Distributed Management Task Force, Inc.

COMMON INFORMATION MODEL (CIM) INFRASTRUCTURE SPECIFICATION

DSP0004
Version 2.3 Final
October 4, 2005

==

Terminology

The key phrases and words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY and OPTIONAL in this document are to be interpreted as described in the IETF's RFC 2119. The complete reference for this document is: "Key words for Use in RFCs to Indicate Requirement Levels", IETF RFC 2119, March 1997 (http://www.ietf.org/rfc/rfc2119.txt).

==

Abstract

The DMTF Common Information Model (CIM) Infrastructure is an approach to the management of systems and networks that applies the basic structuring and conceptualization techniques of the object-oriented paradigm. The 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.

A management schema is provided to establish a common conceptual framework at the level of a fundamental typology both with respect to classification and association, and with respect to a basic set of classes intended to establish a common framework for a description of the managed environment. The management schema is divided into these conceptual 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, databases, networks and devices. The information model is specific enough to provide a basis for the development of management applications. This model provides a set of base classes for extension into the area of technology-specific schemas. The Core and Common models together are expressed 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[†]).

[†] Other product and corporate names may be trademarks of other companies and are used only for explanation and to the owners' benefit, without intent to infringe.

Contents

Table of Figures

1

1 Introduction and Overview

This section describes the many ways in which the Common Information Model (CIM) can be used. It provides a context in which the details described in the later chapters can be understood.

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, the DMTF refers to this information model as CIM, the Common Information Model.

This document describes an object-oriented meta model 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 – those topics will be addressed in a future version of the specification. For information on the current core and common schemas developed using this meta model, contact the Distributed Management Task Force.

Throughout this document, elements of formal syntax are described in the notation defined in [7], with these deviations:

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).

The vertical bar ("|") character is used to express alternation, rather than the virgule ("/") specified in [7].

1.1 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 these 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).

1.1.1 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.

1.1.2 Common Model

The Common model is a basic set of classes that define various technology-independent areas. The areas are systems, applications, networks and 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 the Common model is extended, it will offer a broader range of information.

1.1.3 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.

1.2 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. It is possible to use these ways in combination within a management application.

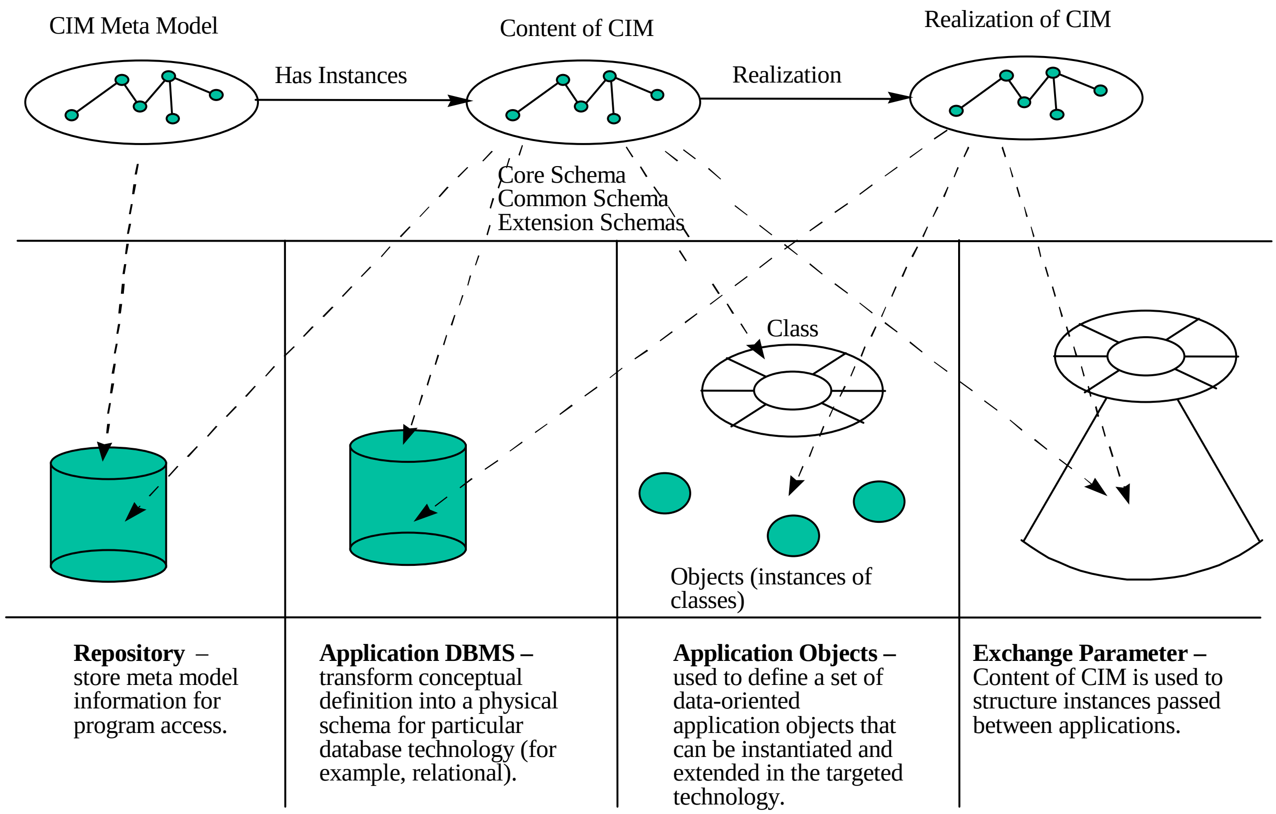


Figure 1-1 Four Ways to Use CIM

As a repository (see the Repository Perspective section for more detail), 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 meta model used by CIM is stored in a repository that becomes a representation of the meta model. This is accomplished by mapping the meta-model constructs into the physical schema of the targeted repository, then populating the repository with the classes and properties expressed in the Core model, Common model and Extension schemas.

For an application 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 APIs which can create the appropriate object in the local implementation technology.

1.2.1 CIM Implementation 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,[1] no programming interfaces or protocols are defined by (and hence cannot be considered as) 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 an implementation detail 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.

[1] The standard CIM application programming interface and/or communication protocol will be defined in a future version of the CIM Infrastructure specification.

2 Meta Schema

The Meta Schema is a formal definition of the model. It defines the terms used to express the model and its usage and semantics (see Appendix B).

The Unified Modeling Language (UML) is used to define the structure of the meta schema. In the discussion that follows, italicized words refer to objects in the figure. The reader is expected to be familiar with UML notation (see http://www.rational.com/uml) 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 Meta Schema

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 their owning schemas.

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).

Each instance provides values for the Properties associated with the instance's defining Class. An Instance does not carry values for any other Properties or Methods that aren't defined in (or inherited by) its defining Class. An Instance may not redefine the Properties or Methods defined in (or inherited by) its defining Class.

Instances are not Named Elements, and cannot have Qualifiers associated with them. (Qualifiers MAY be associated with the Instance's Class, however, as well as the Properties and Methods defined in, or inherited by, that Class.) Instances are also not permitted to attach new Qualifiers to Properties, Methods, or Parameters, as the association between Qualifier and Named Element is not restricted to the context of a particular Instance.

A *Property* assigns values 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,[2] return state and set state, respectively.

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.

A *Trigger* is 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 result 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. An Association cannot be a subclass of a non-association Class. Any subclass of an Association is an Association.

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.

[2] Note the equivocation between "object" as instance and "object" as class. This is common usage in object-oriented literature and reflects the fact that in many cases, operations and concepts may apply to or involve both classes and instances.

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 characteristics of a Property or the key of a Class). Qualifiers provide a mechanism that makes the meta schema extensible in a limited and controlled fashion. It is possible to add new types of Qualifiers by the introduction of a new Qualifier name, thereby providing new types of meta data to processes that manage and manipulate classes, properties and other elements of the meta schema. See below for details on the qualifiers provided.

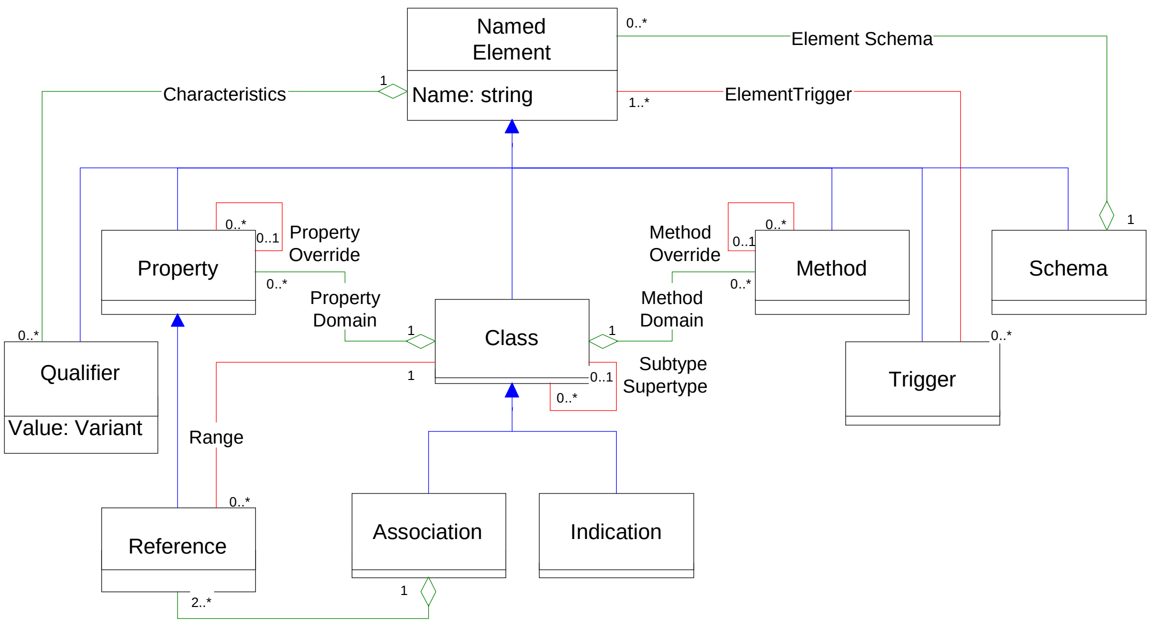


Figure 2-1 Meta Schema Structure

Figure 2-1 provides an overview of the structure of the meta schema. The complete meta schema is defined by the MOF found in Appendix B. The rules defining the meta schema are:

1. Every meta construct is expressed as a descendent of a Named Element.
2. A Named Element has 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 only one schema.
5. A Qualifier Type (not 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 Name, a Type (intrinsic data type), a Value of this type, a Scope, a Flavor and a default Value. The type of the Qualifier Value must agree with the type of the Qualifier Type.
7. A Property is a Named Element and has only exactly one Domain: the Class that owns the Property. The Property is applicable to Instances of the Domain (including Instances of subclasses of the Domain), and not to any other Instances.

8. A Property can have an Override relationship with another Property from a different class. The Domain of the overridden Property must be a supertype of the Domain of the overriding Property. For non-Reference Properties, the type associated with the overriding Property MUST agree with (be the same as) the type of the overridden Property.

9. The Class referenced by the Range association (Figure 2-4) 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 cannot 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. The Method is applicable to Instances of the Domain (including Instances of subclasses of the Domain), and not to any other Instances.

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 changes that invoke a trigger are 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 meta-schema object is a descendent of a 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 (that is, the Class name prefixed by the schema name) are unique within the schema. (See the discussion of schemas later in this section).

    B Fully-qualified Association and Indication Names are unique within the schema (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 Namespace. An implicitly-defined Qualifier must agree in type, scope and flavor with any explicitly-defined 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. Note 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.

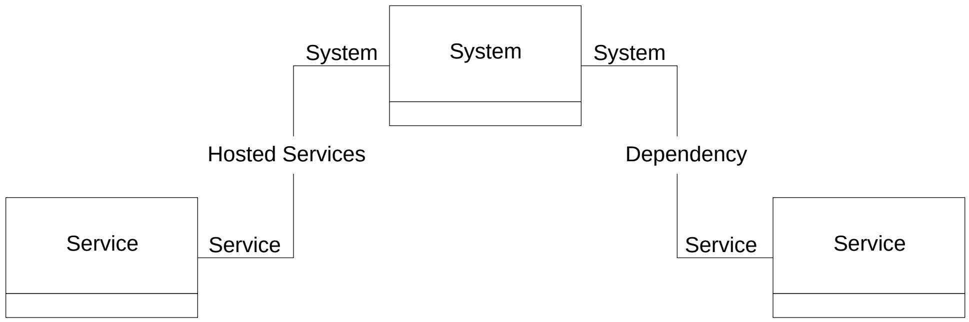


Figure 2-2 Reference Naming

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.*

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 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, they can be the Domain for Methods, Properties and References (that is, Associations and Indications can have Properties and Methods in the same way as a Class does). 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 because of that occurrence—usually a Trigger. Indications are not required to have keys. Typically, Indications are very short-lived 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.

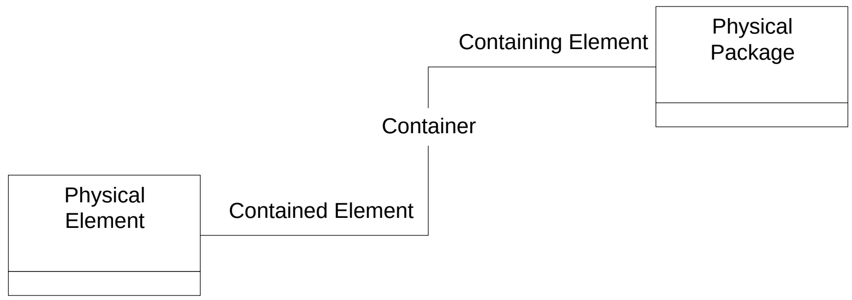


Figure 2-3 References, Ranges, and Domains

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).

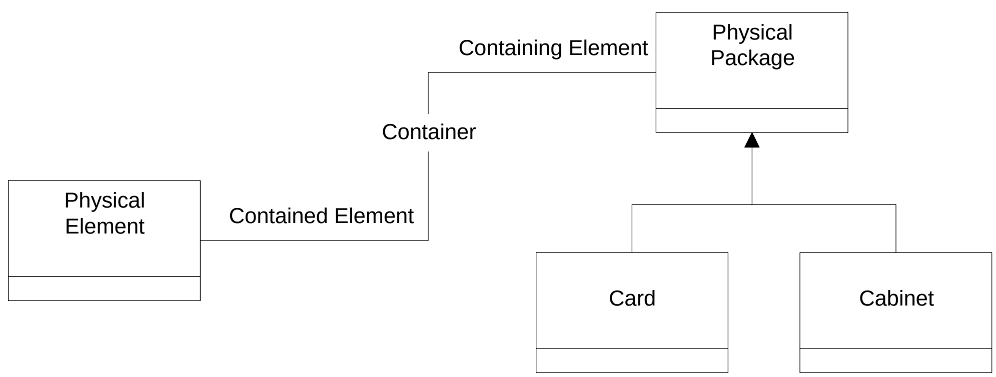


Figure 2-4 References, Ranges, Domains and Inheritance

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).

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. There are no subtype relationships among the intrinsic data types uint8, sint8, uint16, sint16, uint32, sint32, uint64, sint64, string, boolean, real32, real64, datetime, and char16.

This table contains 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

2.2.1 Datetime Type

The datetime type is used to specify a timestamp (point in time) or an interval. If it specifies a timestamp, it allows to preserve the timezone offset. In both cases, datetime allows to specify varying precision of the date and time information.

Datetime uses a fixed string-based format. The format for timestamps is: yyyymmddhhmmss.mmmmmmsutc

where the meaning of each field is:

- yyyy is a 4 digit year
- mm is the month within the year (starting with 01)
- dd is the day within the month (starting with 01)
- hh is the hour within the day (24-hour clock, starting with 00)
- mm is the minute within the hour (starting with 00)
- ss is the second within the minute (starting with 00)
- mmmmmm is the microsecond within the second (starting with 000000)
- s is a "+" or "-", indicating that the value is a timestamp, and indicating the sign of the UTC (Universal Coordinated Time; for all intents and purposes the same as Greenwich Mean Time) correction field. A "+" is used for time zones east of Greenwich, and a "-" is used for time zones west of Greenwich.
- utc is the offset from UTC in minutes (using the sign indicated by s).

Since datetime contains the time zone information, it is possible to reconstruct the original time zone from the value. However, this also makes it possible to specify the same point in time using different UTC offsets by adjusting the hour and possibly the minutes fields accordingly.

For example, Monday, May 25, 1998, at 1:30:15 PM EST would be represented as: 19980525133015.0000000-300.

An alternative representation of the same point in time would be: 19980525183015.0000000+000.

The format for intervals is: ddddddddhhmmss.mmmmmm:000

where the meaning of each field is:

- dddddddd is the number of days
- hh is the remaining number of hours
- mm is the remaining number of minutes
- ss is the remaining number of seconds
- mmmmmm is the remaining number of microseconds
- ":" is indicating that the value is an interval
- 000 (the UTC offset field) is always zero for interval properties

For example, an interval of 1 day, 13 hours, 23 minutes, 12 seconds and 0 microseconds would be: 00000001132312.000000:000.

For both timestamps and intervals, the field values MUST be zero-padded so that the entire string is always the same 25-character length.

For both timestamps and intervals, fields that are not significant MUST be replaced with asterisk ("*") characters. Not significant fields are those that are beyond the resolution of the data source. This is used to indicate the precision of the value and can only be done for an adjacent set of fields, starting with the least significant field (mmmmmm), and continuing to more significant fields. The granularity of using asterisks is always the entire field, except for the mmmmmm field where the granularity is single digits. The UTC offset field MUST NOT contain asterisks.

For example, if an interval of 1 day, 13 hours, 23 minutes, 12 seconds, and 125 milliseconds was measured with a precision of 1 millisecond, the format would be: 00000001132312.125***: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 qualifier names chosen are not part of this standard):

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

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

2.3 Supported Schema Modifications

This is a list of supported schema modifications, some of which, when used, 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.

A class can be added to or deleted from a schema.

A property can be added to or deleted from a class.

A class can be added as a subtype or supertype of an existing class.

A class can become an association as a result of the addition of an Association qualifier, plus two or more references.

A qualifier can be added to or deleted from any Named Element.

The Override qualifier can be added to or removed from a property or reference.

A method can be added to a class.

A method can override an inherited method.

Methods can be deleted, and the signature of a method can be changed.

A trigger may be added to or deleted from a class.

In defining an extension to a schema, the schema designer is expected to operate within the constraints of the classes defined in the Core model. With respect to classification, it is recommended that any added component of a system be defined as a subclass of an appropriate Core model class. It is expected that the schema designer will address the following question to each of the Core model classes: "Is the class being added a subtype of this class?" Having identified the Core model 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 superclasses 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 meta schema, but is an important device for introducing uniformity across schemas intended to represent aspects of the managed environment.

2.3.1 Schema Versions

Schema Versioning is described in Section 4 of DSP0129 (the "DMTF Release Process") and in Section 2.5.2 where the Version qualifier is discussed. Versioning takes the form m.n.u, where:

m = major version identifier in numeric form
n = minor version identifier in numeric form
u = update (errata or coordination changes) in numeric form

The Usage Rules for the Version qualifier (Section 2.5.2) provide additional information.

Classes are versioned in the CIM Schemas. A class' Version qualifier indicates the schema release when the last change to the class occurred. Class Versions in turn dictate the "Schema Version". A major version change for a class would require that the major version number of the Schema Release be incremented. All class Versions must be at the same level, or a previous level, than the Schema Release. This is because classes and models which differ in minor version numbers MUST be backwards compatible. In other words, valid instances MUST continue to be valid if the minor version number is incremented. Classes and models which differ in major version numbers will not be backwards compatible. This then requires that the major version number of the Schema Release be incremented.

The following table describes modifications to the CIM Schemas in FINAL status that would cause a major version number change. (Preliminary Models are allowed to evolve based on implementation experience.) These modifications result in changes in application behavior and/or customer code. Therefore, they force a major version update and are discouraged. (Note that the table is "exhaustive" based on current CIM experience and knowledge. Items may be added as new issues are raised and in response to the evolution of the CIM Standards.)

Other alterations (beyond those listed in the table) are considered to be interface-preserving and require that the minor version number be incremented. Updates/errata are NOT classified as having "major" or "minor" impact – but are required to correct errors, or to coordinate across standards bodies.

Changes that Increment the CIM Schema Major Version Number
(This table may be updated in the future, as experience and extensions require.)

Description	*Explanation or exceptions*
Class deletion	
Property deletion or data type change	
Method deletion or signature change	

Reorganization of values in an enumeration	The semantics and mappings of an enumeration cannot change, but values can be added in unused ranges as a minor change or update.
Movement of a class "upwards" in the inheritance hierarchy – I.E., the removal of superclasses from the inheritance hierarchy	The removal of superclasses results in the deletion of properties or methods. New classes CAN be inserted as superclasses as a minor change or update. Inserted classes MUST NOT change keys or add "Required" properties.
Addition of Abstract, Indication or Association qualifiers to an existing class	
Change of an association reference "downward" in the object hierarchy (i.e., to a subclass) or to a different part of the hierarchy	The change of an association reference to a subclass could invalidate existing instances.
Addition or removal of a Key or Weak qualifier	
Addition of a Required qualifier	
Decrease in MaxLen, decrease in MaxValue, increase in MinLen or increase in MinValue	Decreasing a maximum, or increasing a minimum invalidates current data. The opposite change (increasing a maximum) results in truncated data, where necessary.
Decrease in Max or increase in Min cardinalities	
Addition or removal of Override qualifier	There is one exception to this – An Override qualifier can be added if a property is promoted to a superclass, and it is necessary to maintain the specific qualifiers and descriptions in the original subclass. In this case, there is no change to existing instances.
Change in the following qualifiers: In/Out, Units	

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> although 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 when the schema is implied by the context.

Examples of fully-qualified class names:

- CIM_ManagedSystemElement: the root of the CIM managed system element hierarchy.
- CIM_ComputerSystem: the object representing computer systems in the CIM schema.
- CIM_SystemComponent: 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, 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 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 Meta Qualifiers

This table lists the qualifiers that are used to refine the definition of the meta constructs 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.

2.5.2 Standard Qualifiers

This 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 extension 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 meta-model constructs. These limitations are identified in the “Applies To” column. Second, for a particular meta-model construct like associations, the use of the legal qualifiers may be further constrained because some qualifiers are mutually exclusive or the use of one qualifier implies some restrictions on the value of another qualifier, and so on. These usage rules are documented in the “Meaning” column of the table. Third, legal qualifiers are not inherited by meta-model constructs. For example, the MAXLEN qualifier that applies to properties is not inherited by references.

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

QUALIFIER	DEFAULT	APPLIES TO	TYPE
ABSTRACT	FALSE	Class, Association, Indication	BOOLEAN
MEANING: 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
MEANING: Defines the "parent" component of an Aggregation association. **Usage Rule:** The Aggregation and Aggregate qualifiers are used together – Aggregation qualifying the association, and Aggregate specifying the "parent" reference			
AGGREGATION	FALSE	Association	BOOLEAN
MEANING: Indicates that the association is an aggregation.			

<table>
<tr><th>QUALIFIER</th><th>DEFAULT</th><th>APPLIES TO</th><th>TYPE</th></tr>
<tr><td>ARRAYTYPE</td><td>"Bag"</td><td>Property, Parameter</td><td>STRING</td></tr>
<tr><td colspan="4">MEANING: Indicates the type of the qualified array. Valid values are "Bag", "Indexed" and "Ordered".

For a “Bag” array type, no significance is defined for the array index other than as a convenience for accessing the elements of the array. For example, there can be no assumption that the same index returns the same value for every access to the array.

For an “Ordered” array type, the array index is significant as long as no array elements are added, deleted, or changed. If this is true, the same index returns the same value for every access to the array.

For an “Indexed” array type, the array maintains the correspondence between element position and value.

Usage rule: The ArrayType qualifier MUST only be applied to properties and method parameters that are arrays (defined using the square bracket syntax specified in Appendix A).

Usage Note: For any of the ArrayTypes, elements in the array may be duplicated.</td></tr>
<tr><td>BITMAP</td><td>NULL</td><td>Property, Method, Parameter</td><td>STRING ARRAY</td></tr>
<tr><td colspan="4">MEANING: Indicates bit positions that are significant in a bit map. The bit map is evaluated from the right, starting with the least significant value. This value is referenced as “0”. For example, using a uint8 data type, the bits take the form, Mxxx xxxL, where M and L designate the most and least significant bits, respectively. The least significant bits referenced as “0”, and the most significant bit is “7”. The position of a specific value in the BitMap array defines an index that is used in selecting a string literal from the BitValues array.

Usage Rule: The number of entries in the BitValues and BitMap arrays MUST match.</td></tr>
<tr><td>BITVALUES</td><td>NULL</td><td>Property, Method, Parameter</td><td>STRING ARRAY</td></tr>
<tr><td colspan="4">MEANING: Provides translation between a bit position value and an associated string. See the description for the BitMap qualifier.

Usage Rule: The number of entries in the BitValues and BitMap arrays MUST match.</td></tr>
<tr><td>COUNTER</td><td>FALSE</td><td>Property, Method, Parameter</td><td>BOOLEAN</td></tr>
<tr><td colspan="4">MEANING: Applicable only to unsigned integer types. Represents a non-negative integer that monotonically increases until it reaches a maximum value of 2^n-1, when it wraps around and starts increasing again from zero. N can be 8, 16, 32 or 64 depending on the datatype of the object that the qualifier is applied to. Counters have no defined "initial" value, and thus, a single value of a Counter has (in general) no information content</td></tr>
<tr><td>COMPOSITION</td><td>FALSE</td><td>Association</td><td>BOOLEAN</td></tr>
<tr><td colspan="4">MEANING: Refines the definition of an Aggregation association, adding the semantics of a whole-part/compositional relationship, to distinguish it from a collection or basic aggregation. This is necessary to better map CIM associations into UML where whole-part relationships are considered compositions. The semantics conveyed by Composition align with that of the UML Specification from the OMG. Quoting from Section 3.48 of the V4 UML Specification (September 2001):

Composite aggregation is a strong form of aggregation, that requires that a part instance be included in AT MOST ONE composite at a time and that the composite object has sole responsibility for the disposition of its parts.

Use of this qualifier imposes restrictions on the membership of the “collecting” object (the whole). Care should be taken when entities are added to the aggregation – since they MUST be “parts” of the whole. Also, if the collecting entity (the whole) is deleted, it is the responsibility of the implementation to dispose of the parts. The behavior may vary with the type of collecting entity whether the parts are also deleted. This is very different from that of a collection, since a collection may be removed without deleting the entities that are collected.

Usage Rule: The Aggregation and Composition qualifiers are used together. Aggregation indicating the general nature of the association and Composition indicating more specific semantics of whole-part relationships. This duplication of information is necessary since Composition is a more recent addition to the list of qualifiers – and applications may be built only</td></tr>
</table>

QUALIFIER	DEFAULT	APPLIES TO	TYPE
understanding the earlier Aggregation qualifier.			
DEPRECATED	NULL	Any	STRING ARRAY
MEANING: Indicates that the feature the qualifier is applied to (e.g., class, property, etc.) is tolerated but not recommended, and may be superceded by another Method. Existing instrumentation should continue to support the old feature, so as not to break current applications, and are encouraged to support the new approach as well. Replacement or substitution information is conveyed in the contents of the qualifier's string value. The qualifier serves only as an indication that the Deprecated item is not recommended for use in new development efforts. Existing and new implementations MAY still use this feature, but SHOULD move to the replacement or substitution feature as soon as possible. The Deprecated feature MAY be removed in a future major version release of the CIM Schema (ex, CIM 2.x to CIM 3.0). The qualifier acts inclusively in that if a class is deprecated, all of the properties, references and methods in that class are also deprecated. However, no subclasses or associations that reference that class are deprecated, unless explicitly declared as such. For clarity, all deprecated properties and associations SHOULD be specifically labeled as DEPRECATED. The syntax for the Deprecated qualifier's string value should indicate replacement values (if any exist) as follows: <className> [*("." <propertyName>) ["." <methodName> ["(" <parameterName> ")"]]] The property is defined as a string ARRAY to allow a single feature to be replaced by multiple other constructs. **Usage Rule:** The string value for the qualifier indicates what the replacement or substitution feature is for the deprecated construct. If there is no replacement feature, then the string value MUST be set to "No value". When a feature is Deprecated, its description MUST indicate why it was deprecated and how the replacement features (if any are defined) are used. For example, "The property X is deprecated in lieu of the method Y defined in this class. The reason for this change is that the property is actually causing a change of state and requires an input parameter." **Note 1**: Publishing a feature by deprecating and replacing it results in duplicate representations of the feature. This is of particular concern when deprecated classes are replaced by new classes, and instances may be duplicated. To allow a management application to detect such duplication, implementations SHOULD document in a ReadMe, in MOF or by other documentation, how duplicate instances are detected. **Note 2**: Properties with the KEY qualifier CAN be deprecated but MUST NOT be left blank or NULL, MUST keep the KEY qualifier, and MUST NOT be replaced with a new KEY property. If any of the latter 3 changes occur, this breaks existing instrumentation and applications and hence would not be "tolerated" changes (a requirement for using the DEPRECATED qualifier, as documented in the first sentence of this description). In scenarios where a KEY property should be deprecated AND replaced, only one option is available- it is necessary to deprecate the entire class and therefore its properties, methods, references, etc.			
DESCRIPTION	NULL	Any	STRING
MEANING: Provides a description of a Named Element.			
DISPLAYNAME	NULL	Any	STRING
MEANING: Defines a name that is displayed on UI instead of the actual name of the element.			
DN	FALSE	Property, Parameter, Method	BOOLEAN
MEANING: When applied to a string element the DN qualifier specifies that the string MUST be a distinguished name as defined in Section 9 of X.501 "Information Technology – Open Systems Interconnection – The Directory: Models" and the string representation defined in RFC2253 "Lightweight Directory Access Protocol (v3): UTF-8 String Representation Of Distinguished Names". This qualifier MUST NOT be applied to qualifiers that are not of intrinsic data type string.			
EMBEDDEDOBJECT	FALSE	Property, Parameter, Method	BOOLEAN
MEANING: This qualifier can only be used in conjunction with string-valued features. It indicates that the qualified entity contains an encoding of an instance's data. Please see Appendix H for examples and important details.			

QUALIFIER	DEFAULT	APPLIES TO	TYPE
EXCEPTION	FALSE	Class	BOOLEAN
MEANING: Indicates that the class and all subclasses of this class describe transient Exception information. The definition of this qualifier is identical to the Abstract Qualifier except that this qualifier cannot be overridden. It is not possible to create instances of Exception classes. **Usage Rule**: The Exception Qualifier is used to denote a class hierarchy that defines transient (i.e., very short-lived) Exception objects. Instances of Exception classes are used to communicate exception information between CIMEntities. The Exception Qualifier cannot be used with the Abstract Qualifier. The subclass of an Exception class, if it exists, MUST be an Exception class.			
EMBEDDEDINSTANCE	NULL	Property, Parameter, Method	STRING
MEANING: The EMBEDDEDINSTANCE qualifier modifies a string property, method parameter or return value to indicate that it contains an embedded instance. The definition of this qualifier is identical to the EMBEDDEDOBJECT Qualifier except that it restricts the set of permissible values for the qualified property, parameter, or method return to instances of the class, or to one of its subclasses, that is designated by the string value. Usage Rule: When the EMBEDDEDINSTANCE qualifier is used, its value MUST be specified and not NULL. The encoding of the instance contained in the string qualified by EMBEDDEDINSTANCE follows the same rules as for EMBEDDEDOBJECT. This allows the same parsing engine that decodes the instance/indication envelope to decode the EMBEDDEDOBJECT string. (As with EMBEDDEDOBJECT, encodings are defined by an implementation and are not the subject of this document.) Please see Appendix H for examples and further explanation.			
EXPERIMENTAL	FALSE	Any	BOOLEAN
MEANING: Represents that the specified element is proposed for inclusion in a future release of the CIM Schemas, but is not currently a part of the standard Schema. The specified element is in an interim state - available for experimentation and implementation experience, but not a part of the standard. For example, if an implementation uses schema from a DMTF Preliminary Standard, the features from the Preliminary Standard should be labeled with the EXPERIMENTAL qualifier. Based on implementation experience, changes may occur to this element in future releases, it may be standardized "as is", or it may be removed. An implementation does not have to support an EXPERIMENTAL feature.			
GAUGE	FALSE	Property, Method, Parameter	BOOLEAN
MEANING: Applicable only to unsigned integer types. Represents an integer that may increase or decrease in any order of magnitude. The value of a Gauge is capped at the implied limits of the property's datatype. If the information being modeled exceeds an implied limit, the value represented is that limit. Values do not wrap. For unsigned integers, the limits are zero (0) to 2^n-1, inclusive. For signed integers, the limits are –(2^(n-1)) to 2^(n-1)-1, inclusive. N can be 8, 16, 32, or 64 depending on the datatype of the property that the qualifier is applied to.			
IN	TRUE	Parameter	BOOLEAN
MEANING: Indicates that the associated parameter is used to pass values to a method.			
KEY	FALSE	Property, Reference	BOOLEAN
MEANING: Indicates that the property is part of the namespace handle (see Section 5.3.1.2 for information about namespace handles). If more than one property has the KEY qualifier, then all such properties collectively form the key (a compound			

QUALIFIER	DEFAULT	APPLIES TO	TYPE
key). **Usage Rule:** Keys are written once at object instantiation and MUST NOT be modified thereafter. Default values are not applied to a KEY-qualified properties. Properties of an array type MUST NOT be qualified with KEY.			
MAPPINGSTRINGS	NULL	Any	STRING ARRAY
MEANING: Mapping strings for one or more management data providers or agents. See Section 2.5.5 and 2.5.6 for more details.			
MAX	NULL	Reference	uint32
MEANING: Indicates the maximum cardinality of the reference (i.e., the maximum number of values a given reference may 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, Method, Parameter	uint32
MEANING: Indicates the maximum length, in characters, of a string data item. MAXLEN may also be applied to elements with a string array value, in which case it is interpreted as applying to each string in the string array. When overriding the default value, any unsigned integer value (uint32) can be specified. A value of NULL implies unlimited length.			
MAXVALUE	NULL	Property, Method, Parameter	sint64
MEANING: Maximum value of this element.			
MIN	0	Reference	uint32
MEANING: Indicates the minimum cardinality of the reference (i.e., the minimum number of values a given reference may 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.			
MINLEN	0	Property, Method, Parameter	uint32
MEANING: Indicates the minimum length, in characters, of a string data item. When overriding the default value, any unsigned integer value (uint32) can be specified. The default is zero length for the minimum length..			
MINVALUE	NULL	Property, Method, Parameter	sint64
MEANING: Minimum value of this element.			
MODELCORRESPONDENCE	NULL	Any	STRING ARRAY
MEANING: Indicates a correspondence between two elements in the CIM Schema. The referenced elements MUST be defined in a standard or extension MOF file, such that the correspondence can be examined. If possible, forward referencing of elements SHOULD be avoided. Object elements are identified using the following syntax: <className> [*("."(<propertyName> | < referenceName>)) ["." <methodName> ["(" <parameterName> ")"]]] Note that the basic relationship between the referenced elements is a "loose" correspondence – simply indicating that the elements are coupled. And, this coupling MAY be unidirectional. Additional qualifiers MAY be used to describe a tighter			

QUALIFIER	DEFAULT	APPLIES TO	TYPE
coupling. The following list provides examples of several correspondences found in CIM and vendor schemas: · A vendor defines an Indication class "corresponding" to a particular CIM property or method – i.e., Indications are generated based on the values or operation of the property or method (in this case, the ModelCorrespondence MAY only be on the vendor's Indication class which is an extension to CIM) · A property provides more information for another - for example, an enumeration has an allowed value of "Other", and another property further clarifies the intended meaning of "Other"; alternately, a property specifies status and another property provides human-readable strings (using an array construct) expanding on this status (in these cases, ModelCorrespondence is found on both properties, each referencing the other – note also that referenced array properties MAY NOT be ordered but carry the default ArrayType qualifier definition of "Bag") · A property is defined in a subclass to supplement the meaning of an inherited property (in this case, the ModelCorrespondence is only found on the construct in the subclass) · Multiple properties taken together are needed for complete semantics – for example, one property might define units, another a multiplier and a third property define a specific value (in this case, ModelCorrespondence is found on all related properties, each referencing all the others) · Multi-dimensional arrays are desired – for example, one array may define names while another the name formats (in this case, the arrays would each be defined with the ModelCorrespondence qualifier, referencing the other array properties or parameters, and would also be indexed - i.e., also carry the ArrayType qualifier with the value "Indexed") The specific semantics of the correspondence are based on the elements themselves. ModelCorrespondence is only a hint or indicator of a relationship between the elements.			
NONLOCAL			
MEANING: This instance-level qualifier, and the corresponding pragma, were removed as an erratum by CR1461.			
NONLOCALTYPE			
MEANING This instance-level qualifier, and the corresponding pragma, were removed as an erratum by CR1461.			
NULLVALUE	NULL	Property	STRING
MEANING: Defines a value that indicates the associated property is NULL (i.e., the property is considered to have a valid or meaningful value. The NullValue qualifier may only be used with string and integer valued properties. When used with an integer type, the qualifier value is a MOF integerValue. The syntax for representing an integer value is: ["+" / "-"] 1*<decimalDigit> The content, maximum number of digits and represented value are constrained by the datatype of the qualified property. Note this qualifier cannot be overridden as it seems unreasonable to permit a subclass to return a different null value to that of the superclass.			
OCL	NULL	Class, Association, Indication, Method	STRING ARRAY
MEANING: Indicates that the qualified element specifies one or more constraints. These constraints are defined using the Object Constraint Language syntax, as defined by the Open Management Group (OMG), in the UML v2 OCL specification (http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf). USAGE: The OCL array contains string values using the syntax defined by UML's OCL 'inv:', 'pre:' and 'post:' clauses (invariants, and method pre and post clauses). The context of the constraint is defined by the qualified element (i.e., the context is the class or method that carries the qualifier).			

QUALIFIER	DEFAULT	APPLIES TO	TYPE
Note: • 'self' refers to the context • 'result' refers to the result of a method • A property in a class is defined using the syntax: ("self" | <class name>) "." <property name> • Following an association to a specific role is defined using the syntax: ("self" | <class name>) "." <association class name> "[" <role name> "]" • Various set operations can be performed as a result of following an association: sum, size, isEmpty, notEmpty, includes, excludes, count(<specific object>) These are invoked by specifying "->" and the operation name, after the association and role name are selected as above. • A few basic tests exist: oclIsTypeOf(<class name>) for specific class checks oclIsKindOf(<class name>) for class or superclass checking oclIsNew() for operation post condition checking that a new instance is created (i.e., did not previously exist) oclIsUndefined() checking for null values For example, checking that both property x and y cannot be null is specified using the following syntax, defined on a class: OCL {"inv: not(self.x.oclIsUndefined() AND self.y.oclIsUndefined())"}			
OCTETSTRING	FALSE	Property, Parameter, Method	BOOLEAN
MEANING: This qualifier is used to identify the qualified property or parameter as an octet string. When used in conjunction with an unsigned 8-bit integer (uint8) array, the OCTETSTRING qualifier indicates that the unsigned 8-bit integer array represents a single octet string. When used in conjunction with arrays of strings, the OCTETSTRING qualifier indicates that the qualified character strings are encoded textual conventions representing octet strings. The text encoding of these binary values conforms to the following grammar: “0x” 4*(<hexDigit> <hexDigit>). In both cases, the first 4 octets of the octet string (8 hexadecimal digits in the text encoding) are the number of octets in the represented octet string with the length portion included in the octet count (e.g., "0x00000004" is the encoding of a 0 length octet string).			
OUT	FALSE	Parameter	BOOLEAN
MEANING: Indicates that the associated parameter is used to return values from a method.			
OVERRIDE	NULL	Property, Method, Reference	STRING
MEANING: Indicates that the element in the derived class overrides another element (of the same name) defined in the ancestry of the class. The overriding element effectively takes the place of the element being overridden, for instances of the			

QUALIFIER	DEFAULT	APPLIES TO	TYPE
class on which the overriding element appears, and any of its subclasses (unless, of course, the overriding element is itself overridden in one of the subclasses). The flavor of the qualifier is defined as 'Restricted', so that the Override qualifier is NOT repeated in (i.e., inherited by) each subclass. The effect of the override is inherited, but not the identification of the Override qualifier itself. This enables new Overrides in subclasses to be easily located and applied. The value of this qualifier MAY identify the ancestral class whose subordinate element (property or method) is overridden. The format of the string to accomplish this is: [<className> "."] <IDENTIFIER>. If the class name is omitted, the element being overridden is found by searching the ancestry of the class until a definition of an appropriately-named subordinate element (of the same meta-schema class) is found. **Usage Rule**: The Override qualifier can only refer to elements of the same meta-schema class, e.g., properties can only override properties, etc. Also, it is not allowed to change an element's name or signature when overriding.			
PROPAGATED	NULL	Property	STRING
MEANING: 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: [<className> "."] <IDENTIFIER> **Usage Rule:** When the PROPAGATED qualifier is used, the KEY qualifier MUST be specified with a value of TRUE.			
READ	TRUE	Property	BOOLEAN
MEANING: Indicates that the property is readable.			
REQUIRED	FALSE	Property, Reference	BOOLEAN
MEANING: Indicates that a non-NULL value is required for the property or reference. Properties of a class that are inherent characteristics of a class and identifying in nature (e.g., domain name, file name, burned-in device identifier, IP address, etc.) that are likely to be useful for applications as query entry points and that are not KEY properties should be REQUIRED properties			
REVISION (DEPRECATED)	NULL	Class, Association, Indication,	STRING
MEANING: **DEPRECATED - See VERSION Qualifier** Provides the minor revision number of the schema object. **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
MEANING: The name of the schema that contains the feature.			
SOURCE			
MEANING: This instance-level qualifier, and the corresponding pragma, were removed as an erratum by CR1461.			
SOURCETYPE			
MEANING: This instance-level qualifier, and the corresponding pragma, were removed as an erratum by CR1461.			
STATIC	FALSE	Property, Method	BOOLEAN

QUALIFIER	DEFAULT	APPLIES TO	TYPE
MEANING: For methods indicates that the method is a class method that does not depend on any per-instance data. For properties, indicates that the property is a class variable rather than an instance variable.			
TERMINAL	FALSE	Class, Association, Indication	BOOLEAN
MEANING: Indicate that the class can have no subclasses. If such a subclass is declared the compiler will generate an error. Note this qualifier cannot coexist with the Abstract qualifier. If both are specified the compiler generates an error.			
UNITS	NULL	Property, Method, Parameter	STRING
MEANING: Indicates the units of the associated data item (e.g., a Size data item might have Units of "bytes"). The complete set of standard units is defined in Appendix C.			
VALUEMAP	NULL	Property, Method, Parameter	STRING ARRAY
Defines the set of permissible values for the qualified property, method return or method parameter. The ValueMap qualifier can be used alone, or in combination with the Values qualifier. When used with the Values qualifier, the location of the value in the ValueMap array determines the location of the corresponding entry in the Values array. Where: ValueMap may only be used with string or integer types. When used with a string type, a ValueMap entry is a MOF stringvalue. When used with an integer type, a ValueMap entry is a MOF integervalue or an integervaluerange as defined here. integervaluerange: [integervalue] ".." [integervalue] A ValueMap entry of : "x" claims the value x, "..x" claims all values less than and including x, "x.." claims all values greater than and including x, and, ".." claims all values not otherwise claimed. The values claimed are constrained by the type of the associated property. ValueMap = ("..") is not permitted. If used with a Value array, then all values claimed by a particular ValueMap entry applies to the corresponding Value entry. Example: [Values {"zero&one", "2to40", "fifty", "the unclaimed", "128-255"}, ValueMap {"..1","2..40" "50", "..", "x80.." }] uint8 example; In this example, where the type is uint8, "..1" and "zero&one" map to "0" and 1" "2..40" and "2to40" map to "2" thru "40" ".." and "the unclaimed" map to "41" thru "49" and to "51" thru "127" "0x80.." and "128-255" map to "128" thru "255"			
VALUES	NULL	Property, Method, Parameter	STRING ARRAY
MEANING: Provides translation between an integer values and strings (such as abbreviations or English terms) in the ValueMap array, and an associated string at the same index in the Values array. If a ValueMap qualifier is not present, the Values array is indexed (zero relative) using the value in the associated property, method return type or method parameter. If a ValueMap qualifier is present, the Values index is defined by the location of the property value in the ValueMap.			

QUALIFIER	DEFAULT	APPLIES TO	TYPE
Usage Rule: The number of entries in the Values and ValueMap arrays MUST match.			
VERSION	NULL	Class, Association, Indication	STRING
MEANING: Provides the version information of the object. This is incremented when changes are made to the object. **Usage Rule:** Starting with CIM Schema 2.7 (including extension schema), the VERSION qualifier MUST be present on each class to indicate the version when the class was last updated. The string representing the version comprises three decimal integers separated by periods, i.e., M.N.U, or, more formally, 1*<decimalDigit> “.” 1*<decimalDigit> “.” 1*<decimalDigit>, where **M -** The major version in numeric form that changed the class **N** - The minor version in numeric form that changed the class **U** - The update (e.g. errata, patch, ...) in numeric form that changed the class. NOTES: The addition/removal of the *Experimental* qualifier does not require that the version information be updated. The version change is applicable to only those elements that are local to the class. In other words, the version change of a superclass does not require the version in the subclass to be updated. Examples: Version("2.7.0") Version("1.0.0")			
WEAK	FALSE	Reference	BOOLEAN
MEANING: 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 other classes in the association are repeated in the referenced class and tagged with a propagated qualifier.			
WRITE	FALSE	Property	BOOLEAN
MEANING: Indicates that the modeling semantics of a property support modification of that property by “consumers”. This qualifier is only intended to capture modeling semantics and specifically is not intended to address more dynamic characteristics such as provider capability or authorization rights.			

2.5.3 Optional Qualifiers

The optional qualifiers listed in this table 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 are provided in the specification to avoid random user-defined qualifiers for these recurring situations.

Qualifier	Default	Applies To	Type
ALIAS	NULL	Property, Reference, Method	STRING
MEANING: Establishes an alternate name for a property or method in the schema.			

<table>
<tr><td>Qualifier</td><td>Default</td><td>Applies To</td><td>Type</td></tr>
<tr><td>DELETE</td><td>FALSE</td><td>Association, Reference</td><td>BOOLEAN</td></tr>
<tr><td colspan="4">MEANING: 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.
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.
Usage Rule: Applications MUST chase associations according to the modeled semantic and delete objects appropriately. Note: This usage rule must be verified when the CIM security model is defined.</td></tr>
<tr><td>EXPENSIVE</td><td>FALSE</td><td>Any</td><td>BOOLEAN</td></tr>
<tr><td colspan="4">MEANING: Indicates the element is expensive to manipulate and/or compute.</td></tr>
<tr><td>IFDELETED</td><td>FALSE</td><td>Association, Reference</td><td>BOOLEAN</td></tr>
<tr><td colspan="4">MEANING: Indicates that all objects qualified by DELETE within the association MUST be deleted if the referenced object or the association, respectively, is deleted.</td></tr>
<tr><td>INVISIBLE</td><td>FALSE</td><td>Association, Property, Method, Reference, Class</td><td>BOOLEAN</td></tr>
<tr><td colspan="4">MEANING: Indicates that the element is defined only for internal purposes (for example, as an intermediate value in a calculation or to facilitate association semantics) and should not be displayed or otherwise relied upon.</td></tr>
<tr><td>LARGE</td><td>FALSE</td><td>Property, Class</td><td>BOOLEAN</td></tr>
<tr><td colspan="4">MEANING: Indicates the property or class requires a large amount of storage space.</td></tr>
<tr><td>PROVIDER</td><td>NULL</td><td>Any</td><td>STRING</td></tr>
<tr><td colspan="4">MEANING: An implementation specific handle to the instrumentation that populates those elements in the schemas that refer to dynamic data.</td></tr>
<tr><td>PROPERTYUSAGE</td><td>“CURRENTCONTEXT”</td><td>Property</td><td>STRING</td></tr>
</table>

<table>
<tr><td>Qualifier</td><td>Default</td><td>Applies To</td><td>Type</td></tr>
<tr><td colspan="4">
MEANING: This qualifier allows classification of properties according to how they are intended to be used from the managed elements point of view. They are intended to allow the managed element to convey intent for property usage for the managed element. It is not intended to convey what access CIM has to the properties, i.e.. not all properties classified as configuration are necessarily writeable. Some configuration properties may be maintained by the provider or resource that the managed element represents, and not by CIM. The PropertyUsage qualifier is meant to allow distinguishing between properties that represent attributes of a managed resource vs. capabilities of a managed resource vs. configuration data for a managed resource vs. metrics about or from a managed resource vs. state information for a managed resource. If the qualifiers value is set to CurrentContext (i.e., the default value) then the PropertyUsage qualifiers actual value should be determined by looking at what class the property is placed in. The rules for which classes/subclasses result in which default PropertyUsage values are defined below:

Class>CurrentContext PropertUsage Value
Setting > Configuration
Configuration > Configuration
Statistic > Metric ManagedSystemElement > State Product > Descriptive
FRU > Descriptive
SupportAccess > Descriptive
Collection > Descriptive

Usage Rules: The valid values for this qualifier are UNKNOWN, OTHER, CURRENTCONTEXT, DESCRIPTIVE, CAPABILITY, CONFIGURATION, STATE and METRIC.

UNKNOWN indicates that the property's usage qualifier has not been determined and set.

OTHER indicates that the property's usage is not Descriptive, Capabilities, Configuration, Metric, or State.

CURRENTCONTEXT indicates that the PropertyUsage qualifiers value shall be inferred based on the class placement of the property according to the following rules:

*If the property is in a subclass of Setting or Configuration then the PropertyUsage value of CURRENTCONTEXT should be treated as CONFIGURATION.

*If the property is in a subclass of Statistics then the PropertyUsage value of CURRENTCONTEXT should be treated as METRIC.

*If the property is in a subclass of ManagedSystemElement then the PropertyUsage value of CURRENTCONTEXT should be treated as STATE.

*If the property is in a subclass of Product, FRU, SupportAccess or Collection then the PropertyUsage value of CURRENTCONTEXT should be treated as DESCRIPTIVE.

DESCRIPTIVE indicates that the property contains information that describes the managed element. This can be properties like vendor, description, caption, etc. These properties are generally not good candidates for representation in Settings subclasses.

CAPABILITY indicates that property contains information that reflects the inherent capabilities of the managed element regardless of its configuration. These are usually specifications of a product. For example, VideoController.MaxMemorySupported=128 would be a capability.

CONFIGURATION indicates that the property contains information that influences or reflects the configurational state of the managed element. These properties are candidates for representation in Settings subclasses. VideoController.CurrentRefreshRate would be a configuration value.

STATE indicates that the property contains information that reflects or can be used to derive the current status of the managed element.

METRIC indicates that the property contains a numerical value representing a statistic or metric reporting performance oriented and/or accounting oriented information for the managed element. This would be appropriate for properties containing counters like 'BytesProcessed'.
</td></tr>
<tr><td>SYNTAX</td><td>NULL</td><td>Property, Reference, Method, Parameter</td><td>STRING</td></tr>
</table>

Qualifier	Default	Applies To	Type
MEANING: Specific type assigned to a data item. **Usage Rule:** Must be used with the SyntaxType qualifier.			
SYNTAXTYPE	NULL	Property, Reference, Method, Parameter	STRING
MEANING: Defines the format of the SYNTAX qualifier. **Usage Rule:** Must be used with the SYNTAX qualifier			
TRIGGERTYPE	NULL	Class, Property, Method, Association, Indication, Reference	STRING
MEANING: Indicates the circumstances that cause a trigger to be fired. **Usage Rule:** The trigger types vary by meta-model 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.			
UNKNOWN VALUES	NULL	Property	STRING ARRAY
MEANING: Defines a set of values whose presence indicates that the value of the associated property is unknown – that is that the property cannot be considered as having a valid or meaningful value. The conventions and restrictions used for defining unknown values are the same as those applicable to the ValueMap qualifier. Note this qualifier cannot be overridden as it seems unreasonable to permit a subclass to treat as a known value a value that is treated as unknown by some superclass.			
UNSUPPORTED VALUES	NULL	Property	STRING ARRAY
MEANING: Defines a set of values whose presence indicates that the value of the associated property is unsupported – that is that the property cannot be considered as having a valid or meaningful value. The conventions and restrictions used for defining unsupported values are the same as those applicable to the ValueMap qualifier. Note this qualifier cannot be overridden as it seems unreasonable to permit a subclass to treat as a supported value a value that is treated as unknown by some superclass.			

2.5.4 User-defined Qualifiers

The user can define any additional arbitrary named qualifiers. However, it is recommended that only defined qualifiers be used, and that the list of qualifiers be extended only 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 MAPPINGSTRINGS 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.

Therefore, the mapping qualifier can 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 is 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 would be as follows:

```
   [MAPPINGSTRINGS{"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 an 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 name

standards body (e.g., "IETF"), a company name (e.g., "Acme") for defining the mappings to a company's private MIB, and/or an appropriate management protocol (e.g., "SNMP"). For the IETF case, the ASN.1 module name, not the RFC number, should be used as the MIB name (e.g., 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 (e.g., directories) using an application-specific protocol (e.g., LDAP).

The syntax for mapping MIF attributes as defined in Section 2.5.5 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" means that the rest of the string is interpreted as MIF data

"MIB" means that the rest of the string is interpreted as a variable name of a MIB

"OID" means that 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" is used for MIF mappings

"Naming_Authority | MIB_Name" is used for MIB mappings

"Naming_Authority | Protocol_Name" is 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" is used for MIF mappings

"Variable_Name" is used for MIB mappings

"OID" is 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;
```

3 Managed Object Format

The management information is described in a language based on Interface Definition Language (IDL) [3] called the Managed Object Format (MOF). This document 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 [7]; 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 Section 5). 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. In particular, MOF declarations must be ordered correctly with respect to the target implementation state. For example, if the specification references a class without defining it first, the reference is valid only if the server already has a definition of that class. 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 add classes or modify classes.

If the specification references a class without defining it first, the server must reject it as invalid if it does not already have a definition of that class.

3.3 Instance Declarations

If the specification references a class without defining it first, the server must reject it as invalid if it does not already have a definition 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.

4 MOF 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 carriage return, line feed 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 meta model and as such, will not be preserved across compilations. In other words, 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 Section 1. 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 must 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 F, Unicode Usage.

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 these 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 class 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 may have an optional qualifier list. A method has a signature consisting of its return type, plus its parameters and their type and usage.

This sample shows how to declare a class:

```
          [abstract]
class Win32_LogicalDisk
{
          [read]
    string DriveLetter;
          [read, Units("KiloBytes")]
    sint32 RawCapacity = 0;
          [write]
    string VolumeLabel;
          [Dangerous]
    boolean Format([in] boolean 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 any such 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 meta data about a property, method, method parameter, or class, 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 meta schema). 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 { "this", "a", "bag", "is" }    // string []
id(12)                               // uint32
idlist { 21, 22, 40, 43 }            // uint32 []
apple(3.14)                          // real32
oranges { -1.23E+02, 2.1 }           // real32 []
```

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

For example:

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

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;
    };
```

Qualifiers can be transmitted automatically from classes to derived classes, or from classes to instances, subject to certain rules. The rules behind how the transmission 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 transmitted to all of its derived classes, or it may be designated as belonging specifically to that class and not transmittable.

The former is achieved by using the ToSubclass flavor, and the latter by using the Restricted flavor. These two flavors MUST NOT be used at the same time. In addition, if a qualifier gets transmitted to its derived classes, the qualifier flavor can be used to control whether or not derived classes can 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.

This is done by using the EnableOverride or DisableOverride flavors, which MUST NOT be used at the same time. If a qualifier does not get transmitted to its derived classes, these two flavors are meaningless and MUST be ignored.

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. 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.

The recognized flavor types are:

PARAMETER	Interpretation	Default
ToSubclass	The qualifier is inherited by any subclass.	ToSubclass
Restricted	The qualifier applies only to the class in which it is declared.	ToSubclass
EnableOverride	If ToSubclass is in effect: The qualifier is overridable.	EnableOverride
DisableOverride	If ToSubclass is in effect: The qualifier cannot be overriden.	EnableOverride
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 :	no

PARAMETER	Interpretation	Default
	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. 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. Example: if an English description is translated into Mexican Spanish the actual name of the qualifier is: DESCRIPTION_es_MX.	

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, this class has one property (Volume) which serves as its' key:

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

In this example, 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 each 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 Association Declarations

An association is a special kind of a class describing a link between other classes. As such, they also provide a type system for instance constructions. Associations are just like other classes with a few additional semantics explained below.

4.6.1 Declaring an Association

An association is declared by specifying these components:

The qualifiers of the association (at least the ASSOCIATION qualifier, if it doesn't have a supertype). Further qualifiers may be specified as a list of qualifier/name bindings separated by commas ",". The entire qualifier list is enclosed in square brackets ("[" and "]").

. The association name.

The name of the association from which this association is derived (if any).

The association references which define pointers to other objects linked by this association. References may also have qualifier lists, expressed in the same way as the association qualifier list. Especially the qualifiers to specify cardinalities of references are important to be mentioned (see2.5.2. "Standard Qualifiers"). In addition, a reference has a data type, and (optionally) a default (initializer) value.

Additional association properties which define further data members of this association. They are defined in the same way as for ordinary classes.

The methods supported by the association. They are defined in the same way as for ordinary classes.

The following example shows how to declare an association (assuming given classes CIM_A and CIM_B):

```
  [Association]
class CIM_LinkBetweenAandB : CIM_Dependency
{
     [Override ("Antecedent")]
  CIM_A Ref Antecedent;
     [Override ("Dependent")]
  CIM_B Ref Dependent;
};
```

4.6.2 Subassociations

To indicate that an association is a subassociation of another association, the same notation as for ordinary classes is used, i.e. the derived association is declared by using a colon followed by the superassociation name. (An example is provided above.)

4.6.3 Key References and Properties

Instances of an association also require some mechanism through which the instances can be distinguished, implied by the fact that they are just a special kind of a class. Designating one ore more references/properties with the reserved KEY qualifier provides instance identification.

A reference/property of an association is (part of) the association key if the KEY qualifier is applied.

```
    [Association, Aggregation]
class CIM_Component
{
        [Aggregate, Key]
    CIM_ManagedSystemElement Ref GroupComponent;
        [Key]
    CIM_ManagedSystemElement Ref PartComponent;
};
```

In principle, the key definition of association follows the same rules as for ordinary classes. Compound keys are supported in the same way. Also a new subassociation **cannot** define any additional key properties/references.

If any reference to a class has the WEAK qualifier, the KEY-qualified properties of the other class, whose reference is not WEAK-qualified are propagated to the class. (see subchapter 4.5.5 "Key Properties").

4.6.4 Object References

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

The namespace in which the object resides.

The class name of the object.

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_AnotherClass ref Inst1;
    Acme_Aclass       ref Inst2;
};
```

In the above declaration, Inst1 can only be set to point to objects of type Acme_AnotherClass. Also see Section 4.12.2on Initializing References Using Aliases.

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

4.7 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 meta qualifiers are declared as:

```
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.

4.8 Instance Declarations

This section is specific to instance declarations, Method declarations are covered in Section 4.9. Instances are declared using the keyword sequence "instance of" and the class name. The property values of the instance may be initialized within an initialization block. Any Qualifiers specified for the Instance MUST already be present in the defining Class, and MUST have the same value and flavor(s).

Property initialization consists of an optional list of preceding qualifiers, the name of the property and an optional value. Any Qualifiers specified for the Property MUST already be present in the Property definition from the defining Class, and MUST have the same value and flavor(s). Any property values not initialized have 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: CIM_Partition
{
                [key]
    string DriveLetter;
                [Units(“kilo bytes”)]
    sint32 RawCapacity = 128000;
                [write]
    string VolumeLabel;
                [Units(“kilo bytes”)]
    sint32 FreeSpace;
};
```

an instance of the above class might be declared as:

```
instance of Acme_LogicalDisk
{
    DriveLetter = "C";
    VolumeLabel = "myvol";
};
```

The resulting instance would take these 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 have their values initialized along with the properties in the subclass. Any property values not specifically assigned in the instance block will have 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 property values. See Section 4.11.6 on behavior when there is no property value initialization.

As described in item 21-E of Section 2.1, a class may have, by inheritance, more than one property with a particular name. If a property initialization has a property name which applies to more than one property in the class, the initialization applies to the property defined "closest" to the class of the instance. That is, the property can be located by starting at the class of the instance; if the class defines a property with the name from the initialization, then that property is initialized, otherwise, the search is repeated from the class' direct superclass. See also Appendix L for further discussion of the name conflict issue.

Instances of Associations may also be defined. For example:

```
instance of CIM_ServiceSAPDependency
{
    Dependent = "CIM_Service.Name = \"mail\"";
    Antecedent = "CIM_ServiceAccessPoint.Name = \"PostOffice\"";
};
```

4.8.1 Instance Aliasing

An alias can be assigned to an instance using this 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 Section4.12.2 Initializing References using Aliases.

4.8.2 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. Indexes start at 0 and have no gaps.

The current release of CIM does not support n-dimensional arrays.

Arrays of any basic data type are legal for properties. Arrays of references are not legal for properties. 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 this 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, this 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};
};
```

This 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[];
};
```

4.9 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. Method return types defined in MOF must be one of the data types described in Section 2.2. Return types cannot be arrays, but otherwise are unrestricted.

Note: Methods are expected, but not required, to return a status value indicating the result of execution of the method. Methods may use their parameters to pass arrays.

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 this 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 this example, a Configure method is defined on the Physical DiskDrive class. It takes a DiskPartitionConfiguration object reference 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);
};
```

4.10 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:

<table>
<tr><td>compiler Directive</td><td>Interpretation</td></tr>
<tr><td>#pragma include()</td><td>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.</td></tr>
<tr><td>#pragma instancelocale()</td><td>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.</td></tr>
<tr><td>#pragma locale()</td><td>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".

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.</td></tr>
<tr><td>#pragma namespace( )</td><td>This pragma is used to specify a Namespace path.</td></tr>
<tr><td>#pragma nonlocal()</td><td rowspan="4">These compiler directives, and the corresponding instance-level qualifiers, were removed as errata by CR1461.</td></tr>
<tr><td>#pragma nonlocaltype()</td></tr>
<tr><td>#pragma source()</td></tr>
<tr><td>#pragma sourcetype()</td></tr>
</table>

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

4.11 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 is found in Appendix A, MOF Syntax Grammar Description.

4.11.1 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 such as `\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 [10], 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.

4.11.2 Character Constants

Character and wide-character constants are specified as.

```
'a'
'\n'
'1'
'\x32'
```

Note: Forms such as octal escape sequences (e.g. '\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.

4.11.3 Integral Constants

Integer constants may be decimal, binary, octal or hexadecimal.

For example, these 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.

4.11.4 Floating-Point Constants

Floating point constants are declared as specified by IEEE in Ref. [6].

For example, these 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.

4.11.5 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=\"acme\",LogicalDisk.Drive=\"C\""
"//./root/default:NetworkCard=2"
```

An object reference can also be an alias. See Section 4.12.2 for more details.

4.11.6 NULL

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

4.12 Initializers

Initializers are used both in class declarations for default values and instance declarations to initialize a property to a value. The format of initializer values is specified in Section 2 and its subsections.

The initializer value **must** match the property data type. The only exceptions are the NULL value, which may be used for any data type, and integral values, used for characters.

4.12.1 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 of array elements is delimited with curly brackets.

For example, given this 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
};
```

this is a valid instance declaration:

```
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 4.8.2 for additional information on declaring arrays, and the distinctions between bags, ordered arrays and indexed arrays.

4.12.2 Initializing References Using Aliases

Aliases are symbolic references to an object located elsewhere in the MOF specification. They only have significance within the MOF specification 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.

Instances may be assigned an alias as described in Section 4.8.1. Aliases are identifiers which begin with the $ symbol. When a subsequent reference to that instance is required 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.

5 Naming

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.

1. CIM Naming addresses these requirements:

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 vs. SNMP objects requires the management platform to understand how the underlying objects are implemented

The KEY qualifier is the CIM Meta-Model 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.

5.1 Background

CIM MOF files can contain definitions of instances, classes or both, as illustrated in this diagram:

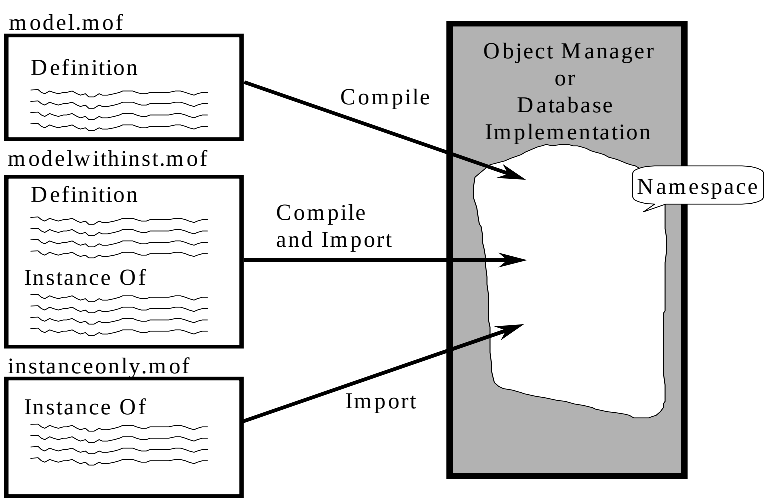


Figure 5-1 Definitions of instances and classes

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.

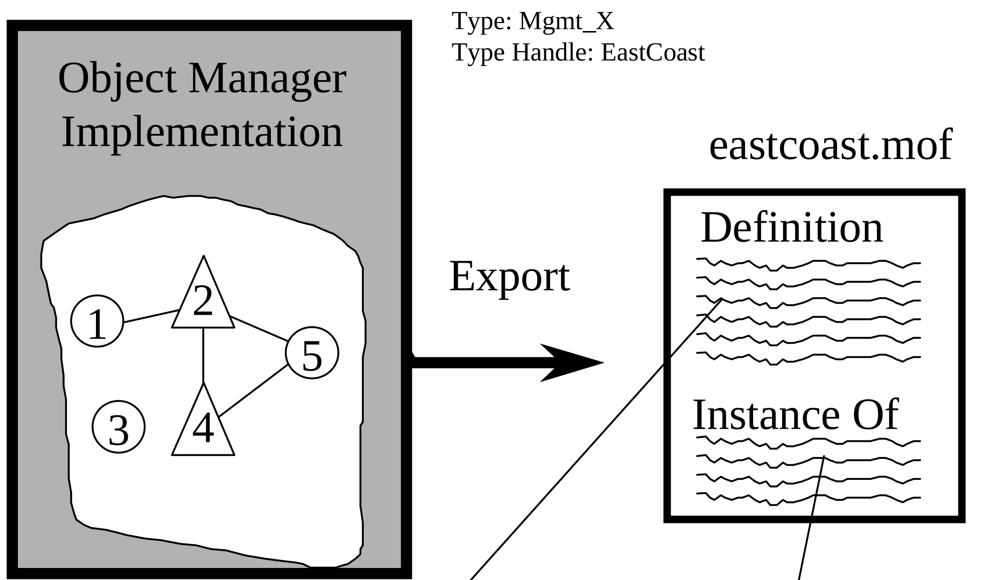
Namespaces can be used to:

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. 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";  };
```

1159

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.

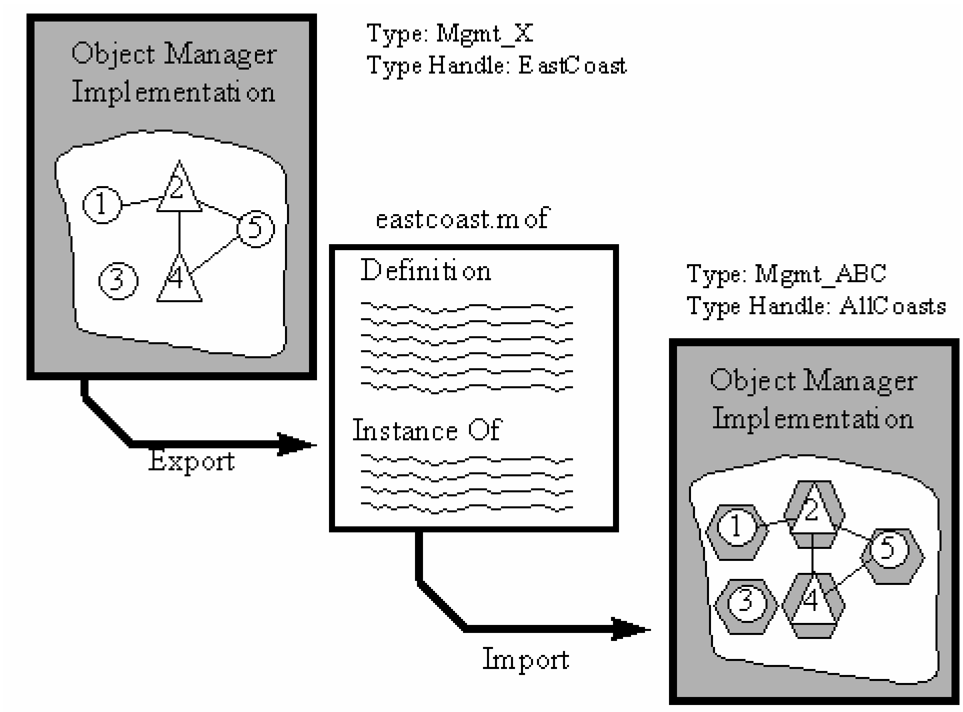


Figure 5-3 Information Exchange

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 the figure by wrapping the five instances in hexagons. The transformation process must maintain the original keys.

5.1.1 Management Tool Responsibility for an 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.

5.1.2 Management Tool Responsibility for an 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.

5.2 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.

This figure 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.

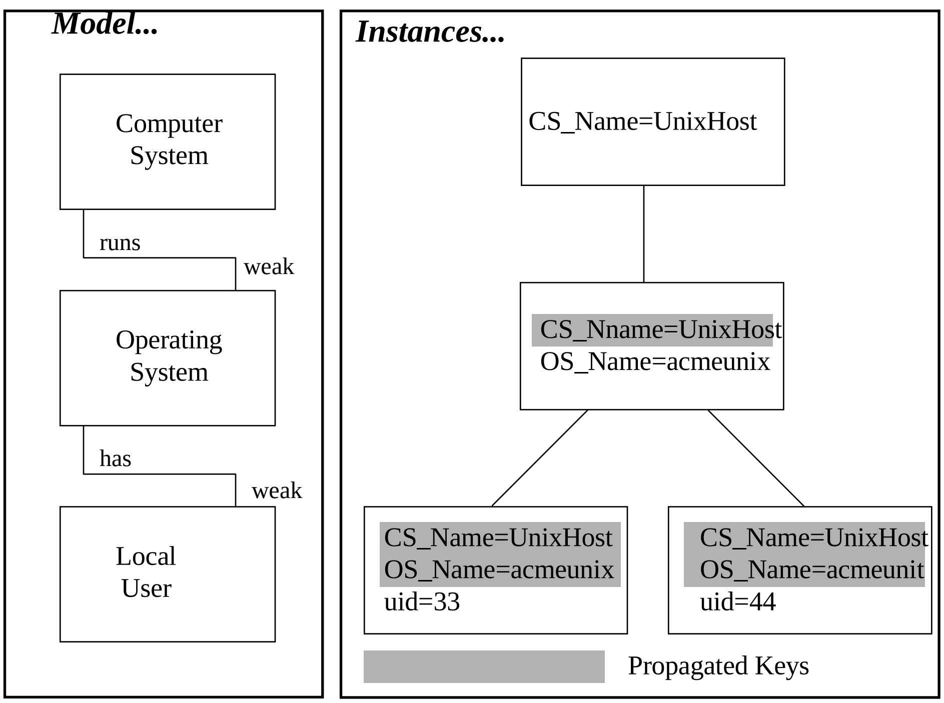


Figure 5-4 Example of Weak Association

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.

5.2.1 Referencing Weak Objects

A reference to an instance of an association includes the propagated keys. The properties must have the propagated qualifier that identifies which class the property originates in and what the name of the property is in that class – for example

```
instance of Acme_has
{
    anOS = "Acme_OS.Name=\"acmeunit\",SystemName=\"UnixHost\"";
    aUser = "Acme_User.uid=33,OSName=\"acmeunit\",SystemName=\"UnixHost\"";
};
```

The operating system being weak to system would be declared as:

```
Class Acme_OS
{
        [key]
    String Name;
        [key, Propagated("CIM_System.Name")]
    String SystemName;
};
```

The user class being weak to operating system would be declared as:

```
Class Acme_User
{
        [key]
    String uid;
        [key, Propagated("Acme_OS.Name")]
    String OSName;
        [key, Propagated("Acme_OS.SystemName")]
    String SystemName;
};
```

5.3 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 class name and the properties of the 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. The following figure shows the various components of an object name. These are described in more details in the following sections. See the objectName non-terminal in Appendix A for the formal description of object name syntax.

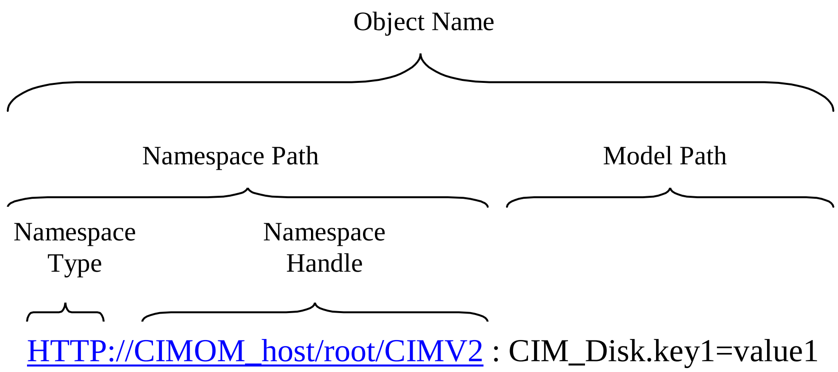


Figure 5-5 Object Naming

5.3.1 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.

5.3.1.1 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: (1) generating a MOF file for a particular scope of classes and associations, (2) importing a MOF file and (3) 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.

5.3.1.2 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.

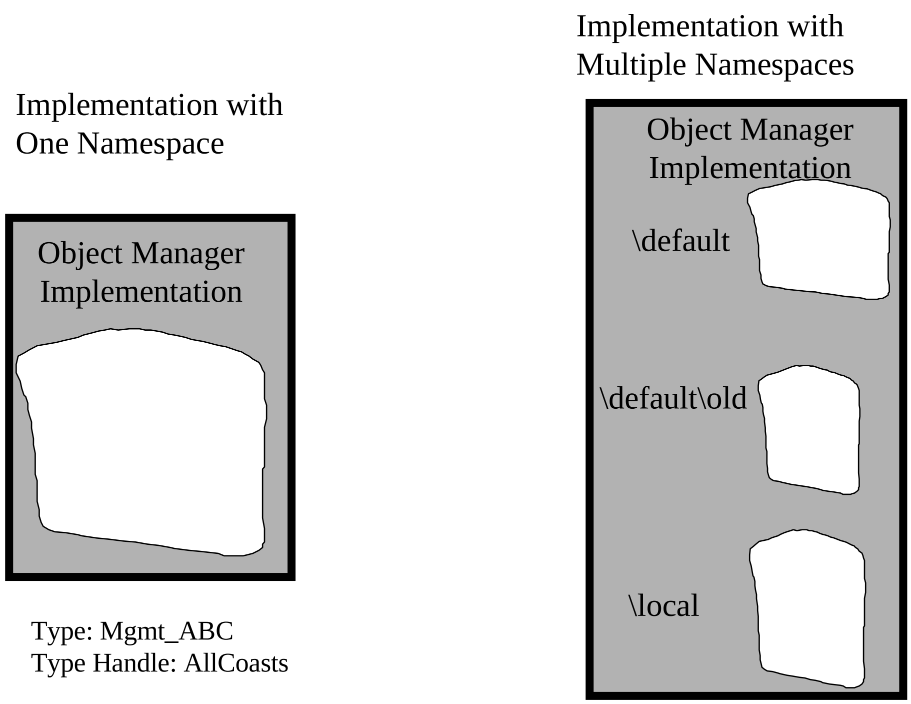


Figure 5-6 Namespaces

There are two important observations to make:

Namespaces can overlap with respect to their contents.

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.

5.3.2 Model Path

The object name constructed as a scoping path through the CIM schema is referred to as a Model Path. In the case of a model path for an instance, the model path is a combination of the key properties names and values qualified by the class name. 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 instance is a concatenation of named key property values, including all key values of its scoping objects.. When the class is weak with respect to another class, the model path includes all key properties from the scoping objects.

The formal syntax of Model Path is provided in Appendix A.

The syntax of Model Path is:

<className>.<key1>=<value1>[,<keyx>=<valuex>]*

5.3.3 Specifying the Object Name

There are various mechanisms for specifying the object name details for any class instance or any association reference in a MOF file.

The model path is specified for object and association differently. For objects (instances of classes), the model path is the combination of property value pairs that are marked with the KEY qualifier. So the model path for the following is: "ex_sampleClass.label1=9921,label2=8821". Since the order of the key properties is not significant, the model path could also be: "ex_sampleClass.label2=8821,label1=9921".

```
Class ex_sampleClass
{
        [key]
    uint32 label1;
        [key]
    string label2;
    uint32 size;
    uint32 weight;
};

instance of ex_sampleClass
{
    label1 = 9921;
    label2 = "SampleLabel";
    size = 80;
    weight = 45
};

instance of ex_sampleClass
{
    label1 = 0121;
    label2 = "Component";
    size = 80;
    weight = 45
};
```

For associations, a model path is used to specify the value of a reference in an INSTANCE OF statement for an association. In the following composedof-association example, the model path "ex_sampleClass.label1=9921,label2=8821" is used to reference an instance of the ex_sampleClass that is playing the role of a composer.

```
    [Association ]
Class ex_composedof
{
    [key] composer REF ex_sampleClass;
    [key] component REF ex_sampleClass;
};

instance of ex_composedof
{
    composer = "ex_sampleClass.label1=9921,label2=\"SampleLabel\"";
    component = "ex_sampleClass.label1=0121,label2=\"Component\"";
}
```

An object path for the ex_composedof instance would be (note the handling of double quote characters):

```
ex_composedof.composer="ex_sampleClass.label1=9921,label2=\"SampleLabel\"",
component="ex_sampleClass.label1=0121,label2=\"Component\""
```

Even in the unusual case of a reference to an association, the object name is formed the same way:

```
    [Association ]
Class ex_moreComposed
{
    composedof REF ex_composedof;
    . . .
};

instance of ex_moreComposed
{
    composedof =
"ex_composedof.composer=\"ex_sampleClass.label1=9921,label2=\\\"SampleLabel
\\\"\",component=\"ex_sampleClass.label1=0121,label2=\\\"Component\\\"\"";
    . . .
};
```

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

6 Mapping Existing Models Into CIM

Existing models have their own meta model and model. There are three types of mapping that can occur between meta schemas: technique, recast and domain. Each of these mappings can be applied when converting from MIF syntax to MOF syntax.

6.1 Technique Mapping

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

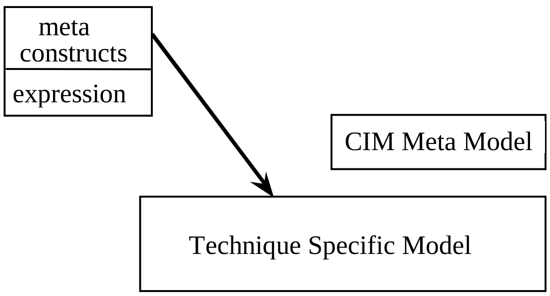


Figure 6-1 Technique Mapping Example

The DMTF uses the management information format (MIF) as the meta model to describe distributed management information in a common way. Therefore, it is meaningful to describe a technique mapping in which the CIM meta model 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 meta model as classes. In addition, associations are defined to document how these are combined. Figure 6-2 illustrates the results:

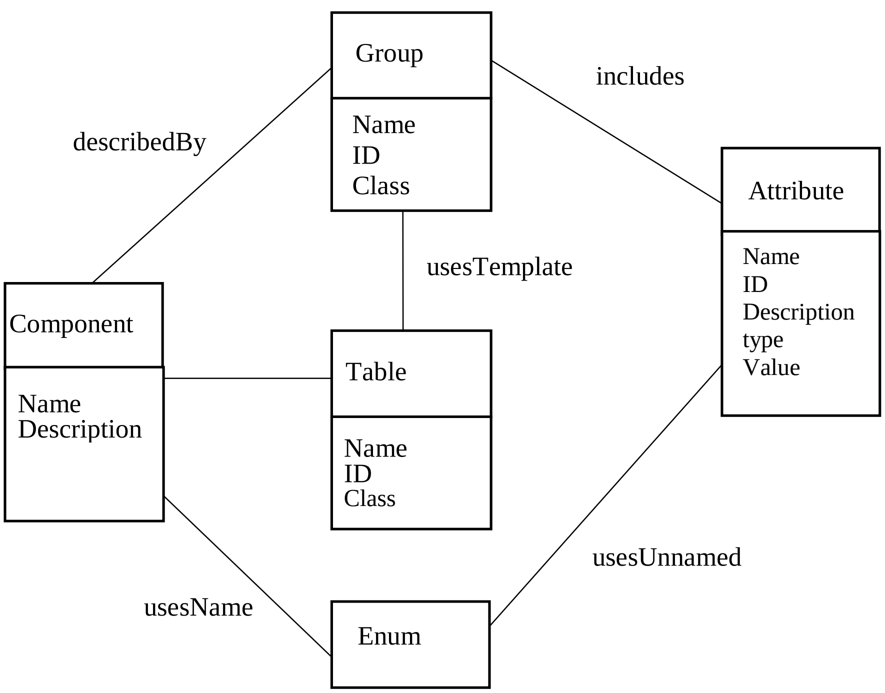


Figure 6-2 MIF Technique Mapping Example

6.2 Recast Mapping

A recast mapping provides a mapping of the sources' meta constructs into the targeted meta constructs, 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' meta model and the CIM meta model. Once this is done, the source expressions are recast.

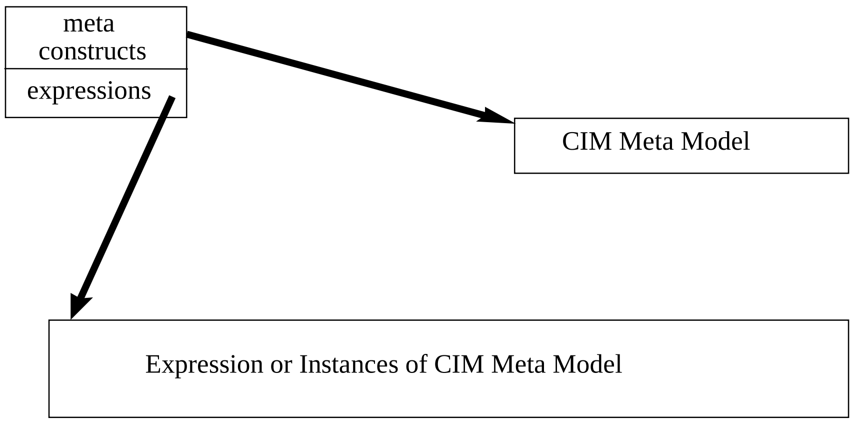


Figure 6-3 Recast mapping

This 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 or common models, 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 one case, the mapping is actually a re-expression of 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:

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"

6.4 Mapping Scratch Pads

In general, when the content of models are mapped between different meta schemas, information gets lost or is missing. To fill this gap, “scratch pads” are expressed in the CIM meta model using qualifiers, which are actually extensions to the meta model (for example, see section 2.5.5 Mapping MIF Attributes and section 2.5.6 Mapping Generic Data to CIM Properties). These scratch pads are critical to the exchange of core, common and extension model content with the various technologies used to build management applications.

7 Repository Perspective

This section 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 Section 6, the repository can be organized into the four partitions (meta, technique, recast and domain).

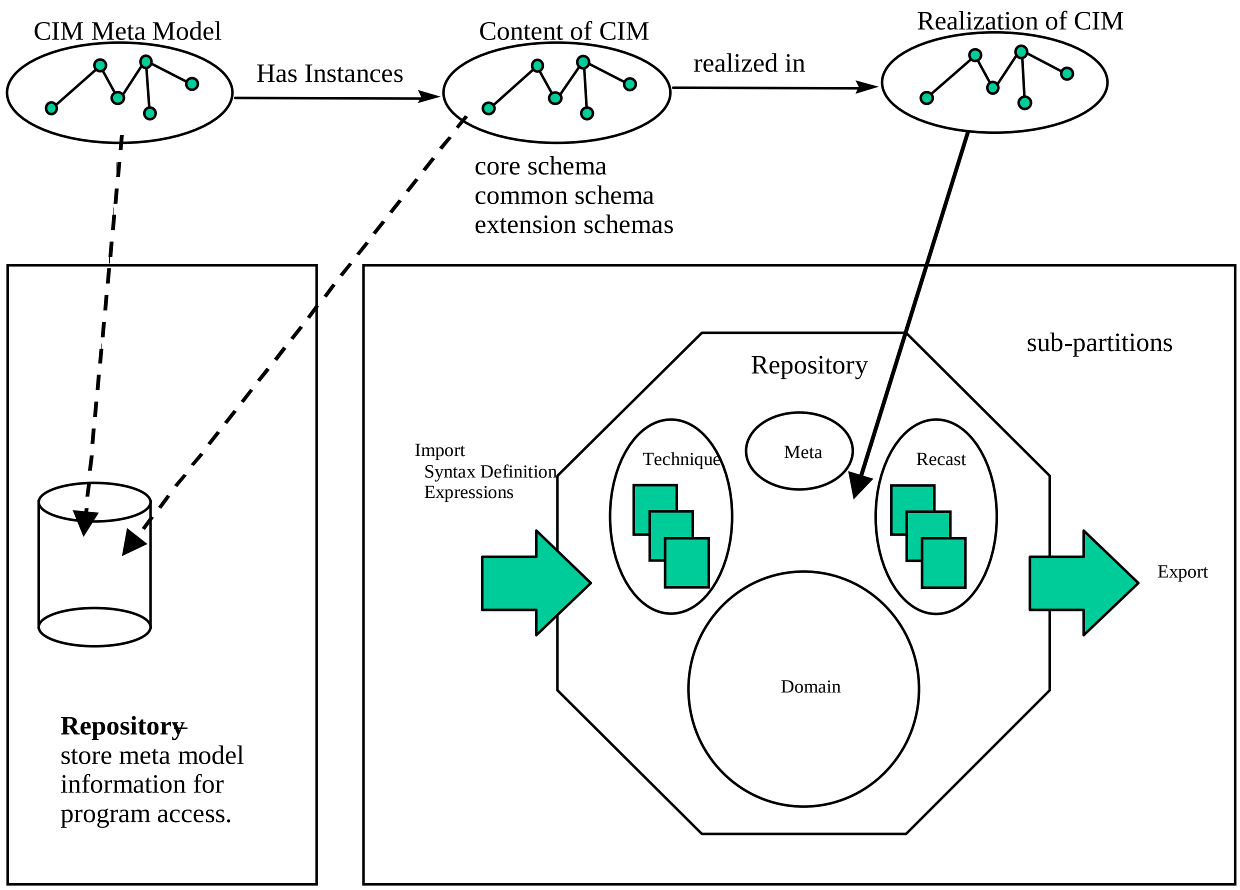


Figure 7-1 Repository Partitions

The repository partitions have the following characteristics:

- Each partition is discrete. The meta partition refers to the definitions of the CIM meta model. 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 and common models, 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 meta language encountered (that is, one for MIF, GDMO, SMI, and so on). In the re-cast partition, there would be a sub-partition for each meta language.

- The act of importing the content of an existing source can result in entries in the recast or domain partition.

7.1 DMTF MIF Mapping Strategies

Assume the meta-model 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:

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.

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.

To Domain Partition: Create a Domain Mapping for the content of a particular standard group that overlaps with the content of the CIM schema.

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.

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 must 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:

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.

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.

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).

From Domain Partition: Create a Domain Mapping for the content of the CIM schema that overlaps with the content of an existing management model

From Domain Partition: Create a Domain Mapping for the entire content of the CIM schema to an existing management model with the necessary extensions.

7.2 Recording Mapping Decisions

In order to understand the role of the scratch pad (see Section 6.4) 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 and 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.

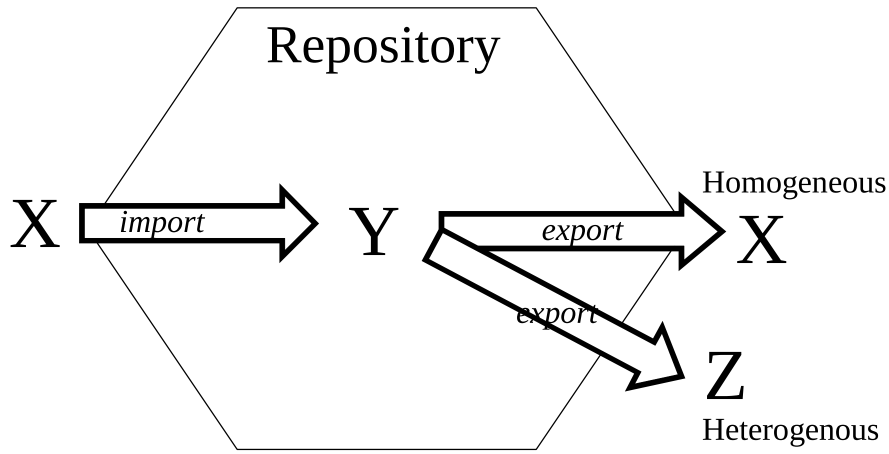


Figure 7-2 Homogeneous and Heterogeneous Export

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 (Figure 7-3).

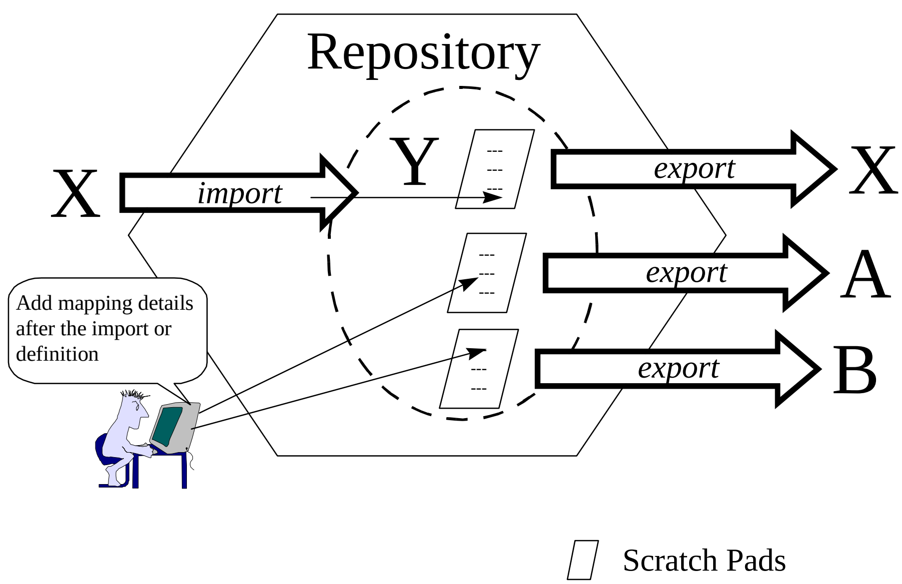


Figure 7-3 Scratch Pads and Mapping

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 meta model is used to describe the constructs used in the source meta schema. Therefore, by definition, there is one homogeneous mapping for each meta schema covered by the technique partition. These mappings create CIM objects 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 required for each export target and there can be multiple mapping algorithms for each target. The latter occurs because part of creating a recast mapping involves mapping the constructs of the source into CIM meta-model constructs. Therefore, for the MIF syntax, a mapping must be created for component, group, attribute, and so on, into appropriate CIM meta-model constructs like object, association, property, and so on. These mappings can be arbitrary. As a specific example, one of the decisions that must be made is whether a group or a component maps into an object. It would be possible to have two different recast mapping algorithms, one that mapped groups into objects with qualifiers that preserved the component, and one that 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 meta schema. 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 that 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 that capture 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 would 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.

Appendix A MOF Syntax Grammar Description

This section contains the grammar for MOF syntax. While the grammar presented here is convenient for describing the MOF syntax clearly, it should be noted that the same MOF language can also be described by a different, LL(1)-parseable, grammar. This has been done to allow low-footprint implementations of MOF compilers.

In addition, note these points:

1. An empty property list is equivalent to "*".
2. All keywords are case-insensitive.
3. 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 1 of Appendix F.
4. A string Value may contain quote (") characters, provided that each is immediately preceded by a backslash (\) character.
5. In the current release, the MOF BNF does not support initializing an array value to empty (i.e., an array with no elements). In the 3.0 version of this specification, it is the intention of the DMTF to extend the MOF BNF to support this functionality as follows:

   arrayInitialize = “{“ [arrayElementList] “}”

   arrayElementList = constantValue *(“,” constantValue)

   In order to ensure interoperability with V2.x implementations, the DMTF recommends that, where possible, the value of NULL rather than empty (i.e.,"{}") be used to represent the most common use cases. However, where this may result in confusion or other issues, implementations MAY use the above v3.0 syntax to initialize an empty array.

```
mofSpecification       =  *mofProduction

mofProduction          =  compilerDirective    |
                          classDeclaration     |
                          assocDeclaration     |
                          indicDeclaration     |
                          qualifierDeclaration |
                          instanceDeclaration

compilerDirective      =  PRAGMA pragmaName  "(" pragmaParameter ")"

pragmaName             =  IDENTIFIER

pragmaParameter        =  stringValue

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

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

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

indicDeclaration       =  "[" INDICATION *( "," qualifier ) "]"
                          CLASS className  [ superClass ]
                          "{" *classFeature "}" ";"

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 ]
parameterName          =  IDENTIFIER

array                  =  "[" [positiveDecimalValue] "]"

positiveDecimalValue   =  positiveDecimalDigit *decimalDigit

defaultValue           =  "=" initializer

initializer            =  ConstantValue | arrayInitializer | referenceInitializer

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

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

integerValue           =  binaryValue | octalValue | decimalValue | hexValue

referenceInitializer   =  objectHandle | aliasIdentifier

objectHandle           =  stringValue
                          // the(unescaped)contents of which must form an
                          // objectName; see examples

objectName                [ namespacePath ":" ]  modelPath

namespacePath             [ namespaceType "://" ] namespaceHandle

namespaceType             One or more UCS-2 characters NOT including the sequence
                          "://"

namespaceHandle        =  One or more UCS-2 character, possibly including ":"
                          // Note that modelPath may also contain ":" characters
                          // within quotes; some care is required to parse
                          // objectNames.

modelPath              =  className "." keyValuePairList
                          // Note: className alone represents a path to a class,
                          // rather than an instance

keyValuePairList       =  keyValuePair *( "," keyValuePair )
```

```
keyValuePair           =  ( propertyName "=" constantValue ) | ( referenceName "="
                          objectHandle )

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

qualifierName          =  IDENTIFIER

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

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

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

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

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

valueInitializer       =  [ qualifierList ]
                          ( propertyName | referenceName ) "=" initializer ";"
```

1626 These productions do not allow whitespace between the terms:

```
schemaName            =  IDENTIFIER
                         // Context:
                         // Schema name must not include "_" !
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*( """ *stringChar """ )

stringChar            =  "\" """ | // encoding for double-quote
                         "\" "\" | // encoding for backslash
                         any UCS-2 character but """ or "\"

booleanValue          =  TRUE | FALSE

nullValue             =  NULL
```

1627 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 META SCHEMA

```
// ==================================================================
//    NamedElement
// ==================================================================
        [Version("2.3.0"), Description(
        "The Meta_NamedElement class represents the root class for the "
        "Metaschema. It has one property: Name, which is inherited by all the "
        "non-association classes in the Metaschema. Every metaconstruct is "
        "expressed as a descendent of the class Meta_Named Element.") ]
class Meta_NamedElement
{
        [Description (
        "The Name property indicates the name of the current Metaschema element. "
        "The following rules apply to the Name property, depending on the "
        "creation type of the object:<UL><LI>Fully-qualified class names, such "
        "as those prefixed by the schema name, are unique within the schema."
        "<LI>Fully-qualified association and indication names are unique within "
        "the schema (implied by the fact that association and indication classes "
        "are subtypes of Meta_Class).<LI>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."
        "<LI>Explicitly-defined qualifier names are unique within the defining "
        "schema. An implicitly-defined qualifier must agree in type, scope and "
        "flavor with any explicitly-defined qualifier of the same name."
        "<LI>Trigger names must be unique within the property, class or method "
        "to which the trigger applies.<LI>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.<LI>Reference names "
        "must be unique within the scope of their defining association class. "
        "Reference names obey the same rules as property names.</UL><B>Note:</B> "
        "Reference names are not required to be unique within the scope of the "
        "related class. Within such a scope, the reference provides the name of "
        "the class within the context defined by the association.") ]
    string Name;
};

// ==================================================================
//    QualifierFlavor
// ==================================================================
        [Version("2.3.0"), Description (
        "The Meta_QualifierFlavor class encapsulates extra semantics attached "
        "to a qualifier such as the rules for transmission from superClass "
        "to subClass and whether or not the qualifier value may be translated "
        "into other languages") ]
class Meta_QualifierFlavor:Meta_NamedElement
{
};

// ==================================================================
//    Schema
// ==================================================================
        [Version("2.3.0"), Description (
        "The Meta_Schema class represents a group of classes with a single owner."
        " Schemas are used for administration and class naming. Class names must "
        "be unique within their owning schemas.") ]
class Meta_Schema:Meta_NamedElement
{
};
```

```
// ==================================================================
//    Trigger
// ==================================================================
        [Version("2.3.0"), Description (
        "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.") ]
class Meta_Trigger:Meta_NamedElement
{
};

// ==================================================================
//    Qualifier
// ==================================================================
        [Version("2.3.0"), Description (
        "The Meta_Qualifier class represents characteristics of named elements. "
        "For example, there are qualifiers that define the characteristics of a "
        "property or the key of a class. Qualifiers provide a mechanism that "
        "makes the Metaschema extensible in a limited and controlled fashion."
        "<P>It is possible to add new types of qualifiers 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.") ]
class Meta_Qualifier:Meta_NamedElement
{
        [Description ("The Value property indicates the value of the qualifier.")]
    string Value;
};

// ==================================================================
//    Method
// ==================================================================
        [Version( "2" ), Revision( "2" ), Description (
        "The Meta_Method class represents 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.") ]
class Meta_Method:Meta_NamedElement
{
};

// ==================================================================
//    Property
// ==================================================================
        [Version( "2" ), Revision( "2" ), Description (
        "The Meta_Property class represents 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,  return state and set "
        "state, respectively.") ]
class Meta_Property:Meta_NamedElement
{
};
```

```
// ==================================================================
//    Reference
// ==================================================================
        [Version( "2" ), Revision( "2" ), Description (
        "The Meta_Reference class represents (and defines) the role each object "
        "plays in an association. The reference represents the role name of a "
        "class in the context of an association, which supports the provision of "
        "multiple relationship instances for a given object. For example, a "
        "system can be related to many system components.") ]
class Meta_Reference:Meta_Property
{
};

// ==================================================================
//    Class
// ==================================================================
        [Version( "2" ), Revision( "2" ), Description (
        "The Meta_Class 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.<P>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.") ]
class Meta_Class:Meta_NamedElement
{
};

// ==================================================================
//    Indication
// ==================================================================
        [Version( "2" ), Revision( "2" ), Description (
        "The Meta_Indication class represents an object created as a result of a "
        "trigger. Because Indications are subtypes of Meta_Class, they can have "
        "properties and methods, and be arranged in a type hierarchy. ") ]
class Meta_Indication:Meta_Class
{
};

// ==================================================================
//    Association
// ==================================================================
        [Version( "2" ), Revision( "2" ), Description (
        "The Meta_Association class represents a class that contains two or more "
        "references and represents a relationship between two or more objects. "
        "Because of how associations are defined, it is possible to establish a "
        "relationship between classes without affecting any of the related "
        "classes.<P>For example, the 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 . Any subclass of "
        "Meta_Association is an association.") ]
class Meta_Association:Meta_Class
{
};
```

```
// ==================================================================
//    Characteristics
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Aggregation, Description (
        "The Meta_Characteristics class relates a Meta_NamedElement to a "
        "qualifier that characterizes the named element. Meta_NamedElement may "
        "have zero or more characteristics.") ]
class Meta_Characteristics
{
        [Description (
        "The Characteristic reference represents the qualifier that "
        "characterizes the named element.") ]
    Meta_Qualifier REF Characteristic;
        [Aggregate, Description (
        "The Characterized reference represents the named element that is being "
        "characterized.") ]
    Meta_NamedElement REF Characterized;
};

// ==================================================================
//    PropertyDomain
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Aggregation, Description (
        "The Meta_PropertyDomain class represents an association between a class "
        "and a property.<P>A property  has only one domain: the class that owns "
        "the property. A property can have an override relationship with another "
        "property from a different class. The domain of the overridden property "
        "must be a supertype of the domain of the overriding property.  The "
        "domain of a reference must be an association.") ]
class Meta_PropertyDomain
{
        [Description (
        "The Property reference represents the property that is owned by the "
        "class referenced by Domain.") ]
    Meta_Property REF Property;
        [Aggregate, Description (
        "The Domain reference represents the class that owns the property "
        "referenced by Property.") ]
    Meta_Class REF Domain;
};

// ==================================================================
//    MethodDomain
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Aggregation, Description (
        "The Meta_MethodDomain class represents an association between a class "
        "and a method.<P>A method has only one domain: the class that owns the "
        "method, which can have an override relationship with another method "
        "from a different class. The domain of the overridden method must be a "
        "supertype of the domain of the overriding method. The signature of the "
        "method (that is, the name, parameters and return type) must be "
        "identical.") ]
class Meta_MethodDomain
{
        [Description (
        "The Method reference represents the method that is owned by the class "
        "referenced by Domain.") ]
    Meta_Method REF Method;
        [Aggregate, Description (
        "The Domain reference represents the class that owns the method "
        "referenced by Method.") ]
    Meta_Class REF Domain;
};
```

```
// ==================================================================
//    ReferenceRange
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Description (
        "The Meta_ReferenceRange class defines the type of the reference.") ]
class Meta_ReferenceRange
{
        [Description (
        "The Reference reference represents the reference whose type is defined "
        "by Range.") ]
    Meta_Reference REF Reference;
        [Description (
        "The Range reference represents the class that defines the type of "
        "reference.") ]
    Meta_Class REF Range;
};

// ==================================================================
//    QualifiersFlavor
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Aggregation, Description (
        "The Meta_QualifiersFlavor class represents an association between a "
        "flavor and a qualifier.") ]
class Meta_QualifiersFlavor
{
        [Description (
        "The Flavor reference represents the qualifier flavor to "
        "be applied to Qualifier.") ]
    Meta_QualifierFlavor REF Flavor;
        [Aggregate, Description (
        "The Qualifier reference represents the qualifier to which "
        "Flavor applies.") ]
    Meta_Qualifier REF Qualifier;
};

// ==================================================================
//    SubtypeSupertype
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Description (
        "The Meta_SubtypeSupertype class represents subtype/supertype "
        "relationships between classes arranged in a generalization hierarchy. "
        "This generalization hierarchy is a rooted, directed graph and does not "
        "support multiple inheritance.") ]
class Meta_SubtypeSupertype
{
        [Description (
        "The SuperClass reference represents the class that is hierarchically "
        "immediately above the class referenced by SubClass.") ]
    Meta_Class REF SuperClass;
        [Description (
        "The SubClass reference represents the class that is the immediate "
        "descendent of the class referenced by SuperClass.") ]
    Meta_Class REF SubClass;
};
```

```
// ==================================================================
//    PropertyOverride
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Description (
        "The Meta_PropertyOverride class represents an association between two "
        "properties where one overrides the other.<P>Properties have reflexive "
        "associations that represent property overriding. A property can "
        "override an inherited property, which implies that any access to the "
        "inherited property will result in the invocation of the implementation "
        "of the overriding property. A Property can have an override "
        "relationship with another property from a different class.<P>The domain "
        "of the overridden property must be a supertype of the domain of the "
        "overriding property. The class referenced by the Meta_ReferenceRange "
        "association of an overriding reference must be the same as, or a "
        "subtype of, the class referenced by the Meta_ReferenceRange "
        "associations of the reference being overridden.") ]
class Meta_PropertyOverride
{
        [Description (
        "The OverridingProperty reference represents the property that overrides "
        "the property referenced by OverriddenProperty.") ]
    Meta_Property REF OverridingProperty;
        [Description (
        "The OverriddenProperty reference represents the property that is "
        "overridden by the property reference by OverridingProperty.") ]
    Meta_Property REF OverriddenProperty;
};

// ==================================================================
//    MethodOverride
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Description (
        "The Meta_MethodOverride class represents an association between two "
        "methods, where one overrides the other. Methods have reflexive "
        "associations that represent 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.") ]
class Meta_MethodOverride
{
        [Description (
        "The OverridingMethod reference represents the method that overrides the "
        "method referenced by OverriddenMethod.") ]
    Meta_Method REF OverridingMethod;
        [Description (
        "The OverriddenMethod reference represents the method that is overridden "
        "by the method reference by OverridingMethod.") ]
    Meta_Method REF OverriddenMethod;
};
```

```
// ==================================================================
//    ElementSchema
// ==================================================================
        [Association, Version( "2" ), Revision( "2" ), Aggregation, Description (
        "The Meta_ElementSchema class represents the elements (typically classes "
        "and qualifiers) that make up a schema.") ]
class Meta_ElementSchema
{
        [Description (
        "The Element reference represents the named element that belongs to the "
        "schema referenced by Schema.") ]
    Meta_NamedElement REF Element;
        [Aggregate, Description (
        "The Schema reference represents the schema to which the named element "
        "referenced by Element belongs.") ]
    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, KiloBits, MegaBits, GigaBits> 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 Temperature

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, Revolutions per Second

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, Tenths of Decibels

Grays, Sieverts

MilliWatts

DBm

<Bytes, KiloBytes, MegaBytes, GigaBytes> per Second

BTU per Hour

PCI clock cycles

<Numeric value> <Minutes, Seconds, Tenths of Seconds, Hundreths of Seconds, MicroSeconds, MilliSeconds, Nanoseconds>

Us[3]

Amps at <Numeric Value> Volts

Clock Ticks

Packets

[3] Standard Rack Measurement equal to 1.75 inches.

Appendix D UML Notation

The CIM meta-schema 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, which 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 the segment 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; 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

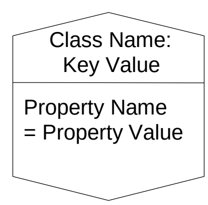
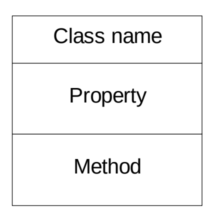
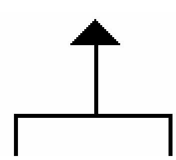
“0…1” indicates an optional single-valued reference

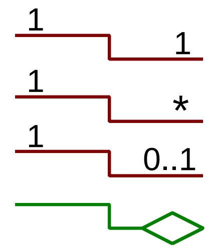
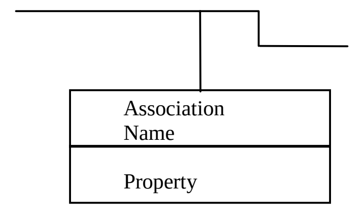
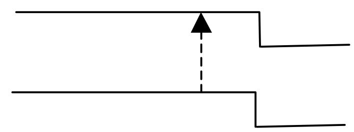
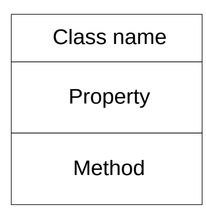
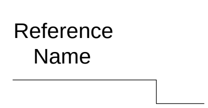
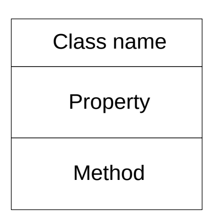
“*” indicates an optional many-valued reference (as does “0..*”)

“1..*” indicates a required many-valued reference

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

The diagramming notation and its interpretation are summarized in this table:

Meta element	Interpretation	Diagramming Notation
Object		
Primitive type	Text to the right of the colon in the center portion of the class icon	
Class		
Subclass		

Meta element	Interpretation	Diagramming Notation
Association	1:1 1:Many 1:zero or 1 Aggregation	
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.	

Meta element	Interpretation	Diagramming Notation
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.	

2041 # Appendix E Glossary

Aggregation	A strong form of an *association*. For example, the containment relationship between a system and the components that make up the system can be called an *aggregation*. 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 model*, *Common model* and *extended schemas*.
CIM Schema	The schema representing the *Core* and *Common models*. Versions of this schema will be released by the DMTF over time as the schema evolves.
Class	A collection of instances, all of which support a common 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*.
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 i*mplicit 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 model. An example is the Win32 *schema*.
Feature	A *property* or *method* belonging to a *class*.
Flavor	Part of a *qualifier* specification indicating overriding and *inheritance* rules. For example, the *qualifier* KEY has Flavor(DisableOverride ToSubclass), meaning that every subclass must inherit it and cannot override it.
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 result 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	One or more qualified class properties that can be used to construct a name. One or more qualified object properties which uniquely identify instances of this object

	in a namespace.
Managed Object	The actual item in the system environment that is accessed by the *provider*. For example, a Network Interface Card.
Meta model	A set of *classes, associations* and *properties* that expresses the types of things that can be defined in a *Schema*. For example, the *meta model* 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.
Meta schema	The schema of the meta model.
Method	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.
Model	A set of *classes, properties* and *associations* that allows the expression of information about a 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.
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.
Name	Combination of a Namespace path and a Model path that identifies a unique object.
Trigger	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*.
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 availability as a string, and may return the values “powersave”, "on" or "off." The Modem *subclass* of Device may redefine (override) availability by returning "on," "off," but not "powersave". If all Devices are enumerated, any Device that happens to be a modem will not return the value "powersave" for the availability *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 *meta schema*. For example, if a property has the qualifier KEY with the value TRUE, the property is a key for the class.
Reference	Special *property types* that are references or "pointers" to other instances.
Schema	A management schema is provided to establish a common conceptual framework at the level of a fundamental topology-both with respect to classification and association, and with respect to a basic set of classes intended to establish a common framework for a description of the managed environment. A *Schema* is 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 with which meta constructs the *Qualifier* can be used. 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*.
Subclass	See Inheritance.
Superclass	See Inheritance.
Top Level Object	A class or object that has no scoping object.

Appendix F Unicode Usage

All punctuation symbols 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 so on. 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:

Initial identifier characters must be in set S1, where S1 = {U+005F, U+0041...U+005A, U+0061...U+007A, U+0080...U+FFEF) [This is alphabetic, plus underscore]

All following characters must be in set S2 where S2 = S1 union {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, published by the Internet Engineering Task Force (IETF), as an example of a Universal Transformation Format that has specific characteristics for dealing with multi-octet characters on an application-specific basis.

F.1 MOF Text

MOF files using Unicode must 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).

U+FFFE is little endian.

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

F.2 Quoted Strings

In all cases where non-identifier string values are required, delimiters must surround them.

The supported delimiter for strings is U+0027. Once a quoted string is started using the delimiter, the same delimiter, U+0027, 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 G Guidelines

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.

Leading underscores in identifiers are to be discouraged and not be used at all in the standard schemas.

As a general rule, it is recommended that class names 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.

G.1 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 the current release, CIM does not support octet strings as a separate data type. To map a single octet string (i.e., octets of binary data), 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. The OCTETSTRING qualifier may be used to indicate that the uint8 array conforms to this encoding.

In the case where an array of octet strings must be mapped, since arrays of uint8 arrays are not supported, a textual convention encoding the binary information as hexadecimal digit characters (i.e., 0x<&lt0-9,A-F>&lt0-9,A-F>>*) is used for each of the octet strings in the array. The number of octets in the octet string is encoded in the first 8 hexadecimal digits of the string with the most significant digits in the left- most characters of the string. The length count octets are included in the length count (i.e., "0x00000004" is the encoding of a 0- length octet string.

The OCTETSTRING qualifier is used to qualify the string array.

Example usages of the OCTETSTRING qualifier on a property follows:

```
[Description ("An octet string"), Octetstring]
uint8 Foo[];
[Description ("An array of octet strings"), Octetstring]
String Bar[];
```

G.2 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:

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	

2114 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			

2115 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		

2116 sql3part4.txt (ANNEX E):

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

2117

Appendix H Embedded Object Qualifier

The EmbeddedObject qualifier is used motivated by the need to include a specific instance's data in an indication (event notification) or to capture the contents of an instance at a point in time (for example, to include the CIM_DiagnosticSetting properties that dictated a particular CIM_DiagnosticResult, in the Result object).

To address this need, it is expected that the next major version of the CIM Specification will include a separate data type for representing instances (or snapshots of instances) directly. Until that time, the EmbeddedObject qualifier can be used to achieve an approximately equivalent effect by permitting a CIM object manager (or other entity) to simulate embedded instances by encoding them as strings when they are presented externally. Clients which aren't concerned with handling embedded objects may treat properties with this qualifier as they would any other string-valued property, while clients that wish to realize the capability of embedded objects can extract the embedded object information by decoding the presented string value.

In order to reduce the parsing burden on embedded object-aware consumers of CIM data, the encoding by which the embedded object is represented in the string value depends on the protocol or representation being used for transmission of the containing instance. This makes the string value appear to vary depending on the circumstances in which it is observed; this is an acknowledged weakness of using a qualifier instead of a new data type.

The DMTF defines only two representations by which CIM data are made available from a CIM object manager, although it allows for other representations and protocols. The two defined representations are MOF and CIM-XML; accordingly, the DMTF defines EmbeddedObject string formats only for these two representations. When other protocols or representations are used, they MUST include particulars on the encoding to be used for EmbeddedObject-qualified string values if that protocol or representation will be used to communicate with embedded object-aware consumers of CIM data.

H.1 Encoding for MOF

When CIM data including an EmbeddedObject-qualified string value are rendered in MOF by an embedded object-aware entity (e.g., CIM object manager or client), the embedded object MUST be encoded into string form following the MOF syntax for the instanceDeclaration nonterminal in the case of embedded instances, or for the classDeclaration, assocDeclaration, or indicDeclaration nonterminals, as appropriate, in the case of embedded classes (see Appendix A).

Examples:

```
Instance of CIM_InstCreation {
  EventTime = "20000208165854.457000-360";
  SourceInstance =
    "Instance of CIM_FAN {"
    "DeviceID = \"Fan 1\";"
    "Status = \"Degraded\";"
    "};";
};

Instance of CIM_ClassCreation {
  EventTime = "20031120165854.457000-360";
  ClassDefinition =
    "class CIM_Fan : CIM_CoolingDevice {"
   " boolean VariableSpeed;"
   "   [Units (\"Revolutions per Minute\") ]"
   "uint64 DesiredSpeed;"
   "};"
};
```

H.2 Encoding for CIM-XML

When CIM data including an EmbeddedObject-qualified string value are rendered in CIM-XML by an embedded object-aware entity, the embedded object MUST be encoded into string form as either an INSTANCE element (in the case of instances) or a CLASS element (in the case of classes), as defined in the DMTF CIM-XML specification (Representation of CIM in XML, DSP0201).

Appendix I Schema Errata

It is expected that the CIM Schema (based on the constructs and thinking laid out in this Specification) will evolve. Evolution occurs for three reasons:

To add new classes, associations, qualifiers, properties and/or methods
To correct errors in the "Final Release" versions of the Schema
To deprecate and update the model (labeling classes, associations, qualifiers, etc. as "not recommended for future development" and replacing them with "new" constructs)

#1 is easily accommodated and addressed in Section 2.3 of this Specification.
#3 is addressed by the Deprecated qualifier explained in Section 2.5 of this Specification.
#2 is an Errata to the CIM Schemas, after their "Final Release." Some examples of errata are:
Incorrectly or incompletely defined keys (an array defined as a key property, or incompletely specified propagated keys)
Invalid subclassing (subclassing an optional association from a weak relationship i.e., a mandatory association, subclassing a nonassociation class from an association, or subclassing an association but having different reference names resulting in 3 or more references on an association)
Class references reversed as defined by an association's roles (Antecedent/Dependent references reversed)
Use of SQL reserved words as property names
Violation of semantics (Missing Min(1) on a Weak relationship, contradicting that a Weak relationship is mandatory)

Errata are a serious matter since the Schema should be correct, but existing implementations must be taken into account. Therefore, the DMTF will implement the following process (in addition to the normal release process) with respect to any Schema errata:

(a) As soon as an error is found, it should be raised to the Technical Committee (technical@dmtf.org) for review. Suggestions for correcting the error should also be described, if possible.

(b) The findings of the Technical Committee will be documented in an email to the submitter, within 21 days. These findings will describe the Committee's decision regarding whether the submission is a valid Errata, the reasoning behind the decision, the recommended strategy to correct the error, and whether backward compatibility is possible.

(c) If the error is valid, an email will be sent simultaneously (with the reply to the submitter) to all DMTF members (members@dmtf.org). The email will highlight the error, the findings of the Technical Committee and the strategy to correct the error. In addition, the Committee will indicate what versions of the Schema will be affected i.e., only the latest, or all schemas after a specific version.

(d) All members are invited to respond to the Technical Committee -- within 30 days regarding impacts to their implementations due to the "correction strategy." Impacts should be explained as thoroughly as possible, as well as alternate strategies to correct the error.

(e) If one or more members are affected, then the Technical Committee will evaluate all proposed, alternate correction strategies. It will choose one of three options:

To stay with the correction strategy proposed in (b)
To move to one of the proposed alternate strategies
To define a new correction strategy based on the evaluation of member impacts.

(f) If an "alternate" strategy is proposed in (e), the Technical Committee may decide to reenter the Errata process, resuming with Item (c) -- an email to all DMTF members regarding the alternate correction strategy. However, if the Technical Committee believes that further comment will not raise any new issues, then the outcome of Item (e) will be declared "final."

(g) If a "final" strategy is decided, this strategy will be implemented via a Change Request to the affected Schema(s). The Change Request will be written and issued by the Technical Committee. Affected Models and MOF will be updated, and their introductory comment section flagged to indicate that a correction for an ERRATA was applied.

Appendix J References

[1] Grady Booch and James Rumbaugh, *Unified Method for Object-Oriented Development Document Set*, Rational Software Corporation, 1996, http://www.rational.com/uml

[2] HyperMedia Management Protocol, Protocol Encoding, draft-hmmp-encoding-03.txt, February, 1997

[3] *Interface Definition Language*, DCE/RPC, The Open Group.

[4] Georges Gardarin and Patrick Valduriez, *Relational Databases and Knowledge Bases*, Addison Wesley, 1989

[5] Coplein, James O., Schmidt, Douglas C (eds). *Pattern Languages of Program Design*, Addison-Wesley, Reading Mass., 1995

[6] IEEE Standard for Binary Floating-Point Arithmetic, *ANSI/IEEE Standard 754-1985*, Institute of Electrical and Electronics Engineers, August 1985.

[7] Augmented BNF for Syntax Specifications: ABNF, RFC 2234, Nov 1997

[8] G. Weinberger, *General Systems Theory*

[9] *The Unicode Standard*, Version 2.0, by The Unicode Consortium, Addison-Wesley, 1996.

[10] Universal Multiple-Octet Coded Character Set, ISO/IEC 10646

[11] UCS Transformation Format 8 (UTF-8), ISO/IEC 10646-1:1993 Amendment 2 (1996)

[12] Code for the Representation of Names of Languages, ISO/IEC 639:1988 (E/F)

[13] Code for the Representation of Names of Territory, ISO/IEC 3166:1988 (E/F)

Appendix K Change History

Version 1	April 09, 1997	First Public Release
Version 1.1	October 23, 1997	Output after Working Groups input
Version 1.2a	November 03, 1997	Naming
Version 1.2b	November 17, 1997	Remove reference qualifier
Version 2.0a	December 11, 1997	Apply pending changes and new metaschema
Version 2.0d	December 11, 1997	Output of 12/9/1997 TDC, Dallas
Version 2.0f	February 16, 1998	Output of 2/3/1998 TDC, Austin
Version 2.0g	February 26, 1998	Apply approved change requests and final edits submitted through 2/26/1998.
Version 2.2	June 14, 1999	Incorporate Errata and approved change requests through 1999-06-08
Version 2.2.1	June 07, 2003	Incorporate Addenda 1
Version 2.2.2	June 07, 2003	Incorporate Addenda 2
Version 2.2.1000	June 7, 2003	00638-Replace Section 2.3.1 of the CIM Spec 00664-Rules for Versioning CIM Classes 00665-Remove Participants Section 00685-Add Units in the CIM Spec App C 00707-New “Composition” Qualifier 00713-Corections to Qualifiers 00714-Clarification of the BitMap Qualifier 00715-Clarification of the Deprecated qualifier 00727-Required Qualifier for Identification 00729-ALIAS qualifier from Standard to Optional 00731-New Units value - Packets 00762-Correction to Qualifiers 00813-Clarify Semantics of the Write Qualifier 00814-Add syntax to MOF BNF to allow the specification of an empty array 00817-Errata Class Alias Support 00881-Clarify Model Path 00910-New Exception Qualifier 01062-Updates to Experimental and Version 01069-Clarification of ranges in ValueMaps
Version 2.3 Draft	November 05, 2003 Lawrence J. Lamers	00716-New MinLen qualifier
Version 2.3 Draft	November 12, 2003	Updated Appendix K - fixed Table of Contents formating.
Version 2.3 Draft	January 8, 2004	ARCH00001 - Add RFC2119 terminology to MOF files ARCH00002 CR1133 - Overriding (non-reference) property type ARCH00004 CR1169 - Instances don't have qualifiers ARCH00005 CR1148 - Cleanups for Standard Qualifiers ARCH00006 CR1149 - Datetime cleanup ARCH00008 CR1158 - Parameters can be arrays of references ARCH00009 CR1153 - Formal syntax reference ARCH00010 CR1155 - Instances don't define properties or methods ARCH00011 CR1156 - MODELCORRESPONDENCE syntax ARCH00012 CR1158 - NULLVALUE qualifier syntax and wording ARCH00013 CR1159 - MAXVALUE qualifier wording

		ARCH00016 CR1172 - ModelPath syntax
Version 2.3 Draft	February 10, 2004	ARCH00015 CR1167 -Correct DisableOverride flavor on OVERRIDE qualifier ARCH00017 CR1168 - Clarify definitions of Qualifier flavors ARCH00021 CR1237 - Identifying properties
Version 2.3 Draft	July 6, 2004	ARCH00019 CR1174.003 EMBEDDEDOBJECT qualifier.
Version 2.3 Draft	July 7, 2004	ARCH00020.004 CR1461.000 Deprecate use of SOURCE, SOURCETYPE, NONLOCAL,NONLOCALTYPE qualifiers and pragmas. ARCH00022.001 CR1278.000 Model Correspondence Scope ARCH00027.000 CR1289.001 Change name of DSP0004 to CIM Infrastructure Specification ARCH00029.004 CR1390.001 EMBEDDED INSTANCE qualifier ARCH00031.002 INT type; MINVALUE and MAXVALUE usage ARCH00032.001 Usage Rules for DN string arrays
Version 2.3 Draft	October 4, 2005	ARCHCR0048 Remove XML of embedded object example ARCHCR0051 Starting numbers for datetime fields ARCHCR0052 OCL qualifier ARCHCR0059 Remove mention of "instance" with qualifiers ARCHCR0060 Remove inappropriate mention of "alias" ARCHCR0061 Remove incorrect #pragma namespace syntax rule ARCHCR0062 Remove moot qualifier/pragma conflict rule ARCHCR0063 Explain struck-through text for compiler directives ARCHCR0064 Remove "Synchronizing Namespaces" section ARCHCR0065 Remove qualifier defs and Revision qual from Appendix B

Appendix L Ambiguous Property and Method Names

Item 21-E of Section 2.1 explicitly allows a subclass to define a property which may have the same name as a property defined by a superclass, and for that new property NOT to override the superclass' property. (The subclass may override the superclass' property by attaching an OVERRIDE qualifier; this situation is well-behaved, and not part of the problem under discussion.)

Similarly, a subclass may define a method with the same name as a method defined by a superclass, without overriding the superclass' method. This appendix will refer only to properties in what follows, but it's to be understood that the issues regarding methods are essentially the same, and for any statement about properties, a similar statement about methods can be inferred.

This same-name capability is provided to allow the superclass to be enhanced or extended in a minor schema change by one group (the DMTF, in particular), without coordination with, or even knowledge of, the development of the subclass in another schema by another group. That is, a subclass that has been defined using one version of the superclass should not become invalid if a subsequent version of the superclass introduces a new property whose name is the same as a property defined on the subclass. Any other use of the same-name capability is strongly discouraged, and additional constraints on allowable cases may well be added in future versions of CIM.

It's natural for CIM applications to be written under the assumption that property names alone suffice to uniquely identify properties. However, applications written this way are at risk of failure if they refer to properties from a subclass whose superclass has been modified to include a new property, and the new property has the same name as a previously-existing property defined by the subclass. As an example, suppose we have:

```
 [abstract]
class CIM_Superclass
{
};

class VENDOR_Subclass
{
        string      Foo;
};
```

Suppose further that there's just one instance of VENDOR_Subclass; a call to enumerateInstances("VENDOR_Subclass") might produce the following XML result from the CIMOM (assuming it didn't bother to ask for CLASSORIGIN information):

```
<INSTANCE CLASSNAME="VENDOR_Subclass">
        <PROPERTY NAME="Foo" TYPE="string">
                <VALUE>Hello, my name is Foo</VALUE>
        </PROPERTY>
</INSTANCE>
```

If the definition of CIM_Superclass changes to:

```
 [abstract]
class CIM_Superclass
{
        string foo = "You lose!";
};
```

then the enumerateInstances call might return:

```
<INSTANCE>
        <PROPERTY NAME="Foo" TYPE="string">
                <VALUE>You lose!</VALUE>
        </PROPERTY>
```

```
        <PROPERTY NAME="Foo" TYPE="string">
                <VALUE>Hello, my name is Foo</VALUE>
        </PROPERTY>
</INSTANCE>
```

If the client application attempts to retrieve the 'foo' property, which value it obtains (if it doesn't experience an error) is dependent on the implementation, if not the phase of the moon.

While a class may define a property with the same name as an inherited property, it is not permitted in and of itself to define two (or more) properties with the same name. This means that the combination of defining class plus property name uniquely identifies a property. (Most CIM operations that return instances have a flag controlling whether or not to include the originClass for each property: see e.g. DSP0200, Section 2.3.2.11, enumerateInstances, and DSP0201, Section 3.1.4, ClassOrigin.) However, the use of class-plus-property-name for identifying properties makes an application vulnerable to failure if a property is "promoted" up to a superclass in a subsequent schema release. As a concrete example, consider:

```
class CIM_Top
{
};

class CIM_Middle : CIM_Top
{
        uint32    foo;
};

class VENDOR_Bottom : CIM_Middle
{
        string    foo;
};
```

An application that identifies the uint32 property as "the property named 'foo' defined by CIM_Middle" will no longer work if a subsequent release of the CIM schema changes the hierarchy to:

```
class CIM_Top
{
        uint32    foo;
};

class CIM_Middle : CIM_Top
{
};

class VENDOR_Bottom : CIM_Middle
{
        string    foo;
};
```

Strictly speaking, there is no longer a "property named 'foo' defined by CIM_Middle"; it's now defined by CIM_Top and merely inherited by CIM_Middle, just as it's inherited by VENDOR_Bottom. An instance of VENDOR_Bottom returned in XML from a CIMOM might look like this:

```
<INSTANCE CLASSNAME="VENDOR_Bottom">
        <PROPERTY NAME="Foo" TYPE="string" CLASSORIGIN="VENDOR_Bottom">
                <VALUE>Hello, my name is Foo!</VALUE>
        </PROPERTY>
        <PROPERTY NAME="Foo" TYPE="uint32" CLASSORIGIN="CIM_Top">
```

```
                <VALUE>47</VALUE>
        </PROPERTY>
</INSTANCE>
```

A client application looking for a PROPERTY element with NAME="Foo" and CLASSORIGIN="CIM_Middle" will fail with this XML fragment.

Although CIM_Middle no longer defines a 'foo' property directly in the example above, our intuition says that we should be able to point to the CIM_Middle class and locate the 'foo' property that's defined in its nearest superclass. In the general case, the application must be prepared to perform this search, separately obtaining information from the server when necessary about the (current) class hierarchy and implementing an algorithm to select the appropriate property information from the instance information returned from a server operation.

Although it's technically allowed, schema writers SHOULD NOT introduce properties that cause name collisions within the schema, and are strongly discouraged from introducing properties whose names are known to conflict with property names of any subclass or superclass in another schema.