*Open Group Technical Standard*

# Systems Management: Common Information Model (CIM)

*The Open Group*

Open Group Technical Standard

Systems Management: Common Information Model (CIM)

ISBN: 1-85912-255-8
Document Number: C804

Published in the U.K. by The Open Group, August 1998.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

# Contents

## List of Figures

## List of Tables

# Preface

**The Open Group**

The Open Group is the leading vendor-neutral, international consortium for buyers and suppliers of technology. Its mission is to cause the development of a viable global information infrastructure that is ubiquitous, trusted, reliable, and as easy-to-use as the telephone. The essential functionality embedded in this infrastructure is what we term the *IT DialTone*. The Open Group creates an environment where all elements involved in technology development can cooperate to deliver less costly and more flexible IT solutions.

Formed in 1996 by the merger of the X/Open Company Ltd. (founded in 1984) and the Open Software Foundation (founded in 1988), The Open Group is supported by most of the world's largest user organizations, information systems vendors, and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to meet its new mission, as well as to assist user organizations, vendors, and suppliers in the development and implementation of products supporting the adoption and proliferation of systems which conform to standard specifications.

With more than 200 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- Consolidating, prioritizing, and communicating customer requirements to vendors
- Conducting research and development with industry, academia, and government agencies to deliver innovation and economy through projects associated with its Research Institute
- Managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- Adopting, integrating, and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- Licensing and promoting the Open Brand, represented by the ''X'' Device, that designates vendor products which conform to Open Group Product Standards
- Promoting the benefits of the IT DialTone to customers, vendors, and the public

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development, and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trademark on behalf of the industry.

## Development of Product Standards

This process includes the identification of requirements for open systems and, now, the IT DialTone, development of Technical Standards (formerly CAE and Preliminary Specifications) through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product.

The ‘‘X’’ Device is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the Open Brand they guarantee, through the Open Brand Trade Mark License Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

## Open Group Publications

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical Standards and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys, and business titles.

There are several types of specification:

- *Technical Standards* (formerly *CAE Specifications*)

  The Open Group Technical Standards form the basis for our Product Standards. These Standards are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement a Technical Standard can enjoy the benefits of a single, widely supported industry standard. Where appropriate, they can demonstrate product compliance through the Open Brand. Technical Standards are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

- *CAE Specifications*

  CAE Specifications and Developers’ Specifications published prior to January 1998 have the same status as Technical Standards (see above).

- *Preliminary Specifications*

  Preliminary Specifications have usually addressed an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is as stable as can be achieved, through applying The Open Group’s rigorous development and review procedures.

  Preliminary Specifications are analogous to the *trial-use* standards issued by formal standards organizations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a Technical Standard. While the intent is to progress Preliminary Specifications to corresponding Technical Standards, the ability to do so depends on consensus among Open Group members.

- *Consortium and Technology Specifications*

  The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif, and CDE.

  Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as Technical Standards, in which case the relevant Technology Specification is superseded by a Technical Standard.

In addition, The Open Group publishes:

- *Product Documentation*

  This includes product documentation—programmer's guides, user manuals, and so on—relating to the Pre-structured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

- *Guides*

  These provide information that is useful in the evaluation, procurement, development, or management of open systems, particularly those that relate to the Technical Standards or Preliminary Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

- *Technical Studies*

  Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Program. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

## Versions and Issues of Specifications

As with all *live* documents, Technical Standards and Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.
- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

## Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at **http://www.opengroup.org/corrigenda**.

## Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at **http://www.opengroup.org/pubs**.

## This Document

This Technical Standard is a joint publication with the Desktop Management Task force. The Common Information Model (CIM) is an approach to the management of systems and networks that applies the basic structuring and conceptualization techniques of the object-oriented paradigm. This approach uses a uniform modeling formalism that, together with the basic repertoire of object-oriented constructs, supports the cooperative development of an object-oriented schema across multiple organizations. This enables sharing of information in systems and across networks.

CIM consists of a language definition (this document), that describes the various constructs and techniques used to model resources, together with a set of schema that describe how specific resources are represented. The set of schema are provided to establish a common framework that describes the managed environment. The management schema consists of:

- a Core Schema, applicable to all areas of management
- a Common Schema, applicable to particular areas of management, including systems, applications, databases, networks, and devices
- Extension Schema, each extension schema being applicable to a technology-specific extension, such as a particular operating system implementation.

## Revision History

This Technical Standard is the formally adopted Open Group publication of the DMTF CIM version 2.0 specification.

It includes the revisions specified in DMTF Errata 2.0.2, dated 26 June 1998.

## Audience

This specification is intended for implementers of the Common Information Model (CIM) for sharing of information in systems and across networks.

## Structure

- **Chapter 1** provides an introduction and overview of the CIM conceptual model. its use to exchange management information, and key issues concerning conformance of CIM implementations to this Technical Standard.
- **Chapter 2** describes the CIM meta schema.
- **Chapter 3** describes the Managed Object Format (MOF).
- **Chapter 4** describes the MOF components.
- **Chapter 5** describes the CIM naming mechanism, which facilitates enterprise-wide identification of objects, as well as the sharing of management information.
- **Chapter 6** describes mappings from existing MIF syntax schemes to the CIM MOF syntax.
- **Chapter 7** describes the basic nature of repositories, and how these can be exploited by CIM.
- **Appendix A** describes the MOF syntax grammar.
- **Appendix B** defines the CIM meta schema.
- **Appendix C** defines values for the UNITS qualifier.

- **Appendix D** describes the Unified Modeling language (UML) notation used by CIM.
- **Appendix E** describes the Unicode usage in CIM.
- **Appendix F** gives guidelines for implementation of CIM.

A Glossary and Index are also provided.

# *Trademarks*

# Acknowledgements

This Technical Standard is the formally adopted Open Group publication of the DMTF CIM version 2.0 specification.

The Open Group acknowledges the work of the Desktop Management Task Force (DMTF) Common Information Model (CIM) project members of the Desktop Management Task Force (DMTF), in their development of the CIM version 2.0 specification.

The following companies provided the members of the DMTF CIM project team:

Compaq Computer Corp.
Computer Associates International, Inc.
Hewlett-Packard Company
Intel Corporation
Microsoft Corporation
Novell, Inc.
Sun Microsystems, Inc.
Tivoli Systems, Inc.

The Open Group acknowledges the leadership and technical editing work of Raymond Williams (Tivoli Systems) in the DMTF CIM project.

# Referenced Documents

The following documents are referenced in this Technical Standard:

DCE RPC

CAE Specification, August 1994, X/Open DCE: Remote Procedure Call (ISBN: 1-85912-041-5, C309), published by The Open Group.

This specification is now also ISO International Standard ISO/IEC 11578:1996, Information technology — Open Systems Interconnection — Remote Procedure Call (RPC)

UTF-8

CAE Specification, April 1995, File System Safe UCS Transformation Format (UTF-8) (ISBN: 1-85912-082-2, C501), published by The Open Group.

UNICODE Standard, Version 2

The Unicode Consortium, The Unicode Standard, Worldwide Character Encoding Version 2.0, Volume One, Addison-Wesley, 1996.

ANSI/IEEE Std 754-1985

Standard for Binary Floating-Point Arithmetic.

ISO 639

ISO 639: 1988, Codes for the Representation of Names of Languages, Bilingual edition.

RFC 2234

Augmented BNF (Backus-Naur Form) for Syntax Specifications: ABNF, November 1997.

RFC 2279

UTF-8 — a Universal Transformation Format for multi-octet characters defined by ISO/IEC 10646 (UCS — Universal Character Set), January 1998.

*Chapter 1*

# *Introduction*

There are many ways in which the Common Information Model (CIM) can be used. This introductory Chapter provides a context in which the details described in subsequent Chapters and Appendices can be understood.

## 1.1 Overview

Ideally, information used to perform tasks is organized or structured to allow disparate groups of people to use it. This can be accomplished by developing a model or representation of the details required by people working within a particular domain. Such an approach can be referred to as an information model.

An information model requires a set of legal statement types or syntax to capture the representation, and a collection of actual expressions necessary to manage common aspects of the domain (in this case, complex computer systems).

Because of the focus on common aspects, this information model is referred to as the Common Information Model (CIM).

This document describes an object-oriented metamodel based on the Unified Modeling Language (UML). This model includes expressions for common elements that must be clearly presented to management applications (for example, object classes, properties, methods and associations). This document does not describe specific CIM implementations, APIs, or communication protocols.

Further development work on CIM is planned by the Desktop Management Task Force (DMTF) CIM Technical Development Committee. Up-to-date information on this work may be found at their Web site, at http://www.dmtf.org/work/cim.html.

## 1.2 CIM Management Schema

Management schemas are the building blocks for management platforms and management applications, such as device configuration, performance management, and change management. CIM is structured in such a way that the managed environment can be seen as a collection of interrelated systems, each of which is composed of a number of discrete elements.

CIM supplies a set of classes with properties and associations that provide a well-understood conceptual framework within which it is possible to organize the available information about the managed environment. It is assumed that CIM will be clearly understood by any programmer required to write code that will operate against the object schema, or by any schema designer intending to make new information available within the managed environment.

CIM itself is structured into three distinct layers:

- Core model — an information model that captures notions that are applicable to all areas of management.
- Common model — an information model that captures notions that are common to particular management areas, but independent of a particular technology or implementation. The common areas are systems, applications, networks and devices. The information model

is specific enough to provide a basis for the development of management applications. This schema provides a set of base classes for extension into the area of technology-specific schemas. The Core and Common models together are referred to in this document as the CIM schema.

- Extension schemas — represent technology-specific extensions of the Common model. These schemas are specific to environments, such as operating systems (for example, UNIX or Microsoft Windows).

Development of CIM schema is being undertaken as a continuing activity that of necessity has to follow behind the definition of the CIM language described in this document. The current set of approved schema will be referenced from the on-line version of this specification, which can be found at http://www.opengroup.org/pubs/catalog/c804.htm

At the time of publication it has not been determined whether the management schema will be made available in printed form.

### 1.2.1 Core Model

The Core model is a small set of classes, associations and properties that provide a basic vocabulary for analyzing and describing managed systems. The Core model represents a starting point for the analyst in determining how to extend the common schema. While it is possible that additional classes will be added to the Core model over time, major re-interpretations of the Core model classes are not anticipated.

### 1.2.2 Common Model

The Common model is a basic set of classes that define various technology-independent areas. These areas are:

- Systems
- Applications
- Networks
- Devices

The classes, properties, associations and methods in the Common model are intended to provide a view of the area that is detailed enough to use as a basis for program design and, in some cases, implementation.

Extensions are added below the Common model, in platform-specific additions that supply concrete classes and implementations of the Common model classes. As new extensions become available, the Common model will offer a broader range of information.

### 1.2.3 Extension Schema

The Extension schemas are technology-specific extensions to the Common model. It is expected that the Common model will evolve as a result of the promotion of objects and properties defined in the Extension schemas.

## 1.3 CIM Implementations

CIM is a conceptual model that is not bound to a particular implementation. This allows it to be used to exchange management information in a variety of ways. Four of these ways are illustrated in Figure 1-1, and described below. It is possible to use these ways in combination within a management application.

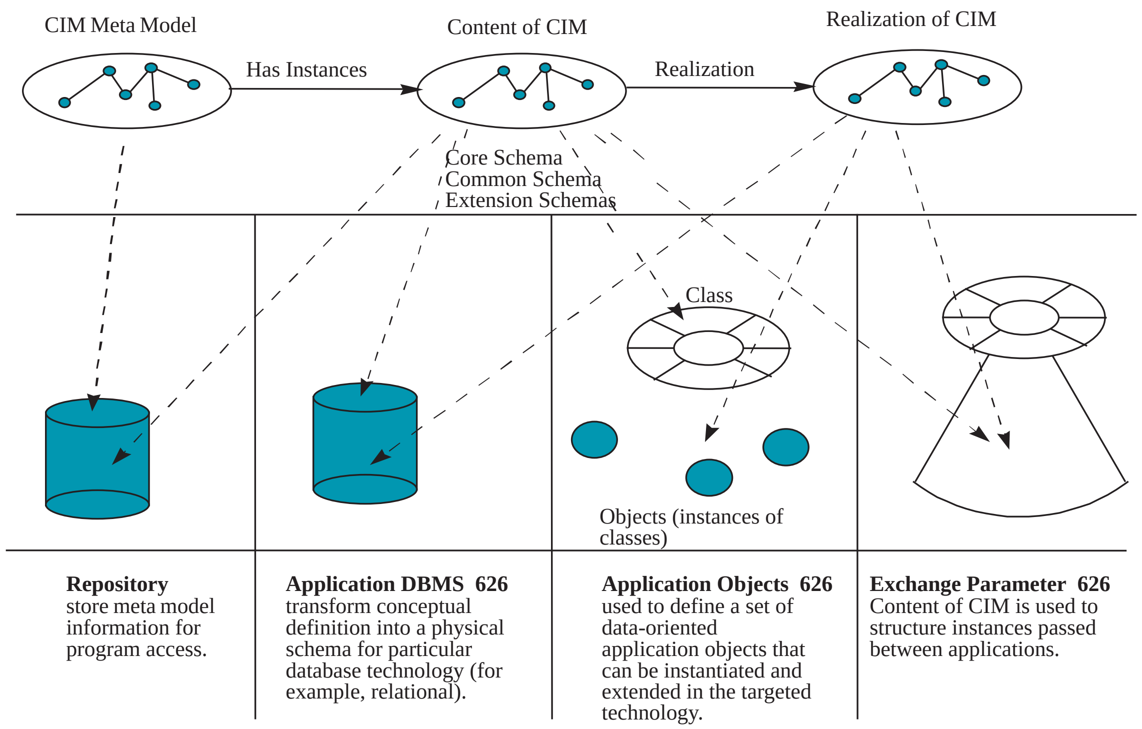

CIM Meta Model
Has Instances
Content of CIM
Realization
Realization of CIM
Core Schema
Common Schema
Extension Schemas
Class
Objects (instances of classes)
**Repository**
store meta model information for program access.
**Application DBMS 626**
transform conceptual definition into a physical schema for particular database technology (for example, relational).
**Application Objects 626**
used to define a set of data-oriented application objects that can be instantiated and extended in the targeted technology.
**Exchange Parameter 626**
Content of CIM is used to structure instances passed between applications.

**Figure 1-1** Four Ways to Use CIM

As a repository, the constructs defined in the model are stored in a database. These constructs are not instances of the object, relationship, and so on; but rather are definitions for someone to use in establishing objects and relationships. The metamodel used by CIM is stored in a repository that becomes a representation of the metamodel. This is accomplished by mapping the metamodel constructs into the physical schema of the targeted repository, and then populating it with the classes and properties expressed in the Core schema, Common schema, and Extension schemas.

For an application Data Base Management System (DBMS), the CIM is mapped into the physical schema of a targeted DBMS (for example, relational). The information stored in the database consists of actual instances of the constructs. Applications can exchange information when they have access to a common DBMS and the mapping occurs in a predictable way.

For application objects, the CIM is used to create a set of application objects in a particular language. Applications can exchange information when they can bind to the application objects.

For exchange parameters, the CIM (expressed in some agreed-to syntax) is a neutral form used to exchange management information by way of a standard set of object APIs. The exchange can be accomplished via a direct set of API calls or it can be accomplished by exchange oriented API which can create the appropriate object in the local implementation technology.

## 1.4 Conformance

The ability to exchange information between management applications is fundamental to CIM. The current mechanism for exchanging management information is the Management Object Format (MOF). At the present time, no programming interfaces or protocols are defined by this CIM document, and hence t does not provide an exchange mechanism. Therefore, a CIM-capable system must be able to import and export properly formed MOF constructs. How the import and export operations are performed is implementation-defined for the CIM-capable system.

Objects instantiated in the MOF must, at a minimum, include all key properties and all properties marked as required. Required properties have the *REQUIRED* qualifier present and set to TRUE.

Chapter 2

# Metaschema

The Metaschema is a formal definition of the model. It defines the terms used to express the model and their usage and semantics. See also Appendix B.

The Unified Modeling Language (UML) is used to define the structure of the metaschema. In the discussion that follows, italicized words refer to objects in Figure 2-1. The reader is expected to be familiar with UML notation (see Appendix D) and with basic object-oriented concepts in the form of classes, properties, methods, operations, inheritance, associations, objects, cardinality and polymorphism.

## 2.1 Definition of the Metaschema

The elements of the model are *Schemas*, *Classes*, *Properties*, and *Methods*. The model also supports *Indications* and *Associations* as types of *Classes*, and *References* as types of *Properties*.

- A *Schema* is a group of classes with a single owner. Schemas are used for administration and class naming. Class names must be unique within its owning schema.
- A *Class* is a collection of instances that support the same type, that is, the same properties and methods. Classes can be arranged in a generalization hierarchy that represents subtype relationships between Classes. The generalization hierarchy is a rooted, directed graph and does not support multiple inheritance.

  Classes can have Methods , which represent the behavior relevant for that Class. A Class may participate in Associations by being the target of one of the References owned by the Association. Classes also have instances (not represented in Figure 2-1).
- A *Property* is a value used to characterize instances of a class. A Property can be thought of as a pair of "get and "set" functions that, when applied to an object[1] return state and set state, respectively.

  A *Method* is a declaration of a signature (that is, the method name, return type, parameters), and in the case of a concrete class, may imply an implementation.

  A *Trigger* is a recognition of a state change, such as create, delete, update, or access of a Class instance; and update or access of a Property.

  An *Indication* is an object created as a side effect of a Trigger. Because Indications are subtypes of Class, they can have Properties and Methods and be arranged in a type hierarchy.

  An *Association* is a class that contains two or more References. It represents a relationship between two or more objects. Because of the way associations are defined, it is possible to establish a relationship between Classes without affecting any of the related Classes, that is, addition of an Association does not affect the interface of the related Classes. Associations have no other significance. Only Associations can have References. Associations can be a subclass of a non-association Class[2]. Any subclass of an Association is an Association.

1. Note the equivocation between ''object'' as instance and ''object'' as class; this is common usage in object-oriented documents, and reflects the fact that in many cases, operations and concepts may apply to or involve both classes and instances.
2. Associations should not be declared as subtypes of classes that are not associations. This feature may be disallowed by future versions of the CIM standard.

*References* define the role each object plays in an Association. The reference represents the role name of a Class in the context of an Association. Associations support the provision of multiple relationship instances for a given object. For example, a system can be related to many system components.

*Properties* and *Methods* have reflexive associations that represent Property and Method overriding. A Method can override an inherited Method, which implies that any access to the inherited Method will result in the invocation of the implementation of the overriding Method. A similar interpretation implies the overriding of Properties.

*Qualifiers* are used to characterize Named Elements (for example, there are Qualifiers that define the type of a Property or the key of a Class). Qualifiers provide a mechanism that makes the metaschema extensible in a limited and controlled fashion. It is possible to add new types of Qualifier by the introduction of a new Qualifier name, thereby providing new types of metadata to processes that manage and manipulate classes, properties, and other elements of the metaschema. See below for details on the qualifiers provided.

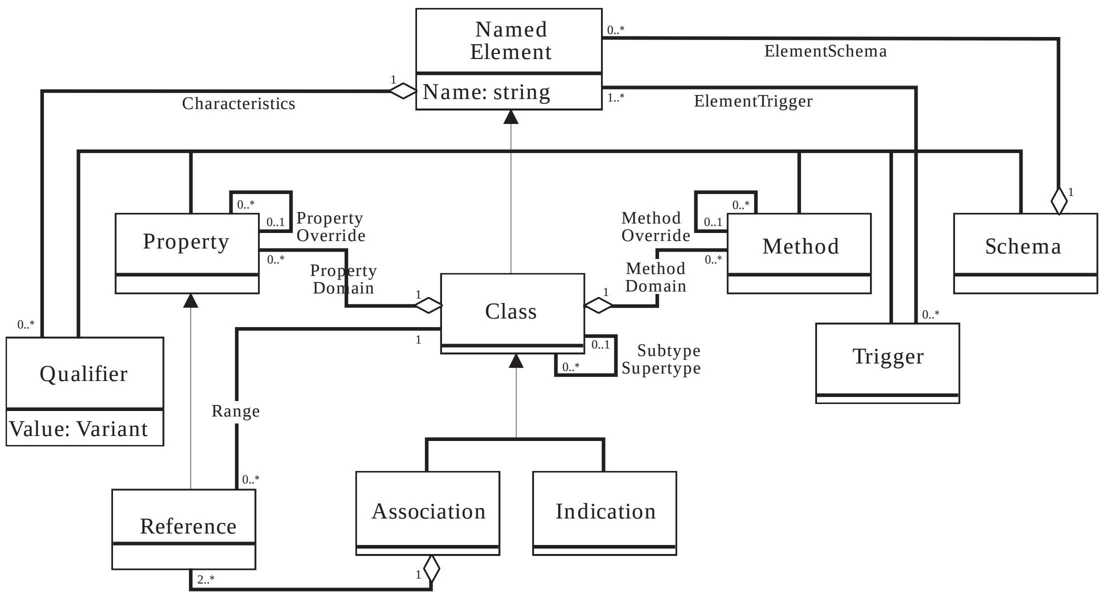


**Figure 2-1** Metaschema Structure

Figure 2-1 provides an overview of the structure of the metaschema. The complete metaschema is defined by the MOF found in Appendix B. The rules defining the metaschema are listed below in 28 items:

1. Every metaconstruct is expressed as a descendent of Named Element.
2. A Named Element is made up of zero or more Characteristics. A Characteristic is a Qualifier that characterizes a Named Element.
3. A Named Element can Trigger zero or more Indications.
4. A Schema is a Named Element and can contain zero or more classes. A Class must belong to exactly one schema .
5. A Qualifier Type (nor shown in Figure 2-1) is a Named Element and must be used to supply a type for a Qualifier (that is, a Qualifier must have a Qualifier Type). A

Qualifier Type can be used to type zero or more Qualifiers.

6. A Qualifier is a Named Element and has a Value that can be thought of as a value plus a type. The type of the Qualifier Value must agree with the type of the Qualifier Type.
7. A Property is a Named Element and has exactly one Domain: the Class that owns the Property.
8. A Property can have an Override relationship with another Property from a different class. The Domain of the overridden Property must be a super-type of the Domain of the overriding Property.
9. The Class referenced by the Range association (see Figure 2-4 on page 9) of an overriding Reference must be the same as, or a subtype of, the Class referenced by the Range associations of the Reference being overridden.
10. The Domain of a Reference must be an Association.
11. A Class is a type of Named Element. A Class can have instances (not shown on the diagram) and is the Domain for zero or more Properties. A Class is the Domain for zero or more Methods.
12. A Class can have zero or one Supertype and zero or more Subtypes.
13. An Association is a type of Class (Associations are Classes with an Association qualifier).
14. An Association must have two or more references.
15. An Association can inherit from a non-Association class.
16. Any subclass of an Association is an Association.
17. A Method is a Named Element and has exactly one Domain: the Class that owns the Method.
18. A Method can have an Override relationship with another Method from a different class. The Domain of the overridden Method must be a superclass of the Domain of the overriding Method.
19. A Trigger is an operation that is invoked on any state change, such as object creation, deletion, modification or access, or on property modification or access. Qualifiers, Qualifier Types, and Schemas may not have triggers. The operation that invokes a trigger is specified as a qualifier.
20. An Indication is a type of Class and has an association with zero or more Named Triggers that can create instances of the Indication.
21. Every metaschema object is a descendent of Named Element and, as such, has a Name. All names are case insensitive. The rules applicable to Name vary depending on the creation type of the object:
    a. Fully-qualified Class Names are unique within the repository. (See the discussion of schemas later in this section)
    b. Fully-qualified Association and Indication Names are unique within the repository (implied by the fact that Associations and Indications are subtypes of Class).
    c. Implicitly defined Qualifier Names are unique within the scope of the characterized object (that is, a Named Element may not have two Characteristics with the same Name). Explicitly defined Qualifier Names are unique within the

defining Schema. An implicitly defined qualifier must agree in type, scope, and flavor with any explicitly define Qualifier of the same name.

d. Trigger names must be unique within the property, class or method to which the Trigger applies.

e. Method and Property names must be unique within the Domain Class. A class can inherit more than one property or method with the same name. Property and method names can be qualified using the name of the declaring class.

f. Reference Names must be unique within the scope of their defining Association. Reference Names obey the same rules as Property Names. It should be noted that reference names are not required to be unique within the scope of the related class. In such a scope, the reference provides the name of the class within the context defined by the association.

   It is legal for the class System to be related to Service by two independent Associations (Dependency and Hosted Services, each with roles System and Service). It would not be legal for Hosted Services to define another Reference Service to the Service class, since a single association would then contain two references called Service.

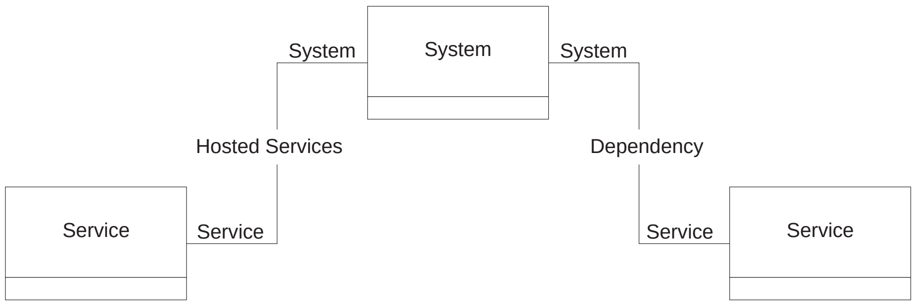


**Figure 2-2** Reference Naming

22. Qualifiers are Characteristics of Named Elements. A Qualifier has a Name (inherited from Named Element) and a Value. The Value is used to define the characteristics of the characterized Named Element. For example, a Class might have a Qualifier with the Name ‘‘Description’’, the Value of which is the description for the Class. A Property might have a Qualifier with the Name ‘‘Units’’, which has Values such as ‘‘Bytes’’ or ‘‘KiloBytes.’’ The Value can be thought of as a variant (that is, a value plus a type).

23. Association and Indication are types of Class; as such can be the Domain for Methods, Properties, and References (that is, Associations and Indications can have Properties and Methods in the same way as that of a Class). Associations and Indications can have instances. The instance of an Association has a set of references that relate one or more objects. An instance of an Indication represents the occurrence of an event, and is created as a result of that occurrence — usually by a Trigger. The key of an association is whatever is specified as its keys — typically this will be the concatenation of all of its references, though in some cases associations are possible that have a unique identifier of their own. Indications are not required to have keys. Typically Indications are very shortlived objects used to communicate information to an event consumer.

24. A Reference has a Range that represents the type of the Reference. For example, in the model of *PhysicalElements* and *PhysicalPackages*, there are two References:

*ContainedElement*, which has PhysicalElement as its Range and Container as its Domain, and *ContainingElement*, which has PhysicalPackage as its Range and Container as its Domain.

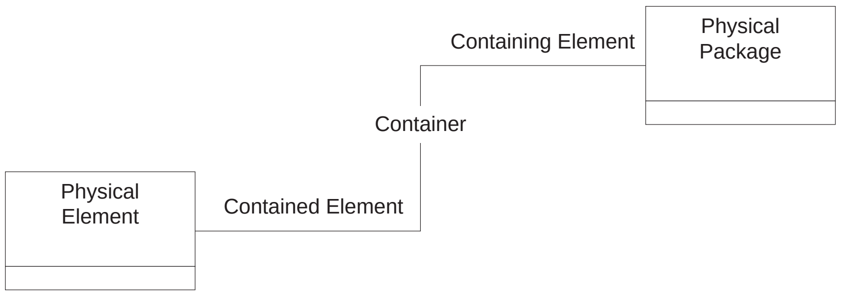


**Figure 2-3** References, Ranges, and Domains

25. A Class has a Subtype-Supertype association that represents substitutability relationships between the Named Elements involved in the relationship. The association implies that any instance of a subtype can be substituted for any instance of the supertype in an expression without invalidating the expression.

    Revisiting the Container example: Card is a Subtype of PhysicalPackage. Therefore, Card can be used as a value for the Reference ContainingElement (that is, an instance of Card can be used as a substitute for an instance of PhysicalPackage).

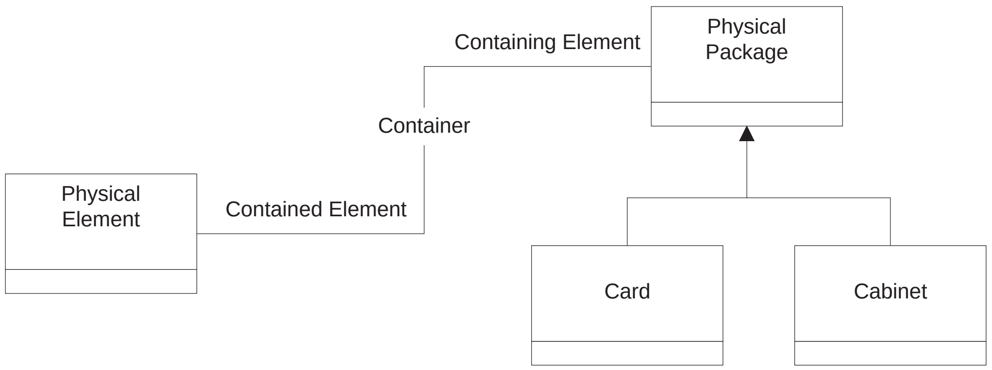


**Figure 2-4** References, Ranges, Domains, and Inheritance

A similar relationship can exist between Properties. For example, given that PhysicalPackage has a Name property (which is a simple alphanumeric string), Card Overrides Name to a name of alpha-only characters.

The same idea applies to Methods. A Method that overrides another Method must support the same signature as the original Method and, most importantly, must be substitutable for the original method in all cases.

26. The Override relationship is used to indicate the substitution relationship between a property or method of a subclass and the overridden property or method inherited from the superclass. This is the opposite of the C++ convention in which the superclass property or method is specified as virtual, with overriding occurring thereafter as a side effect of declaring a feature with the same signature as the inherited virtual feature.

27. The number of references in an Association class defines the ‘‘arity’’ of the Association. An Association containing two references is a binary Association; an Association containing three references is a ternary association. Unary Associations (Associations containing one reference) are not meaningful. Arrays of references are not allowed. When an association is sub-classed, its ‘‘arity’’ cannot change.

28. Schemas provide a mechanism that allows ownership of portions of the overall model by individuals and organizations who are responsible for managing the evolution of the schema. In any given installation, all classes are mutually visible, regardless of schema ownership. Schemas have a universally unique name. The schema name is considered part of the class name. The full class name (that is, class name plus owning schema name) is unique within the namespace and is referred to as the fully-qualified name (see Section 2.4 on page 14).

## 2.2 Property Data Types

Property data types are limited to the intrinsic data types, or arrays of such. Structured types are constructed by designing new classes. If the Property is an array property, the corresponding variant type is simply the array equivalent (fixed or variable length) of the variant for the underlying intrinsic type.

Table 2-1 shows the intrinsic data types and their interpretation:

| Intrinsic Data Type | Interpretation |
|---|---|
| uint8 | Unsigned 8-bit integer |
| sint8 | Signed 8-bit integer |
| uint16 | Unsigned 16-bit integer |
| sint16 | Signed 16-bit integer |
| uint32 | Unsigned 32-bit integer |
| sint32 | Signed 32-bit integer |
| uint64 | Unsigned 64-bit integer |
| sint64 | Signed 64-bit integer |
| string | UCS-2 string |
| Boolean | Boolean |
| real32 | IEEE 4-byte floating-point |
| real64 | IEEE 8-byte floating-point |
| datetime | A string containing a date-time |
| <classname>ref: | Strongly typed reference |
| char16 | 16-bit UCS-2 character |

**Table 2-1** Intrinsic Data Types

### 2.2.1 Date, Time, and Interval Types

Date, datetime, interval and time property types are aliases for each other and use the same fixed string-based format:

```
yyyymmddhhmmss.mmmmmmsutc
```

where:

- yyyy is a 4 digit year
- mm is the month
- dd is the day
- hh is the hour (24-hour clock)
- mm is the minute
- ss is the second
- mmmmmm is the number of microseconds
- s is a ‘‘+’’ or ‘‘-’’ indicating the sign of the UTC correction field (Universal Coordinated Time is, for all intents and purposes, the same as Greenwich Mean Time), or a ":". In this case, the value is interpreted as a time interval, and yyyymm are interpreted as days.
- utc is the offset from UTC in minutes (using the sign indicated by s). It is ignored for a time interval.

For example, Wednesday, May 25, 1998, at 1:30:15 PM EDT would be represented as:

```
19980525133015.0000000-300
```

Values must be zero-padded so that the entire string is always the same 25-character length. Fields which are not significant must be replaced with asterisk characters.

Similarly, intervals use the same format, except that the interpretation of the fields is based on elapsed time. For example, an elapsed time of 1 day, 13 hours, 23 minutes, and 12 seconds would be:

```
00000001132312.000000+000
```

A UTC offset of zero is always used for interval properties.

The string-based interval format is:

```
ddddddddhhmmss.mmmmmm:000
```

## 2.2.2 Indicating Additional Type Semantics with Qualifiers

Since *counter* and *gauge* types (as well as many others) are actually simple integers with specific semantics, they are not treated as separate intrinsic types. Instead, qualifiers must be used to indicate such semantics when properties are being declared (note the example below merely suggests how this may be done — the qualifiers names chosen should not be regarded as part of this standard):

```
class Acme_Example
{
        [counter]
    uint32 NumberOfCycles;
        [gauge]
    uint32 MaxTemperature;
        [octetstring, ArrayType("Indexed")]
    uint8 IPAddress[10];
};
```

Implementers are permitted to introduce arbitrary qualifiers in this manner, for documentary purposes. The semantics are not enforced.

## 2.3 Supported Schema Modifications

The following is a list of supported schema modifications, the use of some of which will result in changes in application behavior. Changes are all subject to security restrictions; in particular, only the owner of the schema or someone authorized by the owner can make modifications to the schema.

1. A Class can be added to or deleted from a schema.
2. A Property can be added to or deleted from a class.
3. A Class can be added as a subtype or supertype of an existing class.
4. A Class can become an association as a result of the addition of an ASSOCIATION qualifier and two or more references.
5. A Qualifier can be added to or deleted from any Named Element.
6. The Override qualifier can be added to or removed from a property or reference.
7. A Class can alias a property (or reference, if the class is a descendent of Association) using the Alias qualifier. Both inherited and immediate properties of the class may be aliased.
8. A Method can be added to a class.
9. A Method can override an inherited method.
10. Methods can be deleted and the signature of a method can be changed.
11. A Trigger may be added to or deleted from a class.

In defining an extension to the schema, the schema designer is expected to operate within the constraints of the classes defined in the Core schema. With respect to classification, any component of a system added to the schema must be defined as a subclass of an appropriate Core schema class. It is expected that the schema designer will address the following question to each of the Core schema classes:

‘‘Is the class being added a subtype of this class?’’

Having identified the Core schema class to be extended, the same question should be addressed with respect to each of the subclasses of the identified class. This process, which defines the super classes of the class to be defined, should be continued until the most detailed class is identified. The Core model is not a part of the metaschema, but is an important device for introducing uniformity across schemas intended to represent aspects of the managed environment.

### 2.3.1 Schema Versions

Certain modifications to a schema can cause failure in applications that operated against the schema prior to the modification. These modifications are:

- Deletion of classes, properties, or methods
- Movements of a class anywhere other than down a hierarchy
- Alteration of property type or method signature
- Alterating reference range to anything other than a supertype of the original specification

Other alterations are considered to be interface preserving. Any use of the schema changes listed above implies the generation of a new major version of the schema (as defined by the VERSION qualifier described in Section 2.5.2).

## 2.4 Class Names

Fully qualified class names are in the form:

```
<schema name><class name>
```

An underscore is used as a delimiter between the <schema name> and the <class name>. The delimiter is not allowed to appear in the <schema name> though it is permitted in the <class name>.

The format of the fully qualified name is intended to allow the scope of class names to be limited to a schema, that is the schema name is assumed to be unique and the class name is only required to be unique within the schema. The isolation of the schema name using the underscore character allows user interfaces to conveniently strip off the schema where the schema is implied by the context.

Examples of fully qualified class names are:

| | |
|---|---|
| `CIM_ManagedSystemElement` | The root of the CIM managed system element hierarchy |
| `CIM_ComputerSystem` | The object representing computer systems in the CIM schema |
| `CIM_Component` | The association relating systems to their components |
| `Win32_ComputerSystem` | The object representing computer systems in the Win32 schema |

## 2.5 Qualifiers

Qualifiers are values that provide additional information about classes, associations, indications, methods, method parameters, triggers, instances, properties or references. All qualifiers have a name, type, value, scope, flavor and default value. Qualifiers cannot be duplicated; there cannot be more than one qualifier of the same name for any given class, instance, or property.

The following sections describe meta, standard, optional and user-defined qualifiers. When any of these qualifiers are used in a model, they must be declared in the MOF file before being used. These declarations must abide by the details (name, applied to, type) specified in the tables below. It is not valid to change any of this information for the meta, standard and optional qualifiers. It is possible to change the default values. A default value is the assumed value for a qualifier when it is not explicitly specified for particular model elements.

### 2.5.1 Metaqualifiers

The following table lists the qualifiers that are used to refine the definition of the metaconstructs in the model. These qualifiers are used to refine the actual usage of an object class or property declaration within the MOF syntax. These qualifiers are all mutually exclusive.

| **Qualifier** | **Default** | **Type** | **Meaning** |
|---|---|---|---|
| ASSOCIATION | FALSE | BOOLEAN | The object class is defining an association. |
| INDICATION | FALSE | BOOLEAN | The object class is defining an indication. |

**Table 2-2** Qualifiers for Metaconstructs

### 2.5.2 Standard Qualifiers

The following table is a list of standard qualifiers that all CIM-compliant implementations are required to handle. Any given object will not have all of the qualifiers listed. It is expected that additional qualifiers will be supplied by concrete classes to facilitate the provision of instances of the class and other operations on the class.

It is also important to recognize that not all of these qualifiers can be used together. First, as indicated in the table, not all qualifiers can be applied to all metamodel constructs. These limitations are identified in the ‘‘Applies To’’ column of the following table. Second, for a particular metamodel construct like associations, the use of the legal qualifiers may be further constrain because some qualifiers are mutually exclusive or the use of one qualifier implies some restrictions on the value of another qualifier, etc. These usage rules are documented in the ‘‘Meaning’’ column of the table. Third, legal qualifiers are not inherited by metamodel constructs. For example, the MAXLEN qualifier that applies to properties is not inherited by references.

The ‘‘Applies To’’ column in the table identifies the metamodel construct(s) that can use a particular qualifier. For the qualifiers like ASSOCIATION discussed in the previous section, there is an implied usage rule that the metaqualifier must also be present. For example, the implicit usage rule for the AGGREGATION qualifiers is that the ASSOCIATION qualifier must also be present.

**Table 2-3** Standard Qualifiers

| **Qualifier** | **Default** | **Applies to** | **Type** | **Meaning** |
|---|---|---|---|---|
| ABSTRACT | FALSE | Class | BOOLEAN | Indicates that the class is abstract and serves only as a base for new classes. It is not possible to create instances of such classes. |
| AGGREGATE | FALSE | Reference | BOOLEAN | Defines the "parent" component of an Aggregation association.<br><br>**Usage Rule:** The Aggregation and Aggregate qualifiers are used together — Aggregation qualifying the association, and Aggregate specifying the "parent" reference. |
| AGGREGATION | FALSE | Association | Indicates that the association is an aggregation. | |
| ALIAS | NULL | Property, Reference, Method | STRING | Establishes an alternate name for a property or method in the schema. |
| ARRAYTYPE | "Bag" | Property | STRING | Indicates the type of the qualified array. Valid values are "Bag", "Indexed" and "Ordered".<br><br>**Usage Rule:** The ArrayType qualifier should only be applied to properties that are arrays (defined using the square bracket syntax specified in Appendix A on page 69). |
| DESCRIPTION | NULL | Any | STRING | Provides a description of a Named Element. |
| IN | TRUE | Parameter | BOOLEAN | Indicates that the associated parameter is used to pass values to a method. |
| KEY | FALSE | Property | BOOLEAN | Indicates that the property is a namespace-level key. If more than one property has the KEY qualifier, then all such properties collectively form the key (a compound key).<br><br>**Usage Rule**: be modified thereafter. It does not make sense to apply a default value to a KEY-qualified property. |
| MAPPINGSTRINGS | NULL | Class, Property, Association, Indication, Reference | STRING ARRAY | Mapping strings for one or more management data providers or agents. See Section 2.5.5 on page 21 and Section 2.5.6 on page 22 for more details. |
| MAX | NULL | Reference | INT | Indicates the maximum cardinality of the reference (i.e. the maximum number of values a given reference can have for each set of other reference values in the association). For example, if an association relates A instances to B instances, and there must be at most one A instance for each B instance, then the reference to A should have a *Max(1)* qualifier. |
| MAXLEN | NULL | Property | INT | Indicates the maximum length, in characters, of a string property. When overriding the default value, any unsigned integer value (uint32) can |

| **Qualifier** | **Default** | **Applies to** | **Type** | **Meaning** |
|---|---|---|---|---|
| | | | | be specified. A value of NULL implies unlimited length. |
| MIN | 0 | Reference, Property | INT | Indicates the minimum cardinality of the reference (i.e. the minimum number of values a given reference can have for each set of other reference values in the association). For example, if an association relates A instances to B instances, and there must be at least one A instance for each B instance, then the reference to A should have a *Min(1)* qualifier. |
| MODEL CORRESPONDENCE | NULL | Property | STRING ARRAY | Indicates a correspondence between an object's property and other properties in the CIM Schema. Object properties are identified using the following syntax:<br>&lt;schema name&gt; "_" &lt;class or association name&gt; "." &lt;property name&gt; |
| NONLOCAL | NULL | Reference | STRING | Indicates the location of an instance. Its value is:<br>&lt;namespacetype&gt;: &lt;namespacehandle&gt; |
| OUT | FALSE | Parameter | BOOLEAN | Indicates that the associated parameter is used to return values from a method. |
| OVERRIDE | NULL | Property, Method, Reference | STRING | Indicates that the property in the derived class intentionally overrides the property in the parent class. The value of this qualifier needs to identify the class and subordinate construct (property, method, or reference) that is being overridden. The format of the string to accomplish this is:<br>[&lt;class&gt;.]&lt;subordinate construct&gt;<br>If the class name is omitted, the Override applies to the subordinate construct in the parent class in the inheritance tree.<br>**Usage Rule:** The OVERRIDE qualifier can only refer toame metamodel for which it is specified. |
| PROPAGATED | NULL | Property | STRING | The propagated qualifier is a string-valued qualifier that contains the name of the key that is being propagated. Its use assumes the existence of only one weak qualifier on a reference that has the containing class as its target. The associated property must have the same value as the property named by the qualifier in the class on the other side of the weak association. The format of the string to accomplish this is:<br>[&lt;class&gt;.]&lt;subordinate construct&gt;<br>**Usage Rule**: When the PROPAGATED qualifier is used, the KEY qualifier must be specified with a value of TRUE. |
| READ | TRUE | Property | BOOLEAN | Indicates that the property is readable. |

| **Qualifier** | **Default** | **Applies to** | **Type** | **Meaning** |
|---|---|---|---|---|
| REQUIRED | FALSE | Property | BOOLEAN | Indicates that a non-NULL value is required for the property. |
| REVISION | NULL | Class, Schema, Association, Indication | STRING | Provides the minor revision number of the schema object.<br>**Usage Rule:** The VERSION qualifier must be present to supply the major version number when the REVISION qualifier is used. |
| SCHEMA | NULL | Property Method | STRING | The name of the schema in which the feature is defined. |
| SOURCE | NULL | Class Association Indication | STRING | Indicates the location of an instance. Its value is:<br>&lt;namespacetype&gt;:&lt;namespacehandle&gt; |
| UNITS | NULL | Property | STRING | Provides units in which the associated property is expressed. For example a Size property might have Units(‘‘bytes’’). The complete set of standard units is defined in Appendix C. |
| VALUEMAP | NULL | Property | STRING ARRAY | Defines the set of permissible values for this property. The ValueMap can be used alone, or in combination with the Values qualifier. When used in combination with the Values qualifier, the location of the property value in the ValueMap array provides the location of the corresponding entry in the Values array.<br>ValueMap may only be used with string and integer values. The syntax for representing an integer value in the ValueMap array is:<br>[+\|-]digit[*digit]<br>The content, maximum number of digits and represented value are constrained by the type of the associated property. For example, uint8 may not be signed, must be less than four digits, and must represent a value less than 256. |
| VALUES | NULL | Property | STRING ARRAY | Provides translation between an integer value and an associated string. If a ValueMap qualifier is not present, the Values array is indexed (zero relative) using the value in the associated property. If a ValueMap qualifier is present, the Values index is defined by the location of the property value in the ValueMap. |
| VERSION | NULL | Class, Schema, Association, Indication | STRING | Provides the major version number of the schema object. This is incremented when changes are made to the schema that alter the interface. |
| WEAK | FALSE | Reference | BOOLEAN | Indicates that the keys of the referenced class include the keys of the other participants in the association. This qualifier is used when the identity of the referenced class depends on the identity of the other participants in the association. No more than one reference to any given class can be weak. The other classes in the association must define a key. The keys of the |

| Qualifier | Default | Applies to | Type | Meaning |
|---|---|---|---|---|
| | | | | other classes in the association are repeated in the referenced class and tagged with a propagated qualifier. |
| WRITE | TRUE | Property, Reference | BOOLEAN | Applies to properties. Indicates that the property is writable. If used alone and not in combination with READ, then the property is to be considered write-only. |

### 2.5.3 Optional Qualifiers

The following table is a list of optional qualifiers. These address situations that are not common to all CIM-compliant implementations. Thus, CIM-compliant implementations can ignore optional qualifiers since they are not required to interpret or understand these qualifiers.

These optional qualifiers are defined to avoid deployment of random user-defined qualifiers for these recurring situations.

**Table 2-4** Optional Qualifiers

| Qualifier | Default | Applies to | Type | Meaning |
|---|---|---|---|---|
| DELETE | FALSE | Association<br>Reference | BOOLEAN | **For associations:** Indicates that the qualified association must be deleted if any of the objects referenced in the association are deleted, AND the respective object referenced in the association is qualified with IFDELETED.<br>**For references:** Indicates that the referenced object must be deleted if the association containing the reference is deleted, AND qualified with IFDELETED, or if any of the objects referenced in the association are deleted AND the respective object referenced in the association is qualified with IFDELETED.<br>**Usage Rule:** Applications must to chase associations according to the modeled semantic and delete objects appropriately. Note that this usage rule must be verified when the CIM security model is defined. |
| EXPENSIVE | FALSE | Property<br>Reference<br>Class<br>Association<br>Method | BOOLEAN | Indicates the property or class is expensive to compute. |
| IFDELETED | FALSE | Association<br>Reference | BOOLEAN | Indicates that all objects qualified by DELETE within the association must be deleted if the referenced object or the association, respectively, is deleted. |
| INVISIBLE | FALSE | Association<br>Property<br>Method<br>Reference<br>class | BOOLEAN | Indicates that the association is defined only for internal purposes (for example, for definition of dependency semantics) and should not be displayed (for example, in maps). |
| LARGE | FALSE | Property<br>Class | BOOLEAN | Indicates the property or class requires a large amount of storage space. |
| SYNTAX | NULL | Property<br>Reference | STRING | Specific type assigned to a property.<br>**Usage Rule:** Must be used with the SYNTAXTYPE qualifier. |
| SYNTAXTYPE | NULL | Property<br>Reference | STRING | Defines the format of the SYNTAX qualifier.<br>**Usage Rule:** Must be used with the SYNTAX qualifier. |
| TRIGGERTYPE | NULL | Class<br>Property<br>Method<br>Association<br>Indication<br>Reference | STRING | Indicates the circumstances under which a trigger is fired.<br>**Usage Rule:** The trigger types vary by metamodel construct. For classes and associations, the legal values are CREATE, DELETE, UPDATE and ACCESS. For properties and references, the legal values are: UPDATE and ACCESS. For methods, the legal values are BEFORE and AFTER. For indications, the legal values are THROWN. |

### 2.5.4 User-Defined Qualifiers

The user can define any additional arbitrary named qualifiers. However, it is recommended that only defined qualifiers are used (as much as possible), and to only add them if there is no other way to accomplish a particular objective.

### 2.5.5 Mapping MIF Attributes

Mapping Management Information Format (MIF) attributes to CIM Properties can be accomplished using the MAPPINGSTRING qualifier. This qualifier provides a mechanism to specify the mapping from DMTF and vendor defined MIF groups to specific properties. This allows for mapping using either Domain or Recast Mapping.

Every MIF group contains a unique identification that is defined using the class string which is defined as follows:

```
defining body|specific name|version
```

where defining body is the creator and owner of the group, specific name is the unique name of the group and version is a three digit number that identifies the version of the group definition. In addition, each attribute has a unique numeric identifier starting with the number one.

The mapping qualifier can therefore be represented as a string that is formatted as follows:

```
MIF.defining body|specific name|version.attributeid
```

where MIF is a constant defining this as a MIF mapping followed by a dot. This then followed by the class string for the group this defines and optionally followed by a dot and the identifier of a unique attribute.

In the case of a Domain Mapping, all of the above information is required, and provides a way to map an individual MIF attribute to a particular CIM Property.

In the case of the recast mapping, a CIM class can be recast from a MIF group and only the MIF constant followed by the dot separator followed by the class string is required.

For example, a Domain Mapping of a DMTF MIF attribute to a CIM property could be as follows:

```
[MAPPINGSTRING("MIF.DMTF|ComponentID|001.4"),READ]
SerialNumber = "";
```

The above declaration defines a mapping to the SerialNumber property from the DMTF Standard Component ID group's serial number attribute. Because the qualifiers of CIM are a superset of those found in MIF syntax, any qualifier may be overridden in the CIM definition.

To recast an entire MIF group into a CIM Object, the mapping string can be used to define entire Class. For example:

```
[MAPPINGSTRINGS {"MIF.DMTF|Software Signature|002"}]
class MicroSoftWord : SoftwareSignature
{
    ...
}
```

### 2.5.6 Mapping Generic Data to CIM Properties

In addition to mapping MIF attributes, the MAPPINGSTRINGS qualifier can be used to map SNMP variables to CIM properties. Every standard SNMP variable has associated with it a variable name and a unique object identifier (OID) that is defined by a unique naming authority. This naming authority is a string. This string can either be a standards body (for example, "IETF"), a company name (for example, "Acme") for defining the mappings to a company's private MIB, or an appropriate management protocol (for example, "SNMP").

For the IETF case, the ASN.1 module name, not the RFC number, should be used as the MIB name. For example, instead of saying RFC1493, the string "BRIDGE-MIB" should be used). This is also true for the case of a company name being used as the naming authority. For the case of using a management protocol like SNMP, the SNMP OID can be used to identify the appropriate SNMP variable. This latter is especially important for mapping variables in private MIBs.

It should be noted that the concept of a naming authority for mapping data other than SNMP data into CIM properties could be derived from this requirement. As an example, this can be used to map attributes of other data stores (like directories) using an application-specific protocol (for example, LDAP).

The syntax for mapping MIF attributes as defined in Section 2.5.5 on page 21 is as follows:

```
MIF.<defining_body | specific_name | version>.attributeid
```

The above MIF format can be reconciled with the more general syntax needed to map generic data to CIM properties by realizing that both forms can be represented as follows:

```
<Format>.<Scoping_Name>.<Content>
```

where:

- `Format` defines the format of the entry. It has the following values:

  MIF The rest of the string is interpreted as MIF data.

  MIB The rest of the string is interpreted as a variable. name of a MIB

  OID The rest of the string is interpreted as an OID that is defined by a particular protocol to represent a variable name.

- `Scoping_Name` defines the format used to uniquely identify the entry. It has the following values:

  `defining_body | specific_name | version`
  Used for MIF mappings.

  `Naming_Authority | MIB_Name`
  Used for MIB mappings.

  `Naming_Authority | Protocol_Name`
  Used for protocol mappings that use OIDs to represent a variable name.

- `Content` defines the value of the entry. It has the following values:

  `attributeid`
  Used for MIF mappings.

  `Variable_Name`
  Used for MIB mappings.

  `OID` Used for protocol mappings.

Here are two examples of the syntax. The first uses the MIB format and looks as follows:

```
[Description ("OperatingSystem’s notion of the local date and time of
day"),
    MappingStrings {"MIB.IETF | HOST-RESOURCES-MIB.hrSystemDate"}]
datetime LocalDateTime;
```

The second example uses the OID format and looks as follows:

```
[Description ("OperatingSystem’s notion of the local date and time of
day"),
    MappingStrings {"OID.IETF | SNMP.1.3.6.1.2.1.25.1.2"}]
datetime LocalDateTime;
```

Chapter 3

# *Managed Object Format*

The management information is described in a language based on the Interface Definition Language (IDL) — see the DCE RPC Specification (C309, listed in **Referenced Documents**) called the Managed Object Format (MOF).

This specification uses the term MOF specification to refer to a collection of management information described in a manner conformant to the MOF syntax.

Elements of MOF syntax are introduced on a case-by-case basis, with examples. In addition, a complete description of the MOF syntax is provided in Appendix A.

**Note:** All grammars defined in this specification use the notation defined in the UNICODE Standard, Version 2.0 (see **Referenced Documents**). Any exceptions are stated with the grammar.

The MOF syntax is a way to describe object definitions in textual form. It establishes the syntax for writing definitions. The main components of a MOF specification are textual descriptions of classes, associations, properties, references, methods and instance declarations and their associated qualifiers. Comments are permitted.

In addition to serving the need for specifying the managed objects, a MOF specification can be processed using a compiler. To assist the process of compilation, a MOF specification consists of a series of compiler directives.

A MOF file can be encoded in either Unicode or UTF-8.

## 3.1 MOF Usage

The managed object descriptions in a MOF specification can be validated against an active namespace (see Chapter 5 on page 43). Such validation is typically implemented in an entity acting in the role of a Server. This section describes the behavior of an implementation when introducing a MOF specification into a namespace. Typically, such a process validates both the syntactic correctness of a MOF specification, as well as the semantic correctness of such a specification against a particular implementation. A MOF specification can be validated for the syntactic correctness alone, in a component such as a MOF compiler.

## 3.2 Class Declarations

A class declaration is treated as an instruction to create a new class. It is a local matter as to whether the process of introducing a MOF specification into a namespace is allowed to change classes or modify classes.

Any class referenced in the specification of a class or reference specification must exist at the time of the specification (that is, forward references are not allowed).

## 3.3 Instance Declarations

Classes *must* be defined before they are used to declare instances. However, if a class definition is already resident within the namespace, then that class declaration need not appear in a MOF specification that introduces the instances of that class.

Any instance declaration is treated as an instruction to create a new instance where the object's key values do not already exist, or an instruction to modify an existing instance where an object with identical key values already exists.

*Chapter 4*

# *Managed Object Format Components*

## 4.1 Keywords

All keywords in the MOF syntax are case-insensitive.

## 4.2 Comments

Comments can appear anywhere in MOF syntax and are indicated by either a leading double slash "//", or a pair of matching "/*" and "*/" sequences.

A "//" comment is terminated by an EOL or by the end of the MOF specification (whichever comes first).

For example:

```
// This is a comment
```

A "/*" comment is terminated by the next "*/" sequence, or by the end of the MOF specification (whichever comes first). Comments are not recognized by the metamodel and as such will not be preserved across compilations, that is, the output of a MOF compilation is not required to include any comments.

## 4.3 Validation Context

Semantic validation of a MOF specification involves an explicit or implied namespace context. This is defined as the namespace against which the objects in the MOF specification are validated and the namespace in which they are created. Multiple namespaces typically indicate the presence of multiple management spaces or multiple devices.

## 4.4 Naming of Schema Elements

This section describes the rules for naming of schema elements. This applies to classes, properties, qualifiers, methods and namespaces.

CIM is a conceptual model that is not bound to a particular implementation. This allows it to be used to exchange management information in a variety of ways, examples of which are described in Figure 1-1 on page 3. Some implementations may use case-sensitive technologies, while others may use case-insensitive technologies. The naming rules defined in this section are chosen to allow efficient implementation in either environment, and to enable the effective exchange of management information between all compliant implementations.

All names are case insensitive, in that two schema item names are identical if they differ only in case. This is mandated so that scripting technologies that are case insensitive can leverage CIM technology. (Note however that string values assigned to properties and qualifiers are not covered by this rule, and should be treated in a case-sensitive manner.)

The case of a name is set by its defining occurrence and must be preserved by all implementations. This is mandated so that implementations can be built using case-sensitive technologies such as Java and object databases. (This also allows names to be consistently displayed using the same user-friendly mixed-case format).

For example, an implementation, if asked to create class "Disk", must reject the request if there is already a class "DISK" in the current schema. Otherwise, when returning the name of the class "Disk", it must return the name in mixed case as it was originally specified.

CIM does not currently require support for any particular query language. It is assumed that implementations will specify which query languages are supported by the implementation and will adhere to the case conventions that prevail in the specified language. That is, if the query language is case-insensitive, statements in the language will behave in a case-insensitive manner.

For the full rules for schema names, see Appendix E.

## 4.5 Class Declarations

A class is an object describing a grouping of data items that are conceptually related and thought of as modeling an object. Class definitions provide a type system for instance construction.

### 4.5.1 Declaring a Class

A class is declared by specifying each of the following components:

1. The qualifiers of the class. This may be empty, or a list of qualifier name/value bindings separated by commas "," and enclosed with square brackets ("[" and "]").
2. The class name.
3. The name of the class from which this is derived (if any).
4. The class properties, which define the data members of the class. A property may also have an optional qualifier list, expressed in the same way as the class qualifier list. In addition a property has a data type, and (optionally) a default (initializer) value.
5. The methods supported by the class. A method has a signature consisting of its return type plus its parameters and their type and usage.

The following sample shows how to declare a class:

```
          [abstract]
class Win32_LogicalDisk
{
          [read]
    string DriveLetter;
          [read, Units(kilo byes)]
    sint32 RawCapacity     =     0;
          [write]
    string VolumeLabel;
           [Dangerous]
    boolean Format(IN bool FastFormat);
};
```

### 4.5.2 Subclasses

To indicate that a class is a subclass of another class, the derived class is declared by using a colon followed by the superclass name.

For example, if the class "Acme_Disk_v1" is derived from the class "CIM_Media":

```
class Acme_Disk_v1 : CIM_Media
{
    // Body of class definition here ...
};
```

The terms Base class, Superclass and Supertype are used interchangeably, as are Derived class, Subclass and Subtype.

The Superclass declaration *must* appear at a prior point in the MOF specification or already be a registered class definition in the namespace in which the derived class is defined.

### 4.5.3 Default Property Values

Any properties in a class definition can have default initializers. For example:

```
class Acme_Disk_v1 : CIM_Media
{
    string Manufacturer    =     "Acme";
    string ModelNumber     =     "123-AAL";
};
```

When new instances of the class are declared, then such a property is automatically assigned its default value unless the instance declaration explicitly assigns a value to the property.

### 4.5.4 Class and Property Qualifiers

Qualifiers are metadata about a property, method, method parameter, class, or instance and are not part of the definition itself. For example, a qualifier is used to indicate whether a property value is modifiable (using the WRITE qualifier). Qualifiers always precede the declaration to which they apply.

Certain qualifiers are well known and cannot be redefined (see the description of the metaschema). Apart from these, arbitrary qualifiers may be used.

Qualifier declarations include an explicit type indicator, which must be one of the intrinsic types. A qualifier with an array-based parameter is assumed to have a type, which is a variable-length homogeneous array of one of the intrinsic types. Note that in the case of boolean arrays, each element in the array is either TRUE or FALSE.

Examples:

```
Write(true)                                 // boolean
profile { true, false, true }               // boolean []
description("A string")                     // string
info { "first", "second", "third" }         // string []
id(12)                                      // sint32
idlist { 21, 22, 40, 43 }                   // sint32 []
apple(3.14)                                 // real32
oranges { -1.23E+02, 2.1 }                  // real32 []
```

Qualifiers are applied to a class by preceding the class declaration by a qualifier list, comma-separated, and enclosed within square brackets. Qualifiers are applied to a property in a similar fashion.

For example:

```
class CIM_Process:CIM_LogicalElement
{
    uint32 Priority;
        Write(true)]
    string Handle;
};
```

Qualifiers can be automatically propagated from classes to derived classes, or from classes to instances subject to certain rules. The rules behind how the propagation occurs are attached to each qualifier and encapsulated in the concept of the qualifier flavor. For example, a qualifier may be designated in the base class as automatically propagating to all of its derived classes, or the qualifier could be designated as belonging specifically to that class and not to be propagated. This aspect of the flavor is referred to as the propagation rule. In addition, the qualifier flavor can be used to control whether or not derived classes can be override the qualifier value, or whether it must be fixed for an entire class hierarchy. This aspect of qualifier flavor is referred to as override permissions.

Qualifier flavors are indicated by an optional clause after the qualifier and preceded by a colon. They consist of some combination of the key words EnableOverride, DisableOverride, ToSubclass and Restricted, indicating the applicable propagation and override rules. The recognized flavor types are listed in Table 4-1 on page 31.

For example:

```
class CIM_Process:CIM_LogicalElement
{
   uint32 Priority;
       [Write(true):DisableOverride  ToSubclass]
   string Handle;
};
```

In this example, Handle is designated as writable for the Process class and for every subclass of this class.

When specifying a boolean qualifier in a class or property declaration, the name of the qualifier can be used without also specifying a value. From the previous example:

```
class CIM_Process:CIM_LogicalElement
{
    uint32 Priority;
        [Write]      // Equivalent declaration to Write (True)
    string Handle;
};
```

If only the qualifier name is listed for a boolean qualifier, it is implicitly set to TRUE.

In contrast, when a qualifier is not specified at all for a class or property, the default value for the qualifier is assumed. Using another example:

```
        [Association,
        Aggregation]       // Specifies the Aggregation qualifier
                           // to be True
class CIM_SystemDevice: CIM_SystemComponent
{
        [Override ("GroupComponent"),
        Aggregate] // Specifies the Aggregate qualifier to be True
    CIM_ComputerSystem Ref GroupComponent;
        [Override ("PartComponent"),
        Weak]      // Defines the Weak qualifier to be True
    CIM_LogicalDevice Ref PartComponent;
};

        [Association]      // Since the Aggregation qualifier
                           // is not specified, its default value,
                           // False, is set
class Acme_Dependency: CIM_Dependency
{
        [Override ("Antecedent")]   // Since the Aggregate and Weak
                           // qualifiers are not used, their default
                           // values, False, are assumed
    Acme_SpecialSoftware Ref Antecedent;
        [Override ("Dependent")]
    Acme_Device Ref Dependent;
};
```

| Parameter | Interpretation | Default |
|---|---|---|
| EnableOverride | The qualifier is overridable | yes |
| DisableOverride | The qualifier cannot be overriden | no |
| ToSubclass | The qualifier is inherited by any subclass | yes |
| Restricted | The qualifier applies only to the class in which it is declared | no |
| Translatable | Indicates the value of the qualifier can be specified in multiple locales (language and country combination). When Translatable(yes) is specified for a qualifier, it is legal to create implicit qualifiers of the form: `label_ll_cc` where "label" is the name of the qualifier with Translatable(yes), and ll and cc are the language code and country code designation, respectively, for the translated string. In other words, a label_ll_cc qualifier is a clone, or derivative, of the "label" qualifier with a postfix to capture the translated value's locale. The locale of the original value (that is, the value specified using the qualifier with a name of "label") is determined by the locale pragma.<br><br>When a label_ll_cc qualifier is implicitly defined, the values for the other flavor parameters are assumed to be the same as for the "label" qualifier. When a label_ll_cc qualifier is defined explicitly, the values for the other flavor parameters must also be the same. A "yes" for this parameter is only valid for string-type qualifiers.<br><br>**Example:**<br>If an English description is translated into Mexican Spanish, the actual name of the qualifier is: DESCRIPTION_es_MX. | no |

**Table 4-1** Qualifier Flavors for Keyword Parameters

### 4.5.5 Key Properties

Instances of a class require some mechanism through which the instances can be distinguished within a single namespace. Designating one or more properties with the reserved qualifier "key" provides instance identification.

For example, the following class has one property ("Volume") which serves as its key.

```
class Acme_Drive
{
        [key] string Volume;
    string FileSystem;
    sint32 Capacity;
};
```

In this case, instances of "Drive" are distinguished using the "Volume" property, which acts as the key for the class.

Compound keys are supported and are designated by marking all of the required properties with the key qualifier.

If a new subclass is defined from a superclass, and the superclass has key properties (including those inherited from other classes), the new subclass *cannot* define any additional key properties. New key properties in the subclass can be introduced only if all classes in the inheritance chain of the new subclass are keyless.

If any reference to the class has the Weak qualifier, the properties that are qualified as Key in the other classes in the association are propagated to the referenced class. The key properties are duplicated in the referenced class using the name of the property, prefixed by the name of the original declaring class. For example:

```
class CIM_System:CIM_LogicalElement
{
        [Key]
    string Name;
};

class CIM_LogicalDevice: CIM_LogicalElement
{
        [Key]
    string DeviceID;
        [Key, Propagated("CIM_System.Name")]
    string SystemName;
};

[Association]
class CIM_SystemDevice: CIM_SystemComponent
{
        [Override ("GroupComponent"), Aggregate, Min(1), Max(1)]
    CIM_System Ref GroupComponent;
        [Override ("PartComponent"), Weak]
    CIM_LogicalDevice Ref PartComponent;
};
```

## 4.6 Qualifier Declarations

Qualifiers may be declared using the keyword "qualifier". The declaration of a qualifier allows the definition of types, default values, propagation rules (also known as Flavors), and restrictions on use.

The default value for a declared qualifier is used when the qualifier is not explicitly specified for a given schema element (explicit specification includes when the qualifier specification is inherited).

The MOF syntax allows specifying a qualifier without an explicit value. In this case, the assumed value depends on the qualifier type: booleans are true, numeric types are null, strings are null and arrays are empty.

For example the alias qualifier is declared as follows:

```
qualifier alias : string = null, scope (property, reference, method);
```

This declaration establishes a qualifier called alias. The type of the qualifier is string. It has a default value of null and may only be used with properties, references and methods.

The metaqualifiers are declared as follows:

```
Qualifier Association : boolean = false,
    Scope(class, association), Flavor(DisableOverride);

Qualifier Indication : boolean = false,
    Scope( class, indication), Flavor(DisableOverride);
```

See Appendix B for the complete list of standard qualifiers.

## 4.7 Instance Declarations

Instances are declared using the keyword sequence "instance of" and the class name. The properties of the instance may be initialized within an initialization block.

Property initialization consists of an optional list of preceding qualifiers, the name of the property and an optional value. Any properties not initialized will receive their default values as specified in the class definition, or (if no default value has been specified) the special value NULL to indicate "absence of value". For example, given the class definition:

```
class Acme_LogicalDisk
{
        [key]
    string DriveLetter;
        [Units("kilo bytes")]
    sint32 RawCapacity   =   128000;
        [write]
    string  VolumeLabel;
        [Units("kilo bytes")]
    sint32 FreeSpace;
};
```

then an instance of the above class might be declared as follows:

```
instance of Acme-LogicalDisk
{
    DriveLetter = "C";
    VolumeLabel = "myvol";
};
```

The resultant instance would take the following property values:

1. "DriveLetter" would be assigned the value "C".
2. "RawCapacity" would be assigned the default value 128000.
3. "VolumeLabel" would be assigned the value "myvol".
4. "FreeSpace" would be assigned the value NULL.

For subclasses, all of the properties in the superclass must be initialized along with the properties in the subclass. Any properties not specifically assigned in the instance block will receive either the default value for the property if there is one, or else the value NULL (if there is not one).

The values of all key properties must be specified in order for an instance to be identified and created. There is no requirement to explicitly initialize other properties. See Section 4.10.6 on page 40 on behavior when there is no property initialization.

Instances of Associations may also be defined, for example:

```
Instance of CIM_ServiceSAPDependency
{
  Dependent = "cim://root/default/Service.Name = \"mail\"";
  Antecedent = "cim://root/default/ServiceAccessPoint.Name = \"PostOffice\"";
}
```

## 4.7.1 Instance Aliasing

An alias can be assigned to an instance using the following syntax:

```
instance of Acme_LogicalDisk as $Disk
{
    // Body of instance definition here ...
};
```

Such an alias can later be used within the same MOF specification as a value for an object reference property. For more information, see Section 4.11.2 on page 41.

## 4.7.2 Object References

Object references are properties which are links or pointers to other objects (classes or instances). The value of an object reference is a string, which represents a path to another object. The path includes:

1. The namespace in which the object resides
2. The class name of the object
3. If the object represents an instance, the values of all key properties for that instance

Object reference properties are declared by "XXX ref", indicating a strongly typed reference to objects of the class with name "XXX" (or a derived class thereof).

For example:

```
        [Association]
class Acme_ExampleAssoc
{
    Acme_AnotherAssoc ref Inst1;

    Acme_Aclass ref Inst2;
};
```

In associations, object references have cardinalities — denoted using MIN and MAX qualifiers. Here are examples of UML cardinality notations and their respective combinations of MIN and MAX values:

| UML | MIN | MAX | Required MOF Text* | Description |
|---|---|---|---|---|
| * | 0 | NULL | | Many |
| 1..* | 1 | NULL | Min(1) | At least one |
| 1 | 1 | 1 | Min(1), Max(1) | One |
| 0,1 (or 0..1) | 0 | 1 | Max(1) | At most one |

See also Section 4.11.2 on page 41, on Initializing References Using Aliases.

### 4.7.3 Arrays

Arrays of any of the basic data types can be declared in the MOF specification by using square brackets after the property identifier. Fixed-length arrays indicate their length as an unsigned integer constant within the square brackets; otherwise, the array is assumed to be variable length. Arrays can be bags, ordered lists or indexed arrays. An array's type is defined by the ARRAYTYPE qualifier, whose values are BAG, ORDERED or INDEXED. The default array type is BAG.

Regarding each of the array types:

- An array of type BAG is unordered and multi-valued, allowing duplicate entries.
- An ordered list (ORDERED) is a special case of a bag, which is multi-valued and allows duplicate entries. It returns the property values in an implementation dependent, but fixed order.
- An indexed array (INDEXED) maintains the order of the elements, and could be implemented based on an integer index for each of the array values.

Note that for the Bag array type, no significance is defined for the array index other than a convenience for accessing the elements of the array. For example, there can be no assumption that the same index will return the same value for every access to the array. The only assumption is that a complete enumeration of the indices will return a complete set of values.

For the Ordered array type, the array index is significant as long as no array elements are added, deleted or changed. In this case the same index will return the same value for every access to the array. If an element is added, deleted or changed, the index of the elements might change according to the implementation-specific ordering algorithm.

The Indexed array maintains the correspondence between element position and value. Array elements can be overwritten, but not deleted.

This version of the CIM specification does not support n-dimensional arrays.

Arrays of any basic data type are legal. Arrays of references are not legal. Arrays must be homogeneous. Arrays of mixed types are not supported. In MOF, the data type of an array precedes the array name. Array size, if fixed length, is declared within square brackets, following the array name. If a variable length array is to be defined, empty square brackets follow the array name.

Arrays are declared using the following MOF syntax:

```
class A
{
  [Description("An indexed array of variable length"), ArrayType("Indexed")]
  uint8 MyIndexedArray[];

  [Description("A bag array of fixed length")]
  uint8 MyBagArray[17];
};
```

If default values are to be provided for the array elements, the following syntax is used:

```
class A
{
  [Description("A bag array property of fixed length")]
  uint8 MyBagArray[17] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17};
};
```

The following MOF presents further examples of Bag, Ordered and Indexed array declarations:

```
class Acme_Example
{
    char16 Prop1[];         // Bag (default) array of chars, Variable length

    [ArrayType ("Ordered")] // Ordered array of double-precision reals,
    real64 Prop2[];         // Variable length

    [ArrayType ("Bag")]     // Bag array containing 4 32-bit signed integers
    sint32 Prop3[4];

    [ArrayType ("Ordered")] // Ordered array of strings, Variable length
    string Prop4[] = {"an", "ordered", "list"};

        // Prop4 is variable length with default values defined at the
        // first three positions in the array

    [ArrayType ("Indexed")] // Indexed array of 64-bit unsigned integers
    uint64 Prop5[];
};
```

## 4.8 Method Declarations

A method is defined as an operation together with its signature. The signature consists of a possibly empty list of parameters and a return type. There are no restrictions on the type of parameters other than they must be one of the data types described in Section 2.2, a fixed or variable length array of one of those types, or be an object reference. Return types must be one of the data types described in Section 2.2 on page 11. Return types cannot be arrays, but otherwise are unrestricted. Syntactically, the only thing that distinguishes a method from a property is the parameter list. The fact that methods are expected to have side-effects is outside the scope of this specification.

In the following example, Start and Stop methods are defined on the Service class. Each method returns an integer value:

```
class CIM_Service:CIM_LogicalElement
{
        [Key]
    string  Name;
    string  StartMode;
    boolean Started;

    uint32  StartService();
    uint32  StopService();
};
```

In the following example, a Configure method is defined on the Physical DiskDrive class. It takes a DiskPartitionConfiguration object as a parameter and returns a boolean value:

```
class Acme_DiskDrive:CIM_Media
{
    sint32  BytesPerSector;
    sint32  Partitions;
    sint32  TracksPerCylinder;
    sint32  SectorsPerTrack;
    string  TotalCylinders;
    string  TotalTracks;
    string  TotalSectors;
    string  InterfaceType;
    boolean Configure([IN] DiskPartitionConfiguration  REF config);
};
```

## 4.9 Compiler Directives

Compiler directives are provided as the keyword **pragma**, preceded by a hash (#) character, and followed by a string parameter.

The current standard compiler directives are listed in the following table:

| Complier Directive | Interpretation |
|---|---|
| #pragma namespace( ) | Determines the current default namespace. |
| #pragma source( ) | Defined in Chapter 5. |
| #pragma nonlocal( ) | Defined in Chapter 5. |
| #pragma include( ) | Has a file name as a parameter. The file is assumed to be a MOF file. The pragma has the effect of textually inserting the contents of the include file at the point where the include pragma is encountered. |
| #pragma locale( ) | Declares the locale used for a particular MOF file. The locale is specified as a parameter of the form ll_cc, where ll is the language code based on ISO/IEC 639, and cc is the country code based on ISO/IEC 3166. When the pragma is not specified, the assumed locale is "en_US".<br><br>It is important to note that this pragma does not apply to the syntax structures of MOF. Keywords, such as "class" and "instance", are always in en_US. |
| #pragma instancelocale( ) | Declares the locale used for instances described in a MOF file. This pragma specifies the locale when "INSTANCE OF" MOF statements include string or char16 properties, and the locale is not the same as the locale specified by a #pragma locale() statement. The locale is specified as a parameter of the form ll_cc where ll is the language code based on ISO/IEC 639, and cc is the country code based on ISO/IEC 3166. |

**Table 4-2** Standard Compiler Directives

Additional pragma directives may be added as a MOF extension mechanism. Unless standardized in a future CIM specification, such new pragma definitions must be considered vendor-specific. Use of non-standard pragmas will affect interoperability of MOF import and export functions.

When a qualifier value is derived from either a qualifier or a pragma, the qualifier overrides the pragma.

## 4.10 Value Constants

The constant types supported in the MOF syntax are described in the subsections that follow. These are used in initializers for classes and instances, and in the parameters to named qualifiers.

A formal specification of the representation may be found in Appendix A.

### 4.10.1 String Constants

A string constant is a sequence of zero or more UCS-2 characters enclosed in double-quotes ("). A double-quote is allowed within the value as long as it is preceded immediately by a backslash (\).

For example:

```
"This is a string"
```

Successive quoted strings are concatenated as long as only white space or a comment intervenes:

```
"This" "becomes a long string"
"This" /* comment */ "becomes a long string"
```

The escape sequences \n, \t, and \r are recognized as legal characters within a string.

The complete set is:

```
\b        // \x0008: backspace BS
\t        // \x0009: horizontal tab HT
\n        // \x000A: linefeed LF
\f        // \x000C: form feed FF
\r        // \x000D: carriage return CR
\"        // \x0022: double quote "
\'        // \x0027: single quote '
\\        // \x005C: backslash \
\x<hex>   // where <hex> is one to four hex digits
\X<hex>   // where <hex> is one to four hex digits
```

The character set of the string depends on the character set supported by the local installation. While the MOF specification may be submitted in UCS-2 form (see UCS Transformation Format-8 — UTF-8 in **Referenced Documents**), the local implementation may only support ANSI, and vice-versa. Therefore, the string type is unspecified and dependent on the character set of the MOF specification itself. If a MOF specification is submitted using UCS-2 characters outside of the normal ASCII range, then the implementation may have to convert these characters to the locally equivalent character set.

### 4.10.2 Character Constants

Character and wide-character constants are specified as follows:

```
"a"
"\en"
"1"
"\x32"
```

Note that trigraphs such as \020 are not supported.

Integer values can also be used as character constants, as long as they are within the numeric range of the character type. For example, wide-character constants must fall within the range 0 to 0xFFFF.

### 4.10.3 Integral Constants

Integer constants may be either decimal, binary, octal or hexadecimal. For example, the following are all legal:

```
1000
–12310
0x100
01236
100101B
```

Note that binary constants have a series of 1 and 0 digits, with a "b" or "B" suffix to indicate that the value is binary.

The number of digits permitted depends on the current type of the expression. For example, it is not legal to assign the constant 0xFFFF to a property of type uint8.

### 4.10.4 Floating-Point Constants

Floating point constants are declared as specified by the IEEE in ANSI/IEEE Std 754-1985 (see **Referenced Documents**).

For example, the following are legal:

```
3.14
–3.14
–1.2778E+02
```

The range for floating point constants depends on whether float or double properties are used and must fit within the range specified for IEEE 4-byte and 8-byte floating point values, respectively.

### 4.10.5 Object Ref Constants

Object references are simple URL-style links to other objects (which may be classes or instances). They take the form of a quoted string containing an object path. The object path is a combination of a namespace path and the model path.

For example:

```
"//./root/default:LogicalDisk.SystemName=
"//./root/default:NetworkCard=2"
```

An object reference can also be an alias. See Section 4.11.2 on page 41 for more details.

### 4.10.6 NULL

All types can be initialized to the predefined constant NULL, which indicates no value has been provided at all. The details of the internal implementation of the NULL value are not mandated by this document.

## 4.11 Initializers

Initializers are used both in class declarations for default values, and instance declarations to initialize a property to a value. The format of intializer values is as specified in Section 4.10 on page 39. The initializer value *must* match the property data type (the only exception is the NULL value, which may be used for any data type).

### 4.11.1 Initializing Arrays

Arrays can be defined to be of type, Bag, Ordered or Indexed, and can be initialized by specifying their values in a comma-separated list (as in the C programming language). The list is delimited with braces { }.

For example, given the class definition:

```
class Acme_ExampleClass
{
    [ArrayType ("Indexed")]
    string ip_addresses [];    // Indexed array of variable length
    sint32 sint32_values [10]; // Bag array of fixed length = 10
};
```

then a valid instance declaration is:

```
instance of Acme_ExampleClass
{
    ip_addresses = { "1.2.3.4", "1.2.3.5", "1.2.3.7" };

        // ip_address is an indexed array of at least 3 elements, where
        // values have been assigned to the first three elements of the
        // array

    sint32_values = { 1, 2, 3, 5, 6 };
};
```

Refer to Section 4.7.3 on page 35 for additional information on declaring arrays, and the distinctions between bags, ordered arrays and indexed arrays.

### 4.11.2 Initializing References using Aliases

Aliases are symbolic references to an object located elsewhere in the MOF specification. They only have significance within the MOF specifiation in which they are defined, and are only used at compile time to facilitate establishment of references. They are not available outside of the MOF specification.

Classes and instances may be assigned an alias in the manner described in Section 4.7.1 on page 34.

Aliases are identifiers which begin with the "$" symbol. When a subsequent reference to that instance is needed for an object reference property, the identifier is used in place of an explicit initializer.

Assuming that $Alias1 and $Alias2 have been declared as aliases for instances, and the obref1 and obref2 properties are object references, this example shows how the object references could be assigned to point to the aliased instances:

```
instance of Acme_AnAssociation
{
    strVal = "ABC";
    obref1 = $Alias1;
    obref2 = $Alias2;
};
```

Forward-referencing and circular aliases are permitted.

*Chapter 5*

# *Naming*

## 5.1 Overview

Because CIM is not bound to a particular technology or implementation, it promises to facilitate sharing management information between a variety of management platforms. The CIM Naming mechanism was defined to address enterprise-wide identification of objects, as well as the sharing of management information.

CIM Naming addresses these requirements:

1. Ability to locate and uniquely identify any object in an enterprise:
   - Unambiguous enumeration of all objects
   - Ability to determine when two object names reference the same entity
   - Location transparency (no need to understand which management platforms proxy other platforms' instrumentation)
2. Allow sharing of objects and instance data among management platforms:
   - Allow creation of different scoping hierarchies which vary by "time" (for example, a "current" vs. "proposed" scoping hierarchy)
3. Facilitate move operations between object trees (including within a single management platform):
   - Hide underlying management technology/provide technology transparency for the domain-mapping environment
   - Object name identifiable regardless of instrumentation technology
   - Allowing different names for DMI *versus* SNMP objects requires the management platform to understand how the underlying objects are implemented

   The KEY qualifier is the CIM Metamodel mechanism used to identify the properties that uniquely identify an instance of a class (and indirectly an instance of an association). CIM Naming enhances this base capability by:

   - Introducing the WEAK and PROPOGATED qualifiers to express situations in which the keys of one object are to be propagated to another object
   - Introducing the SOURCE pragma and qualifier
     ("namespacetype://namespace_handle")
     to allow details about the implementation source to be recorded in a MOF file
   - Introducing the NONLOCAL qualifier
     ("namespacetype://namespace_handle")
     to reference an object instance kept in another implementation

## 5.2 Background

CIM MOF files can contain definitions of instances, classes or both, as illustrated in Figure 5-1.

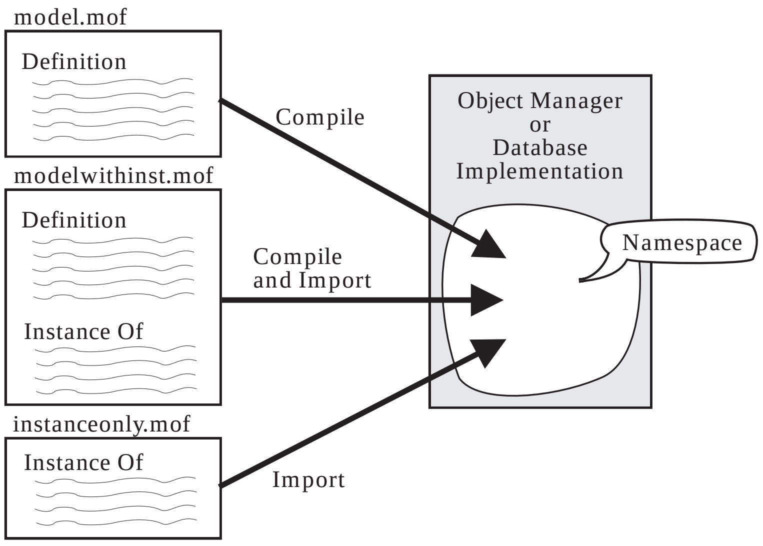


**Figure 5-1** Definitions of Instances and Classes

MOF files can be used to populate a technology that understands the semantics and structure of CIM. When a MOF file is consumed by a particular implementation, there are two operations that are actually being performed, depending on the file's content. First, a compile or definition operation is performed to establish the structure of the model. Second, an import operation is performed to insert instances into the platform or tool.

Once the compile and import are completed, the actual instances are manipulated using the native capabilities of the platform or tool. In other words, in order to manipulate an object (for example, change the value of a property), one must know the type of platform the information was imported into, the APIs or operations used to access the imported information, and the name of the platform instance that was actually imported. For example, the semantics become:

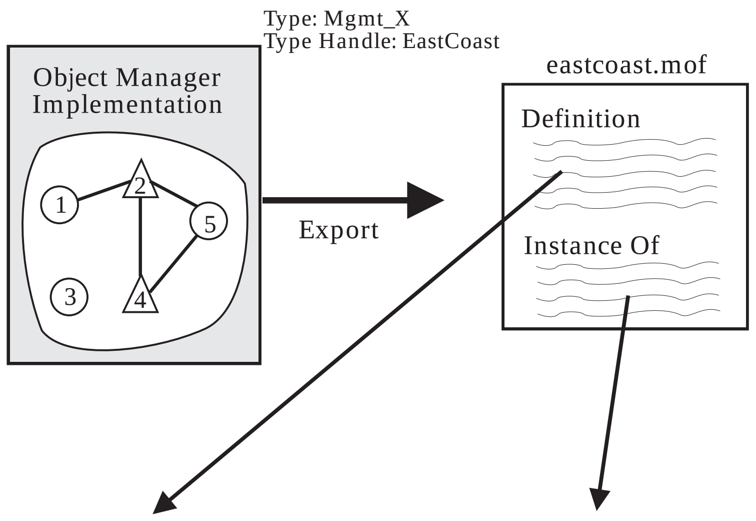
> Set the Version property of the Logical Element object with Name="Cool" in the relational database named LastWeeksData to "1.4.0".

The contents of a MOF file are loaded into a namespace that provides a domain (in other words, a container), in which the instances of the classes are guaranteed to be unique per the KEY qualifier definitions. The term namespace is used to refer to an implementation that provides such a domain.

Namespaces can be used to:

- Define chunks of management information (objects and associations) to limit implementation resource requirements, such as database size.
- Define views on the model for applications managing only specific objects, such as hubs.
- Pre-structure groups of objects for optimized query speed.

Another viable operation is exporting from a particular management platform — see Figure 5-2 on page 45. Essentially, this operation creates a MOF file for all or some portion of the information content of a platform.



```
[ ]
class Figs_Circle
{
   [ key ] uint32  Name;
           string   Color;   };

class Figs_Triangle
{
   [ key ] uint32  Label;
           string   Color ;
           uint32  Area;
};

[Association]  class Figs_CircleToTriangle
{
   Figs_Circle REF ACircle;
   Figs_Triangle REF ATriangle;
};

[Association]  class Figs_Covers
{
   Figs_Triangle REF Over;
   Figs_Triangle  REF Under;
};
```

```
instance of Figs_Triangle {Label=2 ; Color="Blue";Area=12 };
instance of Figs_Triangle {Label=4 ; Color="Blue";Area=12 };
instance of Figs_Circle { Name=1 ; Color="Blue" };
instance of Figs_Circle { Name=3 ; Color="Blue" };
instance of Figs_Circle { Name=5 ; Color="Blue" };

instance of Figs_CircleToTriangle
{   ACircle =  "Circle.Name=1";
    ATriangle =  "Triangle.Label=2";  };

instance of Figs_ CircleToTriangle
{   ACircle =  "Circle.Name=5";
    ATriangle =  "Triangle.Label=2";  };

instance of Figs_ CircleToTriangle
{   ACircle =  "Circle.Name=5";
    ATriangle =  "Triangle.Label=4";  };

instance of Figs_ Covers
{   Over =  "Triangle.Label=2";
    Under =  "Triangle.Label=4";  };
```

**Figure 5-2** Exporting to MOF

For example, information is exchanged when the source system is of type Mgmt_X and its name is EastCoast. The export produces a MOF file with the circle and triangle definitions and instances 1, 3, 5 of the circle class and instances 2, 4 of the triangle class. This MOF file is then compiled and imported into the management platform of type Mgmt_ABC with the name AllCoasts. See Figure 5-3 on page 46.

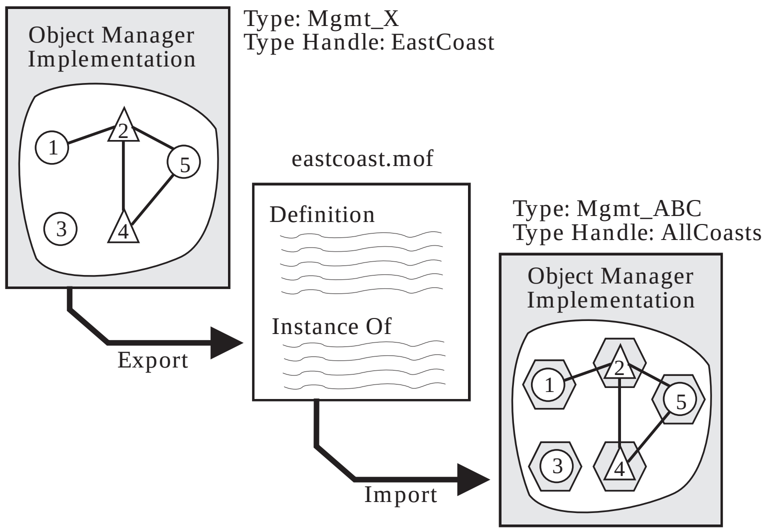


**Figure 5-3** Information Exchange

The import operation involves storing the information in a local or native format of Mgmt_ABC so its native operations can be used to manipulate the instances. The transformation to a native format is shown in Figure 5-3 by wrapping the five instances in hexagons. The transformation process must maintain the original keys.

**Management Tool Responsibility for Export Operation**

The management tool must be able to create unique key values for each distinct object it places in the MOF file.

For each instance placed in the MOF file, the management tool must maintain a mapping from the MOF file keys to the native key mechanism.

**Management Tool Responsibility for Import Operation**

The management tool must be able to map the unique keys found in the MOF file to a set of locally-understood keys.

## 5.3 Weak Associations: Supporting Key Propagation

CIM provides a mechanism to name instances within the context of other object instances. For example, if a management tool is handling a local system, then it can refer to the C drive or the D drive. However, if a management tool is handling multiple machines, it must refer to the C drive on machine X and the C drive on machine Y. In other words, the name of the drive must include the name of the hosting machine. CIM supports the notion of weak associations to specify this type of key propagation.

A weak association is defined using a qualifier. For example:

```
Qualifier Weak: boolean = false, Scope(reference), Flavor(DisableOverride);
```

The key(s) of the referenced class includes the key(s) of the other participants in the WEAK association. This situation occurs when the referenced class identity depends on the identity of other participants in the association.

> **Usage Rule:** This qualifier can only be specified on one of the references defined for an association. The Weak referenced object is the one that depends on the other object for identity.

Figure 5-4 shows an example. There are three classes: ComputerSystem, OperatingSystem and Local User. The Operating System class is weak with respect to the Computer System class, since the runs association is marked weak. Similarly, the Local User class is weak with respect to the Operating System class, since the association is marked weak.

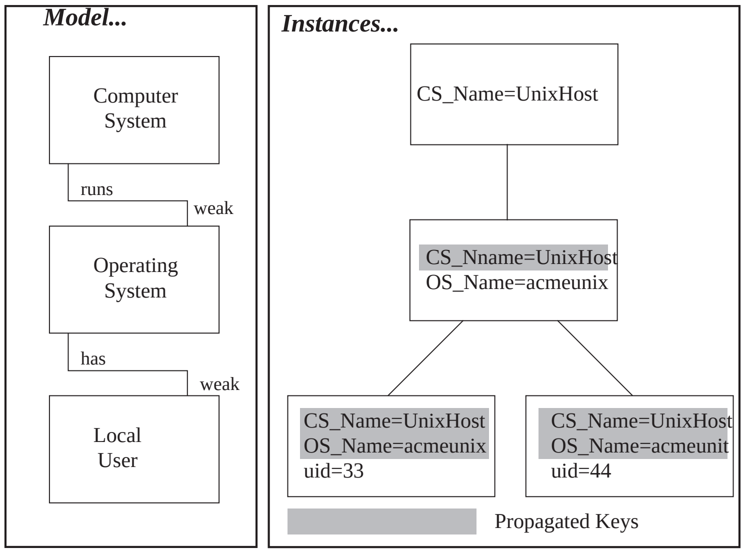


**Figure 5-4** Example of Weak Association

In the context of a weak association definition, the Computer System class is a scoping class for the Operating System class, since its keys are propagated to the Operating System class. The Computer System and the Operating System classes are both scoping classes for the Local User class, since the Local User class gets keys from both. Finally, the Computer System is referred to as a Top Level Object (TLO) because it is not weak with respect to any other class. The fact that a particular class is a top-level object is inferred because no references to that class are marked with the WEAK qualifier. In addition, Top Level Objects must have the possibility of an enterprise-wide, unique key. An example may be a computer's IP address in a company's enterprise-wide IP network. The goal of the TLO concept is to achieve uniqueness of keys in the model path portion of the object name. In order to come as close as possible to this goal, TLO must have relevance in an enterprise context.

Objects in the scope of another object can in turn be a scope for other objects; hence, all model object instances are arranged in directed graphs with the Top Level Object's (TLO's) as peer roots. The structure of this graph - in other words, which classes are in the scope of another given class - is defined as part of CIM by means of associations qualified with the WEAK qualifier.

### Referencing Weak Objects

A reference to an instance of an association includes the propagated keys. The properties must be qualified by the scoping class name when the property name is not unique. This implies the following for the association in the previous example:

```
instance of Acme_has
{
  anOS = "ComputerSystem.Name=UnixHost,OperatingSystem.Name=acmeunit";
  aUser = "ComputerSystem.Name=UnixHost,OperatingSystem.Name=acmeunit,uid=33";
};
```

## 5.4 Naming CIM Objects

Since CIM allows for multiple implementations, it is not sufficient to think of the name of an object as just the combination of properties that have the KEY qualifier. The name must also identify the implementation that actually hosts the objects. The object name consists of the Namespace Path, which provides *access to*
a CIM implementation, plus the Model Path, which provides full *navigation within* the CIM schema. The namespace path is used to locate a particular name space. The details of the namespace path are dependent on a particular implementation. The model path is the concatenation of the properties of a class that are qualified with the KEY qualifier. When the class is weak with respect to another class, the model path includes all key properties from the scoping objects.

### 5.4.1 Namespace Path

A Namespace path references a namespace within an implementation that is capable of hosting CIM objects.

A Namespace path resolves to a namespace hosted by a CIM-Capable implementation (in other words, a CIM Object Manager). Unlike the Model Path, the details of the Namespace path are implementation-specific. Therefore, the Namespace path provides two pieces of information: it identifies the type of implementation or namespace type, and it provides a handle that references a particular implementation or namespace handle.

#### *5.4.1.1 Namespace Type*

The namespace type classifies or identifies the type of implementation. The provider of such an implementation is responsible for describing the access protocol for that implementation. This is analogous to specifying http or ftp in a browser.

Fundamentally, a namespace type implies an access protocol or API set that can be used to manipulate objects. These APIs would typically support:

- Generating a MOF file for a particular scope of classes and associations
- Importing a MOF file
- Manipulating instances

A particular management platform may have a variety of ways to access management information. Each of these ways must have a namespace type definition. Given this type, there would be an assumed set of mechanisms for exporting, importing and updating instances.

#### *5.4.1.2 Namespace Handle*

The Namespace handle identifies a particular instance of the type of implementation. This handle must resolve to a namespace within an implementation.

The details of the handle are implementation-specific. It might be a simple string for an implementation that supports one namespace, or it might be a hierarchical structure if an implementation supports multiple namespaces. Either way, it resolves to a namespace.

It is important to note that some implementations can support multiple namespaces. In this case, the implementation-specific reference must resolve to a particular namespace within that implementation.

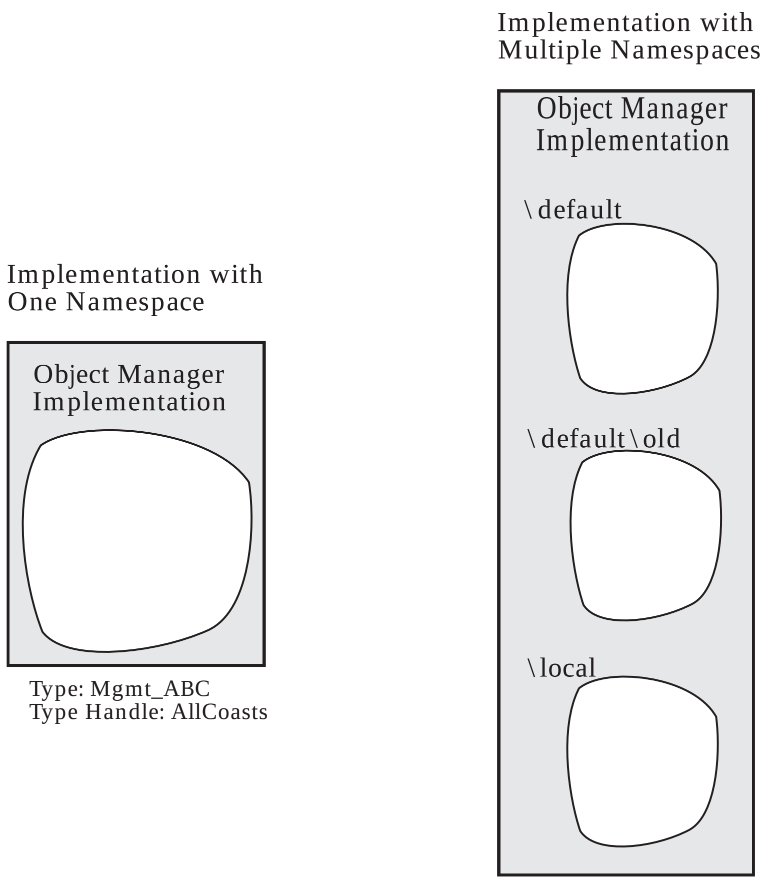


**Figure 5-5** Namespaces

There are two important observations to make:

1. Namespaces can overlap with respect to their contents.
2. An object in one name space, which has the same model path as an object in another name, space does not guarantee that the objects are representing the same reality.

## 5.4.2 Model Path

The object name constructed as a scoping path through the CIM schema is referred to as a Model Path. It is solely described by CIM elements and is absolutely implementation-independent. It is used to describe the path to a particular object or to identify a particular object within a namespace. The name of any object is a concatenation of named key property values, including all key values of its scoping objects.

## 5.5 Specifying Object Names in MOF Files

The object name can be used as the value for object references and for object queries.

### 5.5.1 Synchronizing Namespaces

When a MOF is loaded into a system that is able to access and manipulate the source implementation, a higher level of integration is possible between two CIM-based implementations. In particular, the receiving implementation can synchronize changes with the sending implementation. This situation is shown in Figure 5-6, and requires a way to record information about the namespace path of the source in the MOF. The arrow labeled "Dynamic Access to Loaded Information" implies that Mgmt_ABC has the capability to access information about an instance of Mgmt_X because it understands Mgmt_X's access protocol. All it must know is the handle (namespace path) for the source.

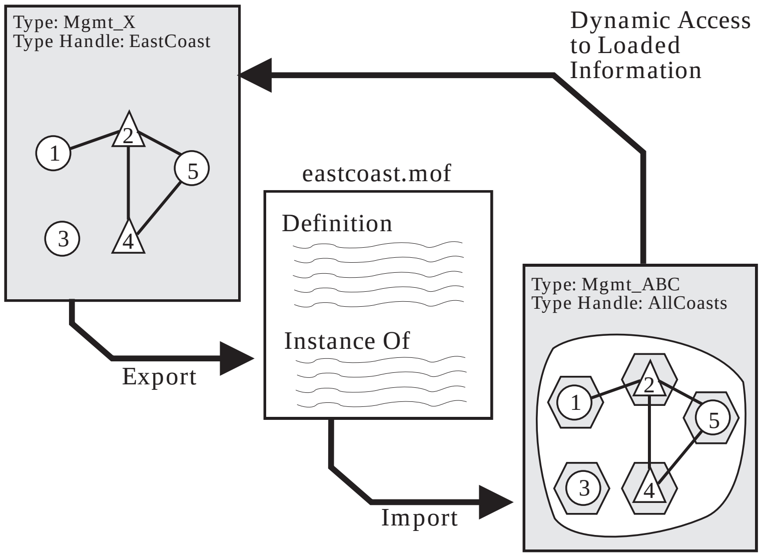


**Figure 5-6** Namespace Path

The namespace path can be provided in one of two ways:

- A qualifier on each object and association
- A pragma

The value for the pragma and the qualifier is exactly the same:

```
Source(<namespacetype>:  <namespace_handle>)
```

When the information is provided as a pragma, it is assumed to be the same for all instances in the MOF file. This pragma is shown in Figure 5-7 for the circle and triangle example.

```
#pragma source("Mgmt_X:EastCoast")

class Figs_Circle {
   [key] uint32 Name;
        string Color;
};

class Figs_Triangle {
   [key] uint32 Label;
        string Color;
        uint32 Area;
};

[Association]
class Figs_CircleToTriangle {
   Figs_Circle REF ACicrle;
   Figs_Triangle REF ATriangle;
};

[Association]
class Figs_Covers {
   Figs_Triangle REF Over;
   Figs_Triangle REF Under;
};
```

```
instance of Figs_Triangle {Label=2;Color="Blue";Area=12};
instance of Figs_Triangle {Label=4;Color="Blue";Area=12};
instance of Figs_Circle {Name=1;Color="Blue"};
instance of Figs_Circle {Name=3;Color="Blue"};
instance of Figs_Circle {Name=5;Color="Blue"};

instance of Figs_CircleToTriangle {
   ACircle="Circle.Name=1";
   ATriangle="Triangle.Label=2";
};

instance of Figs_CircleToTriangle {
   ACircle="Circle.Name=5";
   ATriangle="Triangle.Label=2";
};

instance of Figs_CircleToTriangle {
   ACircle="Circle.Name=5";
   ATriangle="Triangle.Label=4";
};

instance of Figs_Covers {
   Over="Triangle.Label=2";
   Under="Triangle.Label=4";
};
```

**Figure 5-7** Pragma Example

The import operation must preserve namespace path information so if either this platform or another platform understands how to manipulate an implementation of type <namespacetype> and has access to the <namespace_handle>, it can manipulate one or more of the instances in the source.

The namespace path can also be specified using the instance-based Source qualifier. This qualifier marks a particular object or an association. This is illustrated in Figure 5-8. Note that when a pragma is specified and a qualifier is also specified, the qualifier overrides the pragma.

```
class Figs_Circle {
   [key] uint32 Name;
        string Color;
};

class Figs_Triangle {
   [key] uint32 Label;
        string Color;
        uint32 Area;
};

[Association]
class Figs_CircleToTriangle {
   Figs_Circle REF ACicrle;
   Figs_Triangle REF ATriangle;
};

[Association]
class Figs_Covers {
   Figs_Triangle REF Over;
   Figs_Triangle REF Under;
};
```

```
[source("Mgmt_X:EastCoast")]
instance of Figs_Triangle {Label=2;Color="Blue";Area=12};

[source("Mgmt_X:EastCoast")]
instance of Figs_Triangle {Label=4;Color="Blue";Area=12};

[source("Mgmt_X:EastCoast")]
instance of Figs_Circle {Name=1;Color="Blue"};

[source("Mgmt_X:EastCoast")]
instance of Figs_Circle {Name=3;Color="Blue"};

[source("Mgmt_X:EastCoast")]
instance of Figs_Circle {Name=5;Color="Blue"};

[source("Mgmt_X:EastCoast")]
instance of Figs_CircleToTriangle {
   ACircle="Circle.Name=1";
   ATriangle="Triangle.Label=2";
};

[source("Mgmt_X:EastCoast")]
instance of Figs_CircleToTriangle {
   ACircle="Circle.Name=5";
   ATriangle="Triangle.Label=2";
};

[source("Mgmt_X:EastCoast")]
instance of Figs_CircleToTriangle {
   ACircle="Circle.Name=5";
   ATriangle="Triangle.Label=4";
};

[source("Mgmt_X:EastCoast")]
instance of Figs_Covers {
   [nonlocal("Mgmt_X:EastCoast")]
   Over="Triangle.Label=2";
   Under="Triangle.Label=4";
};
```

**Figure 5-8** Namespace Path Example

### 5.5.2 Building References Between Management Systems

The Nonlocal instance qualifier for references allows a targeted management system to selectively import instances in a MOF file. This is used when a targeted management system knows how to access a source management platform (in other words, it has verified, using the source pragma or qualifier, that it knows how to access the source platform) and it does not want to store some class instances locally. Using the circle and triangle MOF as an example, the target management system, Mgmt_ABC, only wants to store circle information locally. When a Mgmt_ABC user requests information about a triangle, the Mgmt_ABC implementation contacts the source platform Mgmt_X to get the instance information. This is illustrated in Figure 5-9.

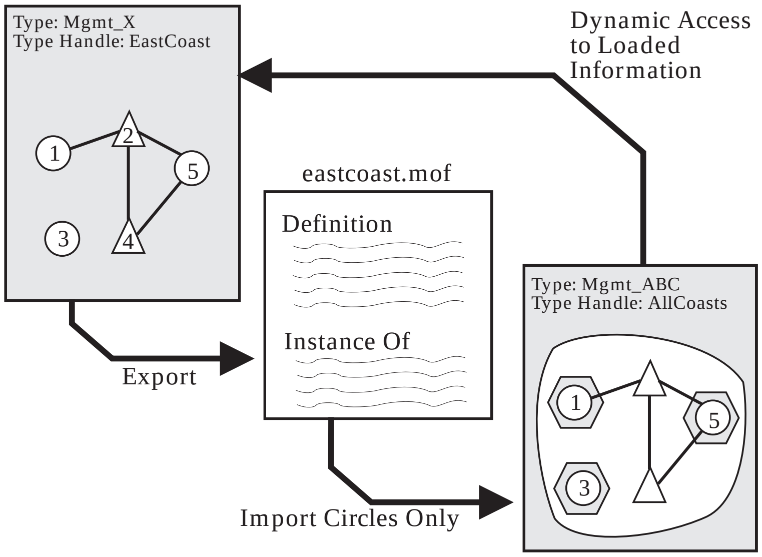


**Figure 5-9** References Between Management Systems

The Nonlocal qualifier is similar to the Source qualifier since its value is a string:

```
<namespacetype>:<namespacehandle>
```

The content of Mgmt_ABC after importing only circle information looks like the example in Figure 5-10.

In particular, the two instances of triangle are not imported, and the references to triangle in the associations are also marked with the nonlocal qualifier.

The above schema also allows intelligent import operations to avoid importing all the objects if there are associations between the objects.

```
class Figs_Circle {
   [key] uint32 Name;
        string Color;
};

class Figs_Triangle {
   [key] uint32 Label;
        string Color;
        uint32 Area;
};

[Association]
class Figs_CircleToTriangle {
   Figs_Circle REF ACicrle;
   Figs_Triangle REF ATriangle;
};

[Association]
class Figs_Covers {
   Figs_Triangle REF Over;
   Figs_Triangle REF Under;
};
```

```
instance of Figs_Circle {Name=1; Color="Blue"};
instance of Figs_Circle {Name=3; Color="Blue"};
instance of Figs_Circle {Name=5; Color="Blue"};

instance of Figs_CircleToTriangle {
   ACircle="Circle.Name=1";
   [nonlocal("Mgmt_X:EastCoast")]
   ATriangle="Triangle.Label=2";
};

instance of Figs_CircleToTriangle {
   ACircle="Circle.Name=5";
   [nonlocal("Mgmt_X:EastCoast")]
   ATriangle="Triangle.Label=2";
};

instance of Figs_CircleToTriangle {
   ACircle="Circle.Name=5";
   [nonlocal("Mgmt_X:EastCoast")]
   ATriangle="Triangle.Label=4";
};

instance of Figs_Covers {
   [nonlocal("Mgmt_X:EastCoast")]
   Over="Triangle.Label=2";
   [nonlocal("Mgmt_X:EastCoast")]
   Under="Triangle.Label=4";
};
```

**Figure 5-10** Example of Nonlocal Qualifier

Chapter 6

# Mapping Existing Models into CIM

Existing models have their own metamodel and model. There are three types of mapping that can occur between metaschemas:

- Technique
- Recast
- Domain

Each of these mappings can be applied when converting the MIF syntax to MOF syntax.

## 6.1 Technique Mapping

A technique mapping provides a mapping that uses the CIM metamodel constructs to describe the source modeling technique’s metaconstructs (for example, MIF, GDMO, SMI). Essentially, the CIM metamodel is actually a meta-metamodel for the source technique.

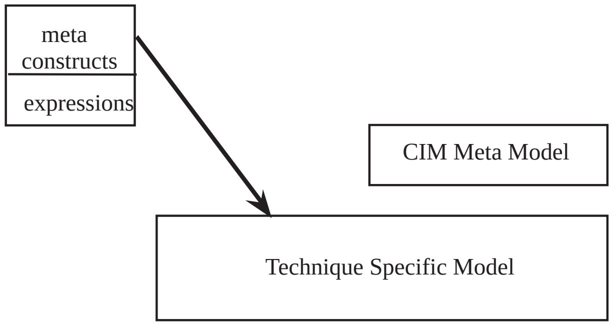


**Figure 6-1** Technique Mapping Example

The DMTF uses the management information format of MIF as the metamodel to describe desktop management information in a common way. Therefore, it is meaningful to describe a technique mapping in which the CIM metamodel is used to describe the MIF syntax.

The mapping presented here takes the important types that can appear in a MIF file and then creates classes for them. Thus, component, group, attribute, table, and enum are expressed in the CIM metamodel as classes. In addition, associations are defined to document how these are combined. Figure 6-2 illustrates the results.

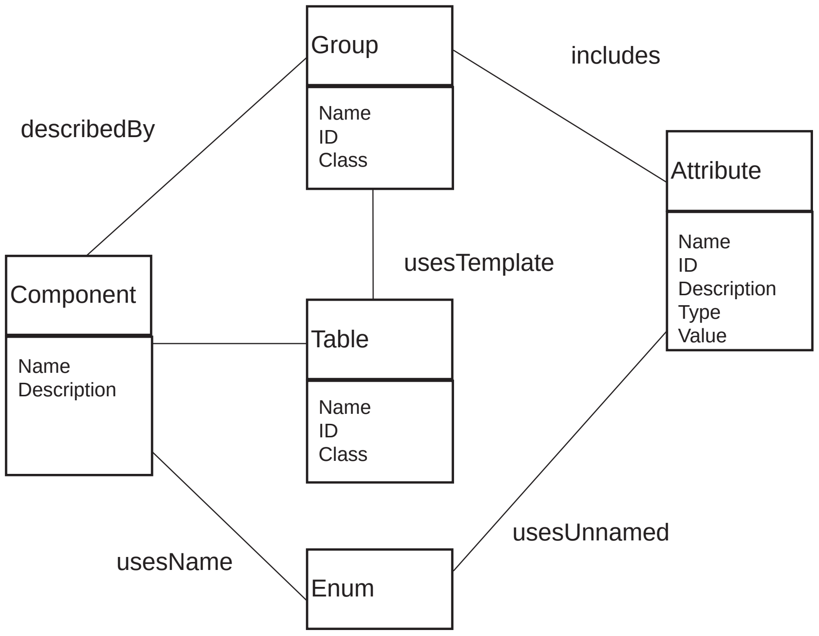


**Figure 6-2** MIF Technique Mapping Example

## 6.2 Recast Mapping

A recast mapping provides a mapping of the sources' metaconstructs into the targeted metaconstructs, so that a model expressed in the source can be translated into the target. The major design work is to develop a mapping between the sources' metamodel and the CIM metamodel. Once this is done the source expressions are recast.

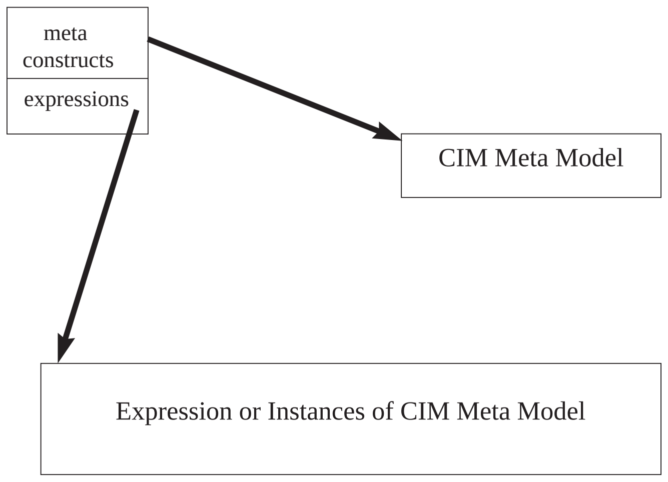


**Figure 6-3** Technique Mapping Results

The following is an example of a recast mapping for MIF, assuming:

```
DMI attributes -> CIM properties
DMI key attributes -> CIM key properties
DMI groups -> CIM classes
DMI components -> CIM classes
```

The standard DMI ComponentID group might be recast into a corresponding CIM class:

```
Start Group
Name = "ComponentID"
Class = "DMTF|ComponentID|001"
ID = 1
Description = "This group defines the attributes common to all "
          "components.  This group is required."
Start Attribute
    Name = "Manufacturer"
    ID = 1
    Description = "Manufacturer of this system."
    Access = Read-Only
    Storage = Common
    Type = DisplayString(64)
    Value = ""
End Attribute
Start Attribute
    Name = "Product"
    ID = 2
    Description = "Product name for this system."
    Access = Read-Only
    Storage = Common
    Type = DisplayString(64)
    Value = ""
End Attribute
Start Attribute
    Name = "Version"
    ID = 3
    Description = "Version number of this system."
    Access = Read-Only
    Storage = Specific
    Type = DisplayString(64)
    Value = ""
End Attribute
Start Attribute
    Name = "Serial Number"
    ID = 4
    Description = "Serial number for this system."
    Access = Read-Only
    Storage = Specific
    Type = DisplayString(64)
    Value = ""
End Attribute
Start Attribute
    Name = "Installation"
    ID = 5
    Description = "Component installation time and date."
    Access = Read-Only
    Storage = Specific
    Type = Date
    Value = ""
End Attribute
```

```
Start Attribute
    Name = "Verify"
    ID = 6
    Description = "A code that provides a level of verification that"
                "the component is still installed and working."
Access = Read-Only
    Storage = Common
    Type = Start ENUM
           0 = "An error occurred; check status code."
           1 = "This component does not exist."
           2 = "Verification is not supported."
           3 = "Reserved."
           4 = "This component exists, but the functionality is untested."
           5 = "This component exists, but the functionality is unknown."
           6 = "This component exists, and is not functioning correctly."
           7 = "This component exists, and is functioning correctly."
    End ENUM
    Value = 1
End Attribute
End Group
```

A corresponding CIM class might be the following. Note that properties in the example include an ID qualifier to represent the corresponding DMI attribute's ID. Here, a user-defined qualifier may be necessary.

```
[Name ("ComponentID"), ID (1), Description (
    "This group defines the attributes common to all components.  "
    "This group is required.")]

class DMTF|ComponentID|001 {
    [ID (1), Description ("Manufacturer of this system."), maxlen
(64)]
    string Manufacturer;
    [ID (2), Description ("Product name for this system."), maxlen
(64)]
    string Product;
    [ID (3), Description ("Version number of this system."), maxlen
(64)]
    string Version;
    [ID (4), Description ("Serial number for this system."), maxlen
(64)]
    string Serial_Number;
    [ID (5), Description("Component installation time and date.")]
    datetime Installation;
    [ID (6), Description("A code that provides a level of verification"
             "that the component is still installed and working."),
             Value (1)]
    string Verify;
};
```

## 6.3 Domain Mapping

A domain mapping takes a source expressed in a particular technique and maps its content into either the core, common, or extension sub-schemas of the CIM. This mapping does not rely heavily on a meta-to-meta mapping. It is primarily a content-to-content mapping. In our case, the mapping is actually a re-expression of some content in a more common way using a more expressive technique.

This is an example of how CIM properties can be supplied by DMI, using information from the DMI disks group ("DMTF|Disks|002"). For a hypothetical CIM disk class, the CIM properties are expressed as shown in Table 6-1.

| **CIM "Disk" property** | **Can be sourced from DMI group/attribute** |
|---|---|
| StorageType | "MIF.DMTF\|Disks\|002.1" |
| StorageInterface | "MIF.DMTF\|Disks\|002.3" |
| RemovableDrive | "MIF.DMTF\|Disks\|002.6" |
| RemovableMedia | "MIF.DMTF\|Disks\|002.7" |
| DiskSize | "MIF.DMTF\|Disks\|002.16" |

**Table 6-1** Domain Mapping Example: DMI to CIM

## 6.4 Mapping Scratch Pads

In general, when the content of models are mapped between different metaschemas, information gets lost or is missing. To fill this gap, "scratch pads" are expressed in the CIM metamodel using qualifiers, which are actually extensions to the metamodel (for example, see the Mapping MIF Attributes and Mapping SNMP Variables sections). These scratch pads are critical to the exchange of core, common and extension model content with the various technologies used to build management applications.

*Chapter 7*

# *Repository Perspective*

## 7.1 Overview

This Chapter provides a basic description of a repository and a complete picture of the potential exploitation of it. A repository stores definitional and/or structural information, and includes the capability to extract the definitions in a form that is useful to application developers. Some repositories allow the definitions to be imported into and exported from the repository in multiple forms. The notions of importing and exporting definition can be refined so that they distinguish between three types of mappings.

Using the mapping definitions in Chapter 6, the repository can be organized into the four partitions (meta, technique, recast and domain), as shown in Figure 7-1.

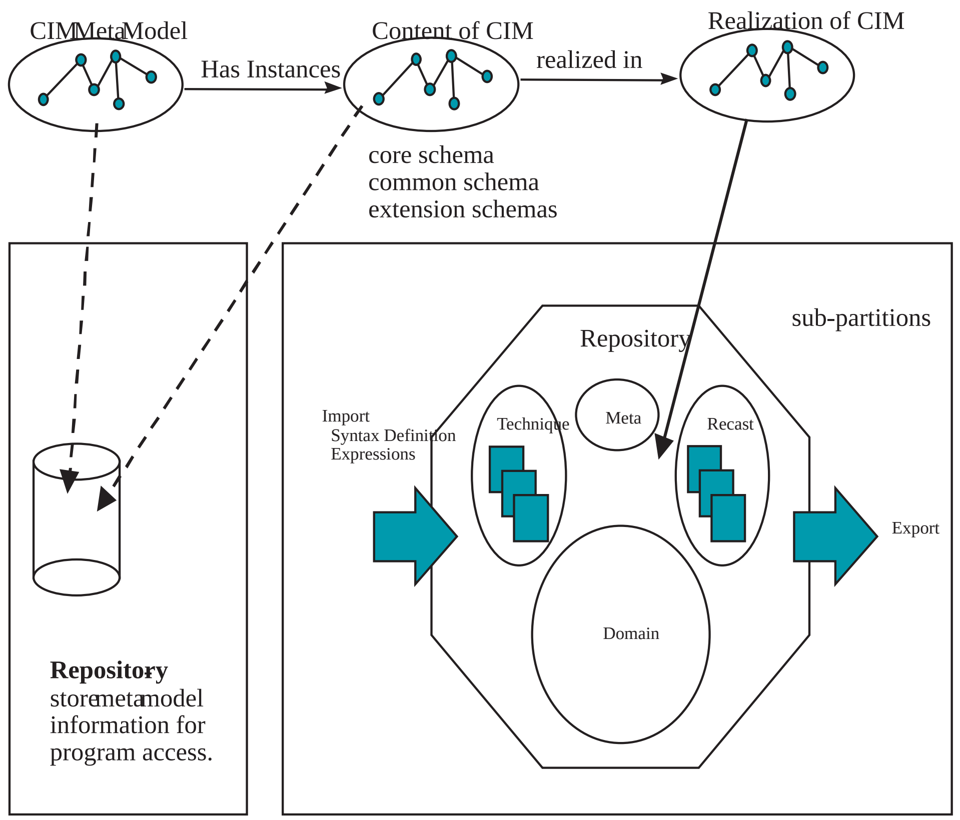


**Figure 7-1** Repository Partitions

The repository partitions have the following characteristics:

- Each partition is disjoint.
  - — The Meta partition refers to the definitions of the CIM metamodel.

   - The Technique partition refers to definitions that are loaded using technique mappings.
   - The Recast partition refers to definitions that are loaded using recast mappings.
   - The Domain partition refers to the definitions that are associated with the core, common, and Extension schemas.

- The Technique and Recast partitions can be organized into multiple sub-partitions in order to capture each source uniquely. For example, there would be a Technique sub-partition for each unique metalanguage encountered (that is, one for MIF, GDMO, SMI, and so on). In the Recast partition there would be a sub-partition for each metalanguage.
- The act of importing the content of an existing source can result in entries in the Recast or Domain partition.

## 7.2 DMTF MIF Mapping Strategies

Assume the metamodel definition and the baseline for the CIM schema are complete. The next step is to map another source of management information (such as standard groups) into the repository. The primary objective is to do the work required to import one or more of the standard group(s).

The possible import scenarios for a DMTF standard group are:

1. *To Technique Partition:*
   Create a technique mapping for the MIF syntax. This mapping would be the same for all standard groups and would only need to be updated if the MIF syntax changed.
2. *To Recast Partition:*
   Create a recast mapping from a particular standard group into a sub-partition of the recast partition. This mapping would allow the entire contents of the selected group to be loaded into a sub-partition of the recast partition. The same algorithm can be used to map additional standard groups into that same sub-partition.
3. *To Domain Partition:*
   Create a Domain Mapping for the content of a particular standard group that overlaps with the content of the CIM schema.
4. *To Domain Partition:*
   Create a Domain Mapping for the content of a particular standard group that does not overlap with CIM's schema into an extension sub-schema.
5. *To Domain Partition:*
   Propose extensions to the content of the CIM schema and then perform Steps 3 and/or 4.

Any combination of these five scenarios can be initiated by a team that is responsible for mapping an existing source into the CIM repository. There are many other details that need to be addressed as the content of any of the sources changes and/or when the core or common model changes.

Assuming numerous existing sources have been imported using all the import scenarios, now look at the export side. Ignoring the technique partition, the possible scenarios are:

1. *From Recast Partition:*
   Create a recast mapping for a sub-partition in the recast partition to a standard group (that is, inverse of import 2). The desired method would be to use the recast mapping to translate a standard group into a GDMO definition.
2. *From Recast Partition:*
   Create a Domain Mapping for one of the recast sub-partitions to a known management model that was not the original source for the content that overlaps.
3. *From Domain Partition:*
   Create a recast mapping for the complete content of the CIM to a selected technique (for MIF, this results in a non-standard group).
4. *From Domain Partition:*
   Create a Domain Mapping for the content of the CIM schema that overlaps with the content of an existing management model
5. *From Domain Partition:*
   Create a Domain Mapping for the entire content of the CIM schema to an existing management model with the necessary extensions.

## 7.3 Recording Mapping Decisions

In order to understand the role of the scratch pad (see Section 6.4 on page 62) in the repository, it is necessary to look at the import and export scenarios for the different partitions in the repository (technique, recast and application). These mappings can be organized into two categories:

- Homogeneous
- Heterogeneous

The homogeneous category includes the mapping where the imported syntax and expressions are the same as the exported (for example, software MIF in and software MIF out). The heterogeneous category addresses the mappings where the imported syntax and expressions are different from the exported (for example, MIF in and GDMO out). For the homogenous category, the information can be recorded by creating qualifiers during an import operation so the content can be exported properly. For the heterogeneous category, the qualifiers must be added after the content is loaded into a partition of the repository.

Figure 7-2 shows the X schema imported into the Y schema, and then being homogeneously exported into X or heterogeneously exported into Z. Each of the export arrows works with a different scratch pad.

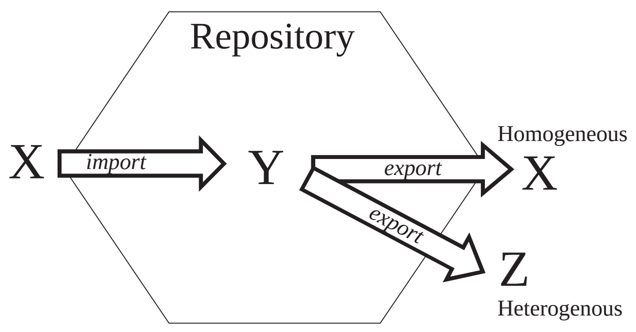


**Figure 7-2** Homogeneous and Heterogeneous Export

The definition of the heterogeneous category is actually based on knowing how a schema was loaded into the repository. A more general way of looking at this is to think of the export process using one of multiple scratch pads. One of the scratch pads was created when the schema was loaded and the others were added to handle mappings to schema techniques other than the import source (see Figure 7-3).

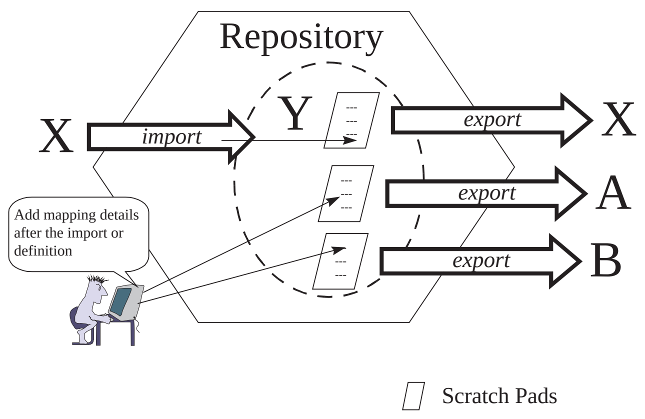


**Figure 7-3** Scratch Pads and Mapping

Figure 7-3 shows how the scratch pads of qualifiers are used without factoring in the unique aspects of each of the partitions (technique, recast, applications) within the CIM repository. The next step is to put this discussion in the context of these partitions.

For the technique partition, there is no need for a scratch pad since the CIM metamodel is used to describe the constructs used in the source metaschema. Therefore, by definition, the assumption is that there is one homogeneous mapping for each metaschema covered by the technique partition. These mappings create CIM object for the syntactical constructs of the schema and create associations for the ways they can be combined (for example, MIF groups include attributes).

For the recast partition, there are multiple scratch pads for each of the sub-partitions, since one is needed for each export targets (as shown in the previous figure) and there can be multiple mapping algorithms for each target. The latter occurs because part of creating recast mapping involves mapping the constructs of the source into CIM metamodel constructs. Therefore, for the MIF syntax, a mapping needs to be created for component, group, attribute, etc. into appropriate CIM metamodel constructs like object, association, property, etc. These mappings can be arbitrary. As a specific example, one of the decisions that needs to be made is whether a group maps into an object or a component maps into an object. It would be possible to have two different recast mapping algorithms, one which mapped groups into objects with qualifiers that preserved the component and one which mapped components into objects with qualifiers that preserved the group name for the properties. Therefore, the scratch pads in the recast partition are organized by target technique and employed algorithm.

For the domain partitions, there are two types of mappings. The first is similar to the recast partition in that some portion of the domain partition is mapped into the syntax of another metaschema. These mappings can use the same qualifier scratch pads and associated algorithms that are developed for the recast partition. The second type of mapping facilitates documenting the content overlap between the domain partition and some other model (for example, software groups). These mappings cannot be determined in a generic way at import time; therefore it is best to consider them in the context of exporting. The mapping uses filters to determine the overlaps and then performs the necessary conversions. The filtering can be done using qualifiers which indicate a particular set of domain partition constructs map into some combination of constructs in the target/source model. The conversions would be documented in the repository using a complex set of qualifiers which capture the notion of how to write or insert the overlapped content into the target model. The mapping qualifiers for the domain partition would be organized like the recasting partition for the syntax conversions and there be scratch pads for each of the models for documenting overlapping content.

In summary, pick the partition, develop a mapping, and identify the qualifiers necessary to capture potentially lost information when developing mapping details for a particular source. On the export side, the mapping algorithm checks to see if the content to be exported includes the necessary qualifiers for the logic to work.

# *MOF Syntax Grammar Description*

This Appendix contains the grammar for Managed Object Format (MOF) syntax. The grammar is described in the notation defined in RFC 2234 (see **Referenced Documents**), with this deviation:

- Each token may be separated by an arbitrary number of white space characters, except where stated otherwise (at least one tab, carriage return, line feed, form feed, or space).

However, while this notation is convenient for describing the MOF syntax clearly, it should be noted that the MOF syntax has been defined so as to be expressible in an LL(1)-parsable grammar. This has been done to allow low-footprint implementations of MOF compilers.

In addition, the following points should be noted:

- An empty property list is equivalent to "*".
- All keywords are case-insensitive.
- The IDENTIFIER type is used for names of classes, properties, qualifiers, methods and namespaces; the rules governing the naming of classes and properties are to be found in Section E.2 on page 85.
- A stringValue may contain quote (") characters, provided that each is immediately preceded by a backslash (\) character.

```
mofSpecification      =  *mofProduction

mofProduction         =  compilerDirective    |
                         classDeclaration     |
                         assocDeclaration     |
                         qualifierDeclaration |
                         instanceDeclaration

compilerDirective     =  PRAGMA pragmaName  "(" pragmaParameter ")"

pragmaName            =  IDENTIFIER

pragmaParameter       =  stringValue

classDeclaration      =  [ qualifierList ]
                         CLASS className [ alias ] [ superClass ]
                         "{" *classFeature "}" ";"

assocDeclaration      =  "[" ASSOCIATION *( "," qualifier ) "]"
                         CLASS className [ alias ] [ superClass ]
                         "{" *associationFeature "}" ";"

                         // Context:
                         // The remaining qualifier list must not include
                         // the ASSOCIATION qualifier again. If the
                         // association has no super association, then at
                         // least two references must be specified
                         // ASSOCIATION qualifier may be omitted in
```

```
                                  // sub associations.

className               =   schemaName "_" IDENTIFIER   // NO whitespace

                            // Context:
                            // Schema name must not include "_"

alias                   =   AS aliasIdentifer

aliasIdentifer          =   "$" IDENTIFIER   // NO whitespace

superClass              =   ":" className

classFeature            =   propertyDeclaration | methodDeclaration

associationFeature      =   classFeature | referenceDeclaration

qualifierList           =   "[" qualifier *( "," qualifier ) "]"

qualifier               =   qualifierName [ qualifierParameter ] [ ":" 1*flavor ]

qualifierParameter      =   "(" constantValue ")" | arrayInitializer

flavor                  =   ENABLEOVERRIDE | DISABLEOVERRIDE | RESTRICTED |
                            TOSUBCLASS | TRANSLATABLE

propertyDeclaration     =   [ qualifierList ] dataType propertyName
                            [ array ] [ defaultValue ] ";"

referenceDeclaration    =   [ qualifierList ] objectRef referenceName
                            [ defaultValue ] ";"

methodDeclaration       =   [ qualifierList ] dataType methodName
                            "(" [ parameterList ] ")" ";"

propertyName            =   IDENTIFIER

referenceName           =   IDENTIFIER

methodName              =   IDENTIFIER

dataType                =   DT_UINT8 | DT_SINT8 | DT_UINT16 | DT_SINT16 |
                            DT_UINT32 | DT_SINT32 | DT_UINT64 | DT_SINT64 |
                            DT_REAL32 | DT_REAL64 | DT_CHAR16 |
                            DT_STR | DT_BOOL | DT_DATETIME

objectRef               =   className REF

parameterList           =   parameter *( "," parameter )

parameter               =   [ qualifierList ] (dataType|objectRef) parameterName
                            [ array ]

                            // Context:
                            // The qualifier list for "parameter" is restricted
                            // to IN, OUT, or both.

parameterName           =   IDENTIFIER

array                   =   "[" [positiveDecimalValue] "]"
```

```
positiveDecimalValue  =  positiveDecimalDigit *decimalDigit

defaultValue          =  "=" initializer

initializer           =  simpleInitializer | arrayInitializer | objectPath

simpleInitializer     =  constantValue | aliasIdentifier

arrayInitializer      =  "{" simpleInitializer *( "," simpleInitializer )"}"

constantValue         =  integerValue | realValue | charValue | stringValue |
                         booleanValue | nullValue | objectPath

integerValue          =  binaryValue | octalValue | decimalValue | hexValue

qualifierDeclaration  =  QUALIFIER qualifierName qualifierType scope
                         [ defaultFlavor ] ";"

qualifierName         =  IDENTIFIER

qualifierType         =  ":" dataType [ array ] [ defaultValue ]

scope                 =  "," SCOPE "(" metaElement *( "," metaElement ) ")"

metaElement           =  SCHEMA | CLASS | ASSOCIATION | INDICATION | QUALIFIER
                         PROPERTY | REFERENCE | METHOD | PARAMETER | ANY

defaultFlavor         =  "," FLAVOR "(" flavor *( "," flavor ) ")"

instanceDeclaration   =  [ qualifierList ] INSTANCE OF className [ alias ]
                         "{" 1*propertyInit "}" ";"

propertyInit          =  [ qualifierList ] propertyName "=" initializer ";"

                         // Context:
                         // Note that associations may be instantiated also,
                         // and thus references may be initialized. The
                         // assigned value must be compatible with the
                         // feature type.  For a reference, the assigned
                         // value is either a string that holds an object
                         // key or an alias that indirectly holds it.
```

The following productions do not allow whitespaces between the terms:

```
namespacePath         =  """ [ host ] localNsPath """

host                  =  ( "\\" | "//" ) ( "." | hostName | address )

hostName              =  1*ucs2Character

                         // Context:
                         // An unquoted string that specifies a host name
                         // that can be resolved to a network address.
                         // May not contain forward slash, backslash, or
                         // colon.


address               =  1*ucs2Character
```

```
                                  // Context:
                                  // An unquoted string that specifies a network
                                  // address in a standard format (for example,
                                  // an IPaddress in dotted decimal notation).

localNsPath             =         1*( ( "/" | "\" ) namespaceName )

namespaceName           =         IDENTIFIER

schemaName              =         IDENTIFIER
                                  // Context:
                                  // Schema name must not include "_"

objectPath              =         stringValue

                                  // Context:
                                  // A special string that serves as reference to
                                  // an object instance. It consists of an optional
                                  // namespace path followed by the object keys,
                                  // including the scoping hierarchy. See related
                                  // sections on object keys and naming for more
                                  // information.

fileName                =         stringValue

binaryValue             =         [ "+" | "-" ] 1*binaryDigit ( "b" | "B" )

binaryDigit             =         "0" | "1"

octalValue              =         [ "+" | "-" ] "0" 1*octalDigit

octalDigit              =         "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"

decimalValue            =         [ "+" | "-" ] ( positiveDecimalDigit *decimalDigit | "0" )

decimalDigit            =         "0" | positiveDecimalDigit

positiveDecimalDigit    =         "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

hexValue                =         [ "+" | "-" ] ( "0x" | "0X" ) 1*hexDigit

hexDigit                =         decimalDigit | "a" | "A" | "b" | "B" | "c" | "C" |
                                  "d" | "D" | "e" | "E" | "f" | "F"

realValue               =         [ "+" | "-" ] *decimalDigit "." 1*decimalDigit
                                  [ ( "e" | "E" ) [ "+" | "-" ] 1*decimalDigit ]

charValue               =         // any single-quoted Unicode-character, except
// single quotes

stringValue             =         1*( """ *ucs2Character """ )

ucs2Character           =         // any valid UCS-2-character

booleanValue            =         TRUE | FALSE

nullValue               =         NULL
```

The remaining productions are case-insensitive keywords.

```
ANY              =   "any"
AS               =   "as"
ASSOCIATION      =   "association"
CLASS            =   "class"
DISABLEOVERRIDE  =   "disableOverride"
DT_BOOL          =   "boolean"
DT_CHAR16        =   "char16"
DT_DATETIME      =   "datetime"
DT_REAL32        =   "real32"
DT_REAL64        =   "real64"
DT_SINT16        =   "sint16"
DT_SINT32        =   "sint32"
DT_SINT64        =   "sint64"
DT_SINT8         =   "sint8"
DT_STR           =   "string"
DT_UINT16        =   "uint16"
DT_UINT32        =   "uint32"
DT_UINT64        =   "uint64"
DT_UINT8         =   "uint8"
ENABLEOVERRIDE   =   "enableoverride"
FALSE            =   "false"
FLAVOR           =   "flavor"
INDICATION       =   "indication"
INSTANCE         =   "instance"
METHOD           =   "method"
NULL             =   "null"
OF               =   "of"
PARAMETER        =   "parameter"
PRAGMA           =   "#pragma"
PROPERTY         =   "property"
QUALIFIER        =   "qualifier"
REF              =   "ref"
REFERENCE        =   "reference"
RESTRICTED       =   "restricted"
SCHEMA           =   "schema"
SCOPE            =   "scope"
TOSUBCLASS       =   "tosubclass"
TRANSLATABLE     =   "translatable"
TRUE             =   "true"
```

Appendix B

# CIM Metaschema

```
// version 2.0

Qualifier    Abstract : boolean = false, Scope(class, Association, Indication),
                 Flavor(DisableOverride, Restricted);

Qualifier    Aggregate : boolean = false, Scope(reference),
                 Flavor (DisableOverride);

Qualifier    Aggregation : boolean = false, Scope(association)
                 Flavor (DisableOverride);

Qualifier    Alias : string = null, Scope(property, reference, method),
                 Flavor(Translatable);

Qualifier    ArrayType : string = "Bag", Scope(property);

Qualifier    Association : boolean = false, Scope(class, association),
                 Flavor(DisableOverride);

Qualifier    Delete : boolean = false, Scope(association, reference);

Qualifier    Description : string = null, Scope(any), Flavor(Translatable);

Qualifier    Expensive : boolean = false,
                 Scope(property, reference, method, class, association);

Qualifier    Ifdeleted : boolean = false, Scope(association, reference);

Qualifier    In : boolean = true, Scope(parameter);

Qualifier    Indication : boolean = false, Scope(class, indication),
                 Flavor(DisableOverride);

Qualifier    Invisible : boolean = false,
                 Scope(reference, association, class, property, method);

Qualifier    Key : boolean = false, Scope(property, reference),
                 Flavor(DisableOverride);

Qualifier    Large : boolean = false, Scope(property, class);

Qualifier    MappingStrings : string[],
                 Scope(class, property, association, indication, reference);

Qualifier    Max : uint32 = null, Scope(reference);

Qualifier    MaxLen : uint32 = null, Scope(property);

Qualifier    Min : uint32 = 0, Scope(reference);

Qualifier    ModelCorrespondence : string[], Scope(property);

Qualifier    Nonlocal : string = null, Scope(reference);
Qualifier    Out : boolean = false, Scope(parameter);

Qualifier    Override : string = null, Scope(property, method, reference);

Qualifier    Propagated : string = null, Scope(property),
                 Flavor(DisableOverride);

Qualifier    Read : boolean = true, Scope(property);

Qualifier    Required : boolean = false, Scope(property);

Qualifier    Revision : string = null, Scope(schema, class, association,
                 indication ), Flavor(Translatable);

Qualifier    Schema : string = null, Scope (property, method),
                 Flavor(DisableOverride, Translatable);

Qualifier    Source : string = null, Scope(class, association, indication);

Qualifier    Syntax : string = null, Scope(property, reference);

Qualifier    SyntaxType : string = null, Scope(property, reference);

Qualifier    TriggerType : string = null,
                 Scope(class, property, reference, method, association,
                 indication);

Qualifier    Units : string = null, Scope(property), Flavor(Translatable);

Qualifier    ValueMap : string[], Scope(property);

Qualifier    Values : string[], Scope(property), Flavor(Translatable);

Qualifier    Version : string = null, Scope(schema, class, association,
                 indication), Flavor(Translatable);

Qualifier    Weak : boolean = false, Scope(reference),
                 Flavor (DisableOverride, ToSubclass);

Qualifier    Write : boolean = true, Scope(property);



class Meta_NamedElement
{
    string Name;
};

class Meta_QualifierFlavor: Meta_NamedElement
{
    string Type;
};

class Meta_Schema: Meta_NamedElement
{
};

class Meta_Trigger: Meta_NamedElement
```

```
{
};

class Meta_Qualifier: Meta_NamedElement
{
    string Value;
};

class Meta_Method: Meta_NamedElement
{
};

class Meta_Property: Meta_NamedElement
{
};

class Meta_Class: Meta_NamedElement
{
};

class Meta_Indication: Meta_Class
{
};

class Meta_Association: Meta_Class
{
};

class Meta_Reference: Meta_Property
{
};

        [Association]
class Meta_Characteristics
{
    Meta_Qualifier REF Characteristic;
    Meta_NamedElement REF Characterized;
};

        [Association]
class Meta_PropertyDomain
{
    Meta_Property REF Property;
    Meta_Class REF Domain;
};

        [Association]
class Meta_MethodDomain
{
    Meta_Method REF Method;
    Meta_Class REF Domain;
};

        [Association]
class Meta_ReferenceRange
{
    Meta_Reference REF Reference;
    Meta_Class REF Range;
};
```

```
        [Association]
class Meta_QualifiersFlavor
{
    Meta_QualifierFlavor  REF Flavor;
    Meta_Qualifier  REF Qualifier;
};

        [Association]
class Meta_SubTypeSuperType
{
    Meta_Class  REF SuperClass;
    Meta_Class  REF SubClass;
};

        [Association]
class Meta_PropertyOverride
{
    Meta_Property  REF OverridingProperty;
    Meta_Property  REF OverriddenProperty;
};

        [Association]
class Meta_MethodOverride
{
    Meta_Method  REF OverridingMethod;
    Meta_Method  REF OverriddenMethod;
};

        [Association]
class Meta_ElementSchema
{
    Meta_NamedElement  REF Element;
    Meta_Schema  REF Schema;
};
```

Appendix C

# Values for UNITS Qualifier

The UNITS qualifier specifies the unit of measure in which the associated property is expressed. For example, a *Size* property might have Units ("bytes").

Currently recognized values are:

- Bits, KiloBits, MegaBits, GigaBits
- Bits per Second
- Bytes, KiloBytes, MegaBytes, GigaBytes Words, DoubleWords, QuadWords
- Degrees C, Tenths of Degrees C, Hundredths of Degrees C Degrees F, Tenths of Degrees F, Hundredths of Degrees F Degrees K, Tenths of Degrees K, Hundredths of Degrees K Color Temp Degrees K
- Volts, MilliVolts, Tenths of MilliVolts Amps, MilliAmps, Tenths of MilliAmps Watts, MilliWattHours
- Joules, Coulombs, Newtons
- Lumen, Lux, Candelas
- Pounds, Pounds per Square Inch
- Cycles, Revolutions, Revolutions per Minute
- Minutes, Seconds, Tenths of Seconds, Hundredths of Seconds, MicroSeconds, MilliSeconds, NanoSeconds
- Hours, Days, Weeks
- Hertz, MegaHertz
- Pixels, Pixels per Inch
- Counts per Inch
- Percent, Tenths of Percent, Hundredths of Percent
- Meters, Centimeters, Millimeters, Cubic Meters, Cubic Centimeters, Cubic Millimeters
- Inches, Feet, Cubic Inches, Cubic Feet Ounces, Liters, Fluid Ounces
- Radians, Steradians, Degrees
- Gravities, Pounds, Foot-Pounds
- Gauss, Gilberts, Henrys, MilliHenrys, Farads, MilliFarads, MicroFarads, PicoFarads
- Ohms, Siemens
- Moles, Becquerels, Parts per Million Decibels
- Grays, Sieverts

# *Unified Modeling Language (UML) Notation*

The CIM metaschema notation is based directly on the notation used in Unified Modeling Language (UML). There are distinct symbols for all of the major constructs in the schema, with the exception of qualifiers (as opposed to properties that are directly represented in the diagrams).

In UML, a class is represented by a rectangle. The class name either stands alone in the rectangle or is in the uppermost segment of the rectangle. If present, the segment below that containing the name contains the properties of the class. If present, a third region indicates the presence of methods.

A line decorated with a triangle indicates an inheritance relationship, in which the lower rectangle represents a subtype of the upper rectangle. The triangle points to the superclass.

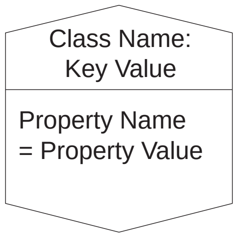
Other solid lines represent relationships. The cardinality of the references on either side of the relationship is indicated by a decoration on either end. The following character combinations are commonly used:

| | |
|---|---|
| 1 | Indicates a single valued, required reference. |
| 0 . . . 1 | Indicates an optional single valued reference. |
| * | Indicates an optional many valued reference, as does 0 . . . *. |
| 1 . . . * | Indicates a required many valued reference. |

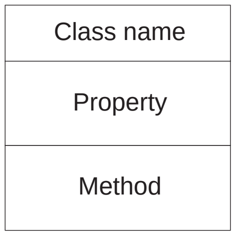
A line connected to a rectangle by a dotted line represents a subclass relationship between two associations.

For each metaelement, the diagram notation and/or its interpretation are summarized below.

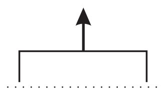
- **Object**



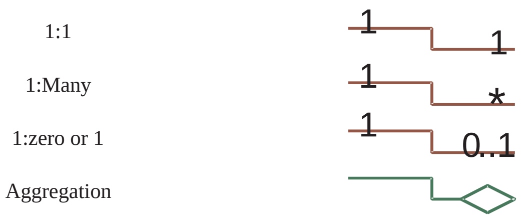
- **Primitive type**
  Text to the right of the colon in the center portion of the class icon
- **Class**



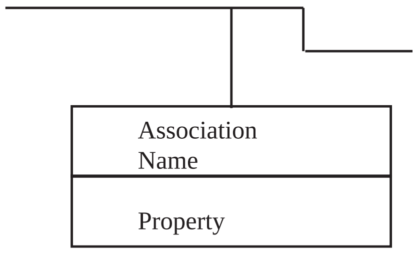
- **Subclass**



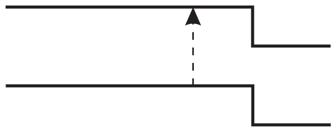
- **Association**



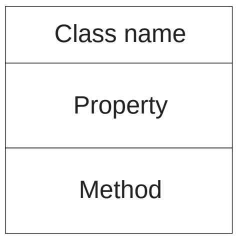
- **Association with properties**
  link class with the link class having the same name as the association and using normal conventions for representing properties and methods



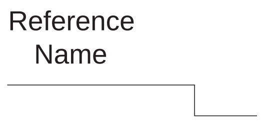
- **Association with subclass**
  A dashed line running from the sub association to the super class



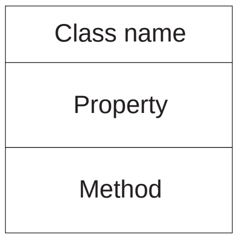
- **Property**
  Middle section of the class icon is a list of the properties of the class.



- **Reference**
  One end of the association line labeled with the name of the reference



- **Method**
  Lower section of the class icon is a list of the methods of the class.



- **Overriding**
  No direct equivalent.

  Note: Use of the same name does not imply overriding.
- **Indication**
  Message trace diagram in which vertical bars represent objects and horizontal lines represent messages
- **Trigger**
  State transition diagrams
- **Qualifier**
  No direct equivalent

Appendix E

# UNICODE Usage

## E.1 Basic Character Set

All punctuators associated with object path or MOF Syntax occur within the Basic Latin range U+0000 to U+007F. These include normal punctuators, such as slashes, colons, commas, and the like. No important syntactic punctuation character occurs outside of this range.

All characters above U+007F are treated as parts of names, even though there are several reserved characters such as U+2028 and U+2029 which are logically whitespace.

Therefore, all namespace, class and property names are identifiers composed as follows:

1. Initial identifier characters must be in set $S_1$, where
   $S_1$ = {U+005F, U+0041...U+005A, U+0061...U+007A, U+0080...U+FFEF)
   This is alphabetic, plus underscore.
2. All following characters must be in set $S_2$, where
   $S_2 = S_1$ È {U+0030 . . .U+0039}
   This is alphabetic, underscore, plus Arabic numerals 0 through 9.

Note that the Unicode specials range (U+FFF0...U+FFFF) are not legal for identifiers.

While the above sub-range of U+0080 . . . U+FFEF includes many diacritical characters which would not be useful in an identifier, as well as the Unicode reserved sub-range which has not been allocated, it seems advisable for simplicity of parsers to simply treat this entire sub-range as "legal" for identifiers.

Refer to RFC2279 (see **Referenced**Documents**)** as an example of a Universal Transformation Format that has specific characteristics for dealing with multi-octet characters on an application-specific basis.

## E.2 MOF Text

MOF files using UNICODE should contain a signature as the first two bytes of the text file, either U+FFFE or U+FEFF, depending on the byte ordering of the text file (as suggested in section 2.4 of the UNICODE specification ISO / IEC 639: 1988 — see **Referenced Documents**).

U+FFFE is little endian.

All MOF keywords and punctuation symbols are as described in the MOF Syntax document and are not locale-specific. All such characters are composed of characters falling in the range U+0000...U+007F, regardless of the locale of origin for the MOF or its identifiers.

## E.3 Quoted Strings

In all cases where string values are needed which are not identifiers, delimiters must surround them.

The supported delimiters are U+0027 or U+0022. Once a quoted string is started using one of these delimiters, the same delimiter is used to terminate it.

In addition, the digraph U+005C ("\") followed by U+0027 "'" constitutes an embedded quotation mark, not a termination of the quoted string.

The characters permitted within the quotation mark delimiters just described may fall within the range U+0001 through U+FFEF.

*Appendix F*

# *Guidelines for CIM Usage*

## F.1 General

- Method descriptions are recommended and must, at a minimum, indicate that method's side-effects (pre- and post-conditions).
- Associations must not be declared as subtypes of classes that are not associations.
- Although the Override qualifier allows the overridden property, method or reference to be renamed, this is discouraged because it introduces additional names into the schema and makes instances harder to interpret.
- As a general rule, it is recommended that class names should not be reused as part of property or method names. Property and method names are already unique within their defining class.
- To enable information sharing between different CIM implementations, the MAXLEN qualifier should be used to specify the maximum length of string properties. This qualifier must always be present for string properties used as keys.
- A class that has no ABSTRACT qualifier must define, or inherit, key properties.
- Leading underscores should not be used in identifiers, and must not be used in the standard schemas.

## F.2 Mapping of Octet Strings

Most management models, including SNMP and DMI, support octet strings as data types. The octet string data type represents arbitrary numeric or textual data. This data is stored as an indexed byte array of unlimited, but fixed size. Typically, the first N bytes indicate the actual string length. Since some environments only reserve the first byte, they do not support octet strings larger than 255 bytes.

In CIM Version 2.0, CIM does not support octet strings as a separate data type. To map octet strings, it is recommended that the equivalent CIM property be defined as an array of unsigned 8-bit integers (uint8). The first four bytes of the array contain the length of the octet data: byte 0 is the most significant byte of the length; byte 3 is the least significant byte of the length. The octet data starts at byte 4.

## F.3 SQL Reserved Words

It is recommended that SQL reserved words be avoided in the selection of class and property names. This particularly applies to property names, since class names are prefixed by the schema name, making a clash with a reserved word unlikely.

The current set of SQL reserved words are listed below:

**From sql1992.txt:**

| | | | |
|---|---|---|---|
| AFTER | ALIAS | ASYNC | BEFORE |
| BOOLEAN | BREADTH | COMPLETION | CALL |
| CYCLE | DATA | DEPTH | DICTIONARY |
| EACH | ELSEIF | EQUALS | GENERAL |
| IF | IGNORE | LEAVE | LESS |
| LIMIT | LOOP | MODIFY | NEW |
| NONE | OBJECT | OFF | OID |
| OLD | OPERATION | OPERATORS | OTHERS |
| PARAMETERS | PENDANT | PREORDER | PRIVATE |
| PROTECTED | RECURSIVE | REF | REFERENCING |
| REPLACE | RESIGNAL | RETURN | RETURNS |
| ROLE | ROUTINE | ROW | SAVEPOINT |
| SEARCH | SENSITIVE | SEQUENCE | SIGNAL |
| SIMILAR | SQLEXCEPTION | SQLWARNING | STRUCTURE |
| TEST | THERE | TRIGGER | TYPE |
| UNDER | VARIABLE | VIRTUAL | VISIBLE |
| WAIT | WHILE | WITHOUT | |

**From sql1992.txt (Annex E):**

| | | | |
|---|---|---|---|
| ABSOLUTE | ACTION | ADD | ALLOCATE |
| ALTER | ARE | ASSERTION | AT |
| BETWEEN | BIT | BIT_LENGTH | BOTH |
| CASCADE | CASCADED | CASE | CAST |
| CATALOG | CHAR_LENGTH | CHARACTER_LENGTH | COALESCE |
| COLLATE | COLLATION | COLUMN | CONNECT |
| CONNECTION | CONSTRAINT | CONSTRAINTS | CONVERT |
| CORRESPONDING | CROSS | CURRENT_DATE | CURRENT_TIME |
| CURRENT_TIMESTAMP | CURRENT_USER | DATE | DAY |
| DEALLOCATE | DEFERRABLE | DEFERRED | DESCRIBE |
| DESCRIPTOR | DIAGNOSTICS | DISCONNECT | DOMAIN |
| DROP | ELSE | END-EXEC | EXCEPT |
| EXCEPTION | EXECUTE | EXTERNAL | EXTRACT |
| FALSE | FIRST | FULL | GET |
| GLOBAL | HOUR | IDENTITY | IMMEDIATE |
| INITIALLY | INNER | INPUT | INSENSITIVE |
| INTERSECT | INTERVAL | ISOLATION | JOIN |
| LAST | LEADING | LEFT | LEVEL |
| LOCAL | LOWER | MATCH | MINUTE |
| MONTH | NAMES | NATIONAL | NATURAL |
| NCHAR | NEXT | NO | NULLIF |
| OCTET_LENGTH | ONLY | OUTER | OUTPUT |
| OVERLAPS | PAD | PARTIAL | POSITION |

| | | | |
|---|---|---|---|
| PREPARE | PRESERVE | PRIOR | READ |
| RELATIVE | RESTRICT | REVOKE | RIGHT |
| ROWS | SCROLL | SECOND | SESSION |
| SESSION_USER | SIZE | SPACE | SQLSTATE |
| SUBSTRING | SYSTEM_USER | TEMPORARY | THEN |
| TIME | TIMESTAMP | TIMEZONE_HOUR | TIMEZONE_MINUTE |
| TRAILING | TRANSACTION | TRANSLATE | TRANSLATION |
| TRIM | TRUE | UNKNOWN | UPPER |
| USAGE | USING | VALUE | VARCHAR |
| VARYING | WHEN | WRITE | YEAR |
| ZONE | | | |

**From sql3part2.txt (Annex E):**

| | | | |
|---|---|---|---|
| ACTION | ACTOR | AFTER | ALIAS |
| ASYNC | ATTRIBUTES | BEFORE | BOOLEAN |
| BREADTH | COMPLETION | CURRENT_PATH | CYCLE |
| DATA | DEPTH | DESTROY | DICTIONARY |
| EACH | ELEMENT | ELSEIF | EQUALS |
| FACTOR | GENERAL | HOLD | IGNORE |
| INSTEAD | LESS | LIMIT | LIST |
| MODIFY | NEW | NEW_TABLE | NO |
| NONE | OFF | OID | OLD |
| OLD_TABLE | OPERATION | OPERATOR | OPERATORS |
| PARAMETERS | PATH | PENDANT | POSTFIX |
| PREFIX | PREORDER | PRIVATE | PROTECTED |
| RECURSIVE | REFERENCING | REPLACE | ROLE |
| ROUTINE | ROW | SAVEPOINT | SEARCH |
| SENSITIVE | SEQUENCE | SESSION | SIMILAR |
| SPACE | SQLEXCEPTION | SQLWARNING | START |
| STATE | STRUCTURE | SYMBOL | TERM |
| TEST | THERE | TRIGGER | TYPE |
| UNDER | VARIABLE | VIRTUAL | VISIBLE |
| WAIT | WITHOUT | | |

**From sql3part4.txt (Annex E):**

| | | | |
|---|---|---|---|
| CALL | DO | ELSEIF | EXCEPTION |
| IF | LEAVE | LOOP | OTHERS |
| RESIGNAL | RETURN | RETURNS | SIGNAL |
| TUPLE | WHILE | | |

# Glossary

**Aggregation**
A strong form of an association. The relationship between a system and the components that make up the system can be called an aggregation, for example. An aggregation is expressed as a Qualifier on the association class. Aggregation often implies, but does not require, that the aggregated objects have mutual dependencies.

**Association**
A class that expresses the relationship between two other classes. The relationship is established by the presence of two or more references in the association class pointing to the related classes.

**Cardinality**
A relationship between two classes that allows more than one object to be related to a single object. For example, Microsoft Office is made up of the software elements Word, Excel, Access, and PowerPoint.

**CIM**
Common Information Model is the schema of the overall managed environment. It is divided into a Core schema, Common schemas and extended schemas.

**CIM schema**
The schema representing the Core and Common models. Versions of this schema will become available as the schema evolves.

**Class**
A collection of instances, all of which support a common a type, that is, 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.

**Common model**
A collection of models specific to a particular area, derived from the Core model. Included are the system model, the application model, the network model and the device model.

**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.

**DMTF**
Desktop Management Task Force

**Domain**
A virtual room for object names that establishes the range in which the names of objects are unique.

**Explicit Qualifier**
A qualifier defined separately from the definition of a class, property or other schema element (see implicit qualifier). Explicit qualifier names must be unique across the entire schema. Implicit qualifier names must be unique within the defining schema element that is a given schema element may not have two qualifiers with the same name.

**Extended schema**
A platform specific schema derived from the Common schema. An example is the Win32 schema.

**Feature**
A property or method belonging to a class.

**Flavor**
Part of a qualifier spcification indicating overriding and inheritance rules. For example the qualifier KEY has Flavor(NoOverrideToSubclass) meaning that every subclass must inherit it and cannot override it.

**GDMO**
Guidelines for the Definition of Managed Objects, ISO / IEC 10165 Part 4, 1992; equivalent to ITU X.722.

**Implicit Qualifier**
A qualifier defined as a part of the definition of a class, property or other schema element (see explicit qualifier).

**Indication**
A type of class usually created as a by-product of the occurrence of a trigger.

**Inheritance**
A relationship between two classes in which all the members of the subclass are required to be members of the superclass. Any member of the subclass must also support any method or property supported by the superclass. For example, Modem is a subclass of Device.

**Instance**
A unit of data. An instance is a set of property values that can be uniquely identified by a key.

**Key**
A value used to identify an object within the scope of a namespace. For example, a drive letter in the scope of a system. A property that is a key will have the Qualifier KEY set to ‘‘true’’.

**Managed object**
The actual item in the system environment that is accessed by the provider. For example, a Network Interface Card.

**Metamodel**
A set of classes, associations and properties that expresses the types of things that can be defined in a Schema. For example, the metamodel includes a class called property which defines the properties known to the system, a class called method which defines the methods known to the system, and a class called class which defines the classes known to the system.

**Metaschema**
The schema of the metamodel.

**Method**
A Method is a declaration of a signature, that is, the method name, return type and parameters, and in the case of a concrete class may imply an implementation.

**MIF**
Management Information File

**Model**
A set of classes, properties and associations that allow the expression of information about some specific domain. For example, a Network may consist of Network Devices and Logical Networks. The Network Devices may have attachment associations to each other, and may have member associations to Logical Networks.

**Model path**
A reference to an object within a namespace.

**MOF**
Managed Object Format

**Name**
Combination of a Namespace path and a Model path that identifies a unique object.

**Namespace**
An object that defines a scope within which object keys must be unique.

**Namespath path**
A reference to a namespace within an implementation that is capable of hosting CIM objects.

**Polymorphism**
A subclass may redefine the implementation of a method or property inherited from its superclass. The property or method is thereby redefined even if the superclass is used to access the object. For example, Device may define status as a string, and may return the values ‘‘connected’’, ‘‘on’’ or ‘‘off’’. The Modem subclass of Device may redefine (override) status by returning ‘‘on’’, ‘‘off’’, but not connected. If all Devices are enumerated, any Device that happens to be a modem will not return the value ‘‘connected’’ for the status property.

**Property**
A value used to characterize an instance of a class. For example, a Device may have a property called status.

**Provider**
An executable that can return or set information about a given managed object.

**Qualifier**
A value used to characterize a method, property, or class in the metaschema. For example, if a property has the qualifier KEY with the value ‘‘true’’, the property is a key for the class.

**Reference**
References are special property types that are references or ‘‘pointers’’ to other instances.

**Schema**
A namespace and unit of ownership for a set of classes. Schemas may come in forms such as a text file, information in a repository, or diagrams in a CASE tool.

**Scope**
Part of a Qualifier specification indicating which metaconstructs the Qualifier can be used with. For Example the Qualifier ABSTRACT has Scope(Class Association Indication) meaning that it can only be used with Classes, Associations, and Indications.

**Scoping object**
Objects which represent a real-world managed element, which in turn propagate keys to other objects.

**Signature**
The return type and parameters supported by a method.

**SMI**
Structure of Management Information, IETF RFC 1155

**SNMP**
Simple Network Management Protocol, IETF RFC 1157

**SQL**
Structured Query Language. The International Standard for the Database Language SQL is ISO / IEC 9075: 1992.

**Subclass**
See Inheritance.

**Superclass**
See Inheritance.

**Top level object**
A class or object that has no scoping object.

**Trigger**
A trigger is the occurrence of some action such as the creation, modification or deletion of an object, access to an object or modification or access to a property. Triggers may also be fired as a result of the passage of a specified period of time. A trigger typically results in an Indication.

**UML**
Unified Modeling Language

# Index