HP WBEM Services Software Developer's Kit for HP-UX Provider and Client Developer's Guide

Version 1.5

Manufacturing Part Number: T1434-90009

Sept 2003

© Copyright 2000-2003 Hewlett-Packard Company.

Legal Notices

The information contained in this document is subject to change without notice.

Warranty. *Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.* Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Restricted Rights Legend. Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DOD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

Corporate Offices:

Hewlett-Packard Co.
3000 Hanover St.
Palo Alto, CA 94304

Use of this manual and flexible disk(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs may be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

Copyright Notices

© Copyright 2000, 2001, 2002, 2003, Hewlett-Packard Development Company, L.P., all rights reserved.

Reproduction, adaptation, or translation of this document without prior written permission is prohibited, except as allowed under copyright laws.

This document contains information which is protected by copyright. All rights are reserved. Reproduction, adaptation, or translation without prior written permission is prohibited, except as allowed under the copyright laws.

PA-RISC and HP-UX are registered trademarks of Hewlett-Packard Company.
PDF is a registered trademark of Adobe Systems Inc.
Internet Explorer, Windows, WMI and Visio are registered trademarks of Microsoft Corporation.
Solaris and Java are registered trademarks of Sun Microsystems, Inc.
UML is a registered trademark of Object Management Group, Inc.
Itanium is a registered trademark of Intel Corporation.
UNIX is a registered trademark of The Open Group.

HP WBEM Services for HP-UX includes software developed by The Open Group Pegasus Project (`http://www.opengroup.org/pegasus`). The Open Group Pegasus Project Copyright (c) 2000, 2001, 2002, 2003 BMC Software, Hewlett-Packard Company, IBM, The Open Group, Tivoli Systems.

HP WBEM Services for HP-UX includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (`http://www.openssl.org`). OpenSSL Copyright (c) 1998-2002 The OpenSSL Project. All rights reserved.

HP WBEM Services for HP-UX includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com). This package is an SSL implementation written by Eric Young (eay@cryptsoft.com), written so as to conform with Netscape's SSL. Original SSLeay License: Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com) All rights reserved.

Contents

1. Introduction to WBEM, CIM, CIM Server, and Providers

A Common Model of Systems and Devices 14
Controlled Extensibility 16
Client-Server Architecture 17
The CIM Server 18
Providers 19

2. The Software Development Kit (SDK)

Product Contents 24
System Requirements 25
Supported Platforms 25
Installation 26
Removal 27

3. Schema Design and Implementation

Schema Design 31
STEP 1: Define High-Level Client-Use Models 31
STEP 2: Draft Object Model 32
STEP 3: Consult DMTF Model 35
STEP 4: Identify Properties and Methods 37
STEP 5: Finalize Schema Details 39
STEP 6: Write MOF Files 43
STEP 7: Optimize 44
Schema Implementation 46
MOF File Content Guidelines 46
Registering Providers 47
Rules for Updating Schema 47

4. Provider Implementation

Provider Basics 52
Provider Design Considerations 54
Flow Of A Client Request 54
Provider Execution Context 54
Provider Registration and Naming 55
Provider Naming Conventions 63

Key Values: Uniquely Specifying an Instance. 64
Multi-Threading for Concurrent Requests 67
Global Symbols 67
Security Architecture 68
The Provider Programming Interfaces 70
Relationship to Client Operations 70
General Considerations 70
Main Arguments 71
Returning Results. 71
Handling Error Conditions. 72
PegasusCreateProvider and the Provider Class Constructor and Destructor 72
Instance Provider 73
Method Provider 74
Building Providers 75
Testing and Debugging Providers 78
Installing and Running a Provider 78
Removing and Re-Installing a Provider 79
Testing With Clients. 81
Packaging and Release 82
Provider Product Content. 82
Installation, Upgrade and Removal. 83
Operation 86
Starting the Provider 86
Stopping the Provider. 86
Documentation 87
Provider Data Sheet 87
Operator's Guide. 87
Special Issues: Coexistence With Other Manageability Products. 88

5. Client Implementation
Discovery 91
Discovering Platforms 91
Discovering namespaces. 91
Discovering Classes 91
Navigating Schema 93
Keys. 93

Propagated Keys 95
Associations and Aggregations. 98
Best Practices 99
Client Security Considerations 101
Local vs. Remote Requests and Username/Password Authentication 101
SSL (Secure Socket Layer) for Encrypted Communication 101
Client Development Best Practices. 103
Using Provider Data Sheets. 103
Prototyping with wbemexec 104
Client Development Use Cases 105
General System Information 105
Multiple Dynamic Instances 106
Special Purpose Clients 106
Command Line Clients. 107
Building Clients 108
Sample Clients 109
Packaging and Release 110
Additional Sources of Information. 110

A. CIM Naming Guidelines

namespaces. 113
General Syntax 113
Managed System namespace 113
Special Purpose namespaces 114
Classes 116
DMTF Defined Classes. 116
Subclassing to Specialize 117
Subclassing to Add Properties/Methods 118
Properties and Methods. 119
Properties/Methods in Superclasses 119
Examples from Elsewhere with the Schema. 120
Being Descriptive 120

B. XML Example

C. Code Example Showing CIM DateTime Conversion

D. Example Provider Data Sheet

Provider Overview 128
Setting Up This Provider. 129
Using This Provider 130
Links to More Information 138

E. Factors for Estimating Provider Effort

F. Glossary

Printing History

Table 1

Printing Date	Part Number	Edition
September 2002	electronic release only	First
September 2003	T1434-90009	Second

The last printing date and part number indicate the current edition, which applies to the A011.05 version of HP WBEM Services Software Developer's Guide for HP-UX, Client and Developer's Guide.

The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The part number is revised when extensive technical changes are incorporated.

New editions of this manual will incorporate all material updated since the previous edition.

HP Printing Division:

Infrastructure Solutions Division
Hewlett-Packard Co.
19111 Pruneridge Ave.
Cupertino, CA 95014

Preface

Audience

This document is intended for use by software professionals who will design, implement and/or support clients or providers for HP WBEM Services for HP-UX.

Requisite Knowledge

Developers should be familiar with the following topics:

- Object-oriented design and programming concepts
- The C++ programming language
- The Common Information Model (CIM), as described on the **DMTF** (Distributed Management Task Force) web site: `http://www.dmtf.org`.

It is also essential that the provider developer be familiar with the elements of the operating system he or she is intending to instrument. That is, knowledge of system calls and their data structures, and how these will be modeled in CIM, is assumed.

Developers of CIM client applications should consult the documentation for the providers they plan to use. Appendix D, *Example Provider Data Sheet,* shows the kind of information typically provided.

Document Structure

The main body of this document is organized generally in the order in which the developer will undertake the various steps of the project. However, it will be useful to refer to the Examples included in the Software Development Kit as an aid to understanding the material in the text.

The programming interfaces (APIs) used by providers and clients are in the HP WBEM Services Software Developer's Kit in the directory `/opt/wbem/html`.

This material describes how to use the interface functions, their parameters, exceptions thrown, and other related topics. Other detailed reference information can be found in the appendices. References to other sources of information will also be found throughout the document. A Glossary is also available.

Navigation and Document Conventions

Typefont is used to help identify *CIM element names*. (Italic type is used for other things as well, including *titles* and for *emphasis*.)

Typefont is used to set off `user-specified names`, `commands` and `filenames`, and when introducing a **new term**.

The names of functions and methods will generally be indicated with parentheses, for example, `getInstance()`. The word "function" will refer to a C++ function, and the word "method" will refer to a CIM operation.

1 Introduction to WBEM, CIM, CIM Server, and Providers

A Common Model of Systems and Devices

Web-Based Enterprise Management (WBEM) is a set of standards, developed by the DMTF, to unify the management of enterprise computing environments, allowing a variety of information processing elements from different vendors to be managed in a uniform way. Systems managed with WBEM may be general-purpose computer systems running any operating system, or they may be printers, network switches or routers, storage arrays, or any other device reachable on a network.

WBEM goes beyond similar network management standards such as SNMP and DMI, and defines the following specifications:

- a rich model of manageable entities featuring inheritance and associations (the Common Information Model, or CIM)
- an extensible set of operations that can be performed on these objects (CIM Operations)
- a protocol to encode the objects and operations for communication over a network (xmlCIM).

The CIM standard specifies a model (or representation) for management data. It defines a set of objects (or classes) that are commonly found in information-processing environments. CIM has representations for entities such as disks, files, user accounts, memory, video cards, software, processes, queues, and many other familiar concepts.

The standard also specifies operations that can be performed on these objects. A set of operations that can be performed on any type of object is called the **intrinsic methods**, and includes accessing, creating and deleting instances of these objects; modifying specific properties; and so forth. Additional operations, called **extrinsic methods**, are defined for specific classes of objects. The definition for the class `CIM_LogicalDevice`, for example, contains the extrinsic methods `EnableDevice()` and `QuieseDevice()`, which would not be meaningful when referring to an object such as a `CIM_Account`, which is used to represent a user's account.

Object definitions are organized in a hierarchy. The model features the object-oriented concept of inheritance, where a new class of object may be defined as "a kind of" an already defined class: a disk is a kind of device; a device is a kind of managed element, and so forth. The new class has

all of the properties of its parent, or superclass, and possibly other properties of its own. All of its properties would in turn be inherited by any subclasses.

Inheritance offers an important benefit for client applications, since a client need not be developed to explicitly know about platform-specific features. For example, a client developed to manage user accounts could manipulate instances of a generic account object (`CIM_Account`), even though user accounts on different operating systems would typically be represented with additional, different properties. The platform-specific, fully represented account object would be defined as a subclass of CIM_Account (or one of its subclasses), but a request for all instances of CIM_Account would return the instances of subclasses, as well.

The model also describes associations between different objects. Associations are defined as classes, and therefore may also have properties and methods, and may make use of inheritance. An example of an association is the `CIM_ControlledBy` class, which allows a device to be associated with a device controller. In addition to pointers to the device and its associated controller, the definition for this class contains other information that is specific to the association itself, such as the selected data rate.

A hierarchy of object (or class) definitions resides in a namespace. A namespace can be thought of as a kind of directory or folder; namespaces themselves are organized hierarchically. HP WBEM Services for HP-UX is installed with the DMTF's CIM V2.5 class definitions preloaded in the `root/cimv2 namespace`. Other namespaces can exist on the same system, and can be used to manage separate sets of objects. The uses of namespaces and namespace security are described in more detail in the Appendix A: *CIM Naming Guidelines* and in the *Security Architecture* section of Chapter 4.

Controlled Extensibility

The hierarchical, inheritance-based nature of CIM allows it to be easily extended to account for differences on different operating systems. For example, there is no exact equivalent in Windows of the "mount point" concept of UNIX platforms. Conversely, there is no consistent notion in UNIX of the "registered user" (owner) of the operating system or other software components. Extensions to accommodate platform-specific differences are encouraged and must follow well-defined rules. New classes of objects are created by defining subclasses of existing classes in order to add new properties or to represent concepts that are unique to the platform. The task of implementing a provider often involves extending the CIM model by defining new classes of objects. Guidelines for extending the model are discussed in Chapter 3, *Schema Design and Implementation*.

Client-Server Architecture

Another component of the WBEM standards, `xmlCIM` (a specialization of the eXtensible Markup Language or XML), defines the encoding that CIM clients and servers use to comunicate about CIM objects, and the operations that can be performed on them. An example of a typical XML exchange is shown in the Appendix B, *XML Example*.

The HyperText Transfer Protocol (HTTP) is used to transport xmlCIM messages over a network. WBEM-compliant systems and devices implement an HTTP server. In the recommended implementation, the HTTP server listens on port 5988 for unencrypted communication or 5989 for encrypted communication.

The CIM Server

A WBEM/CIM Server can operate directly on the underlying system by calling the system's commands, services, and library functions, or it can pass requests from clients onward to plug-in modules called Object Providers or simply Providers. This plug-in approach is preferred for general-purpose computer systems, which may have hundreds of different classes of objects and an almost unlimited number of possible configurations.

For a device like a network printer or a **SOHO** (Small Office Home Office) router, where memory and other resources are limited, an appropriate optimization may be a monolithic server: that is, one that does all the work in a single program and does not use plug-ins.

A general-purpose WBEM server, implemented using the plug-in approach, contains a CIM Server that directs the communication between clients and providers. HP WBEM Services for HP-UX contains a CIM Server. (other components such as an HTTP server, as well as utilities and administrative commands, are also included in the product.) The CIM Server maintains the information it needs to direct client requests, as well as other information that controls its behavior, in a type of database called a repository. In future discussion, we will use the term CIM Server to refer to the combined HTTP server and CIM Object Manager request director system.

Providers

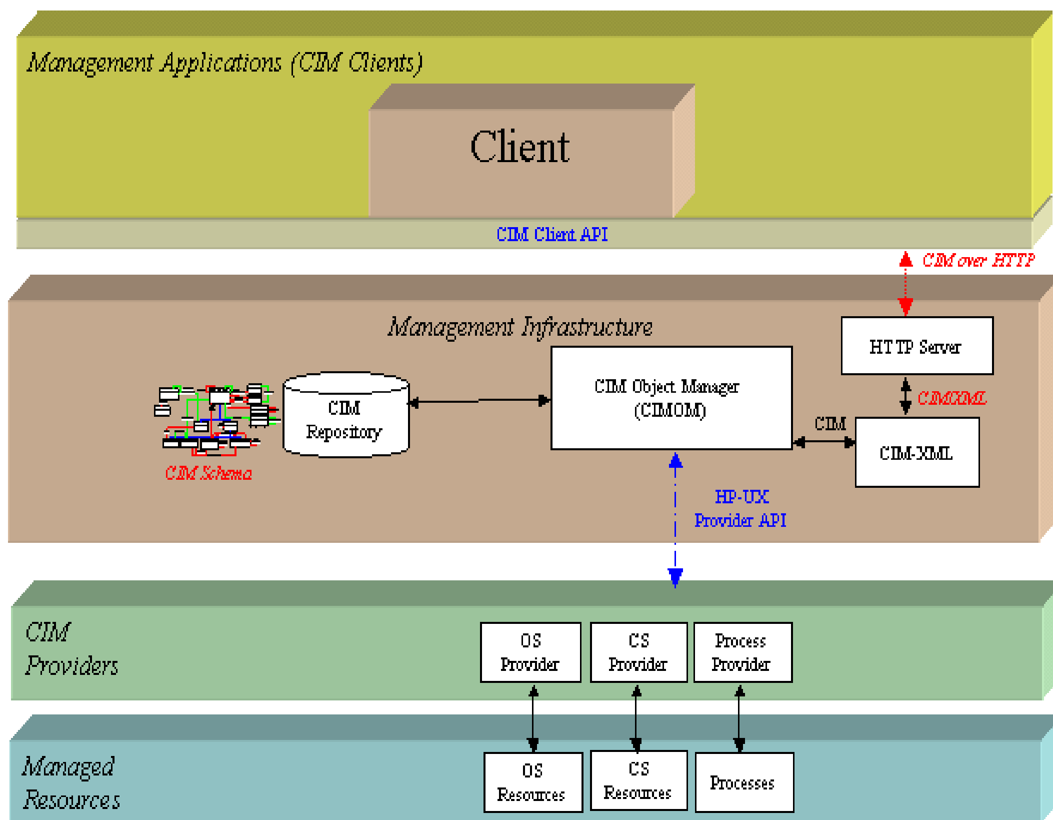
Providers get information about the system and fulfill other requests on behalf of client applications. Clients deal with instances of the classes of objects defined by CIM. An instance is a representation of a single manageable entity (such as a disk); it contains data obtained from the system by a provider. Providers perform the conversion between CIM and the platform's commands, services and data formats. CIM is the object-based representation that a WBEM client application understands.

The interface between the server/CIM Server infrastructure and the plug-in providers is called the Provider Application Programming Interface (API). The CIM Server calls a function of the Provider API when it needs to have a provider perform an operation requested by a client. The provider API functions are documented in the HP WBEM Services SDK in the `/opt/wbem/html` directory. Information and guidelines for designing and implementing a provider are described in the section on Provider Implementation.

Providers are written in C++ and are compiled and linked to create a shared library. A single shared library may contain one or more providers.

The following diagram shows the relationship among clients, the CIM Server, providers, and the underlying system features.

Figure 1-1 Figure: HP WBEM Services for HP-UX System Architecture



Two types of providers are currently supported by HP WBEM Services for HP-UX: *Instance* and *Method*. The instance provider allows manipulation of instances of classes, and of properties of those instances. The method provider allows clients to invoke extrinsic methods on instances. A single provider can be both a method and a instance provider, and can serve more than one class of object.

Instance providers must contain code for the following six functions (although a minimum implementation for a function can simply throw a `NotSupported` exception):

- `createInstance`: creates a new instance of a class of object
- `getInstance`: returns a single instance of a class

- `enumerateInstances`: returns all the instances of a class
- `enumerateInstanceNames`: returns the names of all instances of a class (see the section on Key Values: Uniquely Specifying an Instance for information about instance names)
- `modifyInstance`: replaces the values of specific properties of an instance
- `deleteInstance`: deletes a specified instance

Method providers implement just one function: `invokeMethod`. This function tells the provider to execute an extrinsic method on a specified instance.

Implementing a provider for HP WBEM Services for HP-UX consists of delivering the following three components:

- Definitions of the class or classes of objects (schema) that represent the system resources to be managed. (See Chapter 3, *Schema Design*)
- A shared library containing code that implements the functions of the chosen provider interface: instance and/or method (See Chapter 4, *Provider Implementation*.)
- Registration of the provider to inform the CIM Server of its correspondence with the defined schema. (See *Provider Registration and Naming* later in this chapter.)

These steps will be discussed in the sections that follow.

2 The Software Development Kit (SDK)

The SDK for HP WBEM Services for HP-UX is distributed as an SD depot, bundle T1434BA. Installed with the `swinstall` utility.

Product Contents

The following files are copied to the target system by the installation procedure:

- Header files containing the C++ Client API and the C++ Provider API (.h)
- Documentation:
  - API descriptions in HTML format
  - Readme file for example providers and clients
- Examples:
  - Instance Provider
  - Method Provider
  - Enumerate Instances Client
  - Invoke Method Client
- Example contents:
  - Sources
  - Makefile
- Provider examples:
  - Schema (.mof)
  - XML file(s) for testing with `wbemexec`

System Requirements

Supported Platforms

For the list of currently supported platforms, see the Release Notes for this product and for HP WBEM Services for HP-UX. HP WBEM Services documents are posted at `http://www.docs.hp.com/hpux/netsys`.

Installation

Use the HP-UX `swinstall` utility to install the Software Development Kit.

After the installation has completed, the user may build any of the examples included in the development kit to verify that the installation was successful.

Removal

Use the HP-UX `swremove` utility to remove the Software Development Kit, product bundle number T1434BA.

3 Schema Design and Implementation

Schema design is the process wherein real-world entities (such as processors, disks, software, or user accounts) are transformed into a

logical representation. The schema contains three kinds of information:

- a definition of the details of each entity
- the operations the entity supports
- how classes of these entities are related to one another on a managed system

Since real-world entities have attributes (properties) about themselves and actions (methods) they can perform, these are also aspects of the design of a schema.

A further aspect to consider is that some entities are actually specialized types of other entities.

A well-understood schema encompassing all these considerations is critical for interoperable platform-independent WBEM-based management application interaction with managed resources.

To ensure interoperability between clients and providers, it is vital that developers base their designs on standardized schemas. While a given management application and associated managed resource provider(s) could decide to standardize on some private schema, this is very likely to result in redundant information being provided and an inability for clients and providers to share information in an interoperable manner. As noted in the introduction of this document, the DMTF has defined the Common Information Model (CIM), a common data model of an implementation-neutral schema for describing overall management information in a network/enterprise environment. Using this common CIM schema enables general-purpose management clients to use a common representation of managed objects, even without prior knowledge of the providers implementing the schema. Developing with this common CIM schema lets providers add application-specific value-added functions while still supporting general-purpose clients. For example, the provider could implement an OS-specific function, while also exposing a CIM-compliant view of general OS capabilities.

Schema Design

The schema design process begins once required information and capabilities have been identified. The first prerequisite is to define use models. It is important to understand what information and capabilities clients require, and also how they support given tasks. If such information is unavailable, these requirements may come from an external view of what information is correctly exposed via a standard (for example SNMP or DMI), a command line interface (CLI), an application program interface (API), or an internal view of what information and capabilities exist for the managed resources.

A second prerequisite is to determine data sources for the management information and control points from which a provider can implement desired functions. Within this context meaningful schema design can begin.

The schema design process can be viewed as a process consisting of seven (7) steps:

- STEP 1: Define High-Level Client-Use Models
- STEP 2: Draft Object Model
- STEP 3: Consult DMTF Model
- STEP 4: Identify Properties and Methods
- STEP 5: Finalize Schema Details
- STEP 6: Write MOF Files
- STEP 7: Optimize

For further reference, DMTF also has a simple set of steps for schema design defined in the CIM tutorial at *http://www.dmtf.org/education/*.

STEP 1: Define High-Level Client-Use Models

In developing a schema, it is not enough to simply understand what data is available. Of greater importance is understanding how an end user would use the data in a management application. A thorough analysis and understanding of how an end user and associated administrative client application will use the management data will usually result in a simpler, more useful schema.

Strategies for identifying client-use models include:

- Consider one or more client use cases or scenarios (potential clients may be GUI-based tools or CLIs).
- Identify which information needs to be provided.
- Determine which actions must be performed.
- Determine logical groupings of the information/actions.

In addition, one may need to consider how the client-use model would be affected if the schema is hosted on several heterogeneous platforms. Early scoping of the model to the target platforms assists in achieving interoperability, a high-level goal of WBEM. Clients would then be able to use common access schemes to obtain the information from all platform types. The information derived from these analyses serves as a basis for creating a draft object model.

STEP 2: Draft Object Model

An analysis of managed objects entails identifying the objects and determining relationships between them. There are several types of objects that should be considered in designing a schema, including:

- **Managed Elements** are the most commonly considered objects. They are used to track the state of some managed entity and perform changes as required. Examples uses of these include devices, systems, and software elements.
- **Associations** are objects which relate two other objects. (Note that CIM associations can also have their own properties.) An example use of associations is to represent the relationship between a device driver and its associated device.
- **Aggregations** are a special type of association, with the notion that an object is made up of an aggregation of some number of other objects. For example, various logical devices can be aggregated to make up a complete computer system.
- **Services** are objects describing services and service access points implemented by managed elements. An example use is a print spooler service associated with a printer.
- **Settings** are a description of settings for managed elements. An example use is representing the kernel parameters associated with an operating system;

- **Statistics** are statistical information relevant to a given managed element. An example use is representing resource utilization of a running process.
- **Diagnostics** are diagnostic invocation, settings, and results. An example use is disk diagnostics.
- An **Indication** is the representation of the occurrence of an event. An example use is disk errors.
- **Services for Provider Control** are schema that allow control by a client of a provider's internal operations.

The process of identifying appropriate objects and their relationships is referred to as Object Oriented Design and Analysis (OOD/OOA). While a full discussion of OOD/OOA is out of the scope of this document and a comprehensive understanding is not required for effective CIM modeling, a general understanding of the concepts of object modeling is helpful. A good overview of the most useful concepts can be found in the DMTF *CIM Tutorial*, posted at `http://www.dmtf.org/education/` , which has a more theoretical approach to Schema Design and Implementation. Still, the most important prerequisite for effective modeling is expertise in the domain being modeled.

Creating a draft model of a specific domain is usually best begun by sketching diagrams using the Unified Modeling Language™ (UML™). UML provides a simple visualization of object classes and their relationships. Thinking of objects as being of the types listed above can be helpful. It is best to identify the **managed elements** first. These are the primary entities that are to be managed.

Next, determine the **associations** and **aggregations** that relate these entities. All associations and aggregations have at least two (2) properties, the references to the classes being associated. These relationships have a *cardinality* or minimum/maximum count on the number of objects that can relate to another object. Cardinality is represented in UML by placing the ranges next to the classes at the ends of the association or aggregation (for example "`1..n`" indicates there can be 1 to "*n*" instances of the object tied by that relationship to another object). If the cardinality is labelled "`*`" or if no cardinality is explicitly specified, it is assumed to be "`0..n`".

Note that associations and aggregations inherit. That is, if a superclass has an association to another class `x`, all subclasses also inherit an association relationship to `x`. Also, it is possible to create specialized subclasses of the association if required (for example to define additional properties or methods).

Next consider ancillary objects that should be associated with the primary managed elements. These include entities such as **services**, **settings**, and **statistics**. Services and services access points (SAP) are ways of representing services and the availability of those services as supported by managed element. A service class provides methods for enabling and disabling those services. Settings represent managed elements operational parameters. Several settings can be associated with an entity. Statistics represent statistical information or metrics related to a managed element. Each of these ancillary object classes should be associated with a managed element with appropriate cardinalities.

Next, consider what **diagnostics** and **indications** may be associated with a managed element. Diagnostics, as modeled using the *Common Diagnostics Model (CDM)*, enable one to represent the diagnostic test itself (including status, characteristics, and methods to start and stop the diagnostic test), settings for the diagnostic, and results of previously run diagnostic tests.

Finally, in some cases it may be useful for a client to be able to communicate with the provider itself (as opposed to the resources for which it acts as a gateway), treating the provider as a resource to be managed. This may be the case if the provider has "side effects": state or resources of its own that affect its operation. Examples of such resources could include configuration information (of the provider itself), logging behavior (stopping, starting), databases, communication facilities, and other features.

When such a need exists, it is not appropriate to model the provider's own resources in the classes it serves. These classes must be reserved for the management of the underlying platform resources. Instead, it is appropriate to define a new class by extending `CIM_Service` and/or related areas of the CIM model (it is permissible to define a class at the root level of the hierarchy, but this is not generally recommended). This class can define local properties or methods as necessary. The provider being managed should be the provider for this class of object, as well as for the classes representing the resources that it instruments.

STEP 3: Consult DMTF Model

DMTF has developed a general-purpose, implementation-independent schema for use in a large variety of applications, such as general computer systems, network devices, embedded systems, and storage arrays. The CIM schema is divided into several major categories:

- **Core**: base classes from which all other schemas are derived
- **System**: high-level representations of systems, with additional details for computer systems
- **Application**: used to manage the lifecycle of software products, drivers, firmware, and applications
- **Device**: logical representations of configuration and operation on hardware devices
- **Event**: representation of indications and subscriptions
- **Metrics**: classes for units of work and associated metrics
- **Network**: representations of network cloud and associated services
- **Physical**: the descriptions of entities which can be labeled, occupy space, and are subject to physical laws (that is physical vs. logical entities in the Device schema)
- **Policy**: representation of rules and their dependencies
- **Support**: models of service incidents
- **User**: the CIM user and security models

Analysis of a schema for a specific domain can usually be confined to 2 to 4 of the above categories. For example, modeling a managed resource such as a Network Interface Card (NIC) would encompass objects in the physical (the card itself), the device (the logical representation), the network (the protocol stack), and the application (the driver) schemas. The basic task is to identify DMTF CIM classes intended to represent similar information to that of draft classes for the domain. This exercise may result in either dividing the draft classes (identified in Step 1 above) into multiple DMTF-derived classes (extensions to the DMTF model) with appropriate associations, or merging draft classes to correspond to a defined DMTF class. Use of the classes from the DMTF schema facilitates interoperability between clients and providers. Providers can

be confident that clients which have implemented using the standard CIM schema will be able to find desired management data in expected places in the schema hierarchy.

During this process of refining the draft schema through examination of the DMTF CIM schema, it may also be interesting to identify properties and methods for some of the classes as a means to better understand the objects being modeled. A high-level look at the DMTF classes can be seen through their *Visio or PDF UML diagrams*. Further detail is available in the Managed Object Format (MOF) file class, property, and method descriptions. (Note: MOFs are text files describing CIM schema. This is similar to the way MIBs describe SNMP and MIFs describe DMI.)

Note that classes may not only be those specifically defined by the DMTF, but also may be *DMTF-derived* classes. This *class extension* is facilitated through the objected oriented concept of *inheritance*, wherein a class may be *specialized* to add more specific properties and methods. Thus, if the DMTF schema does not have the richness required for the domain being modeled, a provider can add additional properties and methods through the use of subclassing.

While a provider can (and is encouraged to) instrument a DMTF CIM class, there are several good reasons to subclass. Reasons for extending the DMTF CIM schema include:

- Needing to add properties/methods to the subclass.
- Needing to override properties/methods to add/change qualifiers.
- Allowing clients to easily enumerate (list) a subset of the instances as represented by a particular subclass.
- Allowing multiple providers to implement instances of the class. (For example, if a given provider isn't sure that it owns all the instances and another not-yet identified provider may have additional instances.)
- Needing to derive a concrete class from an abstract one defined by the DMTF. (For example, needing to derive a concrete type of service from the abstract `CIM_Service` class).

In some cases, there may appear to be no significant coverage of the modeled domain within the DMTF CIM schema. In this case, one should focus on the aspects of the Core Schema which most closely resemble the domain. Generally it is best to subclass from some DTMF class rather than making up an entirely new schema from scratch.

When searching the DMTF schema for applicable modeling elements, one may discover that the structure of the CIM data does not readily match the domain of interest. This is usually recognized by noting that the designed management data is broadly scattered across many classes, and that for most classes a provider would be able to fill in very few values for properties or instrument very few of the methods. The implication is that clients will have to search many classes and traverse many associations to get the wealth of management data they require, causing potential performance issues through too many interactions between client and provider. Discarding the DMTF schema at this point is probably NOT a good solution for resolving this issue. Instead it calls for schema optimization, as discussed in STEP 7: Optimize below.

STEP 4: Identify Properties and Methods

Having selected the objects and relationships to represent managed resources, the next step is to refine the schema by identifying properties and methods. The properties and methods for a given class define the unique characteristics of that class of object. Via inheritance, a subclass also has all the properties and methods of its superclass. The DMTF has already defined an extremely rich set of classes, properties, and methods. Therefore, it is highly advantageous to derive a sub-class from an existing CIM superclass and to use the superclass's already defined properties as this enables a greater degree of interoperability for clients and providers. Clients which need information at the superclass level can obtain the superclass derived information from any subclass (assuming the subclass provider chooses to implement the inherited property or method).

Properties can be added to describe additional attributes of a given class not present at the superclass level. It is important to ensure that a property makes sense for all instances of a class. Otherwise additional subclassing may be required wherein one subclass has the property and others do not. Some other factors for consideration when identifying new properties in a subclass include:

- Ensure that a similar property does not exist in any superclass or any closely associated class (this check helps eliminate redundant information or a difference in modeling approach than what a general-purpose management client might expect).

- Try to place the property as high in the class hierarchy as possible while maintaining appropriate meaning. Note that changes to the DMTF schema need to be worked through an appropriate DMTF working group as discussed in MOF File Content Guidelines below.
- Consider if the property would still make sense in a lower subclass derived at some point in the future.
- Make sure that qualifiers on the property (those special attributes characterizing a property, see the *DMTF web site* on qualifiers) are all necessary and sufficient; important qualifiers such as "Description" will ensure that there is a common understanding of the property between clients and providers.
- Consider what values a property will hold across various platforms. Providers on various operating systems and platforms could choose different values for properties in instrumenting similar objects. Will this implementation create confusion for client applications? Does there need to be a clearer *Description* of a property or a whitepaper to clarify a property's usage?
- Consider the client need to identify resources shared across multiple servers as being the same (making shared resources both *cluster-aware* and *SAN-aware*). If two servers share some resource (for example a disk array, backup device, or SAN element), is it important that a management client managing both platforms be able to identify that the resource being modeled is in fact the same resource? If so, it may be important to ensure that the definition of one or more properties ensures an easy recognition of the common resource (for example a common GUID or a serial number property).

Key Properties

One or more properties for a given class are specified as being key properties via the *Key* qualifier. Key properties are used to uniquely identify each instance of a given class within a `namespace`. A CIM class cannot redefine (change, add to, or delete from) key properties defined in a superclass. For the most common case, where a given class is derived from a DMTF class which already has keys defined, key identification is trivial. However, in some rare cases where a high-level, "non-keyed" class is used for inheritance, definition of key properties is necessary. Care must be taken to ensure that the properties are defined in a way that will facilitate that uniqueness and ensure interoperability across multiple platforms (for example IA-32, IPF, PA-RISC, Sparc).

Rules about Key properties when creating a subclass include:

- If a superclass already has (or inherits) key properties, *no key properties may be added* to or deleted from its subclasses.
- All concrete classes must have keys. (A concrete class is a class that may have instances.)

In conjunction with the identification of keys for a given class, one must consider possible associations with the class. To effectively implement a reference ("REF") property on an association (or aggregation), the provider for the association must know the values for all key properties for all the instances which will be referenced by that association class.

Methods

Methods specific to classes are sometimes called *extrinsic methods* to differentiate them from *intrinsic methods*, such as `enumerateInstances` or `getInstance` which are available to all classes. Extrinsic methods identify that a class possesses some capability which a client could request be performed on a particular instance of that class. Class-specific methods have very similar considerations to those of properties. However, they have some additional considerations in determining when to define methods within a class which include:

- Implement a method when it is necessary to model an operation on an instance that is not clearly modeled by setting the value of a property, especially if one or more result values must be returned. For example, while setting the operational state of a device or service could be modeled by setting the value of a property (perhaps named `State`), running a diagnostic or installing a software component which may involve several input and output parameters.
- If the method could be implemented using an *intrinsic method* supported by the CIM ServerCIM Server (for example `deleteInstance`, `createInstance`, `setProperty`), consider documenting that as an appropriate client usage (perhaps in a profile or whitepaper describing schema usage), rather than implementing a redundant extrinsic method.

STEP 5: Finalize Schema Details

This step finalizes the schema by considering it in the larger context of standard schemas, other providers, and client usage. There are several considerations that may be applicable for a given schema:

Validate Schema with Client Use Models

At this point in the process, it is useful to refer back to the client-use models derived in STEP 1: Define High-Level Client-Use Models above. A successful schema design will be clear and simple to use for management applications. Aspects of the client-use models and scenarios to consider include:

- Ensure that all the information required is available as properties on the expected objects.
- Validate that the defined operations provide the capabilities required to support the use cases without unintended side effects.
- Verify that management applications will be able to traverse associations and aggregations with reasonable efficiency to obtain the necessary information. Otherwise, it may necessary to optimize the schema, as described in STEP 7: Optimize below.
- General purpose applications may use portions of the schema at a superclass-level. Validate that appropriate properties, methods, and associations/aggregations are in place for expected functions of these management applications. Consider what information a general-purpose client which only accesses classes defined in the DMTF CIM schema.
- Any of the above considerations may surface a performance issue. These should be considered in the STEP 7: Optimize step below.

Compatibility with Industry Initiatives

Understanding the efforts of *DMTF working groups*, industry groups using the CIM schema (for example *Storage Networking Industry Association*), and the *WBEMsource schema consistency community* helps ensure that the schema will be accessible by cross-platform and general purpose management applications. The work of on-going initiatives may be helpful in evaluating applicable parts of the schema. Areas of general community interest which are not currently being discussed in a forum may suggest that a new effort should be initiated to gain community consistency and consensus.

Schema Interfaces Between Providers (Associations and Aggregations)

A significant consideration (see also Provider Design Considerations below) in understanding a provider's interaction with other providers is that of how classes instrumented by separate providers are related. In CIM, classes are related via associations and aggregations. When two related classes are supported by two separate providers, the association (or aggregation) linking the classes is an interface point between those providers.

Providers for all association (aggregation) classes must know the values of the key properties for instances on both sides of the association. However, the classes referenced by an association are not necessarily supported by the same provider as the association itself. It is important that all providers supporting instances or references to the instances have a common understanding of the values for the key properties in the instances. In selecting a provider to support the association class, there are several considerations:

- The provider must know the values for the keys on both sides of the association.
- Is it possible that both providers would not be installed? It may be important that the provider supporting the association have an installation script dependency on the other provider.
- Is there some third provider that knows how to associate these classes, perhaps at a higher level in the schema hierarchy? Careful definition of the key properties is essential.

Based on these considerations, determine which provider should own and implement the association.

Future Schema Enhancements/Extensions

If capabilities which may become future enhancements can be identified, it is also helpful to consider how these areas would affect the schema. Backward compatibility of schema is highly desirable. It ensures that existing clients will not be adversely affected by upgrading to a new version of the schema. A simple rule is that CIM schemas can be ensured to be backward compatible by just adding (not deleting or changing) classes, properties, methods, and relationships. Additional details are discussed in Rules for Updating Schema below.

Standard CIM practice does not permit removal of classes, properties, or methods for minor version updates; instead `deprecation` is permitted through the `Deprecate` qualifier. Thus, while deprecation is possible, it does imply continued provider support of the deprecated schema, enabling clients to continue functioning with deprecated schema, re-implementing to the new schema over time. Hence, providers must support both the deprecated and revised schema elements.

General Schema Use Cases

One may find several situations when doing schema design that require further analysis. While the list below is by no means complete, it is an initial reference list from which to analyze a proposed schema.

- Common Properties

  When implementing weak associations, some key properties are propagated from their scoping class. It is important to ensure consistency among these properties. Information is also available in the Key Properties section. An example of a weak association is the `CIM_SystemDevice` aggregation of `CIM_LogicalDevice` to `CIM_System`. If more than one instance of `CIM_System` exists, it is particularly important to ensure that the key properties propagated from `CIM_System` to `CIM_LogicalDevice` (`SystemName` and `SystemCreationClassName`) match between instances to ensure that it is clear which devices are a part of a particular system.

- "Unknown" Subclass

  There may be situations in which a single provider is unable to appropriately place all instances in specific subclasses. Thus, the appropriate subclass for a particular instance is unknown. An approach in this case to create a special subclass indicating the unique properties that are unknown. (There are probably no additional properties on the subclass from those present in the superclass.) Use of this special "unknown" subclass facilitates discovery of the instance when enumerating instances of the superclass. If at some point, additional information becomes available to the provider to determine the specific subclass, the "unknown" instance can be reclassified and a more appropriate subclass instance created.

  An example of this use case would be a provider for instances of devices attached to a USB port. The provider(s) would be able to recognize devices attached to the port for which drivers are currently

installed, identifying them as appropriate subclasses of CIM_USBDevice (for example ACME_USBPrinter or ACME_USBMouse). However, very little is known about an attached device which does not have a driver. Such an unknown device could be modeled as a separate "unknown" subclass such as ACME_UnknownUSBDevice. If at some time, the required driver were installed to permit identification of the device, the provider could then delete the instance of ACME_UnknownUSBDevice and create an instance of the appropriate subclass (for example ACME_USBScanner) to represent the additional attributes of the device.

In-Schema Documentation

Each class and property may have a Description qualifier. The Description qualifier, which can be obtained and displayed by client programs, supplies important information about the schema element it is describing. In the case of a property, for example, it can elaborate on the meaning of specific values that the property could have. In the case of a method, it could discuss the intended use of parameters, the meaning of return values, or other information.

STEP 6: Write MOF Files

The next step is creation of Managed Object Format (MOF) files. MOF files are used to specify schema to the CIM Server via the MOF Compiler (really a MOF Loader). This step details the process and other considerations in creating MOF files. Specifics on MOF file syntax are available in the and the CIM Specification. Guidelines for creating MOF files are described in the section below on Schema Implementation.

Identify Property Types and Method Signatures

Identify the appropriate type for each property and method within a class. Also be sure that the parameters within the method are typed appropriately. It is better to select a larger data type if size is in doubt (for example uint16 vs. uint32) since property types and method signatures cannot be changed in future schema versions without breaking backward compatibility.

In addition, one should consider if it is necessary to override a property (via the Override qualifier) from a superclass (perhaps to add a Write or more detailed Description qualifier) as discussed below.

Determine Qualifiers

Qualifiers are attached to classes, properties, and methods. Select necessary and sufficient qualifiers for each. Particular qualifiers to consider include:

- `Association` to indicate a class is an association;
- `Description` which should be used for all properties, methods, and classes to describe their purpose;
- `Version` which is required to ensure clarity of which definition is being used for a given class.

A list of all possible qualifiers is available for reference on the *DMTF web site*.

STEP 7: Optimize

The final step before designing the provider itself is to "desk-test" the draft schema and perform schema optimizations as required. Having the released management client which will use the schema is ideal, but if one is not available, several other options exist:

The `wbemexec` command is a simple client supplied with HP WBEM Services for HP-UX that can be used to exercise all provider functions (please refer to the `wbemexec man` page).

Key things to consider include:

1. Availability of the data to be displayed and methods to be invoked
2. Ease of accessing the required data (not too many association traversals)
3. Performance of the client interaction

Considerations (1) and (2) would suggest that a non-inherited property or method may need to be moved/copied from one class to another. Consideration (2) is of particular interest when there are too many interactions between client and provider, and the resultant latency adversely impacts client performance. If it is necessary to make a copy of a property in a second location within the schema to permit simpler access, be sure that the data comes from the same underlying managed resource so the data is kept consistent as viewed by the clients and providers.

Performance considerations in case (3) usually involve situations where a large number of instances are retrieved (and perhaps filtered) to obtain the necessary information.

An example of this case would be an application which wishes to show the total disk space available on a platform. Retrieving all disk instances and adding up the disk space available on each would be fine for a small number of disks, but as the number of disks grows to the hundreds or even thousands, performance of sending large XML messages across the network may become a concern. An alternative would be to use another class to represent the total disk space (such as a subclass of `CIM_PhysicalCapacity`) which when queried would have the provider (instead of the client) add up the total space before sending a smaller message with the data required to the client application.

A second example would be a client looking for instances of `CIM_SoftwareElement` which correspond to device drivers. In this case enumerating all instances of `CIM_SoftwareElement` and then filtering those of interest could be very inefficient. Instead, creating a separate subclass of `CIM_SoftwareElement` for the device drivers would allow the client to enumerate just those instances of interest.

Schema Implementation

In addition to the DMTF-specified syntax for MOF files as described in the CIM specification and the guidance provided by the DMTF MOF Style Guide, additional factors should be considered for the development, testing, and on-going maintenance of MOF files.

MOF File Content Guidelines

MOF is the standard format for defining CIM objects. HP WBEM Services for HP-UX includes a MOF compiler for loading of schema into the CIM Server's repository (please see the man page for the `cimmof` command, as well as discussion below). Tasks that can be performed through loading MOF file via the CIM Server's MOF compiler include: (1) defining classes (and associated superclasses), (2) defining qualifiers, and (3) creating instances of classes.

- Class/Superclass Definitions

  Classes, along with their associated superclasses, are defined in MOF files. All superclasses should be included in the schema definition. Order of definition is important, so superclasses must be defined prior to their subclasses (no forward referencing). When referencing DMTF-defined objects, include the class definitions verbatim.

  Changes to DMTF classes are implemented through the DMTF change request (CR) process, and so should not be implemented *ad hoc* by a provider implementer. HP WBEM Services for HP-UX is installed with DTMF CIM v2.5 preloaded in the `root/cimv2 namespace`.

- Qualifiers

  Generally speaking, there is no need to define additional qualifiers via MOF files. Use of the existing DMTF-defined qualifiers is sufficient. Qualifiers are defined during the creation of each `namespace` in the CIM Server's repository and should not be redefined in MOF files.

  A full list of the qualifiers and their definitions can be found on the DMTF web site, `http://www.dmtf.org`.

- Instance Definitions

  Instances can be defined in MOF. The result of these definitions is that a provider supporting instances of the class receives a request to create an instance with the values specified in the MOF file.

Registering Providers

The CIM Server associates classes with the providers that instrument them through instances of the Provider Registration classes in the special `root/PG_InterOp` namespace: `PG_Provider`, `PG_ProviderModule`, and `PG_ProviderCapabilities`. An example is shown in the section on Provider Registration and Naming. While provider registration instances are conveniently specified in MOF, they should be in a different file than the one used to specify the schema definition so that these different components can be managed separately.

Note that the provider registration mechanism of HP WBEM Services for HP-UX eliminates difficulties encountered with the `Provider` qualifier. These include:

- Requires modification of schema definition (which is in principle improper if the schema being modified is part of the DMTF-supplied CIM schema)
- Does not distinguish between an `Instance` and an `Association` provider for an association class
- Does not permit more than one provider to a class
- Cannot specify supported properties or methods

HP WBEM Services for HP-UX ignores the `Provider` qualifier if present.

Rules for Updating Schema

There are several considerations when either planning or implementing an upgrade to a previously released MOF file.

Backward Compatibility

It is very important to consider backward compatibility when upgrading a MOF file because CIM clients may continue operation before and after a MOF/provider upgrade. Causes of failures in applications due to upgrades include (but are not limited to):

- Deletion of a class, property, or method
- Movement of a class within the hierarchy in any direction except "down" as a side effect of a new superclass being inserted in the line of descent above
- Changing a property's type or a method's signature (method type, parameter type, names of parameters, number of parameters)
- Decreasing Max or increasing Min cardinalities of associations

A detailed specification of changes which cause a major revision to the CIM schema, and thus can affect applications, can be found in the section labeled *Schema Versions* in the latest CIM Specification on the DMTF web site: http://dmtf.org.

Note that the DMTF CIM schema will never shrink through the life of a major version (for example Version going from CIM v2.6 to CIM v2.7). It continues to grow due to this upgrade strategy.

Deprecation

Deprecation and use of the `Deprecate` qualifier is the appropriate route for dealing with classes, properties, and methods within the schema for which there is a plan to discontinue support in the future. While within a major version of the CIM schema (for example Version 2.x), the `Deprecate` qualifier signals that the entity will no longer be available in a future major release (for example Version 3.x). The class, property, or method should continue to be supported until the future release but the qualifier indicates that there is a preferred alternative in place (the parameter of the `Deprecate` qualifier).

Discontinuing Support

When upgrading to a new major version of the CIM schema, it is possible to discontinue support for a capability implemented in the schema through a class, property, or method. A major release (for example from Version 2.x to Version 3.x) is most likely placed within a different namespace, reducing compatibility issues between clients and providers. Because it is difficult to synchronize installation of clients and providers, clients would need to support both the old schema and the new schema.

Discontinuing support on a minor version changed within a major version (for example Version 2.x to Version 2.y, where x < y) is not recommended, as discussed in Backward Compatibility above. However,

it could be accomplished if a provider could be certain that all clients using the class, property, or method were upgraded simultaneously with the provider. This is very difficult to ensure and is not recommended.

Class "Version" qualifiers

The DMTF uses the major and minor version numbers to reflect which version of the CIM schema originated a class. The format for class version numbers is "`m.n.u`" where:

- `m` = major version (numeric)
- `n` = minor version (numeric)
- `u` = update (errata or DMTF coordinated changes) (numeric)

In CIM Schema version 2.6, for example, the major version number is 2 and the minor version number is 6. For more information, see the section labeled *Schema Versions* in the latest CIM Specification.

When upgrading a provider, it is important to be sure that all affected classes (those which are extensions created by the provider implementer) have their version number incremented appropriately. When defining additional subclasses, provider developers can use version numbers associated with their product as long as they conform to the syntax described in the list above. It is important to be sure that the major version number is only incremented when major revisions are made to the schema (those possibly breaking backward compatibility). Incrementing the minor version indicates that the schema upgrade will not adversely impact client applications. The update number can be used for internal changes relative to a defined (probably released) baseline. The `Version` qualifier for the class should only be changed if the class definition is changed. Developers can use whatever version numbers they'd like as long as they comply with the rules stated above.

Superclasses Shared with Other Providers

Extensions to the DMTF CIM schema which result in superclasses that are shared among providers must be handled carefully when considering a MOF file upgrade. If changes to these shared classes must be implemented, coordination among those sharing the class is essential. The "owner" of the shared class must coordinate changes between the affected parties and adhere to the upgrade policies described above. The

newest version (with correct version number) of the class should be used with a new release of any of the MOFs which all share the same superclass.

4 Provider Implementation

This section has information and examples to help you design and implement a provider.

Provider Basics

The provider development task has three components:

- define the resources or system parameters to be manipulated as CIM objects (the schema)
- write C++ functions as described in the Provider API specifications (built as a shared library). For API information, see HP WBEM Services Software Developers Kit documentation in `/opt/wbem/html` directory.
- inform the CIM Server how to associate the defined CIM objects with the shared library (through provider registration)

The basic steps for writing a provider are:

1. Determine what information to expose:

   Developers need to determine what information and/or operations this provider will provide to clients. Providers can choose to allow read and/or write access to properties. Examples of properties include system time, operating system type, serial number, and so on. Providers can also support extrinsic methods to allow clients access to operations. For example, a provider can support a method that allows users to set the system time. Tips and concerns for choosing which properties and methods to expose are discussed in Chapter 3, in the section entitled STEP 1: Define High-Level Client-Use Models.

2. Determine how to get the information:

   Developers need to determine how the provider will get access to and/or write the information or get the operation it is exposing to occur. This is very specific to the type of information so there is little this document can do to help with this step. Providers can use standard system calls to get information or can use private files and databases.

3. Define the schema to use and the properties and methods that will be supported

4. Determine which namespaces the provider will use to support the specified objects

5. Write two MOF files: one defines the schema to be instrumented, and the other is used to register the provider
6. Design the provider and make implementation decisions
7. Implement the provider
8. Build the provider
9. Test and debug the provider
10. Document, package, and release the provider

We recommend that developers start by writing the Provider Data Sheet (PDS). The provider data sheet (see the example in Appendix D) supplies information on what properties and operations are exposed by this provider, what interfaces are used, how to install this provider, and what platforms and operating systems are covered. Starting with the PDS requires that the first three tasks in the list above be addressed, and so helps the developer focus on the most important decisions about the provider and what it will do before implementation is started.

The following sections give details on each of these steps along with examples, guidelines, issues and concerns.

Developers can refer to Appendix E, *Factors for Estimating Provider Effort*, for guidelines in determining the effort required for provider development.

Provider Design Considerations

Flow Of A Client Request

The following is a high-level description of the control flow of a client request:

- A client issues a request for data or an operation; the request is encoded in `xmlCIM` by the Client API (if used) and sent to the CIM server.
- The CIM server receives the xmlCIM request, decodes it, and authenticates it by determining whether the username and password are valid (authentication is automatic for local requests - see the `connectLocal()` client function).
- The CIM server determines whether the user is authorized to perform the requested operation in the specified `namespace`.
- The CIM server searches the provider registration information to determine which provider can service the request.
- The CIM server loads the appropriate shared library (if it's not already loaded), and calls the appropriate provider.
- The provider has the responsibility to determine whether the client is authorized to issue the request if necessary (in addition to the simple `namespace` authorization already performed by the CIM server).
- The provider performs whatever operations are required to satisfy the request, and delivers information requested, as appropriate, to the CIM server.
- The CIM server encodes the information received from the provider into xmlCIM and returns it in its reply to the client.

Provider Execution Context

A provider module is implemented as a shared library. The CIM Server loads the shared library and initializes the provider on demand (on the first access of an object served by the provider). In HP WBEM Services

for HP-UX, the provider functions are called in the process context of the CIM Server. However, the **provider must not assume that it is running in the same process as the CIM Server**.

The process context in which a provider runs is determined by the CIM Server, and is not configurable. This means that it is not possible to set environment variables, working directory, permission mask, or any other environment parameters before a provider is activated. Furthermore, the provider must not set any of these during operation. If it is necessary to do so for correct operation, the provider should create a separate process with the needed environment, and communicate with this process by any suitable means of interprocess communication (for example, pipes, shared memory, files, or another mechanism).

Providers run under the root user ID in HP WBEM Services for HP-UX. However, **developers must not assume that this will continue to be true** in future releases. Please refer to the section on Security Architecture for more information.

Providers built for IPF deployment should be compiled as native programs. They will not work if compiled in emulation mode.

Provider Registration and Naming

Provider source code is compiled and linked to build a shared library. The CIM Server will call the functions in this shared library when a client requests an operation on a CIM object that the provider instruments. The CIM Server finds the shared library containing the provider for a particular CIM object by searching the Provider Registrations that it was supplied when providers were installed. Provider Registration consists of CIM instances of three classes of objects, all defined in the `root/PG_InterOp` namespace:

- `PG_ProviderModule`
- `PG_Provider`
- `PG_ProviderCapabilities`

PG_ProviderModule

The `PG_ProviderModule` object corresponds to a single shared library. Example 4-1 shows its definition, in a MOF declaration:

Example 4-1 PG_ProviderModule: An abridged MOF declaration

```
class PG_ProviderModule : CIM_LogicalElement
{
[ Key ] string Name;
  string Vendor;
  string Version;
  string InterfaceType;
  string InterfaceVersion;
  string Location;
  uint16 OperationalStatus[];
  string OtherStatusDescription;


  uint32 start();
  uint32 stop();
};
```

A shared library (provider module) may contain more than one provider. This can simplify sharing code among several providers that must use the same system services to perform their operations. The name of the shared library file must be of the form `lib<`*`library-name`*`>.sl` where `<`*`library-name`*`>` is the string specified in the Location property of an instance of `PG_ProviderModule`.

The shared library file may reside in any directory on the system, but a symbolic link to the file must exist in `/opt/wbem/providers/lib` in order for the CIM Server to find it.

The `Name` property of `PG_ProviderModule` is an arbitrary string that will be referenced in instances of `PG_Provider` and `PG_ProviderCapabilities`. Like the `Location` property (and the shared library name), it is advisable to use the word Module in the string.

Example 4-2 PG_ProviderModule: Instance shown in MOF

```
instance of PG_ProviderModule
{
  // Properties inherited from CIM_ManagedElement
  Caption = "Dynamic Information Provider Module";    // not required

  Description = "Module containing several providers";// not required
  // Properties inherited from CIM_ManagedSystemElement
 InstallDate = "20020517140341.000000-420";          // not required

  // Properties local to PG_ProviderModule
```

```
  Name = "DynamicInfoModule";            // required
  Location = "DynamicInfoModule";        // required
  Vendor = "ACME Computer Corp.";
  Version = "2.0.0";
  InterfaceType = "C++Default";         // required
  InterfaceVersion = "2.1.0";           // required
};
```

The `Location` property is used to construct the path of the shared library containing the provider module as described in Figure 4-2, above.

The extension is either `.sl` or `.so`, depending on the platform:

- On HP-UX PA, the path in this example would be:

  `/opt/wbem/providers/lib/libDynamicInfoModule.sl`

- On HP-UX IA, the path in this example would be:

  `/opt/wbem/providers/lib/libDynamicInfoModule.so`

The `InterfaceType` and `InterfaceVersion` properties are required with the values exactly as shown in order to allow possible future implementations to support different provider interface protocols and versions.

PG_Provider

The `PG_Provider` object identifies a single provider and the `PG_ProviderModule` in which it can be found. Example 4-3 shows its definition in a MOF declaration:

Example 4-3 PG_Provider: an abridged MOF declaration

```
class PG_Provider : CIM_LogicalElement
{
    [ Key, Propagated("PG_ProviderModule.Name") ]
  string ProviderModuleName;
    [ Key ]
  string Name;
};
```

The ProviderModuleName property is a string whose value must be the same (case insensitive) as the Name property of an instance of PG_ProviderModule.

The Name property in PG_Provider is the string the CIM Server will pass to the PegasusCreateProvider function when it needs a pointer to the provider (the first time it calls a function in the provider). Example 4-4 below shows a source fragment with an example of C++ provider class declarations and an implementation of the PegasusCreateProvider function. Example 4-5 shows how an instance of PG_Provider associates a provider name with a provider module corresponding to a shared library.

Example 4-4 Provider source fragment showing provider class declarations and implementation of the PegasusCreateProvider entry point

```
// This module contains two providers
//    ProcessStatusProvider
//    MemoryInfoProvider

// class declaration for the Process Status Provider
class ProcessStatusProvider : public CIMInstanceProvider
{
  ...
};

// class declaration for the Memory Info Provider
class MemoryInfoProvider : public CIMInstanceProvider
{
  ...
};

// This module (shared library) contains two providers.
// PegasusCreateProvider returns a pointer to the provider
// that was requested by the CIM Server. For each provider, it
// will create an instance of the class on the heap
// (i.e., with "new") and return the pointer.
//
// The string passed to PegasusCreateProvider is the
// PG_Provider.Name, and does not need to match the
// name of the C++ provider class declared above.
//
// extern "C" tells the C++ compiler to generate a
// pure C symbol name for PegasusCreateProvider, rather
// than a "mangled" C++ symbol name
//
```

```
extern "C" CIMProvider *
  PegasusCreateProvider(const String &providerName)
{
  // create new provider instance on heap and return its
  // address. will require the provider's terminate() function
  // to use "delete this;" to deallocate
  if String::equalNoCase( providerName, "ProcessStatusProvider" )
    return new ProcessStatusProvider;

  if String::equalNoCase( providerName, "MemoryInfoProvider" )
    return new MemoryInfoProvider;

  // If neither name was recognized, there could be an
  // error in a registration instance
  return 0;
}
```

Example 4-5 PG_Provider: A MOF showing an instance

```
instance of PG_Provider
{
  // Properties inherited from CIM_ManagedElement
  Caption = "Process Status Provider";                //not required
  Description = "Instruments several classes in the
    "CIM_LogicalElement";                             //not required

  // Properties inherited from CIM_ManagedSystemElement
  InstallDATE = “20020517140341.000000-420”;   // not required

  // Properties local to PG_Provider
  ProviderModuleName = "DynamicInfoModule";  // required
  Name = "ProcessStatusProvider";            // required
};
```

In Example 4.5, the instance of `PG_Provider` informs the CIM Server that the provider named `ProcessStatusProvider` will be found in the Provider Module named `DynamicInfoModule`. For our example, there would be another instance of `PG_Provider` to tell the CIM Server that the `MemoryInfoProvider` is also in the `DynamicInfoModule`.

PG_ProviderCapabilities

The class `PG_ProviderCapabilities` is used to associate a provider with the class for which it supplies instances, methods, indications, or associations in the specified namespace. Example 4-6 shows the definition of `PG_ProviderCapabilities` in a MOF declaration:

Example 4-6 PG_ProviderCapabilities: An abridged MOF declaration

```
class PG_ProviderCapabilities : CIM_ManagedElement
{
[ Key, Propagated ("PG_Provider.ProviderModuleName") ]
  string ProviderModuleName;
    [ Key, Propagated ("PG_Provider.Name") ]
  string ProviderName;
    [ Key ]
  string CapabilityID;
  string ClassName;
  string namespaces[];
    [ ArrayType ("Indexed"),
      ValueMap { "2", "3", "4", "5" },
      Values {"Instance","Association","Indication","Method"}]
  uint16 ProviderType[];
  string SupportedProperties[];
  string SupportedMethods[];
};
```

As in the `PG_Provider` object (Example 4-3), the `ProviderModuleName` tells the CIM Server which module contains the provider that this capabilities instance describes. It must be the same as the `Name` property of an instance of `PG_ProviderModule`. Likewise, the `ProviderName` property must be the same as the `Name` property of an instance of `PG_Provider`.

The `CapabilityID` key makes it possible to have several instances naming the same provider and module, but with different capabilities. They might differ in the class that they instrument, or they may name different sets of properties or implement different provider types. However, *there can be no more than one Instance Provider for each class in a namespace*! There may be more than one `Method Provider` as long as they do not claim service for the same method.

The following descriptions are taken from the complete MOF definition of `PG_ProviderCapabilities`:

- `SupportedProperties`

  lists the properties supported by this provider. If this array is NULL, the provider MUST support all of the properties defined in the class. If the provider does not support all of the properties, the properties supported MUST be included in the array.

- `SupportedMethods`

  lists the methods supported by this provider. If this array is NULL, the provider MUST support all the methods defined in the class. If the provider does not support all the methods, the methods supported MUST be included in the array.

MOF Example 4-7, below, shows a class definition. Example 4-8, following that, shows an instance of `PG_ProviderCapabilities` that associates the provider with the newly created class:

Example 4-7 MOF declaration of the class ACME_ComputerSystem

```
  [ Description (
      "The class ACME_ComputerSystem extends CIM_ComputerSystem "
      "to add several properties containing additional owner "
      "contact information." ),
    Version( "1.2" ) ]
class ACME_ComputerSystem : CIM_ComputerSystem{
    [ Description(
        "Additional contact information for the primary "
        "owner for this computer system. This is intended "
        "to be the telephone number of a pager." ) ]
  string PrimaryOwnerPager;
  ...
};
```

Example 4-8 PG_ProviderCapabilities: A MOF instance

```
instance of PG_ProviderCapabilities
{
// Properties inherited from CIM_ManagedElement
Caption = "ComputerSystemProvider Capabilities"; // not required
Description = "First capability description for the "
"Computer System Provider";         // not required

  // Properties local to PG_ProviderCapabilities
  ProviderModuleName = "ComputerSystemModule"; // required
  ProviderName = "ACME_ComputerSystemProvider";  // required
  CapabilityID = "1";                          // required
```

```
  ClassName = "ACME_ComputerSystem";              // required
  namespaces = { "root/cimv2" };                 // required
  ProviderType = { 2 }; // 2=InstanceProvider   // required
  SupportedProperties = NULL; // NULL=All properties
  SupportedMethods = NULL;    // NULL=All methods
};
```

The instance of `PG_ProviderCapabilities` in Example 4-8 informs the CIM Server as follows:

- This capability instance is for a provider named `ACME_ComputerSystemProvider` in the module `ComputerSystemModule`.
- There can be more than one instance for the same provider; this one has a unique `CapabilityID` of 1. A different set of capabilities might specify a different class instrumented by this provider, or the same (or different) classes in different `namespace`s, possibly with different characteristics.
- The `ClassName` property identifies the class instrumented by the provider. A single provider can support more than one class, but each Capabilities instance describes the provider's capabilities for a single class. If, for example, the provider will respond to operations at different levels of the model hierarchy, additional Capabilities instances would be required. In this example, we could have another Capabilities instance specifying that the same provider supports operations on `CIM_ComputerSystem`, the parent class of `ACME_ComputerSystem`.
- The provider instruments `ACME_ComputerSystem` only in the `root/cimv2` `namespace`. It is possible to support the same class in several `namespace`s with the same provider. Note that the definition of the instrumented class itself (`ACME_ComputerSystem`, in this example) must be declared in the specified `namespace`.
- This provider implements the *Instance Provider* interface, supporting instance manipulation operations. The values for the `ProviderType` property are:
  - — Instance
  - — Method

A provider may implement more than one interface, but *there can be no more than one Instance Provider for a class in a* `namespace`! Often, it is useful to implement several interfaces in the same provider, as this allows the platform-specific code that accesses system resources to be shared.

- The `SupportedProperties` property is an array of strings, each of which name a property of the instrumented class. If the value is `NULL`, or if `SupportedProperties` is absent, all properties are supported. If `SupportedProperties` is an empty set ({ }), no properties are supported. Note that declaring support for all properties of a class does not obligate the provider to actually supply (or accept) all properties. It is permissible to throw `CIMNotSupportedException` for properties that are not supported. The declaration of support for a property informs the CIM Server that this provider accepts responsibility for the property.
- Like `SupportedProperties`, the `SupportedMethods` property specifies methods that are supported. The provider must register as a Method Provider, by placing a “5” in the `ProviderType` array, for the `SupportedMethods` to be meaningful. While there can be more than one method provider for a given class, only one method provider may be registered for the same method of a class.

The files containing MOF fragments shown above must be loaded into the CIM Server's repository: the files containing provider registration instances must be loaded into the `root/PG_InterOp namespace`, and the file containing the definition of the `ACME_ComputerSystem` class must be loaded into a normal, non-system `namespace` (please refer to the section on Installing and Running a Provider). When a client requests an operation to be performed on a class, the CIM Server will examine these instances to determine which provider(s) to call.

A `Version` qualifier should be used on the class definition for `ACME_ComputerSystem`. This allows a provider installation procedure to determine whether an existing schema definition is a lower version (and can therefore be upgraded) or equal to or greater than the version supported by the new provider (in which case it should not be modified).

Provider Naming Conventions

There is no required relationship between the name of the provider (`PG_Provider.Name`), the name of the module (`PG_ProviderModule.Name`), the name of the shared library into which it

is built (PG_ProviderModule.Location), and the schema element with which it is associated (PG_ProviderCapabilities.ClassName). However, for clarity, it is useful to choose names that help to identify the element. The conventions evident in the previous example show how this may be done.

Key Values: Uniquely Specifying an Instance

Most providers manipulate instances or instance references (also called *object paths*) in some way. The set of all keys of an instance defines the unique identity of an instance within a namespace of managed objects. For example, there are possibly many users whose Name is "smith", but on any given UNIX computer system, there can be only one called "smith." In order to know exactly which user "smith" we are interested in (to address email, for example), we must know the name of the system in question. Therefore, for a user account, a unique identification must include the SystemName as well as the Name (just as an Internet email address is specified as username@domain-name). In this example of a class representing a user account, the SystemName and Name, together, form a set of key properties that uniquely define each instance. In this example, the SystemName is said to specify the context (or scope) in which the Name will be unique. For many classes in the CIM model in which keys are defined, there will be at least one key property to specify the scope of the class's other keys.

Keys that specify necessary scope are always propagated from another class of object, as mentioned in the section on *Common Properties* in Chapter 3, under *General Schema Use Cases*. The SystemName key present in many CIM classes is propagated from the Name property of the CIM_System class (or one of its subclasses). When scoping keys exist in a class definition, the classes from which the scoping keys are propagated are also specified as key properties. So if there is a SystemName key, there must be a property specifying the name of the class from which the SystemName is obtained. This key is the SystemCreationClassName. For a typical class in the hierarchy of CIM_ManagedSystemElement, (CIM_LogicalDevice, for example), the full set of key properties might be:

- **SystemCreationClassName**

  The name of the class which has an instance from which the value of the SystemName property is obtained. This could be, for example, CIM_ComputerSystem.

- **SystemName**

  The `Name` property in the instance of the system that identified the object in question. A typical value could be a fully qualified IP hostname, such as `idsys.hp.com`. The values returned for these properties by different providers must be consistent. That is, all providers that return the `SystemCreationClassName` and `SystemName` keys must return consistent values for these keys when referring to the same scope.

- **CreationClassName**

  The name of the class to which the instance belongs, usually the class that the provider is instrumenting. Clients can request an enumeration of instances of a higher class in the hierarchy, so this property identifies which class the instance is from. This is important, because there may be several subclasses of a given class on which a client requests an enumeration, and all instances returned must have a unique set of key values. Consider, for example, the classes `CIM_TapeDrive` and `CIM_DisketteDrive`, both subclasses of `CIM_MediaAccessDevice`. The providers for these classes may have been implemented to return a simple integer for their `DeviceID` property, so that the first tape drive's `DeviceID` is "0" (zero), and the first diskette drive's `DeviceID` is also "0". Without the `CreationClassName` key, a client enumerating the superclass `CIM_MediaAccessDevice` would receive two instances with identical sets of keys. Only the `CreationClassName` key guarantees that the full set of key values will be unique.

  When a provider is registered to serve more than one level of the hierarchy, as in the example above, it *must always return the same value for `CreationClassName`*, regardless of which class it is responding to. This is essential so that clients will always receive the same set of keys for a given instance. In general, it is useful to return the name of the highest class served by the provider, because this affords the greatest portability (clients will generally "know" about more general classes, and may not know about extension classes on specific platforms).

- **Name (or analogous)**

The unique name of the instance within the scope defined by the previous keys. In many classes, this non-propagated key will be called `Name`, `DeviceID`, `Handle` or something that allows the user to understand that it will contain a value that is unique within that class of object.

It is entirely up to the provider to determine the value of a `Name` (or analogous) key. The primary requirement is that *there may be only one instance of a class with a given key value*. If this were not the case, it would be impossible to distinguish between different instances in a given context. Although not a requirement, the value can be chosen to be representative of what a client would expect to see. For example, the name of a disk partition on HP-UX, as displayed by the `df` command, is often something like `/dev/vg00/lvol3`, so this would be an appropriate value for the `Name` property of a disk partition object.

The values for `SystemCreationClassName` and `SystemName` must be chosen with equal care, since the provider must coexist with other providers on the same platform (indeed, in the same `namespace`). The values of these properties must be chosen to ensure consistency and avoid conflicts. When the meaning of `SystemName` is clearly a system's IP hostname (as it will often be), the provider must supply a standard fully qualified Internet hostname. This value is not reliably obtained from the `gethostname()` library function, but rather from `gethostbyname()`. The following code fragment illustrates a suitable means to obtain this value:

Example 4-9 Code Fragment to obtain value for SystemName

```
#include <sys/param.h>
#include <netdb.h>
#include <Pegasus/Common/String.h>
...

  struct hostent *he;
char hn[MAXHOSTNAMELEN];

  // fill in hn with what this system thinks is its name
  gethostname(hn,MAXHOSTNAMELEN);

  // find out what the nameservices think is its full name
  if (he=gethostbyname(hn)) return String(he->h_name);

  // but if that failed, return what gethostname said
  else return String(hn);
  ...
```

Use of Empty String Key Values

While the full set of keys is necessary to uniquely identify an instance within a `namespace`, it is useful to allow clients to specify empty strings for key values when there is no ambiguity. For example, when a client submits a `getInstance()` operation to a CIM Server, it is clear that the value of the `SystemName` key will normally be the same as the system to which the client has connected. This is especially useful when managing a *multi-homed* system (a system that may have more than one IP address or DNS name), since the client need not guess the value of `SystemName` when there can be several different hostnames for the same system. Unless the provider has a reason to require a value for `SystemName`, it should accept an empty string for this key without returning an error. As a general rule, *providers should accept empty strings for any keys that they do not actually require to identify an instance* from among those they manage. However, the provider should always supply values for all keys when returning data to the client.

Multi-Threading for Concurrent Requests

The CIM Server can process several client requests in different threads concurrently in the same provider. Therefore, **provider code must be 100% thread-reentrant** and must avoid/prevent concurrent access of any shared resources. If shared (global) resources are used, access must be serialized by the use of a suitable *semaphore* (or *mutex*) to protect the shared resource against modification that could lead to erroneous behavior. This is particularly important in provider interface methods that alter platform behavior, such as `modifyInstance()`, `createInstance()`, and `deleteInstance()`, but applies to any code that manipulates a globally accessible resource or may have a side effect. Information on developing thread-safe applications can be found online in documentation for the aCC C++ compiler at `http://www.docs.hp.com`.

Global Symbols

While all providers implement functions of the same names (for example, `getInstance()`), there is no duplication of symbol names for symbols defined within a C++ provider classes. The C++ compiler generates

unique symbols, because the functions are declared in differently named provider classes. However, it is possible to declare global symbols (technically, `named ::`*<name>*), and these may indeed "collide" with symbols of the same name. This issue is not unique to C++ nor to HP WBEM Services for HP-UX providers. Care should be taken to avoid the use of global symbol names whenever possible. When necessary, names should be chosen with a component, such as a module name, that will guarantee, or at least increase the probability of, uniqueness.

Security Architecture

User Authentication

Access to WBEM Services is restricted to users with valid accounts on the system being managed. Requests from remote clients contain a username/password pair that the CIM Server will authenticate. Requests may also be received through a local connection using the `connectLocal()` function in the Client API. This function does not take username or password arguments. The user ID for a local client request is that of the process issuing the request.

`namespace` Authorization

In addition to user authentication, if the feature is enabled, the CIM Server performs `namespace` authorization (this is disabled when the product is installed, but can be enabled with the `cimconfig` command). There are several `namespace`s serviced by the CIM Server. Each `namespace` has an associated list of users who are authorized to access its objects, and what level of access is permitted (read, write, or read+write). The `namespace` authorization database is managed with the `cimauth` command, described in the `cimauth` man page.

The section on Provider Registration and Naming describes how providers can register to serve classes in multiple `namespaces`. `namespaces` are also discussed in Appendix A, *CIM Naming Guidelines*.

Execution Context

Once a client request has been authenticated and authorized, the username is passed to the provider in the `OperationContext` parameter present in all function calls. No password is passed to the provider.

As previously mentioned, providers run under the `root` user ID. Providers must use the username to determine whether the user has permission to perform the requested operation. This determination must be made in addition to the `namespace` authorization that the CIM Server may have performed. The provider must not perform any operation that would be unauthorized for the user on whose behalf it is executing the request. While it may seem technically possible, `setuid()` must never be called in the provider process, since other requests may be running concurrently in other threads (in the same provider or in others that may be loaded in the same process context). If done in a thread-safe manner, it is permissible to create a separate process under a specific user ID. This may be an appropriate design strategy in cases where it is the most or only reliable means of ensuring secure operation. Care should be taken to consider performance and resource utilization.

If needed for additional authorization, the `namespace` of the target object can be obtained from the object identification parameter of the request, as in the API documentation of the HP WBEM Services SDK, in the `/opt/wbem/html` directory.

The Provider Programming Interfaces

This section discusses the provider interfaces at a general level. Specific information about the functions, their interface, arguments and data types, can be found in the HP WBEM Services Software Developer's Kit documentation in the `/opt/wbem/html` directory.

Relationship to Client Operations

The interfaces of the Provider API are the mechanism by which the CIM Server passes requests from clients onward to providers, which then manipulate the actual system resources modeled (represented) by CIM objects. The operations that clients can request generally correspond to the functions of the provider interfaces. However, it is important to remember that they are not identical. For example, the client operations `getProperty` and `setProperty` are converted by the CIM server to calls to the Instance Provider functions `getInstance()` and `modifyInstance()`, respectively.

General Considerations

- All providers implement the CIMProvider interface (`initialize()` and `terminate()`).
- A provider must implement (be declared as a C++ derived class of) one or more other interfaces: either the CIMInstanceProvider or the CIMMethodProvider interface. A single provider can implement both of these:

  ```
  #include <Pegasus/Provider/CIMInstanceProvider.h>
  #include <Pegasus/Provider/CIMMethodProvider.h>
  class MyProvider : public CIMInstanceProvider, public
  CIMMethodProvider
  ```

  All methods in an interface must be implemented; if a method is to be unsupported, it should throw a `CIMNotSupportedException` exception.
- A provider may service more than one class, but there can be at most one instance provider for a given class, and it must "know about" all instances of the class(es) that it services.

Main Arguments

Most provider API functions take a class or an instance reference (an instance of the C++ `CIMObjectPath` class, sometimes called an instance name or object path) as the principle argument. A reference uniquely identifies an object (a class definition or an instance). It specifies a class name and, if an instance reference, the set of keys that uniquely identify the object.

Most functions that perform write operations (modify, delete) take a full instance (an instance of the C++ CIMInstance class) or a class definition (an instance of the C++ CIMClass class), as opposed to a reference (a `CIMObjectPath`), as the principle argument.

Other arguments and how they should be handled are covered in the *Provider API* sections in the `/opt/wbem/html` directory.

Returning Results

The C++ functions of the provider interfaces are of type void, and return no value. All normal (non-error) responses are returned to the CIM Server via a callback mechanism: functions receive a `ResponseHandler` object. The `ResponseHandler` class contains the following public member functions for returning data:

- `processing()`

  Informs the CIM Server that the provider is going to deliver results

- `deliver()`

  Delivers a potentially partial result to the CIM Server

- `complete()`

  All results for the operation have been delivered

Providers are encouraged to return partial results to enumerations in calls to `deliver()`, rather than constructing a complete result array. This aids overall system performance, as well as serving as a means for the CIM Server to know that a provider is still operating in cases where the result may be large.

Handling Error Conditions

All errors must be reported to the CIM Server by throwing an exception. The exception codes are listed in the API documentation in the Software Developers Kit, in the `/opt/wbem/html` directory.

PegasusCreateProvider and the Provider Class Constructor and Destructor

As previously discussed, a provider is an instance of a C++ class derived from one or more of the provider interfaces. When the CIM Server needs to invoke a provider function for the first time, after loading the appropriate shared library (if it has not already been loaded), it will call `PegasusCreateProvider()` with the name of the provider specified in the corresponding instance of `PG_Provider`. There must be exactly one `PegasusCreateProvider` entry point in a shared library, regardless of how many CIM classes may be served and regardless of how many C++ provider classes may be implemented.

`PegasusCreateProvider()` must return a pointer to an instance of the C++ provider class that will handle requests for the specified provider. The C++ provider instance can be created with new (so-called heap storage), or statically (an example is shown below). If the provider instance was created with new, it must be explicitly deallocated by a delete this; statement at the end of the provider's `terminate()` function. However, if it was created statically, it must not be explicitly deallocated with delete.

The provider source fragment shown in the section on Provider Registration and Naming illustrates how `PegasusCreateProvider` may be coded, creating the provider instance with new. The fragments below show static provider allocation.

Example 4-10 Code fragment showing static allocation of a provider instance

```
// class declaration for Example Provider
class ExampleProvider : public CIMInstanceProvider
{
...
};
...
// global static instantiation - exists in shared library
// and comes into existence when shared library is loaded.
// Note that the use of the "static" storage class
```

```
// specifier is not required for a global instance declaration,
// as shown.
ExampleProvider exp;

extern "C" CIMProvider*
PegasusCreateProvider(const String &providerName)
{
// return  the address of the global provider instance
if String::equalNoCase(providerName, "ExampleProvider")
return &exp;
else
return 0;

}
```

The provider constructor and destructor should do as little as possible. Any initialization should be done in the `initialize()` function, and cleanup in the `terminate()` function.

Instance Provider

The InstanceProvider interface is the most commonly implemented provider interface. An instance provider is required for any class that will have instances that can be displayed or manipulated by a client.

An instance provider must supply values for all required and key properties (required properties may be inherited from superclasses). This is required by the CIM standard for DMTF compliance, not HP WBEM Services for HP-UX. Properties not supported can be omitted by simply not adding the property when building a CIMInstance for responses to read operations (`getInstance()` and `enumerateInstances()`). However, on write operations (`createInstance()` and `modifyInstance()`), the provider must throw a `CIMNotSupportedException` exception if the specified instance contains properties that it does not support.

If an instance provider performs authorization, it may do so before attempting the requested operation. If the provider determines that the client is not authorized to perform the requested operation, it must throw an `CIMAccessDeniedException` exception.

If there are no instances to return in a request to `enumerateInstanceNames()` or `enumerateInstances()`, the provider should not throw an exception. It can simply return, or it may call the

ResponseHandler `processing()` and `complete()` functions without calling `deliver()`. In either case, the client will receive an empty response.

If the instance named by `getInstance()`, `modifyInstance()`, or `deleteInstance()` does not exist, the provider must throw a `CIMObjectNotFoundException` exception.

Method Provider

A MethodProvider implements methods that are defined in a class. These are referred to in CIM as extrinsic methods (the operations of the Client API, such as `getInstance()`, are referred to as intrinsic methods). When a client wishes to invoke such a schema-specified method, it uses the `invokeMethod()` operation. The Method Provider must implement the provider interface function, `invokeMethod()`. In the implementation, it may examine the argument that contains the name of the requested extrinsic method, and thereby implement any number of extrinsic methods.

Building Providers

For detailed information on the following material, please refer to the documentation for the HP-UX C++ compiler (aCC) and linker (ld) at `http://www.docs.hp.com/`hpux/dev.

The provider C++ source file must be compiled with the aCC compiler. The options are:

- **-c** (optional)

  Causes the compiler to stop after the compilation phase and generate an object file output (.o) instead of continuing on to the link phase. This flag is inserted automatically when using implicit rules in the make utility (see the man page for make(1) for more information).

- **-mt**

  Instructs the compiler to generate symbols and macros needed to support multi-threaded operation

- **+Z**

  Produces position independent code (PIC) necessary to build a dynamic shared library

- **+DAportable** (PA only)

  Generates code for any HP9000 hardware architecture (although the generated code may execute slightly more slowly, this is strongly recommended, because products compiled for specific architectures must be generated for each architecture and packaged as separate products or as single products with multiple, architecture-specific file sets)

- **+DD64** (IA only)

  Generates code using the LP64 model.

- **-AP** (PA only)

  Turns off AA mode; uses the older classic C++ runtime libraries.

- **-AA** (IA only)

  Turns on newly suppported ANSI C++ Standard features, such as `namespace std` and the new C++ Standard Library.

- **-I/opt/wbem/include**

  Instructs the compiler where to look for header files that are needed to compile providers

- **-DPEGASUS_PLATFORM_HPUX_PARISC_ACC** (PA)
  -DPEGASUS_PLATFORM_HPUX_IA64_ACC (IA)

  Defines compile-time macro needed to control conditional compilation on HP-UX

The shared library should be linked with the `aCC` command using the following options (shared libraries could also be linked with the `ld` command, but using aCC automatically links the necessary C/C++ standard run-time libraries):

- **-b**

  Generates a shared library, rather than an executable image

- **-Wl,+hlib<name>.<library_suffix>**

  Passes the `+h` flag to the linker, which causes the specified string to be inserted in the shared library as its internal name (please see the man page for the `ld` command). This feature, while not absolutely required, allows management of different versions of a shared library, in that programs can be built to run with versions at or later than a specified value.

  The <*name*> string should match the Location property value of the `PG_ProviderModule` registration object. The extension `<library suffix>` must be the appropriate shared library suffix for the platform. Use `.sl` for PA, or `.so` for IA; for example:

  ```
  -Wl,+hlibMyProviderModule.sl
  or
  -Wl,+hlibMyProviderModule.so
  ```

  The same string should be specified as the output file to be generated by the linker with the `-o` flag:

  ```
  -olibMyProviderModule.0
  ```

- **-L/opt/wbem/lib**

  Instructs the linker to look in `/opt/wbem/lib` for the following shared libraries that will be dynamically accessed during execution

- **-lpegcommon**
  -lpegprovider

Inserts into the list of dynamically linked shared libraries:

— For PA: `/opt/wbem/lib/libpegcommon.sl` and `/opt/wbem/lib/libpegprovider.sl`

— For IA: `/opt/wbem/lib/libpegcommon.so` and `/opt/wbem/lib/libpegprovider.so`

Other options may be specified, as needed, as long as they do not conflict with those shown above.

The following example shows a Makefile constructed according to the previous discussion.

Example 4-11 Makefile Example

```
MODULE = MyProviderModule
OUTFILE = lib$(MODULE).sl
CXX = aCC
## -c not needed in CXXFLAGS because it's always used
## automatically by make when using implicit compile rules
CXXFLAGS= -mt +Z +DAportable \
-I/opt/wbem/include
-DPEGASUS_PLATFORM_HPUX_PARISC_ACC
LD = aCC
LDFLAGS = -b -Wl,+h$(OUTFILE)
LIBS = \
-L/opt/wbem/lib \
-lpegcommon \
-lpegprovider \
-lpthread \
-lrt
OBJS = MyProviderMain.o MyProvider.o
$(OUTFILE) : $(OBJS)
$(LD) -o$@ $(LDFLAGS) $(OBJS) $(LIBS)
```

Testing and Debugging Providers

The following kinds of testing should be considered:

- Installation
- Schema modification during installation: verify that guidelines for upgrading schema are observed. (See Chapter 3, Rules for Updating Schema.)
- Normal operation
- Invalid requests
- Continuous hours of operation (CHO)
- Memory leaks
- Thread-safe operation
- Code coverage/path analysis
- Requests that return very large responses; for example, returning a large number of instances
- Very large/long requests
- Large number of concurrent requests
- Provider shutdown
- Update/re-install
- Removal

Installing and Running a Provider

Run-time testing of providers must be done on a system on which HP WBEM Services for HP-UX is running. In order to make a provider available for testing, the following steps should be taken:

1. Create a symbolic link (with the `ln -s` command) to the shared library in `/opt/wbem/providers/lib`. For example:

```
$ ln -s /<test-path>/libTestProviderModule.sl
/opt/wbem/providers/lib
```

Note that the ownership and permissions on both the link and the target file (and its path) must not rely on the fact that the current CIM Server runs under the root user ID.

2. Create schema definitions for the class or classes that the provider serves (if not already done) by using the `cimmof` command to compile and load MOF files containing the class definitions. For example:

   ```
   $ cimmof TestClass.mof
   ```

   Since TestClass.mof should contain the class definitions of all parent classes as well as the new class to be defined (as described in the section on MOF File Content Guidelines), the `cimmof` command will usually generate warning messages for classes that already exist. This is normal and can be ignored.

   Note that in this example, the new schema is being created in the default `namespace`, `root/cimv2`. In some cases, the new class will be defined in a different `namespace`. This can be specified with the -n flag on the `cimmof` command:

   ```
   $ cimmof -n root/altnamespace TestClass.mof
   ```

   In some cases this may be a new `namespace` that does not yet exist. If it is necessary to create a new `namespace`, this must be done first. It may be done by loading the qualifier definitions with the `cimmof` command:

   ```
   $ cimmof -n root/altnamespace
   /etc/opt/wbem/mof/CIM_Qualifiers25.mof
   ```

3. Register the provider by creating instances of the provider registration classes in the `PG_Interop` `namespace` using the `cimmof` command (information on Provider Registration can be found in the section on Provider Registration and Naming). *This must be done by the `root` user.* For example:

   ```
   # cimmof -n root/PG_InterOp TstProviderRegister.mof
   ```

Removing and Re-Installing a Provider

It is normal to modify and rebuild a program many times during a software development project, and provider development is no exception. In order to test a rebuilt provider, the previous version must be stopped and removed so that the new one can be installed. Once started,

providers remain loaded until a system administrator invokes a command to stop and unload them. The `cimprovider` command is used for this purpose, and is described in the `cimprovider` man page.

The steps required to stop and unload a provider so that a new shared library can be installed are:

1. Stop the provider using the `cimprovider` command. (Note that this command actually operates on the provider module and stops all providers in the module.) For example:

   ```
   $ cimprovider -d -m TestProviderModule
   ```

   Disabling provider module...

   Provider module disabled successfully.

   In some cases, it may be necessary to change the Provider Registration. This must be done by using the `cimprovider` command to remove the Provider Registration instances, rather than simply stop the provider as described in the previous step.

   ```
   $ cimprovider -r -m TestProviderModule
   ```

   Note that this form of the `cimprovider` command will remove registration for all providers in the specified module. Registrations for individual providers can be removed with:

   ```
   $ cimprovider -r -m TestProviderModule -p ABCProvider
   ```

   After registrations have been removed, new registration instances must be created as described in the previous section, Installing and Running a Provider.

2. The provider shared library file may now be replaced with the new version:

   ```
   $ mv <build-path>/libTestProviderModule.sl
   <test-path>
   ```

   Note that it is not generally possible to build the new provider shared library directly into the location of an existing shared library which is currently in use. HP-UX locks binary files that are in use, and any attempt to modify will result in a message that the file is busy. This is why it is necessary to direct the CIM Server to disable and unload the shared library with the `cimprovider` command.

3. Finally, the provider can be re-enabled with the `cimprovider` command:

```
$ cimprovider -e -m TestProviderModule
```

Enabling provider module...

Provider module enabled successfully.

Again, if the provider was removed in order to update the Provider Registration, rather than simply stopped, then it must be re-registered with `cimmof`, rather than re-enabled with cimprovider, as described in the previous section on Installing and Running a Provider.

Testing With Clients

Once the provider has been installed for testing as described in the previous section, it is available for testing. The shared library will be automatically loaded and the provider initialized on the first client operation on a CIM class served by the provider.

HP WBEM Services for HP-UX is delivered with a simple, general-purpose command-line client suitable for testing providers: `wbemexec`, described in the associated man page. It is used to send `xmlCIM` messages containing CIM operations to the CIM server, which will forward them to the appropriate provider. An example of an `xmlCIM` request and response is included in Appendix B, *XML Example*.

The following example command illustrates how `wbemexec` can be used to send this request to the CIM Server on the local (same) system, redirecting the reply to a file for later analysis:

```
$ wbemexec enumInstNames.xml > reply.xml
```

The sample providers included with the SDK are accompanied by XML requests that may be modified to test providers. The sample providers are located in the `/opt/wbem/sample/Providers` directory.

The sample client programs included with the SDK illustrate the use of `EnumerateInstances` and `InvokeMethod` client APIs. They may be modified to test a given provider. Information on client development can be found in the Client Implementation section of this document.

Packaging and Release

Providers for HP WBEM Services for HP-UX are software products. They may be delivered through any vehicle suitable for the delivery of a software product intended for installation on HP-UX. The Software Distributor facility (SD) is particularly well suited for this purpose (SD is described on HP's web site at `http://software.hp.com/products/SD_AT_HP/faqs/general.html`).

When appropriate, a provider may be packaged with the system component that it makes accessible through WBEM. For example, a peripheral device is generally shipped with the software drivers and management commands that make it usable. The WBEM provider for the device is ideally included in the software components that accompany the device.

Provider Product Content

A provider product package contains software that will perform the provider installation, and the following product components:

- **Schema definition**

  The class or classes that will be instrumented by the provider. This can be delivered as MOF files that will be loaded during provider installation with the `cimmof` command, or the schema definition can be created by a small client program developed specifically for this purpose. Even if the provider will instrument CIM classes that will normally already be found on the system, the package should contain the schema definition component, specifying the entire line of descent of the class.

- **Provider binary(ies)**

  One or more shared libraries that contain the code that implements the functions in the *Provider API*.

- **Provider Registration**

  The instance specifications that register the provider for its schema. As with the schema definition, this can be delivered as MOF files that will be loaded with the `cimmof` command, or the instances can be created by a client program developed specifically for this purpose.

- **Provider Data Sheet (PDS)**

  A description of the provider which includes a list of properties and methods served by the provider. This component is not required for a provider package, but is strongly encouraged.

- **Operator's Guide**

  A conventional document providing a detailed description of system requirements and the procedures for installing, upgrading, and removing the provider, as well as other information, if required, for configuring and operating the provider. This information may be delivered in the PDS. It is not required.

- **Other Components**

  In some cases, the implementation of a provider may include other executable components, data or configuration files, online documentation, clients that use the provider, and other components.

Installation, Upgrade and Removal

Installation, upgrade and removal is done with HP WBEM Services for HP-UX running.

Installing on Systems with HP WBEM Services for HP-UX

The basic steps in a provider installation are:

- Check for prerequisites
- Disable existing provider
- Move files to target locations
- Update schema
- Perform other tasks, as necessary, such as provider configuration, file initialization, or similar functions.
- Enable new provider

Target File Locations

Provider installation must deliver the following two types of information to the system (though provider implementations may additionally include other components not specifically required by HP WBEM Services for HP-UX):

- Executable binary (built as a shared library)
- Schema being instrumented and provider registration (installed with either the `cimmof` command or a client application

The following directory is reserved for the required components listed above:

`/opt/wbem/providers/lib`

Installation of the schema instrumented and provider registration are the responsibility of the provider developer, and are normally performed in the provider's SD configure script.

The CIM Server will look in this directory for the provider shared library. The provider installation should create a symbolic link in this directory that points to the actual file location.

Other provider components, such as MOF files, other scripts and binaries, configuration or database files, and other components may be installed in any directory. To simplify system administration, however, it is advisable to follow generally accepted conventions for file names and locations, where appropriate.

Disable Existing Provider

When upgrading a provider on a system on which hp WBEM services is running, it is necessary to first disable the existing provider. This is done with the `cimprovider` command, described in the command's man page and also in the section on *testing providers*. If there is no existing provider that would be affected by this installation, or if HP WBEM Services is either not installed or not running, this step is not necessary.

Move Files to Target Locations

Files should be copied to their final locations, and symbolic links created as described above.

Update Schema and Register the Provider

As discussed in the section on Schema Design and Implementation, the provider product must supply its own class definition(s), plus the definitions of all superclasses, as these are not guaranteed to be present when the provider is installed. Schema may be updated by using the `cimmof` command on MOF files that are included in the installation

procedure, or by a client program that has been implemented specifically for this purpose. In either case, it is imperative that the schema update follow the Rules for Updating Schema described previously.

Upgrading an Existing Provider

The basic steps in upgrading a provider are:

- Check for prerequisites
- Disable the provider. (See section on *Stopping the Provider* in this chapter.)
- Move component files to target locations
- Update schema if necessary
- Complete installation tasks such as configuration
- Enable the provider

In general, providers are installed while the CIM Server is running. Client requests will be denied while the provider is disabled.

Please refer to the section on *Rules for Updating Schema* when it is necessary to modify the schema definition. Some provider upgrades may not modify supported class schema definitions but ALL provider upgrades must change the registration MOF since the provider version has changed.

If a provider is implemented to operate with an upgraded schema, it may also be designed to function correctly if it is installed with a prior version of schema, but this is not required.

Removal

The basic steps in removing (uninstalling) a provider are:

- Check for dependencies (does anything else depend on this provider?)
- Remove the provider (see section on Removing a Provider)
- Remove schema definitions. This step is not required, and generally may be omitted.
- Remove files from system (using `swremove` if files were installed with SD)
- Perform other cleanup operations, if necessary

Operation

Starting the Provider

A provider is started automatically by the CIM Server when a client requests an operation on a class that it services and the module containing the provider is not currently disabled. The `cimprovider -l` command can be used to verify that the provider module has been loaded. The `cimprovider` command can be used to disable and enable the provider module. Please see the cimprovider man page for more information.

Stopping the Provider

The `cimprovider -d` command can be used to disbable all providers in a module so that the shared library can be updated or removed. Please refer to the `cimprovider` man page for more information.

Documentation

Provider Data Sheet

The Provider Data Sheet is a description of the important characteristics of the provider. It lists:

- System requirements (dependencies)
- Files included in the bundle
- Installation steps (quick guide)
- Class(es) supported
- *interfaces* that are implemented
- Properties and methods
- Exceptions thrown
- Files and environment variables used

A Example Provider Data Sheet is included in the Appendix D.

Operator's Guide

An Operator's Guide, if included, should describe the installation procedures, as well as specific information about provider operation, such as configuration, use of special features, and other related topics.

Special Issues: Coexistence With Other Manageability Products

HP WBEM Services for HP-UX can co-exist with other applications. Providers can co-exist with other applications as long as they do not both require exclusive use of the same resource.

- **SNMP**

  SNMP and HP WBEM Services for HP-UX can run on the same system simultaneously. It is worthwhile considering WBEM providers for future development due to the richness of modeling data with CIM, the limitations on information included with SNMP traps and the benefits of handling WBEM data.

- **EMS**

  EMS (Event Monitoring Service) and HP WBEM Services for HP-UX can run on the same system simultaneously. EMS monitors and WBEM providers can co-exist as long as they do not require exclusive use of the same resource.

- **DMI**

  DMI can co-exist with HP WBEM Services for HP-UX. If DMI is installed, some information that it provides will also be available through WBEM in the `PG_ComputerSystem` object.

5 Client Implementation

A WBEM client application is any software application that makes manageability requests (CIM operations) of a CIM server using the `CIM-XML` protocol over HTTP. Clients can range from general purpose to extremely specific. Clients may be designed to interoperate in

heterogeneous platform environments or to operate on a particular platform, manage only the local system on which they are running, or manage one or more remote systems or entities. Platforms may be general-purpose servers or specific-purpose devices such as printers or storage arrays.

While there are many factors in designing a client application (graphical user interface design, human factors evaluations, plug-in and infrastructure architecture and others), this section focuses only on the aspects of interacting with the WBEM CIM Server (discovery, CIM operations, interpreting results, and other topics). Clients may also want to make use of their own CIM classes and thus may also be CIM providers.

To write a client application, you need to:

- Define the management operations you need to perform. This could include simply monitoring information on a single system or actively managing several managed entities.
- Identify the schema and applicable providers on the managed platforms which support those requirements.
- Implement the Client application, including:
  - *Discovery* on platforms of interest, `namespace`s, classes, and instrumented elements;
  - *Client navigation* of the schema;
  - *Security*;
  - *Compiling and linking*;
  - Test strategy and documentation; and
  - Release (including distribution, installation, and upgrade).

Documentation for the C++ Client API may be found in the HP WBEM Services SDK in the `/opt/wbem/html` directory.

Discovery

Discovering Platforms

HP WBEM Services for HP-UX listens for HTTP communication on port 5989 (for encrypted communication, the default behavior) or 5988 (for unencrypted communication). A client can attempt to connect to either of these ports to determine if a CIM Server is running.

Discovering namespaces

When a client can successfully issue requests to a system on which HP WBEM Services for HP-UX is running, it can determine what namespaces exist by performing an enumerateInstanceNames operation on a special class named __namespace (not case sensitive), specifying the root namespace in the operation. (Note that this functionality is deprecated by the DMTF, to be replaced by another mechanism, and support may be discontinued at some future date.) The following code fragment shows an example:

```
#include <Pegasus/Client/CIMClient.h>

int main()

{

CIMClient c;

Array<CIMObjectPath> namespaces;

...namespaces =
c.enumerateInstanceNames("root","__namespace");

}
```

Discovering Classes

Once a namespace of interest has been identified, the process of discovering classes is straightforward. One can use the enumerateClassNames and enumerateClasses operations to retrieve subclasses of any class in the namespace. Specifying NULL for the class

name will begin enumeration at the root level in the namespace. A value of true will return all derived classes. The CIM_ERR_INVALID_CLASS status will be returned if the specified class does not exist.

Thus, for a client to discover if schema of interest is supported on the target platform, it can perform enumerateClassNames operations for the classes that it would use. If the client is capable of using properties or methods in subclasses, the enumerateClasses operation can be used to obtain the properties and methods defined for the subclasses.

Navigating Schema

An important consideration in client design is what methodologies will be used in navigating the schema presented by CIM servers on various managed platforms. A correct understanding of the use of keys and associations in schema navigation can facilitate greater efficiency and performance of client/server interactions as well as enabling interoperability across heterogeneous managed systems.

Keys

Specific objects, or instances of classes, are accessed through the use of keys. Keys are identifiers that uniquely define instances within a `namespace`. A particular class's key can be made up of one or more key properties.

Qualifiers for Classes with Keys

A key property has all the characteristics of any other property in a CIM class, only with the additional semantics associated with having the Key qualifier. These characteristics of a property include that its related qualifiers are inherited by all subclasses. The combination of all key properties defined for a class makes up the compound key for the class. The following example MOF fragment shows a class definition for the class `CIM_OperatingSystem` where the combination of properties `CSCreationClassName`, `CSName`, `CreationClassName`, and `Name` (the key properties) makes up the key of the class.

Example 5-1 Abridged MOF declaration of CIM_OperatingSystem

```
// ===========================================================
// OperatingSystem
// ===========================================================
class CIM_OperatingSystem : CIM_LogicalElement {
[Propagated ("CIM_ComputerSystem.CreationClassName"),
       Key, . . . ]
string CSCreationClassName;
[Propagated ("CIM_ComputerSystem.Name"), Key, . . . ]
string CSName;
[Key, . . . ]
string CreationClassName;
[Override ("Name"), Key, . . .]
```

```
string Name;
uint16 OSType;
string OtherTypeDescription;
. . .
// additional properties and methods . . .
};
```

Key properties may be defined for an abstract class (a class with the Abstract qualifier). If a class has a concrete class as its parent, it must have the same keys. Only concrete classes can have instances.

Keys on Subclasses

A class inherits all the keys of its superclass. This fact is of particular importance when considering a class whose parent is a class with keys. All subclasses of a class with keys will have the same set of key properties; a subclass of a class that has keys cannot add additional keys. While initially not being able to add keys to a subclass may sound like a limitation to the CIM model, in fact it is a powerful feature which enables greater flexibility for clients in dealing with classes for which a provider has identified additional subclasses. Clients are able to access a class and all its subclasses in a consistent manner through the use of a common key structure.

CreationClassName

Many important DMTF CIM superclasses include a key property called `CreationClassName`. The creation class name is typically the name of the class itself or in some cases that of a superclass.

The creation class name is of particular use on managed systems where subclasses are implemented by a number of independent providers. In such cases, the providers can ensure uniqueness of their instances through specifying the name of the class itself for the `CreationClassName` property. Thus, while other properties on a given class and its subclasses may have the same values, the `CreationClassName` property's unique value ensures that each instance of the class and its subclasses is unique. A portion of the DMTF schema which extensively uses the `CreationClassName` key property is CIM_LogicalDevice and its subclasses. While instances of `CIM_DiskDrive`, `CIM_PowerSupply`, and `CIM_SCSIController` (all

subclasses of CIM_LogicalDevice) might have common values for DeviceID, having values for CreationClassName which match their individual class names ensures uniqueness.

Enumerating Instance Names

When a client performs the enumerateInstanceNames CIM operation, it specifies a namespace and a class name. The CIM server then returns values of the key properties for instances of that class within the specified namespace. Each instance is uniquely defined by the values of its key properties. The enumerateInstanceNames operation returns all instances of the class specified, which include instances of all of its subclasses, since those are also instances of the specified class. For example, enumerating instance names of CIM_LogicalPort would return any instances of its subclasses, CIM_USBPort and CIM_FibrePort, as well as possible instances of CIM_LogicalPort, if any, that are not instances of those subclasses.

The enumerateInstances operation works in a similar manner though it also returns the other properties of the class and subclasses. Thus in the example above, in addition to the set of keys defined for CIM_LogicalPort, the enumerateInstances operation would return all properties on the instances returned. In particular, it would also return any additional properties of CIM_USBPort and CIM_FibrePort that may be unique to these subclasses.

Propagated Keys

There are two techniques for associating instances of classes with one another: using propagated keys, which define weak associations (described here), and using association classes (discussed below). Each technique has its own particular advantages. Understanding those advantages enables the client application to more efficiently access information and functionality within the schema.

Scoping

Propagated keys define a context or scope within which instances of a particular class are unique. Examples of scoping relationships include:

- Logical Device on a Computer System
- User Account within a Domain
- Operating System running on a Computer System

- Process within an Operating System running on a Computer System

For each of these examples, an instance of the first class is more completely defined by understanding within which instance of the second class it is scoped. Note that in the last example, a process is scoped within the context of both its operating system and its computer system.

The scope for a given class is defined by the set of key properties that are copied (propagated) from the scoping class or classes. A propagated key property, making up a part of a class's key, will not only have the Key qualifier as discussed above, but also will have the Propagated qualifier to indicate that it has been propagated from a scoping class. The following MOF fragments illustrate the relationship between the `CIM_System` and CIM_LogicalDevice classes, showing how keys in CIM_LogicalDevice are propagated from `CIM_System`:

Example 5-2 Abridged MOF declaration of CIM_LogicalDevice

```
class CIM_System : CIM_LogicalElement {
[ Key ]

string CreationClassName;

      [ Key ]

string Name;

...

};

...

class CIM_LogicalDevice : CIM_LogicalElement {

[ Propagated("CIM_System.CreationClassName"), Key ]

string SystemCreationClassName;

[ Propagated("CIM_System.Name"), Key ]

string SystemName;

[ Key ]

string CreationClassName;

[ Key ]

string DeviceID;

...

};
```

Finding Associated Instances

The use of propagated keys effectively creates associations between classes without explicitly defining an association class for which instances would need to exist in order for a client to be able to find associated objects. When a client receives an instance of an explicitly defined association class, the paths to referenced objects are obtained directly from the properties of the association instance. In contrast, when a client application encounters an association linking two classes through the use of propagated keys, it must construct object paths from the values of the propagated keys in order to access the referenced objects.

For example, as shown above, instances of CIM_LogicalDevice are associated with a `CIM_System` by the presence of two keys, `SystemCreationClassName` and `SystemName`, that are propagated from the keys of `CIM_System`. The referenced `CIM_System` can be accessed by constructing an object path specifying:

- class name of `CIM_System`
- key named `CreationClassName` whose value is the value of the `SystemCreationClassName` key (perhaps it was `CIM_ComputerSystem`)
- key named `Name` whose value is the value of the `SystemName` key (let's say it was `sys7.hp.com`)

The object path so constructed would then look like:

```
CIM_System.CreationClassName="CIM_ComputerSystem",Name="sys7.hp.com"
```

Specifying Empty Propagated Key Properties

As discussed in the provider implementation section on Keys, a short cut may be available when getting an instance of a class that contains propagated keys. Instead of requiring filling in the propagated key properties when obtaining an instance of a scoped class, the client may be able to use the empty string as the value for the propagated keys.

Support for the short cut realizes a performance benefit for both clients and providers. Thus, clients can realize greater interoperability in managing heterogeneous platforms through not needing to specify different key values for each platform. Further, clients do not need to obtain and store key information from a class where there is likely to be

only one instance in the `namespace` (e.g. `CIM_System`) and then use that information in performing a getInstance operation on the scoped class (e.g. CIM_LogicalDevice).

This short cut may not be available with all providers; if it is, this should be identified in the provider's documentation.

Associations and Aggregations

Associations are classes that contain two or more references to instances of other classes. A client uses associations to identify instances of other classes related to an instance of interest. For example a client application could enumerate instances of `CIM_DeviceConnection` (a association between one or more instances of CIM_LogicalDevice) to find instances of connected devices.

Example 5-3 Abridged MOF declaration of CIM_DeviceConnection

```
//
==============================================================
// DeviceConnection
//
==============================================================
   [Association,
Description (
"The DeviceConnection relationship indicates that two or more "
"Device are connected together.") ]
class CIM_DeviceConnection : CIM_Dependency {
[Override ("Antecedent"),
Description ("A LogicalDevice.") ]
CIM_LogicalDevice REF Antecedent;
[Override ("Dependent"),
Description (
"A second LogicalDevice connected to the Antecedent Device.") ]
CIM_LogicalDevice REF Dependent;
. . .
};
```

Aggregations are a special form of assocation. Aggregations have all the attributes of associations, however with the additional semantics that classes scoped to another class via an aggregation make up the scoping class (instead of merely being "associated" with it).

Best Practices

Listed below are several client application "best practices" for navigating schema.

Enumerating DMTF-Defined CIM Classes

Client applications can maximize platform interoperability by using DMTF-defined classes (classes with names usually having the `CIM_` prefix). Clients can successfully use DMTF superclasses, yet still obtain information from platform-specific subclasses, since:

- providers will subclass from the appropriate DTMF classes, and
- the CIM Server will pass enumeration operations to providers for all subclasses.

For example, a general-purpose client could enumerate instances of `CIM_USBPort`, obtaining property values of interest, rather than referencing a particular vendor's subclass (e.g. vendor ACME's USB port, supported by the `ACME_USBPort` subclass of `CIM_USBPort`). When performing one of the enumeration operations, the client must specify appropriate values for the DeepInheritance and LocalOnly flags as discussed below (see also the *DMTF Specification for CIM Operations* for additional details). Thus, when a client performs one of the these operations on a DMTF-defined class, the CIM Server will return all instances of the DMTF-defined class, which include those of any subclasses.

Only those clients that actually plan to use additional properties or methods of a vendor subclass need to write their application specific to the added information in that subclass. Clients may find they can support their management needs with the properties and methods in the DMTF classes only.

Deep Inheritance Flag

The DeepInheritance flag indicates that the enumerateInstances operation should return all properties defined in the designated class and its subclasses. A value of `FALSE` would indicate that none of the

properties defined in subclasses should be returned (just those properties inherited from superclasses and local to the class specified in the operation).

HP WBEM Services only supports the value `TRUE` for the `DeepInheritance` flag.

Local Only Flag

The `LocalOnly` flag indicates that the enumerateInstance and getInstance operations should return only properties defined in the designated class and none of those defined in superclasses. A value of `FALSE` would indicate that those properties defined in superclasses should also be returned. Note that for either value of the `LocalOnly` flag, properties that are overridden in the specified class are returned.

HP WBEM Services only supports the value `FALSE` for the `LocalOnly` flag.

Client Security Considerations

As discussed in the section on *provider security*, the HP WBEM Services for HP-UX CIM Server performs authentication (test for valid user and, if remote, password) and authorization (test for permission to perform an operation on an object in a `namespace`, if enabled) on all requests. Also, the CIM Server can be configured to accept encrypted or unencrypted communication. Clients may need to consider all of these security-related features.

Local vs. Remote Requests and Username/Password Authentication

A local connection mechanism exists for clients to communicate with the CIM Server on the same system. The `connectLocal()` function is used for this purpose, and does not take any arguments. The user ID passed to the provider is that of the process in which the client program is running. The CIM Server verifies that the user ID of the request is indeed that of the requesting process. `namespace` authorization, if enabled, is still performed.

When the client must be able to connect to a CIM Server on a remote system, or when it must be able to specify a different user than that of the process, it must use the `connect()` function. This function allows a hostname and port number to be specified, as well as a username and password.

SSL (Secure Socket Layer) for Encrypted Communication

When a client connects to a remote CIM server, it can specify the port number to which it wishes to connect. As normally configured, HP WBEM Services for HP-UX supports encrypted communication on port 5989 or unencrypted on port 5988. If the client specifies the address of an encrypted port, then it should use the form of `connect()` that takes an SSLContext argument. The SSLContext supplies the information needed by the client to perform a certificate-based authentication transaction with the target host, and causes all communication to be encrypted.

More information on the authentication and encryption algorithms used may be found in the HP WBEM Services for HP-UX System Administrator's Guide on `docs.hp.com/hpux/netsys`.

Since the client may not know in advance which port the target system has been configured to use, it can attempt to connect to one first, then the other. The preferred order would be to try encrypted port 5989 first when connecting to HP-UX systems.

Client Development Best Practices

This section describes several best practices for the development of WBEM client applications.

Using Provider Data Sheets

Provider Data Sheets (PDSs) greatly help an implementer of client applications to understand the available management functionality. Provider developers are expected to publish a PDS for their provider. The Example Provider Data Sheet shows the kind of information typically contained in these data sheets. A PDS contains the following information:

- Provider Overview (Description, Requirements, Release History, Supported Managed Resources, Special Requirements & Dependencies)
- Setting Up This Provider (Installing, Configuring)
- Using This Provider (Schema Supported, Indications Generated, Associations Provided)
- Links To More Information (Additional Provider Documentation, WBEM Information, Managed Resource Information, Client Information, Support Contacts, Migration and Co-existence Information, Possible Provider Enhancements)
- Limitations, Known Defects, and Performance Considerations

Of particular interest are the tables in the section on Using Provider Data Sheets. These tables describe:

- Schema elements (classes, properties, methods) supported
- Property values and data sources (useful in understanding the implementation beyond the general description in the MOF)
- Methods implemented and exceptions that could be thrown

Client implementers should carefully consider the implications of provider implementations on managed platforms. Differences in format and content of the returned results on platforms may have implications on the client application.

Prototyping with wbemexec

When analyzing the appropriate techniques for schema navigation and determining the type of data available on a managed system, it is often useful to prototype the functionality using the general-purpose wbemexec client included with HP WBEM Services for HP-UX. wbemexec takes a `CIM-XML` request from a file or standard input, sends it to a CIM server, receives the response, and displays it to standard output.

Thus, a client developer can encode various CIM operations of interest (e.g. `enumerateInstanceNames`, `getInstance`, or `createInstance`) in XML and observe the CIM Server/provider's response to the operations. The `CIM-XML` requests can be executed against multiple target platforms to get a better understanding of supported schema, data volume, data availability, and provider performance. Revisions can then be made to the information being retrieved and to the way of navigating available schema to improve the client design. Once these adjustments have been made, the client developer can then incorporate the lessons learned from this prototype into the client implementation based on the client APIs.

More information on wbemexec is available in the man page included with HP WBEM Services for HP-UX.

Client Development Use Cases

By way of example, several use cases are described below. These use cases illustrate issues that WBEM client developers may encounter in designing and implementing their applications. A general description and recommended best practices are included with each use case.

General System Information

Description

Some clients only need general system information describing the platform and its running operating system. This general information is often available in particular subclasses of `CIM_System`, each of which probably has just a single instance. Examples include the `CIM_ComputerSystem` (or `CIM_UnitaryComputerSystem`) and `CIM_OperatingSystem` classes.

Special Considerations

The classes which represent general system information may have only single instances. Further, those instances may have many properties which are relatively static, i.e. there may need to be a significant (and probably infrequent) event such as a system reboot or the changing a host name before certain properties on the instance will change. These classes may contain key properties that are propagated to other classes.

In addition, some providers may implement special subclasses with additional properties or methods. It may be helpful to the user of the client application to have a mechanism to access these platform or provider specific properties and methods.

Best Practices

Consider obtaining the system information upon first client contact with a target platform. If the information is relatively static, it is more efficient to avoid repeatedly obtaining the same information.

If the client application displays properties of CIM objects to the user, consider supporting a mechanism whereby users can view additional properties supplied in the subclasses. It may be helpful to enumerate the subclasses of the class of interest using the `enumerateClasses` operation

to see if this case exists. If so, the class can be accessed using the `enumerateInstance` operation with the `DeepInheritance` flag set to `TRUE`. In addition, it may be useful to support some general mechanism for the user to invoke methods specific to subclasses.

Multiple Dynamic Instances

Description

Some clients may need to retrieve dynamic information. For example, process information (`CIM_Process`) is dynamic since processes are frequently created and terminated.

Special Considerations

Client applications may need to update the list of instances between retrievals of the information when instances may be deleted or created dynamically.

Best Practices

It is best to use the `enumerateInstances` operation for these types of classes. Use of `enumerateInstanceNames` followed by `getInstance` would not be guaranteed to return an instance.

Special Purpose Clients

Description

Some special-purpose clients have prior knowledge about a platform based on some simple facts. This may include the value of some special property, the existence of a special-purpose `namespace`, or the existence of some special-purpose subclasses. An example of this is a platform partition management client that uses a special `namespace` containing partition-related classes to manage the platform. Frequently, clients of this type may be released in conjunction with an applicable provider or providers.

Special Considerations

Clients may be able to simplify their discovery processing to identify that the appropriate `namespace` and schema elements are present on the target platform. Also, it may be useful to check the version of the provider to ensure that it can support the client requests.

Best Practices

It is useful to perform appropriate discovery operations during initial connection to a target platform. Also, be careful not to over-simplify error checking.

Command Line Clients

Description

Command line clients can be implemented to make underlying WBEM instrumentation transparent to the end user. WBEM clients can be implemented to replace existing CLIs.

Special Considerations

Because WBEM is based on HTTP, it is possible to implement command line clients that access remote hosts. In such cases, a remote host name will probably be a part of the command line or otherwise furnished to the client.

It is important to consider the security model of the command line client. If the client is on the same host as the CIM Server, the local connection mechanism can be used. However, if the CLI permits a user to access a remote platform, a username and password will need to be supplied for the remote CIM Server to authenticate.

Best Practices

If the command accesses a CIM Server on a remote host, include a command line argument to specify the host to be accessed (managed). Command lines may include the ability to specify a user. However, it is inadvisable to include a command line argument for the password since this argument can show up as plan text in the argument list. Prompting the user for the password or using some authentication brokering scheme are better approaches.

Building Clients

Clients should be compiled and linked with the aCC C++ compiler. The make utility can be used to control this. The following example `Makefile` illustrates the necessary compilation and link flags for building a client.

```
LIBRARIES = -L/opt/wbem/lib -lpegcommon -lpegclient
INCLUDE = -I/opt/wbem/include
ACC_OPTIONS = $(PLATFORM_ACC_OPTIONS) $(INCLUDE)

OUTPUT = $(PROGRAM)
CURRENTPATH = $(PWD)

OBJECTS = $(SOURCES:.cpp=.o)
.cpp.o:
	aCC -c -o $@ $(ACC_OPTIONS) $*.cpp

$(OUTPUT): $(OBJECTS)
	aCC $(ACC_OPTIONS) -O$@ $(OBJECTS) -lrt \
	$(LIBRARIES)
```

In this example, the setting for `PLATFORM_ACC_OPTIONS` depends on the platform:

- For PA, use:

  `+DAportable -AP -mt -DPEGASUS_PLATFORM_HPUX_PARISC_ACC`

- For IA, use:

  `+DD64 -AA -mt -DPEGASUS_PLATFORM_HPUX_IA64_ACC`
  `-DHPUX_IA64_NATIVE_COMPILER`

Sample Clients

The sample client programs included with the SDK illustrate the use of EnumerateInstances and InvokeMethod client APIs. The sample client programs are located in the `/opt/wbem/sample/Clients` directory.

Packaging and Release

A WBEM client program is a conventional, independent HP-UX application, in that HP WBEM Services for HP-UX does not invoke it or depend on it. For this reason, conventional HP-UX application packaging and release processes may be employed without need for special consideration. Extensive information for developers can be found on `software.hp.com`. For example, information about the Software Distributor architecture (SD) may be found at *http://software.hp.com/products/SD_AT_HP/faqs/general.html*.

Additional Sources of Information

Documentation and Standards

- *HP WBEM Services for HP-UX System Administrator's Guide* on *docs.hp.com*
- *DMTF web site* (for WBEM and CIM standards)
- *MOF Specification*
- *CIM Qualifiers*

A CIM Naming Guidelines

A goal of WBEM is to make client access to managed systems as platform- and OS-independent as possible. However, arbitrary choices for naming client-visible elements would preclude identifying consistent information based on names and result in implementation-specific client

code to handle a plethora of implementations.

This section defines naming conventions for CIM `namespace`s, classes, properties, and methods.

namespaces

namespaces are the contexts within which schemas are defined. Within a given namespace, the instances of the classes defined are guaranteed to be unique. Different namespaces may be created to group related information, to define separate views of managed resources, or to limit client authorization to access certain objects. Consistent naming conventions for namespaces enables client applications to easily find the context within which their required management information will be available.

General Syntax

For WBEM-compliant CIM Object Managers (CIM Servers), the namespaceís name is the part of the namespace path within the context of a namespace type (for example http://) and a CIM Server host (for example coyote.rdrunner.com). The full namespace path is the concatenation of the namespaceís type, the CIM Server host, a forward slash (/), and the namespaceís name (for example http://coyote.rdrunner.com/root/ACMEv2).

The namespaceís name has syntax much like that of UNIX directories with their slash-delimited (/) sub-directories. However, namespaces with similar prefixes do not necessarily have hierarchical relationships as in UNIX. Thus, a/b/c can exist without a/b, and deleting namespace e/f does not imply deletion of e/f/g. Also, CIM namespaces are case insensitive, so the names root, Root, and ROOT are equivalent. One exception is WMI, the WBEM implementation for Windows platforms, where namespaces are hierarchical. For the Windows platform, this exception implies that a/b must be created before a/b/c, and successful deletion of a/b requires deletion of a/b/c.

Managed System namespace

A Managed System namespace is the namespace on a managed node wherein a client can expect to find the general system information. This namespace contains DMTF-defined classes and derived subclasses that describe the systemís general purpose managed elements. The classes in

this `namespace` are distinct from those which have are considered to be special purpose, for example those for the purpose of security or some specialized system view.

The name of the Managed System `namespace` corresponds to the context within which the managed objects are placed (for example platform vendor, OS), and the major version of the CIM schema supported. For systems manufactured by the fictitious ACME Corporation, the default `namespace` within which CIM version 2.x derived objects are placed could be `root/ACMEv2`. When CIM version 3.x is released, an additional `namespace` would be created called `root/ACMEv3`.

Windows-based platforms using WMI present an interesting exception in that they all have a `root/CIMv2` `namespace`. For these platforms, the `root/CIMv2` `namespace` is populated with Microsoft-supported operating system objects. However, the `root/ACMEv2` `namespace` should also exist on these platforms for all other ACME-supported objects, ACME extensions to the DMTF CIM schema.

Special Purpose namespaces

In addition to these the `namespace`s supplied with HP WBEM Services for HP-UX, other `namespace`s will need to be defined for special purpose applications. Client management applications may wish to create special `namespace`s to store host-specific information that relates to their application. Providers may specify special-purpose `namespace`s to group special objects that are not for general-purpose use, or to support unique security authorization requirements.

In these cases, the most important guideline is to make the name as descriptive as possible. Putting the special `namespace` in the implied context of a System `namespace` (for example `root/ACMEv2/special`) can assist in clarifying the purpose of the `namespace`. Other examples include:

- `root/ACMEv2/Par` — `namespace` for partition
- `root/ACMEv2/cluster` —`namespace` for a cluster-related objects on an ACME system
- `root/ACMEv2/bigapp` — BigApp-specific objects on an ACME system
- `root/MgmtAppA` — `namespace` for a non-platform specific client (Management App A)

In addition, giving related `namespace`s similar names adds clarity about their associated purposes. For example, `applicationXYZ` could be a `namespace` for objects related to a particular application, and `applicationXYZ/special-auth` could be used for special objects that have additional client authorization constraints.

Classes

Class names are defined within the context of a particular `namespace`. Naming conventions for classes help clarify the managed domain a class is scoped to support. Adhering to the naming conventions also has the added benefit of making it possible for clients to find class extensions across the various platforms and devices supported by a vendor.

DMTF Defined Classes

DMTF has established a naming convention for classes wherein each class is prefixed with `CIM_` (an exception to this is the Support schema classes that begin with `PRS_`). DMTF names then contain one or more descriptive words, each of which is begun with an uppercase letter. Examples include:

- `CIM_ManagedSystemElement`
- `CIM_Service`
- `CIM_ComputerSystem`
- CIM_LogicalDevice
- `CIM_EthernetAdapter`
- `CIM_SoftwareElement`
- `PRS_ServiceRequester`

Where possible it is best to instrument the DMTF classes themselves. However, that said, one must be very careful in implementing a provider based on the DMTF class itself. Implementing a provider for the DMTF `CIM_` class assumes there are no required changes to the class (providers cannot change the definition of a CIM class from a released DMTF definition). In addition, using the DMTF class implies that the provider can enumerate all instances of the given class within the `namespace`. A second provider implementing a subset of the instances as a subclass of that DMTF class could result in unexpected and difficult to interpret results for client applications. Because it is difficult to ensure one would always know all instances of a DMTF-defined class, some conservative providers choose to subclass from the DMTF-defined classes.

Subclassing to Specialize

The normal case for subclassing is when a class represents a subset of instances of its superclass. In these cases, the most important naming convention is to use the most descriptive and precise name possible. Creating intermediate subclasses should be considered if applicable.

One should also consider the potential for definition of additional subclasses in some future schema release. Pitfalls to avoid include:

- Class name too general: a provider of another (parallel) subclass could have instances that overlap with the subclass being defined. In this case a more specific name should have been selected.
- Class name too specific: a provider of another (parallel) subclass would define identical properties. In this case, a more general intermediate subclass should have been defined, and then another subclass derived from it with any specific properties/methods.

When extending the schema, prefixes should reflect the context within which the class is applicable. If a class would be applicable to Linux systems only, a `Linux_` prefix is appropriate. A class specific to all ACME systems would have an `ACME_` prefix. Classes specific to various operating systems supported by the ACME Corporation would be prefixed with `ACMEUnix_`, `ACMELinux_`, and `ACMEWin_`.

An example of subclassing would be an ACME-specific Fast Ethernet provider. The fictitious ACME Corporation could create a subclass of CIM_EthernetPort called `ACME_EthernetPort` that would have ACME-specific properties and methods. Then a subclass of `ACME_EthernetPort` could be called `ACME_FastEthernetPort` that has any attributes specific to a Fast Ethernet device. This would leave open the option of defining a separate subclass of `ACME_EthernetPort` called `ACME_GigabitEthernetPort`.

A second example for use of an intermediate class would be that of kernel parameters for ACME's Unix system. `ACMEUnix_KernelParameter` could be subclassed from `CIM_Setting`. Then from this intermediate class, several additional subclasses could be defined, for example:

- `ACMEUnix_BaseKernelParameter`,
- `ACMEUnix_PrivateBaseKernelParameter`,
- `ACMEUnix_DerivedKernelParameter`, and
- `ACMEUnix_PrivateDerivedKernelParameter`.

Subclassing to Add Properties/Methods

Extending the schema to create a new subclass is also appropriate when adding properties or methods to a DMTF class. In addition, overriding a property or method (such as to include the `write` qualifier on a property) is also a valid reason to create a subclass.

In these cases, the base part of the class name would stay the same, so an ACME-specific subclass of `CIM_EthernetPort` could be called `ACME_EthernetPort`, while the ACME Unix extension to `CIM_OperatingSystem` would be `ACMEUnix_OperatingSystem`.

Properties and Methods

Properties and methods are defined for individual classes and inherited by their subclasses. Property and Method names should be constructed of one or more words, each of which begins with a capital letter. Example property and method names from the DMTF schema include:

- `Name`
- `DeviceID`
- `ErrorDescription`
- `UpperThresholdNonCritical`
- `Range1InputVoltageLow`
- `StartService()`
- `SetPowerState()`

Note that while property or method names may contain numbers, they do not contain underscores or dashes so as not to be confused with class names.

Properties/Methods in Superclasses

Since all properties and methods of superclasses are inherited by the subclass, it is unnecessary to redefine them in the subclass. Thus, there is no opportunity to rename a property or method from a superclass. Properties and methods can be overridden in a subclass (using the Override qualifier), but the override cannot change the name. Overriding can be used to provide a more detailed Description or to add other qualifiers (for example Write) to indicate that additional capabilities exist.

DMTF-defined classes should not be modified to redefine properties or methods. Such a change will adversely affect providers that have been constructed to adhere to the DMTF standard CIM schema.

If a necessary property or method is not in the identified superclass, the following two sections describe recommended approaches.

Examples from Elsewhere with the Schema

When a needed property or method is not defined in a superclass, the best policy is to look elsewhere in the schema for a similar property or method. It is possible that other classes may exist which bear similarities to the class being defined. These other classes may have similar properties or methods that if used in the class being defined could make use of the class more intuitive to the client application user. Further, reusing the property or method name can add clarity and hint at possible code reuse for the client application. Note that reusing names for dissimilar or slightly different purposes can be confusing, so care should be taken to ensure that reuse of the property/method name is beneficial.

Being Descriptive

In the event that a property or method must be defined wherein no other examples apply, it is still necessary to ensure that the name is as descriptive as possible. The property or method name should adhere to the capitalization rules discussed above. Additional information should be supplied via the Description qualifier to clarify use of the property or method. Further, white papers describing the client use model for a schema are an effective tool for documenting the intended class, property, or method use.

B XML Example

The following example shows a request to get the names of instances of the `CIM_ComputerSystem` class (there is generally only one instance), followed by a response containing the name of the system. Items showing important elements of the message are highlighted.

Example B-1 Request sent by client

```
<?xml version="1.0" ?>

<CIM CIMVERSION="2.0" DTDVERSION="2.0">

<MESSAGE ID="3075" PROTOCOLVERSION="1.0">

<SIMPLEREQ>

<IMETHODCALL NAME="EnumerateInstanceNames">

<LOCALnamespacePATH>

<namespace NAME="root"/>

<namespace NAME="cimv2"/>

</LOCALnamespacePATH>

<IPARAMVALUE NAME="CLASSNAME">

<CLASSNAME NAME="CIM_ComputerSystem"/>

</IPARAMVALUE>

</IMETHODCALL>

</SIMPLEREQ>

</MESSAGE>

</CIM>
```

Example B-2 Reply received from CIM Server

```
<?xml version="1.0" encoding="utf-8"?>
<CIM CIMVERSION="2.0" DTDVERSION="2.0">

<MESSAGE ID="3075" PROTOCOLVERSION="1.0">

<SIMPLERSP>

<IMETHODRESPONSE NAME="EnumerateInstanceNames">

<IRETURNVALUE>

<INSTANCENAME CLASSNAME="CIM_ComputerSystem">
```

```
<KEYBINDING NAME="CreationClassName">

<KEYVALUE
VALUETYPE="string">ACME_ComputerSystem</KEYVALUE>

</KEYBINDING>

<KEYBINDING NAME="Name">

<KEYVALUE VALUETYPE="string">idsys.hp.com</KEYVALUE>

</KEYBINDING>

</INSTANCENAME>

...

</IRETURNVALUE>

</IMETHODRESPONSE>

</SIMPLERSP>

</MESSAGE>

</CIM>
```

C Code Example Showing CIM DateTime Conversion

The following code fragment shows how a time value obtained from a library function call can be converted to the CIM `DateTime` format. More

information on the CIM `DateTime` format can be found in the *CIM Specification* on the DMTF web site.

```
#include <time.h>

...

{

...

struct tm *t = localtime(); // localtime() used as an
example

char tmp[26];

sprintf(tmp, "%04d%02d%02d%02d%02d%02d.000000%c%03d",

t->tm_year+1900,

t->tm_mon+1, //HP-UX stores month 0-11

t->tm_mday,

t->tm_hour,

t->tm_min,

t->tm_sec,

(timezone>0)?'-':'+',

labs(timezone/60 - (t->tm_isdst? 60:0)));

CIMDateTime d(tmp);

...

}
```

D Example Provider Data Sheet

HP-UX operating system CIM provider

Provider Overview

Description—The Operating System Provider makes available operating system information such as operating system type, version, last boot up time, local date and time, number of users, swap space size, and free physical memory. The Operating System Provider instruments the `CIM_OperatingSystem` class and `PG_OperatingSystem` subclass. Not all properties of the `CIM_OperatingSystem` are currently supported. This provider does not support the Reboot and Shutdown methods of the `CIM_OperatingSystem` class. The `PG_OperatingSystem` subclass adds the `SystemUpTime` and `OperatingSystemCapability` properties.

- `SystemUpTime` is a convenience property. It provides direct access to this value, versus having client/providers calculate the value from `LastBootUpTime` and the `LocalDateTime`.
- OperatingSystemCapability indicates whether the OS itself is 32-bit or 64-bit capable.

This provider is for use by clients as part of a basic understanding of the identity of the managed system on which it is running (typically a server or appliance). The current implementation is for HP-UX only.

Requirements—The provider requires HP WBEM Services for HP-UX.

Release history—Initial release with HP WBEM Services for HP-UX, version 1.0. Re-released with HP WBEM Services for HP-UX, version 1.1. Re-released with HP WBEM Services for HP-UX, version 1.5.

Supported managed resources—Managed systems running HP WBEM Services for HP-UX.

Setting Up This Provider

Installing this provider—The Operating System Provider is contained within an operating system provider module (it is the only provider within that module). The provider module is bundled with HP WBEM Services for HP-UX including:

- Schema MOF file: `PG_OperatingSystem20.mof` (in `/etc/opt/wbem/mof`)
- Provider registration MOF file: `PG_OperatingSystem20R.mof` (in `/etc/opt/wbem/mof`)
- Provider module executable: `libOSProvider.sl` (in `/opt/wbem/providers/lib`, and is the target of the `/opt/wbem/lib/libOSProvider.1` symbolic link)

The HP-UX Operating System Provider is registered to support the `root/cimv2 namespace` as an instance provider. There are no special installation instructions; the provider will be installed by default with HP WBEM Services for HP-UX.

Configuring this provider—This provider does not accept specific configuration adjustments (beyond standard WBEM support).

Using This Provider

Schema supported by this provider—This provider supports the CIM_OperatingSystem and PG_OperatingSystem classes. Tables 1 through 4 describe the properties and methods supported by the provider.

Table 1 describes the properties of the CIM_OperatingSystem class. It has three columns. The first is the property name (including type and units), the second is the property inheritance (indicating which class or superclass defines the property), and the third is the propertyís value and data source. Each row describes a property.

Table D-1 **CIM_OperatingSystem Properties**

Property Name	Property Inheritance	Property Value (and data source)
string Caption	Inherited from CIM_ManagedElement	HP-UX: Returns the string "The current Operating System".
string Description	Inherited from CIM_ManagedElement	HP-UX: Returns the string "This instance reflects the Operating System on which the CIM Server is executing (as distinguished from instances of other installed operating systems that could be run)."
datetime InstallDate	Inherited from CIM_ManagedSystemElement	HP-UX: Not implemented.

Table D-1 `CIM_OperatingSystem` **Properties (Continued)**

Property Name	Property Inheritance	Property Value (and data source)
string `Status`	Inherited from `CIM_ManagedSystemElement`	HP-UX: Returns "Unknown" (although if the OS is running enough for CIM Server to function, may consider "OK", but would need to distinguish it from "Degraded" or "Stopping").
string `CSCreationClassName` [Key]	Local to `CIM_OperatingSystem`	HP-UX: Always returns string "CIM_UnitaryComputerSystem".
string `CSName` [Key]	Local to `CIM_OperatingSystem`	HP-UX: Returns the system name (fully qualified if possible). Gets the fully qualified system name from the `gethostbyname()` system call, or if that fails, uses the hostname obtained from the `gethostname()` system call.
string `CreationClassName` [Key]	Local to `CIM_OperatingSystem`	HP-UX: Returns "`CIM_OperatingSystem`" (even for PG_OS, since actually the same instance, thus the same keys, just additional properties).

Table D-1 `CIM_OperatingSystem` **Properties (Continued)**

Property Name	Property Inheritance	Property Value (and data source)
string Name [Key]	Inherited from CIM_ManagedSystemElement (and made one of 4 keys of CIM_OperatingSystem)	HP-UX: Returns the name of this system from the sysname field of the uname() system call.
uint16 OSType	Local to CIM_OperatingSystem	HP-UX: Returns 8 (defined by DMTF as the value to set for HP-UX).
string OtherTypeDescription	Local to CIM_OperatingSystem	HP-UX: Not implemented
string Version	Local to CIM_OperatingSystem	HP-UX: Returns the release information from the uname() system call (for example, B.11.20).
datetime LastBootUpTime	Local to CIM_OperatingSystem	HP-UX: Returns boot_time field from pstat() system call (local time with GMT offset).
datetime LocalDateTime	Local to CIM_OperatingSystem	HP-UX: Returns information from time() system call (local time with GMT offset).
uint16 CurrentTimeZone	Local to CIM_OperatingSystem	HP-UX: Returns information from time() system call (local time with GMT offset)

Table D-1 `CIM_OperatingSystem` Properties (Continued)

Property Name	Property Inheritance	Property Value (and data source)
Uint32 `NumberOfLicensedUsers`	Local to `CIM_OperatingSystem`	HP-UX: Returns number of licensed users from interpreting the version string of the `uname()` system call (0 if unlimited or 128 or 256).
Uint32 `NumberOfUsers`	Local to `CIM_OperatingSystem`	HP-UX: Returns the number of `ut_usrs` from `getutent` calls.
Uint32 `NumberOfProcesses`	Local to `CIM_OperatingSystem`	HP-UX: Returns the number of active processes from the `pstat()` system call.
Uint32 `MaxNumberOfProcesses`	Local to `CIM_OperatingSystem`	HP-UX: Returns `max_proc` from the `pstat()` system call.
Uint64 `TotalSwapSpaceSize` (in KiloBytes)	Local to `CIM_OperatingSystem`	HP-UX: Returns the sum of KB available across all swap from the swapinfo — q command.
Uint64 `TotalVirutalMemorySize` (in KiloBytes)	Local to `CIM_OperatingSystem`	HP-UX: Returns the sum of KB available across all swap from the swapinfo — q command (same as SwapSize).
Uint64 `FreeVirtualMemory` (in KiloBytes)	Local to `CIM_OperatingSystem`	HP-UX: Returns the sum of KB free across all swap from the swapinfo — at command.

Table D-1 `CIM_OperatingSystem` **Properties (Continued)**

Property Name	Property Inheritance	Property Value (and data source)
Uint64 FreePhysicalMemory (in KiloBytes)	Local to CIM_OperatingSystem	HP-UX: Returns information from the pstat() system call (psd_free).
Uint64 TotalVisibleMemorySize (in KiloBytes)	Local to CIM_OperatingSystem	HP-UX: Returns information from the pstat() system call; physical memory adjusted for page size.
Uint64 SizeStoredInPagingFiles (in KiloBytes)	Local to CIM_OperatingSystem	HP-UX: Returns 0 (no paging files).
Uint64 FreeSpaceInPagingFiles (in KiloBytes)	Local to CIM_OperatingSystem	HP-UX: Returns 0 (no paging files).
Uint64 MaxProcessMemorySize (in KiloBytes)	Local to CIM_OperatingSystem	HP-UX: Returns the sum of the data, text, and stack sizes retrieved via the kmtune command or gettuneinfo() call.
boolean Distributed	Local to CIM_OperatingSystem	HP-UX: Returns FALSE.
Uint32 NumberOfUsers	Local to CIM_OperatingSystem	HP-UX: Returns the number of ut_usrs from getutent calls.

Table D-1 `CIM_OperatingSystem` **Properties (Continued)**

Property Name	Property Inheritance	Property Value (and data source)
Uint32 `MaxProcessesPerUser`	Local to `CIM_OperatingSystem`	HP-UX: Returns the `_SC_CHILD_MAX` value from `sysconf()` system call.

Table 2 describes the properties of the `PG_OperatingSystem` class. It has three columns. The first is the property name (including type and units), the second is the property inheritance (indicating which class or superclass defines the property), and the third is the propertyís value or data source. Each row describes a property. The `PG_OperatingSystem` class inherits properties of superclass `CIM_OperatingSystem` (as described in Table 1 and not repeated here).

Table D-2 **PG_OperatingSystem properties**

Property Name	Property Inheritance	Property Value (and data source)
Uint64 `SystemUpTime` (in seconds)	Local to `PG_OperatingSystem`	HP-UX: Returns a value calculated from current time and boot up time.
string `OperatingSystemCapability`	Local to `PG_OperatingSystem`	
Defined as a String (vs. enum) for maximum flexibility across OS capability representations.	HP-UX: Returns "32 bit" or "64 bit" (number of bits used by the kernel for pointers).	

Tables 3 and 4 describe the intrinsic and extrinsic methods for `CIM_OperatingSystem` and inherited by `PG_OperatingSystem`. There are no local methods within `PG_OperatingSystem`. The `GetInstance` operation is supported on both classes (returning only CIM_ `properties` for `CIM_OperatingSystem`). Enumerate operations only return instances

on the PG_ subclass (as the CIM Server will invoke subclass providers on enumerations). The provider current registers as an instance provider and a method provider.

Table 3 describes the intrinsic methods supported by this provider. It has three columns. The first is the method name, the second is a description of the provideris actions based on invoking that method, and the third is a list of any exceptions that could result from invoking the method. Each row describes a method.

Table D-3 **intrinsic methods for CIM_OperatingSystem and PG_OperatingSystem**

Method Name	Description	Exceptions Thrown
EnumerateInstances	Returns all instances of class (expect one unless additional installed OSs) with all properties.	None
EnumerateInstanceNames	Returns object path of all instances of class (expect one unless additional installed OSs) with key properties.	None
GetInstance	Returns the requested instance (with all properties).	CIM_ERR_INVALID_PARAMETER
(if wrong class or wrong number of keys or wrong keys)		
CIM_ERR_NOT_FOUND (from CIM Server) if no instance.		
ModifyInstance		
Returns Not Supported.	CIM_ERR_NOT_SUPPORTED	

Table D-3 **intrinsic methods for `CIM_OperatingSystem` and PG_OperatingSystem (Continued)**

Method Name	Description	Exceptions Thrown
`DeleteInstance`	Returns Not Supported.	`CIM_ERR_NOT_SUPPORTED`
Initialize	Not Supported.	None
Terminate	Not Supported.	None
CreateInstance	Returns Not Supported.	`CIM_ERR_NOT_SUPPORTED`

Table 4 describes the extrinsic methods supported by this provider. It has three columns. The first is the method name, the second is a description of the provideris actions based on invoking that method, and the third is a list of any exceptions that could result from invoking the method. Each row describes a method.

Table D-4 **extrinsic methods for `CIM_OperatingSystem`**

Method Name	Description	Exceptions Thrown
Reboot	Reboot the operating system	HP-UX: Not implemented
Shutdown	Shutdown the operating system	HP-UX: Not implemented

- Indications generated by this provider

  This provider does not currently generate any indications.

- Associations provided by this provider

  This provider does not currently support any associations.

Links to More Information

- Additional provider documentation

  There is currently no additional documentation for this provider beyond this information.

  See also man pages for information on the various commands and system calls noted in the descriptions above.

- WBEM information

  For a CIM tutorial, go to `http://www.dmtf.org/education/cimtutorial.php`.

  For information about HP WBEM Services for HP-UX, see software.hp.com and hp.com (the Network and Systems Management areas).

- Managed resource documentation

  Information regarding the HP-UX operating system can be found in the man pages and manuals of that operating system.

- Client information

  The osinfo command bundled with HP WBEM Services for HP-UX is a client making use of the Operating System Provider.

- Support contacts

  The HP-UX Operating System Provider is supported as part of HP WBEM Services for HP-UX.

E Factors for Estimating Provider Effort

This section summarizes the factors to consider when determining how much time it will take to design, implement, test and release a provider.

This is just a tool to help with estimates. Providers can take as little as two weeks to as long as several months depending on a variety of factors. Each item below adds time to the schedule. To get a very rough estimate of provider effort, answer the questions below and add the days together.

- Schema complexity

  Defining the schema can be very simple or can be very complex and time consuming. The effort required for schema is one of the most variable parts of writing a provider.

  — Schema defined and solid (e.g. implementing existing CIM Classes with no changes) - 1 Day

  — Schema defined but may change slightly - 3 Days

  — Schema not defined and/or requires coordination with other groups and/or other companies - ranges greatly

- Number and kind of operations implemented

  Some operations require a lot more work (for example write operations in Instance Providers, and Association, Query, and Indication Providers).

  — No difficult operations - 0 Days

  — Otherwise, could be up to 4-6 weeks extra

- Number of properties and extrinsic methods

  The more of these, the more code needs to be written, tested, and documented.

  — < 10 - 3 Days

  — < 25 - 6 Days

  — < 50 - 10 Days

  — more - ranges greatly

- Stability and definition of how to get the information

  If you know how to get the information (via APIs, system calls, CLIs, or data structure access) and you don't have to test usage of the calls, provider development will be much quicker.

  — Know exactly how to get the information and trust usage of the call - 0 Days

- — Don't know exactly how to get the information but someone else does and can help you - 3 Days
- — Have to test the calls or don't know how to get the information and no one else does either - ranges greatly

- Static vs. dynamic results and multiple instances

  Will the properties always for the same every time and everywhere you run the provider or will it always be different? Testing is much simpler if you know what the value of each property should be and it is easier to automate the validation of the value. If the property values are not the same for each test (for example, a value like time), it will take more effort to automate a test for validating the value. (For example, supporting acceptable deltas between returned and validation values) In addition, if your provider creates multiple instances, it takes more time to select a good sample of instances to test and then test.

  - — All the properties always return the same value on all systems (for example OS type) - 1 Day
  - — Properties return the same value each time the call is made on the same system configured in the same way (for example, system name) - 2 Days
  - — No guarantee ñ results can vary each time ñ need to make the call separately to validate results (for example, time) - 4 Days to write test client
  - — Provider creates multiple instances - 5 Days

- Is there existing code or a template that is specific to your provider

  Provider development will be much easier the more you can leverage from other's work. If you have example code such as DMI code that already gets the information, it will take less time to write a provider.

  - — Have existing code that is specific to provider (skeleton for creating provider) and code on how to get the information - 0 Days
  - — Have samples but they aren't specific to provider (samples from dev kit are similar enough to use as starting point) - 2 Days
  - — Provider will do something very different then what is in samples - 5 Days

- Number of different interfaces, operating systems and platforms that will be supported

  If you are supporting Windows, Linux and HP-UX on PA-RISC and IPF, you will have much more work then if you have one set of calls to make. If you have multiple interfaces required for the same information (for example, across OS types and versions and different hardware), it will take longer to code and test.

  — Will support one set of interface calls - 0 Days

  — Can get the information mostly through one set of interface calls - 1 Day

  — Will support two similar interface calls - 2 Days

  — Will support different interface calls (like on windows and Linux) - ranges greatly

- Amount of documentation needed

  If you need more than a provider data sheet for your provider, you need to factor in more time.

  — Just provider data sheet needed - 2-5 Days

  — More documentation needed, work with your documentation experts to estimate

- Delivery mechanism

  If you have a defined and simple delivery process then it will be faster. We can not give estimates on how much effort is required for your delivery process since it ranges greatly.

  — Just submit to open source - .5 Days

  — Other delivery mechanism - ranges greatly

- Packaging

  What is required to package a product (i.e., getting product numbers, building depots, setting up Build Environments, creating SD scripts).

  — Use tar package or some other simply bundle - .5 Days

  — Use SD - 2 Days

  — Other - ranges greatly

- Other factors to consider

Testing, quality goals, robustness, equipment availability, dependencies, high availability requirements, experience with WBEM, C++, HP WBEM Services for HP-UX.

F Glossary

Common Information Model (CIM)

A hierarchical object-based model developed by the DMTF that defines a large number of concepts common to most computer systems. The CIM specification can be found on the DMTF web site.

CIM Client

An application that issues CIM operation requests to a CIM server.

CIM Object Manager (CIM Server)

The CIM Server manages the definitions of CIM objects, and directs the flow of information between clients and providers in a CIM server.

CIM Operations

A set of operations, specified by the DMTF, that can be requested of a CIM server to be performed on CIM objects. The specification can be found on the DMTF web site.

CIM Schema

A collection of class definitions used to represent managed objects that exist in every management environment.

CIM server

A CIM server is a server that processes requests for operations on CIM objects. It can have an HTTP server component that communicates with clients using the HTTP protocol, transfering messages encoded in xmlCIM. It can also have a CIM Server that distributes requests to providers for translation to platform-specific operations.

Class

A collection of instances, all of which support a common type. A set of properties and methods. The common properties and methods are defined as features of the class. For example, the class called Modem represents all the modems present in a system.

Core Model

A subset of CIM, not specific to any platform. The Core model is set of classes and associations that establish a conceptual framework for the schema of the rest of the managed environment. Systems, applications, networks and related information are modeled as extensions to the Core model.

Distributed Management Task Force (DMTF)

An industry wide standards organization committed to making computing resources and environments easier to use, understand, configure, and manage. (web site http://www.dmtf.org/)

Extensible Markup Language (XML)

A simplified subset of SGML that offers powerful and extensible data modeling capabilities. An XML Document is a collection of data represented in XML. An XML Schema is a grammar that describes the structure of an XML Document.

Extrinsic Method

A method defined on a CIM Class in some Schema that is unique to that class (versus intrinsic methods which apply across all classes). See also intrinsic method.

HTTP

(Hypertext Transfer Protocol) An application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic stateless protocol that can be used for many tasks through extensions of its request methods, error codes and headers.

Instance

A representation of a real-world managed object that belongs to a particular class, or a particular occurrence of an event.

Intrinsic Method

A method defined for the purpose of modeling a CIM operation. Standard intrinsic methods (such as enumerateInstances, getInstance, modifyInstance) are relevant to all classes. See also extrinsic method.

Itanium Processor Family (Itanium-2™)

Also called IPF or IA-64. Systems based on Intel's new 64-bit processor architecture.

Lightweight HTTP Server

A small footprint server that processes HTTP requests and returns standard HTTP responses. The server is not intended as a replacement for a web server. The server does not serve up HTML web pages and does not run CGI applications. .

Managed object

The actual item in the system environment that is accessed by the provider. For example, a Network Interface Card, an Operating System kernel parameter, a system user, a print spooler.

Managed Object Format (MOF)

A compiled language for defining classes and instances. The MOF compiler compiles .mof text files, adding the data to the CIM Object Manager Repository. MOF eliminates the need to write code, thus providing a simple and fast technique for modifying the CIM Object Manager Repository. DMTF makes their schemas available as MOF files.

Management Application

An application or service that uses information or request operations from one or more managed objects in a managed environment. Management applications retrieve this information and request operations through calls to the CIM Object Manager API from the CIM Object Manager and from providers.

MOF File

A text file that contains definitions of classes and instances using the Managed Object Format (MOF) language. Such files can be used to load descriptions of schemas supported by providers (via the cimmof command).

Multiple Operation

A CIM request that requires the invocation of more than one method.

namespace

A directory-like structure that can contain classes, instances, and other `namespace`s. Objects are located within a `namespace`, and have unique names (specified by one or more key values) within that `namespace`. Objects in different `namespace`s can have the same keys, yet are unique since they reside in separate `namespace`s. Access to a `namespace` can be restricted to an authorized set of users.

Provider

An executable that can return/set information, execute methods, generate indications, or respond to other requests regarding a given managed object.

Provider Data Sheet (PDS)

Provides basic provider information to software professionals who will design, implement, enhance, and/or support client applications that will use this provider. It contains information about what this provider does, what interfaces it uses, how to install it and what platforms and operating systems are supported. An example and template are included in this manual.

Provider Registration

A provider must be registered with the CIM Server so that the CIM Server will know what properties and methods it supports. A special object is created during registration to relate the information about the provider to the classes in the CIM schema that the provider supports. Refer to the section on Provider Registration and Naming for detailed information.

Repository

The CIM Server repository contains the definitions of classes and instances that describe managed objects and the relationships among them. The repository is not available for use by clients or providers for static or persistent data storage.

SAN-aware

Storage Area Network aware.

Schema

A collection of class definitions that describe managed objects in a particular environment.

Simple Operation

A CIM request that requires the invocation of a single method.

SOHO

"Small Office/Home Office", a class of products which are typically more powerful and more expensive than those sold to consumers, but smaller than those generally used by large institutions.

Unified Modeling Language (UML)

More information can be found on the Object Management Group web site at http://www.omg.org/. A UML diagram is used to visualize an object-oriented concept such as the hierarchy of CIM classes, where each box represents a class of object. The arrows may be thought of as meaning "is-a-kind-of", so that a Human is a kind of Primate. Humans,

being Primates, have all of the Attributes (or Properties) or Primates, but have additional properties, such as Nationality. By convention, in a UML diagram of CIM classes, only properties specific to a class, and not those inherited from the parent class and those above, are shown. But the inherited properties are also understood to be present.

Web-Based Enterprise Management (WBEM)

A standard developed by the DMTF that defines network protocols for the communication of CIM objects and operations. WBEM is a set of management and Internet standard technologies developed to unify the management of enterprise computing environments.

Web Server

Web servers act as HTTP servers, but in addition they enable a wide variety of other capabilities (e.g., running CGI scripts). The distinction between an HTTP server and a Web server is critical to understanding the security implications of HP WBEM Services for HP-UX. HP WBEM Services for HP-UX uses an embedded HTTP server not a web server.

xmlCIM

A specification for the CIM document type, a specialization of the eXtensible Markup Language (XML) that describes how CIM objects and operations should be encoded using XML for communication over a network. The full xmlCIM specification can be found on the DMTF web site.