White Paper **DSP0111**

Status: Final

Common Information Model (CIM) Core Model

Version 2.4

August 30, 2000

Abstract

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

A management model is provided to establish a common conceptual framework for a description of the managed environment. A fundamental taxonomy of objects is defined — both with respect to classification and association, and with respect to a basic set of classes intended to establish a common framework.

The management model is divided into the following conceptual layers:

- Core Model—an information model that captures notions applicable to all domains of management
- Common Models—information models that capture notions common to particular management domains but independent of a particular technology or implementation. The common domains include Systems, Applications, Devices, Users, Networks, Policies and Databases.
- Extension Models—represent technology-specific extensions of the Common Models. These models are specific to environments, such as operating systems, or to vendors.

This document describes the DMTF CIM Core Model.

Change History

Version 0.1	December 10, 1997	Technical Committee Input, V1
Version 0.9 (Draft)	June 23, 1998	Rewritten for V2
Version 0.91 (Draft)	July 13, 1998	Minor changes addressing comments. Added Appendix of Referenced Documents.
Version 2.0 (Releas e)	August 5, 1998	Approved by Technical Committee
Versions 2.1-2.3 (Skipped)		Skipped to keep numbering in sync with Schema versioning.
Version 2.4 (Draft)	August 30, 2000	Added text to clarify keys, naming and MOF syntax. Updated many of the existing object explanations and modeling discussions. Addressed new objects in CIM Versions 2.1 to 2.4.

Editor

Andrea Westerinen
Cisco Systems

John Strassner
Cisco Systems

For the DMTF Technical Committee and System/Devices Working Group

Table of Contents

Table of Figures

1. Introduction

Before jumping into the many details of the Core Model, it might be wise to discuss the relevancy of CIM, the Common Information Model.

CIM is an information model, a conceptual view of the managed environment, that attempts to unify and extend the existing instrumentation and management standards (SNMP, DMI, CMIP, etc.) using object-oriented constructs and design. Note that the word, "unify," is used in the preceding sentence, not the word, "replace." CIM does not require any particular instrumentation or repository format. It is only an information model – unifying the data, using an object-oriented format, made available from any number of sources.

The value of CIM stems from its object orientation. Object design provides support for the following capabilities, that other "flat" data formats do not allow:

- Abstraction and classification – To reduce the complexity of the problem domain, high level and fundamental concepts (the "objects" of the management domain) are defined. These objects are then grouped into types ("classes") by identifying common characteristics and features (properties), relationships (associations) and behavior (methods).
- Object inheritance – Subclassing from the high level and fundamental objects, additional detail can be provided. A subclass "inherits" all the information (properties, methods and associations) defined for its higher level objects. Subclasses are created to put the right level of detail and complexity at the right level in the model. This can be visualized as a triangle – where the top of the triangle is a "fundamental" object, and more detail and more classes are defined as you move closer to the base.
- Ability to depict dependencies, component and connection associations – Relationships between objects are extremely powerful concepts. Before CIM, management standards captured relationships in multi-dimensional arrays or cross-referenced data tables. The object paradigm offers a more elegant approach in that relationships and associations are directly modeled. In addition, the way that relationships are named and defined describe the semantics of the object associations. Further semantics and information can be provided in properties (specifying common characteristics and features) of the associations.
- Standard, inheritable methods – The ability to define standard object behavior (methods) is another form of abstraction. Bundling standard methods with an object's data is encapsulation. Imagine the flexibility and possibilities of a standard able to invoke a "Reset" method against a hung device, or a "Reboot" method against a hung computer system regardless of the hardware, operating system or devi ce.

In addition, CIM's goal is to model all the various aspects of the managed environment, not just a single problem space. To this end, various "Common Models" have been created to address System, Device, Network, User and Application problem spaces. These problem domains are interrelated via associations and subclassing. They all derive from the same fundamental objects and concepts - as defined in the Core Model.

In order to understand why a unifying model is important, consider the following scenario. A payroll application generates an alert, due to multiple timeouts, when communicating with a database server. The alert is forwarded to a management application and network administrator who is monitoring alerts and events within a particular network domain, using the management application. The administrator performs a network route analysis to display all the "managed objects" and their percent utilization, in the path from the client application to the database server.

The administrator discovers that one card in a network device is at 98% utilization. In investigating further (by traversing CIM associations and exploring additional subclasses and data objects), the network administrator finds that one port of that card has a very high traffic rate, and its traffic is not related to the payroll application. Again following associations, the administrator can determine the "owner" of the System currently attached to the network port. If the owner cannot be reached, the administrator can use a standard method to "disable" the port, thereby returning the hub to normal bandwidth levels and allowing the payroll application to proceed.

In the preceding example, Network, Device, System and User Models were all exercised. It was required that the models' interrelationships and interactions be consistently defined, as well as the correct basic objects.

2. Overview of the Core Model

The Core Model establishes a basic classification of the elements and associations of the managed environment. The class hierarchy begins with the abstract Managed Element class which is in turn subclassed to Managed System Element, the Product related classes, Setting and Configuration, Collection and the Statistical Information classes. From the classes in the Core Model, the model expands in many directions, addressing many problem domains and relationships between managed entities.

> ***Note***: All classes defined by the DMTF in the Core and Common Models are named using the following syntax: CIM_<Class Name>. For reading convenience, the CIM_ prefix is omitted on class names throughout this paper, unless required for clarity.

The significant associations in which Managed and Managed System Elements are involved are the Statistics, Member Of Collection, Component and Dependency relationships, and the many subclasses of these associations.

Objects in the Core Model are defined and associated as shown in Figure 1. Each area of the Model is described in detail in the sections below. Note that all classes in the Core Model are not included in this figure (for example, Computer System). However, they are discussed and explained in the document.

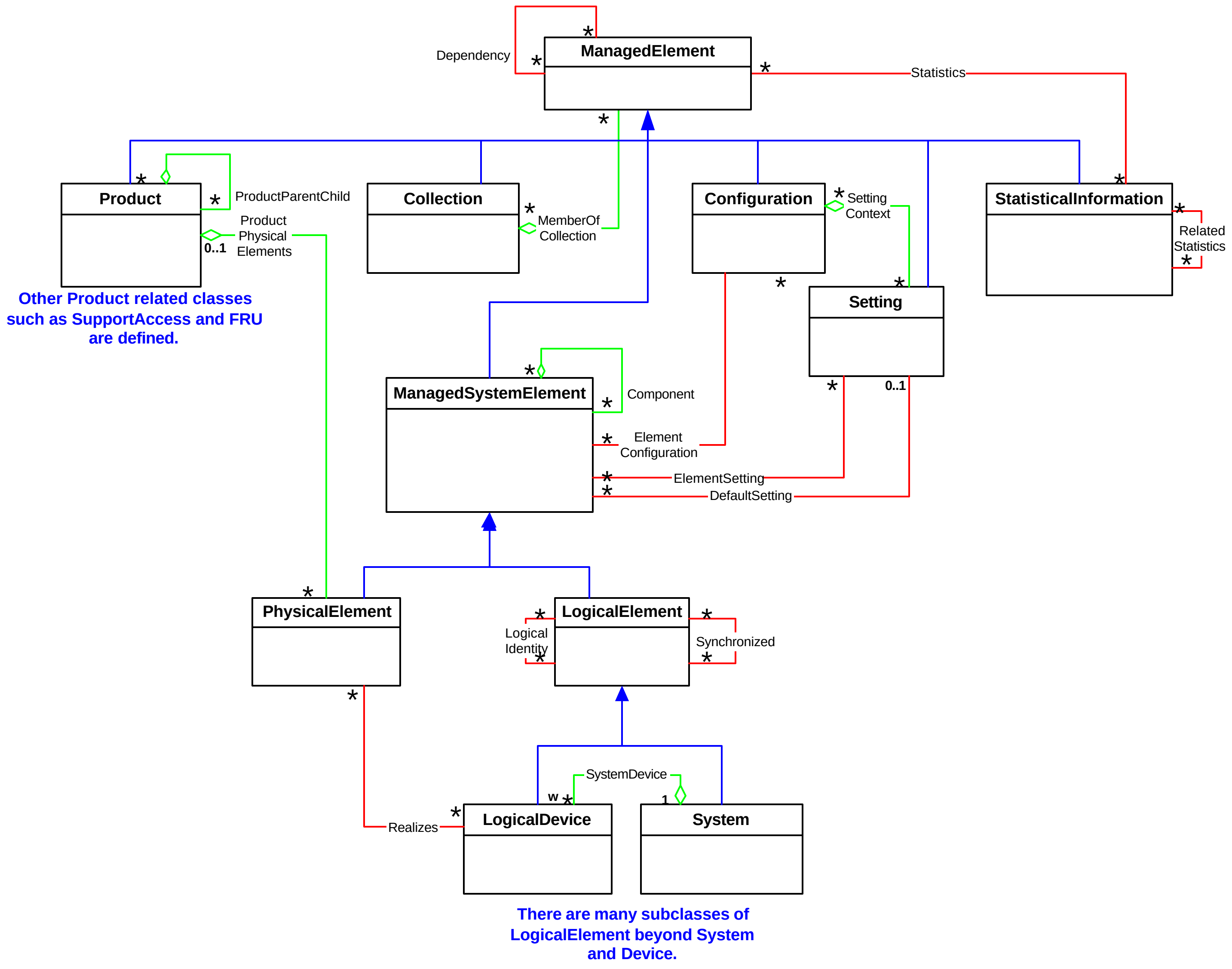


Figure 1. The "Top" of the CIM Object Hierarchy

Briefly summarizing the significant classes and associations of the Core Model …

- The Managed Element class roots the CIM object hierarchy and acts as a reference for associations that apply to all entities in the hierarchy.
- Managed System Elements represent Systems, components of Systems, any kinds of services (functionality), software and networks. The definition of "System" in the CIM context is quite broad, ranging from computer systems and dedicated devices, to application systems and network domains.
- Both Logical and Physical Elements are subclasses of Managed System Element. Further definition and specification of these subclasses are provided in the Core and Common Models. For example, System and Logical Device objects are subclasses of Logical Element, defined in the Core Model.

- Products represent contracts between vendors and consumers, and capture information about how the Product was acquired, how it is supported, and where it is installed.

- Settings define specific, pre-configured parameter data to be "applied" (loosely transactionally) to one or more Managed System Elements. Their definition is very much tied to the properties of existing objects through the Element Setting association. Configurations aggregate Settings and Dependencies, representing a certain behavior or desired functional state for Managed System Elements.

- The Statistical Information class is the abstract superclass for any kind of statistical data for a Managed Element. The Element to which the Statistical Information applies is indicated via the Statistics association.

- Collections represent arbitrary "bags" that group Managed Elements together. Membership can be described by the class definition and/or indicated by explicit instantiation of the Member Of Collection association.

- Component associations establish 'part of' relationships between Managed System Elements.

- Dependency associations describe functional dependencies (one object cannot function without the other) or existence dependencies (the object cannot exist without the other).

CIM's Core and Common Models provide a detailed accounting of the Systems, Devices, Networks, Users, Policies and related entities (such as software) in the managed environment.

3. Stability of the Core Model

Because the Core Model is a framework for the class structures that make up the overall CIM Schema, it is expected to be stable. Deletions from, or modifications to, the basic structure or content of the Core Model indicate a shift in interpretation of the classes and methodology that underlie the schema as a whole. These types of changes would impact and cause churn to the Common and Extension Models, and are not anticipated.

It is possible (and probable) that additional objects and properties will be defined in subsequent releases of the Core Model. The addition of classes, properties and associations does not have the same impact as deletions and modifications. Additions do not impact current Common and Extension Models. Existing application and instrumentation code should continue to function.

4. UML

Figure 1 and many of the other figures in this document are based on UML (Unified Modeling Language). Quoting from Appendix D of the *Common Information Model (CIM) Specification, V2.2* (June 14, 1999), “there are distinct symbols for all of the major constructs in the schema … In UML, a class is represented by a rectangle. The class name either stands alone in the rectangle or is in the uppermost segment … 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.”

Lines in the figures indicate:

- Inheritance relationships (blue lines with arrows) – Otherwise known as “is-a” relationships
- Aggregation/component relationships (green lines with a diamond shape at the “aggregating” end) - Otherwise known as “has-a” relationships
- Dependency and other relationships (red lines) – Some of which are “uses-a” relationships

Inheritance relationships are not specifically labeled or named, while all other associations are named. The cardinalities of the references on both sides of an association are indicated by numeric values or an asterisk (*) at the endpoints of the association. The following cardinalities are typically used in the CIM Schema:

0..1	Indicates an optional single-valued reference
1	Indicates a required, single-valued reference
1..n or 1..*	Indicates either a single or multi-valued reference, that is required
, 0..n or 0..	Indicates an optional, single or multi-valued reference

In the table, above, a “required” reference means that the object and the association MUST be instantiated when the other referenced class is instantiated. For example, in Figure 1, the System class has a cardinality of 1 in the System Device association. Therefore, when a Logical Device is instantiated, a scoping System and the System Device association must also be instantiated.

The symbol “w” is also used to label an association and can be seen in Figure 1. It indicates that the referenced endpoint or class is “weak” with respect to the other class participating in the association. This means that the referenced class is scoped or named relative to the other class, and the identifying keys of the class are propagated to the “weak” class. Note that this is not standard UML convention, but an added symbol in the CIM diagrams.

Taking the System Device association as an example, one can conclude:

- It is a component relationship, where the System object aggregates its Device “parts”
- There is one System object that must be referenced by a Device (the cardinality on System, “1”, indicates a required, single-valued reference)

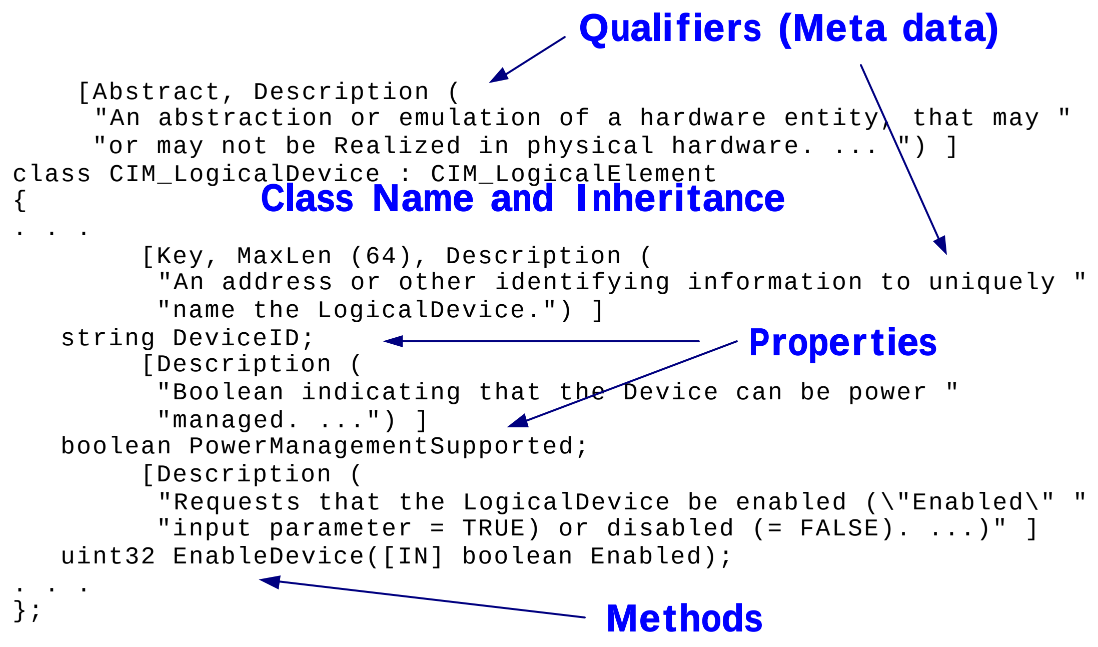
- There can be many Devices associated with a System
- Devices are scoped or named relative to the System to which they are “weak”

Occasionally, there is confusion related to the cardinalities of the Model associations. Cardinalities define the number of INSTANCES OF THE ASSOCIATION for single INSTANCES of the referenced classes. They do not define the total number of instances for the class as a whole.

For example, a particular instance of a Keyboard Device is associated with a particular instance of a Computer System, on which the keyboard is installed (and, by which it is scoped). However, the CIM_Keyboard CLASS has many instances of many keyboards, each associated with their own Computer Systems. Does this then mean that the association, System Device, should be many Keyboards related to many Computer Systems? No – it defines the relationship between an instance of Computer System and an instance of the Keyboard, not for the Keyboard class as a whole.

5. CIM's Syntax - MOF

The following figure illustrates MOF (Managed Object Format), the syntax of the CIM Schemas.



```
    [Abstract, Description (
     "An abstraction or emulation of a hardware entity, that may "
     "or may not be Realized in physical hardware. ... ") ]
class CIM_LogicalDevice : CIM_LogicalElement
{
. . .
        [Key, MaxLen (64), Description (
         "An address or other identifying information to uniquely "
         "name the LogicalDevice.") ]
   string DeviceID;
        [Description (
         "Boolean indicating that the Device can be power "
         "managed. ...") ]
   boolean PowerManagementSupported;
        [Description (
         "Requests that the LogicalDevice be enabled (\"Enabled\" "
         "input parameter = TRUE) or disabled (= FALSE). ...)" ]
   uint32 EnableDevice([IN] boolean Enabled);
. . .
};
```

Figure 2. MOF Example

Inheritance is indicated by placing a colon and the superclass name after the class name. In the figure above, the CIM_LogicalDevice class is defined. It subclasses from CIM_LogicalElement.

Qualifiers are mentioned, and several are used in the MOF example above. Qualifiers are meta data, providing information about a class, property, method or reference in the CIM Schema. The intent and purpose of most of them is quite obvious (for example, *Description ("xxx"), MaxLen (256)* or *Units ("Seconds")*). All of the CIM qualifiers are listed and explained in Section 2.5 of the *Common Information Model (CIM) Specification, V2.2* (June 14, 1999). However, some of them need additional explanation. These are overviewed below:

- *Key* - Is a Boolean indicating that the property that carries the qualifier is part of the "key structure" of a class, uniquely identifying instances of the class. The *Key* qualifier may be placed on multiple properties of a class. If this is done, the combination of the values for the *Key* properties must be unique across all instances of the class, in a particular namespace.

- *Weak* and *Propagated* – The *Weak* qualifier is placed on a reference in an association and indicates the class that IS "weak." When a class is weak to another class, it includes the keys of that other class in its own key structure. These key properties are labeled with the *Propagated* qualifier, and essentially act as foreign keys (from a database perspective). For example, a disk drive can be the C: drive of a computer system. But in an enterprise, many C: drives may exist. An instance is only completely named when you specify the system name and then the name, "C:". This is true for any system device. To uniquely identify a device, the system that contains the device must be identified. So, the keys of the System object are part of the property/key structure of CIM's Logical Device. These System keys are each labeled with the *Propagated* qualifier.

  Note: *Propagated* keys are not listed in the UML diagram, only in the MOF. It is assumed that having the "w" (*Weak*) designation in the UML is sufficient to indicate that the other class' Key properties are propagated to the *Weak* class.

- *Min and Max* - Cardinalities on association references are denoted using the *Min(#)* and *Max(#)* qualifiers. If these are absent, the defaults apply - *Min(0)* and *Max (NULL).* Cardinalities are defined from the perspective of the other referenced entity in an association. They define the number of instances of one class that may be related, using the association, to a single instance of the other class. For example, if an association relates class A to class B, then A's cardinality indicates how many instances of A may be associated with a single instance of B. Similarly, B's cardinality describes the number of instances of B that may be associated with a single instance of A. Cardinality does not dictate the number of instances of the association or of the individual classes.

- *ModelCorrespondence* - Is an array of strings of the form, "<ClassName>.<PropertyName>". The qualifier may only be used on a property and indicates that the property is related to those listed in the *ModelCorrespondence* array. The relationship may be that the properties have identical enumerations, or should be set equivalently or similarly. Another use of *ModelCorrespondence* is where one property provides more detail for another. *ModelCorrespondence* is about any "correspondence," not just identical enumerations. An example of "providing more detail" occurs in the CIM System Model, in the BIOS Feature class. The Characteristics property has a *ModelCorrespondence* with an array of Characteristic Descriptions. An example of "identical enumerations" occurs in the Device Model where a Print Job's paper and language requirements (PrintJob.RequiredPaperType and Language) have *ModelCorrespondences* with what a Printer supports (Printer.PaperTypesAvailable and LanguagesSupported).

ModelCorrespondence does not require symmetry - if PropertyA has a ModelCorrespondence listing PropertyB, it is not required that PropertyB also have a ModelCorrespondence. When the correspondence is only one-way (for example, this property is set equal to that property), then the correspondence is one-way.

- *Required* - Is a Boolean indicating that a value must be provided for the labeled property in any instance of the class.

- *ValueMap* - Is an array of strings indicating the permissible values for a property. Strings are chosen as the basis for describing values, since all data types are translatable to a string, and the concept of variants is not currently supported in CIM. For example, in the Core Model, the Status Info (uint16) property of Logical Device can be set to one of the values, "1", "2", "3", "4" or "5". As another example, the Status (string) property of Managed System Element can be set to one of the values, "OK", "Error", "Degraded", "Unknown", "Pred Fail", "Starting", "Stopping", "Service", "Stressed", "NonRecover", "No Contact" and "Lost Comm". If no *ValueMap* is specified, the property can take any legal values for its data type.

- *Values* - An array of strings related to *ValueMap* that describes the textual representation of the *ValueMap's* contents. This qualifier is particularly useful when the property is of type, integer. Examining the MOF below illustrates the use of this qualifier:

```
[Description (
   "StatusInfo is a string indicating whether the Logical"
   "Device is in an enabled (value = 3), disabled (value = "
  "4) or some other (1) or unknown (2) state. If this "
   "property does not apply to the LogicalDevice, the value, "
   "5 (\"Not Applicable\"), should be used. . . ."),
 ValueMap {"1", "2", "3", "4", "5"},
 Values {"Other", "Unknown", "Enabled",
         "Disabled", "Not Applicable"},
 MappingStrings {"MIF.DMTF|Operational State|004.3"} ]
 uint16 StatusInfo;
```

In this example, the value 1 "translates" to the string "Other", 2 translates to "Unknown", etc. If translations to other languages are desired, the *Values* array can be written as *Values-ll-cc*, where ll and cc are language and country codes as specified by the ISO/IEC 639 and ISO/IEC 3166 standards. (This convention of adding ll-cc to a qualifer, to provide its information in another language, is valid for any qualifier that is defined as "*Translatable*". Qualifiers are defined in the front of the Core Model MOF.)

There is one final convention that should be understood when using *Values* arrays - a *Values* array may be specified without a *ValueMap* qualifier. This is only valid when the property is a numeric data type, and results in an implicit *ValueMap* definition. It is assumed that the permissible values start with 0 and increment by 1, ending with the number of *Values* array entries minus 1. An example is shown in the following MOF from the Core Model:

```
[Description (
   "Enumeration indicating whether the ComputerSystem is "
  "a special-purpose System (ie, dedicated to a particular "
   "use), versus being 'general purpose'. For example, one "
```

```
      "could specify that the System is dedicated to \"Print\" "
      "(value=11) or acts as a \"Hub\" (value=8)."),
    Values {"Not Dedicated", "Unknown", "Other", "Storage",
      "Router", "Switch", "Layer 3 Switch",
      "Central Office Switch", "Hub", "Access Server",
      "Firewall", "Print", "I/O", "Web Caching", "Management"} ]
   uint16 Dedicated[ ];
```

In this example, there is an implicit *ValueMap* qualifier enumerating the integers starting with 0 and ending with 13.

- *BitValues* and *BitMaps* - Designed to be complementary to the *Values* and *ValueMap* qualifiers, except that the enumerations apply to positions in a bitmapped parameter.

- *OctetString* - Is a Boolean indicating that the labeled property or method parameter uses a derived data type, octet string. This qualifier is defined in "Approved Errata" to the CIM Specification V2.2 (found at http://www.dmtf.org/spec/errata.html). There are two forms of octet strings in CIM - an ordered uint8 array for single-valued strings, and a string array for multi-valued properties. Both are described by adding the *OctetString* qualifier to the feature.

  The first four numeric elements of both of the *OctetString* representations are a length field. (The reason that the "numeric" adjective is added to the previous sentence is that the string property also includes '0' and 'x', as its first characters.) In both cases, these 4 numeric elements (octets) are included in calculating the length. For example, the octet string X'7C' would be represented by the uint8 array, X'00 00 00 05 7C' or by the 12-character string “0x000000057C”. The latter would be the value of one of the array elements in a CIM array of strings. (Since CIM uses the UCS-2 character set, it requires 24 octets to encode the 12-character string.)

5.1 Extending CIM's Enumerations

The addition of values to an enumeration (a *Values* or *ValueMap* array) is not permitted in a class or its subclasses. A subclass can restrict the permissible values, but not extend them. The reason for this limitation is that an administrator or application querying for the value of the enumerated property, may be querying at the superclass level. At this level, a fixed set of values is defined. To be returned additional or more specific values would not allow for consistent and predictable responses and processing.

Let's go through an example. The Managed System Element class has a Status property. Its permissible values include "Unknown", "OK", "Error" and "Degraded", among others. Now a printer, which inherits from Managed System Element, might have additional statuses such as "Jammed", "Low Toner", etc. These values certainly are not applicable to all Managed System Elements, nor would an application working at the very high level of Managed System Element be prepared to deal with such extended statuses as "Low Toner", "Dial Tone Lost" or "Incoming Call". For this reason, extensions to enumerations are not allowed.

But, all is not lost. There are four possible mechanisms to supplement the values in an enumeration:

- Most *Values* and *ValueMap* arrays include a value of "Other". Typically, an "Other Description" string property is also defined, with a *ModelCorrespondence* to the property with the *Values*/*ValueMap* arrays. To extend an enumeration, "Other" can be used, and any additional meaning conveyed in the "Other Description" string.

- Additional properties, supplementing an enumerated property, may be defined in subclasses. The printer example above is actually taken from the CIM Device Model. In the CIM_Printer class, both Printer Status and Detected Error State properties are defined (again as enumerations) - supplementing the information in Managed System Element's Status property. Detected Error State's *Values* array includes "Low Paper", "No Paper", "Low Toner", "Jammed" and "Output Bin Full". Printer Status' *Values* array includes "Idle", "Printing", "Warmup" and "Offline".

- A developer may contact the DMTF Technical Committee (technical@dmtf.org) to recommend the addition of new permissible values for an enumeration. This would be handled via a Change Request. If approved, the new values would appear in the next release of the CIM Schema.

6. Subclassing in CIM

One of the strengths of the CIM Schema is its use of object-oriented design techniques. Inheritance (via subclassing) is one of these techniques. It is a powerful concept, but often questions arise regarding the appropriateness of subclassing - especially with respect to associations.

Listed below are various rationale for creating subclasses. Some of these are specific to associations since they address references and cardinalities.

The most common reason to subclass is to specialize semantics, moving from more general to more specific semantics. For example, a Logical Device is a subclass and specialization of Managed System Element. As another example, the association CIM_ServiceServiceDependency subclasses from CIM_Dependency, describing the specific dependency of one Service on another.

As part of specialization, new properties, methods or associations are defined for the subclass. The semantics conveyed by the new constructs are not appropriate for the entire population of the superclass, hence the subclass must be created.

When specializing associations (i.e., relationships), you often constrain their references or restrict their cardinalities. If semantics are specialized, it is likely that the references of the association are also specialized. For example, CIM_Dependency relates Managed System Elements, but CIM_ServiceServiceDependency only relates instances of CIM_Service. On the cardinalities side of the argument, a superclass may define a many-to-many relationship, but a subclass could restrict one of the references by modifying its cardinality to "0..1".

Sometimes, subclassing is done to indicate where a specific concept is located in the model, or what is particularly important and/or frequently instantiated in the model. Related to this, a subclass may be defined to easily query for its instances. This is the reasoning behind the many specializations of the Based On association in the Device Model. For example, the CIM_LogicalDiskBasedOnPartition association describes how a "C:" drive is built on top of a disk partition. It is a specific subclass of the Based On relationship between general Storage Extents.

Lastly, subclasses may be defined to specifically align with and "map" an existing standard. For example, the definition of Protected Space Extents and their relationship to Physical Extents is described in the Device Model as a direct mapping of the SCC (SCSI Controller Command) Specification.

7. Naming in CIM

In any instrumentation and management scheme, it is necessary to uniquely identify instances of classes. Various means of identification exist such as:

- Simple numeric counter - for example, an incrementing counter on messages sent between two entities
- GUIDs (globally unique identifiers)
- One or more parameters, which (when taken together) provide a unique identifier - for example, key properties in a database table

The latter is the approach chosen by the CIM designers.

The CIM Schema is a “keyed” object model, as opposed to an object identity model (for example, a GUID-oriented model). All class instances are uniquely named and referenced by the class’ keys. Associations are uniquely identified by their keys, which have always included their reference properties. References consist of a class' keys and an instance's values for these keys. All concrete classes (those that are instantiated) must define or inherit a key structure. If inherited, the key structure cannot be changed.

As noted in Section 5, CIM specifies the key properties of a class using the *Key* qualifier. In native CIM implementations, it is required that the combination of properties with the *Key* qualifier is unique across all instances of a class, within a namespace. This combination of *Key* properties is the identification scheme for a native CIM class.

Since CIM is an information model, it may be implemented using various protocols and/or repositories. Different implementations of the CIM Schema may require different identification schemes and additional properties. For example, GUIDs may be used as one of the values in a class' key structure. On the other hand, a directory implementation of CIM (as specified by the DEN initiative) uses Distinguished Name as its native identification scheme, and specifies that the combination of CIM *Key* properties with their individual values is an RDN (Relative Distinguished Name). As is done in DEN, maintaining the CIM key values is critical when communication with a native CIM implementation is required.

A question often arises regarding guidelines for class identification/naming in CIM - when should a modeler use regular keys and when should a class also have *Propagated*/*Weak* keys? The design is decided for almost all the CIM classes (and therefore by inheritance to their subclasses). Most classes are weak to another, and therefore use propagated keys. For some classes, however - for example, subclasses of CIM_Setting - the key structure is left up to the modeler. Guidelines for designing keys are to use the *Weak* qualifier in an association, and then *Propagate* the scoping class' keys, when an object does not exist on its own (i.e., is dependent on another entity for existence), or needs additional scoping information to create a unique key. For example, there may be many document1.doc files on many file systems on many computers. This would indicate that files are weak to their file systems, and file systems be weak to the computers on which they are hosted. So, you could have key properties that identify the file, the file system and the computer. Indeed, this is how the keys are defined for CIM_LogicalFile.

7.1 The Name Property

Note that many CIM classes specify Name as one of their *Key* properties. The Name property is inherited from Managed System Element to all Logical and Physical Element subclasses, and is also defined in several other classes such as Configuration and Product. In many cases, the property is specified as a *Key* property, or overridden in a subclass to identify it as a key. This is true for Systems, Services and Service Access Points. However, the Logical Device and Physical Element classes do NOT include the Name property in their key structures.

In all cases, Name is a label for a class. However, some classes have the additional constraint that Name be part of the unique identifier for the class. You must look at a class' definition in order to determine its *Key* properties.

Some implementations may want to consistently define and use the Name property as a "common name" or simple textual label. This is not possible since these "common names" and labels often are not unique. Duplicate keys could result. An implementation may be forced to use a different semantic for "Name" (i.e., not "common name") to achieve uniqueness. For example, Name might hold an algorithmically generated OID, or object id. In this case, the semantics of "common name" and textual label could apply to another property, such as Caption or a new CommonName property in a subclass.

7.2 Creation Class Name

Most of the CIM classes include a Creation Class Name property in their key structure. This provides another dimension (the class name) to an instance's key in an effort to avoid naming collisions. Creation Class Name helps in defining a unique instance identifier, in the specific case of instances belonging to two different subclasses of a common superclass. The number of objects over which the Name property must be unique is limited, by the use of the Creation Class Name, to all instances of the class being instantiated. The following example illustrates how Creation Class Name works.

Suppose we have instances of two different subclasses of CIM_System, one is a node participating in a cluster and the other is the cluster itself. Assume that both the computer node and the cluster have their Name properties set to “George”. This is allowed - since the subclass that is instantiated for a computer node is CIM_UnitaryComputerSystem; while for a cluster, CIM_Cluster is instantiated. (Both CIM_UnitaryComputerSystem and CIM_Cluster are defined in the CIM System Model.) The keys of a CIM_System are the CreationClassName AND the Name properties. In this example, two instances are defined, identified as follows:

- CreationClassName = “CIM_UnitaryComputerSystem”, Name= “George”
- CreationClassName = “CIM_Cluster”, Name=”George”

This is illustrated in Figure 3.

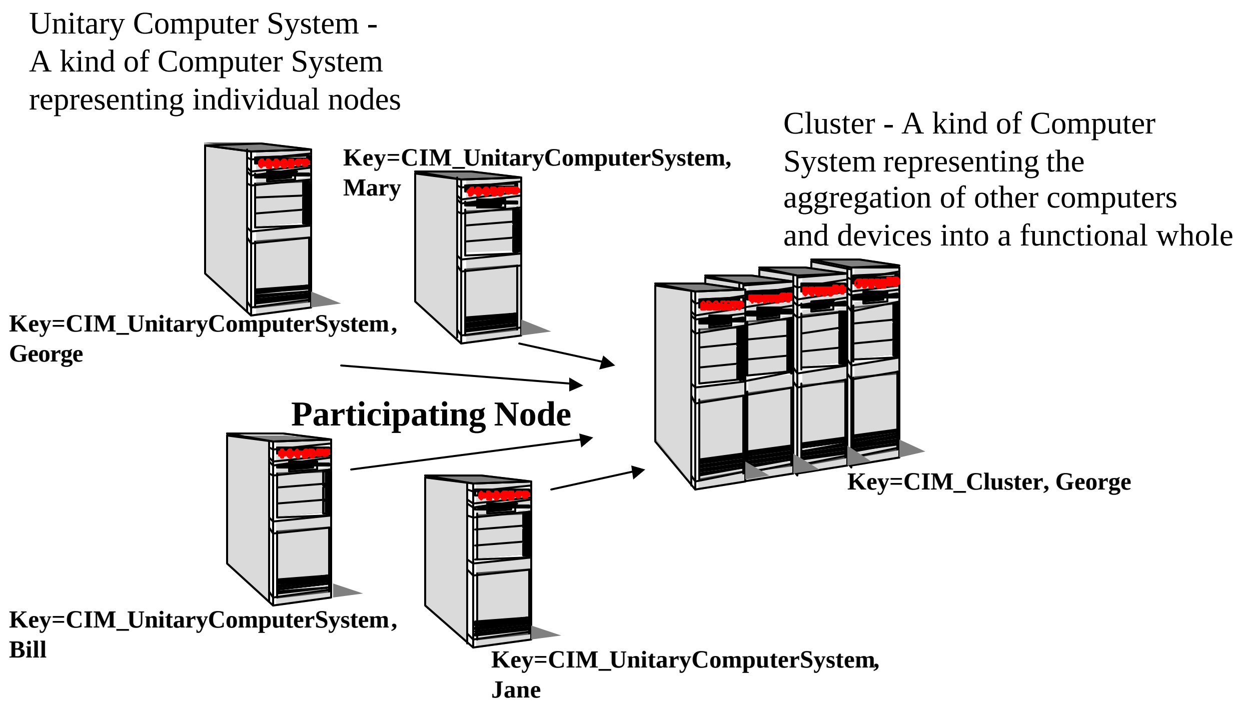

Unitary Computer System -
A kind of Computer System
representing individual nodes
Key=CIM_UnitaryComputerSystem, Mary
Cluster - A kind of Computer System representing the aggregation of other computers and devices into a functional whole
Key=CIM_UnitaryComputerSystem, George
Participating Node
Key=CIM_Cluster, George
Key=CIM_UnitaryComputerSystem, Bill
Key=CIM_UnitaryComputerSystem, Jane

Figure 3. Naming of Systems

If we had only the single key property - Name - available for distinguishing the two instances, then a collision would result from naming both of the instances with the value, "George". With Creation Class Name, collisions of this type are eliminated, without requiring coordination among domain administrators or instrumentation providers. The two instances are distinguished and uniquely identified based on their Creation Class Name values.

An instance's value for Creation Class Name is constrained to be the name of a concrete class in the instance's superclass object chain, or the name of the instance class itself. Creation Class Name is set according to the conventions of the key algorithms of an implementation. For example, it can be set to:

1. A constant (which is not recommended nor is it useful for distinguishing instance names)
2. The name of the first concrete class (moving down from the top of the object hierarchy)
3. The name of the last (leaf) concrete class in the hierarchy

There is a significant difference between #'s 2 and 3. For the latter (#3), an instance's key values are defined by the leaf class' instrumentation. For the former (#2), the instance's key values are defined by the instrumentation of the first concrete class. (A concrete class is one that can be instantiated.) A direct result of #2 is that instances cannot be created for subclasses that are not known to (and identified by) the instrumentation of the first concrete superclass.

Note: The topics discussed in the preceding paragraph are implementation-specific, and are not dictated or restricted by the CIM Schema.

8. Properties and Associations of Managed Element

In CIM V2.3, the object hierarchy became rooted by Managed Element. Before V2.3, there were various unrooted objects at the top of individual hierarchies, with associations tying the classes together. A rooted hierarchy is very useful to describe relationships that span all the entities in the CIM Schemas. Examples of these relationships are CIM_Dependency, CIM_MemberOfCollection and CIM_Statistics.

Since Managed Element is such a high level, abstract class, few properties could be defined as appropriate to all CIM elements. The current properties are Caption (a short textual description) and Description (a more detailed explanation of an instance). However, even these properties are not appropriate for a few subclasses of Managed Elements that are designed to be "lightweight." For example, the class CIM_UnitOfWork in the DAP Model recommends that Description not be used in order to reduce the amount of data in an instance. This is allowed, since neither of the properties of Managed Element is required. The Caption and Description properties can be left NULL.

It should also be noted that Caption and Description may be overridden in subclasses (using the *Override* qualifier) to indicate specific information that should be provided in these properties or to attach additional qualifiers such as *MappingStrings*. The *MappingStrings* qualifier indicates where other standards (such as DMI MIFs and SNMP MIBs) address the same information as conveyed by the labeled feature.

With the move to root the object hierarchy, one existing association was also moved up and references it - CIM_Dependency. Before CIM V2.3, Dependency only referenced Managed System Elements. However, it is very reasonable and desirable to model dependencies between all the managed entities. There are no constraints on dependency relationships in the Core Model. Any object can be dependent on any other object. There are however a number of distinguishable “flavors” of dependency, notably existence and functional dependency.

Some subtypes and examples of existence dependency are the Realizes association (between a Logical Device and its basis in hardware / Physical Elements) and the Hosted Service association (between a Service and a System on which it resides). A subtype and example of functional dependency is the Service SAP Dependency association – describing that a Service uses an Access Point to provide its underlying functionality.

A common source of misunderstanding related to the CIM_Dependency association is which reference actually IS the dependent one. The answer is that the entity referenced as Dependent is the one that IS dependent (like a Dependent on your tax return). The Antecedent reference is the one that is independent.

With several associations on Managed Element, an attempt was also made in CIM V2.3 to root existing associations - subclassing from the three relationships on Managed Element (Dependency, Member Of Collection and Statistics). For example, it is reasonable and straightforward to subclass CIM_CollectedMSEs from CIM_MemberOfCollection. Yet, if you examine the MOF, you find many examples of classes which semantically should subclass from one of these associations, but do not. One example is CIM_CollectedCollections, which should subclass from CIM_MemberOfCollection. It currently subclasses from nothing. The reason that these associations are not subclassed is that their reference names are not the same.

Similar to a class inheriting a property with a particular name, associations inherit references with particular names - and these names typically also are defined as *Key* properties. You cannot override these names - or else key structure and manipulation of the associations become a nightmare. For example, with unchanging reference names, it is easy to query for all the Dependent objects of an instance, even if related via subclasses of the Dependency association. It is much harder to query and manipulate all the possible references if the name "Dependent" could be overridden to be "DependentService", "DeviceDependency" or "ElementA". Similarly, you have key construction and manipulation issues if the Key property is referred to as "Dependent" in the superclass, but "ElementA" in the subclass.

Therefore, where reference names match, association subclassing was added to the Model. Where names do not match, the CIM designers felt it better to maintain current naming, and consistency for existing instrumentation and applications. This will be corrected in a future major release of the Schema.

As regards the associations, Statistics and Member Of Collection, these are fairly obvious in their intent. They relate a Managed Element to another entity holding instance-level statistics, or acting as a set or bag, grouping the Managed Elements into it.

9. Properties and Associations of Managed System Element

Managed System Elements represent Systems (Computer, Network, Storage Library and Application Systems), the software that runs on them, the functionality provided by them, and abstractions of the hardware that compose them. The subclasses of Managed System Element deal with everything related to “systems management” today. Its high level concepts are shown in Figure 4, below.

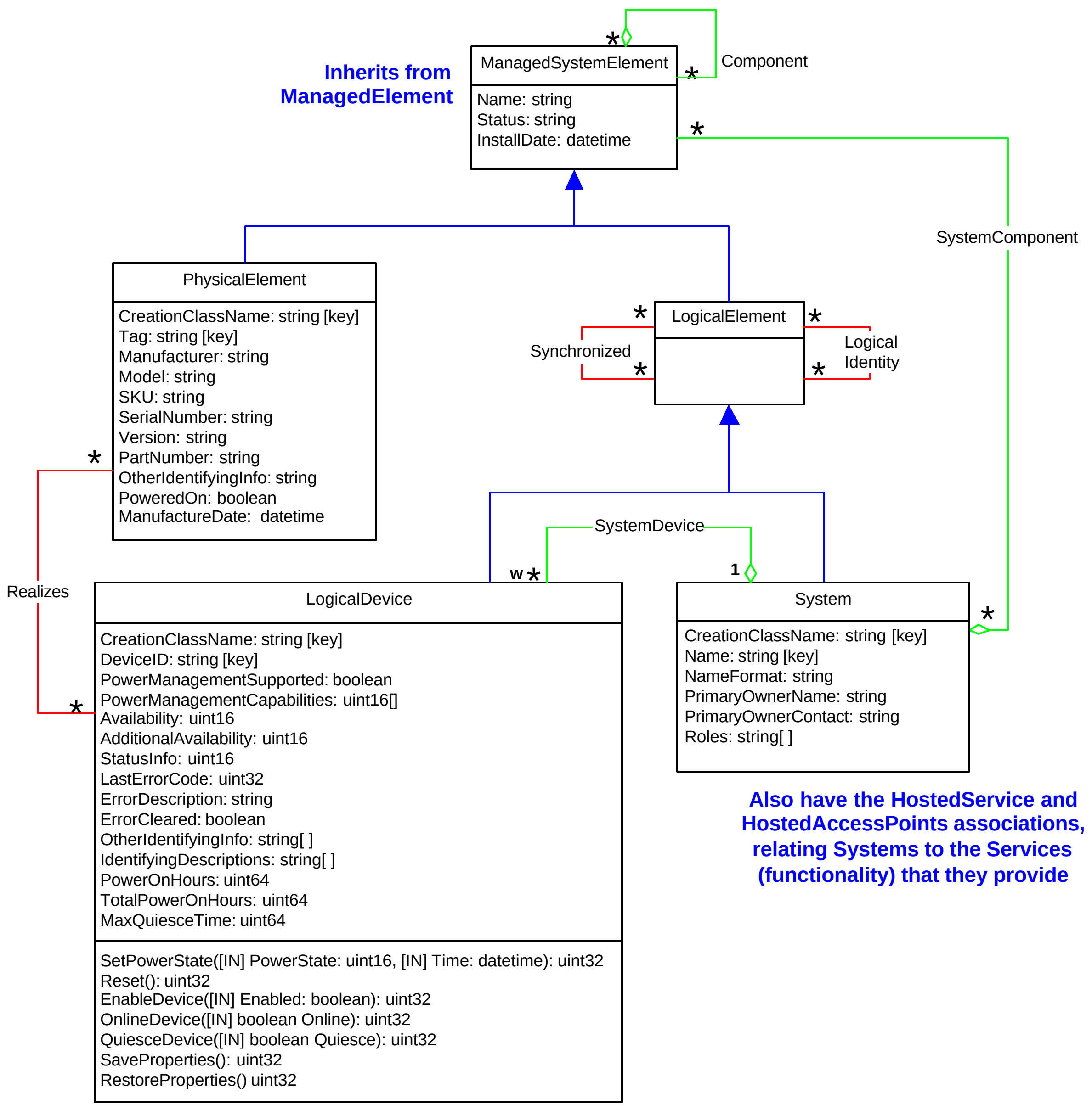

Inherits from ManagedElement
ManagedSystemElement
Name: string
Status: string
InstallDate: datetime
Component
SystemComponent
PhysicalElement
CreationClassName: string [key]
Tag: string [key]
Manufacturer: string
Model: string
SKU: string
SerialNumber: string
Version: string
PartNumber: string
OtherIdentifyingInfo: string
PoweredOn: boolean
ManufactureDate: datetime
LogicalElement
Synchronized
Logical Identity
SystemDevice
Realizes
LogicalDevice
CreationClassName: string [key]
DeviceID: string [key]
PowerManagementSupported: boolean
PowerManagementCapabilities: uint16[]
Availability: uint16
AdditionalAvailability: uint16
StatusInfo: uint16
LastErrorCode: uint32
ErrorDescription: string
ErrorCleared: boolean
OtherIdentifyingInfo: string[ ]
IdentifyingDescriptions: string[ ]
PowerOnHours: uint64
TotalPowerOnHours: uint64
MaxQuiesceTime: uint64
SetPowerState([IN] PowerState: uint16, [IN] Time: datetime): uint32
Reset(): uint32
EnableDevice([IN] Enabled: boolean): uint32
OnlineDevice([IN] boolean Online): uint32
QuiesceDevice([IN] boolean Quiesce): uint32
SaveProperties(): uint32
RestoreProperties() uint32
System
CreationClassName: string [key]
Name: string [key]
NameFormat: string
PrimaryOwnerName: string
PrimaryOwnerContact: string
Roles: string[ ]
Also have the HostedService and HostedAccessPoints associations, relating Systems to the Services (functionality) that they provide

Figure 4. The Managed System Element Hierarchy

Determining the properties of a generic object, from which all Systems and their components subclass, is a difficult task. Much greater property-related detail is provided in the subclasses of Managed System Element in the Common Models for Systems, Devices, Networks and Applications. In the Managed System Element class, only a few properties can reasonably be defined:

- Name
- Description and its shorter form, Caption (Inherited from Managed Element)
- Install date (if one exists)

- Status – An enumeration defining the values: Unknown, OK, Error, Degraded, Pred Fail (failure predicted), Starting, Stopping, Service (i.e., being serviced/in maintenance), Stressed (for example, overloaded or overheated), NonRecover (non-recoverable error occurred), No Contact (entity is known to exist via other means but no management contact has been made), Lost Comm (communication to the entity has been lost)

The last property, Status, describes both operating and non-operational states for a Managed System Element. For example, the statuses, "OK", "Error", "Degraded", "Stressed" and "Pred Fail", all denote that the entity is functioning, although perhaps in a non-perfect state. On the other hand, the statues, "Starting", "Stopping", "Service" and "NonRecover" indicate that the entity is not currently operational and is not performing its normal functions. The remainder of the property values deal with lack of information. The last two statuses ("No Contact" and "Lost Comm") could all be grouped under the catch-all, "Unknown". However, these values provide additional information on *why* Status is "Unknown".

In examining the Status property, one oddity may be noticed - that the enumeration is a 10-character string and not a uint16 data type. The next question is usually "why is this?" The Status property was not converted into an enumeration in moving to CIM V2.x since existing V1.0 implementations relied on it. These implementations had already standardized on a "short string" representation (i.e., a 10 character string). So, the CIM designers felt that it was more important to be consistent with existing implementations, than to move to an unsigned integer data type. This is especially true, since the semantics were not affected.

That Component and Dependency associations exist between Managed System Elements was mentioned earlier in this document. (The Dependency association is inherited from Managed Element.) These association types — like many of the Core Model classes — are abstract. This means that there are no direct instances of these classes. Instances must always belong to one of their subclasses. The relationships between subclasses of the Core Model are mainly defined by descendents of these abstract Component and Dependent associations.

Element composition (the Component association) is considered a fundamental relationship of which all Managed System Elements are capable. Expressed another way, any Managed System Element may be described in terms of other Elements, of which it is “composed”. However, the Component association, on its own, is too abstract to convey any specific information. Its main significance is to introduce an association type that can be subtyped to establish concrete relationships between descendent classes of Managed System Element. Programmers, schema designers, and browsers can then reasonably query, “What composition relationships does this Element have with other Elements?” The answer is found by locating all subclass of the CIM_Component association for the element of interest.

There is one important specialization of the composition association in the Core schema: the System Component association — relating a System to the Managed System Elements (both logical and physical) of which it is composed. This association is then further subclassed by the System Device relationship, to indicate that Logical Devices are parts of Systems - and indeed are *Weak* to those Systems.

10. Logical and Physical Split

The elements that make up a System can be:

- Physical Elements, occupying space and conforming to the elementary laws of Physics. The Physical Element class represents any component of a System that has a physical identity - it can be touched or seen.

- Logical Elements, representing abstractions used to manage, configure and coordinate aspects of the physical or software environment. Logical Elements typically represent Systems themselves, System components, System capabilities and software.

The distinction between Logical and Physical Elements is fundamental to the structure of the Core Model. The principal distinguishing feature of a Physical Element is that it cannot have a "realization" (per Webster's definition of realization, it can not be "brought into being"). It can be composed of parts, but there is no sense in which, for example, a system enclosure is "realized" (or "brought into being") by a piece of molded plastic; it simply *is* a piece of molded plastic.

Logical Elements (especially Logical Devices) can be "realized" by installing Physical Elements and/or software. For example, it is not possible to attach a label to a modem. It is only possible to attach a label to the Card that "realizes" the modem. The same card could also "realize" a LAN adapter. These tangible Managed System Elements have a physical manifestation of some sort. However, the physical manifestation is very different than its management aspects and attributes. The latter are addressed by the Logical Elements, realized from the physical, usually accessed via software.

Figure 5 visualizes the discussion of the previous paragraph.

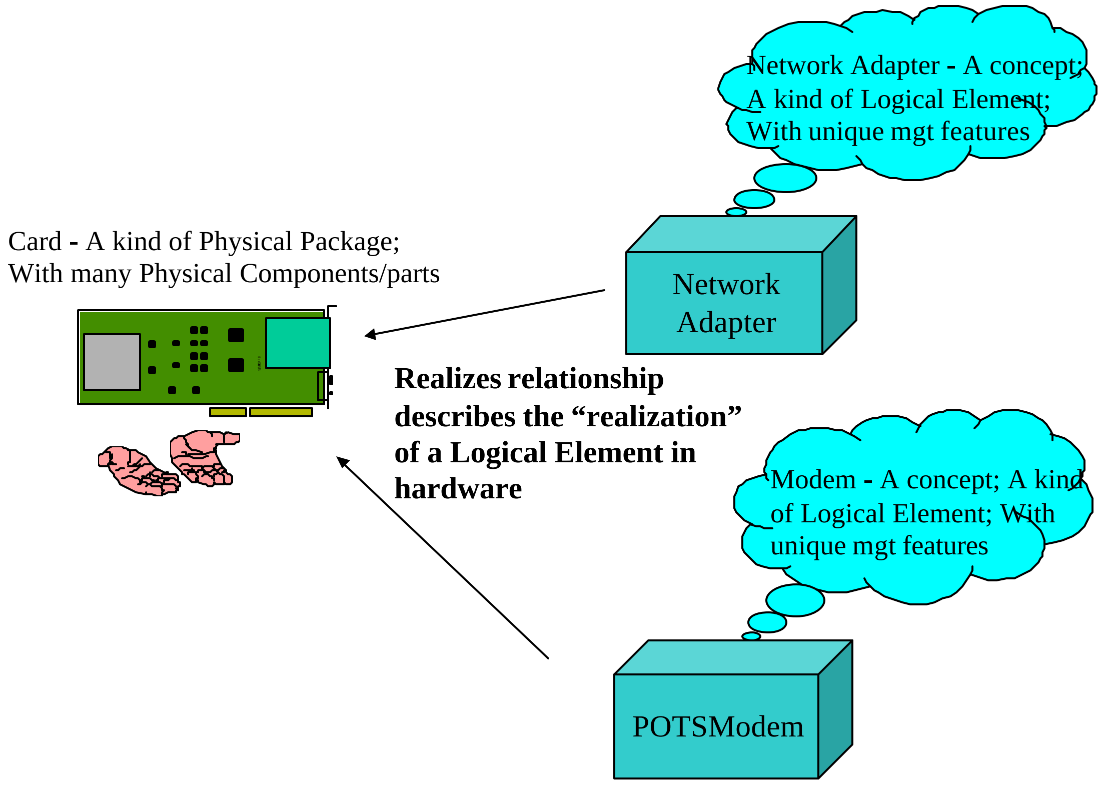

Network Adapter - A concept; A kind of Logical Element; With unique mgt features
Card - A kind of Physical Package; With many Physical Components/parts
Network Adapter
Realizes relationship describes the "realization" of a Logical Element in hardware
Modem - A concept; A kind of Logical Element; With unique mgt features
POTSModem

Figure 5. Physical Versus Logical

Note that it is not required that a Managed System Element be a discrete component. For example, it is possible for a single Card—which is a type of Physical Element—to host more than one Logical Device. Each of these Devices[1] would be associated with the Physical Element representing the Card by the "Realizes" relationship.

It must be understood that this fundamentally dualistic view of the systems management universe is just a view, and must be accepted and used as such. At both this very high level of abstraction and at lower levels, any particular modeling decision is likely to be a compromise between alternatives. Anyone making use of the schema, particularly in the context of extending or programming against it, must understand these compromises to avoid misunderstanding or introducing confusion into the schema itself.

Regarding properties (and referencing Figure 4 on page 18), the Logical Element class has none. It is a very abstract class, used to represent the management and configuration of hardware (in operation) and software. On the other hand, the Physical Element class has several properties, inherited by all Physical Model subclasses:

- Name (Inherited from Managed System Element)
- Description and Caption (Inherited from Managed Element)

[1] The noun, Device, when capitalized refers to the Logical Device class. It is a shorthand mechanism for identifying the class.

- Manufacturer, Manufacture Date
- Model, Version, Part Number
- SKU (stock keeping unit number)
- Serial Number
- Powered On boolean
- Tag – A string property that acts as the “key”, uniquely identifying the object within the CIM Schema
- Other Identifying Info - A string property that can be used for additional data, beyond that stored in the Tag property, to identify a Physical Element. One example where this property is used occurs if both an asset tag and a bar code exist on a Physical Element. One's value is chosen for the Tag property and the other's data can be stored in the Other Identifying Info property.

A few words are needed related to the Tag property, which is part of the key structure for Physical Elements (along with Creation Class Name). Tag is a free-form string that can hold asset tag information or other data that uniquely identifies the Physical Element. One may ask why a simple string key structure is defined for Physical Element - or, asked in another way, why no scoping/containment relationships are used to define keys. A single key structure, independent of all containment relationships, is defined because Physical Elements can be removed from their containing Packages and continue to exist. They must be identified independently of their containers. (However, containment relationships are critical and are identified as associations in the *Physical Model*.) For example, removable media or PCMCIA hardware continues to exist and is identifiable, even when not inserted into a Computer System or other Package.

A follow-on question is typically "why are not lower-level physical components weak to specific packaging elements like Chassis or Card?" The answer is based on the fact that a class can only be *Weak* to *one* other object/class. Take Physical Connectors as an example. They can be located on Chassis or Cards (kinds of Physical Packages), but also on cables (Physical Links). Therefore, Connectors would either have to be weak to both (invalid in the current specification) or neither. The “neither” option was selected. A single, asset-identifying string (the Tag property, with the additional clarification of Creation Class Name) is used as the Physical Element key.

11. Logical Element's Associations

Two general relationships are defined for Logical Elements - the Synchronized relationship and Logical Identity. Synchronized is very straightforward, and is considered first. Synchronized indicates that one element may be aligned with another, perhaps only at a particular point in time. Examples of Elements that are synchronized are replicated databases or data stores, clocks and "snapshoted" files. The relationship is on Logical Element since almost any logical entity may be synchronized with another. However, since cardinalities are many to many, the association indicates that not all elements are synchronized. The "0" part of the cardinality indicates that it is not necessary to instantiate the relationship for any specific Logical Element.

Properties of the Synchronized association are:

- WhenSynced - A date/time when the referenced Elements were last synced
- SyncMaintained - A Boolean indicating that synchronization is automatically maintained

The Logical Identity relationship is a bit more ethereal. It describes that a real world entity can have different aspects that would be likely be modeled using multiple inheritance. Two examples of the semantics of multiple inheritance occur in the Networks and Devices domains, and are described by subclasses of Logical Identity - Device Identity (in the CIM Device Model) and Endpoint Identity (in the CIM Network Model). Examining the use of these Identity subclasses in detail will help to explain the concept.

When managing Devices, it is usually necessary to represent both the 'bus' and the 'functional' aspects of the Element. For example, a Device could be both a PCI Device (or a USB Device), as well as a CIM_Keyboard. It would be very wrong (and would make the Device object unwieldy) to try to combine all the details of the different hardware bus protocols (PCI, USB, I2O, VME, etc.) into the basic Device class. Therefore, the Device Identity association was created to describe these different aspects of a Logical Device.

Similar to Device Identity, protocol endpoints (addresses and protocol-specific IDs) are related in an Endpoint Identity association. This association ties together the "LAN" and the protocol-specific aspects of an address. Using basically the same text as above ... it would be very wrong (and make the Protocol Endpoint object unwieldy) to try to combine all the details of the different network protocols (DHCP, DNS, BGP, SNMP, etc.) into the basic Endpoint class. So, the Endpoint Identity association was defined to describe these different aspects of a Protocol Endpoint.

In addition, it sometimes occurs that an Element plays multiple functional roles that are not distinguished in hardware. For example, in the Device world, a Fibre Channel adapter might have aspects of both a CIM_NetworkAdapter (its fiber channel network aspects) and a CIM_SCSIController (its use to access storage devices in a Storage Area Network). Describing this semantic is another use of the Device Identity association.

The concepts conveyed by Logical Identity and its subclasses (basically, multiple inheritance) are very powerful. Management applications should routinely query for any instances of the relationship associated with a Logical Element. For example, the reason that the CIM_Keyboard object may have an Error status is because its PCI controller is misconfigured or malfunctioning. Or, the reason that an IP address lease did not renew may be related to its DHCP endpoint configuration. To determine this, the Logical Identity association would be traversed and the "other" aspects of the Keyboard or the IP address examined.

On the instrumentation side, developers may be wondering when to use the Logical Identity subclasses. Whenever multiple objects are instantiated as different aspects of the same underlying entity (and typically have matching identifying information such as a PCI or USB ID, or a network address) - these objects are candidates to be related by a subclass of the Logical Identity association.

Note: If you examine the class structure for Logical Device and Unitary Computer System (in the CIM System Model), it may have been better to model all the power-related features of a system or device as a separate class, related via a subclass of Logical Identity - than to embed all the power management features in the base Device and Unitary Computer System classes. This would have afforded more modeling flexibility and reduced the size of the Computer System and Device objects. However, at the time that power management was added to these classes, these associations were not available in the Schemas.

12. Systems and Their Components

The System subclass of Logical Element aggregates Managed System Elements. For any subclass of System, there is a basic set of Elements whose instances can or must be aggregated. Systems represent individual entities that can be uniquely identified and are more than the sum of their parts. A System operates as a functional whole, and provides scope to its aggregated components.

Whenever compound entities can be identified in the managed environment, and these entities provide some functionality as a cooperating “whole,” the element that represents the “whole” is a good candidate for modeling as a subclass of System. Current subclasses are Computer System, Storage Library, Admin Domain and Application System (from the Core, System, Network and Application Models, respectively).

> **Note:** The definition and subclasses of CIM_System are not limited to computer systems, but are much more general.

The relevant associations and properties of the System class are shown in Figure 4, above. Systems have the following properties:

- Name – A string property that is combined with the Creation Class Name to create the System key (Inherited from Managed System Element and overridden to be a key property)
- Name Format – A string that identifies the heuristic used to define the System Name (for example, “IP” address)
- Primary Owner Name and Primary Owner Contact
- Roles - A string array representing the functions and characteristics of the System in the enterprise

The class' associations focus on the Component relationship - since Systems are essentially aggregations but viewed as functional "wholes."

The individual elements that make up a System can be enumerated using a number of different strategies. A function that lists the components of a System will start with a System object. Then, the System Component association is traversed to discover the comprising entities. The listing function must select the components to be enumerated based on the type of picture of the System to be presented. There are any number of alternative views depending on the circumstances at hand:

- If a list of the physical components is required, the function will list all the System Components that are Physical Elements.
- If the logical components of the System are required, the components of type Logical Element will be selected.
- Selecting Logical Elements that are not "weak to" (or scoped by[2]) any other Logical Elements would return top-level objects.
- Low-level elements (typically, a device-level view) could be constructed by selecting Logical Elements that subclass from Logical Device and/or have a Physical Element realization.
- Dependency or configuration trees could be constructed by pursuing suitable associations.

12.1 The Class, Computer System

Computer System is a subclass of System and is a special collection of Managed System Elements. It is part of the Core Model since the various types of computers - general purpose as well as dedicated systems - cross all the Common Models. A Computer System provides compute capabilities, hosts services, and aggregates devices, firmware and software.

Computer System serves as an abstraction layer and association definition point for the specific classes (Unitary Computer System, Cluster and Virtual Computer System) derived from it (all defined in System Model). It includes component associations to other Managed System Elements, such as Operating System, Logical Devices and File Systems, that execute on or are given scope by the Computer System. Note that the majority of these associations are defined in the CIM System Model, whose focus is on modeling computer systems and storage libraries. None of the associations are mandatory from the System's perspective, but *could* be defined if necessary to manage the System.

The view of a Computer System as an aggregation point, can be seen in the following figure.

[2] Systems are a type of Top Level Object (TLO) in the CIM Schema. Per the *Common Information Model (CIM) Specification, V2.2*, Section 5, Top Level Objects "should have relevance in an enterprise context." They "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." TLOs are *scoping objects* for any classes that are defined to be *weak* with respect to them. Scoping objects propagate their keys to weak objects.

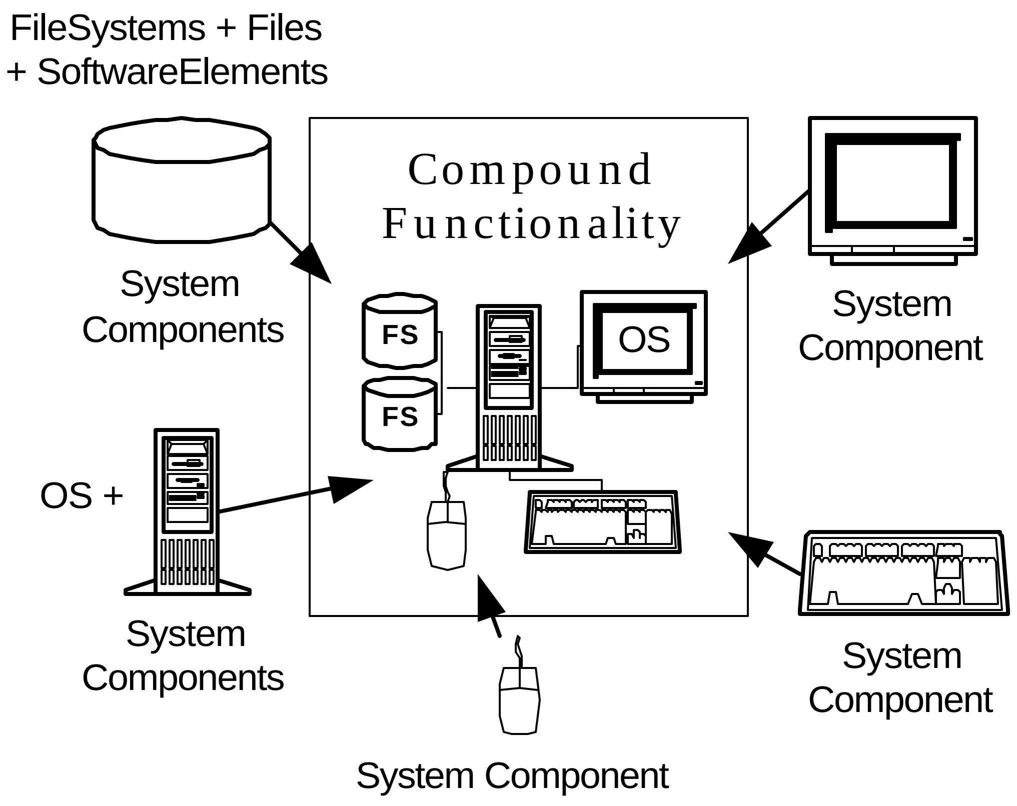


Figure 6. Computer System Aggregation

Especially when the Computer System is "dedicated," information such as Operating System may not be critical to the system's management. In many dedicated systems, the Operating System may be in firmware, or have very limited functionality. Obviously, however, some software boots the system and handles its operation. This is the essence of the definition of Operating System in the System Model, but more sophisticated implementations are also supported. Quoting from the System MOF:

```
"An OperatingSystem is software/firmware that makes a "
"ComputerSystem's hardware usable, and implements and/or "
"manages the resources, file systems, processes, user "
"interfaces, services, ... available on the ComputerSystem."
```

When examining the Computer System class, developers often ask how specific systems should be instantiated and/or subclassed. Many schema designers are tempted to subclass a router “system” or a network printer from Computer System. These entities seem to be special kinds of computer systems, dedicated to routing or to printing. However, what happens when a general-purpose computer gets a new Service – that supports and provides routing or print capabilities? Should that instance of Computer System be instantiated as a router or print “system” instead? Probably not – since the System is not dedicated. Computer Systems are distinguished by having compute hardware as Logical Devices (processor and memory) and the capability to run an OS (even if it is limited, vendor-specific firmware). Systems aggregate their components and host Services and Service Access Points. These aspects of a Computer System do not change just because the “System” is dedicated to routing or printing.

Before panicking, however, be assured that the CIM designers understood the value of labeling a System as "dedicated." Therefore, a Dedicated enumerated array property was added to the definition of CIM_ComputerSystem, to provide this information. In fact, there are three new properties on Computer System (beyond those inherited from System), as well as an Override of the existing Name Format property. Each of the properties is explained below:

- Other Identifying Info - A string array that can be used for additional data, beyond that stored in the Name property, to identify a Computer System. One example where this is used occurs when both a Fibre Channel World-Wide-Name (WWN) and a "System Name" exist for a Computer System. One's value is chosen for the Name property (likely "System Name") and the other's data can be stored in the Other Identifying Info array.

- Identifying Descriptions - A string array that has a *ModelCorrespondence* with Other Identifying Info. There is a correspondence between the entries at the same index in both arrays. The Description strings provide explanations and details behind the values in the Other Identifying Info array.

- Dedicated - An array of enumerated integer values, where the permissible values are: "Not Dedicated", "Unknown", "Other", "Storage" (for example, a storage array or network attached storage platform), "Router", "Switch", "Layer 3 Switch", "Central Office Switch", "Hub", "Access Server", "Firewall", "Print", "I/O" (for example, an Infiniband I/O system attached to and supporting multiple Computer Systems), "Web Caching" and "Management" (for example, a Service Processor or management controller in a high-end server).

- Name Format - An enumeration (overriding the free-form string property of CIM_System) defining the mechanism by which the computer Name is determined. Because computers may have several instrumentation packages and may be discovered via different mechanisms and technologies, it is likely that one system could have multiple Names. This property attempts to distinguish between the different mechanisms that are used to name a computer. Its values are:

  Other
  IP - Identification is related to IP-based networking information such as a fully qualified hostname or a permanent IP address that is assigned to the System
  Dial - Identification is based on relatively unchanging information such as a call-back number or other identification of the Owner of a System
  HID (Hardware ID) - Identification is based on a processor, chassis, cryptographic hardware or other hardware ID
  NWA (Network Address) - Identification is determined by some "other" network address than is specified in the current enumeration
  HWA (Hardware Address) - Identification is based on the hardware address of the main network interface of the Computer System
  X25 - Identification is related to X.25-based networking information
  ISDN (Integrated Services Digital Network) - Identification is related to ISDN-based networking information
  IPX (Internetwork Packet Exchange) - Identification is related to IPX-based networking information
  DCC (Data Country Code) - Identification is based on a network address using an ISO DCC format
  ICD (International Code Designator) - Identification is based on a network address using an ISO ICD format
  E.164 - Identification is based on an E.164 format, a telephone number-like address used in ISDN

SNA (Systems Network Architecture) - Identification is related to SNA-based networking information
OID/OSI (Object ID) - Identification is based on an Object ID

Before closing this discussion of Computer Systems, one last item needs to be clarified. It appears that there are two ways to describe the packaging of a subclass of Computer System, the Unitary Computer System. (Unitary Computer Systems represent personal computers, handhelds, servers, etc. - basically computers that are directly realized due to the installation of hardware and software.) Either the System Component association in the Core Model, or the Computer System Package relationship defined in the CIM Physical Model could be used to describe the enclosures of a Unitary Computer System. There is a difference between these associations since one is a Component and the other a Dependency relationship. It is correct to describe an enclosure as "part of" a Computer System, although it is a bit of stretch. The packaging of a Computer System does not affect the System itself. It could be packaged in a variety of different ways and form factors and still be a Computer System. It is more correct to say that the System is realized by the components contained within the enclosure - which are the semantics of the Computer System Package association.

13. Logical Devices

Important components of a System are its devices. These are represented as subclasses of Logical Device. There are many subclasses and associations, and much detail defined and described in the CIM Device Model.

A Logical Device provides an abstraction of hardware, has a distinct function (such as a modem or keyboard) and participates in providing or implementing Services and Service Access Points. Actually, Devices and Services are closely related on the basis of "functionality." Devices (i.e., hardware) provide certain functionality and may be required in the implementation of a Service or an Access Point. To visualize this, imagine having a networking service without a LAN Adapter to interface to the network. Although Services "officially" represent the CIM semantic of functionality, it is assumed that all Devices have a basic function that they perform. This could be modeled as a separate Service, but is typically rolled into the definition of the Device.

Any characteristics of a Logical Device that are used to manage its operation or represent its current configuration are contained in, or associated with, the Device object. Examples of the “operational” properties in a Printer Device would be Paper Sizes Supported, Languages Supported, Job Count Since Last Reset and Time of Last Reset. Examples of the “current configuration” properties of a Sensor Device would be threshold settings. These are all properties of subclasses of Logical Device, defined in the CIM Device Model. In addition, various *potential* configurations can exist for a Device. These potential configurations could be contained in Setting objects and associated with the Device. (Setting and Configuration objects are explained in a later section of this document. Refer there for more information.)

As stated above, Logical Devices abstract hardware - not the touching/seeing parts of hardware (which are modeled as Physical Elements), but the logical/operating side of hardware. Since the tie between physical and logical is paramount in the definition of Devices, a special association characterizes it - CIM_Realizes. This is a Dependency association between Logical Devices and Physical Elements. Its cardinality is many-to-many, indicating that hardware can realize many Devices, and a device may be realized across several Physical Elements.

On the topic of hardware, sometimes a Device is used on one System, but its hardware is located in another. Or, a Device may be shared across multiple Systems. These scenarios raise questions regarding how a Device is identified and how it can be determined that the

Device is local, remote or shared. Regarding identification, it is likely that the basic Device is modeled as a part of the System that hosts its hardware (whether general purpose or dedicated). Then, there are the "other" Systems that may use the Device, and may include it as one of their System Devices. In each of the Systems, the Device may be identified by a different name. Ultimately, however, a network or protocol address will tie the distinct Elements together. In addition, on the non-hardware hosting Systems, there is certainly no realization of the Device in the Physical Packages of that System (or there may be no Realizes relationship at all). So, how might the local Device and the real, remote Device on another System be associated? The Device could have a Device Identity relationship with its counterpart in the hosting System, or may be based/dependent on this other Device. Alternately, the local Device may require a networking or connection Service to get to the real, remote Device. Instrumentation would have to use one of these mechanisms to describe use of a Device on another System. Once the lowest level, real, remote Device was located (with a distinct Realizes relationship to locate its hardware), its associations could be examined to determine if any other Systems were using it in a similar fashion. In this way, it could be determined whether the Device was shared. This is illustrated in Figure 7, below.

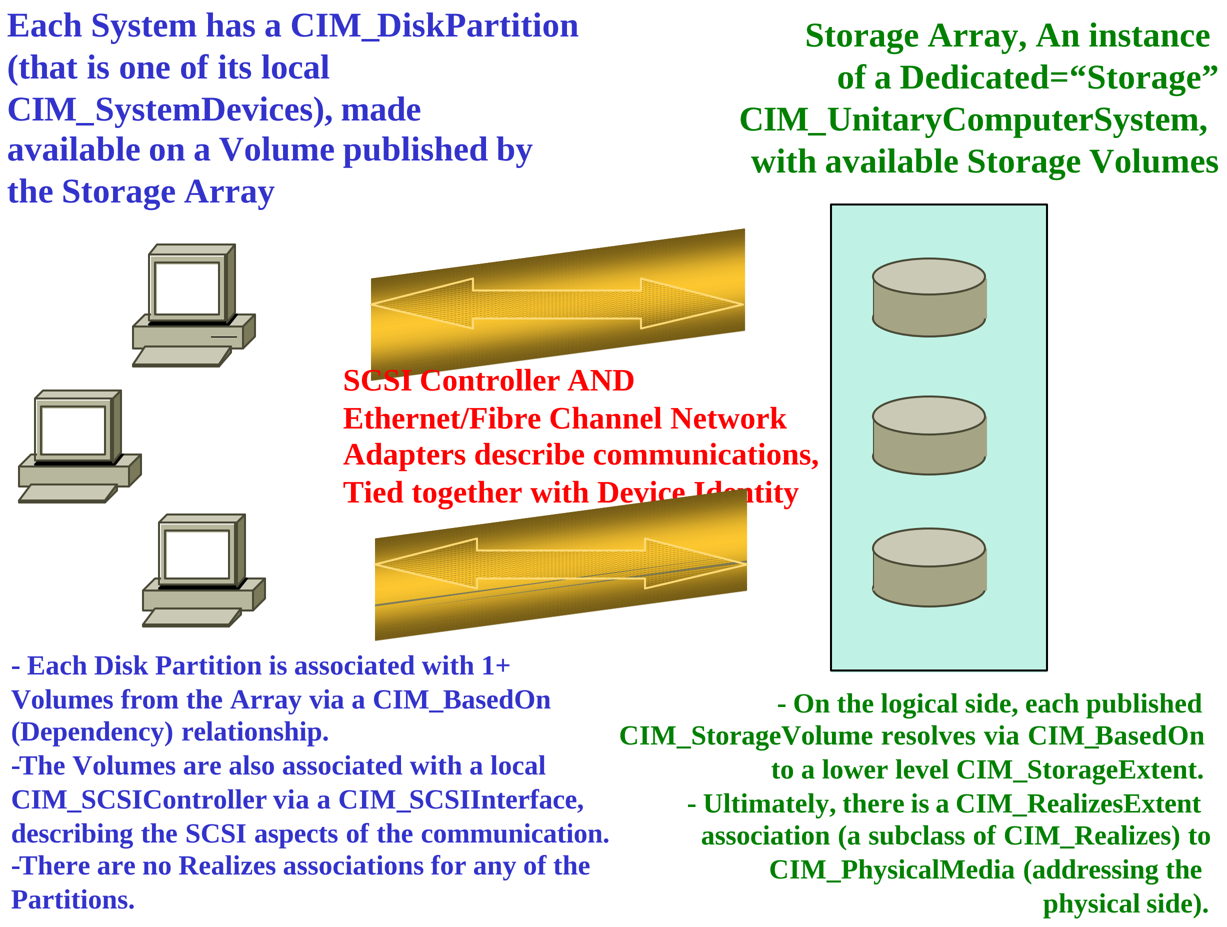

Each System has a CIM_DiskPartition (that is one of its local CIM_SystemDevices), made available on a Volume published by the Storage Array
Storage Array, An instance of a Dedicated="Storage" CIM_UnitaryComputerSystem, with available Storage Volumes
SCSI Controller AND Ethernet/Fibre Channel Network Adapters describe communications, Tied together with Device Identity
- Each Disk Partition is associated with 1+ Volumes from the Array via a CIM_BasedOn (Dependency) relationship.
-The Volumes are also associated with a local CIM_SCSIController via a CIM_SCSIInterface, describing the SCSI aspects of the communication.
-There are no Realizes associations for any of the Partitions.
- On the logical side, each published CIM_StorageVolume resolves via CIM_BasedOn to a lower level CIM_StorageExtent.
- Ultimately, there is a CIM_RealizesExtent association (a subclass of CIM_Realizes) to CIM_PhysicalMedia (addressing the physical side).

Figure 7. Example of Remote Devices

As shown in Section 9, Figure 4, Logical Device has many properties and methods. A brief overview of each is provided here:

- System Creation Class Name and System Name - The scoping System's keys

- Creation Class Name and Device ID - Device-specific identification

- Power Management Supported - A boolean indicating that power management is supported for the Device

- Power Management Capabilities - An array of enumerated integer values, where the permissible values are:

  Unknown
  Not Supported
  Disabled
  Enabled - Indicates that power management is known to be enabled, but the exact feature set is not known
  Power Saving Modes Entered Automatically (based on Device usage or other criteria)
  Power State Settable (using the Set Power State method on Logical Device)
  Power Cycling Supported (using the Set Power State method) - Power cycling is a particular value of one of the input parameters of the Set Power State method
  Timed Power On Supported (using the Set Power State method and "power cycling") - Specifies that power should be reapplied at a particular point in the future. This is a fairly advanced power management capability that is not routinely supported by a Device.

- Availability - An enumeration characterizing the status of the Logical Device, expanding on the Status information inherited from Managed System Element. Its values are fairly explanatory and are specified as follows:

  Other
  Unknown
  Running/Full Power
  Warning
  In Test
  Not Applicable
  Power Off
  Off Line
  Off Duty
  Degraded
  Not Installed (Device was known to have existed on the System, but has been "un-installed" - OR - the System has been configured to support a specific Device but it has not yet been installed)
  Install Error (Device is installed but experienced an error during the installation process)
  Power Save - Unknown (Unknown indicates that the current power state of the Device is unknown, but it is currently in a reduced power mode of operation)
  Power Save - Low Power Mode (Low Power Mode indicates that the Device is still functioning, but performance may be degraded)
  Power Save - Standby (Standby indicates that the Device is NOT functioning but could be brought to a full power state more quickly than starting from boot)
  Power Cycle

Power Save - Warning (Warning indicates that the Device is in a reduced power mode of operation, but some sort of error or warning has occurred. The Device should be commanded back to a full power mode in order to diagnose the problem.)
Paused
Not Ready
Not Configured
Quiesced (The Device is functioning but not currently processing requests or performing its normal function)

- Additional Availability - An array of enumerated values providing additional "Availability" data. The permissible values for this property are equivalent to those specified for the Availability property. The reason for this property is that Availability is defined as a single-valued property. As the Availability property and its enumeration evolved, it became apparent that more than one "Availability" status could be applicable to a Device. So, the Additional Availability array was added to the Logical Device class.

- Status Info - An enumeration indicating whether the Device is in an enabled, disabled or unknown state

- Last Error Code (a uint32), Error Description (a string) and Error Cleared (a boolean) - Error information for the Device (Further detail and error counters are defined in the DeviceErrorCounts class in the CIM Device Model.)

- Power On Hours and Total Power On Hours - Operational hours

- Other Identifying Info - A string array that can be used for additional data, beyond that stored in the Device ID property, to name and identify a Logical Device. One example where this is used occurs when both an operating system's device identification and a device internal ID exist. One's value is chosen for the Device ID property and the other's data can be stored in the Other Identifying Info array.

- Identifying Descriptions - A string array that has a *ModelCorrespondence* with Other Identifying Info. There is a correspondence between the entries at the same index in both arrays. The Description strings provide explanations and details behind the values in the Other Identifying Info array.

- Max Quiesce Time - Maximum time in milliseconds, that a Device can run in a "Quiesced" state. What occurs at the end of the time limit is device-specific. The Device may unquiesce, may offline or take other action. A value of 0 indicates that a Device can remain quiesced indefinitely.

- Reset method

- Set Power State - A method that takes two parameters, the power state to be set and the date/time when the set should take effect

- Enable Device - A method indicating whether the Device should be enabled or disabled

- Online Device - A method indicating whether the Device should be onlined or offlined

- Quiesce Device - A method indicating whether the Device should be quiesced or returned to operation. A request to Quiesce means that the Device should cleanly cease all current activity. Once quiesced, a Device may be offlined for diagnostics, or disabled for power off and hot swap.

- Save Properties - A method requesting that a Device capture its current configuration, setup and/or state information in some backing store. The goal would be to use this information at a later time (via the Restore Properties method) to return a Device to its "present" condition.

- Restore Properties - A method requesting a Device to re-establish its configuration, setup and/or state information from a backing store.

All the different Availability and Status states of a Logical Device can be confusing and should be positioned relative to the others. The following paragraph attempts to do this …

If a Device is "Enabled" (indicated by a Status Info value of 3), it has been powered up, and is configured and operational. The Device may or may not be functionally active, depending on whether its Availability (or Additional Availability) indicates that it is "Running/Full Power", "Offline" or "Quiesced". In an enabled but offline mode, a Device may be performing out-of-band requests, such as running Diagnostics. It only makes sense to quiesce a Device that is "Running/Full Power" and "Enabled". If a Device is "Disabled" (indicated by a Status Info value of 4), a Device can only be "enabled" or powered off. In a personal computer environment, "Disabled" means that the Device's driver is not available in the stack. In other environments, a Device can be disabled by removing its configuration file. A disabled device is physically present in a System and consuming resources, but cannot be communicated with until a load of a driver, a load of a configuration file, or some other "enabling" activity has occurred.

14. Properties and Associations of Services and Their Access Points

The CIM_Service class represents the configuration, operational data and management of "function." The semantics are not about the function itself, but information needed to manage it. For example, although you might have "email" or "word processing" services on your computer, you would not use CIM to handle your individual mail messages, or open and edit documents and files. You would use the native utilities and software of your computer system. These are the operating and executing entities that implement Services. CIM is used to describe the existence of email and word processing Services, to configure them, and to diagnose them if a problem occurs.

Note: Where specific functionality and behavior are needed for management, these are modeled as methods on the various classes in the CIM hierarchy.

Designed as a complementary class to Service, CIM_ServiceAccessPoint models the utilization and invocation of a Service. It represents that a Service is made available for use by other entities. Access Points are not the APIs, DLLs, OS commands, … to invoke Services (these are actually the software implementations of Services), but the abstraction of access to a Service.

One could distinguish Service and Service Access Points (SAPs) in the context of a “provider-consumer” relationship. Service represents the current configuration and operational data of the *provided* function, and Service Access Points are the way to manage the *consumption* (or access) of that function. In client-server terms, Service is the function at the server, while the SAPs are the management of a client's use of the Service.

Service represents the management of any kind of function and is a very abstract concept. For example, on a personal computer, a wide variety of Services may be running - word processing, presentation preparation, email client, meeting scheduling (including a local/offline store) and much more. In addition, when specific Services are not locally available, they can often be accessed using the network. Access could be modeled as instances of Service Access Points. Examples of the latter are SAPs to pull email from a server, to print at a network printer, or to pull the latest meetings from the corporate meeting database. In all of these examples, there is actually a layering or dependency of Access Points. The application oriented ones are dependent on network access. Network access is modeled as instances and subclasses of Protocol Endpoints, which are in turn subclasses of Service Access Points, in the CIM Network Model. The following figure illustrates these concepts.

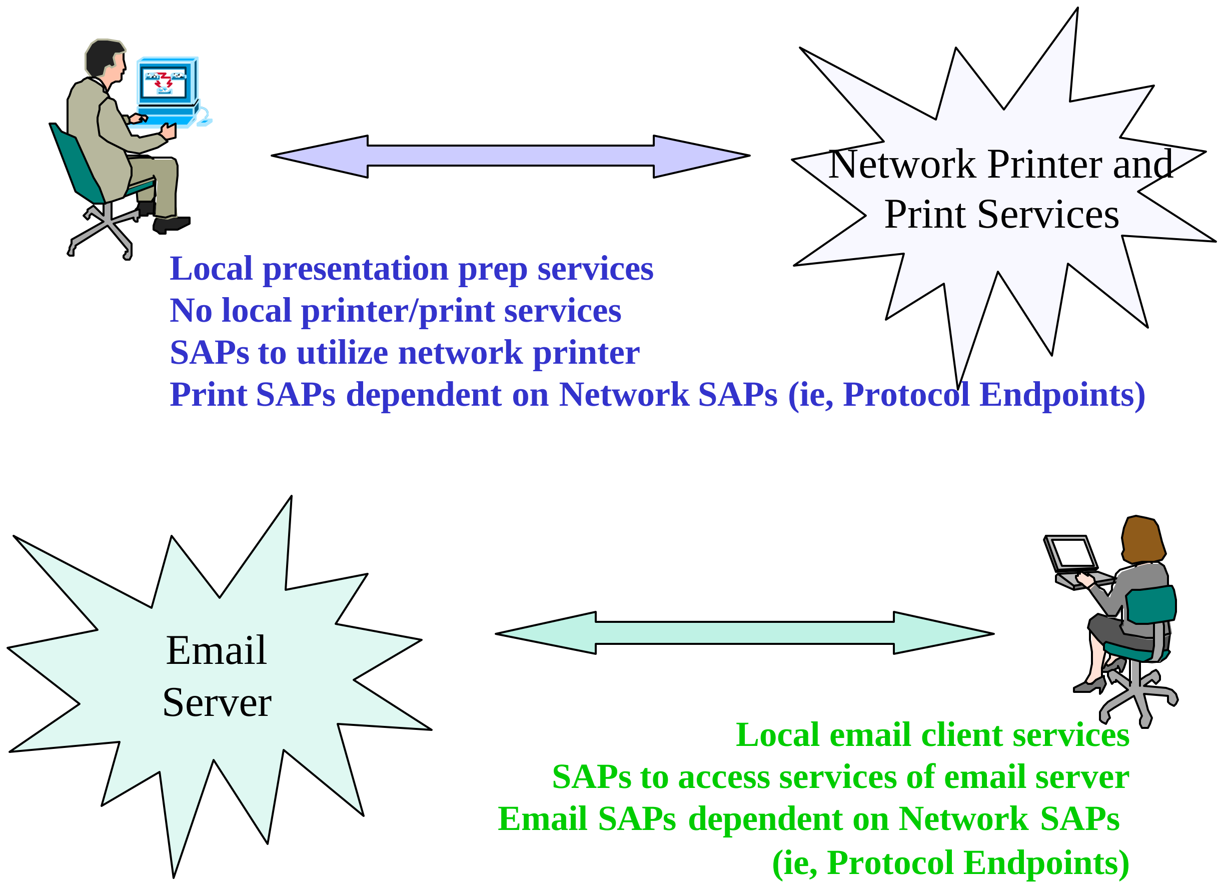

Network Printer and Print Services
Local presentation prep services
No local printer/print services
SAPs to utilize network printer
Print SAPs dependent on Network SAPs (ie, Protocol Endpoints)
Email Server
Local email client services
SAPs to access services of email server
Email SAPs dependent on Network SAPs (ie, Protocol Endpoints)

Figure 8. Services and SAPs in Operation

The CIM Schema attempts to distinguish between the configuration and operational data of a Service and SAP, and the implementations behind these features. For example, the configurations of a Printer Device and its supporting software/device driver are different than the configuration of a Network Print Service and its access from client computers.

Services and Service Access Points are not the Devices and/or software that implement them. The latter are modeled as subclasses and instances of Logical Devices and Software Features/Software Elements, and are associated with Services and SAPs using the following relationships:

- Device Service Implementation and Device SAP Implementation (defined in the CIM Device Model)

- Software Feature Service Implementation and Software Feature SAP Implementation (defined in the CIM Application Model)

- Software Element Service Implementation and Software Element SAP Implementation (defined in the CIM Application Model)

The last two sets of associations are very similar and would be redundant if both were implemented. The Application Model, in fact, states that one approach must be chosen over the other. Either the implementation of a Service or SAP is based on the definition of Software Features (the functions and capabilities of products) or Software Elements (individually deployable and manageable software components - for example, files). The redundancy comes from the fact that Software Features decompose into Software Elements.

Association to Software Elements (versus Software Features) would be chosen when greater granularity is needed. When a Software Feature is fully deployed, it can result in many executable Elements. These Software Elements may each implement different Services. The individual Service implementations could be conveyed using the Software Element-XXX-Implementation associations. These relationships are especially important when Software Feature and Product aspects are not instrumented for a piece of software (i.e., when the acquisition and deployment of the software is not detailed). In this case, the Software Element-XXX-Implementation associations are the only ones available.

One additional clarification is needed regarding Service implementation … It was noted in Section 13 on Logical Devices, that Devices have a distinct function. This function is fairly inseparable from the identity of the Device (as a Printer, as a Keyboard, as a Fan, etc.) Because they are inseparable, there is no need to have another Service class (for example, "FanService") describing Device function. The question then arises as to the purpose and semantics of the Device Service Implementation association. This relationship describes the dependencies of a Service on operating hardware. It states that *this* Device is involved in the implementation of *this* Service. For example, to have Print Services (to actually print a document), it is absolutely necessary to have a CIM_Printer. And, as mentioned above, there is a distinction between Print Services and a Printer Device - "the configurations of a Printer Device and its supporting software/device driver are different than the configuration of a Network Print Service and its access from client computers." Device Service Implementation models the dependency of the Print Service on the Printer, not vice versa.

Given this introduction, Figure 9 shows the Service-related aspects of the Core Model.

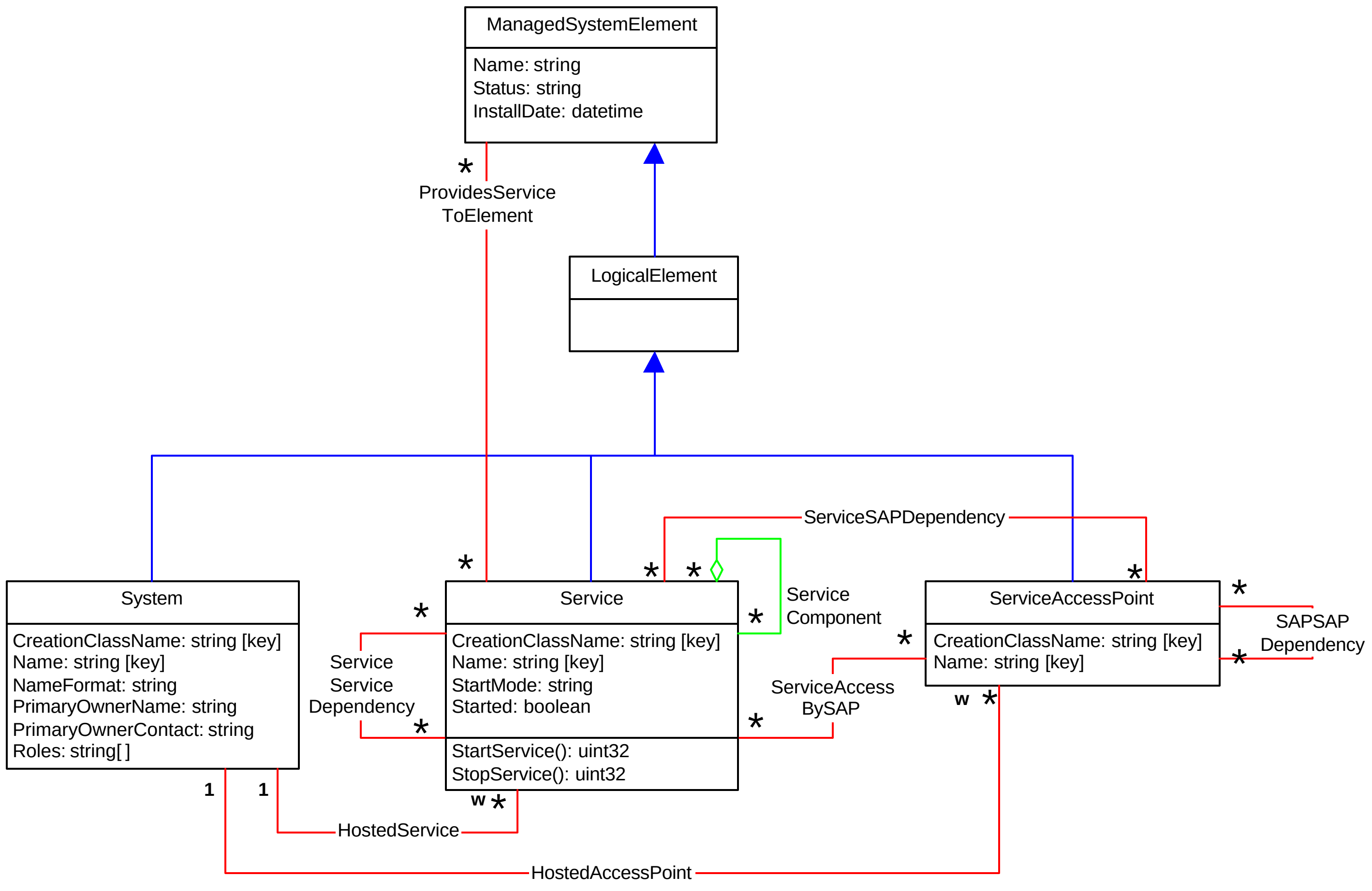


Figure 9. The CIM Service/SAP Classes

Overviewing the properties of a Service object …

- System Creation Class Name and System Name are the hosting (scoping) System's keys. These provide the context or locale in which the Service is provided.
- Creation Class Name and Name are the Service class' individual *Key* properties.
- Start Mode is an enumerated string value indicating whether the Service is automatically started by a System, Operating System, etc. or only started upon request.
- Started is a boolean indicating whether the Service is currently running/has been started (TRUE), or stopped (FALSE).
- Start Service and Stop Service are fairly straightforward methods.

The properties of Service Access Points are only the hosting System’s keys (System Creation Class Name and System Name) and the Creation Class Name and Name of the SAP.

One observation is that a large number of associations exist. This “problem” (or “opportunity”) can be seen throughout the CIM Schema, but most predominantly here. It is important to remember that each of the associations reflects a different semantic. The defined associations can be grouped as follows:

- Provider-Consumer relationships (1) – Service Access By SAP. This association ties an Access Point (the means of consuming or accessing a Service) with its related Service(s).

- Hosting relationships (2) – Hosted Service and Hosted Access Point. Services and SAPs exist within the context of a System, typically where their Device and software implementations are also installed and hosted. These objects are associated at the abstract level of System, rather than to a Computer System, allowing Services to exist in distributed components such as Application or Network Systems.

- Dependencies (4) – Provides Service To Element, Service Service Dependency, Service SAP Dependency and SAP SAP Dependency. The first association, Provides Service To Element, describes the general dependency of a Managed System Element on the functionality provided by a Service. The second association, Service Service Dependency, is actually a subclass of Provides Service To Element. It indicates that a Service depends on other Services having executed, being co-resident or being absent, in order to provide their own functionality. (The nature of the dependency is described using a property of the association, Type Of Dependency, which is an enumerated unsigned integer. Type Of Dependency indicates whether the “other” Service must be completed, started or not started.) Moving to the third association, Service SAP Dependency models the scenario where a Service uses the access points of another Service. For example, Boot Services may be dependent on underlying BIOS disk and initialization Services. In the case of the initialization Services, the Boot Service is dependent on the init Services having completed (Service Service Dependency). But, for the BIOS disk Services, Boot Services may actually utilize the SAPs of the disk Services – modeled as an instance of the Service SAP Dependency. In addition, a SAP may be dependent on another, underlying Service (for example, for connectivity), and utilizes the Access Points of that Service. This relationship is modeled using the SAP SAP Dependency association.

- Component relationships (1) - Service Component. This association describes that subordinate services are aggregated to form a higher level Service.

- Implementation (6, as discussed above) – Software Feature Service Implementation, Software Feature SAP Implementation, Software Element Service Implementation, Software Element SAP Implementation, Device Service Implementation and Device SAP Implementation. These associations reflect the implementations behind the functionality or access to the functionality.

Note that the Core Model does not represent or allow a single Service to be hosted on multiple Systems (for example, a Print Service hosted by a networked Computer System and a "dedicated" Network Printer). If individual Services are grouped together to create a larger entity, this should be modeled as an Application System. (The Application System class is explained in more detail in the CIM Application Model.) An Application System can act as aggregation point for Services, where each Service is located on a single hosting System.

15. Products and FRUs

Product is a concrete class that represents a collection of Software Features and Physical Elements (and possibly other Products), that is acquired and/or deployed as a unit. Acquisition implies an agreement between a supplier and the consumer, which may have implications to licensing, support and warranty. The Product class is intended to cover any form of acquisition, including deployment of internally generated applications, even though no actual purchase is involved.

FRU is an acronym for "field replaceable unit". It is a vendor-defined collection of Products, Physical Elements and/or Software Features. FRUs "replace" and are associated with a Product (via the Product FRU relationship). Their purpose is to maintain or upgrade the related Product.

Figure 10 represents the Product/FRU classes in the Core Model:

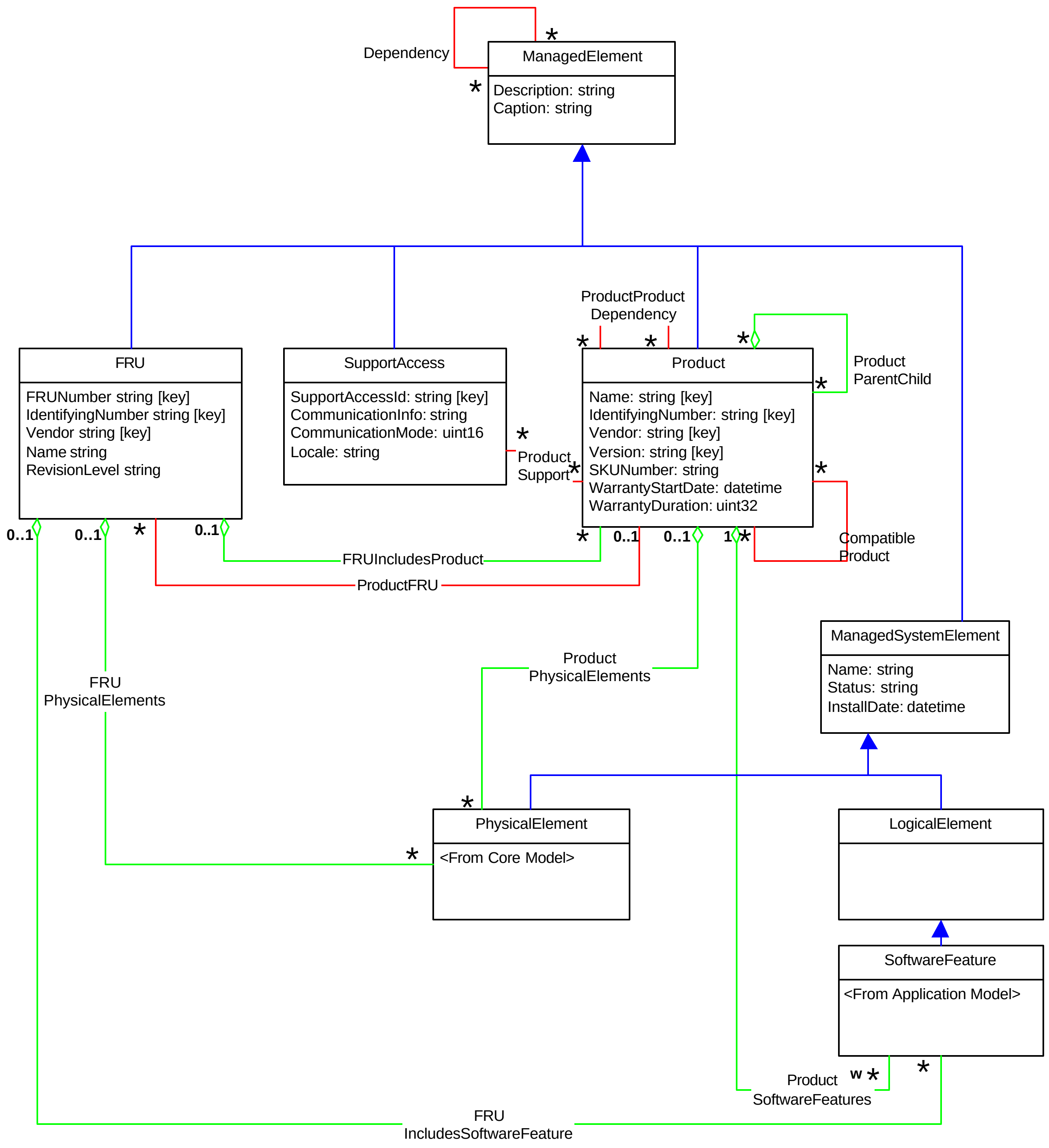


Figure 10. The Product/FRU Object Hierarchy

In reviewing what is acquired as a Product, hardware is purchased (ie, Physical Elements), as well as software and sometimes other Products (bundled with the originally purchased Product). Therefore, the “components” of a Product are defined using the associations, Product Physical Elements, Product Software Features and Product Parent Child. The latter models the fact that one Product can incorporate or bundle other “sub”-Products.

The bond of Product to software is so strong that Software Features are actually weak to (scoped by) Products, as defined by the Product Software Features association. For more information on this association, refer to the CIM Application Model.

Several additional Product-related associations remain to be discussed. They are: Compatible Products and Product Product Dependency. The latter is very similar to the Service Service Dependency relationship discussed in the previous section. It indicates that a Product depends on other Products being installed or being absent. The nature of the dependency is described using a property of the association, Type Of Dependency, which is an enumerated unsigned integer. Type Of Dependency indicates whether the “other” Product must or must not be installed.

The Compatible Product association is designed to convey a wide variety of information. Quoting from the Core MOF, "For example, it can indicate that the two referenced Products interoperate, that they can be installed together, that one can be the physical container for the other, etc. The string property, Compatibility Description, defines how the Products interoperate or are compatible, any limitations regarding interoperability or installation, …"

Similar to Product, a FRU is composed of hardware (Physical Elements), software (Software Features) or other Products. The association of Physical Elements to a FRU is specified using the FRU Physical Elements relationship. The association of component Products to a FRU is specified via the FRU Includes Product relationship. And, the association of Software Features to FRUs is conveyed using the FRU Includes Software Feature association.

At first glance, there appears to be a bit of overlap and redundancy with respect to FRUs including Products and also including Software Features, since Features are weak to Products. Couldn't the same info be conveyed by only having FRUs aggregate Products - and then the Products carry along their Software Features? The answer would be yes if all there were no levels of granularity with respect to Software Features being included in a FRU - i.e., if *all* Features in a Product were always in a FRU. However, this does not often happen. When something is FRUed, it is likely to ship with only a subset of a Product's Features (the ones that are applicable to the thing being replaced). This is the reason that the FRU Includes Software Feature association exists. Software Features are still weak to Products, but not all the Features of a Product may be in a FRU. Said another way, it is not required that an entire Product be associated with a FRU just to indicate that new software is also shipped.

Briefly reviewing a Product’s properties, they are:

- Vendor, Name, Identifying Number and Version – Which combine to define a Product’s key structure. Identifying Number may be a serial number or perhaps a die number on a chip.
- Caption and Description (Inherited from Managed Element)
- SKU Number (where SKU refers to a “stock keeping unit” number)
- Warranty Start Date and Warranty Duration (in days)

The FRU object's properties are:

- Vendor, FRU Number (ordering information) and Identifying Number – Which combine to define a FRU's key structure
- Name
- Caption and Description (Inherited from Managed Element, where Description is overridden to include information from *MappingStrings*)
- Revision Level

One of the predominant reasons for creating the Product class was to reflect how support is obtained. For this reason, the Support Access class was defined. Its properties are:

- Support Access ID – The key for the class.
- Caption and Description (Inherited from Managed Element, where Description is overridden to include information from *MappingStrings*)
- Locale – A string describing the geographic region or language dialect to which this Support applies
- Communication Mode – An enumeration defining whether the support is via BBS, URL, fax, phone, e-mail, etc.
- Communication Info – A string providing more detail related to the Communication Mode. For example, if support is provided via fax, then the fax number should be listed in the Communication Info property.

Instances of the Support Access class are associated with Products using the Product Support relationship.

Products are purchased for installation and subsequent use. Installing the Physical Elements and supporting software of a "hardware" Product typically results in the creation or upgrade of a System object (for example, a Computer System), or a Logical Device (for example, a Modem). Products do not "contain" these Logical Elements, but the Elements are result of the installation of the Products' components.

16. Settings and Configurations

This section summarizes the concepts behind the CIM_Setting and Configuration classes, and suggests guidelines for their use. Figure 11 shows the related classes of the CIM Core Model, as well as the Collection classes since they are referenced in associations.

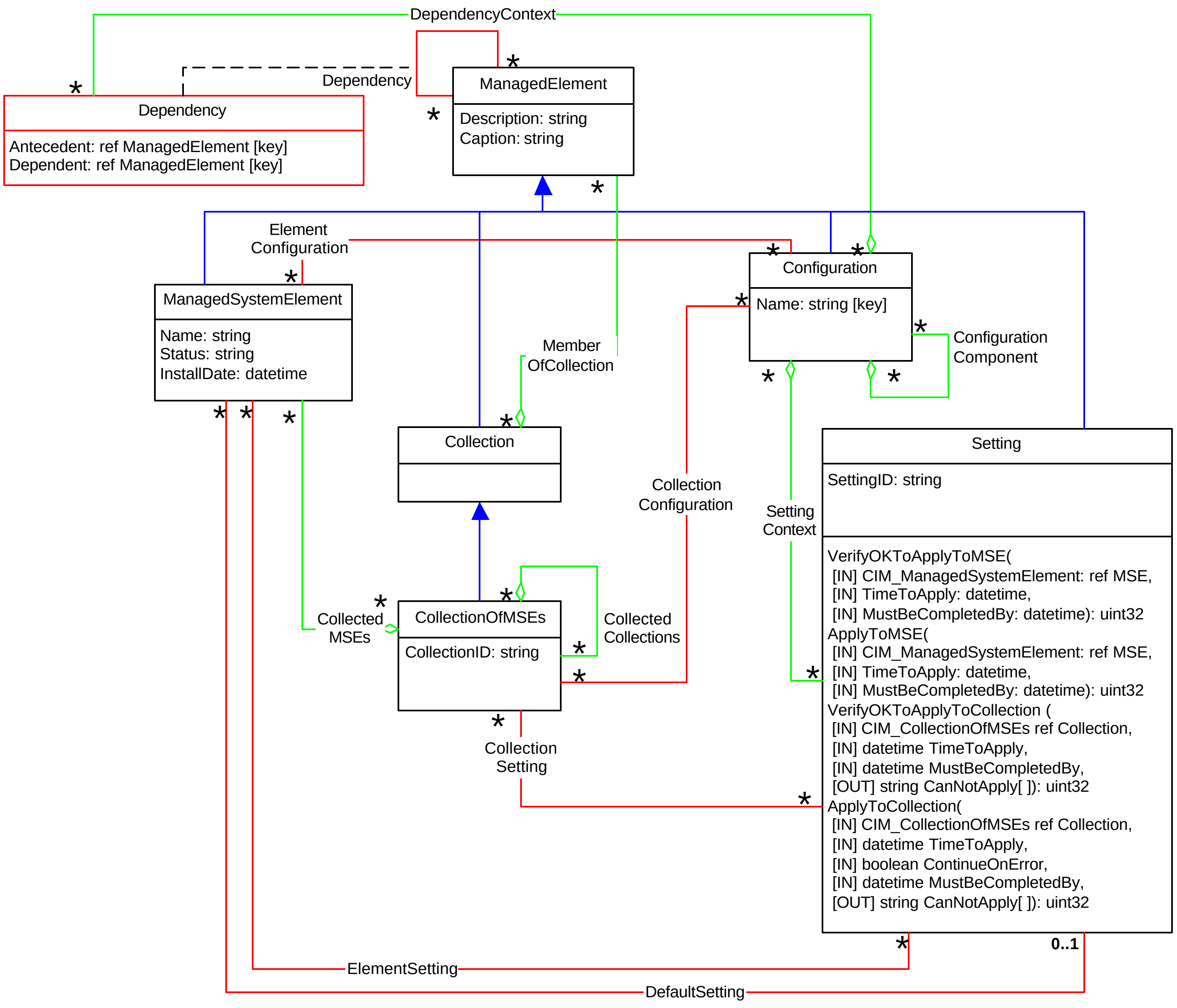

DependencyContext
Dependency
ManagedElement
Description: string
Caption: string
Dependency
Antecedent: ref ManagedElement [key]
Dependent: ref ManagedElement [key]
Element Configuration
ManagedSystemElement
Name: string
Status: string
InstallDate: datetime
Configuration
Name: string [key]
Configuration Component
Member OfCollection
Collection
Collection Configuration
Setting Context
Setting
SettingID: string
VerifyOKToApplyToMSE(
[IN] CIM_ManagedSystemElement: ref MSE,
[IN] TimeToApply: datetime,
[IN] MustBeCompletedBy: datetime): uint32
ApplyToMSE(
[IN] CIM_ManagedSystemElement: ref MSE,
[IN] TimeToApply: datetime,
[IN] MustBeCompletedBy: datetime): uint32
VerifyOKToApplyToCollection (
[IN] CIM_CollectionOfMSEs ref Collection,
[IN] datetime TimeToApply,
[IN] datetime MustBeCompletedBy,
[OUT] string CanNotApply[ ]): uint32
ApplyToCollection(
[IN] CIM_CollectionOfMSEs ref Collection,
[IN] datetime TimeToApply,
[IN] boolean ContinueOnError,
[IN] datetime MustBeCompletedBy,
[OUT] string CanNotApply[ ]): uint32
Collected MSEs
CollectionOfMSEs
CollectionID: string
Collected Collections
Collection Setting
ElementSetting
DefaultSetting
0..1

Figure 11. Setting, Configuration and Collection Classes

As background, this discussion assumes that a System (or one of its components) is modeled using some or all of the following information:

- Current operational state, including "configuration in use"
- Potential configuration
- Potential configuration to be applied "next"
- Configuration last applied

Settings and Configurations are useful to capture some of the information above. Settings define specific, pre-configured parameter data to be "applied" (loosely transactionally) to one or more Managed System Elements. They are very much tied to the properties of existing objects through the Element Setting association. Configurations aggregate Settings and Dependencies, representing a certain behavior or desired functional state for Managed System Elements.

Specifically addressing the 4 sets of information above:

- Potential configuration reflects a predefined set of parameters that are applied or set together. This is either an instance of CIM_Setting (which groups properties/parameters), or an aggregation of CIM_Settings (which is an instance of CIM_Configuration). Settings are aggregated into a Configuration via the Setting Context association. Settings and Configurations are tied to the Managed System Elements to which they apply, using the Element Setting and Element Configuration relationships, respectively. Also, they may be tied to collections of Elements via the Collection Configuration and Collection Setting relationships.

  Note: Two methods are defined on the Setting class related to applying a Setting to a Managed System Element, or a Collection of MSEs. Also, two methods are available to first "Verify" that it is possible and/or acceptable to apply the Setting.

- The configuration to be applied "next" is not currently addressed by CIM (Other than to invoke one of the Apply methods and specify a date/time "ToApply" as one of the method parameters)
- "Configuration last applied" could be handled via a new property, LastApplied, on the Element Setting association or a new association between a Setting and Managed System Element (CIM_CurrentSetting?)
- Current state is defined by an instance of a class and/or its associations. In today's CIM Schema, properties are duplicated in a CIM_Setting to describe what could be or what was set for a Managed System Element. Operational data does not go into the Setting object's properties - since the "specific, pre-configured parameter data" currently stored there would be overwritten and lost. Operational changes (due to policy, operator intervention or real world constraints) affect operational data - Settings are not in this realm. An example is defining a Setting.ModemSpeed property = 56K (a constant value), but putting current speed in the CIM_Modem.Speed property (which might currently be reading 48.2K).

It may be possible to avoid some property replication by relating an instance of a class to its "current" Setting with a new association (as above, CIM_CurrentSetting?). However, to do this and not replicate properties would imply that the instance's operational values never differ from the values specified in the Setting. Otherwise, if values in operation can differ from the Setting values (ie, the Modem example), you need to replicate the properties.

The best uses of Settings and Configurations are:

1. To define groups of parameters/properties and the values of these properties - where the values are predetermined, are interrelated, and/or should be set together

2. To define groups of parameters/properties which may simultaneously apply to multiple objects

3. To define groups of parameters/properties which will be re-applied to one or more Managed System Elements - at a specific time, given specific environmental or operational conditions, or on operator/policy request

4. To aid in the creation/deletion of objects and/or definition of new function. This is best accomplished via methods on subclasses of Managed System Elements (especially CIM_Service), which may have some of their data fed through Settings and Configurations.

Settings and Configurations are not appropriate to:

1. Write a property value to an instance (just use the HTTP ModifyInstance operation)

2. Hold current operational data that is changeable

3. Do "what if" scenarios across objects (create and populate a new namespace instead)

A few associations in Figure 11 have not yet been addressed in the explanation of Settings and Configurations - Default Setting, Configuration Component and Dependency Context. The first two associations are very straightforward. The first allows the specification of the "default" Setting for a Managed System Element. The second describes a layering of Configurations - building higher-level configurations from lower level ones. Configuration Component enables the assembly of complex configurations by grouping together simpler ones. The last association, Dependency Context, is a little more complex. It indicates that a Configuration aggregates Settings AND Dependencies for Managed System Elements. It implies that that the referenced Dependency should be in place in order for the application of the Configuration and its aggregated Settings to succeed.

For example, to connect to a Mail System from “home”, a Dependency on a Modem and security software exists. However, a Dependency on an Ethernet Adapter exists at “work”. Setup and configuration data for the pertinent Logical Devices (in this example, a CIM_Modem and CIM_EthernetAdapter) are defined in individual Setting objects. Two Configuration objects can be instantiated (“John at Home” and “John at Work”) – representing the different Device and software setups, operational characteristics and dependencies of these two usage scenarios.

As a usage example of Setting, consider the following MOF segment from the CIM Device Model:

```
// ==================================================
// MonitorResolution
// ==================================================
  [Description (
     "MonitorResolution describes the relationship between "
     "horizontal and vertical resolutions, refresh rate and scan "
     "mode for a DesktopMonitor. The actual resolutions, etc. that "
     "are in use, are the values specified in the VideoController "
     "object.")
  ]
class CIM_MonitorResolution : CIM_Setting
```

```
{
    [Override ("SettingID"),
     Key, MaxLen (256),
     Description (
          "The inherited SettingID serves as part of the key for a "
          "MonitorResolution instance.")
    ]
  string SettingID;
    [Description ("Monitor's horizontal resolution in Pixels."),
     Units ("Pixels"),
        ModelCorrespondence {
            "CIM_VideoController.CurrentHorizontalResolution"},
     MappingStrings {"MIF.DMTF|Monitor Resolutions|002.2"}
    ]
  uint32 HorizontalResolution;
    [Description ("Monitor's vertical resolution in Pixels."),
     Units ("Pixels"),
        ModelCorrespondence {
            "CIM_VideoController.CurrentVerticalResolution"},
     MappingStrings {"MIF.DMTF|Monitor Resolutions|002.3"}
    ]
  uint32 VerticalResolution;
    [Description (
          "Monitor's refresh rate in Hertz. If a range of rates is "
          "supported, use the MinRefreshRate and MaxRefreshRate "
          "properties, and set RefreshRate (this property) to 0."),
     Units ("Hertz"),
        ModelCorrespondence {
            "CIM_VideoController.CurrentRefreshRate"},
     MappingStrings {"MIF.DMTF|Monitor Resolutions|002.4"}
    ]
  uint32 RefreshRate;
    [Description (
          "Monitor's minimum refresh rate in Hertz, when a range of "
          "rates is supported at the specified resolutions."),
     Units ("Hertz"),
        ModelCorrespondence {
            "CIM_VideoController.MinRefreshRate"},
     MappingStrings {"MIF.DMTF|Monitor Resolutions|002.6"}
    ]
  uint32 MinRefreshRate;
    [Description (
          "Monitor's maximum refresh rate in Hertz, when a range of "
          "rates is supported at the specified resolutions."),
     Units ("Hertz"),
        ModelCorrespondence {
            "CIM_VideoController.MaxRefreshRate"},
     MappingStrings {"MIF.DMTF|Monitor Resolutions|002.7"}
    ]
  uint32 MaxRefreshRate;
    [Description (
          "Integer indicating whether the monitor operates in "
```

```
            "interlaced or non-interlaced mode. Values are: "
            "1=\"Other\", 2=\"Unknown\", 3=\"Not Supported\", "
        "4=\"Non-Interlaced Operation\" and 5=\"Interlaced "
            "Operation\"."),
           ValueMap {"1", "2", "3", "4", "5"},
       Values {"Other", "Unknown", "Not Supported",
               "Non-Interlaced Operation", "Interlaced Operation"},
           ModelCorrespondence {
               "CIM_VideoController.CurrentScanMode"},
       MappingStrings {"MIF.DMTF|Monitor Resolutions|002.5"}
      ]
    uint16 ScanMode;
  };
```

This MOF segment describes a subclass of Setting (Monitor Resolution) that specifies valid configuration options for a Desktop Monitor. The referenced Monitor is associated with the Monitor Resolution object by a subclass of the Element Setting relationship. The various, possible Monitor configurations would be defined as individual instantiations of the Monitor Resolution object. If one wished to “apply” one of these Settings to the CIM Schema, the Model Correspondence qualifier indicates which CIM properties are affected by the Monitor Resolution information (i.e., the Video Controller's properties that output to the Monitor).

One last aspect of the Setting/Configuration Model must be explained. This is the fact that Configuration objects have keys (and are “concrete” classes) versus Setting objects (which do not have keys and are “abstract” classes). A concrete class is one that can be instantiated. An abstract class is very conceptual, and must be subclassed in order to be reasonably instantiated - i.e., to add properties, define further associations, etc.

As currently defined, the Setting object is very conceptual. It only has three properties:

- SettingID
- Description and Caption (Inherited from Managed Element)

There is no parameter information defined at this level of the Core Model, for the Setting object. It would not make sense to define such data here – since it would vary greatly by subclass. Hence, the Setting object is “abstract” and must be subclassed in order to define a key, define any scoping information (i.e., weakness and *Propagated* keys), define its parametric data and instantiate it. This is what was done in the Monitor Resolution subclass, used as an example above.

The Monitor Resolution class could reasonably be instantiated. The SettingID property is overridden to make it a key property for the class. And, since the “abstract” qualifier is not defined, Monitor Resolution is a concrete class.

Let's examine the reasoning why Monitor Resolution is not Weak to any other CIM class. The thinking was that the class' instance values were not unique to (scoped by) a single instance of a Desktop Monitor. For example, it is possible to create single instances of the Monitor Resolution class describing the valid combinations of parameters for an Acme XYZ Monitor. These Settings of valid parameter values may then be associated with all instances of CIM_DesktopMonitor that are Acme XYZ monitors. The Setting would not be weak to a single instance. On the other hand, it may be reasonable to define subclasses of CIM_Setting that are weak to Systems, Devices, Services, etc. These subclasses would indicate that a particular instance of the class holds the Settings/parameters for a particular System/Device/Service/etc., and no other.

Configuration objects do not require additional properties in order to be instantiated. They represent aggregations only – although additional properties and associations can be defined in subclasses. Hence, the Configuration object has a key (its Name property) and is a concrete class (can be instantiated).

In looking back on this decision, however, it seems that Configurations may indeed exhibit "weak"ness - similar to Settings. Future versions of the CIM Schema may specify peer Configuration classes that have scoped (*Weak)* associations to various Managed System Elements.

16.1 Q and A on Settings and Configurations

1. [Q] Should all configurable attributes within the model have corresponding Setting classes?

[A] No, some properties may just be WRITEable. However, preconfigured/predefined data and properties that should be set together are reasonable candidates for Settings. Also, data that would be configured via policy and then "applied" are also candidates.

2. [Q] The Managed System Element to which to apply a Setting is an argument to the ApplyToMSE method of Setting. How is this related to the Setting's relationship to MSEs via the CIM_ElementSetting association? Is there an instance of CIM_ElementSetting relating a given CIM_Setting instance with each CIM_MSE to which it has been applied?

[A] You may require that you have an ElementSetting relationship in place before applying a Setting to a specified Managed System Element. However, this association is not mandatory. If a Setting is applicable to ALL video controllers, then this could be defined without creating the relatively useless ElementSetting relationship to ALL video controllers.

3. [Q] In usual OO modeling, the interface to change an object is usually part of the interface definition of the object itself. The setting model changes this and moves the interface into a separate object associated with the target object only though an association (and a loose association at that when viewed as part of a configuration). How is this problem addressed?

[A] Is this a "set" method for a parameter? If so, it is covered by CIM's ModifyInstance HTTP method and making a specific property WRITEable. Settings define specific, pre-configured parameter data to be "applied" together.

ElementSetting is not a "loose association" regardless of whether a Setting is part of a Configuration or not. The association is still specific, tying a Setting to a Managed System Element. In fact, a subclass of Setting may even be weak to a specific component or system - likely defined by a subclass of ElementSetting that uses the Weak qualifier.

4. [Q] It will be typical in making changes to systems that the actual changes to detailed properties of the objects effected will not be explicitly stated but rather the general desired target state of the system will be stated and the details are left to the object implementations (such as "Add Vlan xyz to this domain"). Many switches, ports on switches, etc. will be changed. It is inappropriate to require that the change request indicate what all of these changes are. How do settings and configurations address this issue?

[A] They don't. Settings and Configurations alone should not be used to "add VLAN xyz". In this example, you are creating many new instances and associations. Settings and

Configurations are closely tied to current instances of Managed System Elements, but could be used to describe the "general desired target state" and/or specific configuration parameters that influence the implementation.

5. [Q] Neither Setting or Configuration is bound to a System. In fact, since Configuration has a key, it cannot be bound to a System. Therefore, you have no way to ensure that the Configuration goes to the right entity.

[A] The reason that Settings and Configurations are not bound to Systems is that they may only apply to specific components of Systems (for example, only applying to a Video Controller). We did not want to tie the definition to the very high level System, when we could tie a specific subclass to a specific component.

Checking that you "apply" to the right entity, is checking that you have a valid ElementSetting. However, as mentioned above (#3), this association is not mandatory.

6. [Q] Why is there a DefaultSetting but no DefaultConfiguration?

[A] This is certainly reasonable to define and could be added with no change in semantics.

7. [Q] Do Configurations group like Settings? In other words, if I had 3 Settings for my monitor, would these be grouped into a single Configuration?

[A] Configurations have the semantics of an aggregator of Settings and Dependencies, to describe a state or behavior - nothing more. From the example in Section 16, a mail setup at "home" versus at the "office" could involve different applications (maybe an email client versus a web browser), different setup of the applications, choice of a modem versus a network adapter, etc. The appropriate Settings would be grouped together within an instance of Configuration. Certainly these are not "like Settings" but quite diverse.

BTW, there really is no need for a Configuration object with respect to setting up a Video Controller. But, let's assume that a scenario existed where one was needed. I could reasonably group three different Setting instances into a single Configuration, or I might need three different Configurations, each with a specific Setting instance, for three different applications. If grouped into one Configuration, this would indicate that any one of the three Settings could be applied within the Configuration.

Configurations don't group "like" Settings - they group the appropriate Settings for a specific behavior or state.

17. Collections

CIM_Collection is an abstract base class for representing groups of Managed Elements. It is graphically shown in Figure 11, in Section 16. Collection defines no properties of its own, but does define a generic aggregation, Member Of Collection, that can be used to collect any type of Managed Element.

Subclassing and specializing Collection, CIM_CollectionOfMSEs is another abstract class grouping only Managed System Elements. It is defined as abstract to force subclassing to capture the specific behavior and semantics of how the Collection will be used.

There are two main motivators for this class. The first is to easily identify a group of Managed System Elements. The second is to simplify the process of associating Configuration and Setting objects with the Managed System Elements to which they apply. Without Collections, a developer would be forced to define or enumerate individual Element Setting and Element Configuration associations to the (potentially many) Elements in a Collection. This would lead to instance explosion. Furthermore, it would not easily capture the semantics that these particular Settings and Configurations are indeed the same objects for each of the Collection's members.

The Collection Of MSEs object defines a single property, Collection ID, which is a string of up to 256 characters. When subclassed, this property may be overridden to define it as part of the key structure of the new class. When aggregating Elements into the Collection Of MSEs class, the Collected MSEs relationship is used. It is a subclass of the Member Of Collection aggregation.

Lastly, the Collected Collections aggregation is a top-level association that allows the nesting of CIM_CollectionOfMSEs objects. Before the definition of Managed Element and the generic Collection class, this association conveyed a very different semantic than Collected MSEs and had no corollary in the Schema. Today, however, the latter is not true. Collected Collections could subclass from Member Of Collection, on the basis of semantics. However, on the basis of reference names, this is not possible. Member Of Collection uses the reference names Member and Collection, but Collected Collections uses the names Collection and CollectionInCollection. For this reason, it is recommended that new development use the Member Of Collection aggregation - even to aggregate Collections into other Collections - and maintain the Collected Collections association as legacy.

18. Statistics

CIM_StatisticalInformation is an abstract base class for the statistics class hierarchy in CIM. This hierarchy is shown in Figure 12, below. It represents a set of statistical data and/or metrics that is provided by and applicable to one or more Managed Elements.

The Statistical Information class inherits the Caption and Description properties from Managed Element, and defines a new property, called Name. This property is defined as a string, of a maximum of 256 characters, that provides a label by which the set of statistics and metrics are known. It can be overridden in a subclass to be a Key property.

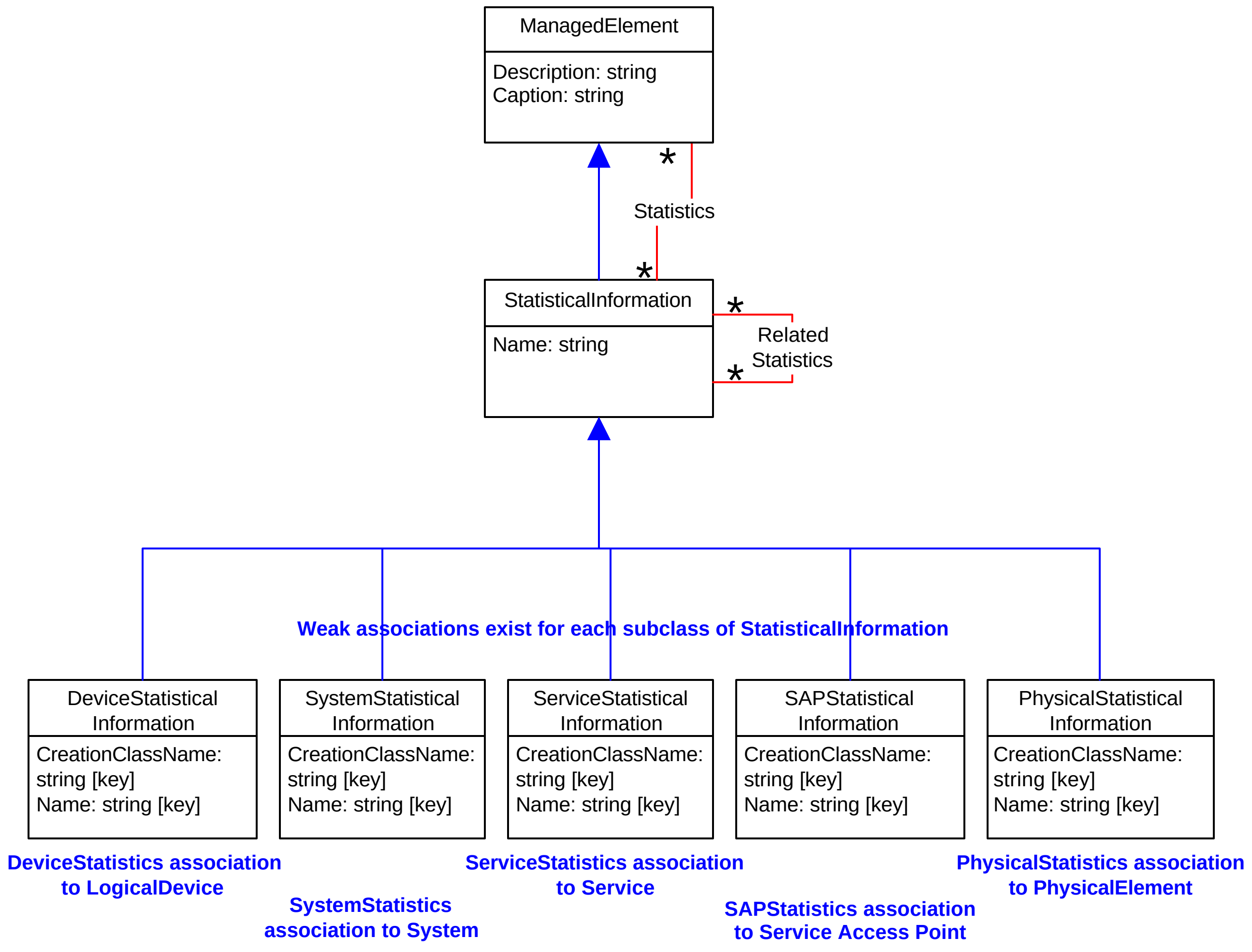

ManagedElement
Description: string
Caption: string
*
Statistics
*
StatisticalInformation
Name: string
*
Related Statistics
*
Weak associations exist for each subclass of StatisticalInformation
DeviceStatistical Information
CreationClassName: string [key]
Name: string [key]
SystemStatistical Information
CreationClassName: string [key]
Name: string [key]
ServiceStatistical Information
CreationClassName: string [key]
Name: string [key]
SAPStatistical Information
CreationClassName: string [key]
Name: string [key]
PhysicalStatistical Information
CreationClassName: string [key]
Name: string [key]
DeviceStatistics association to LogicalDevice
SystemStatistics association to System
ServiceStatistics association to Service
SAPStatistics association to Service Access Point
PhysicalStatistics association to PhysicalElement

Figure 12. The CIM Statistical Information Hierarchy

Two associations are defined for Statistical Information (which are inherited by all subclasses). These two associations are generic to the different types of statistics that are modeled in CIM.

- CIM_Statistics is a structural association relating a Managed Element to its statistical data. This enables a common representation and tying of statistical data to any type of Managed Element.
- CIM_RelatedStatistics is a top-level association for defining hierarchies and/or dependencies of instances of Statistical Information subclasses.

There are five subclasses of Statistical Information in the Core Model today. These are matched to the five fundamentally different subclasses of Managed System Element: Logical Devices, Systems, Services, Service Access Points, and Physical Elements.

The five subclasses are very symmetrical. Each defines a *Weak* association between itself and the subclass of Managed System Element for which statistical information is gathered. And, each has a composite key that is made up of its own Creation Class Name property, and two other components. The first component is the common overriding of the Name property, inherited from Statistical Information. The second component adds the appropriate keys based on the *Weak* association to a particular subclass of Managed System Element.

The five subclasses and their Weak associations are:

- Device Statistical Information and Device Statistics
- System Statistical Information and System Statistics
- Service Statistical Information and Service Statistics
- SAP Statistical Information and SAP Statistics
- Physical Statistical Information and Physical Statistics

In CIM, there are two approaches to modeling statistical data. Statistics can be defined either as properties of a class or in a subclass of Statistical Information. In some CIM classes, statistics are embedded as properties of the class. The Device Model includes several examples of this, such as in the Ethernet and Fibre Channel Adapter classes. Their statistical data are not modeled in a subclass of Statistical Information, because it is viewed as vital to understanding the operation of the hardware. Therefore, it is defined in the class itself. However, some statistics can be found in subclasses of Statistical Information. For example, the Device Model also defines CIM_FCAdapterEventCounters and CIM_FibrePortEventCounters classes as subclasses of CIM_DeviceStatisticalInformation. In these cases, the data would not typically be retrieved as part of the Device's status and operation. So, the data is placed in a subclass of Statistical Information, and associated with the instance for which the statistics are defined.

19. Mappings from Other Standards

As the Core Model is almost entirely composed of abstract classes, there are very few direct mappings to DMI and none to SNMP. Mappings of Schema classes become much more prevalent in the specializations of the Core classes, found in the Common Models. For example in the area of applications, there are specific mappings of software Product information to the application-related SNMP MIBs and the DMI MIFs. In the Network Model, there are many mappings to the network MIBs designed in the IETF.

The DMI attributes that are mapped in the Core Model are from the following Groups:

- ComponentID attributes mapped into Managed System Element and Product properties
- General Information attributes mapped into System properties
- Operational State attributes mapped into Logical Device properties
- FRU attributes mapped into FRU and Support Access properties
- Support attributes mapped into Support Access properties

20. Do's and Don'ts in CIM Design

Sometimes it is as informative to review what was NOT done in a Model, as to understand what is modeled. This section attempts to describe some modeling decisions and approaches taken by the CIM designers - both from the "DO" as well as the "DON'T" sides.

- Do not collapse Logical and Physical Devices.

When using the CIM Schema and/or defining extensions, care should be taken to understand the Logical/Physical dichotomy and correctly analyze classes and properties. Be aware that there are usually different discovery and instrumentation mechanisms for these two classes of objects.

- Do not model System and Device product types as individual classes.

Product types (for example, an Acme 10/100 Ethernet adapter or an Acme Personal Computer Model XYZ) are not represented in the CIM Schemas. Instead, “generic” System and Device classes are defined and instantiated for individually acquired and instrumented elements. Although this leads to a lot of duplicated data (i.e., MaxSpeed is always 100M for the Acme), it is flexible and extensible. Product “types” can change incrementally, which makes standard type info (which is typically static), dynamic. Also, taking this modeling approach would change the focus of CIM - from modeling information that crosses product boundaries to a set of "templates" representing specific products.

- Take care to distinguish between abstract and concrete associations on the basis of whether the association itself can be instantiated.

The concepts of abstract (conceptual) versus concrete (able to be instantiated) classes are important in CIM. Take care when linking concrete objects, as a concrete association must be defined. Alternately, it is allowed to define a concrete association, linking abstract objects. The latter can occur where the association is reasonably instantiated “as is” and where the association makes sense for the concrete subclasses of the original, abstract classes.

- It is not necessary to create a normalized model. Redundant/duplicated data may be needed.

A totally normalized model is not a requirement. Duplicated or derived data is sometimes desirable when the data is difficult to calculate, conceptually valuable in multiple locations in the model, or for other similar reasons. An example of redundant data can be found across the Device and Network Models, where both the CIM_NetworkAdapter and CIM_ProtocolEndpoint classes include a Network Address property.

- Do not force the model to be completely typed.

There are many examples where the CIM Schema specifies subclasses distinguished by the type of the object – for example, CIM_Sensor's Numeric Sensor, Binary Sensor, Discrete Sensor and Multi-State Sensor subclasses (in the Device Model) or all the subclasses of CIM_MediaAccessDevice (also in the Device Model). Subclasses exist where the various "types" have different properties or associations, where the Working Groups felt that the classes were critical or where further vendor-supplied extensions would be created. In other cases, subclasses are not defined - but a "type" property is specified for a class. For example, CIM_PointingDevice has no subclasses but has a Type enumeration distinguishing between mice, track balls, etc. "Type" properties exist where subclasses would likely not have different properties or associations, or where the majority of the Working Groups felt that the classes would not have great impact/meaning in an enterprise level solution.

21. Modeling Methodology

Models are abstractions of "real world" objects and events. However, different abstractions or perspectives may exist for the same "real world". For example, for Logical Devices, a Product, Physical, data flow or configuration perspective may be taken and would result in a very different model, if designed in isolation.

In order to address these issues and aid in the modeling of Core and Common Model extensions, the following methodology may be helpful.

- Define the various "perspectives" that the model(s) should address, and the information required from each perspective
- Separate the information into object/class data and information specific to relationships between objects
- Review the CIM Schemas for similar concepts
- Given the separation of the data and the existing CIM class hierarchy, define appropriate classes and associations
- Attempt to place each property or attribute in one class only, and if it is not possible to do this (for example, data must be duplicated), thoroughly analyze why
- Where possible, collapse similar abstractions from several models and perspectives
- Define levels of granularity for the data (from high level to complex), allowing instantiation and detailed analysis to occur, when and as necessary

22. References

CIM Core and Common Models - Versions 2.0, 2.1, 2.2, 2.3 and 2.4 - Downloadable from http://www.dmtf.org/spec/cims.html

Common Information Model (CIM) Specification, V2.2, June 14, 1999 - Downloadable from http://www.dmtf.org/spec/cims.html

DMTF Specifications - Approved Errata - Downloadable from http://www.dmtf.org/spec/cims.html

Unified Modeling Language (UML) from the Open Management Group (OMG) - Downloadable from http://www.omg.org/uml/

Desktop Management Interface (DMI) - Downloadable from http://www.dmtf.org/spec/dmis.html

Internet Engineering Task Force (IETF) - MIBs and Work Group information at http://www.ietf.org