run *DELGRP /?* from the command line, you will be rewarded with the syntax, and even a basic script example to do your dirty work:

SMS Delete Inventory Class Utility

Usage:

```
DelGrp /L
DelGrp "<Inventory Class Name>" [/C | /H]
```

Options:

```
/L - List all inventory classes available for removal.
     Displays the number of rows in the data table for each one.
/C - Display collections that reference this class.
/H - Remove history data and make the class a no-history class.
     This will prevent history from being maintained for the class.
```

If the class name is specified with no options then the inventory class and all associated data are removed.

To remove unused inventory classes from the SMS database:

- Stop the SMS_EXECUTIVE service.
- Use the /L option to identify classes with no data.
- Use the /C option to make sure no collections will be affected.
- Remove the class.
- Start the SMS_EXECUTIVE service.

Example:

```
NET STOP SMS_EXECUTIVE
DelGrp /L
DelGrp "COMPANY|CUSTOM_CLASS|1.0" /C
DelGrp "COMPANY|CUSTOM_CLASS|1.0"
NET START SMS_EXECUTIVE
```

NOTE: Default classes created by SMS setup cannot be removed or altered.
You must have db_owner permissions for the SMS database to remove classes.

See the script part in bold? Just cut and paste the part after *Example:* into notepad, change the name of the class to delete to your custom class, and save it with a .BAT extension. Running it on the site server (you have to have the script in the same directory as DELGRP for it to work) and it will clean out those tables for you.

One thing extra I do when I use this method is to add another line with just the word *pause*. If you are not familiar with batch files, the word *pause* will cause the script to

halt at completion and display the “Press any key to continue” option. I like this because it stops the script long enough for me to tell if it worked!

Now that the data and history tables for the class are gone, you just need to search through your SQL Enterprise Manager views and stored procedures to delete those that reference the class name you just deleted using DELGRP.

There will be two views to delete. One will start with v_GS and the other with v_HS. Both will be followed by your class name.

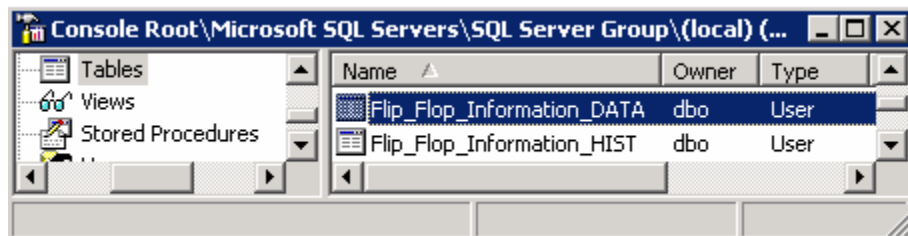
There will be two stored procedures for your class as well. One starts with a “d” and the other a “p”. Delete at will, and be done with this class deletion.

How about an example for those of you that were playing with the flip flop NOIDMIF example? C’mon you know you were playing with it!

If you dropped the flipflop.mif in your NOIDMIF folder prior to running a hardware inventory, you will have the information below in your database:

Tables for flip flop information:

Figure 13.1



Views for flip flop information:

v_GS_Flip_Flop_Information0 and v_HS_Flip_Flop_Information0

Stored procedures for flip flop information:

dFlip_Flop_Information_DATA and pFlip_Flop_Information_DATA

I’d show you pictures here, but that would be just a little bit of overkill—not to mention taking up two pages for the pictures!

Now, let’s get rid of that flip flop information and keep your boss from knowing you’ve been playing with the database! Using DELGRP, your batch file to delete the table information would look like this:

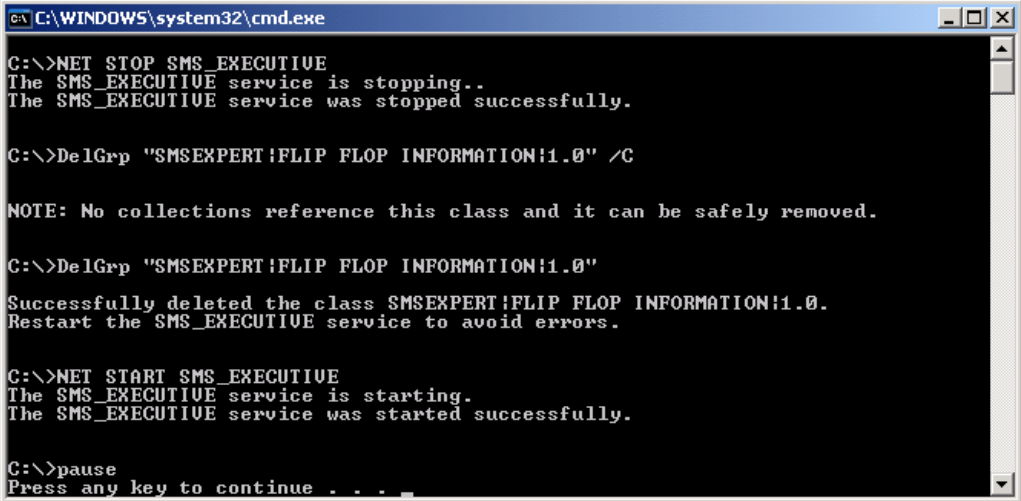
```

NET STOP SMS_EXECUTIVE
DelGrp /L
DelGrp "SMSEXPRT|Flip Flop Information|1.0" /C
DelGrp "SMSEXPRT|Flip Flop Information|1.0"
NET START SMS_EXECUTIVE
PAUSE

```

You don't have to use the *DelGrp /L* line in there if you know exactly which class you're going after, and since it's really not necessary to this example I've taken that part out of my script. Saving the script with a BAT extension on the site server in the same folder as DELGRP and running it would give you results shown in Figure 13.2:

Figure 13.2



```

C:\WINDOWS\system32\cmd.exe

C:\>NET STOP SMS_EXECUTIVE
The SMS_EXECUTIVE service is stopping..
The SMS_EXECUTIVE service was stopped successfully.

C:\>DelGrp "SMSEXPRT|FLIP FLOP INFORMATION|1.0" /C

NOTE: No collections reference this class and it can be safely removed.

C:\>DelGrp "SMSEXPRT|FLIP FLOP INFORMATION|1.0"

Successfully deleted the class SMSEXPRT|FLIP FLOP INFORMATION|1.0.
Restart the SMS_EXECUTIVE service to avoid errors.

C:\>NET START SMS_EXECUTIVE
The SMS_EXECUTIVE service is starting.
The SMS_EXECUTIVE service was started successfully.

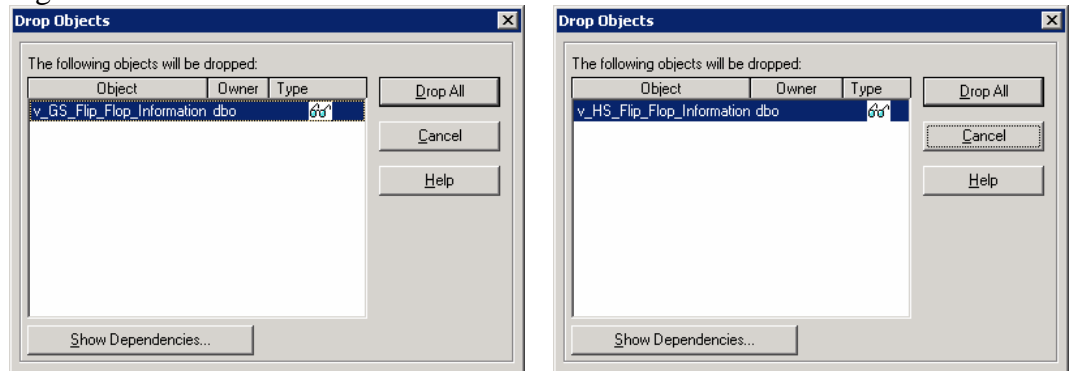
C:\>pause
Press any key to continue . . .

```

Be careful when using DelGrp.exe that you don't include any extra spaces in your class name—and don't forget those quotation marks around it either!

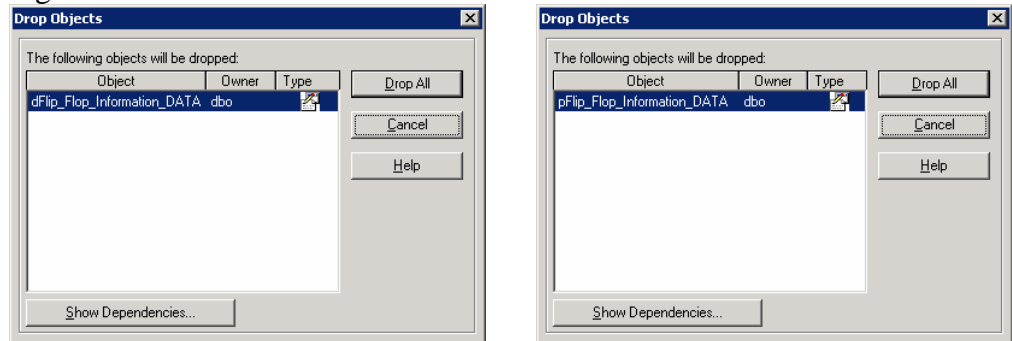
So there! Flip flop information tables are no more. Now you just need to scroll down and find those related views and stored procedures and delete them. When you right click on the View names, you just click *Delete* from the context menu. You will be presented with something similar to Figure 13.3 for the v_GS_Flip_Flop_Information view. Just click on *Drop All* to finish deleting the view. Do the same for the v_HS_Flip_Flop_Information view:

Figure 13.3



Now for those pesky stored procedures. Scroll down until you get to the `dFlip_Flop_Information_Data` stored procedure, right click on it, select *Delete* and you'll get the resulting Drop Objects dialog. Just like the views, click on *Drop All*. Then scroll down until you see the `pFlip_Flop_Information_Data` stored procedure. Right click, select delete, and hit Drop All on the Drop Objects dialog:

Figure 13.4



There! Flip Flop info is gone forever, right?! Yeah, as soon as you do that on *every primary site server in your hierarchy* that had the data in it.

Oh, by the way, don't forget to use `WBEMTEST` to delete those unneeded WMI classes from your site server(s) when you delete data classes that store their information in WMI when you're finished here. I'm not going to go into using `WBEMTEST` again. I think we've covered that in enough detail.

Time to move on to the easy way to do this. Remember this data class is created from a `NOIDMIF` file—one that I haven't deleted from my test machine's `noidmifs` folder yet. Run another hardware inventory on that system and ... guess what?! That class is back! Time to get rid of it...again.

I did this on purpose—let this be a reminder to you that when you do it for real, you need to get rid of those .MIF files for classes you’re trying to delete!

SMS Expert’s Site Sweeper Utility

Site Sweeper is extremely easy to use, but don’t let the ease of use fool you. This utility is extremely powerful and a great time saver. It can be found at:

<http://www.SMSexpert.com/products/sitesweeper.asp>

If you have multiple primary site servers, this tool is going to save you hours, if not days, of trying to clean up your databases. Site Sweeper “sweeps” your site(s) clean of classes you no longer want haunting the cyber halls of your site database. It takes out all the tables, views, and stored procedures from the database, as well as cleaning up the local WMI of the site server it is run against. It is extremely fast and efficient and since I’ve discovered it, I swear by it.

Once you open Site Sweeper, all you need to do is enter a primary site server name into the top left text box and hit the *Load Classes* button. Doing so will cause a list of classes to populate the area below the server name. I’m using <SMS Server> here for the example. Just type in your actual server name here to use the utility.

If you want to run the utility on multiple servers, click on the *Multiple Servers* radio button, and enter their names in the *Servers to Clean* area below the class names. Hitting the >> button adds them into the list of servers to clean.

If you are running this on the central site server for your entire hierarchy and want to clean up your entire setup, just hit the box to add all servers under the specified site to the server list. Don’t worry, Site Sweeper will get the site code for you too!

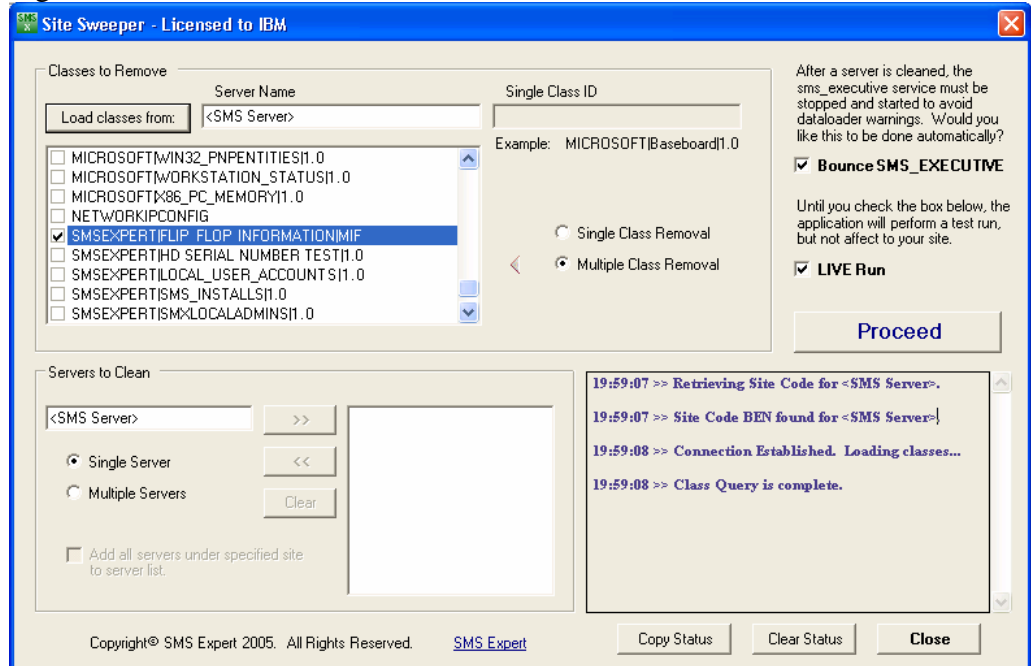
Yes, you can delete multiple classes from multiple sites and servers with one click here! Just leave the LIVE Run box unchecked while testing!

When you are ready to roll with the actual deletions, just check the *LIVE Run* box and hit proceed. Ensure that you check the *Bounce SMS_EXECUTIVE* box to stop that service before the deletions occur, and restart it afterwards.

I’m only going to delete one class from one server in this example. The *SMSEXPERT|FLIP FLOP INFO|1.0* class is going to be deleted from my site server named <SMS Server>. OK that’s not really the name of my server in case you were wondering. I just wanted to show you where your server name actually goes!

Anyway, I check the *Bounce SMS_EXECUTIVE* and *LIVE Run* boxes and hit proceed. This is what the interface looks like Figure 13.5 at this point:

Figure 13.5



The bottom right is where Site Sweeper tells you what is going on. Next is the excerpt from when the *SMSEXPERT|FLIP FLOP INFORMATION|1.0* class was deleted from my site with Site Sweeper.

```

20:04:03 >> Retrieving Site Code for <SMS SERVER>.
20:04:03 >> Site Code <XXX>found for <SMS SERVER>.
20:04:03 >> Connection Established. Loading classes...
20:04:03 >> Class Query is complete.
20:04:28 >> Starting TEST run.
20:04:28 >> Retrieving Site Code for <SMS SERVER>.
20:04:28 >> Site Code <XXX> has been retrieved for <SMS SERVER>.
20:04:28 >> Class Removal TEST
Class:    SMSEXPERT|FLIP FLOP INFO|1.0
Key:     116
Site:    <SMS SERVER> -<XXX>
20:04:28 >> TEST run has successfully completed.
20:05:08 >> Starting LIVE run.
20:05:08 >> Retrieving Site Code for <SMS SERVER>.
20:05:08 >> Site Code <XXX> has been retrieved for <SMS SERVER>.
20:05:08 >> Class Key retrieved for: SMSEXPERT|FLIP FLOP INFO|1.0 - 116.
20:05:08 >> Eliminating classes from WMI on <SMS SERVER>. Please wait...
20:05:08 >> Flip Flop Information has been removed.
20:05:09 >> Stopping SMS_Executive service on <SMS SERVER>.
20:05:51 >> SMS_Executive has stopped on <SMS SERVER>. Restarting
SMS_Executive.
20:05:51 >> Starting SMS_Executive service on <SMS SERVER>.
20:05:51 >> SMS_Executive has started on <SMS SERVER>.
20:05:51 >> Class removal has been successful!

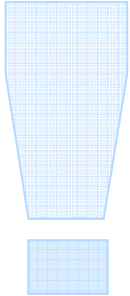
```

```

20:05:51 >> Retrieving Site Code for <SMS SERVER>.
20:05:51 >> Site Code <XXX> found for <SMS SERVER>.
20:05:51 >> Connection Established. Loading classes...
20:05:51 >> Class Query is complete.

```

There are a couple of important things to notice here in this short log. Look at all that Site Sweeper does for you:



1. Retrieves the site code for your site
2. Loads the classes from the site available to be deleted
3. Does a test run and displays what it is after
4. Deletes the class information
5. Deletes the class information from the site
6. Re-queries the WMI classes and displays the new list

Now that #4 line is a little under-described. Not only does Site Sweeper delete the class table information, it deletes *EVERYTHING* associated with it from your site—the tables, views, stored procedures, and even the WMI information about it from the site server itself.

Now you can see why I call this the easy way.



You may have noticed the class name change from “SMSEXPERT|FLIP FLOP INFORMATION|1.0” to “SMSEXPERT|FLIP FLOP INFORMATION|**MIF**” in the Site Sweeper screen shot. Do not adjust your set. I did this on purpose as dramatic foreshadowing for one of my tips in the next chapter.



Chapter Summary

To clean up your SMS Site database follow the steps here:

1. Clean up the SMS_def.mof on the site server
2. Remove the unnecessary WMI classes from your server and clients
3. Remove the class data from the database.

If you just want to stop collecting class data then just turn the class reporting qualifier from TRUE to FALSE and SMS will not inventory the class anymore.

Deleting a data class completely from your SMS_def.mof is not recommended. A better option is just to comment out the class like we talked about in Chapter 3. Just put a `/*` at the top of the section and a `*/` after the end of it in case you ever need it again.

If you're collecting inventory data for the class you're planning on deleting by using MIF files or scripts files instead of the SMS_def.mof, **ensure that you delete those files from the appropriate client directory and/or delete the advertisement causing a script to run that creates them!**

Do not go straight into SQL and delete database tables manually!

Tools for cleaning up your database include:

SMS 2003 Toolkit 2 tool DELGRP.EXE. This tool can be found at:
<http://www.microsoft.com/SMServer/downloads/2003/tools/toolkit.mspix>

And ...

SMS Expert's Site Sweeper utility (*recommended*):
<http://www.SMSexpert.com/products/sitesweeper.asp>

Chapter 14: Troubleshooting and Tips

Being defeated is often a temporary condition. Giving up is what makes it permanent.

–Marlene vos Savant

As with most things, sooner or later, something is going to break. Luckily the kind folks at Microsoft have created log files to track just about everything that SMS does. Finding the actual files involved in hardware inventory and the paths that the hardware inventory information travels on its journey to the database can be quite hard...OK... impossible to find on Microsoft's website.

Log Tag

To make your troubleshooting easier I've created the following diagrams to explain this process in as much detail as I could squeeze onto these pages. I've tried to make the diagrams short and sweet, but with enough information so that you can follow along with your own version of each system and log file with confidence.

Due to space constraints I've given you the shorthand version of some log entries, but left in enough for you to identify the lines I'm demonstrating as you follow along with your own log files.

Remember though, these files are from my lab SMS Site and there's no guarantee that your logs will match exactly. *This particular site is running SMS 2003 RTM with verbose logging turned on.*

I've broken down the hardware inventory diagrams into three distinct sections:

1. SMS_def.mof modification to new inventory policy in database
2. Machine Policy Retrieval & Evaluation Cycle
3. Client hardware inventory Data Path to Database

Hopefully you will find these diagrams helpful to you in your next game of “log tag” when something goes bad during client hardware inventory.

Figure 14.1

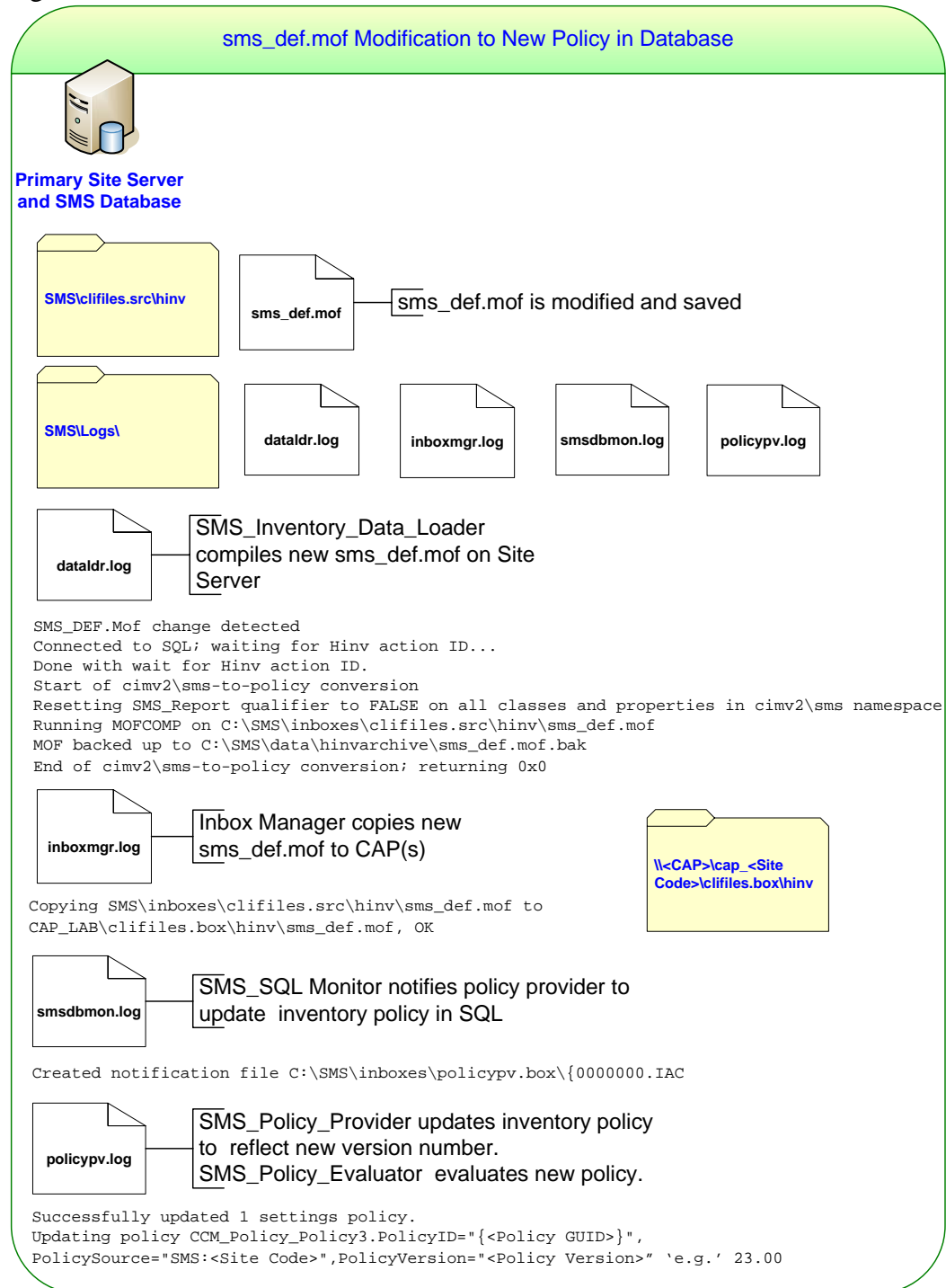


Figure 14.2

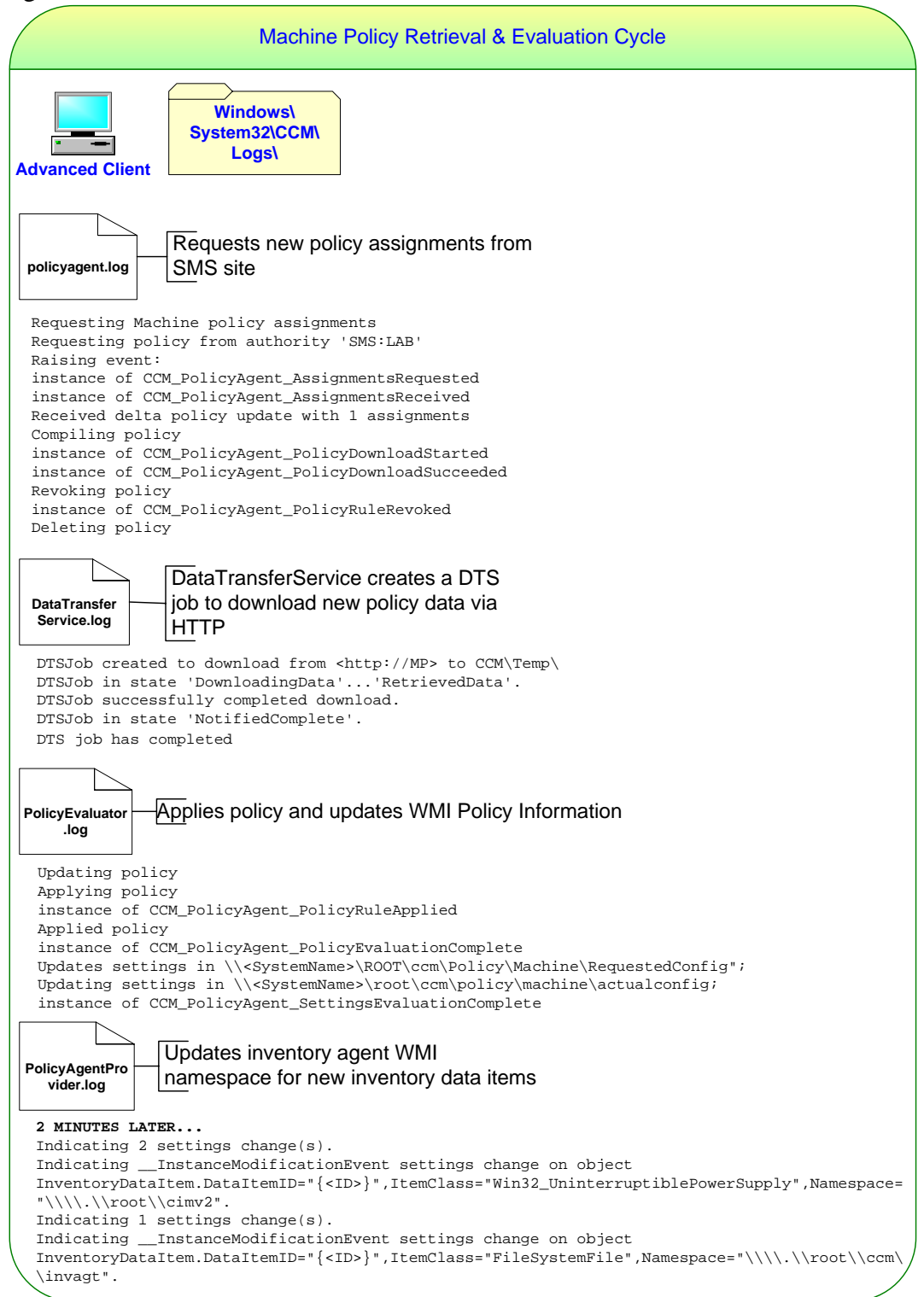


Figure 14.3

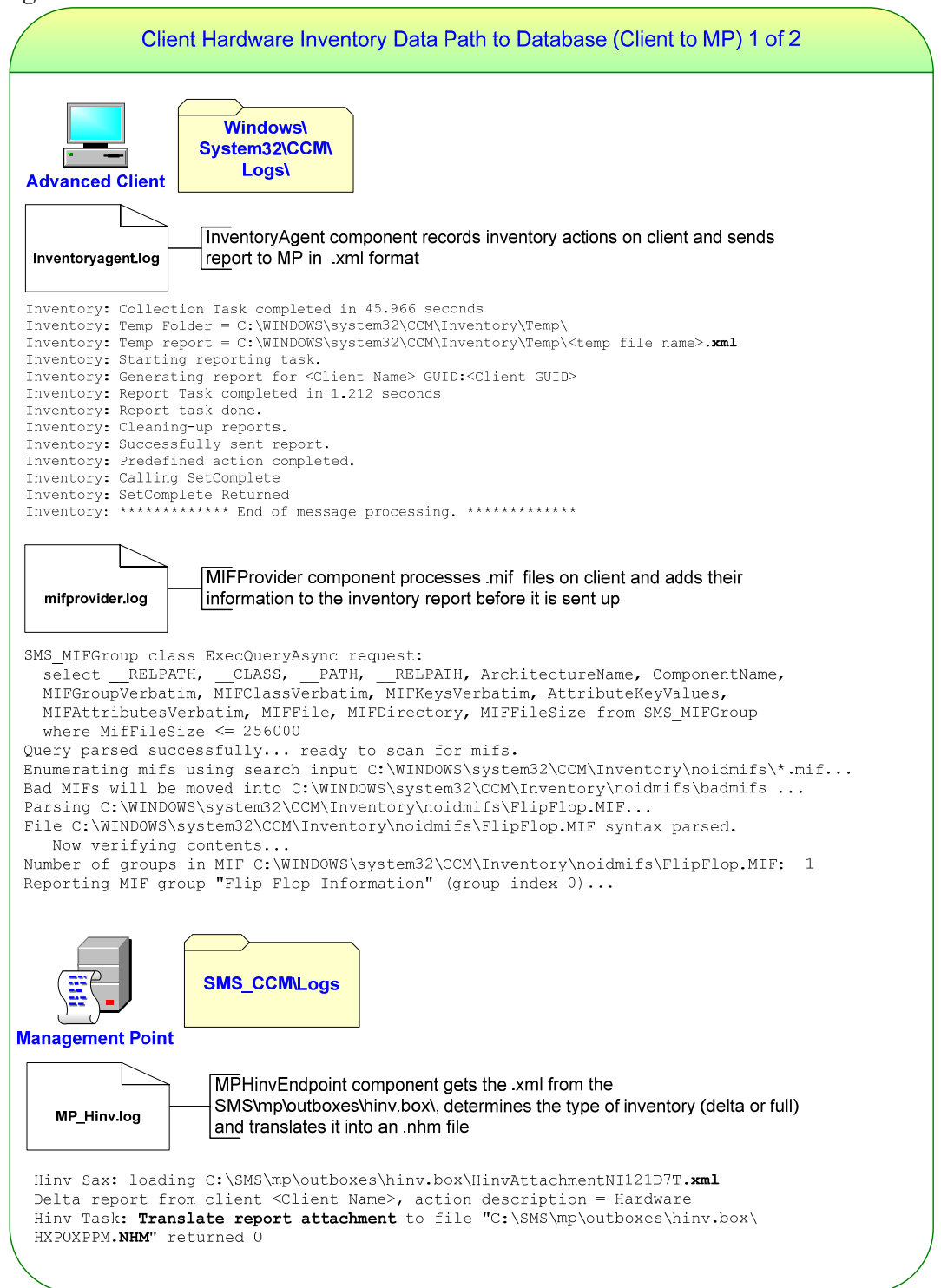
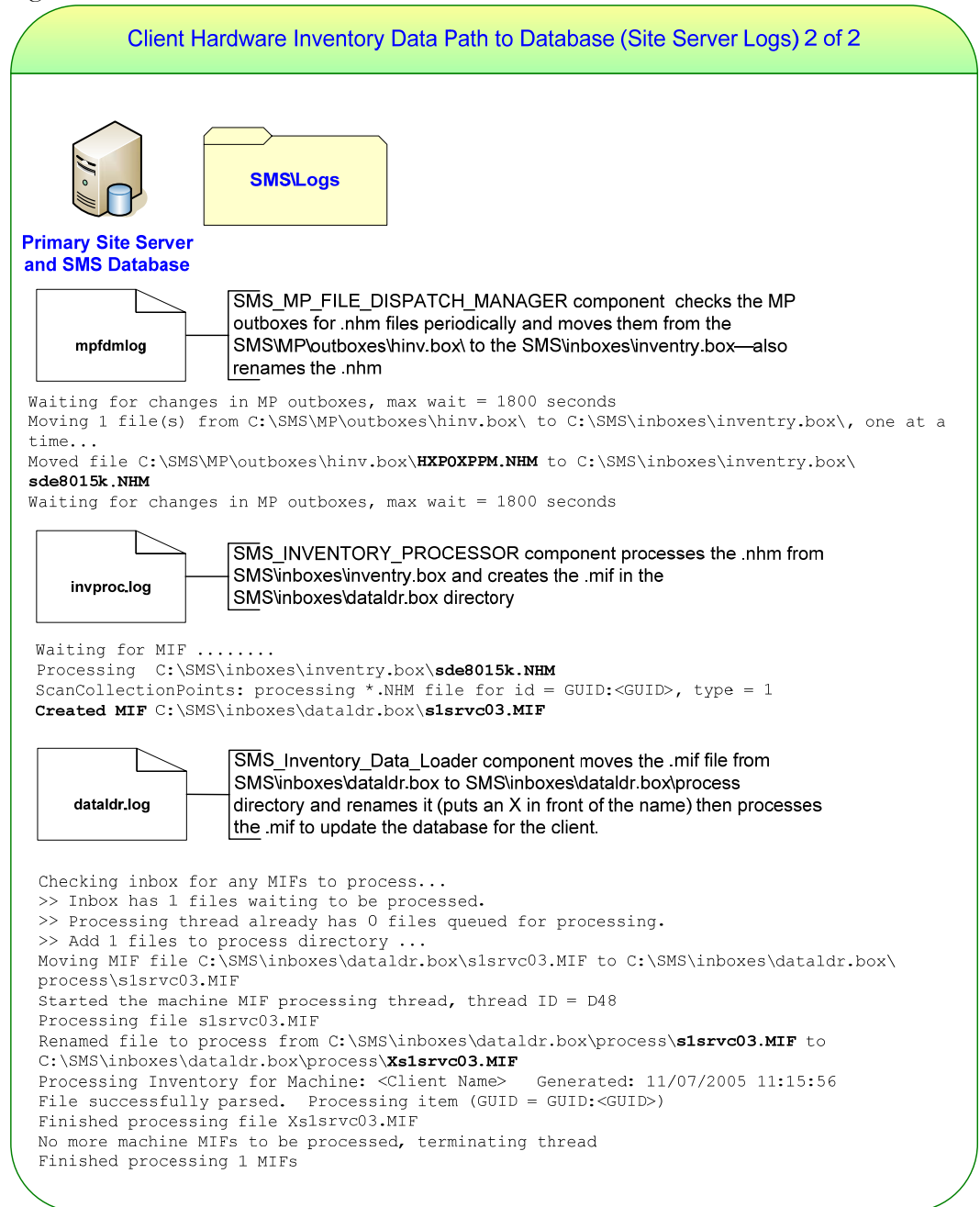


Figure 14.4



Hopefully, those diagrams will help you understand the path that the inventory information follows to get from your client machines into the SQL database. This, of course, assumes that you've done everything correct along the way to ensure valid data is moving along that path.

Next, I'll go over some of the more common problems afflicting those brave enough to wade the murky waters of MOF modification.

MOF Editing Errors

I copied and pasted stuff from someone else's MOF, but the data isn't showing up in resource explorer or my database!

The most common question asked is, "Why isn't it working?" Here are a few of the most common mistakes made when you copy someone else's MOF edits.

- If you are copying just a reporting class, verify that the last namespace change was to `root\CIMV2\SMS`. Remember that reporting classes must be in this namespace to work.
- If you are copying a data class and its respective reporting class, verify that the last namespace change before the data class was to `root\CIMV2`, and then switch to `root\CIMV2\SMS` prior to the reporting class.
- Remember that even if you copy a data class into your MOF, it will not report information unless a matching reporting class is created.
- Is the reporting class set to TRUE? Are the field reporting classes you want to see set to TRUE?
- SMS 2.0? Do you have the registry providers registered in your MOF if you're using them? Often administrators will copy in one registry provider and think they have both the Registry Instance Provider and Registry Property Provider in their MOF. When they copy in a class that's using the *other* provider, their class doesn't report.

If you think your MOF is correct, the problem may not be your MOF, but the SMS processes system themselves. If your MOF edits have passed MOFCOMP scrutiny, before changing your MOF when the new class isn't reporting, make sure you trace the steps of the hardware inventory.

I put my new MOF out on the site, and I have Windows 2000 machines hanging on me. Grrrr.

Did you add the Win32_QuickFixEngineering class to your MOF? If you did, this is most likely the culprit. This error was fixed by Windows 2000 SP3. Don't believe me? Check out the below KB article.

[KB279225 – WMI Win32_QuickFixEngineering Queries Hang Winmgmt Process.](#)

I've created a new class to pull data from Hkey_Current_User\Printers, but the data isn't appearing in my database. I can see the information with WBEMTEST. Where did it go?

Although you can see the data using WBEMTEST, data located in *HKey_Current_User* cannot currently be extracted directly by SMS hardware inventory. When you're looking at the data with WBEMTEST, you are viewing WMI using your own user account credentials. However, when hardware inventory is actually running, it is using the local SMS account, or local system account, to perform the task. This means that the system account is actually the current user, and therefore the HKCU registry information will either not exist, or will be different than it is for the regularly logged-on users.

The best way to obtain HKCU information is to use a script to copy the information from the *HKCU* hive into the *HKLM* hive in the registry. It then can be queried by hardware inventory.

I've changed a class in my MOF, but it continues to look the same in my database! Everything looks to be working 100% and the data is propagating, but it never changes no matter what I do to my class.

As silly as it may sound, it's possible that you have a class with the same name later in your MOF. Even if you're changing the class that's listed first in the MOF, the class listed at the very bottom will always overwrite the first. Usually this occurs from copying and pasting from other people's MOF examples.

I'm able to edit my MOF and I think your book is the best thing since sliced bread. However, I don't have the time to write up my own MOF edits. Where can I find other people's examples?

There are a few good places out there in cyberspace to pick up new MOF edits. SMS Expert's website (www.SMSExpert.com) is a good one. There, you can also pick up a copy of the monster MOF.

The Monster MOF will probably have 99.9% of the edits you are looking for already incorporated into one “Monster MOF” as well as scripts that can be used to further your inventory endeavors.

Another excellent source for all things SMS and for getting new MOF edits is myITforum (www.myITforum.com). On this website there are numerous articles, examples, and forum posts with new MOF edits and ideas. Just copy, paste, test, and deploy in your own environment.

MOFCOMP Errors

Once you believe that your MOF edit is ready for prime time, the next obvious thing to do is to check the syntax with MOFCOMP.

MOFCOMP will alert you to the majority of MOF editing mistakes and these errors are generally pretty intuitive, but sometimes you will receive a cryptic error response to your compilation attempts. Here, I’ll try to decipher a few of them that always seem to come up.

I’m getting the error that “An alias already exists” when I compile my MOF.

If you get this error, most likely it means that you’re trying to define a provider twice. A provider only needs to be defined once per MOF, and therefore if you happen to have it in your MOF twice, you’ll need to search for the duplicate and eliminate it.

I’m getting the error “Class has Instances” when I compile my MOF.

This is a common error when you use the Registry Property Provider. Unlike the Registry Instance Provider, the Property Provider defines its class first, and then defines a separate instance or instances.

When you compile the MOF again, and the compiler sees even the most minor change in the class, it will halt the compilation, stating that the class you’re trying to add already has instances. Because an instance is based upon the class, it doesn’t want to let you change the class while an instance exists. You can’t remove the foundation of a building when the building is still on top of it.

The easiest method to correct this error is to use the `#pragma deleteclass` function just prior to creating the class in your MOF. This will assure that no earlier class or instance will exist before you create the new class. See example below:

```
#pragma namespace("\\\\.\\.\\.\\.root\\CIMV2")
#pragma deleteclass("TravelMode", NOFAIL)
```

```
[DYNPROPS]
class TravelMode
```

```
{
    [key] string    KeyName="";
    uint32         TravelMode;
};
```

The `#pragma deleteclass` trick is a handy one. Whenever I'm testing a new MOF edit, I always include this line above my class lines. That way as I'm testing and modifying the edit I always know I'm starting with a clean slate. Once I begin observing the information I am after appear I take that line out. This has saved me a lot of time during my own experiences in MOF land.

SQL Database Errors

I've spent hours cleaning my database and my MOF, but for some reason these classes keep appearing in my database and I don't want them! Where are they coming from?

Do you have MIF files floating around your hierarchy. You must remember to delete those MIF files out of your client directories when deleting the class from SQL or you will end up chasing your tail and deleting the SQL information over and over again.

If you're getting classes in your database that you don't want, open the table and get the machine resource ID from that table. Locate that resource ID in your database and find out what machine it belongs to. You can then map to the hard drive of that machine and find out if the data is coming from the registry, the WMI, or from a MIF in the `noidmifs` directory.

I am getting custom inventory data in the history table for my new class, but the data table is only showing partial information! What gives?

This is a *key* question ... OK, it's a little late in the book for puns, but it really is a key question. Incorrectly defining the key fields for your class has been the culprit in my experience when this sort of situation occurs.

During the course of your reading this book it may have become apparent that I have a special relationship with the `Win32_PhysicalMemory` class. This is because when I decided it would be a good idea to inventory this information I didn't properly understand what key fields were ... and I paid dearly. It took me about two months to properly inventory the physical RAM chips in systems because of this!

While attempting to inventory the memory chips, I would consistently see only one individual memory chip in the inventory returned. That is, only one chip was represented in my current history view in resource explorer and the data table in SQL. Looking at the inventory history in resource explorer showed all four of the memory chips from my test system. These chips were also represented in the SQL history

table. Because I had declared the key fields for the class improperly in my MOF edit, the inventory information in my current data table was being overwritten by each successive memory chip found until I was left with just the last memory chip in the system being stored in the current inventory table. So when I looked in resource explorer for the current inventory information, I was only seeing one lonely memory chip where I should have seen four—like I did in the history table.

After declaring a key field in your MOF edit, the first time a client runs hardware inventory it will add those new tables, with their corresponding key fields, to the SQL database. Those keys are not easily updated or changed. The best way to do rectify this situation is to completely eradicate any information about those tables from SQL using DELGRP or Site Sweeper and start over again with a proper MOF edit.

I am using DELGRP to delete a table for a class from my database that I know exists, but I keep getting an error telling me that the class doesn't exist!

This is just DELGRP's way of telling you that there is a space or some other typographical mistake in your class name. If you add even one space to the name of your class it will say the class does not exist. Go back and check your command line and I'm willing to bet that there is something wrong with the way the class name was typed.

Troubleshooting WMI

Don't always just assume that WMI is corrupt, right off the bat! There are a few things to check before just pulling the plug on the system's WMI repository.

The Windows WMI service is always running. If you look in task manager for running processes you'll probably see one called winmgmt—unless you're running Windows XP or Server 2003. In this case, winmgmt is hiding within a svchost process.

Anyway, my point here is that you should always try stopping and restarting the winmgmt service before continuing on to any further troubleshooting. The winmgmt service is designed to start automatically when the computer system is started, so it should be running. If for some reason, the service has stopped then it should automatically restart the first time you run something that queries WMI such as WBEMTEST.

To manually stop and restart the winmgmt service, open a command prompt and type the following:

```
net stop winmgmt
net start winmgmt
```

Obviously, the net stop command stops the service and the net start command restarts it for you. If this doesn't help, try rebooting the system ... bet you've never heard that before!

Still having issues? Are you sure you didn't make a typo and try to inventory a non-existent namespace? Is this an XP machine with the firewall turned on? Could there possibly be DCOM permissions interfering with your inventory operations? DCOM can cause access denied errors when WMI fails to connect to a remote system. This is because WMI must establish a connection via DCOM to remote systems in order to query the remote system's WMI repository.

If the Windows firewall is enabled on your clients then some DCOM and firewall configurations may be necessary to allow this traffic. Rather than hurt my head here, I'll just link you to the smart guys:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/connecting_through_windows_firewall.asp

Of course, after all else fails, there will come a day that requires you to re-register all the WMI components or to rebuild the WMI repository.

There are a few ways to do this. It used to be that the only way to fix a corrupted WMI repository was to simply delete it and allow the system to rebuild it automatically for you. This could still work I suppose, but it's not very elegant. Below is a basic batch file that would do the trick and variants of this can be found in countless places around the Internet:

```
C:
cd %windir%\system32\wbem\

MOFcomp ci mwin32.MOF
MOFcomp ci mwin32.mfl

net stop wsmgmt
rmdir /s /q repository
rmdir /s /q Logs
mkdir Logs
net start wsmgmt
```

Nowadays, Windows XP and Server 2003 SP1 make it easy for you to tackle this task in a more refined manner. Built into these operating systems is a nifty set of commands that allow you to check the WMI repository for corruption and repair it all in one. Be warned, though, that you may lose some of those nifty WMI class additions this book has been prompting you to create.

I'll show you how to avoid this later in the chapter, but for now, here's the trick for XP and Server 2003 SP1—pay close attention, as these commands are case sensitive!

To check the WMI repository for errors on a Server 2003 SP1 system, run the following command from command prompt:

```
rundll32 wbemupgd, CheckWMI Setup
```

Once you've run that command, quickly run to the wbem logs location and peer into the setup.log file located at: `%windir%\System32\Wbem\Logs\Setup.log`. Now, just read through the log for any entries from today's date. If you find none, then WMI is OK and you should continue troubleshooting somewhere else. If, however, you see an error message from today saying that it can't find a namespace then, yes, your WMI repository is indeed lifeless.

To perform immediate CPR on your WMI repository run this command:

```
rundll32 wbemupgd, RepairWMI Setup
```

This will create a brand new, stock WMI repository full of everything that Windows came with—minus all those handy custom inventory classes you've spent the better part of your life adding to the client! Keep reading, like I said. I'll show you how to avoid that happening soon.

For Windows XP SP2, use the following command to check for corruption, and repair if necessary:

```
rundll32 wbemupgd, UpgradeRepository
```

For Windows XP SP1, the check and repair commands look like:

```
rundll32 wbemupgd, CheckWMI Setup
```

and ...

```
rundll32 wbemupgd, RepairWMI Setup
```


WMIDiag

It used to be common practice to just whack the WMI repository back to where it came from and recreate the entire repository when you thought something fishy was going on with a system's WMI repository. This was often done without really knowing what was going on or where the actual problem was. It was more of a cross your fingers and hope that was the problem kind of thing. Of course, if you had compiled any custom MOF edits you had to do it all over again to get the information back into WMI. This by itself is enough to make you want to continue reading...

The WMI Diagnosis Utility, WMIDiag, is a VBScript script written by Alain Lissior of the WMI team at Microsoft. The WMI Diagnosis Utility takes the guess work out of WMI troubleshooting. From checking all the WMI namespaces and services to looking for possible corruption, WMIDiag does it all. With detailed reports and even suggestions on how to fix the problems it encounters.

As of right now anyway, the tool can only be run locally by someone possessing local administrator rights. This really isn't such a bad limitation though because if you are troubleshooting possible WMI corruption on a system then a script or remote WMI connection probably wouldn't work anyway.

For more information about WMIDiag make sure to read the ReadMe.doc file included in the WMIDiag download located at the link below:

<http://www.microsoft.com/downloads/details.aspx?familyid=D7BA3CD6-18D1-4D05-B11E-4C64192AE97D&displaylang=en>

Some Tips and Other Silly MOF Tricks

There are a few tricks up any experienced MOF editor's sleeve. Most administrators have their own preferences for doing things and there are a lot of different ways to do the same thing when it comes to inventory techniques. I'll show you a couple of tips here that I feel are important and, hopefully, you will be able to benefit from them.

Create Custom Reminders

Changing the Class ID to help remember how you got the information into your database is one I like. For example, if I've used a NOIDMIF to inventory something, I'll do something like change the version number to MIF as in the example below:

SMSExpert | MyCoolClass | **1.0** changes to SMSExpert | MyCoolClass | **MIF**

Or, if I've used a script to populate the database I'll change the version number to:

SMSExpert | MyCoolClass | **VBS**

This helps when I decide I don't want to inventory that particular "cool class" anymore. I am given a quick reminder that I need to get those MIF files off the clients

to keep the class from returning from the dead or stop an advertisement running a recurring script on clients to populate the database.

Be #Pragmatic

OK, remember when I kept promising to tell you how to keep your custom class definitions from being deleted when you rebuild a client WMI repository? Well, this is how you do it.

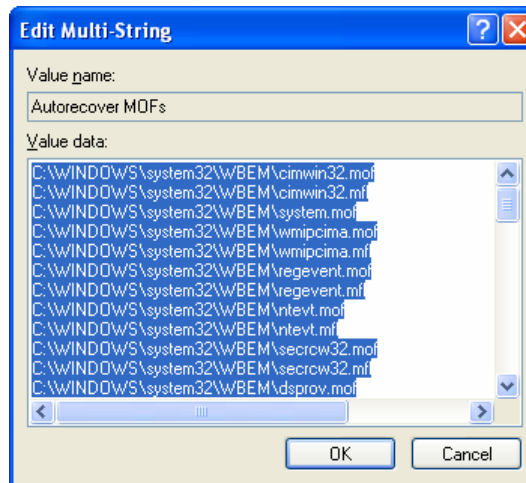
To avoid the problem of your custom WMI class additions being deleted when you rebuild the WMI repository on a system just add `-autorecover` to your MOFCOMP compilation command:

```
MOFcomp -autorecover SMSinstalls.MOF.
```

The `-autorecover` switch tells MOFCOMP to remember that it made this change to the local system WMI and to add it to its “to-do” list when rebuilding the repository on the system. The “to-do” list is stored in the registry at:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WBEM\CIMOM\Autorecover MOFs
```

Figure 14.5



One thing to take note of here: When you use the `-autorecover` switch, the .MOF file you are compiling must reside on the local machine somewhere. Somewhere that it is liable to still be present if the repository is ever rebuilt.

Most examples you see of a .MOF being compiled with MOFCOMP usually say `C:\Temp` or something similar. If you are going to use the `-autorecover` switch, you probably don’t want to store the file in a temporary directory or somewhere that a user is going to inadvertently delete it. Just so you know, the majority of the default MOF files set to autorecover are stored in the `%WINDIR%\System32\WBEM` folder.

Another fixation a lot of SMS admins have is attempting to keep their SMS_def.mof as small as possible while being able to quickly find their own MOF edits. This is easily done. Just use the `#pragma include` line at the bottom of your default MOF to include and any custom “mini-MOFs” you’ve created. Either one `#pragma include` for one “Monster MOF” or many `#pragma include` lines for many “mini MOF” files. I’ll show you what I mean.

Say you’ve created a custom data and reporting class for Symantec anti-virus information that you don’t want cluttering up your otherwise stock SMS_def.mof. Just create a “mini-MOF” including your custom AV inventory classes and name it Symantec.MOF for simplicity’s sake and store it in the same directory as your SMS_def.mof. Now, at the bottom of your SMS_def.mof add the following line:
`#pragma include (“Symantec.MOF”)`

Viola! When your systems run hardware inventory, they will populate your Symantec data tables just as if you had added those classes straight to the SMS_def.mof. Want to keep a separate MOF with just your personal inventory modifications? No problem, just use the `#pragma include` line at the bottom of the default SMS_def.mof to include your very own personal “Monster MOF”. This way, if you ever get tired of inventorying something you made, you can just comment out the include line in your SMS_def.mof and hardware inventory will not even blink.

By using this method with many “mini-MOF’s” and multiple `#pragma include` lines, you can easily get to the custom MOF modifications you have made, and just as easily stop collecting, or modify the collected information resulting from their use. I’ll talk about this again here in a minute, but I wanted you to understand the `#pragma include` line first.

For more `#pragma` tips from Microsoft visit:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/_pragma.asp

Bypass MOFCOMP

Bypass *WHAT* you’re thinking? That’s not possible! OK, you’re right, it’s not possible, but it can be accomplished easier than creating advertisements that run command lines to manually compile custom data classes on your Advanced Clients.

Ever look at the examples for applying custom data classes to Advanced Clients and see that you have to manually run `MOFCOMP MyNewClass.MOF` on all of them, and then look at the site server itself and see that all you need to do there is drop in a new SMS_def.mof and it’s automatically compiled? That didn’t seem fair to me, though, to be fair I spend way too much time pondering things like this.

My point here is this: you can just copy a custom MOF to your Advanced Clients and they will automatically compile the new MOF similarly to the old Legacy Clients!

Granted, you do have to copy them to a different location, but it's still possible. This may be a minor thing to some, but it excites me, so bear with me.

To automatically compile a custom MOF on your Advanced Client systems, just copy your customized MOF to their `%windir%\system32\wbem\MOF` directory. Give it a few seconds and the MOF file will disappear. If you've done everything correctly, the new MOF file will be found in the `%windir%\system32\wbem\MOF\good` directory. If for some reason, the new MOF did not compile correctly it will be found in the `%windir%\system32\wbem\MOF\bad` directory. Either way, you didn't have to run a complicated advertisement to run MOFCOMP manually.

Kind of a small victory over manual MOFCOMP compiling, but I think it's a cool trick nonetheless.

The Mini-Monster MOF

So I've kind of been hinting and lightly touching on a few things throughout this book that I feel I need to wrap up in pretty bow for you here. Alright, I'm not so good at wrapping presents—ask my wife who has to stay up until 4am every Christmas Eve while I'm fighting with a tricycle that I'm positive is missing three screws!

Anyway, this tip is how I like to make custom inventory modifications and hopefully this process will benefit you as well.

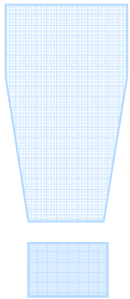
The first thing to do is to find all those modifications you've made to your current `SMS_def.mof` on the server and paste them into a new text file. You can actually comment out or delete all those original custom additions once we're finished here. You may also want to create a comment section at the top of your new file to describe any inventory changes you've made utilizing MIF files or scripts just to remind yourself of how you're getting that data. This will also enable you to maintain one authoritative document for all your inventory modifications

Once you have created your very own 'mini-monster mof' just save it with the "real" `SMS_def.mof` on the site server(s). Hopefully by this point we all know that this location is the `SMS\inboxes\cli\files.src\bin` share.

Since you're in that directory anyway, go ahead and open up the `SMS_def.mof` and add the `#pragma include` line to the bottom. For instance, if your mini-monster mof was named `MyMiniMof.MOF` then your *only* addition to the `SMS_def.mof` would be the below line:

```
#pragma include ("MyMiniMof.MOF")
```

Doesn't get much easier than that right? Sure, it's just as easy to paste those changes into the `SMS_def.mof`, but as you pass the days as an SMS admin there will come a day when it's time to upgrade or apply a service pack to your SMS site. When you do so,



guess what gets automatically zapped? That's right...the SMS_def.mof. If you have forgotten to back it up you will have to go through a new MOF and re-create all of your changes. When you upgrade your site using the mini-monster mof method all you have to do is go into the new MOF and add that one line...I think it's a handy trick anyway.

So now that the SMS_def.mof is bringing your custom additions along for the ride it's time to notify your clients of the new changes. Create a program that copies your mini-monster.mof to the location that the system's reserve for important MOF files--%WINDIR%\System32\WBEM. Next, have the program run the below command to get the customizations entered into you client's WMI repository and safeguard them against deleted WMI repositories:

```
MOFOMP -antorecover MyMiniMof.mof
```

SMS programs don't run without advertisements right? Create an advertisement for your program to run on a recurring basis. The recurring program helps to keep clients updated when you change your custom additions as well as update any new systems that become SMS clients in the future automatically.

Hopefully you'll find this little tip as handy as I do, but if you don't it's fine to use whatever method you find the most palatable for your situation.

More Tips and Tricks

For more cool tips and tricks, just visit the following websites. They will fill you in on a lot of important information, and help you succeed in your endeavors to become the best SMS admin out there.

SMS Expert Website

www.SMSExpert.com

Microsoft newsgroups

SMS Inventory Newsgroup at:

<http://www.microsoft.com/technet/community/newsgroups/dgbrowser/en-us/default.aspx?dg=microsoft.public.SMS.inventory>

SMS Admin Newsgroup at:

<http://www.microsoft.com/technet/community/newsgroups/dgbrowser/en-us/default.aspx?dg=microsoft.public.SMS.admin>

SMS Hardware Inventory FAQ

<http://www.microsoft.com/technet/prodtechnol/SMS/SMS2003/techfaq/tfaq05.mspx>

myITforum.com and the

www.myitforum.com

myITforum.com SMS email list

<http://lists.listleague.com/mailman/listinfo/mssms/>

If you aren't already subscribed to the SMS email list on myITforum then take a minute before you finish the book and *go subscribe!* You will be glad you did—trust me.



Chapter Summary

Common mistakes made when you copy and paste from someone else's MOF:

- If you are **copying just a reporting class**, verify that the last namespace change was to `root\CIMV2\SMS`. Remember that reporting classes must be in this namespace to work.
- If you are **copying a data class and its respective reporting class**, verify that the last namespace change before the data class was to `root\CIMV2`, and then switch to `root\CIMV2\SMS` prior to the reporting class.
- Remember that even if you copy a data class into your MOF, it will not report information unless a matching reporting class is created.
- Is the reporting class set to TRUE? Are the fields you want to see set to TRUE?
- **SMS 2.0?** Do you have the registry providers registered in your MOF if you're using them? Often administrators will copy in one registry provider and think they have both the Registry Instance Provider and Registry Property Provider in their MOF. When they copy in a class that's using the other provider, their class doesn't report.

Troubleshooting WMI

- Stop and restart the Windows Management Instrumentation (winmgmt) service
- Check XP firewall or DCOM permissions
- With XP or Server 2003, you can try the `wbemupgd` commands to repair or reinstall WMI
- Go download WMIDiag!

When all else fails:

- Stop the WMI service
- Rename or delete the old WMI repository
- Restart the service

Changing the Class ID line to reflect the way that the data is collected (MIF, VBS, etc.) may help remind you to delete mif files from clients or stop inventory scripts from running to recreate the data tables you are going to delete.

To avoid custom WMI class additions being deleted when you rebuild the WMI repository on a system just add *-autorecover* to your MOFCOMP compilation command.

Using the *#pragma deleteclass* command ensures that you are starting off with a clean WMI slate each time you test your MOF edit. Once you are confident that you have what you need, just take that line out.

Adding a *#pragma include* line at the bottom of the default SMS_def.mof will cause hardware inventory to include those additional MOF files as if they were a part of the SMS_def.mof file itself.

Chapter 15: MOF Editing in 15 Minutes

To measure up to all that is demanded of him, a man must overestimate his capacities.

—Johann Wolfgang von Goethe

This is the chapter you've all been waiting on! Yeah right, I know you skipped the entire book to get to this chapter. Either way, this chapter is basically a slimmed down version of the entire book for you fast trackers out there. If you really did read the rest of the book before getting here, hopefully this chapter will help some of the concepts and processes I've previously discussed sink in.

My goal is to give you a basic familiarity with the SMS_def.mof and the processes involved in modifying hardware inventory for custom information. I'll highlight the important parts for you here, and direct you to more detailed information within the book for more information where appropriate.

Introducing the SMS_def.mof

Managed Object Format (MOF) files are text files. The SMS_def.mof file is, therefore, a text file. Of all the text files on your SMS server I'm going to go out on a limb and say it's probably the most important—at least as far as hardware inventory is concerned anyway.

For Advanced Clients, the data *inside* this file is what is used by the SMS site server to generate a hardware inventory *policy* which the clients receive from their management point. Legacy Clients retrieve the actual SMS_def.mof itself from their client access point and use it directly to perform hardware inventories. From there, the hardware inventory process queries WMI for the data to collect based on changes made by the SMS_def.mof, and then queries a different namespace within WMI again to actually collect that data. I'll get into WMI in a minute, but that's the general idea at this point.

- Legacy Clients *store a local copy* of the SMS_def.mof file in their `%WTNDR%\ms\SMS\clifiles\bin` share

- Advanced Clients *retrieve an inventory policy* based on the SMS_def.mof stored on the site server in the `SMS\inboxes\clifiles.src\binw` share.

Because the SMS_def.mof is really just a text file, you can open it with NOTEPAD.EXE and view the data contained within it. WMI namespaces and providers are declared at the top in the declarations section. Beneath that section the actual inventory data and reporting classes are listed. These data and reporting classes are modified or created in the local system WMI and used to perform the actual hardware inventories.

So, from here you can see that the real meat and potatoes of the SMS_def.mof consists of three important pieces: The *providers* (how to go about obtaining the information from the system), the *data classes* (where to get the information you're after), and the *reporting classes* (the information to be collected during hardware inventory).



The SMS_def.mof is described in more detail in Chapter 2.

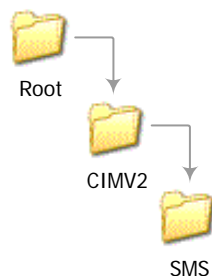
Windows Management Instrumentation

Understanding Windows Management Instrumentation (WMI) is *critical* to understanding hardware inventory and MOF modification. WMI is the Windows implementation of the Web-Based Enterprise Management initiative or WBEM. WMI is used to gain access to data stored in the Common Information Model Version 2 (CIMV2) repository on a system. You will see this commonly referred to as the WMI repository pretty much everywhere you look. For the most part, the data stored within this repository is what we're after when making hardware inventory modifications.

The WMI repository is organized into namespaces (classes and instances) that contain more namespaces (subclasses and instances). Think of the WMI repository as a file cabinet, or a computer's file and folder structure. There are root folders which, in turn, contain subfolders full of significant information.

The main namespaces you will use in MOF modification within the repository are the `root\CIMV2` and `root\CIMV2\SMS` namespaces as shown in Figure 15.1.

Figure 15.1

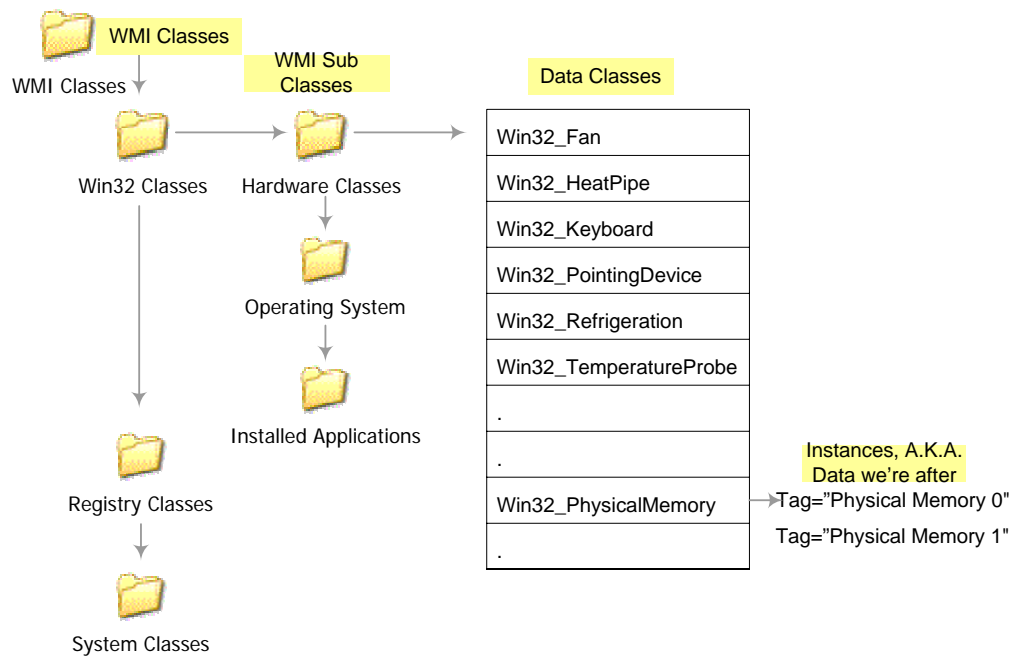


Think of the root as ... well the root, the CIMV2 as the CIMV2 repository containing all the significant information about the system, or *data classes*, and the SMS folder as containing everything you want SMS to report on, or *reporting classes*.

Hardware inventory reads the data classes you've named in the SMS_def.mof from the *root\CIMV2* namespace and adds entries into the *root\CIMV2\SMS* namespace for reporting purposes.

That's a pretty limited view of the CIMV2 repository, but those are the main namespaces you will be dealing with during your MOF modifications. Sometimes the data classes you want to inventory may be in a namespace not included as a root class within CIMV2, but there are other sub classes that can be considered for inventorying as well:

Figure 15.2



To access WMI programmatically and see what SMS hardware inventory sees, you can use tools such as WBEMTEST (included with Windows) and the Microsoft Scripting Guy's Scriptomatic v2.



For more information about WMI and more tools to access it, go back to Chapter 4

Hardware Inventory Process

The hardware inventory process is relatively easy to understand once you have the core concepts of the SMS_def.mof and WMI down. Just remember that compiling the modified SMS_def.mof causes the local system WMI of your clients to change. Inventory data you are after is copied from the data classes to the reporting classes and

sent in a report back up to the site server via the management point for Advanced Clients and client access point for Legacy Clients.

Modifying the SMS_def.mof, located on the site server within the *SMS\inboxes\clifiles.srv\binv* share sets off a chain reaction. Depending on the client type, different events occur.

When you modify the SMS_def.mof the site server compiles it automatically. This new data is then used to generate a hardware inventory policy which Advanced Clients download from their management point—once an hour by default. Upon receiving the new inventory policy, the Advanced Client evaluates it, and two minutes later applies it. Applying the policy, in this case, means making changes to the local WMI repository for newly added or deleted inventory classes and information.



To adjust the time between policy downloads, adjust the polling interval on the Advertised Programs Client Agent Properties tab located in the SMS admin console under *Site Settings\Client Agents\Advertised Programs Client Agent*.

For Legacy Clients, the site server simply copies the newly edited SMS_def.mof to the client access points. The Legacy Client takes note of when the SMS_def.mof is modified and downloads it locally to the *%WINDIR%\ms\SMS\clifiles\binv* folder. Once there, it is automatically compiled and the new data entered into the local system WMI repository.



For more information about the hardware inventory process, see Chapters 1 and 14.

Modifying the MOF

The following are the six methods we will be covering here quickly:

1. Reporting on an existing class
2. Inventorying registry keys using the Registry **Property** Provider
3. Inventorying registry keys using the Registry **Instance** Provider
4. Inventorying data using the **View** Provider
5. Static hardware inventory extensions.
6. Scripted hardware inventory extensions

Reporting on an Existing Data Class

Reporting on an existing data class is the easiest MOF modification there is. You just simply need to add in a new reporting class to your existing SMS_def.mof file, preferably at the end of it, so you can easily find it later!

You don't have to spend a lot of time trying to memorize the reporting class syntax, just find a reporting class already existing in the *root\CIMV2\SMS* namespace within your MOF and copy and paste, and modify it to suit your needs.

Let's say I want to inventory *MyNewClass*. Just copy and paste an existing reporting class and change the class information to match *MyNewClass*. This includes the Group Name, Class ID, Class name, and reporting field types and names.

Here's what this would look like step by step. Below is a reporting class already existing in the SMS_def.mof. I've shortened it up for readability and highlighted the areas you need to change in red:

```
#pragma namespace ("\\\\.\\root\\CIMV2\\SMS")

[ SMS_Report (FALSE),
  SMS_Group_Name ("1394 Controller"),
  SMS_Class_ID ("MICROSOFT|1394_CONTROLLER|1.0") ]

class Win32_1394Controller : SMS_Class_Template
{
  [SMS_Report (FALSE) ]
  uint16 Availability;
  [SMS_Report (FALSE) ]
  string Caption;
;
```

Now, just change the parts listed in red to reflect the new reporting class for *MyNewClass* and set it to be inventoried by changing the SMS_Report lines from FALSE to TRUE.

```
#pragma namespace ("\\\\.\\root\\CIMV2\\SMS")

[ SMS_Report (TRUE),
  SMS_Group_Name ("MyNewClass"),
  SMS_Class_ID ("MICROSOFT|MyNewClass|1.0") ]

class MyNewClass : SMS_Class_Template
{
  [SMS_Report (TRUE) ]
  string Field1;
```

```
[SMS_Report (TRUE) ]
    string Field2;
;
```

Because the *MyNewClass* is an existing data class already defined in WMI, all you had to do was define a new reporting class under the *root\CIMV2\SMS* namespace in the *SMS_def.mof* to tell the hardware inventory to collect information about it.

Too easy, right? Let's move on to a harder one.



For more information about reporting on an existing data class see Chapter 5.

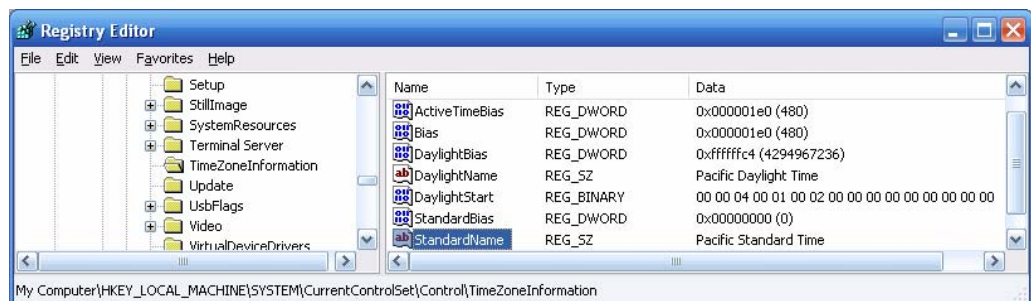
Registry Providers

The registry providers come in two flavors—the **Registry Property Provider (RPP)** and the **Registry Instance Provider (RIP)**. Sure, they sound similar, but they are really quite different.

It takes most people a while for these differences to sink in. The best way I can think of to differentiate them for you is this: the registry property provider basically reads individual registry keys that you ask it to, and the registry instance provider goes after sub keys of a specific registry key you've identified as the key field in your MOF edit. In other words, it finds instances somewhere under a specific registry key that match certain criteria. OK, clear as mud, right? I'll give you a quick demonstration here.

If you wanted to get information about the time zone standard name, you could query *the specific registry key* using the **Registry Property Provider (RPP)**:

Figure 15.3



Your resulting MOF edit would look like so, with both the data and reporting classes:

```

//-----
//Start of Time Zone Information
//-----

#pragma namespace("\\\\.\\root\\CIMV2")

[DYNPROPS]
class TimeZoneInfo
{
    [KEY] string KeyName = "";
    string StandardName;
};

[DYNPROPS]
instance of TimeZoneInfo
{
    KeyName="TimeZoneInformation";

[PropertyContext("local|HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentCont
rolSet\\Control\\TimeZoneInformation|StandardName"),
    Dynamic,Provider("RegPropProv")] StandardName;
};

#pragma namespace("\\\\.\\ROOT\\CIMV2\\SMS")

[SMS_Report(TRUE),SMS_Group_Name("TimeZoneInfo"),SMS_Class_ID("MI
CROSOFT|TimeZoneInfo|1.0")]
class TimeZoneInfo : SMS_Class_Template
{
    [SMS_Report(TRUE),KEY] string KeyName;
    [SMS_Report(TRUE)] string StandardName;
};
//-----
//End of Time Zone Information
//-----

```

Notice here that we're using the RPP defined as *RegPropProv* in the MOF edit to find the data stored in the specific registry key *StandardName*. **It's as basic as writing out the registry path and placing a | right before the exact key you're after.**

HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Control\\TimeZoneInformation|StandardName

The RPP doesn't care that there are other sub keys under *TimeZoneInformation* like *ActiveTimeBias*, *Bias*, or *DaylightBias*. It's only going to query the specific key after the | symbol and so will only see *StandardName*.

Just remember this—the Registry Property Provider goes after *a specific property* of a registry key.

The **Registry Instance Provider (RIP)** goes after registry information in a slightly different manner. Once you give it the registry key to look under, it will query all the *instances* under there and report back to you—at least the ones you’ve told it to anyway. For example, below is what the TimeZoneInfo MOF edit would look like using the RIP:

```
//-----
//Start of Time Zone Information
//-----

#pragma namespace("\\\\.\\root\\CIMV2")

[dynamic, provider("RegProv"),
ClassContext("local|HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Control")
]

class TimeZoneInfo
{
    [KEY] string TimeZoneInformation;
    [PropertyContext("StandardName")] string StandardName;
    [PropertyContext("DaylightName")] string DaylightName;
};

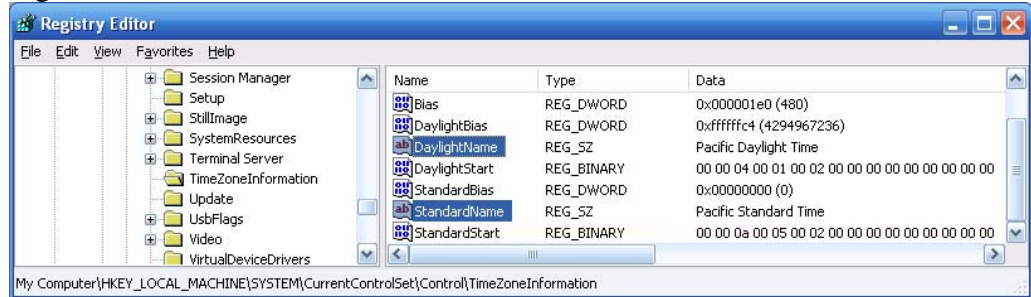
#pragma namespace("\\\\.\\ROOT\\CIMV2\\SMS")

[SMS_Report(TRUE),
SMS_Group_Name("TimeZoneInfo"),
SMS_Class_ID("MICROSOFT|TimeZoneInfo|1.0")]
class TimeZoneInfo : SMS_Class_Template
{
    [SMS_Report(TRUE),key] string TimeZoneInformation;
    [SMS_Report(TRUE)] string StandardName;
    [SMS_Report(TRUE)] string DaylightName;
};
```

Notice that in this case we’re using the RegProv provider. This is how the RIP is defined in the top section of the default SMS 2003 SMS_def.mof. Also, notice that in this example, the *key* field is the actual registry key (in the left pane of regedit) which has a sub key that contains *the instances* (or reporting field names) of data we’re after. In this case, we’re just looking for StandardName and DaylightName keys (in the right pane of regedit).

By the way, I could have named the key “Stanley” and it still would have returned all the same information. The key field in this case is the name of the parent key to the keys found with the StandardName and DaylightName value information somewhere under the registry path given on the ClassContext line.

Figure 15.4



For more information about the RPP and RIP check out Chapters 6 and 7.

The View Provider

The basic purpose of the view provider is to mirror information from one class namespace to another. This was a super important provider when using SMS 2.0 because SMS 2.0 clients could not inventory namespaces other than *root\CIMV2* and *root\CIMV2\SMS*. So if you wanted to inventory Microsoft Internet Explorer version information you would have to use a view provider to copy the information from the *root\CIMV2\applications\microsoftIE* namespace into the *root\CIMV2* and *root\CIMV2\SMS* namespaces.

To do so, you would have to peer into the *MicrosoftIE_Summary* namespace using the view provider:

```
#pragma namespace("\\\\.\\root\CIMV2")
[Union,
ViewSources{"select * from MicrosoftIE_Summary"},
ViewSpaces{"\\.\\root\CIMV2\applications\MicrosoftIE"}, Dynamic
:
ToInstance, provider("MS_VIEW_INSTANCE_PROVIDER")]
class MicrosoftIE_Summary

{
[PropertySources{"Build"}] string Build;
[PropertySources{"IEAKInstall"}] string IEAKInstall;
[PropertySources{"CipherStrength"}] uint32 CipherStrength;
[PropertySources{"Version"}] string Version;
[PropertySources{"Name"},Key] string Name;
};
```

I've highlighted some important parts above in red. Notice how the view provider is called—it is pre-defined in the SMS 2003 SMS_def.mof so we don't have to define it here. Also, notice the **select *** statement in there? The view provider actually allows you to perform some basic WQL filtering if you wanted to only inventory certain parts of namespaces.

The reporting class looks just like any other reporting class in the SMS_def.mof:

```
#pragma namespace("\\\\.\\root\\CIMV2\\SMS")

[SMS_Report(TRUE), SMS_Group_Name("MSFTIE_Summary"),
SMS_Class_ID("SMSExpert|MSFTIE_Summary|1.0")]
class MicrosoftIE_Summary : SMS_Class_Template
{
[SMS_Report(TRUE)] string Build;
[SMS_Report(TRUE)] string IEAKInstall;
[SMS_Report(TRUE)] uint32 CipherStrength;
[SMS_Report(TRUE)] string Version;
[SMS_Report(TRUE),Key] string Name;
};
```

Luckily, SMS 2003 allows you to simply use a namespace qualifier to “point” to the other namespaces instead. To inventory this same information with SMS 2003 all you have to do is use a namespace qualifier in a regular reporting class to clue SMS in on the fact that the data class is really somewhere besides *root\CIMV2*:

```
[ SMS_Report (TRUE), SMS_Group_Name ("MSFTIE_Summary"),
SMS_Class_ID ("SMSExpert|MSFTIE_SUMMARY|1.0"),
Namespace("\\.\\root\\CIMV2\\Applications\\MicrosoftIE")]
class MSFTIE_Summary : SMS_Class_Template
{
[SMS_Report (TRUE)] string Build;
[SMS_Report (TRUE)] string IEAKInstall;
[SMS_Report (TRUE)] uint32 CipherStrength;
[SMS_Report (TRUE)] string Version;
[SMS_Report (TRUE),Key] string Name;
};
```



For more information about view provider see Chapter 8.

Static Hardware Inventory Extensions

These extensions come in a few flavors. There is the basic static MOF, which is basically a “mini-MOF” that contains both the class and instance information of that class. This file must be compiled using MOFCOMP to get the information into the

local system WMI and the reporting class added to the main SMS_def.mof to get the information added to the database. Every time you want to update that information you will need to manually edit that static MOF file and re-compile it using MOFCOMP. A quick example is shown below:

```
#pragma namespace ("\\\\.\\root\\CIMV2")
class Static_MOF
{
    [key]
    string user;
    string office;
    string phone_number;
};
instance of Static_MOF
{
    user = "John Smith";
    office = "Building 4, Room 26";
    phone_number = "(425) 707-9791";
};
instance of Static_MOF
{
    user = "Denise Smith";
    office = "Building 4, Room 26";
    phone_number = "(425) 707-9790";
};
```

Notice the class (Static_MOF) is declared at the top and the instances of the class are listed afterwards. When compiled, this would add both the class and instance information into WMI. Just add the following reporting class to your SMS_def.mof and you're all done.

```
#pragma namespace ("\\\\.\\root\\CIMV2\\SMS")
[ SMS_Report (TRUE),
  SMS_Group_Name ("Static AssetInfo MOF"),
  SMS_Class_ID ("MICROSOFT|Static_MOF|1.0")]
class Static_MOF : SMS_Class_Template
{
    [SMS_Report(TRUE), key]
    string user;
    [SMS_Report(TRUE)]
    string office;
    [SMS_Report(TRUE)]
    string phone_number;
};
```

Just remember, if any of this information ever changed, you would have to manually edit that MOF file and recompile it locally to update the system information.

NOIDMIF and **IDMIF** files are another type of static inventory extension that need to be updated manually or via a script. These follow a completely different format than regular MOF files and are processed right along with the normal hardware inventory process. To use these MIF files, you must enable NOIDMIF and IDMIF collection from the hardware inventory properties in the SMS admin console. The default size for MIF files is 250 KB, but you can enable larger files to be collected by changing the value in the admin console.

Even though the hardware inventory properties say that you are enabling NOIDMIF or IDMIF *collection*, SMS doesn't really collect these files. They stored in specific locations on clients and their data is read, and appended to, the normal hardware inventory report sent up by the hardware inventory agents.

These are a little too complicated for me to throw into a chapter that is supposed to take 15 minutes, but I'll give you a quick explanation of what they are here.

The main difference between NOIDMIF files and IDMIF files is that NOIDMIF files are information relating to an existing client system—they have an ID, so none is needed, thus the name **NOIDMIF**. IDMIF files contain data for non-client systems or for objects that cannot become an SMS client. These files can pertain to stand-alone systems for which you want some kind of inventory data, or other things, like printers, people, etc.

Because the data is *not* associated with a known entity, the IDMIF must contain special comments that provide a unique identifier to tell SMS exactly what this information is referring to. That is what puts the ID in **IDMIF**.



For more information about static MOF extensions, NOIDMIF, and IDMIF files, see Chapter 9.

Scripted Hardware Inventory Extensions

Sometimes there just isn't an easy way to get the inventory information you are after. This is where scripts come in. Scripts can be utilized to get the data that normal hardware inventory cannot, and report the results in a format that SMS can recognize and report on.

For example, say you want the mapped drive information for users. Since SMS hardware inventory does not run under the user context, you won't ever see the actual drives that a user has mapped. In this situation, a script that runs when a user is logged

on, under their credentials, and documents the results into a .MIF, static .MOF file, or even straight to the client system's WMI would be the way to go.

Bear in mind, scripted hardware inventory information is only as good as the last time you ran the script! You would need to set the script to run on a recurring basis to maintain current data.



For more information about scripted inventory extensions, see Chapter 10.

Comments

Before I move on here, there's something I've been meaning to tell you. Notice the comments in these MOF edit examples? Comments are your friends and should never be left out of any good MOF edit. Using comments to remind yourself what you were doing six months ago when you started inventorying something off the wall may be helpful to you—or the SMS administrator who takes over after you move on to bigger and better things somewhere else.

Comments can help remind you just how you got the data you're reporting on as well. It's always good to remind yourself that you are getting some information via NOIDMIF files or recurring script executions for instance.

To add comments to your MOF file, just place two /'s like so: // in front of your comments on a single line:

```
// Hi, I'm a comment.
```

Or, to comment out an entire block of code versus deleting it and wanting it back later just place /* in front of the beginning of the comment section and follow it with */ at the end:

```
/* Hi,  
I am a  
comment */
```

There are three rules that I follow when creating comments:

1. At the end of the stock SMS_def.mof file, create a large block of comments to signify the end of the Microsoft section, and the beginning of your own. Inside this large block of comments, make a summary of all the classes you added, how you added them (.VBS, .MIF, etc...), why you did this, and the date they were added.
2. Before each new class is defined, include a block of comments stating the name of the class, what data it is to retrieve, any relevant KB articles, and any modifications to the class with their respective dates.

3. Create a comment before any line that has been modified or altered from the original, or if you want to remember why a line does, or does not, exist.

Although it sounds like a tremendous amount of work, when you go back to your MOF two months after making modifications, you'll be very thankful that you made the effort.

Compiling the MOF

Once you've modified the `SMS_def.mof`, the next logical step is for you to get those changes into your client system's WMI repositories. To do this you must compile the MOF.

Whether the compilation occurs on the site server to generate an Advanced Client policy or locally for Legacy Clients, a program called MOFCOMP (Managed Object Format (MOF) compiler

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/mofcomp.asp>) is used. MOFCOMP can edit, delete, and insert classes in the local WMI repository based upon what is defined in the MOF that has been compiled. This occurs automatically whenever a new `SMS_def.mof` is introduced at the site server level or on the Legacy Client.

MOFCOMP can also be run manually. To run MOFCOMP manually, just open a command prompt and type in `MOFCOMP` and press enter. Using a `.MOF` file as a parameter, the program will read through the `.MOF`, line by line, top to bottom, and perform whatever action is specified. MOFCOMP simply reads the file and follows the instructions.

If the compiler runs into an error or a class that is not properly structured, it will cease the compilation process and display an error message with the line on which the error occurred. Compilation will stop at that point with whatever changes are listed above the error being made to the local WMI repository, and those below the error in the file being ignored as the compilation will have ceased upon recognizing the error in the file.

Before posting a new `SMS_def.mof` to your site server, it is a good idea to check it for syntax errors with MOFCOMP. To do so, use the command with a `-check` parameter as so: `MOFcomp -check SMS_def.mof`.

If there are any syntax errors in the MOF, then MOFCOMP will alert you to them at this point, with no harm done to any actual WMI classes. Once your MOF is free of syntax errors, you can drop the `-check` and use the command line `MOFCOMP SMS_def.mof` to actually modify the WMI classes.

When hardware inventory runs, MOFCOMP runs just as if you had manually requested it to run. The logs for the hardware inventory, including status messages about the success or failure of the MOF compilation can be found below:

- For the SMS 2.0 client: `%windir%\ms\SMS\logs\him.log`
- For the SMS 2003 Advance client: `%windir%\system32\ccm\logs\inventoryagent.log`



For more information about MOFCOMP and compiling MOF files, see Chapters 3 and 11.

Distributing MOF Updates

Distributing MOF updates is a pretty simple process. Understanding *why* you'd need to distribute MOF updates may be a little hard to understand at first though. Remember that Legacy Clients actually download the SMS_def.mof whenever it is changed via their client access point, and Advanced Clients receive a new inventory policy from their management point, based upon the contents of the SMS_def.mof. Why then do we need to distribute MOF updates?

Whenever a new *data class* is created that SMS doesn't recognize by default, *the Advanced Client has to manually compile a modified MOF file to create the necessary class information*. Since Legacy Clients compile the modified SMS_def.mof file automatically, they get away with no further action necessary.

The easiest way to accommodate this for Advanced Clients is through recurring advertisements created by standard SMS packages and programs to compile the modified inventory information. Just create a package and program to run the manual compilation command—`MOFcomp yourMOF.MOF` and schedule it to run on a recurring basis to adjust the WMI accordingly for new client systems or updated MOF files.



Chapter Summary

Managed Object Format (MOF) files are text files. The SMS_def.mof file is, therefore, a text file, and **the best method to access it is to use NOTEPAD.EXE.**

For Advanced Clients, **the data inside this file is what is used by the SMS site server to generate a hardware inventory policy** which the clients receive from their management point. **Legacy Clients retrieve the actual SMS_def.mof itself** from their client access point and use it directly to perform hardware inventories.

WMI namespaces and providers are declared at the top of the SMS_def.mof in the declarations section. Beneath that section the actual inventory data and reporting classes are listed. These data and reporting classes are modified or created in the local system WMI by the compilation process and are used to perform the actual hardware inventories.

WMI is the Microsoft implementation of the Web-Based Enterprise Management initiative or WBEM. **WMI is used to gain access to data stored in the Common Information Model Version 2 (CIMV2) repository on a system.**

The WMI repository is organized into namespaces (classes and instances) that contain more namespaces (subclasses and instances).

The main namespaces you will use in MOF modification within the repository are the root\CIMV2 and root\CIMV2\SMS namespaces. Hardware inventory reads the data classes you've named in the SMS_def.mof from the root\CIMV2 namespace and adds entries into the root\CIMV2\SMS namespace for reporting purposes.

To access WMI programmatically and see what SMS hardware inventory sees, you can use tools such as WBEMTEST (included with Windows) and the Microsoft Scripting Guy's Scriptomatic v2.

When you modify the SMS_def.mof the site server compiles it automatically and sends a copy of it to the client access point.

- The new data contained within the modified SMS_def.mof is used to generate a *hardware inventory policy* which Advanced Clients download from their management point. Upon receiving the new inventory policy, the Advanced Client evaluates it, and two minutes later applies it.

- When the SMS_def.mof is modified on the site server, the Legacy Client downloads the actual SMS_def.mof file from its client access point to the %WTNDR%ms\SMS\clifiles\binv folder.. Once there, it is automatically compiled and the new data entered into the local system WMI repository.

This book describes six methods for modifying SMS hardware inventory...I won't go into them all here again, I mean, the chapter summary shouldn't be longer than the chapter right?!

1. Reporting on an existing class
2. Inventorying registry keys using the Registry **Property** Provider
3. Inventorying registry keys using the Registry **Instance** Provider
4. Inventorying data using the **View** Provider
5. Static hardware inventory extensions.
6. Scripted hardware inventory extensions

There are two ways to add comments to the SMS_def.mof. Using two /'s like so: // and placing /* before your comment block and */ at the end of it.

Compiling a .MOF file enters the file's information into the local system WMI repository. **MOF files are compiled using MOFCOMP.EXE.**

The logs for the hardware inventory, including status messages about the success or failure of the MOF compilation can be found:

- SMS 2.0 client: %windir%\ms\SMS\logs\binv.log
- SMS 2003 Advance client: %windir%\system32\ccm\logs\inventoryagent.log

You should create a package, program, and advertisement scheduled to run on a recurring basis to deploy your custom MOF extensions to Advanced Clients to ensure that they are properly updated and contain current custom inventory information.