Meta Object Facility (MOF) Specification

Version 1.4.1
formal/05-05-05

This version has been formally published by ISO as the 2005 edition standard / ISO/IEC 19502.

Contents

© ISO/IEC 2005 - All rights reserved

© ISO/IEC 2005 - All rights reserved

© ISO/IEC 2005 - All rights reserved

© ISO/IEC 2005 - All rights reserved

© ISO/IEC 2005 - All rights reserved

© ISO/IEC 2005 - All rights reserved

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 19502 was prepared by Technical Committee ISO/IEC/TC JTC1, *Information technology*, Subcommittee SC 32, *Data Management and Interchange* in collaboration with the Object Management Group (OMG), following the submission and processing as a Publicly Available Specification (PAS) of the OMG Meta Object Facility (MOF) Specification, Version 1.4 (formal/02-04-03).

ISO/IEC 19502 is related to:

- ISO/IEC 19501:2005, Information technology -- Unified Modeling Language (UML)
- ISO/IEC 19503:2005, Information technology -- XML Metadata Interchange (XMI)
- ISO/IEC 14769, Information technology -- Open Distributed Processing -- Type Repository Function

Apart from this Foreword, the text of this International Standard is identical with that for the OMG specification for MOF, v 1.4.1 (formal/05-05-05).

© ISO/IEC 2005 - All rights reserved

© ISO/IEC 2005 - All rights reserved

Introduction

The Meta-Object Facility (MOF) Specification defines a metamodel (defined using MOF), a set of interfaces (defined using ODP IDL (ITU-T Recommendation X.920 (1997) | ISO/IEC 14750: 1997), that can be used to define and manipulate a set of interoperable metamodels and their corresponding models. The MOF specification also defines the mapping from MOF to ODP IDL (ITU rec X920|ISO 14750). These interoperable metamodels include the Unified Modeling Language (UML) metamodel (ISO/IEC 19501 : 2005), the MOF meta-metamodel, as well as future standard technologies that will be specified using metamodels. The MOF provides the infrastructure for implementing design and reuse repositories, application development tool frameworks, etc. The MOF specifies precise mapping rules that enable the CORBA interfaces for metamodels to be generated automatically, thus encouraging consistency in manipulating metadata in all phases of the distributed application development cycle. Mappings from MOF to W3C XML and XSD are specified in the XMI (ISO/IEC 19503) specification. Mappings from MOF to Java™ are in the JMI (Java Metadata Interchange) specification defined by the Java Community Process.

In order to achieve architectural alignment considerable effort has been expended so that the UML and MOF share the same core semantics. This alignment allows the MOF to reuse the UML notation for visualizing metamodels. In those areas where semantic differences are required, well-defined mapping rules are provided between the metamodels. The UML has been the subject of a separate PAS submission.

The OMG adopted the MOF (version 1.0) in November 1997. It was developed as a response to a request for proposal, issued by the OMG Analysis and Design Task Force, for Metadata repository facility (http://www.omg.org/cgi-bin/doc?cf/96-05-02). The purpose of the facility was to support the creation, manipulation, and interchange of meta models. The most recent revision of MOF, 1.4 was adopted in April 2002, and includes corrections and clarifications to the original 1.3 version, and minor modeling feature additions.

The rapid growth of distributed processing has led to a need for a coordinating framework for this standardization and ITU-T Recommendations X.901-904 | ISO/IEC 10746, the Reference Model of Open Distributed Processing (RM-ODP) provides such a framework. It defines an architecture within which support of distribution, interoperability, and portability can be integrated. RM-ODP Part 2 (ISO/IEC 10746-2) defines the foundational concepts and modeling framework for describing distributed systems. RM-ODP Part 3 (ISO/IEC 10746-3) specifies a generic architecture of open distributed systems, expressed using the foundational concepts and framework defined in Part 2.

While not limited to this context, the MOF standard is closely related to work on the standardization of Open Distributed Processing (ODP). In particular, the ODP Type Repository Function (ISO/IEC 14769| Rec. X.960) references the OMG Meta Object Facility, version 1.3. This function specifies how to use the OMG MOF as a repository for ODP types. Future versions of this standard will be changed to reflect the ISO/IEC version of the MOF, which will result from this PAS submission.

© ISO/IEC 2005 - All rights reserved

© ISO/IEC 2005 - All rights reserved

Information technology - Meta Object Facility (MOF)

1 Scope

This International Standard specifies:

a. An abstract language and for specifying, constructing, and managing technology neutral metamodels: A metamodel is in effect an abstract language for some kind of metadata.

b. A framework for implementing repositories & integration frameworks (e.g., tool integration frameworks) that hold metadata (e.g., models) described by the metamodels and uses standard technology mappings to transform MOF metamodels into metadata APIs.

This International Standard also provides:

a. A formal definition of the MOF meta-metamodel; that is, the abstract language for specifying MOF metamodels.

b. A mapping from arbitrary MOF metamodels to CORBA IDL that produces IDL interfaces for managing any kind of metadata.

c. A set of “reflective” CORBA IDL interfaces for managing metadata independent of the metamodel.

d. A set of CORBA IDL interfaces for representing and managing MOF metamodels.

e. An XMI format for MOF metamodel interchange (OMG XMI Specification).

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

2.1 Identical Recommendations | International Standards

- ISO/IEC 10746-2:1995, Information technology -- OpenDistributed Processing -- Reference Model: Foundations
- ISO/IEC 10746-3:1995, Information technology -- OpenDistributed Processing -- Reference Model: Architecture

2.2 International Standards

- ISO/IEC 19501:2005, Information Technology -- Open Distributed Processing -- Unified Modeling Language (UML) Version 1.4.2
- ISO/IEC 19502:2005, Information technology -- Meta Object Facility (MOF)
- ISO/IEC 19503:2005, Information Technology -- XML Metadata Interchange (XMI)

© ISO/IEC 2005 - All rights reserved

3 Abbreviations and Conventions

The use of IDL conventions is as defined in the IDL standard.

CORBA	Common Object Request Broker Architecture
IDL	Interface Definition Language
MOF	Meta Object Facility
UML	Unified Modeling Language
XMI	XML Metadata Interchange Specification

4 List of Documents

The following is a list of the electronic documents that variously specify the MOF meta-models and MOF IDL APIs and the XMI DTD for MOF meta-model interchange. These documents may be downloaded from the OMG's Web server at:

`http://www.omg.org/technology/documents/formal/mof.htm`

MOF1.4/XMI1.1/Model1.4/Model.xml

This document (ptc/2001-10-05) is *normative*. It expresses the MOF 1.4 Model package as using the XMI 1.1 for MOF Model 1.4 interchange format. The XMI document contains cross-links to the PrimitiveTypes.xml document. It was generated from the Model.modl file below using an automatically generated MOF 1.4 metamodel repository and an automatically generated XMI serializer.

MOF1.4/XMI1.1/Model1.4/PrimitiveTypes.xml

This document (ptc/2001-10-06) is *normative*. It expresses the MOF 1.4 PrimitiveTypes package using the XMI 1.1 for MOF Model 1.4 interchange format. The XMI document was produced by serializing a hard-coded representation of the package using an automatically generated XMI serializer.

MOF1.4/XMI1.1/Model1.4/CorbaIdlTypes.xml

This document (ptc/2001-10-07) is *normative*. It expresses the MOF 1.4 CorbaIdlTypes package using the XMI 1.1 for MOF Model 1.4 interchange format. The XMI document was produced by serializing a hard-coded representation of the package using an automatically generated XMI serializer.

MOF1.4/XMI1.1/Model.dtd

This document (ptc/2001-08-09) is *normative*. It is the standard DTD for XMI 1.1 interchange of MOF 1.4 metamodels.

© ISO/IEC 2005 - All rights reserved

5 MOF Usage Scenarios

5.1 Overview

The MOF is intended to support a wide range of usage patterns and applications. To understand the possible usage patterns for the MOF, the first thing one needs to understand is the two distinct viewpoints for the MOF:

1. Modeling viewpoint: The designer's viewpoint, looking "down" the meta levels. From the modeling viewpoint, the MOF is used to define an information model for a particular domain of interest. This definition is then used to drive subsequent software design and/or implementation steps for software connected with the information model.

2. Data viewpoint: The programmer's viewpoint, looking at the current meta-level, and possibly looking up at the higher meta-levels. From the data viewpoint, the MOF (or more accurately, a product of the MOF) is used to apply the OMA-based distributed computing paradigm to manage information corresponding to a given information model. In this mode, it is possible for a CORBA client to obtain the information model descriptions and to use them to support reflection.

The second thing one needs to realize is that this MOF specification is intended to provide an open-ended information modeling capability. The specification defines a core MOF model that includes a relatively small, though not minimal, set of constructs for object-oriented information modeling. The MOF model can be extended by inheritance and composition to define a richer information model that supports additional constructs. Alternatively, the MOF model can be used as a model for defining information models. This feature allows the designer to define information models that differ from the philosophy or details of the MOF model. In this context, the MOF Model is referred to as a meta-metamodel because it is being used to define metamodels such as the UML.

Finally, one needs to understand the purpose and the limitations of the MOF model to the CORBA IDL mapping defined by this specification. The prime purpose of the mapping is to define CORBA interfaces for information models defined in terms of the MOF model[1] using standard interfaces and interoperable semantics. These interfaces allow a client to create, access, and update information described by the model, with the expectation that the information will be managed in a way that maintains the structural and logical consistency constraints specified in the information model definition.

While we anticipate that some vendors will supply tools (for example, IDL generators, server generators, and so on) to support the development of software conforming to the mapping, provision of these tools is not a requirement of this specification. The second limitation is that the mapping is only intended to support the MOF model itself; that is, it does not support extensions to the metamodel or to other unconnected information models. Furthermore, since the IDL mapping is not itself modeled in the MOF, there can be no standardized support for extending the mapping or defining new mappings. Finally, the IDL mapping in this specification supports only CORBA IDL. Mappings from the MOF model to other interface definition languages are certainly feasible, as are direct mappings to programming languages or data definition languages. However, these mappings are beyond the scope of the first version of the MOF specification.

1. Both extensions to the MOF meta-model that are expressible in the meta-model itself, and unconnected information models expressed using the MOF meta-model.

© ISO/IEC 2005 - All rights reserved

5.2 Software Development Scenarios

Initially, one of the most likely applications of the MOF will be to support the development of distributed object-oriented software from high-level models. Such a software development system would typically consist of a repository service for storing the computer representations of models and a collection of associated tools. The latter would allow the programmers and designers to input the models, and would assist in the process of translating these models into software implementations.

In the simple case, the repository service could be an implementation of the MOF model interfaces. This service would be accompanied by tools (for example, compilers or graphical editors) that allow the designer to input information models using a human readable notation for the MOF model. Assuming that the target for software development is CORBA based, the system would include an IDL generator that implements the standard MOF model-to-CORBA IDL mapping.

The usage scenario for this repository service would be along the following lines:

1. The programmer uses the input tools provided by the system to define an object-oriented information model using the notation provided.
2. When the design is complete, the programmer runs the IDL generator to translate the model into CORBA IDL.
3. The programmer examines the IDL, repeating steps 1 and 2 to refine the model as required.
4. The programmer then implements the generated IDL to produce a target object server, and implement the applications that use the object server.

The functionality of the development suite described above can be expanded in a variety of ways. We can:

- Add generator tools to automatically produce the skeleton of an object server corresponding to the generated IDL. Depending on the sophistication of the tool, this skeleton might include code for the query and update operations prescribed by the IDL mapping, and code to check the constraints on the information model.
- Add generator tools to produce automatically stereotypical applications such as scripting tools and GUI-based browsers.
- Extend the repository service to store the specifications and/or implementation code for target server and application functionality that cannot be expressed in the MOF model.

While the MOF model is a powerful modeling language for expressing a range of information models, it is not intended to be the ultimate modeling language. Instead, one intended use of the MOF is as a tool for designing and implementing more sophisticated modeling systems. The following example illustrates how the MOF might be used to construct a software development system centered around a hypothetical “Universal Design Language” (UDL).

Many parallels can be drawn between the hypothetical UDL discussed below and the draft OA&DF UML proposal in that UML is designed to be a general purpose modeling language for visualizing, designing, and developing component software. The UDL can be thought of as an extension, as well as a refinement, of many of the concepts in the UML. The extensions are mainly in the area of providing sufficient detail to complete the implementation framework technologies and defining additional meta models that address various technology domains such as database management, transaction processing, etc.

The developer of a software development system based on UDL might start by using a MOF Model notation to define a meta-model for UDL. Conceivably, the UDL metamodel could reuse part or all of the MOF Model, though this is not necessarily a good idea[2]. The developer could then use a simple MOF-based development system (along the lines described above) to translate the UDL metamodel into CORBA IDL for a UDL repository, and to provide hand-written or generated software that implements the UDL repository and suitable UDL model input tools.

© ISO/IEC 2005 - All rights reserved

The hypothetical UDL development system cannot be considered complete without some level of support for the process of creating working code that implements systems described by the UDL models. Depending on the nature of the UDL, this process might involve a number of steps in which the conceptual design is transformed into more concrete designs and, finally, into program source code. A UDL development system might provide a range of tools to assist the target system designer or programmer. These tools would need to be supported by repository functions to store extra design and implementation information, along with information such as version histories, project schedules, and so on, that form the basis of a mature software development process.

In practice, a software development system implemented along these lines would have difficulty meeting the needs of the marketplace. A typical software engineering "shop" will have requirements on both the technical and the process aspects of software engineering that cannot be met by a "one-size-fits-all" development system. The current trend in software development systems is for Universal Repository systems; that is, for highly flexible systems that can be tailored and extended on the fly.

A MOF-based universal repository system would be based around the core of the MOF Model, and a suite of tools for developing target metamodels (for example, the UDL) and their supporting tools. Many of the tools in the universal repository could be reflective; that is, the tools could make use of information from higher meta-levels to allow them to operate across a range of model types. Functionality, such as persistence, replication, version control, and access control would need to be supported uniformly across the entire repository framework.

5.3 Type Management Scenarios

A second area where early use of the MOF is likely is in the representation and management of the various kinds of type information used by the expanding array of CORBA infrastructure services.

The CORBA Interface Repository (IR) is the most central type-related service in CORBA. The IR serves as a central repository for interface type definitions in a CORBA-based system. The current IR essentially provides access to interface definitions that conform to the implied information model of CORBA IDL. While the IR interfaces are tuned fairly well to read-only access, there is no standard update interface and no way to augment the interface definitions in the IR with other relevant information, such as behavioral semantics.

Given a simple MOF-based development environment (as described above), it would be easy to describe the implied information model for CORBA IDL using a notation for the MOF Model. The resulting CORBA IDL model could then be translated into the IDL for a MOF-based replacement for the CORBA IR. While this replacement IR would not be upwards compatible with the existing IR, the fact that it was MOF-based would provide a number of advantages. The MOF-based IR would:

- Support update interfaces.
- Be extensible in the sense that it would be feasible to extend the CORBA IDL model specification by (MOF Model) composition and inheritance. This ability would help smooth the path for future extensions to the CORBA object model.
- Make it easier to federate multiple IR instances and to represent associations between CORBA interface types and other kinds of type information.
- Automatically include links to its own meta-information definition expressed using MOF meta-objects.

---

2. The MOF meta-model has specific requirements (e.g., model simplicity and support for automatic IDL generation) that are not generally applicable. As a consequence, it is unreasonable to expect the MOF metamodel design to be suitable for all kinds of object modeling.

© ISO/IEC 2005 - All rights reserved

Other candidates for use of MOF-based technology among existing and forthcoming infrastructure services include:

- Trader: The CORBA trader service maintains a database of “service offers” from services in a CORBA-based distributed environment. These offers have associated service types that are represented using the **CosTradingRepos::ServiceTypeRepository** interface. (A trader service type is a tuple consisting of a type name, an interface type, and a set of named property types. Service types can be defined as subtypes of other service types.)
- Notification: At least one initial submission for the forthcoming Notification service includes the notion of an event type. (An event type is a tuple consisting of a type name, a set of named property types, and a set of supertypes.)

In both cases, a MOF-based type repository would have the advantages listed previously for the MOF-based Implementation Repository.

Looking to the future, there are a number of other possible uses for MOF-based type repositories in infrastructure services. For example:

- Service interface bridges: As CORBA matures and there is large-scale deployment as part of enterprise-wide computing infrastructures, it will become increasingly necessary to cope with legacy CORBA objects; that is, with objects that provide or use out-of-date service interfaces. In situations where statically deployed object wrappers are not a good solution, one alternative is to provide an ORB-level service that can insert an interface bridge between incompatible interfaces at bind time. Such a service would depend on types that describe the available bridges and the mechanisms used to instantiate them.
- Complex bindings: RM-ODP supports the idea that bindings between objects in a distributed environment can be far more complex than simple RPC, stream or multicast protocols. RM-ODP defines the notion of a multi-party binding involving an arbitrary number of objects of various types, in which different objects fill different roles in the binding. A CORBA service to manage complex bindings would be based on formally described binding types that specify the numbers and types of objects filling each role and the allowed interaction patterns (behaviors) for a given binding.

5.4 Information Management Scenarios

The previous sub clauses focused on the use of the MOF to support the software development life-cycle and the type management requirements of CORBA infrastructure services. This sub clause broadens the scope to the more general domain of information management; that is, the design, implementation, and management of large bodies of more or less structured information.

First, note that some of the ideas outlined above carry over to the information management domain. In some cases, it may be appropriate to define the information model (that is, the database schema) for the application of interest directly using the MOF Model. In this case, the technology described previously can be used to automate the production of CORBA-based servers to store the information and applications to use it. In other situations, the MOF Model can be used to define a metamodel suitable for defining information models for the domain of interest; for example, a metamodel for describing relational database schemas. Then a development environment can be designed and implemented using MOF-based technology that supports the generation of CORBA-based data servers and applications from information models.

In addition, the MOF potentially offers significant benefits for large-scale information systems by allowing such a system to make meta-information available at run-time. Some illustrative examples follow.

Information discovery: The World-Wide Web contains a vast amount of useful (and useless) information on any topic imaginable. However, this information is largely inaccessible. In the absence of other solutions, current generation web indexing systems or search engines must rely on simple word matching. Unless the user frames queries carefully, the number of “hits” returned by a search engine are overwhelming. Furthermore, it is now apparent that even the largest search engines cannot keep pace with the Web’s rate of growth.

© ISO/IEC 2005 - All rights reserved

In the absence of software that can “understand” English text, the approach most likely to succeed is to build databases of meta-data that describe web pages. If this meta-data is represented using MOF-based technology and an agreed base metamodel for the meta-data, the framework can support local meta-data extensions through judicious use of MOF-supported reflection. In addition, because the meta-data framework is defined in the MOF context, it can be accessible to a larger class of generic tools.

5.5 Data Warehouse Management Scenarios

Data warehousing is a recent development in enterprise-scale information management. The data warehouse technique recognizes that it is impractical to manage the information of an enterprise as a unified logical database. Instead, this technique extracts information from logically- and physically-distinct databases, integrates the information, and stores it in a large-scale “warehouse” database that allows read-only access to possibly non-current data. The extraction and integration processes depend on a database administrator creating a mapping from the schemas for the individual databases to the schema of the warehouse. If the meta-information for the various databases is represented using MOF-based technology, then it should be possible to create sophisticated tools to assist the database administrator in this process.

Meta data is often described as the “heart and soul” of the data warehouse environment. The MOF can be used to automate meta data management of data warehouses. Current meta data repositories that manage data warehouses often use static meta data using batch file-based meta data exchange mechanisms. We expect the use of MOF- and standard CORBA-based event and messaging mechanisms and mobile agent technology (also being standardized by OMG) to drive a new generation of data warehouse management tools and systems that are more dynamic. These tools will enable customers to react in a timelier manner to changing data access patterns and newly discovered patterns, which is the focus of data mining and information discovery systems.

The MOF interfaces and the MOF Model can be used to define specific metamodels for database, data warehouse, model transformation, and warehouse management domains. The integration between these models in a run time data warehouse and the development environment (which has data models) and UML based object models (which describes the corporate data models and operational databases) is a typical use of a MOF. The traceability across these environments is enabled by defining an impact analysis metamodel, which builds on the rich model of relationships supported by the MOF.

The OMG CWM specification represents one possible realization of this scenario.

© ISO/IEC 2005 - All rights reserved

© ISO/IEC 2005 - All rights reserved

6 MOF Conceptual Overview

6.1 Overview

The Meta Object Facility is a large specification. This Clause aims to make the MOF specification easier to read by providing a conceptual overview of the MOF.

The Clause starts by explaining the MOF's conceptual architecture for describing and defining metadata. The next sub clause introduces the metamodeling constructs that are used to describe metadata. This is followed by a sub clause that describes how metamodels are mapped to implementation technologies, including the IDL mapping and XMI.

6.2 Metadata Architectures

The central theme of the MOF approach to metadata management is extensibility. The aim is to provide a framework that supports any kind of metadata, and that allows new kinds to be added as required. In order to achieve this, the MOF has a layered metadata architecture that is based on the classical four layer metamodeling architecture popular within standards communities such as ISO and CDIF. The key feature of both the classical and MOF metadata architectures is the meta-metamodeling layer that ties together the metamodels and models.

The traditional four layer metadata architecture is briefly described below. This is followed by a more detailed description of how this maps onto the MOF metadata architecture.

6.2.1 Four Layer Metadata Architectures

The classical framework for metamodeling is based on an architecture with four meta- layers. These layers are conventionally described as follows:

- The information layer is comprised of the data that we wish to describe.
- The model layer is comprised of the metadata that describes data in the information layer. Metadata is informally aggregated as models.
- The metamodel layer is comprised of the descriptions (i.e., meta-metadata) that define the structure and semantics of metadata. Meta-metadata is informally aggregated as metamodels. A metamodel is an "abstract language" for describing different kinds of data; that is, a language without a concrete syntax or notation.
- The meta-metamodel layer is comprised of the description of the structure and semantics of meta-metadata. In other words, it is the "abstract language" for defining different kinds of metadata.

The classical four layer meta-modeling framework is illustrated in Figure 6.1 on page 10. This example shows metadata for some simple records (i.e., "StockQuote" instances) along with the "RecordTypes" metamodel for describing and the hard-wired meta-metamodel that defines the metamodeling constructs (e.g., meta-Classes and meta-Attributes).

While the example shows only one model and one metamodel, the main aim of having four meta-layers is to support multiple models and metamodels. Just as the "StockQuote" type describes many StockQuote instances at the information level, the "RecordTypes" metamodel can describe many record types at the model level. Similarly, the meta-metamodel level can describe many other metamodels at the metamodel that in turn represent other kinds of metadata describing other kinds of information.

© ISO/IEC 2005 - All rights reserved

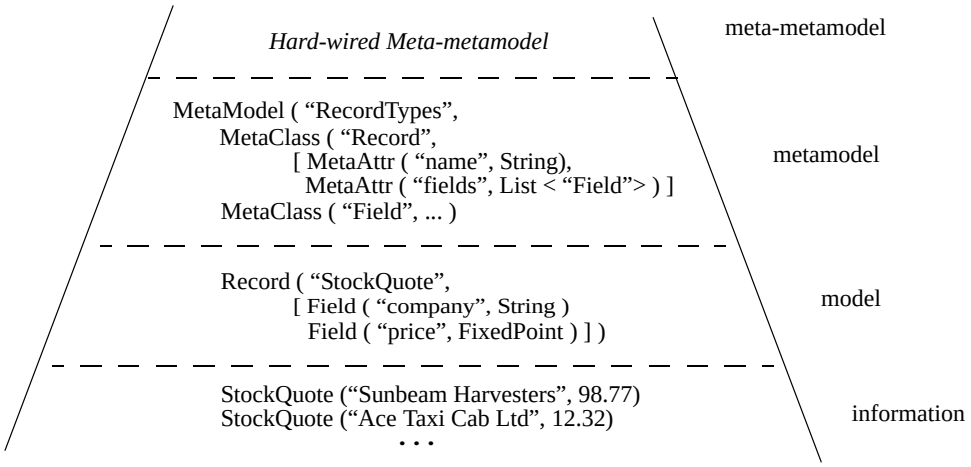

Figure 6.1 - Four Layer Metadata Architecture

The classical four layer metadata architecture has a number of advantages over simple modeling approaches. If the framework is designed appropriately, it can:

- support any kind of model and modeling paradigm that is imaginable,
- allow different kinds of metadata to be related,
- allow metamodels and new kinds of metadata to be added incrementally, and
- support interchange of arbitrary metadata (models) and meta-metadata (metamodels) between parties that use the same meta-metamodel.

6.2.2 The MOF Metadata Architecture

The MOF metadata architecture[1], illustrated by the example in Figure 6.2, is based on the traditional four layer metadata architecture described above. This example shows a typical instantiation of the MOF metadata architecture with metamodels for representing UML diagrams and OMG IDL.

1. One could argue that the term “architecture” is an inappropriate in this context. After all, the MOF metadata “architecture” is little more than a way of conceptualizing relationships between data and descriptions of data. Certainly, there is no intention the “architecture” be used as a benchmark for MOF conformance. However, the term has been used in MOF discussion for a long time, so the reader must forgive any perceived inaccuracy.

© ISO/IEC 2005 - All rights reserved

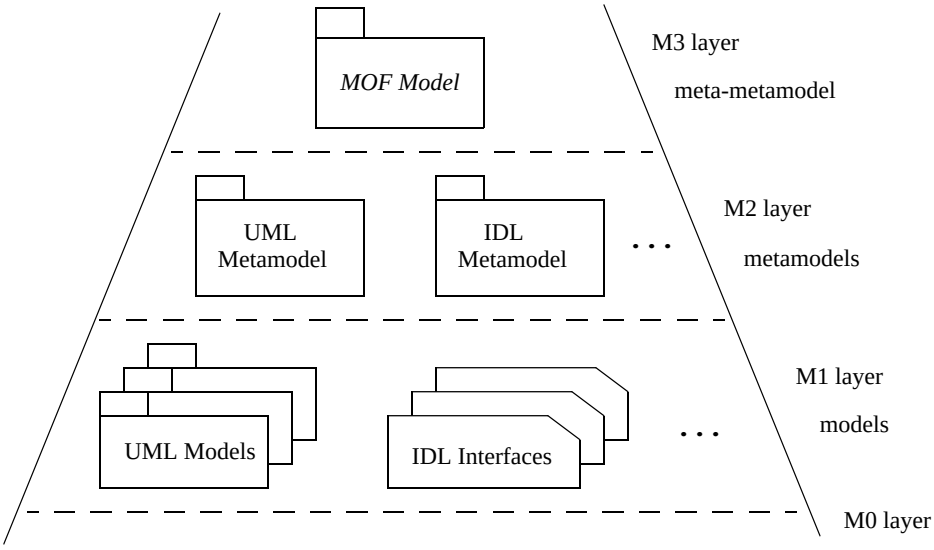

Figure 6.2 - MOF Metadata Architecture

The MOF metadata architecture has a few important features that distinguish it from earlier metamodeling architectures:

- The MOF Model (the MOF's core meta-metamodel) is object-oriented, with metamodeling constructs that are aligned with UML's object modeling constructs. Hence, the example uses UML package icons to denote MOF-based metamodels as well as UML models.
- The meta- levels in the MOF metadata architecture are *not* fixed. While there are typically 4 meta-levels, there could be more or less than this, depending on how MOF is deployed. Indeed, the MOF specification does not require there to be discrete meta- levels at all at the implementation level. MOF meta- levels are *purely* a convention for understanding relationships between different kinds of data and metadata.
- A model (in the broad sense of a collection of metadata) is not necessarily limited to one meta-level. For example, in a data warehousing context, it may be useful to think of the meta-schema "Relational Table" and specific schemas that are instances of relational tables as being one conceptual model.
- The MOF Model is self-describing. In other words, the MOF Model is formally defined using its own metamodeling constructs. Hence, the MOF Model is also denoted by a UML style Package icon.

The self-describing nature of the MOF Model has some important consequences:

- It shows that the MOF Model is sufficiently expressive for practical metamodeling.
- It allows the MOF's interfaces and behavior to be defined by applying the MOF IDL mapping to the MOF Model. This provides uniformity of semantics between computational objects that represent models and metamodels. It also means that when a new technology mapping is defined, the APIs for managing metamodels in that context are implicitly defined as well.
- It provides an architectural basis for extensions and modifications to the MOF Model. Successive MOF RTFs have thus been able to make incremental changes in the MOF Model to address problems that become apparent. In the future, new meta-metamodels may be added to support tasks like specification of modeling notations and model-to-model transformations.

© ISO/IEC 2005 - All rights reserved

- Given an appropriate set of implementation generators, it allows new MOF metamodel repository implementations and associated tools to be created by bootstrapping.

6.2.3 MOF Metamodeling Terminology

There is enormous scope for confusion if standard metamodeling terminology is used in the MOF specification. To avoid this and to make it easier to read, we have opted to simplify the terminology. Some particular points of confusion are as follows:

- Since the number of MOF meta-levels is not fixed and meta-levels are conventionally named upwards from the "information" layer, the "top" meta-level of the stack varies. Some people find this idea hard to grasp.
- There are a number of object modeling concepts that appear at two, three, or even four levels in a well populated MOF metadata framework. For example, a class in a UML is described by an instance of the class "Class" in the UML metamodel. This is in turn described by an instance of the class "Class" in the MOF Model. Finally, the class "Class" in the MOF Model is described by itself. This overloading of names of concepts often confuses people.
- While the "meta-" prefix has a clear meaning in the context of the MOF, evidence suggests that people who encounter it for the first time find it very confusing. This is particularly the case for forms like "meta-meta-" and "meta-meta-meta-". Even for seasoned experts, "meta-meta-meta-" is cumbersome in conversation.

To avoid some of this confusion, we generally try to avoid using the "meta-" prefix. In particular, while the core of the MOF is a meta-metamodel (assuming that there are 4 meta- layers), it is referred to as "the MOF Model." Similarly, rather than using terms like Class, MetaClass, and MetaMetaClass, we use phraseology like "an M1-level instance of an M2-level Class."

The meta-level numbering used in the remainder of this specification (for example, M2-level or M1-level) should be read as *top down* labels *relative* to the MOF Model at M3-level. We assume that the reader can mentally relabel the meta-levels to fit the context; for example, in contexts where the MOF Model is not at M3-level, or when applying the IDL mapping to the MOF Model itself.

NOTE: Even the M1- / M2- terminology above has proved to be confusing to some readers. However, since changing the terminology again will take significant effort and cause considerable disruption, further work on this has been deferred until a clearly superior terminology is proposed.

There are three cases where it is convenient to use the "meta-" prefix as part of MOF terminology:

1. The term "metadata" is used to refer to data whose purpose is to describe other data.
2. The term "metamodel" is used to refer to a model of some kind of metadata.
3. The term "metaobject" is used to refer to an abstract or technology specific object that represents metadata.

In each case, the term is used across all meta-levels and has a deliberately imprecise meaning.

The core modeling concepts in the MOF use terms that are also used in UML with similar meanings. For example, a MOF Class corresponds to a UML Class, a MOF Attribute corresponds to a UML Attribute, and a MOF Association corresponds to a UML Association. However the correspondence is not always a direct match. For example, UML Associations may have many AssociationEnds, but MOF Associations must have precisely two.

© ISO/IEC 2005 - All rights reserved

6.3 The MOF Model - Metamodeling Constructs

This sub clause introduces the MOF's core metamodeling constructs (i.e., the MOF's "abstract language") for defining metamodels.

MOF metamodeling is primarily about defining information models for metadata. The MOF uses an object modeling framework that is essentially a subset of the UML core. In a nutshell, the 4 main modeling concepts are:

1. Classes, which model MOF metaobjects.
2. Associations, which model binary relationships between metaobjects.
3. DataTypes, which model other data (e.g., primitive types, external types, etc.).
4. Packages, which modularize the models.

6.3.1 Classes

Classes are type descriptions of "first class instance" MOF metaobjects. Classes defined at the M2 level logically have instances at the M1 level. These instances have object identity, state, and behavior. The state and behavior of the M1 level instances are defined by the M2 level Class in the context of the common information and computational models defined by the MOF specification.

Instances of classes belong to class extents that impact on certain aspects of their behavior. It is possible to enumerate all instances of a class in a class extent (see 8.8.2, "Class Extents," on page 147).

Classes can have three kinds of features. Attributes and Operations described below and References described in "References" on page 18. Classes can also contain Exceptions, Constants, DataTypes, Constraints, and other elements.

6.3.1.1 Attributes

An Attribute defines a notional slot or value holder, typically in each instance of its Class. An Attribute has the following properties.

Property	Description
name	Unique in the scope of the Attribute's Class.
type	May be a Class or a DataType.
"isChangeable" flag	Determines whether the client is provided with an explicit operation to set the attribute's value.
"isDerived" flag	Determines whether the contents of the notional value holder is part of the "explicit state" of a Class instance, or is derived from other state.
"multiplicity" specification	(see "Attribute and Parameter Multiplicities" on page 14)

The aggregation properties of an Attribute depend on the Attribute's type; see 6.3.3, "Aggregation," on page 17.

© ISO/IEC 2005 - All rights reserved

6.3.1.2 Operations

Operations are “hooks” for accessing behavior associated with a Class. Operations do not actually specify the behavior or the methods that implement that behavior. Instead they simply specify the names and type signatures by which the behavior is invoked. Operations have the following properties.

Property	Description
name	Unique in the scope of the Class.
list of positional parameters having the following properties:	
Parameter name:	
Parameter type	may be denoted by a Class or a DataType
Parameter direction of “in,” “out,” or “in out”	determines whether actual arguments are passed from client to server, server to client, or both.
Parameter “multiplicity” specification	see “Attribute and Parameter Multiplicities” on page 14
An optional return type.	
A list of Exceptions that can be raised by an invocation.	

6.3.1.3 Attribute and Operation Scoping

Attributes and Operations can be defined as “classifier level” or “instance level.” An instance-level Attribute has a separate value holder for each instance of a Class. By contrast, a classifier-level Attribute has a value holder that is shared by all instances of the Class in its class extent.

Similarly, an instance-level Operation can only be invoked on an instance of a Class and will typically apply to the state of that instance. By contrast, a classifier-level Operation can be invoked independently of any instance, and can apply to any or all instances in the class extent.

6.3.1.4 Attribute and Parameter Multiplicities

An Attribute or Parameter may be optional-valued, single-valued, or multi-valued depending on its multiplicity specification. This consists of three parts:

1. The “lower” and “upper” fields place bounds on the number of elements in the Attribute or Parameter value. The lower bound may be zero and the upper may be “unbounded.”
   - A single-valued Attribute or Parameter has lower bound 1 and upper bound 1. An optional-valued Attribute or Parameter has lower bound 0 and upper bound 1. All other cases are called multi-valued parameters (since their upper bound is greater than 1).

NOTE: Multiplicity bounds are typically notated as one or two numbers, with “*” used to denote unbounded. For example, a UML bounds specification of “1” translates to lower and upper bounds of 1, and “2..*” translates to a lower bound of 2 and no upper bound.

2. The “is_ordered” flag says whether the order of values in a holder has semantic significance. For example, if an Attribute is ordered, the order of the individual values in an instance of the Attribute will be preserved.

© ISO/IEC 2005 - All rights reserved

3. The “is_unique” flag says whether instances with equal value are allowed in the given Attribute or Parameter. The meaning of “equal value” depends on the base type of the Attribute or Parameter. See 8.4, “Semantics of Equality for MOF Values,” on page 140, and 9.3.3, “Value Types and Equality in the IDL Mapping,” on page 170 for additional information.

NOTE: The bounds and uniqueness parts of a multiplicity specification can give rise to runtime “structural checks” (see 6.3.7.2, “Structural Consistency,” on page 24). By contrast, orderedness does not imply any runtime checking.

6.3.1.5 Class Generalization

The MOF allows Classes to inherit from one or more other Classes. Following the lead of UML, the MOF Model uses the verb “to generalize” to describe the inheritance relationship (i.e., a super-Class generalizes a sub-Class).

The meaning of MOF Class generalization is similar to generalization in UML and to interface inheritance in CORBA IDL. The sub-Class inherits all of the contents of its super-Classes (i.e., all of the super-Classes Attributes, Operations and References, and all nested DataTypes, Exceptions, and Constants). Any explicit Constraints that apply to a super-Class and any implicit behavior for the super-Class apply equally to the sub-Class. At the M1 level, an instance of an M2-level Class is type substitutable for instances of its M2-level super-Classes.

The MOF places restrictions on generalization to ensure that it is meaningful and that it can be mapped onto a range of implementation technologies:

- A Class cannot generalize itself, either directly or indirectly.
- A Class cannot generalize another Class if the sub-Class contains a model element with the same name as a model element contained or inherited by the super-Class (i.e., no over-riding is allowed).
- When a Class has multiple super-Classes, no model elements contained or inherited by the super-Classes can have the same name. There is an exception (analogous to the “diamond rule” in CORBA IDL) that allows the super-Classes to inherit names from a common ancestor Class.

NOTE: It is also possible to use Tags to specify that the interfaces generated for a Class inherits from pre-existing interfaces.

6.3.1.6 Abstract Classes

A Class may be defined as “abstract.” An abstract Class is used solely for the purpose of inheritance. No metaobjects can ever exist whose most-derived type corresponds to an abstract Class.

NOTE: The MOF uses “abstract Class” in the same sense as UML, and also Java and many other object oriented programming languages. Specifying a MOF Class as “abstract” does not say how instances are transmitted. In particular, the use of the term “abstract class” has no relationship to the IDL keyword “abstract” introduced by the Objects-by-value specification.

6.3.1.7 Leaf and Root Classes

A Class may be defined as a “leaf” or “root” Class. Declaring a Class as a leaf prevents the creation of any sub-Classes. Declaring a Class as a root prevents the declaration of any super-Classes.

© ISO/IEC 2005 - All rights reserved

6.3.2 Associations

Associations are the MOF Model's primary construct for expressing the relationships in a metamodel. At the M1 level, an M2 level MOF Association defines relationships (links) between pairs of instances of Classes. Conceptually, these links do not have object identity, and therefore cannot have Attributes or Operations.

6.3.2.1 Association Ends

Each MOF Association contains precisely two Association Ends describing the two ends of links. The Association Ends define the following properties.

Property	Description
A name for the end	This is unique within the Association.
A type for the end	This must be a Class.
Multiplicity specification	See 6.3.2.2, "Association End Multiplicities," on page 16.
An aggregation specification	See 6.3.3.2, "Association Aggregation," on page 18.
A "navigability" setting	Controls whether References can be defined for the end (see 6.3.4, "References," on page 18).
A "changeability" setting	Determines whether this end of a link can be updated "in place."

6.3.2.2 Association End Multiplicities

Each Association End has a multiplicity specification. While these are conceptually similar to Attribute and Operation multiplicities, there are some important differences:

- An Association End multiplicity does not apply to the entire link set. Instead, it applies to projections of the link set for the possible values of the "other" end of a link. See Figure 6.3.
- Since duplicate links are disallowed in M1-level link sets, "is_unique" is implicitly TRUE. The check for duplicate links is based on equality of the instances that they connect; see 8.4, "Semantics of Equality for MOF Values," on page 140.

© ISO/IEC 2005 - All rights reserved

.

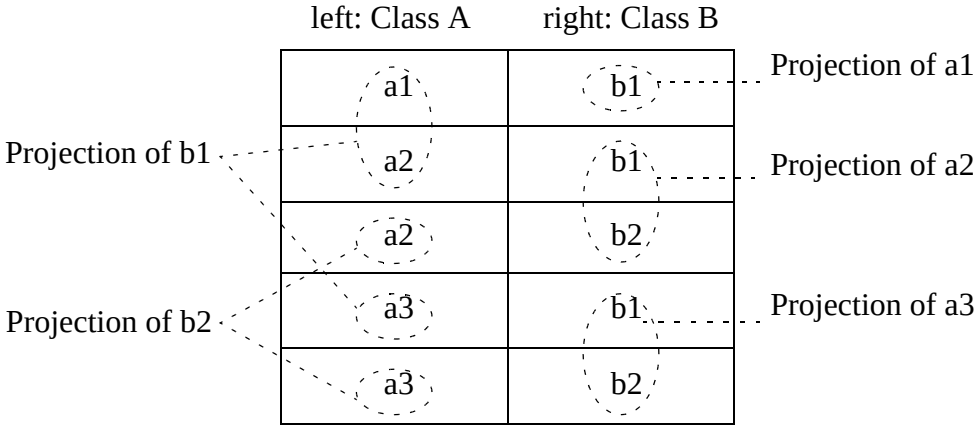

Figure 6.3 - The projections of a link set

Figure 6.3 shows a link set for an Association with an AssociationEnd named “left” whose Class is A, and a second named “right” whose Class is B. Instances of A are shown as “a1,” “a2,” and “a3” and “b1” and “b2” are instances of B. In this example with five links, the projection of “a1” is the collection {b1}, and the projection of “b1” is the collection {a1, a2, a3}. If there is another B instance (say “b3”) with no corresponding links, the projection of that b3 is an empty collection.

The “lower” and “upper” bounds of an Association End constrain the number of instances in a projection. For example, if the “left” End of the Association has a bounds “0..3,” then the projection of the link set for any extant instance of B must contain between zero and three instances of A.

The “is_ordered” flag for the Association End determines whether the projections from the other End have an ordering. The MOF Model only allows one of an Association’s two Association Ends to be marked as “ordered.”

In the above example, this could say whether order of the elements of the projection of “b1” is significant (i.e., whether {a1, a2, a3} is a set or a unique list).

6.3.3 Aggregation

In a MOF metamodel Classes and DataTypes can be related to other Classes using Associations or Attributes. In both cases, aspects of the behavior of the relationships can be described as aggregation semantics.

6.3.3.1 Aggregation Semantics

The MOF supports two kinds of aggregation for relationships between instances (i.e., “composite” and “non-aggregate”). A third aggregation semantic - “shared” - is not supported in this version of the MOF specification.

A non-aggregate relationship is a (conceptually) loose binding between instances with the following properties:

- There are no special restrictions on the multiplicity of the relationships.
- There are no special restrictions on the origin of the instances in the relationships.

© ISO/IEC 2005 - All rights reserved

- The relationships do not impact on the lifecycle semantics of related instances. In particular, deletion of an instance does not cause the deletion of related instances.

By contrast, a composite relationship is a (conceptually) stronger binding between instances with the following properties:

- A composite relationship is asymmetrical, with one end denoting the “composite” or “whole” in the relationship and the other one denoting the “components” or “parts.”
- An instance cannot be a component of more than one composite at a time, under any composite relationship.
- An instance cannot be a component of itself, its components, its components’ components and so on under any composite relationship.
- When a “composite” instance is deleted, all of its components under any composite relationship are also deleted, and all of the components’ components are deleted and so on.
- The Composition Closure Rule: an instance cannot be a component of an instance from a different package extent (see 8.11.2, “The Composition Closure Rule,” on page 155).

6.3.3.2 Association Aggregation

The aggregation semantics of an Association are specified explicitly using the “aggregation” Attribute of the AssociationEnds. In the case of a “composite” Association, the “aggregation” Attribute of the “composite” AssociationEnd is set to true and the “aggregation” Attribute of the “component” AssociationEnd is set to false. Also, the multiplicity for the “composite” AssociationEnd is required to be “[0..1]” or “[1..1]” in line with the rule that an instance cannot be a component of multiple composites.

6.3.3.3 Attribute Aggregation

The effective aggregation semantics for an Attribute depend on the type of the Attribute. For example:

- An Attribute whose type is expressed as a DataType has “non-aggregate” semantics.
- An Attribute whose type is expressed as a Class has “composite” semantics.

It is possible to use a DataType to encode the type of a Class. Doing this allows the metamodel to define an Attribute whose value or values are instances of a Class without incurring the overhead of “composite” semantics.

6.3.4 References

The MOF Model provides two constructs for modeling relationships between Classes (i.e., Associations and Attributes). While MOF Associations and Attributes are similar from the information modeling standpoint, they have important differences from the standpoints of their computational models and their corresponding mapped interfaces.

NOTE: Attributes can also model relationships between Classes and DataTypes, but that is not relevant to this point.

Associations offer a “query-oriented” computational model. The user performs operations on an object that notionally encapsulates a collection of links:

- Advantage: The association objects allow the user to perform “global” queries over all relationships, not just those for a given object.
- Disadvantage: The client operations for accessing and updating relationships tend to be more complex.

© ISO/IEC 2005 - All rights reserved

Attributes offer a “navigation-oriented” computational model. The user typically performs get and set operations on an attribute.

- Advantage: The get and set style of interfaces are simpler, and tend to be more natural for typical metadata oriented applications that “traverse” a metadata graph.
- Disadvantage: Performing a “global” query over a relationship expressed as an Attribute is computationally intensive.

The MOF Model provides an additional kind of Class feature called a Reference that provides an alternative “Attribute like” view of Associations. A Reference is specified by giving the following:

- a name for the Reference in its Class,
- an “exposed” Association End in some Association whose type is this Class or a super-Class of this Class, and
- a “referenced” Association End, which is the “other” end of the same Association.

Defining a Reference in a Class causes the resulting interface to contain operations with signatures that are identical to those for an “equivalent” Attribute. However, rather than operating on the values in an attribute slot of a Class instance, these operations access and update the Association, or more precisely a projection of the Association. This is illustrated in UML-like notation in Figure 6.4.

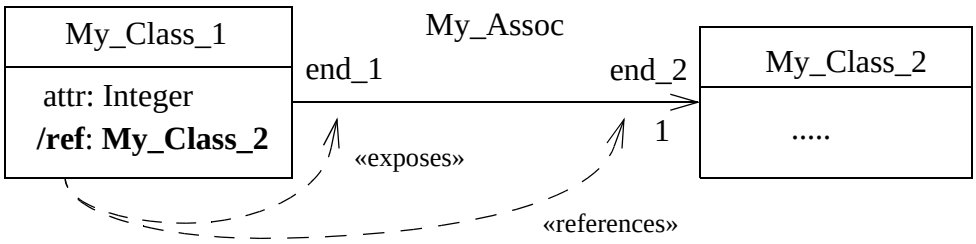

Figure 6.4 - An example of a Reference

Figure 6.4 shows a Class called My_Class_1 that is related to My_Class_2 by the Association My_Assoc. My_Class_1 has an Attribute called “attr” whose type is Integer. In addition, it has a Reference called “ref” that references “end2” of the Association. This provides an API for “ref” that allows a user to access and update a My_Class_1 instance’s link to a My_Class_2 instance using get and set operations.

NOTE: Strictly speaking, the UML notation in the diagram shows “ref” as a derived attribute of My_Class_1 with type of My_Class_2.

The example above shows a Reference that “exposes” an Association End with a multiplicity of “[1..1].” References can actually expose ends with any valid multiplicity specification. The resulting Reference operations are similar to those for an Attribute with the same multiplicity. However, since MOF Associations do not allow duplicates, Association Ends and therefore References must always have their multiplicity “is_unique” flag set to true.

There are some important restrictions on References:

- When the “is_navigable” property of an Association End is false, it is not legal to define a Reference that “references” that Association End.

© ISO/IEC 2005 - All rights reserved

- An M1 instance of a Class that “references” an Association cannot be used to make a link in an instance of the Association in a different extent. This restriction is described in 8.11.1, “The Reference Closure Rule,” on page 153.

6.3.5 DataTypes

Metamodel definitions often need to use attribute and operation parameter values that have types whose values do not have object identity. The MOF provides the metamodeling concept of a DataType to fill this need.

DataTypes can represent two kinds of data type:

1. Primitive data types like Boolean, Integer, and String are the basic building blocks for expressing state. The MOF defines six standard data types that are suitable for technology neutral metamodeling. (Other primitive data types can be defined by specific technology mappings or as user or vendor-specific extensions. However, the core MOF specification says nothing about what they mean.)
2. Data type constructors allow the metamodeler to define more complex data types. The MOF’s standard data type constructors are enumeration types, structure types, collection types, and alias types.

See 7.4.7, “DataType (abstract),” on page 55 and the following sub clauses for more details on how DataTypes subtypes are used to express types.

6.3.6 Packages

The Package is the MOF Model construct for grouping elements into a metamodel. Packages serve two purposes.

1. At the M2 level, Packages provide a way of partitioning and modularizing the metamodel space. Packages can contain most kinds of model element (e.g., other Packages, Classes, Associations, DataTypes, Exceptions, Constants, and so on).
2. At the M1 level, Package instances act as the outermost containers for metadata. Indirectly, they also define the scope boundaries of Association link sets and of “classifier level” Attributes and Operations on Class instances (see 8.8.4, “Package Extents,” on page 147).

The MOF Model provides four mechanisms for metamodel composition and reuse (i.e., generalization, nesting, importing, and clustering). These are described in the following sub clauses.

6.3.6.1 Package Generalization

Packages may be generalized by (inherit from) one or more other Packages in a way that is analogous to Class generalization described in 6.3.1.5, “Class Generalization,” on page 15. When one Package inherits from another, the inheriting (sub-) Package acquires all of the metamodel elements belonging to the (super-) Package it inherits from. Package inheritance is subject to rules that prevent name collision between inherited and locally defined metamodel elements.

At the M1 level, a sub-Package instance has the ability to create and manage its own collections of Class instances and Links. This applies to the Classes and Associations that it defines explicitly, and to those that it acquires by inheritance.

The relationship between instances of the super- and sub-Packages is similar to relationship between instances of super- and sub-Classes:

- A sub-Package instance is type substitutable for instances of its super-Packages (i.e., the sub-Package instance “IS_A” super-Package instance).

© ISO/IEC 2005 - All rights reserved

- A sub-Package instance does not use or depend on an instance of the super-Package (i.e., there is no “IS_PART_OF” relationship).

Packages may be defined as “root” or “leaf” Packages (with analogous meaning to “root” and “leaf” Classes), but “abstract” Packages are not supported.

6.3.6.2 Package Nesting

A Package may contain other Packages, which may in turn contain other Packages. Model elements defined in nested Packages may be strongly coupled to other model elements in the same containment. For example, a Class in a nested Package have a Reference that links it via an Association in its context, or its semantics could be covered by a user-defined Constraint that applies to the enclosing Package.

A nested Package is a component of its enclosing Package. Since, in general, the model elements in a nested Package can be inextricably tied to its context, there are some significant restrictions on how nested Packages can be composed. In particular, a nested Package may not:

- generalize or be generalized by other Packages.
- be imported or clustered by other Packages.

Nested Packages are not directly instantiable. No factory objects or operations are defined for nested Package instances. An M1 level instance of a nested Package can only exist in conjunction with an instance of its containing Package. Conceptually, a nested Package instance is a component of an instance of its containing Package.

NOTE: The main effect of nesting one Package inside another is to partition the concepts and the namespace of the outer Package. Nesting is not a mechanism for reuse. Indeed when a Package is nested, the options for reusing its contents are curtailed.

6.3.6.3 Package Importing

In many situations, the semantics of Package nesting and generalization do not provide the best mechanism for metamodel composition. For example, the metamodeler may wish to reuse some elements of an existing metamodel and not others. The MOF provides an import mechanism to support this.

A Package may be defined as importing one or more other Packages. When one Package imports another, the importing Package is allowed to make use of elements defined in the imported one Package. As a shorthand, we say that the elements of the imported Package are imported.

Here are some examples of how a Package can reuse imported elements. The importing Package can declare:

- Attributes, Operations, or Exceptions using imported Classes or DataTypes,
- Operations that raise imported Exceptions,
- DataTypes and Constants using imported DataTypes or Constants,
- Classes whose supertypes are imported Classes, and
- Associations for which the types of one or both Association Ends is an imported Class.

At the M1 level, an instance of an importing Package has no explicit relationship with any instances of the Packages that it imports. Unlike a sub-Package, an importing Package does not have the capability to create instances of imported Classes. A client must obtain any imported Class instances it needs via a separate instance of the imported Package.

© ISO/IEC 2005 - All rights reserved

6.3.6.4 Package Clustering

Package clustering is a stronger form of Package import that binds the importing and imported Package into a "cluster." As with ordinary imports, a Package can cluster a number of other Packages, and can be clustered by a number of other Packages.

An instance of a cluster Package behaves as if the clustered Packages were nested within the Package. That is, the lifecycle of a clustered Package instance is bound to the lifecycle of its cluster Package instance. In particular:

- When the user creates an instance of a cluster Package, an instance of each of its clustered Packages is created automatically.
- The instances of the clustered Packages created above all belong to the same cluster Package extent.
- Deleting a cluster Package instance automatically deletes its clustered Packaged instances, and the clustered Package instances cannot be deleted except as part of the deletion of the cluster Package instance.

However, unlike a nested Package, it is possible to create an independent instance of a clustered Package. Also, in some situations clustered Package instances are not strictly nested.

NOTE: It is possible to cluster or inherit from Packages that cluster other Packages. The impact of this on M1 level instance relationships is discussed in 8.8.4, "Package Extents," on page 147.

In summary, the relationship between the M1 level instances in a Package cluster is that each clustered Package instance is a component of the cluster Package instance. Unlike nested Packages, there is no composite relationship between the M2 level Packages.

6.3.6.5 Summary of Package Composition Constructs

The properties of the four Package composition mechanisms defined by the MOF Model are summarized by Table 6.1.

Table 6.1 - Package Composition Constructs

Metamodel Construct	Conceptual Relationship	M2 level Relationship Properties	M1 level Relationship Properties
Nesting	P1 contains P2	P1 ◆—— P2	*P1* ◆—— *P2*
Generalization / Inheritance	P1 generalizes P2	P2 - - ▷ P1	*P2* ——▷ *P1*
Importing	P1 imports P2	P1 - - - ▷ P2	none
Clustering	P1 clusters P2	P1 - - ▷ P2	*P1* ◇—— *P2* or none

The symbology of the table is based on UML; that is, a filled diamond means composition, a hollow diamond means aggregation, a hollow triangle means inheritance, and a dotted arrow means "depends on."

Note that P1 and P2 denote different (though related) things in different columns of the table:

- In column 2, they denote conceptual M2 level Packages in a metamodel.
- In column 3, they denote both the conceptual M2 level Packages, and the objects that represent them in a reified metamodel.

© ISO/IEC 2005 - All rights reserved

- In column 4, they denote M1 level Package instances (when underlined) or their types.

6.3.7 Constraints and Consistency

The MOF Model constructs described so far allow the metamodeler to define a metadata information that comprises nodes (Classes) with attached properties (Attributes / DataTypes) and relationships between nodes (Associations). While the above constructs are sufficient to define an "abstract syntax" consisting of metadata nodes and links, this syntax typically needs to be augmented with additional consistency rules.

This sub clause describes the MOF Model's support for consistency rules and model validation.

6.3.7.1 Constraints

The MOF Model defines an element called Constraint that can be used to attach consistency rules to other metamodel components. A Constraint comprises:

- a constraint name,
- a "language" that identifies the language used to express the consistency rules,
- an "expression" in the language that specifies a rule,
- an "evaluation policy" that determines when the rule should be enforced, and
- a set of "constrained elements."

A Constraint expression is an expression in some language that can be "evaluated" in the context of a metamodel to decide if it is valid. The MOF specification does not define or mandate any particular languages for Constraint expressions, or any particular evaluation mechanisms. Indeed, it is legitimate for Constraints to be expressed in informal language (e.g., English) and for validation to be implemented by ad-hoc programming. However, the Constraints that are part of the MOF Model specification itself are expressed in Object Constraint Language (OCL) as described in the UML specification.

The evaluation policy property of a Constraint determines whether the consistency rule should be enforced immediately or at a later time. Figure 6.5 gives a simple example that will be used to illustrate the need for evaluation policies.

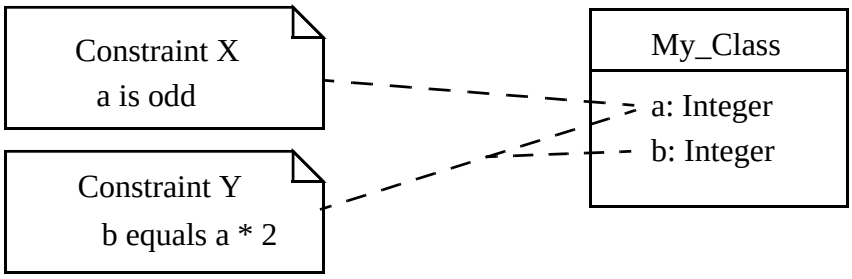

Figure 6.5 - Examples of Constraints

In Figure 6.5, Constraint X constrains only Attribute a while Constraint B constrains both Attributes a and b.

© ISO/IEC 2005 - All rights reserved

It is feasible to check the first Constraint (X: “a is odd” on the Attribute “a”) at any time. It could be checked whenever a value for “a” is supplied (e.g., at instance creation and when “a” is updated). An exception would be raised if the new value for “a” was even. Alternatively, constraint checking could be deferred to a later point (e.g., when the user requests validation of a model).

The second constraint (Y: “b equals a * 2” on both Attributes “a” and “b”) is another matter. If a server enforces Y on every update, the user would never be able to change the values of either “a” or “b.” No matter which order the user invoked the operations, the updates would raise an exception. Instead, enforcement of Y must be deferred until both “a” and “b” have been updated.

NOTE: The Constraint construct is intended to be used for specifying consistency rules for models rather than for defining the computation behavior of (for example) Operations. It is “bad style” to specify Constraint expressions that have side-effects on the state of a model, not least because it is unspecified when Constraints are evaluated.

6.3.7.2 Structural Consistency

As noted previously, a MOF-based metamodel defines an “abstract syntax” for metadata. Some aspects of the abstract syntax are enforced by the corresponding metadata server’s IDL. For example, the operation that creates a link for an Association has a type signature that prevents the user from creating a link with the wrong kind of Class instances. However, some aspects of the abstract syntax can only be enforced by runtime structural consistency checks. While most of the structural checks are made immediately, checks for “underflow” often need to be deferred.

It is not practical for a metamodel to specify *a priori* all possible things that can go wrong in a MOF-based metadata server. It is therefore necessary to recognize that a MOF server may need to perform a variety of runtime checks that are neither defined or implied by the metamodel. These include additional metadata validation that is not specified by the metamodel, resource and access control checks, and internal error checking.

6.3.7.3 Consistency Checking Mechanisms

The MOF specification provides a lot of latitude for metadata server implementations in the area of constraint checking or validation.

- Support for checking of Constraints is not mandatory. In particular, there is no requirement to support any particular language for Constraint expressions.
- The set of events (if any) that may trigger deferred checking is not specified. No general APIs are specified for initiating deferred consistency checking.
- Persistence and interchange of metadata, which is in an inconsistent state may be allowed. (Indeed, this would seem to be a prerequisite for some styles of metadata acquisition.)
- There are no specified mechanisms for ensuring that validated metadata remains valid, or that it does not change.

The one aspect of consistency checking that is mandatory is that a metadata server must implement all structural consistency checks that are labeled as immediate.

6.3.8 Miscellaneous Metamodeling Constructs

This sub clause describes the remaining significant elements of the MOF Model.

© ISO/IEC 2005 - All rights reserved

6.3.8.1 Constants

The Constant model element allows the metamodeler to define simple bindings between a name and a constant value. A Constant simply maps onto a constant declaration in (for example) the IDL produced by the MOF IDL mapping.

6.3.8.2 Exceptions

The Exception model element allows the metamodeler to declare the signature of an exception that can be raised by an Operation. An Exception simply maps onto (for example) an IDL exception declaration.

6.3.8.3 Tags

The Tag model element is the basis of a mechanism that allows a “pure” MOF metamodel to be extended or modified. A Tag consists of:

- a name that can be used to denote the Tag in its container,
- a “tag id” that denotes the Tag’s kind,
- a collection of zero or more “values” associated with the Tag, and
- the set of other model elements that the Tag is “attached” to.

The meaning of a model element is (notionally) modified by attaching a Tag to it. The Tag’s “tag id” categorizes the intended meaning of the extension or modification. The “values” then further parameterize the meaning.

As a general rule, the definition of values and meanings for “tag id” strings is beyond the scope of the MOF specification. The specification recommends a tag id naming scheme that is designed to minimize the risk of name collision, but use of this scheme is not mandatory; see 7.4.28, “Tag,” on page 83.

One exception to this is the MOF to IDL Mapping. This defines some standard tag ids that allow a metamodel to influence the IDL mapping; see 9.6, “Standard Tags for the IDL Mapping,” on page 194 for the complete list. For example:

- “IDL Substitute Name” provides an alternative IDL identifier for an element in a metamodel, and
- “IDL Prefix” allows the metamodeler to specify the IDL “prefix” for a top-level Package.

6.4 Metamodels and Mappings

The previous sub clauses outlined the overall metadata architecture for the MOF, and the metamodeling constructs provided by the MOF Model. This sub clause describes the Mapping approach that is used to instantiate MOF metamodels and metadata in the context of a given implementation technology.

This sub clause is organized as follows. The first sub clause outlines the purpose and structure of MOF Mappings. The next two sub clauses give high-level overviews of the OMG MOF technology mappings defined to date. The final sub clause explains how the standard mappings are applied to the MOF Model to produce the OMG IDL for the MOF Model server and an XML DTD for metamodel interchange.

6.4.1 Abstract and Concrete Mappings

MOF Mappings relate an M2-level metamodel specification to other M2 and M1-level artifacts, as depicted in Figure 6.6.

© ISO/IEC 2005 - All rights reserved

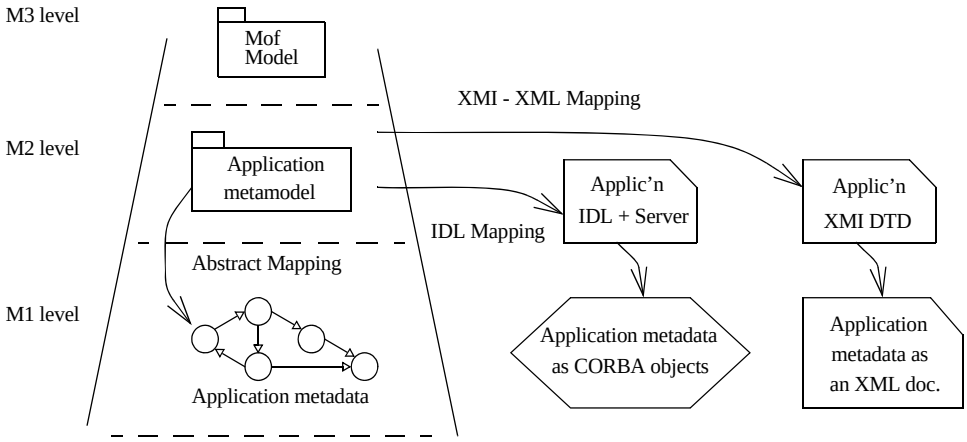

Figure 6.6 - The function of MOF Technology Mappings

Figure 6.6 depicts the Mapping derived relationships for an application metamodel as follows:

- The Abstract mapping (defined in "The MOF Abstract Mapping" Clause) fleshes out a MOF metamodel into an abstract information model; that is, by spelling out the logical structure of the metadata described by the metamodel.
- The IDL Mapping (6.4.2, "The MOF Metamodel to IDL Mapping," on page 26) produces the standard OMG IDL and associated behavioral semantics for metaobjects that can represent metadata conforming to the metamodel.
- The XML Mapping (see 6.4.3, "The MOF Metamodel to XML Mappings," on page 27) produces the standard XML DTD for interchanging metadata conforming to the metamodel.

The Abstract Mapping has two roles: 1) it serves to define the "meaning" of a metamodel, and 2) it provides a point of alignment for current and future MOF technology Mappings.

Since the IDL and XML Mappings are both aligned with the Abstract Mapping there is a precise one-to-one correspondence between abstract metadata and metadata expressed as XMI documents or CORBA metaobjects. This correspondence holds for all metamodels. More significantly, it should also hold for any future metamodel Mappings (e.g., to Java or DCOM technology) that are aligned with the Abstract Mapping.

6.4.2 The MOF Metamodel to IDL Mapping

The MOF IDL Mapping produces a specification for a CORBA metadata service from a MOF metamodel specification. The OMG IDL interfaces and associated behavioral semantics are specified in the "MOF to IDL Mapping" clause and "The Reflective Module" clause. These interfaces support creating, updating, and accessing metadata in the form of CORBA objects, either using "specific" interfaces that are tailored to the metamodel or "generic" interfaces that are metamodel independent.

The MOF IDL Mapping places some additional restrictions in MOF metamodels beyond those set out in the "MOF Model and Interfaces" clause. See 9.5, "Preconditions for IDL Generation," on page 192 for details.

© ISO/IEC 2005 - All rights reserved

6.4.3 The MOF Metamodel to XML Mappings

Interchange of MOF-based metadata is defined in a separate OMG specification. The XMI (XML-based metadata Interchange) specification leverages the W3C's XML (eXtensible Markup Language) technology to support the interchange of metadata and metamodels between MOF-based and other metadata repositories.

The XMI 1.1 specification (formal/2000-11-02) has two main parts:

1. The "XML DTD Production Rules" define a uni-directional mapping from a MOF metamodel to an XML DTD (Document Type Definition) for metadata interchange documents.
2. The "XML Document Production Rules" define a bi-directional mapping between an XML document (structured according to the above DTD) and MOF-based metadata that (implicitly) conforms to the Abstract Mapping.

6.4.4 Mappings of the MOF Model

The MOF IDL mapping has been applied to the MOF Model to produce the normative CORBA IDL for a metamodel repository. The XMI specification has been applied to the MOF Model to produce the normative XMI DTD for metamodel interchange, and a normative rendering of the MOF Model in the interchange format. These and other electronic renderings of the MOF metamodel are described in 4, "List of Documents," on page 2.

© ISO/IEC 2005 - All rights reserved

© ISO/IEC 2005 - All rights reserved

7 MOF Model and Interfaces

7.1 Overview

This Clause describes the model that defines the MOF. The MOF provides a set of modeling elements, including the rules for their use, with which to construct models. Specifically, the MOF modeling elements support development of meta-models. This focus enables the MOF to provide a more domain-specific modeling environment for defining meta-models instead of a general-purpose modeling environment.

A well-designed modeling tool or facility should be based on a meta-model that represents the modeling elements and the rules provided by the tool or facility.

Every meta-model is also a model. If the MOF Model described in this sub clause is the meta-model for the MOF, where is the model for this meta-model? Formally, the MOF is defined in itself; that is, the modeling elements defined in the MOF Model and provided by the MOF are used to define the MOF Model itself. In essence, the MOF Model is its own meta-model. However, this circular definition does not support presentation of the model. Therefore, this specification describes the MOF narratively and through the use of UML notation, tables, and Object Constraint Language (OCL) expressions.

Note that the use of UML notation is a convenience to the designers of the MOF and to the readers of the MOF specification. The semantics of the MOF Model are completely defined in the MOF specification and do not depend on the semantics of any other model. The MOF interfaces used to manipulate meta-models are dependent on CORBA in that these interfaces are specified using CORBA IDL.

A significant amount of the MOF Model syntax and semantics definition is constraint-based. This specification describes the constraint expressions as clearly as possible. In addition, the specification provides a reference to the OCL expression that defines each constraint.

The OCL, which is defined in the UML 1.4 specification (http://www.omg.org/technology/documents/formal/uml.htm), provides a small set of language elements used to define expressions. As an expression language, OCL cannot change the state of objects; however, it can express constraints (including invariants, preconditions, and post-conditions). OCL expressions use operations defined in the MOF Model with the attribute isQuery set to TRUE. (Such operations do not change the state of the object.) To ensure complete specification of constraints, this document provides OCL definitions for MOF-defined operations used in OCL expressions. In addition, to avoid ambiguity or misinterpretation this specification uses OCL to define a few of the most complex concepts of the MOF Model.

The interfaces through which the MOF is utilized are generated from the MOF Model. However, these interfaces do not provide the semantic information necessary to determine the behavior of their operations. Therefore, it is essential to understand the MOF in terms of its model and related semantics, not just its interfaces.

7.2 How the MOF Model is Described

This Clause describes the modeling elements that comprise the MOF Model and provide the building blocks for meta-models. Because these elements are formally described with the MOF Model itself, the characteristics used to describe the model are the same characteristics provided by the model.

The following sub clauses briefly describe the conventions that this specification uses to define the model elements and their characteristics, with a few exceptions noted.

© ISO/IEC 2005 - All rights reserved

7.2.1 Classes

Classes are the fundamental building blocks of MOF meta-models and the MOF Model. A Class can have three kinds of features; Attributes, References, and Operations. They may inherit from other Classes, and may be related to other Classes by Associations. Classes are presented in detail in 8.5, “Semantics of Class Instances,” on page 141.

The MOF uses the term Class with a meaning that is similar to that of Class in UML. A MOF Class is an abstract specification or classification of meta-objects that includes their state, their interfaces, and (at least informally) behavior. A Class specification is sufficient to allow the generation of concrete interfaces with well defined semantics for managing meta-object state. However, a MOF Class specification does not include any methods to implement meta-object behavior.

The Classes that make up the MOF Model are introduced in 7.3, “The Structure of the MOF Model,” on page 36 and specified in detail in 7.4, “MOF Model Classes,” on page 41. Each Class is defined in terms of its name(s), its super-Classes, the Classes whose instances it can contain, its attributes, its references, its operations, its constraints, and whether it is abstract or concrete.

NOTE: Except where stated, the order in which “MOF Model Classes” on page 41 introduces Classes and their component features is not normative. The normative order is defined in the XMI for the MOF Model that may be found in the Preface. This order determines the order in which elements appear in the generated IDL, and is in theory significant.

This document uses a hybrid textual and tabular notation to define the important characteristics of each Class in the MOF Model. The notation defines defaults for most characteristics, so that the Class definitions need only explicitly specify characteristics that are different from the default. The following text explains the notation used for Classes and their characteristics.

7.2.1.1 Class Heading

Each Class in the MOF Model is introduced by a second level sub clause heading. The heading defines the standard ModelElement name for the Class. The Classes name on the heading line can be followed by the word “*abstract*” or by a “*substitute_name*” for some mapping. For example, the following:

3.4.1ModelElement (abstract)

introduces a Class called “ModelElement” and defines its Clause “isAbstract” flag to have the value “true.” On the other hand, the following:

3.4.11Attribute (idl_substitute_name“MofAttribute”)

introduces a Class called “Attribute” and defines its substitute name (for the IDL mapping) as “MofAttribute.” The latter information is encoded using a Tag whose “tagId” is “idl_substitute_name” and whose “values” consist of the Any-ized string “MofAttribute.”

Unless stated otherwise each Class in the MOF Model has “isAbstract” set to false, and has no attached Tags.

NOTE: The MOF uses “abstract Class” in the same sense as UML, and also Java and many other object oriented programming languages. There is no relationship with the IDL keyword “abstract” introduced in CORBA 2.3.

The paragraph or paragraphs following a Class heading give a description of the Class, its purpose, and its meaning.

© ISO/IEC 2005 - All rights reserved

7.2.1.2 Superclasses

The “Superclasses” heading lists the MOF Classes that generalize the Class being described. In the MOF context, generalization is another term for inheritance. Saying that a Class A generalizes a Class B, means the same as saying that Class B inherits from Class A. The sub-Class (B) inherits the contents of the super-Class (A). Multiple inheritance is permitted in the MOF.

This heading is always present, since with the sole exception of ModelElement, all Classes in the MOF Model have super-Classes.

7.2.1.3 Contained Elements

Instances of the sub-Classes of NameSpace can act as containers of other elements. If present, the “Contained Elements” heading lists the Classes whose instances may be contained by an instance of this container Class. It also gives the index of the MOF Model Constraint that defines the containment rule for the Class. For more details, see 7.3.3, “The MOF Model Structure,” on page 38. In particular, Table 7.5 on page 137 expresses the MOF Class containment rules in a concise form.

If the “Contained Elements” heading is absent, instances of the Class may not contain other instances. This occurs if the Class is an abstract Class (and therefore has no instances), or if the Class is not derived from the Namespace Class.

7.2.1.4 Attributes

The “Attributes” heading lists the Attributes for a Class in the MOF Model. Attributes that are inherited from the super-Classes are not listed. If the “Attributes” heading is missing, the Class has no Attributes.

All Attributes defined in the MOF Model have a “visibility” of “public_vis.” All have a “type” that is represented using a DataType, and therefore all have aggregation semantics of “none.” The remaining characteristics of Attributes are defined using the notation described in Table 7.1.

Table 7.1 - Notation for Attribute Characteristics

Entry	Description
type:	This entry defines the base type for the Attribute. It gives either the name of a DataType defined in 7.6, “MOF Model Data Types,” on page 96, or the name of a standard MOF primitive data type (e.g., “Boolean” or “String”) defined in 7.10, “The PrimitiveTypes Package,” on page 134. The base type is represented by the Attribute’s “type.”
multiplicity:	This entry defines the “multiplicity” for the Attribute, consisting of its “lower” and “upper” bounds, an “isOrdered” flag, and an “isUnique” flag. See 7.6.2, “MultiplicityType,” on page 96, and 8.6.2, “Multiplicity,” on page 142 for more details. The multiplicity for an Attribute is expressed as follows: • The “lower” and “upper” bounds are expressed as “exactly one,” “zero or one,” “zero or more,” and “one or more.” • If the word “ordered” appears, “isOrdered” should be true. If it is absent, “isOrdered” should be false. • If the word “unique” appears, “isUnique” should be true. If it is absent, “isUnique” should be false.
changeable:	This optional entry defines the “isChangeable” flag for the Attribute. If omitted, “isChangeable” is true.

© ISO/IEC 2005 - All rights reserved

Table 7.1 - Notation for Attribute Characteristics

derived from:	This optional entry either describes the derivation of a derived Attribute, or if the entry is present, the Attribute's "isDerived" flag will be true. If it is absent, the flag will be false.
scope:	This optional entry defines the "scope" of an Attribute as either "instance_level" or "classifier_level." If the entry is absent, the Attribute's "scope" is "instance_level."

7.2.1.5 References

The "References" heading lists the References for a Class in the MOF Model. A Reference connects its containing Class to an Association End belonging to an Association that involves the Class. This allows a client to navigate directly from an instance of the Class to other instance or instances that are related by links in the Association. If the "References" heading is absent, the Class has no References.

A Class involved in an Association may or may not have a corresponding Reference. A Reference means that a client can navigate to instances of the other Class; however, this comes at the cost of some restrictions. In particular, if one or both Classes in an Association have References for that Association, the Reference Closure rule restricts the creation of links between instances in different "extents" (see 8.11.1, "The Reference Closure Rule," on page 153).

NOTE: The modeling of navigation in MOF differs from UML. In UML, mechanisms for navigating links are available when the "isNavigable" flag is true for a given AssociationEnd. In this case, stronger uniqueness constraints on AssociationEnd names mean that they are unique within the namespaces of the Association and all Classes involved and their sub-Classes. This means that the AssociationEnd names uniquely bind to a "navigator" operation in each context in which navigation might be used.

Most characteristics of References in the MOF Model are either common across all References or derived from other information:

- The "visibility" of all References in the MOF Model is "public_vis."
- The "scope" of all References is "instance_scope."
- The "type" of all References is the same as the "type" of the AssociationEnd it references.
- The "multiplicity" of all References is the same as the "multiplicity" of the AssociationEnd it references.

The variable characteristics of References are defined or documented using the notation described in Table 7.2.

Table 7.2 - Notation for Reference characteristics

Entry	Description
class:	This entry documents the base type of the Reference and is represented as its "type." Note that the "type" of a Reference must be the same as the "type" of the referenced AssociationEnd.
defined by:	This entry defines the Association and AssociationEnd that the Reference is linked to via a RefersTo link.

© ISO/IEC 2005 - All rights reserved

Table 7.2 - Notation for Reference characteristics

multiplicity:	This entry documents the “multiplicity” characteristics for the Reference. These are written the same way as Attribute “multiplicity” characteristics, except that “unique” is omitted because its value is predetermined (see 7.2.2.2, “Ends,” on page 34). Note the following: • the OCL constraints on MultiplicityType and AssociationEnd mean that the “isUnique” field must be “false” if the “upper” bound is 1 and “true” otherwise, and • the “multiplicity” settings for an AssociationEnd and its corresponding Reference(s) must be the same.
changeable:	This optional entry defines the setting of the Reference’s “isChangeable” flag. If the entry is absent, the “isChangeable” flag is true.
inverse:	This optional entry documents the “inverse” Reference for this Reference; that is, the Reference on the link related Class that allows navigation back to this Reference’s Class. If this entry is absent, the Reference does not have an inverse Reference.

7.2.1.6 Operations

The “Operations” heading lists the Operations for a Class in the MOF Model. If the heading is absent, the Class has no Operations.

All Operations for Classes in the MOF Model have “visibility” of “public_vis.” The remaining characteristics of References are defined using notation described in Table 7.3.

Table 7.3 - Notation for Operation Characteristics

Entry	Description
return type:	This optional entry defines the “type” and “multiplicity” of the Operation’s return Parameter; that is, the one with “direction” of “return_dir.” The “type” is denoted by a name of a Class or DataType defined the MOF Model, or a name of standard MOF primitive data type. The “multiplicity” is expressed like an Attribute “multiplicity” (see Table 7.2 on page 32), except that when it is absent, the “multiplicity” defaults to “exactly one.” The return Parameter (if it exists) should be the first contained Parameter of the Operation. If this entry is absent or says “none,” the Operation does not have a return Parameter.
isQuery:	This optional entry defines the Operation’s “isQuery” flag. If it is absent, the “isQuery” flag has the value false.
scope:	This optional entry defines the Operation’s “scope.” If it is absent, the Operation has a “scope” of “instance_level.”
parameters:	This entry defines the Operation’s non-return Parameter list in the order that they appear in the Operation’s signature. Each parameter is defined with the following format: name, direction type multiplicity. If the “multiplicity” is not explicitly specified, it defaults to “exactly one.” If the entry simply says “none,” the Operation has no non-return Parameters.
exceptions:	This optional entry defines the list of Exceptions that this Operation may raise in the order that they appear in the Operation’s signature. If it is absent, the Operation raises no Exceptions.
operation semantics:	This optional entry simply gives a cross reference to the OCL defining the Operation’s semantics. Note that the MOF Model does not provide a standard way of representing an Operation’s semantic specification, and it is not included in the normative XMI serialization of the MOF Model.

© ISO/IEC 2005 - All rights reserved

7.2.1.7 Constraints

The “Constraints” heading lists the Constraints that are attached to this Class in the MOF Model. The OCL for the Constraints may be found in 7.9.4, “The MOF Model Constraints,” on page 105. Each listed Constraint “constrains” the Class, and is also contained by it.

7.2.1.8 IDL

The “IDL” heading shows an excerpt of the MOF Model IDL that corresponds to this Class. The excerpts, which are part of the “Model” module given in the “MOF to IDL Summary” clause, consist of a Class proxy interface and an Instance interface. For information on these interfaces, refer to the “MOF to IDL Mapping” clause.

7.2.2 Associations

The Associations in the MOF Model are defined in 7.5, “MOF Model Associations,” on page 85.

Associations describe relationships between instances of Classes. In short, an Association relates two Classes (or relates one Class to itself) to define a “link set” that contains two-ended “links” between instances of the Classes. The properties of an Association rest mostly in its two AssociationEnds. Refer to 8.9, “Semantics of Associations,” on page 149 for a more detailed explanation.

7.2.2.1 Association Heading

Each Association in the MOF Model is introduced by a second level sub clause heading in “MOF Model Associations” on page 85. The heading defines the standard ModelElement name for the Association. The Classes name on the heading line can be followed by the word “*derived*.” For example, the following:

3.5.4 Exposes (derived)

introduces an Association called “Exposes” and defines its Clause “isDerived” flag to be true. If the word “*derived*” is not present, the Association’s “isDerived” flag is false.

The paragraph or paragraphs following an Association heading give a description of the Association, its purpose, and its meaning.

7.2.2.2 Ends

The “Ends” heading defines the two AssociationEnds for an Association in the MOF Model. The two AssociationEnds are defined by giving their “name” values and defining the remaining characteristics in tabular form.

© ISO/IEC 2005 - All rights reserved

Every AssociationEnd in the MOF Model has both “isNavigable” and “isChangeable” set to true. The remaining characteristics of AssociationEnds are defined using notation described in the table below.

Entry	Description
class:	This entry specifies the Class whose instances are linked at this end of the Association. This is represented by the AssociationEnd’s “name” attribute.
multiplicity:	This entry defines the AssociationEnd’s “multiplicity” attribute. This is expressed in the same way as References (i.e., uniqueness is implicit - see 7.2.1.5, “References,” on page 32). Note the following: • the OCL constraints on MultiplicityType and AssociationEnd mean that the “isUnique” field must be “false” if the “upper” bound is 1 and “true” otherwise, and • the “multiplicity” settings for an AssociationEnd and its corresponding Reference(s) must be the same.
aggregation:	This optional entry defines the AssociationEnd’s “aggregation” attribute as one of “composite,” “shared,” or “none” (see 8.9.4, “Association Aggregation,” on page 152). If the entry is absent, the AssociationEnd’s “aggregation” attribute takes the value “none.”

7.2.2.3 Derivation

The “Derivation” heading defines how a derived Association should be computed. It may include marker for an OCL rule defined in “[C-59]” on page 122.

7.2.2.4 IDL

The “IDL” heading shows an excerpt of the MOF Model IDL that corresponds to this Association. These excerpts, which are part of the “Model” module given in the MOF IDL Summary Clause, consist of an Association interface and related IDL data types. For more information, refer to the MOF to IDL Mapping Clause.

7.2.3 DataTypes

The DataTypes that form part of the MOF Model are described in 7.6, “MOF Model Data Types,” on page 96. All DataTypes in the MOF Model have “visibility” of “public_vis.” The settings of the other attributes are “isAbstract” = false, “isRoot” = true, and “isLeaf” = true as required by various MOF Model constraints.

The DataTypes used in the MOF Model are instances of PrimitiveType, StructureType, or EnumerationType. A StructureType’s StructureFields (and their order) are as shown in the IDL. Similarly, an EnumerationType’s “labels” and their order are as shown in the IDL.

The remaining characteristics of a DataType are its “name” (given in the sub clause heading), and its container (given by the “Container” subheading). If the “Container” subheading is absent, the DataType is contained by the Model Package.

7.2.4 Exceptions

The Exceptions that form part of the MOF Model are described in 7.7, “MOF Model Exceptions,” on page 99.

All Exceptions in the MOF Model have “visibility” of “public_vis” and “scope” of “classifier_level.”

The remaining characteristics are the Exception’s

- “name” - given in the sub clause heading, and

© ISO/IEC 2005 - All rights reserved

- Parameters and Container, which are given in the corresponding headings.

If the Container heading is absent, the Exception is contained by the Model Package.

7.2.5 Constants

The Constants that form part of the MOF Model are described in 7.8, “MOF Model Constants,” on page 99.

The characteristics of a Constant are its

- “name” - given in the sub clause heading, and
- Container - given under the “Container” heading, and
- “type” and “value” that can be determined from the IDL.

If the “Container” heading is absent, the DataType is contained by the Model Package.

7.2.6 Constraints

The Constraints that form part of the MOF Model are described in 7.9, “MOF Model Constraints,” on page 101. The notation used for describing the constraints is described in 7.9.2.1, “Notation for MOF Model Constraints,” on page 101.

7.2.7 UML Diagrams

At various points in this clause, UML class diagrams are used to describe aspects of the MOF Model. To understand these diagrams, the reader should mentally map from UML modeling concepts to the equivalent MOF meta-modeling constructs.

There is one point in which this document’s use of UML notation requires explaining. In standard UML notation, an arrowhead on an Association line indicates that the Association is navigable in the direction indicated. Absence of an arrowhead can mean either that the Association is navigable or that it is navigable in both directions, depending on the context.

As was explained in 7.2.1.5, “References,” on page 32, the MOF models navigable Associations in a different way. Thus in this document, an arrowhead on one end of an Association means that a Reference exists on the Class at the opposite end that allows navigation in the indicated direction. If there are no arrowheads, there are References on the Classes at *both* ends of the Association.

7.3 The Structure of the MOF Model

This sub clause gives an overview of the structure of the MOF Model.

7.3.1 The MOF Model Package

The MOF Model, as it is currently defined, consists of a single non-nested Package called “Model.” This Package explicitly imports the “PrimitiveTypes” Package so that it can use the “Boolean,” “Integer,” and “String” PrimitiveType instances.

The class diagram in Figure 7.2 on page 39 shows the Classes and Associations of the “Model” Package. To aid readability, Class features, Association End names, DataTypes, and other details have been omitted from the diagram. These details are all specified in later sub clauses of this clause.

© ISO/IEC 2005 - All rights reserved

NOTE: This diagram (like other UML diagrams in this clause) is non-normative.

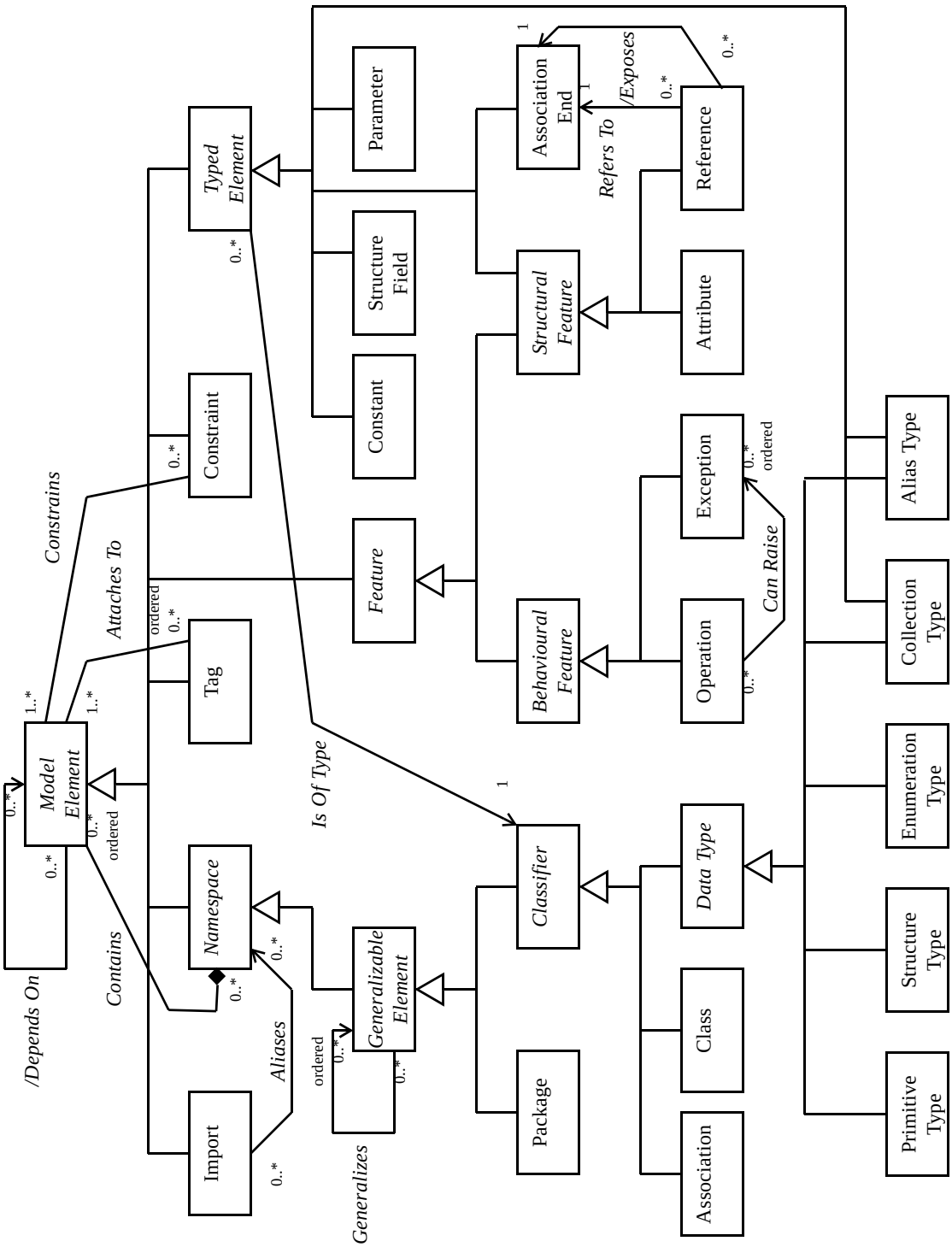

Figure 7.1 - The MOF Model Package

© ISO/IEC 2005 - All rights reserved

7.3.2 The MOF Model Service IDL

The “Model” Package is used to generate the CORBA IDL for the OMG MOF Model service using the MOF to IDL Mapping defined in the “MOF Abstract Mapping” clause through the “Reflective Module” clause. Relevant fragments of the resulting IDL is embedded in the Class, Association, DataType, and Exception descriptions in 7.4, “MOF Model Classes,” on page 41 through 7.7, “MOF Model Exceptions,” on page 99.

The IDL for the MOF Model service requires a “prefix” of “org.omg.mof.” To this end, the “Model” Package is defined to have an “idl_prefix” Tag with value “org.omg.mof.”

The IDL for the MOF Model services requires the “Attribute” and “Exception” elements to have IDL names “MofAttribute” and “MofException” respectively to avoid collision with IDL keywords. To this end, the “Attribute” Class and “Exception” Class have ‘idl_alternate_name” Tags with the values “MofAttribute” and “MofException” respectively.

7.3.3 The MOF Model Structure

The core structure of the MOF Model is shown in the class diagram in Figure 7.2. This diagram shows the key abstract Classes in the MOF Model and the key Associations between them.

7.3.3.1 Key Abstract Classes

The key abstract Classes in the MOF Model are as follows:

- ModelElement - this is the common base Class of all M3-level Classes in the MOF Model. Every ModelElement has a “name.”
- Namespace - this is the base Class for all M3-level Classes that need to act as containers in the MOF Model.
- GeneralizableElement - this is the base Class for all M3-level Classes that support “generalization” (i.e., inheritance).
- TypedElement - this is the base Class for M3-level Classes such as Attribute, Parameter, and Constant whose definition requires a type specification.
- Classifier - this is the base Class for all M3-level Classes that (notionally) define types. Examples of Classifier include Class and DataType.

7.3.3.2 Key Associations

The key Associations in the MOF Model are as follows:

- Contains - this Association relates a ModelElement to the Namespace that contains it (see 7.3.4, “The MOF Model Containment Hierarchy,” on page 40).
- Generalizes - this Association relates a GeneralizableElement to its ancestors (i.e., supertypes) and children (i.e., subtypes) in a model element inheritance graph. Note that a GeneralizableElement may not know about all of its subtypes.
- IsOfType - this Association relates a TypedElement to the Classifier that defines its type.
- DependsOn - this derived Association relates a ModelElement to others that its definition depends on. (It is derived from Contains, Generalizes, IsOfType, and other Associations not shown here.)

© ISO/IEC 2005 - All rights reserved

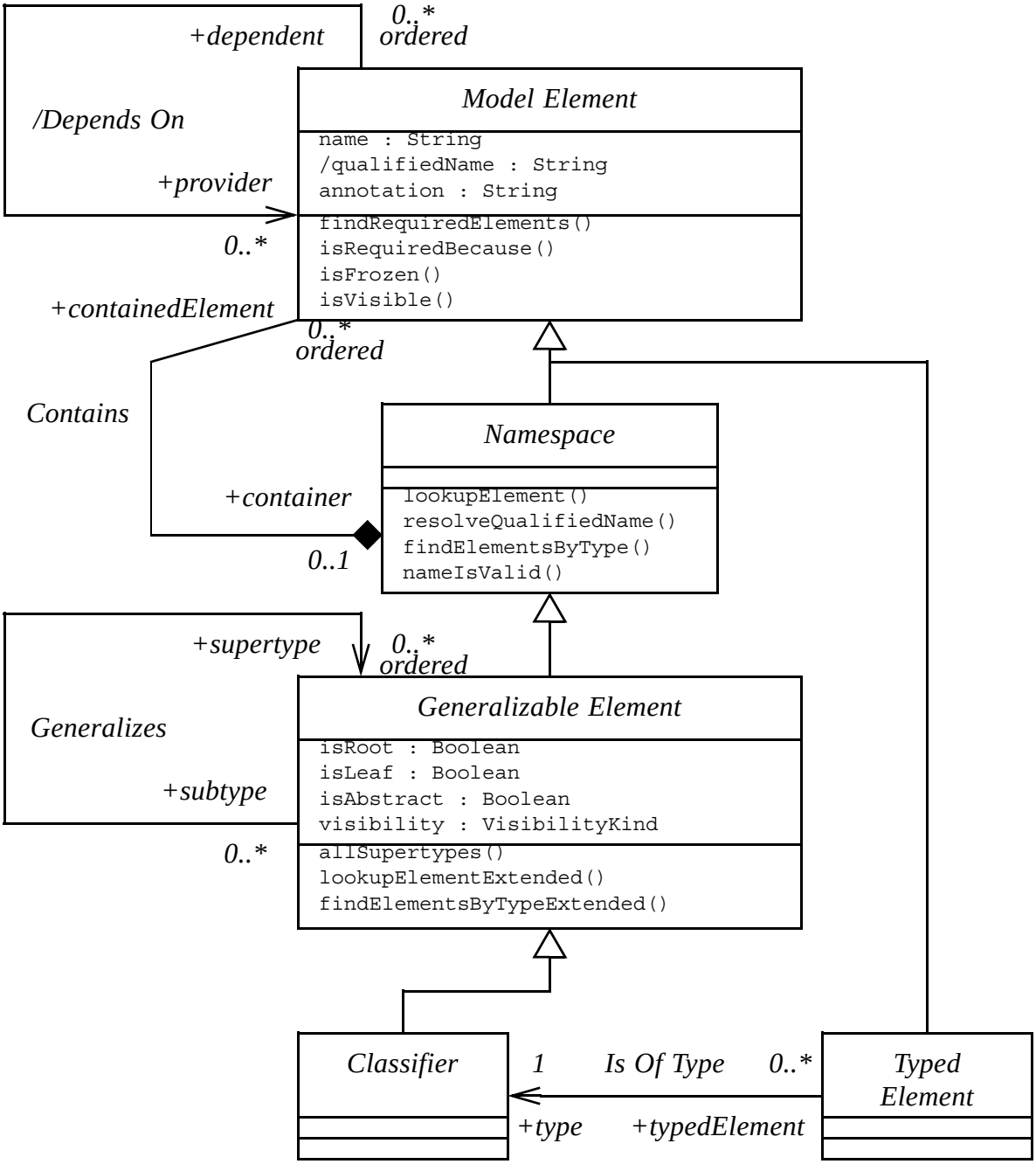

Figure 7.2 - The Key Abstractions of the MOF Model

© ISO/IEC 2005 - All rights reserved

7.3.4 The MOF Model Containment Hierarchy

The most important relationship in the MOF Model is the Contains Association. Containment is a utility Association that is used to relate (for example) Classes to their Operations and Attributes, Operations to their Parameters, and so on. While the class diagram shows that only ModelElement objects that are subtypes of Namespace can contain any other ModelElements, the MOF Model restricts the legal containments to eliminate various nonsensical and problematical cases.

Table 7.4 shows the legal ModelElement containments in matrix form. The rows are the non-abstract subtypes of Namespace (i.e., possible containers) and the columns are the non-abstract subtypes of ModelElements (i.e., possible contained elements). For each combination of container and contained, a “Y” says that containment is legal and an “N” says that it is not.

NOTE: The normative specification of the containments rules is in the OCL rules

Table 7.4 summarizes the OCL containment rules..

Table 7.4 - The ModelElement Containment Matrix

	Package	Class	PrimitiveType	StructureType	CollectionType	EnumerationType	AliasType	Association	Attribute	Reference	Operation	Exception	Parameter	AssociationEnd	Constraint	Constant	Import	StructurewField	Tag
Package	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	Y	N	N	Y	Y	Y	N	Y
Class	N	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	N	N	Y	Y	N	N	Y
Primitive Type	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	Y
Structure Type	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	Y	Y
Collection Type	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	Y
Enumeration Type	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	Y
Alias Type	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	Y
Structure Field	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	Y
Association	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	Y	N	N	N	Y
Operation	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	Y	N	N	N	Y
Exception	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	Y

NOTE: While the MOF Model allows Classes to contain Classes, the MOF to IDL mapping does not support this. Any metamodel in which Classes are nested inside Classes must be considered as not technology neutral.

© ISO/IEC 2005 - All rights reserved

7.4 MOF Model Classes

7.4.1 ModelElement (abstract)

ModelElement classifies the elementary atomic constructs of models. ModelElement is the root Class within the MOF Model.

SuperClasses

None. (While the CORBA IDL for ModelElement inherits from Reflective::RefObject, this is *not* generalization in the MOF Model sense. Rather it is an artifact of the IDL mapping.)

Attributes

name

Provides a meta-modeler supplied name that uniquely identifies the ModelElement in the context of the ModelElement's containing Namespace. When choosing a ModelElement's name, the meta-modeler should consider the rules for translating names into identifiers in the relevant mappings (e.g., 9.7.1, "Generated IDL Identifiers," on page 198). To minimize portability problems, use names that start with an ASCII letter, and consist of ASCII letters and digits, space and underscore. Avoid names where capitalization, spaces, or underscores are significant.	
type:	String
multiplicity:	exactly one

qualifiedName

Provides a unique name for the ModelElement within the context of its outermost containing Package. The qualifiedName is a list of String values consisting of the names of the ModelElement, its container, its container's container, and so on until a non-contained element is reached. The first member of the list is the name of the non-contained element.	
type:	String
multiplicity:	one or more; ordered
changeable:	no
derived from:	[S-12] on page 130

annotation

Provides an informal description of the ModelElement.	
type:	String
multiplicity:	exactly one

© ISO/IEC 2005 - All rights reserved

References

container

Identifies the Namespace that contains the ModelElement. Since the Contains Association is a Composite Association, any ModelElement can have at most one container, and the containment graph is strictly tree shaped.	
type:	Namespace
defined by:	Contains::container
multiplicity:	zero or one
inverse:	ModelElement::contents

requiredElements

Identifies the ModelElements on whose definition the definition of this ModelElement depends. For a definition of dependency, see 7.5.9, “DependsOn (derived),” on page 93.	
type:	ModelElement
defined by:	DependsOn::provider
multiplicity:	zero or more

constraints

Identifies the set of Constraints that apply to the ModelElement. A Constraint applies to all instances of the ModelElement and its sub-Classes.	
type:	Constraint
multiplicity:	zero or more
inverse:	Constraint::constrainedElements.
defined by:	Constrains::provider

Operations

findRequiredElements

This operation selects a subset of the ModelElements that this one depends on, based on their dependency categories. The “kinds” argument gives the kinds of dependency of interest to the caller.

String constants for the standard dependency categories are given in 7.8, “MOF Model Constants,” on page 99 and their meanings are defined in 7.5.9, “DependsOn (derived),” on page 93. In this context, the AllDep pseudo-category (i.e., “all”) is equivalent to passing all of the standard categories, and the IndirectDep pseudo-category (i.e., “indirect”) is ignored.

... continued

© ISO/IEC 2005 - All rights reserved

findRequiredElements

If the “recursive” argument is “false,” the operation returns the direct dependents only. If it is “true,” all dependents in the transitive closure of DependsOn for the specified “kinds” are returned.	
return type:	ModelElement (multiplicity: zero or more; unordered, unique)
isQuery:	yes
parameters:	kinds: in String (multiplicity: one or more; unordered; unique) recursive: in Boolean
operation semantics	[S-4] on page 125

isRequiredBecause

This operation performs two functions: • It checks whether this ModelElement directly or indirectly depends on the ModelElement given by “otherElement.” If it does, the operation’s result is “true;” otherwise, it is “false.” • If a dependency exists; that is, the result is “true,” the operation returns a String in “reason” that categorizes the dependency. String constants for the dependency kind categories are given in 7.8, “MOF Model Constants,” on page 99 and their meanings are defined in 7.5.9, “DependsOn (derived),” on page 93. If the dependency is indirect, IndirectDep is returned. If there are multiple dependencies, any category that applies may be returned in “reason.” If no dependencies exist, an empty string is returned in “reason.”	
return type:	Boolean
isQuery:	yes
parameters:	otherElement: in ModelElement reason: out String
operation semantics	[S-5] on page 126

isFrozen

Reports the freeze status of a ModelElement. A ModelElement, at any particular time, is either frozen or not frozen. All ModelElements of a published model are permanently frozen.	
return type:	Boolean
isQuery:	yes

isVisible

Returns true. This operation is reserved for future use when the MOF visibility rules have stabilized. Then it will determine whether the supplied otherElement is visible to this ModelElement.	
return type:	Boolean
isQuery:	yes
parameters:	otherElement: in ModelElement
operation semantics	[S-3] on page 125

© ISO/IEC 2005 - All rights reserved

Constraints

A ModelElement that is not a Package must have a container. [C-1]

The attribute values of a ModelElement that is frozen cannot be changed. [C-2] on page 105.

A frozen ModelElement that is in a frozen Namespace can only be deleted, by deleting the Namespace. [C-3] on page 106.

The link sets that express dependencies of a frozen Element on other Elements cannot be explicitly changed. [C-4] on page 106.

IDL

```
interface ModelElementClass : Reflective::RefObject {
      readonly attribute ModelElementSet all_of_type_model_element;
      const string MUST_BE_CONTAINED_UNLESS_PACKAGE =
            "org.omg.mof:constraint.model.model_element.must_be_contained_unless_package";
      const string FROZEN_ATTRIBUTES_CANNOT_BE_CHANGED =
            "org.omg.mof:constraint.model.model_element.frozen_attributes_cannot_be_changed";
      const string FROZEN_ELEMENTS_CANNOT_BE_DELETED =
            "org.omg.mof:constraint.model.model_element.frozen_elements_cannot_be_deleted";
      const string FROZEN_DEPENDENCIES_CANNOT_BE_CHANGED =
            "org.omg.mof:constraint.model.model_element.frozen_dependencies_cannot_be_changed";
      const DependencyKind CONTAINER_DEP = "container";
      const DependencyKind CONTENTS_DEP = "contents";
      const DependencyKind SIGNATURE_DEP = "signature";
      const DependencyKind CONSTRAINT_DEP = "constraint";
      const DependencyKind CONSTRAINED_ELEMENTS_DEP = "constrained elements";
      const DependencyKind SPECIALIZATION_DEP = "specialization";
      const DependencyKind IMPORT_DEP = "import";
      const DependencyKind TYPE_DEFINITION_DEP = "type definition";
      const DependencyKind REFERENCED_ENDS_DEP = "referenced ends";
      const DependencyKind TAGGED_ELEMENTS_DEP = "tagged elements";
      const DependencyKind INDIRECT_DEP = "indirect";
      const DependencyKind ALL_DEP = "all";
}; // end of interface ModelElementClass

interface ModelElement : ModelElementClass {
     wstring name ()
       raises (Reflective::MofError);
     void set_name (in wstring new_value)
       raises (Reflective::MofError);
     ::PrimitiveTypes::WStringList qualified_name ()
       raises (Reflective::MofError);
     wstring annotation ()
       raises (Reflective::MofError);
     void set_annotation (in wstring new_value)
       raises (Reflective::MofError);
     ModelElementSet required_elements ()
       raises (Reflective::MofError);
     ModelElementSet find_required_elements (in ::PrimitiveTypes::WStringSet kinds, in boolean recursive)
       raises (Reflective::MofError);
      boolean is_required_because (in ModelElement other_element, out wstring reason)
       raises (Reflective::MofError);
      Namespace container ()
```


© ISO/IEC 2005 - All rights reserved

```
    raises (Reflective::NotSet, Reflective::MofError);
  void set_container (in Namespace new_value)
    raises (Reflective::MofError);
  void unset_container ()
    raises (Reflective::MofError);
  ConstraintSet constraints ()
    raises (Reflective::MofError);
   void set_constraints (in ConstraintSet new_value)
    raises (Reflective::MofError);
  void add_constraints (in Constraint new_element)
    raises (Reflective::MofError);
  void modify_constraints (in Constraint old_element, in Constraint new_element)
    raises (Reflective::MofError);
   void remove_constraints (in Constraint old_element)
    raises (Reflective::NotFound, Reflective::MofError);
   boolean is_frozen ()
    raises (Reflective::MofError);
   boolean is_visible (in ModelElement other_element)
    raises (Reflective::MofError);
};
```

7.4.2 Namespace (abstract)

The Namespace Class classifies and characterizes ModelElements that can contain other ModelElements. Along with containing the ModelElements, a Namespace defines a namespace, the allowable set of names, and the naming constraints for these elements.

Subclasses of the Namespace Class have mechanisms for effectively extending their namespace, without actually containing additional ModelElements. Thus Namespace can be viewed in terms of its two roles, as a container and as a namespace mechanism. Because only subclasses extend the namespace, the namespace and contents are coincident in the definition of the Namespace Class. Each Namespace has four collections (the latter three derivable) that are used in the MOF Model's Constraints. These collections are:

- The contents (also called the direct contents), which are defined by the contents reference.
- All contents, the transitive closure on the contents reference.
- The extended namespace (the contents plus elements included by extension), which Namespace subclasses accomplish through generalization and importation.
- The extended contents (the transitive closure on the contents reference applied to the extended namespace).

The definitions of these collections may be found in 7.9.6, "OCL Helper functions," on page 131.

SuperClasses

ModelElement

References

© ISO/IEC 2005 - All rights reserved

contents

Identifies the set of ModelElements that a Namespace contains.	
class:	ModelElement
defined by:	Contains::containedElement
multiplicity:	zero or more; ordered
inverse:	ModelElement::container

Operations

lookupElement

Searches for an element contained by this Namespace whose name is precisely equal (as a wide string) to the supplied name. The operation either returns a ModelElement that satisfies the above, or raises the NameNotFound exception.	
return type:	ModelElement
isQuery:	yes
parameters:	name : in String
exceptions:	NameNotFound
operation semantics:	[S-6] on page 127

resolveQualifiedName

Searches for a ModelElement contained within this Namespace that is identified by the supplied qualifiedName. The qualifiedName is interpreted as a “path” starting from this Namespace.	
return type:	ModelElement (exactly one). If no element is found, an exception is raised.
isQuery:	yes
parameters:	qualifiedName : in String (multiplicity one or more; ordered; not unique)
exceptions:	NameNotResolved
operation semantics:	[S-7] on page 128

© ISO/IEC 2005 - All rights reserved

findElementsByType

Returns a list of the ModelElements contained by this Namespace that match the Class supplied. If 'includeSubtypes' is false, this operation returns only those elements whose most-derived Class is 'ofType.' If 'includeSubtypes' is true, the operation also returns instances of subtypes of 'ofType.' The order of the elements in the returned list is the same as their order in the Namespace. For example, "findElementsByType(ModelElement, false)" always returns an empty list, since ModelElement is an abstract Class. On the other hand, "findElementsByType(ModelElement, true)" always returns the contents of the Namespace, since all their Classes are subtypes of ModelElement.	
return type:	ModelElement (multiplicity zero or more; ordered; unique)
isQuery:	yes
parameters:	ofType : in Class includeSubtypes : in Boolean
operation semantics:	[S-9] on page 129.

nameIsValid

Determines whether the proposedName can be used as the name for a new member ModelElement in this Namespace. Specifically, it checks that the Namespace uniqueness rules would still be satisfied after adding such a name.	
return type:	Boolean
isQuery:	yes
parameters:	proposedName : in String
operation semantics:	[S-8] on page 128.

Constraints

The names of the contents of a Namespace must not collide. [C-5] on page 107.

IDL

```
interface NamespaceClass : ModelElementClass {
    readonly attribute NamespaceSet all_of_type_namespace;
    const string CONTENT_NAMES_MUST_NOT_COLLIDE =
      "org.omg.mof:constraint.model.namespace.content_names_must_not_collide";
    exception NameNotFound {
       wstring name;
    };
    exception NameNotResolved {
       wstring explanation;
       ::PrimitiveTypes::WStringList rest_of_name;
    };
}; // end of interface NamespaceClass

  interface Namespace : NamespaceClass, ModelElement {
     ModelElementUList contents ()
       raises (Reflective::MofError);
```

© ISO/IEC 2005 - All rights reserved

```
    void set_contents (in ModelElementUList new_value)
      raises (Reflective::MofError);
    void add_contents (in ModelElement new_element)
      raises (Reflective::MofError);
    void add_contents_before (in ModelElement new_element, in ModelElement before_element)
     raises (Reflective::NotFound, Reflective::MofError);
    void modify_contents (in ModelElement old_element, in ModelElement new_element)
     raises (Reflective::NotFound, Reflective::MofError);
    void remove_contents (in ModelElement old_element)
      raises (Reflective::NotFound, Reflective::MofError);

    ModelElement lookup_element (in wstring name)
      raises (NamespaceClass::NameNotFound, Reflective::MofError);
    ModelElement resolve_qualified_name (in ::PrimitiveTypes::WStringList qualified_name)
      raises (NamespaceClass::NameNotResolved, Reflective::MofError);
    ModelElementUList find_elements_by_type (in Class of_type, in boolean include_subtypes)
      raises (Reflective::MofError);
    boolean name_is_valid (in wstring proposed_name)
      raises (Reflective::MofError);
  };
```

7.4.3 GeneralizableElement (abstract)

The GeneralizableElement Class classifies and characterizes ModelElements that can be generalized through supertyping and specialized through subtyping. A GeneralizableElement inherits the features of each of its supertypes, the features of the supertypes of the immediate supertypes, and so on. In other words, all the features of the transitive closure of all the supertypes of the GeneralizableElement.

When a GeneralizableElement inherits a feature, that feature name effectively becomes part of the namespace for the GeneralizableElement and the feature is considered part of the extended namespace of the Namespace. Therefore, a GeneralizableElement cannot have a superclass if it causes an inherited feature to have a namespace collision with its own features - see Constraint [C-8] on page 108.

To the degree that a GeneralizableElement is defined by its features, the superclass / subclass association defines substitutability. Any instance of a GeneralizableElement can be supplied wherever an instance of a superclass of that GeneralizableElement is expected.

SuperClasses

Namespace

Attributes

isRoot

Specifies whether the GeneralizableElement may have supertypes. True indicates that it may not have supertypes, false indicates that it may have supertypes (whether or not it actually has any).	
type:	Boolean
multiplicity:	exactly one

© ISO/IEC 2005 - All rights reserved

isLeaf

Specifies whether the GeneralizableElement may be a supertype of another Generalizable Element. True indicates that it may not be a supertype, false indicates that it may be a supertype (whether or not it actually is).	
type:	Boolean
multiplicity:	exactly one

isAbstract

Indicates whether the GeneralizableElement is expected to have instances. When isAbstract is true, any instance that is represented or classified by this GeneralizableElement is additionally an instance of some specialization of this GeneralizableElement. No operation that supports creation of instances of this GeneralizableElement should be available.	
type:	Boolean
multiplicity:	exactly one

visibility

In the future, this Attribute will be used to limit the ability of ModelElements outside of this GeneralizableElement's container to depend on it; see 7.6.3, "VisibilityKind," on page 97. The rules of visibility of MOF ModelElements are not currently specified.	
type:	VisibilityKind
multiplicity:	exactly one

References

supertypes

Identifies the set of superclasses for a GeneralizableElement. Note that a GeneralizableElement does not have a reference to its subclasses.	
class:	GeneralizableElement
defined by:	Generalizes::supertype
multiplicity:	zero or more; ordered

Operations

© ISO/IEC 2005 - All rights reserved

allSupertypes

Returns a list of direct and indirect supertypes of this GeneralizableElement. A direct supertype is a GeneralizableElement that directly generalizes this one. An indirect supertype is defined (recursively) as a supertype of some other direct or indirect supertype of the GeneralizableElement. The order of the list elements is determined by a depth-first traversal of the supertypes with duplicate elements removed.	
return type:	GeneralizableElement (multiplicity zero or more, ordered, unique)
isQuery:	yes
parameters:	none
operation semantics:	[S-1] on page 125.

lookupElementExtended

Returns an element whose name matches the supplied “name.” Like the “lookupElement” operation on Namespace, this operation searches the contents of the GeneralizableElement. In addition, it tries to match the name in the contents of all direct and indirect supertypes of the GeneralizableElement. For Packages, a subclass of GeneralizableElement, the operation can also match a Namespace associated with an Import object. NameNotFound is raised if no element matches the name.	
return type:	ModelElement (multiplicity exactly one)
isQuery:	yes
parameters:	name : in wstring
exceptions	NameNotFound
operation semantics:	[S-10] on page 129

findElementsByTypeExtended

Provides an extension of the findElementsByType defined for Namespace so that contained elements of all superclasses (direct and indirect) of the GeneralizableElement are included in the search. The order of the returned elements is determined by the order of the elements contained in the GeneralizableElements and a depth-first traversal of the superclasses. Subclasses can include a larger overall area for the lookup. Package, a subclass of GeneralizableElement, also considers the elements brought into this Namespace through the use of Import.	
return type:	ModelElement (multiplicity zero or more; ordered; unique)
isQuery:	yes
parameters:	ofType : in Class includeSubtypes : in Boolean
operation semantics:	[S-11] on page 130.

© ISO/IEC 2005 - All rights reserved

Constraints

A Generalizable Element cannot be its own direct or indirect supertype. [C-6] on page 107.

A supertypes of a GeneralizableElement must be of the same kind as the GeneralizableElement itself. [C-7] on page 107.

The names of the contents of a GeneralizableElement should not collide with the names of the contents of any direct or indirect supertype. [C-8] on page 108.

Multiple inheritance must obey the “Diamond Rule.” [C-9] on page 108.

If a Generalizable Element is marked as a “root,” it cannot have any supertypes. [C-10] on page 108.

A GeneralizableElement’s immediate supertypes must all be visible to it. [C-11] on page 109.

A GeneralizableElement cannot inherit from a GeneralizableElement defined as a “leaf.” [C-12] on page 109.

IDL

```
interface GeneralizableElementClass : NamespaceClass {
  readonly attribute GeneralizableElementUList
  all_of_type_generalizable_element;
  const string SUPERTYPE_MUST_NOT_BE_SELF =
     "org.omg.mof:constraint.model.generalizable_element.supertype_must_not_be_self";
  const string SUPERTYPE_KIND_MUST_BE_SAME =
     "org.omg.mof:constraint.model.generalizable_element.supertype_kind_must_be_same";
  const string CONTENTS_MUST_NOT_COLLIDE_WITH_SUPERTYPES
     "org.omg.mof:constraint.model.generalizable_element”
     “.contents_must_not_collide_with_supertypes";
  const string DIAMOND_RULE_MUST_BE_OBEYED =
     "org.omg.mof:constraint.model.generalizable_element.diamond_rule_must_be_obeyed";
  const string NO_SUPERTYPES_ALLOWED_FOR_ROOT =
     "org.omg.mof:constraint.model.generalizable_element.no_supertypes_allowed_for_root";
  const string SUPERTYPES_MUST_BE_VISIBLE =
     "org.omg.mof:constraint.model.generalizable_element.supertypes_must_be_visible";
  const string NO_SUBTYPES_ALLOWED_FOR_LEAF =
     "org.omg.mof:constraint.model.generalizable_element.no_subtypes_allowed_for_leaf";

}; // end of interface GeneralizableElementClass

interface GeneralizableElement : GeneralizableElementClass, Namespace {
    boolean is_root ()
       raises (Reflective::MofError);
    void set_is_root (in boolean new_value)
      raises (Reflective::MofError);
    boolean is_leaf ()
      raises (Reflective::MofError);
    void set_is_leaf (in boolean new_value)
      raises (Reflective::MofError);
     boolean is_abstract ()
      raises (Reflective::MofError);
    void set_is_abstract (in boolean new_value)
      raises (Reflective::MofError);
    VisibilityKind visibility ()
      raises (Reflective::MofError);
    void set_visibility (in VisibilityKind new_value)
```

© ISO/IEC 2005 - All rights reserved

```
      raises (Reflective::MofError);
    GeneralizableElementUList supertypes ()
      raises (Reflective::MofError);
     void set_supertypes (in GeneralizableElementUList new_value)
      raises (Reflective::MofError);
    void add_supertypes (in GeneralizableElement new_element)
      raises (Reflective::MofError);
    void add_supertypes_before (in GeneralizableElement new_element,
                        in GeneralizableElement before_element)
      raises (Reflective::NotFound, Reflective::MofError);
     void modify_supertypes (in GeneralizableElement old_element,
                        in GeneralizableElement new_element)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove_supertypes (in GeneralizableElement old_element)
      raises (Reflective::NotFound, Reflective::MofError);
    GeneralizableElementSet all_supertypes ()
      raises (Reflective::MofError);
     ModelElement lookup_element_extended (in wstring name)
      raises (NameNotFound, Reflective::MofError);
     ModelElementUList find_elements_by_type_extended (in Class of_type, in boolean include_subtypes)
      raises (Reflective::MofError);
};
```

7.4.4 TypedElement (abstract)

The TypedElement type is an abstraction of ModelElements that require a type as part of their definition. A TypedElement does not itself define a type, but is associated with a Classifier. The relationship between TypedElements and Classifiers is shown in Figure 7.3 on page 53.

SuperClasses

ModelElement

References

type

Provides the representation of the type supporting the TypedElement through this reference.	
class:	Classifier
defined by:	IsOfType::type
multiplicity:	exactly one

Constraints

An Association cannot be the type of a TypedElement. [C-13] on page 109.

A TypedElement can only have a type that is visible to it. [C-14] on page 110.

© ISO/IEC 2005 - All rights reserved

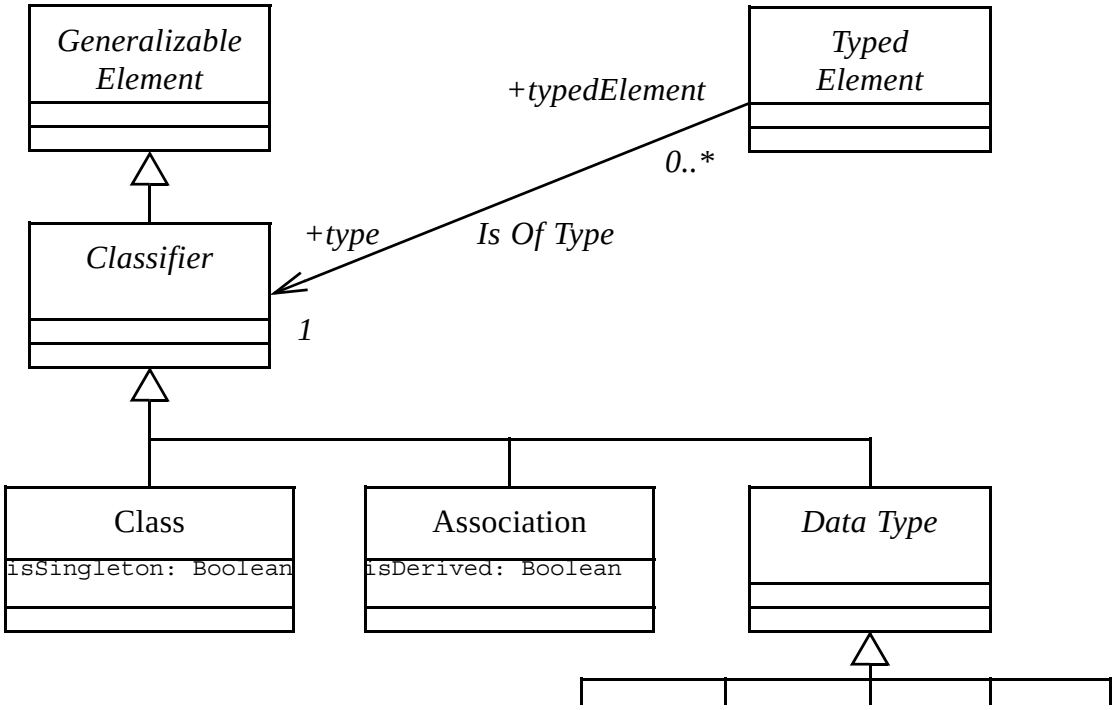

Figure 7.3 - MOF Model Classifiers

IDL

```
interface TypedElementClass : ModelElementClass {
    // get all typed_element including subtypes of typed_element
    readonly attribute TypedElementSet all_of_type_typed_element;
    const string ASSOCIATIONS_CANNOT_BE_TYPES =
     "org.omg.mof:constraint.model.typed_element.associations_cannot_be_types";
    const string TYPE_MUST_BE_VISIBLE =
     "org.omg.mof:constraint.model.typed_element.type_must_be_visible";
  }; // end of interface TypedElementClass

  interface TypedElement : TypedElementClass, ModelElement {
    Classifier type ()
      raises (Reflective::MofError);
    void set_type (in Classifier new_value)
      raises (Reflective::MofError);
  };
```

7.4.5 Classifier (abstract)

A classifier provides a classification of instances through a set of Features it contains.

© ISO/IEC 2005 - All rights reserved

SuperClasses

GeneralizableElement

IDL

```
interface ClassifierClass : GeneralizableElementClass {
    readonly attribute ClassifierSet all_of_type_classifier;
}; // end of interface ClassifierClass

interface Classifier : ClassifierClass, GeneralizableElement { };
```

7.4.6 Class

A Class defines a classification over a set of object instances by defining the state and behavior they exhibit. This is represented through operations, attributes, references, participation in associations, constants, and constraints. Similar concepts are used in other environments for representing Classes and their implementations. However, in the MOF the class characteristics are modeled in an implementation-independent manner. For instance, an attribute of a Class is specified independently of any code to store and manage the attributes value. The implementation simply must insure that its behavior conforms to behavior specified by the chosen technology mapping. The MOF Class construct is more than just an interface specification.

SuperClasses

Classifier

Contained Elements

Class, DataType subtypes, Attribute, Reference, Operation, Exception, Constraint, Constant, Tag; see constraint [C-15] on page 110.

Attributes

isSingleton

When isSingleton is true, at most one M1 level instance of this Class may exist within the M1-level extent of the Class.	
type:	Boolean
multiplicity:	exactly one

Constraints

A Class may contain only Classes, DataTypes, Attributes, References, Operations, Exceptions, Constants, Constraints, and Tags. [C-15] on page 110.

A Class that is marked as abstract cannot also be marked as singleton. [C-16] on page 110.

IDL

```
interface ClassClass : ClassifierClass {
  readonly attribute ClassSet all_of_type_class;
  readonly attribute ClassSet all_of_class_class;
  const string CLASS_CONTAINMENT_RULES =
```


© ISO/IEC 2005 - All rights reserved

```
    "org.omg.mof:constraint.model.class.class_containment_rules";
  const string ABSTRACT_CLASSES_CANNOT_BE_SINGLETON =
    "org.omg.mof:constraint.model.class.abstract_classes_cannot_be_singleton";

  Class create_class (
    /* from ModelElement */              in wstring name,
    /* from ModelElement */              in wstring annotation,
    /* from GeneralizableElement */      in boolean is_root,
    /* from GeneralizableElement */      in boolean is_leaf,
    /* from GeneralizableElement */      in boolean is_abstract,
    /* from GeneralizableElement */      in ::Model::VisibilityKind visibility,
    /* from Class */                     in boolean is_singleton)
        raises (Reflective::MofError);
}; // end of interface ClassClass

  interface Class : ClassClass, Classifier {
    boolean is_singleton ()
      raises (Reflective::MofError);
    void set_is_singleton (in boolean new_value)
      raises (Reflective::MofError);
 };
```

7.4.7 DataType (abstract)

DataType is the superclass of the classes that represent MOF data types and data type constructors as described in 8.2, “MOF Values,” on page 139. The DataType class, its subclasses and related classes are depicted in Figure 7.4.

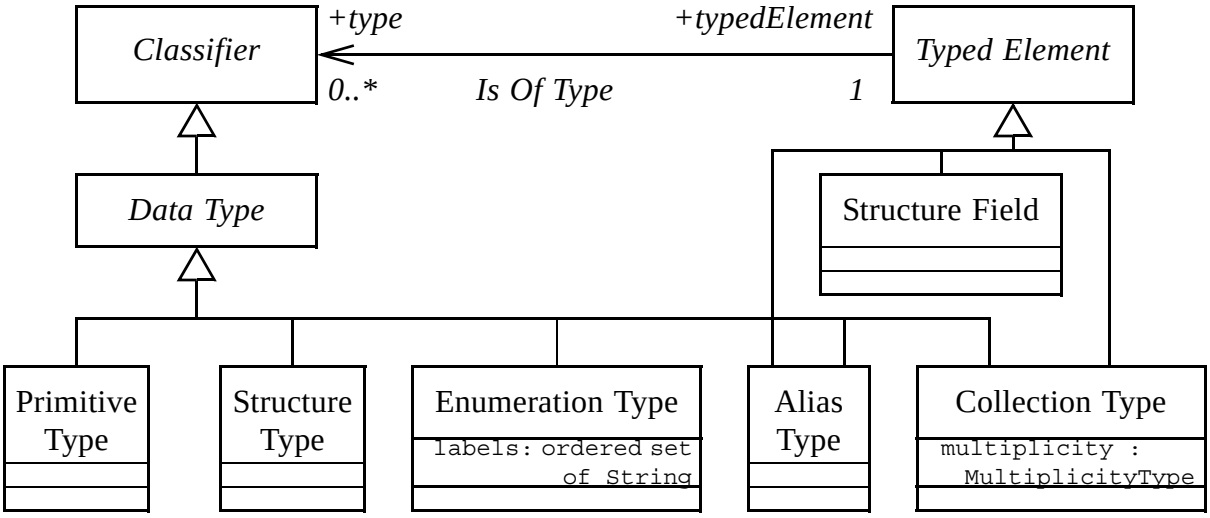

Figure 7.4 - MOF Data Type Elements

SuperClasses

Classifier

© ISO/IEC 2005 - All rights reserved

Contained Elements

StructureField (for a StructureType only), Constraint, Tag. See DataTypeContainmentRules [C-17] on page 111.

Attributes

none

Constraints

Inheritance / generalization is not applicable to DataTypes. [C-19] on page 111.

A DataType cannot be abstract. [C-20] on page 111.

IDL

```
interface DataTypeClass : ClassifierClass {
    readonly attribute DataTypeSet all_of_type_data_type;
    const string DATA_TYPE_CONTAINMENT_RULES =
        "org.omg.mof:constraint.model.data_type.data_type_containment_rules";
    const string DATA_TYPES_HAVE_NO_SUPERTYPES =
        "org.omg.mof:constraint.model.data_type.data_types_have_no_supertypes";
    const string DATA_TYPES_CANNOT_BE_ABSTRACT =
        "org.omg.mof:constraint.model.data_type.data_types_cannot_be_abstract";

}; // end of interface DataTypeClass

  interface DataType : DataTypeClass, Classifier {
};
```

7.4.8 PrimitiveType

Instances of the PrimitiveType class are used to represent primitive data types in a meta-model. The MOF has a small number of built-in primitive data types that may be freely used in any meta-model. These types are defined as instances of PrimitiveType that are contained by the standard “PrimitiveTypes” package. Refer to 7.10, “The PrimitiveTypes Package,” on page 134 for details of the PrimitiveTypes package, and to 8.2, “MOF Values,” on page 139 for more details on data type semantics.

The MOF built-in primitive data types map to different concrete data types in the context of each technology mapping. Each technology mapping is expected to support all of the standard built-in primitive data types.

NOTE: A meta-model may contain PrimitiveType instances other than those defined in the “PrimitiveTypes” package. These instances denote technology specific, vendor specific or user defined primitive data types. They should not be used in technology neutral meta-models.

SuperClasses

DataType

Contained Elements

Constraint, Tag; see DataTypeContainmentRules [C-17] on page 111.

© ISO/IEC 2005 - All rights reserved

IDL

```
  interface PrimitiveTypeClass : DataTypeClass {
    readonly attribute PrimitiveTypeSet all_of_type_primitive_type;
    readonly attribute PrimitiveTypeSet all_of_class_primitive_type;
    DataType create_primitive_type (
          /* from ModelElement */              in wstring name,
          /* from ModelElement */              in wstring annotation,
          /* from GeneralizableElement */      in boolean is_root,
          /* from GeneralizableElement */      in boolean is_leaf,
          /* from GeneralizableElement */      in boolean is_abstract,
          /* from GeneralizableElement */      in ::Model::VisibilityKind visibility)
      raises (Reflective::MofError);
}; // end of interface PrimitiveTypeClass

  interface PrimitiveType : PrimitiveTypeClass, DataType {
};
```

7.4.9 CollectionType

The CollectionType class is a type constructor for MOF collection types. A collection type is a data type whose values are finite collections of instances of some base type. The base type for a collection data type is given by the CollectionType instance's 'type' value. The 'multiplicity' Attribute gives the collection type's lower and upper bounds, and its orderedness and uniqueness properties.

SuperClasses

DataType, TypedElement

Contained Elements

Constraint, Tag; see DataTypeContainmentRules [C-17] on page 111.

Attributes

multiplicity

The multiplicity attribute of a CollectionType specifies upper and lower bounds on the size of a collection, and gives the 'isOrdered' and 'isUnique' flags that subclassify collections into 'bags,' 'sets,' 'lists,' and 'ordered sets.'	
type:	MultiplicityType
multiplicity:	exactly one

IDL

```
  interface CollectionTypeClass : DataTypeClass, TypedElementClass {
    readonly attribute CollectionTypeSet all_of_type_collection_type;
    readonly attribute CollectionTypeSet all_of_class_collection_type;
    DataType create_collection_type (
          /* from ModelElement */              in wstring name,
          /* from ModelElement */              in wstring annotation,
          /* from GeneralizableElement */      in boolean is_root,
          /* from GeneralizableElement */      in boolean is_leaf,
```

© ISO/IEC 2005 - All rights reserved

```
            /* from GeneralizableElement */    in boolean is_abstract,
            /* from GeneralizableElement */    in ::Model::VisibilityKind visibility,
            /* from CollectionType */           in ::Model::MultiplicityType multiplicity)
        raises (Reflective::MofError);
}; // end of interface CollectionTypeClass
  interface CollectionType : CollectionTypeClass, DataType, TypedElement {
    ::Model::MultiplicityType multiplicity()
        raises (Reflective::MofError);
    void set_multiplicity(in ::Model::MultiplicityType multiplicity)
        raises (Reflective::MofError);
}; // end of interface CollectionType
```

7.4.10 EnumerationType

The EnumerationType class is a type constructor for MOF enumeration types. An enumeration type is a data type whose values are the elements of a finite set of enumerators. The enumeration type is specified by defining an ordered set of enumerator labels.

SuperClasses

DataType

Contained Elements

Constraint, Tag; see DataTypeContainmentRules [C-17] on page 111.

Attributes

labels

The labels attribute of an EnumerationType gives the names of the enumerators for the type. The label elements must be unique within the collection, and their order in the collection is significant.	
type:	String
multiplicity:	one or more, ordered, unique

IDL

```
interface EnumerationTypeClass : DataTypeClass {
    readonly attribute EnumerationTypeSet all_of_type_enumeration_type;
    readonly attribute EnumerationTypeSet all_of_class_enumeration_type;
    DataType create_enumeration_type (
            /* from ModelElement */            in wstring name,
            /* from ModelElement */            in wstring annotation,
            /* from GeneralizableElement */    in boolean is_root,
            /* from GeneralizableElement */    in boolean is_leaf,
            /* from GeneralizableElement */    in boolean is_abstract,
            /* from GeneralizableElement */    in ::Model::VisibilityKind visibility,
            /* from EnumerationType */ in ::PrimitiveTypes::WStringUList labels)
        raises (Reflective::MofError);
}; // end of interface EnumerationTypeClass

  interface EnumerationType : EnumerationTypeClass, DataType {
```


© ISO/IEC 2005 - All rights reserved

```
    ::PrimitiveTypes::WStringUList labels()
        raises (Reflective::MofError);
    void set_labels(in ::PrimitiveTypes::WStringUList labels)
        raises (Reflective::MofError);
}; // end of interface EnumerationType
```

7.4.11 AliasType

The AliasType class is a type constructor for MOF alias types. An alias type is a subtype of some other MOF class or data type, given by the 'type' value of the AliasType instance; i.e., a subset of the values of the type given by its 'type.' This subset is typically specified by attaching a Constraint to the AliasType instance. An alias type may convey a different "meaning" to that of its base type.

SuperClasses

DataType, TypedElement

Contained Elements

Constraint, Tag; see DataTypeContainmentRules [C-17] on page 111.

IDL

```
  interface AliasTypeClass : DataTypeClass, TypedElementClass {
    readonly attribute AliasTypeSet all_of_type_alias_type;
    readonly attribute AliasTypeSet all_of_class_alias_type;
    DataType create_alias_type (
        /* from ModelElement */              in wstring name,
        /* from ModelElement */              in wstring annotation,
        /* from GeneralizableElement */      in boolean is_root,
        /* from GeneralizableElement */      in boolean is_leaf,
        /* from GeneralizableElement */      in boolean is_abstract,
        /* from GeneralizableElement */      in ::Model::VisibilityKind visibility)
      raises (Reflective::MofError);

}; // end of interface AliasTypeClass

  interface AliasType : AliasTypeClass, DataType, TypedElement {
};
```

7.4.12 StructureType

The StructureType class is a type constructor for MOF structure data types. A structure type is a tuple type (i.e., a cartesian product) consisting of one or more fields. The fields are defined by StructureField instances contained by the StructureType instance.

SuperClasses

DataType

Contained Elements

StructureField, Constraint, Tag; see DataTypeContainmentRules [C-17] on page 111.

© ISO/IEC 2005 - All rights reserved

Constraints

A StructureType must contain at least one StructureField. [C-59] on page 124.

IDL

```
  interface StructureTypeClass : DataTypeClass {
     readonly attribute StructureTypeSet all_of_type_structure_type;
     readonly attribute StructureTypeSet all_of_class_structure_type;
     const string MUST_HAVE_FIELDS =
          "org.omg.mof:constraint.model.structure_type.must_have_fields";
     DataType create_structure_type (
          /* from ModelElement */              in wstring name,
          /* from ModelElement */              in wstring annotation,
          /* from GeneralizableElement */      in boolean is_root,
          /* from GeneralizableElement */      in boolean is_leaf,
          /* from GeneralizableElement */      in boolean is_abstract,
          /* from GeneralizableElement */      in ::Model::VisibilityKind visibility)
       raises (Reflective::MofError);
}; // end of interface StructureTypeClass

interface StructureType : StructureTypeClass, DataType { };
```

7.4.13 StructureField

The StructureField class is used to specify the fields of a StructureType instance.

SuperClasses

TypedElement

Contained Elements

Constraint, Tag; see StructureFieldContainmentRules [C-58] on page 124.

IDL

```
interface StructureFieldClass : TypedElementClass {
     readonly attribute StructureFieldSet all_of_type_structure_field;
     readonly attribute StructureFieldSet all_of_class_structure_field;
     const string STRUCTURE_FIELD_CONTAINMENT_RULES =
          "org.omg.mof:constraint.modelstructure_field.structure_field_containment_rule";
     StructureField create_structure_field (
               /* from ModelElement */   in wstring name,
               /* from ModelElement */   in wstring annotation)
          raises (Reflective::MofError);
}; // end of interface StructureFieldClass

interface StructureField : StructureFieldClass, TypedElement { );
```

7.4.14 Feature (abstract)

A Feature defines a characteristic of the ModelElement that contains it. Specifically, Classifiers are defined largely by a composition of Features. The Feature Class and its sub-Classes are illustrated in Figure 7.5.

© ISO/IEC 2005 - All rights reserved

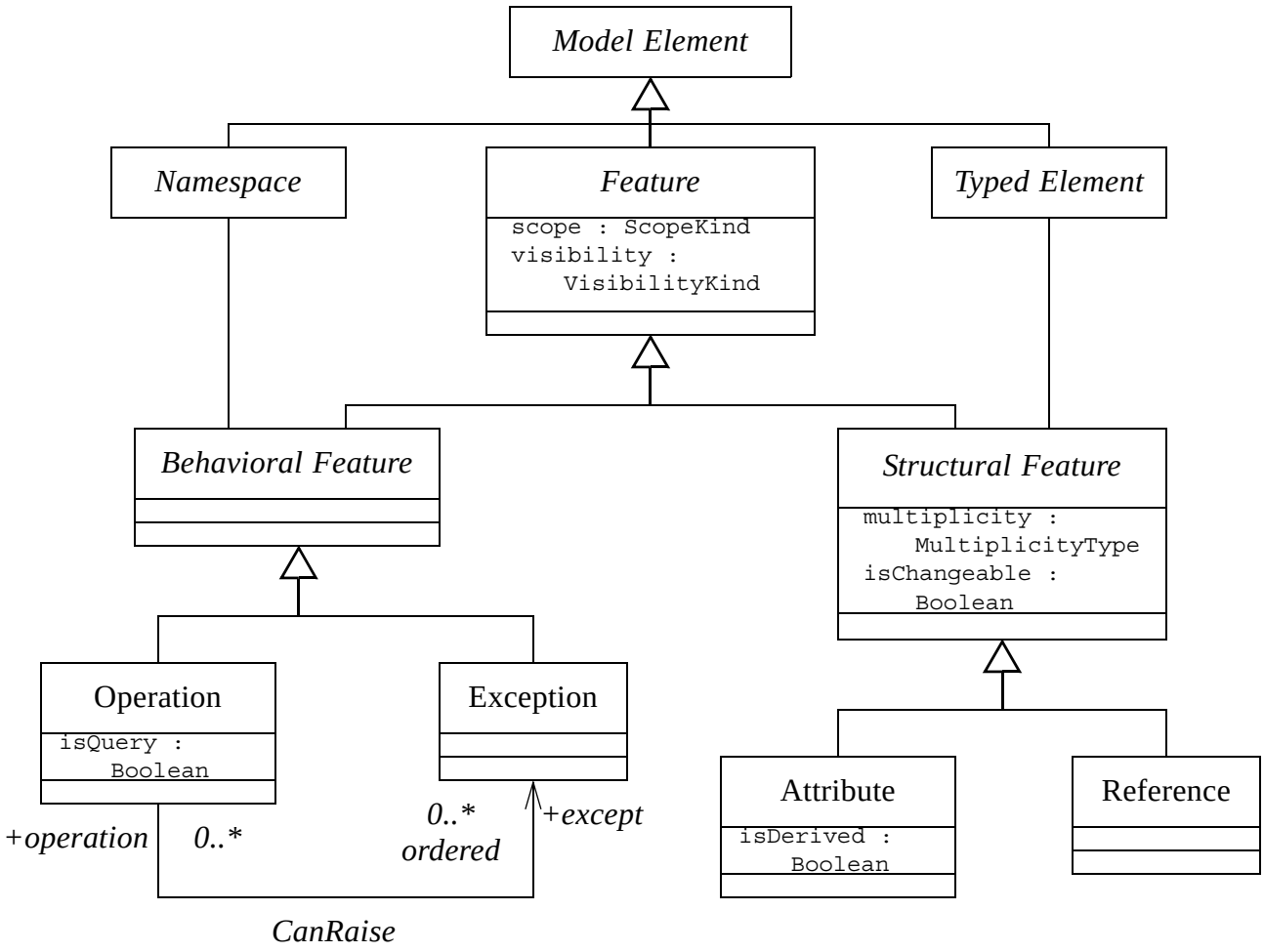

Figure 7.5 - Feature Classes of the MOF Model

SuperClasses

ModelElement

Attributes

© ISO/IEC 2005 - All rights reserved

scope

The scope defines whether a Feature supports the definition of instances of the Classifier owning the Feature or of the Classifier as a whole. When scope is instanceLevel, the Feature is accessed through instances of the Feature's owning Classifier; when scope is classifier, the Feature is accessed through the Classifier itself (or through its instances). For StructuralFeatures, a scope of instanceLevel indicates that a value represented by the StructuralFeature is associated with each instance of the Classifier; a scope of classifierLevel indicates that the StructuralFeature value is shared by the Classifier and all its instances.	
type:	ScopeKind
multiplicity:	exactly one

visibility

In the future, this Attribute will be used to limit the ability of ModelElements outside of this Feature's container to make use of it; see 7.6.3, "VisibilityKind," on page 97. The rules of visibility of MOF ModelElements are not currently specified.	
type:	VisibilityKind
multiplicity:	exactly one

IDL

```
interface FeatureClass : ModelElementClass {
    readonly attribute FeatureSet all_of_type_feature;
}; // end of interface FeatureClass

interface Feature : FeatureClass, ModelElement {
    ScopeKind scope ()
       raises (Reflective::MofError);
    void set_scope (in ScopeKind new_value)
      raises (Reflective::MofError);
    VisibilityKind visibility ()
       raises (Reflective::MofError);
    void set_visibility (in VisibilityKind new_value)
      raises (Reflective::MofError);
};
```

7.4.15 StructuralFeature (abstract)

A StructuralFeature defines a static characteristic of the ModelElement that contains it. The attributes and references of a Class define structural properties, which provide for the representation of the state of its instances.

SuperClasses

Feature, TypedElement

Attributes

© ISO/IEC 2005 - All rights reserved

multiplicity

Multiplicity defines constraints on the collection of instances or values that a StructuralFeature can hold. Multiplicity defines a lower and upper bound to the cardinality of the collection, although the upper bound can be specified as Unbounded. Additionally multiplicity defines two other characteristics of the collection: 1) a constraint on collection member ordering, and 2) a constraint on collection member uniqueness. Specifically, Multiplicity contains an isOrdered field. When isOrdered is true, then the ordering of the elements in the set are preserved. Typically, a mechanism is provided for adding elements to the collection positionally. Multiplicity also has an isUnique field. When isUnique is true, then the collection is constrained to hold no more than one of any value or instance.	
type:	MultiplicityType
multiplicity:	exactly one

isChangeable

The isChangeable attribute places restrictions on the use of certain operations, which could change the set of values or instances of the StructuralFeature, and on the operations that will get generated in IDL or other language generation. For any elaboration, no means are automatically created that provides a means of altering the attribute value. When IDL is generated, for instance, the operations, that are normally generated for changing the StructuralFeature will not be generated. However, isChangeable does not actually constrain the StructuralFeature to make it immutable. Any operations explicitly defined in a model may change the StructuralFeature values or instances (assuming the operation would have otherwise been able to do so).	
type:	Boolean
multiplicity:	exactly one

IDL

```
interface StructuralFeatureClass : FeatureClass, TypedElementClass {
     readonly attribute StructuralFeatureSet all_of_type_structural_feature;
  }; // end of interface StructuralFeatureClass

  interface StructuralFeature : StructuralFeatureClass, Feature, TypedElement {
    MultiplicityType multiplicity ()
       raises (Reflective::MofError);
    void set multiplicity (in MultiplicityType new_value)
       raises (Reflective::MofError),
    boolean is_changeable ()
       raises (Reflective::MofError);
    void set_is_changeable (in boolean new_value)
      raises (Reflective::MofError);
  };
```

7.4.16 Attribute (idl_substitute_name “MofAttribute”)

An Attribute (referred to as a MofAttribute in the mapped IDL) defines a StructuralFeature that contains values for Classifiers or their instances.

© ISO/IEC 2005 - All rights reserved

SuperClasses

StructuralFeature

Contained Elements

None (not a Namespace)

Attributes

isDerived

A derived attribute is one whose values are not part of the state of the object instance, but whose values can be determined or computed. In a sense, all attributes are derived, since it is up to the class's implementation to hold or calculate the values. However, by convention, isDerived indicates that the derived state is based on other information in the model. Modification of the derived Attribute causes the information upon which the Attribute is derived to be updated.	
type:	Boolean
multiplicity:	exactly one

IDL

```
interface MofAttributeClass : StructuralFeatureClass {
    readonly attribute MofAttributeSet all_of_type_mof_attribute;
    readonly attribute MofAttributeSet all_of_class_mof_attribute;

  MofAttribute create_mof_attribute (
    /* from ModelElement */              in wstring name,
    /* from ModelElement */              in wstring annotation,
    /* from Feature */                   in ::Model::ScopeKind scope,
    /* from Feature */                   in ::Model::VisibilityKind visibility,
    /* from StructuralFeature */         in ::Model::MultiplicityType multiplicity,
    /* from StructuralFeature */         in boolean is_changeable,
    /* from MofAttribute */              in boolean is_derived)
      raises (Reflective::MofError);
};// end of interface MofAttributeClass

interface MofAttribute : MofAttributeClass, StructuralFeature {
  boolean is_derived ()
     raises (Reflective::MofError);
  void set_is_derived (in boolean new_value)
    raises (Reflective::MofError);
};
```

7.4.17 Reference

A Reference defines a Classifier's knowledge of, and access to, links and their instances defined by an Association. Although a Reference derives much of its state from a corresponding AssociationEnd, it provides additional information; therefore, the MOF cannot adequately represent some meta-models without this mechanism. The inherited attributes defined in StructuralFeature (multiplicity and is_changeable) are constrained to match the values of its corresponding AssociationEnd. However, it has its own visibility, name, and annotation defined. For further discussion on Reference, its purpose, and how it derives its attributes, see 7.2.2, "Associations," on page 34.

© ISO/IEC 2005 - All rights reserved

NOTE: When creating a Reference, values for the inherited attributes of multiplicity and is_changeable must be supplied. These must be the same as the corresponding attributes on the AssociationEnd to which the Reference will subsequently be linked.

SuperClasses

StructuralFeature

References

exposedEnd

The exposedEnd of a Reference is the AssociationEnd representing the end of the Reference's owning Classifier within the defining Association.	
class	AssociationEnd
defined by:	Exposes::exposedEnd
multiplicity:	exactly one
changeable:	yes

referencedEnd

The referencedEnd of a Reference is the end representing the set of LinkEnds of principle interest to the Reference. The Reference provides access to the instances of that AssociationEnd's class, which are participants in that AssociationEnd's Association, connected through that AssociationEnd's LinkEnds. In addition, the Reference derives the majority of its state information - multiplicity, etc., from that Reference.	
class:	AssociationEnd
defined by:	RefersTo::referencedEnd
multiplicity:	exactly one
changeable:	yes

Constraints

The multiplicity for a Reference must be the same as the multiplicity for the referenced AssociationEnd. [C-21] on page 112.

Classifier scoped References are not meaningful in the current M1 level computational model. [C-22] on page 112.

A Reference can be changeable only if the referenced AssociationEnd is also changeable. [C-23] on page 112.

The type attribute of a Reference and its referenced AssociationEnd must be the same. [C-24] on page 113.

A Reference is only allowed for a navigable AssociationEnd. [C-25] on page 113.

The containing Class for a Reference must be equal to or a subtype of the type of the Reference's exposed AssociationEnd. [C-26] on page 113.

The referenced AssociationEnd for a Reference must be visible from the Reference. [C-27] on page 114.

© ISO/IEC 2005 - All rights reserved

IDL

```
interface ReferenceClass : StructuralFeatureClass {
    readonly attribute ReferenceSet all_of_type_reference;
    readonly attribute ReferenceSet all_of_class_reference;
    const string REFERENCE_MULTIPLICITY_MUST_MATCH_END =
     "org.omg.mof:constraint.model.reference.reference_multiplicity_must_match_end";
    const string REFERENCE_MUST_BE_INSTANCE_SCOPED =
     "org.omg.mof:constraint.model.reference.reference_must_be_instance_scoped";
    const string CHANGEABLE_REFERENCE_MUST_HAVE_CHANGEABLE_END =
     "org.omg.mof:constraint.model.reference.changeable_reference_must_have_changeable_end";
    const string REFERENCE_TYPE_MUST_MATCH_END_TYPE =
     "org.omg.mof:constraint.model.reference.reference_type_must_match_end_type";
    const string REFERENCED_END_MUST_BE_NAVIGABLE =
     "org.omg.mof:constraint.model.reference.referenced_end_must_be_navigable";
    const string CONTAINER_MUST_MATCH_EXPOSED_TYPE =
     "org.omg.mof:constraint.model.reference.container_must_match_exposed_type";
    const string REFERENCED_END_MUST_BE_VISIBLE =
     "org.omg.mof:constraint.model.reference.referenced_end_must_be_visible";

    Reference create_reference (
     /* from ModelElement */                in wstring name,
     /* from ModelElement */                in wstring annotation,
     /* from Feature */                     in ::Model::ScopeKind scope,
     /* from Feature */                     in ::Model::VisibilityKind visibility,
     /* from StructuralFeature */           in ::Model::MultiplicityType, multiplicity,
     /* from StructuralFeature */           in boolean is_changeable)
       raises (Reflective::MofError);
  }; // end of interface ReferenceClass

  interface Reference : ReferenceClass, StructuralFeature {
    AssociationEnd exposed_end ()
      raises (Reflective::MofError);
    void set_exposed_end (in AssociationEnd new_value)
      raises (Reflective::MofError);
    AssociationEnd referenced_end ()
      raises (Reflective::MofError);
    void set_referenced_end (in AssociationEnd new_value)
      raises (Reflective::MofError);
  };
```

7.4.18 BehavioralFeature (abstract)

A BehavioralFeature defines a dynamic characteristic of the ModelElement that contains it. Because a BehavioralFeature is partially defined by the Parameters it contains, it is both a Feature and a Namespace.

SuperClasses

Feature, Namespace

IDL

```
interface BehavioralFeatureClass : FeatureClass, NamespaceClass {
     readonly attribute BehavioralFeatureUList
          all_of_type_behavioral_feature;
```


© ISO/IEC 2005 - All rights reserved

}; // end of interface BehavioralFeatureClass

interface BehavioralFeature :
BehavioralFeatureClass, Feature , Namespace {};

7.4.19 Operation

An Operation defines a dynamic feature that offers a service. The behavior of an operation is activated through the invocation of the operation.

SuperClasses

BehavioralFeature

Contained Elements

Parameter, Constraint; see OperationContainmentRules [C-28] on page 114.

Attributes

isQuery

Defining an Operation with an isQuery value of true denotes that the behavior of the operation will not alter the state of the object. The state of a Classifier, for this definition, is the set of values of all of the Classifier's class-scope and instance-scope StructuralFeatures. For instance, an Operation of a Class, defined with a scope of instance, will not change the values or instances of any instance-scope StructuralFeature of the Class instance, as a result of invoking this Operation. An Operation of a Class with a scope of classifier will not change the values or instances of any of the classifier-scope StructuralFeatures or instance-scope StructuralFeatures. This attribute does not define a constraint enforced by the model, but rather a promise that the operation's implementation is expected to uphold. An operation that is not defined as isQuery equals false is not guaranteed to change the state of its object. The isQuery constraint does not proscribe any specific implementation, so long as the definition of isQuery above is observed.	
type:	Boolean
multiplicity:	exactly one

References

exceptions

An Operation, upon encountering an error or other abnormal condition, may raise an Exception. The exceptions reference provides the Operation with the set of Exceptions it is allowed to raise.	
class:	Exception
defined by:	CanRaise::except
multiplicity:	zero or more, ordered

© ISO/IEC 2005 - All rights reserved

Constraints

An Operation may only contain Parameters, Constraints, and Tags. [C-28] on page 114.

An Operation may have at most one Parameter whose direction is “return.” [C-29] on page 114.

The Exceptions raised by an Operation must be visible to the Operation. [C-30] on page 115.

IDL

```
interface OperationClass : BehavioralFeatureClass {
    readonly attribute OperationSet all_of_type_operation;
    readonly attribute OperationSet all_of_class_operation;
    const string OPERATION_CONTAINMENT_RULES =
     "org.omg.mof:constraint.model.operation.operation_containment_rules";
    const string OPERATIONS_HAVE_AT_MOST_ONE_RETURN =
     "org.omg.mof:constraint.model.operation.operations_have_at_most_one_return";
    const string OPERATION_EXCEPTIONS_MUST_BE_VISIBLE =
     "org.omg.mof:constraint.model.operation.operation_exceptions_must_be_visible";

   Operation create_operation (
     /* from ModelElement */          in wstring name,
     /* from ModelElement */          in wstring annotation,
     /* from Feature */               in ::Model::ScopeKind scope,
     /* from Feature */               in ::Model::VisibilityKind visibility,
     /* from Operation */             in boolean is_query)
          raises (Reflective::MofError);
}; // end of interface OperationClass

  interface Operation : OperationClass, BehavioralFeature {
    boolean is_query ()
       raises (Reflective::MofError);
    void set_is_query (in boolean new_value)
      raises (Reflective::MofError);
    MofExceptionUList exceptions ()
       raises (Reflective::MofError);
     void set_exceptions (in MofExceptionUList new_value)
      raises (Reflective::MofError);
    void add_exceptions (in MofException new_element)
      raises (Reflective::MofError);
    void add_exceptions_before (in MofException new_element, in MofException before_element)
      raises (Reflective::NotFound, Reflective::MofError);
     void modify_exceptions (in MofException old_element,
                    in MofException new_element)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove_exceptions (in MofException old_element)
      raises (Reflective::NotFound, Reflective::MofError);
  };
```

7.4.20 Exception (idl_substitute_name “MofException”)

An Exception (referred to as a MofException in the mapped IDL) defines an error or other abnormal condition. The Parameters of an Exception hold a record of an occurrence of the exceptional condition.

© ISO/IEC 2005 - All rights reserved

SuperClasses

BehavioralFeature

Contained Elements

Parameter; see ExceptionContainmentRules [C-31] on page 115.

Constraints

An Exception may only contain Parameters and Tags. [C-31] on page 115.

An Exception's Parameters must all have the direction "out." [C-32] on page 115.

IDL

```
interface MofExceptionClass : BehavioralFeatureClass {
    readonly attribute MofExceptionSet all_of_type_mof_exception;
    readonly attribute MofExceptionSet all_of_class_mof_exception;
    const string EXCEPTION_CONTAINMENT_RULES =
     "org.omg.mof:constraint.model.mof_exception.exception_containment_rules";
    const string EXCEPTIONS_HAVE_ONLY_OUT_PARAMETERS =
     "org.omg.mof:constraint.model.mof_exception.exceptions_have_only_out_parameters";

   MofException create_mof_exception (
         /* from ModelElement */  in wstring name,
         /* from ModelElement */  in wstring annotation,
         /* from Feature */       in ::Model::ScopeKind scope,
         /* from Feature */       in ::Model::VisibilityKind visibility)
      raises (Reflective::MofError);
}; // end of interface MofExceptionClass

interface MofException : MofExceptionClass, BehavioralFeature {};
```

7.4.21 Association

An association defines a classification over a set of links, through a relationship between Classifiers. Each link that is an instance of the association denotes a connection between object instances of the Classifiers of the Association. The MOF restricts associations to binary, restricting each link to two participating objects. This restriction also means that the association is defined between two Classifiers (which may be the same Classifier). The name of the Association is considered directional if it provides a clearer or more accurate representation of the association when stated with one participating class first rather than the other. For instance, Operation CanRaise Exception is correct; Exception CanRaise Operation is incorrect.

An Association contains at least two AssociationEnds, each of which has a Class as its "type." A Class has knowledge of its participation in an Association if it contains a Reference that is related to the Association's Ends, as shown in Figure 7.6. The "type" of a Reference must be the "type" of the AssociationEnd that is the Reference's "referencedEnd." The "type" of the Reference's "exposedEnd" must be the Reference's containing Class, or a supertype of that Class.

© ISO/IEC 2005 - All rights reserved

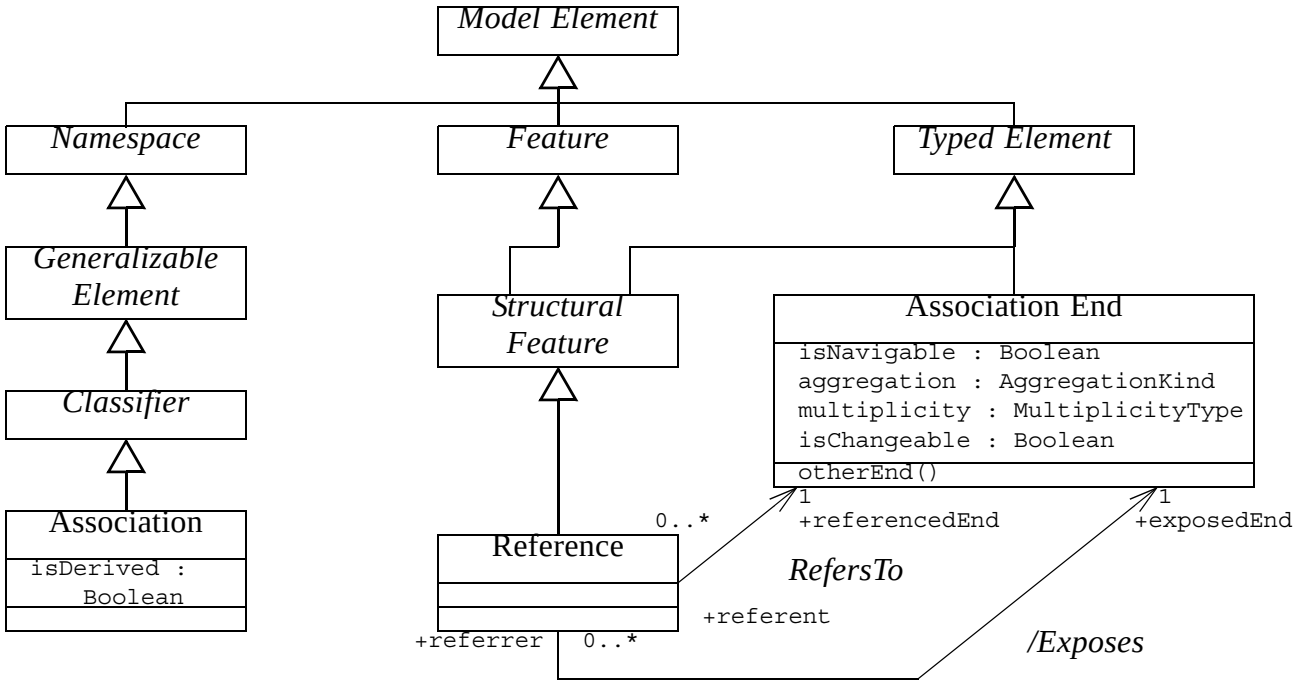

Figure 7.6 - MOF Model Elements for Associations

SuperClasses

Classifier

Contained Elements

AssociationEnd, Constraint; see AssociationContainmentRules [C-33] on page 116.

Attributes

isDerived

A derived association has no Links as instances. Instead, its Links are derived from other information in a meta-model. The addition, removal, or modification of a derived Association's Link causes the information upon which the Association is derived to be updated. The results of such an update are expected to appear, upon subsequent access of the derived Association's Links, to have the same effect as an equivalent operation on an Association that is not derived.	
type:	Boolean
multiplicity:	exactly one

Constraints

An Association may only contain AssociationEnds, Constraints, and Tags. [C-33] on page 116.

© ISO/IEC 2005 - All rights reserved

Inheritance / generalization is not applicable to Associations. [C-34] on page 116.

The values for “isLeaf” and “isRoot” on an Association must be true. [C-35] on page 116.

An Association cannot be abstract. [C-36] on page 117.

Associations must have visibility of “public.” [C-37] on page 117.

An Association must not be unary; that is, it must have at least two AssociationEnds. [C-38] on page 117.

IDL

```
interface AssociationClass : ClassifierClass {
    readonly attribute AssociationSet all_of_type_association;
    readonly attribute AssociationSet all_of_class_association;
    const string ASSOCIATIONS_CONTAINMENT_RULES =
     "org.omg.mof:constraint.model.association.associations_containment_rules";
    const string ASSOCIATIONS_HAVE_NO_SUPERTYPES =
     "org.omg.mof:constraint.model.association.associations_have_no_supertypes";
    const string ASSOCIATIONS_MUST_BE_ROOT_AND_LEAF =
     "org.omg.mof:constraint.model.association.associations_must_be_root_and_leaf";
    const string ASSOCIATIONS_CANNOT_BE_ABSTRACT =
     "org.omg.mof:constraint.model.association.associations_cannot_be_abstract";
    const string ASSOCIATIONS_MUST_BE_PUBLIC =
     "org.omg.mof:constraint.model.association.associations_must_be_public";
    const string ASSOCIATIONS_MUST_BE_BINARY =
     "org.omg.mof:constraint.model.association.associations_must_be_binary";

  Association create_association (
     /* from ModelElement */                 in wstring name,
     /* from ModelElement */                 in wstring annotation,
     /* from GeneralizableElement */         in boolean is_root,
     /* from GeneralizableElement */         in boolean is_leaf,
     /* from GeneralizableElement */         in boolean is_abstract,
     /* from GeneralizableElement */         in ::Model::VisibilityKind visibility,
     /* from Association */                  in boolean is_derived)
          raises (Reflective::MofError);
}; // end of interface AssociationClass

interface Association : AssociationClass, Classifier {

    boolean is_derived ()
       raises (Reflective::MofError);
    void set_is_derived (in boolean new_value)
      raises (Reflective::MofError);
};
```

7.4.22 AssociationEnd

An association is composed of two AssociationEnds. Each AssociationEnd defines a Classifier participant in the Association, the role it plays, and constraints on sets of the Classifier instances participating. An instance of an AssociationEnd is a LinkEnd, which defines a relationship between a link, in instance of an Association, and an instance of the AssociationEnd’s Classifier, provided in its type attribute.

© ISO/IEC 2005 - All rights reserved

SuperClasses

TypedElement

Attributes

isNavigable

The isNavigable attribute determines whether or not the AssociationEnd supports link “navigation.” This has two implications: • A Class defined with an appropriate Reference supports navigation of links from one Class instance to another. If isNavigable is false for an AssociationEnd, no such References may be created. • Setting isNavigable to false also suppress as a mapping’s mechanisms for indexing links based on this AssociationEnd.	
type:	Boolean
multiplicity:	exactly one

aggregation

Certain associations define aggregations - directed associations with additional semantics (see 8.10, “Aggregation Semantics,” on page 152). When an AssociationEnd is defined as composite or shared, the instance at “this” end of a Link is the composite or aggregate, and the instance at the “other” end is the component or subordinate.	
type:	AggregationKind
multiplicity:	exactly one

multiplicity

Multiplicity defines constraints on sets of instances. Each instance of the Classifier defined by the opposite AssociationEnd’s type defines a set that this multiplicity attribute constrains. Given one of those instances, x, the set is defined as the instances connected by LinkEnds of this AssociationEnd to that instance x. Refer to 7.6.1, “PrimitiveTypes used in the MOF Model,” on page 96 for a description on how the multiplicity attribute constrains a set. In its use is describing AssociationEnds, isUnique has been constrained to be true, as a simplification. This constraint means that the same two instances cannot participate in more than one Link while participating under the same AssociationEnd. Normally, two instances cannot be linked by more than one Link of an Association at all. But when the AssociationEnd types allow the two instances switch ends, they can form a second Link without violating the isUnique constraint.	
type:	MultiplicityType
multiplicity:	exactly one

© ISO/IEC 2005 - All rights reserved

isChangeable

The isChangeable attribute restricts the capability to perform actions that would modify sets of instances corresponding to this AssociationEnd (the same sets to which multiplicity is applied). Specifically, the set may be created when the instance defining the set - the instance at the opposite end of the Links - is created. This attribute does not make the set immutable. Instead, it affects the generation of operations in Model Elaboration that would allow modification of the set. For IDL generation, the only operation that allows the set to be modified would be one or more factory operations that create the instance and create the set. The modeler is free to define specific operations that allow modification of the set. Note that defining this AssociationEnd with isChangeable equals false places restrictions on the changeability of the other AssociationEnd, due to their interdependence.	
type:	Boolean
multiplicity:	exactly one

Operations

otherEnd

Provides the other AssociationEnd (i.e., not this one) in the enclosing Association.	
return type:	AssociationEnd
isQuery:	yes
parameters:	none
operation semantics:	[S-2] on page 125

Constraints

The type of an AssociationEnd must be Class. [C-39] on page 117.

The “isUnique” flag in an AssociationEnd’s multiplicity must be true. [C-40] on page 118.

An Association cannot have two AssociationEnds marked as “ordered.” [C-41] on page 118.

An Association cannot have an aggregation semantic specified for both AssociationEnds. [C-42] on page 118.

IDL

```
interface AssociationEndClass : TypedElementClass {
   readonly attribute AssociationEndSet all_of_type_association_end;
   readonly attribute AssociationEndSet all_of_class_association_end;
   const string END_TYPE_MUST_BE_CLASS =
     "org.omg.mof:constraint.model.association_end.end_type_must_be_class";
   const string ENDS_MUST_BE_UNIQUE =
     "org.omg.mof:constraint.model.association_end.ends_must_be_unique";
   const string CANNOT_HAVE_TWO_ORDERED_ENDS =
     "org.omg.mof:constraint.model.association_end.cannot_have_two_ordered_ends";
   const string CANNOT_HAVE_TWO_AGGREGATE_ENDS =
     "org.omg.mof:constraint.model.association_end.cannot_have_two_aggregate_ends";

  AssociationEnd create_association_end (
         /* from ModelElement */                in wstring name,
```

© ISO/IEC 2005 - All rights reserved

```
            /* from ModelElement */              in wstring annotation,
            /* from AssociationEnd */            in boolean is_navigable,
            /* from AssociationEnd */            in ::Model::AggregationKind aggregation,
            /* from AssociationEnd */            in ::Model::MultiplicityType multiplicity,
            /* from AssociationEnd */            in boolean is changeable)
  raises (Reflective::MofError);
}; // end of interface AssociationEndClass

interface AssociationEnd : AssociationEndClass, TypedElement {
     boolean is_navigable ()
        raises (Reflective::MofError);
     void set_is_navigable (in boolean new_value)
       raises (Reflective::MofError);
     AggregationKind aggregation ()
        raises (Reflective::MofError);
     void set_aggregation (in AggregationKind new_value)
       raises (Reflective::MofError);
     MultiplicityType multiplicity ()
       raises (Reflective::MofError);
     void set_multiplicity (in MultiplicityType new_value)
        raises (Reflective::MofError);
     boolean is changeable ()
       raises (Reflective::MofError);
      void set is changeable (in boolean new value);
     AssociationEnd other_end ()
        raises (Reflective::MofError);
}; // end of interface AssociationEnd
```

7.4.23 Package

A Package is a container for a collection of related ModelElements that form a logical meta-model. Packages may be composed and related in the following ways:

- A Package can contain nested Packages via the Contains association.
- A Package can inherit from other Packages via the Generalizes association.
- A Package can import or cluster other Namespaces, including Packages via an Import and the Aliases association.

© ISO/IEC 2005 - All rights reserved

The model elements for representing Packages are shown in Figure 7.7.

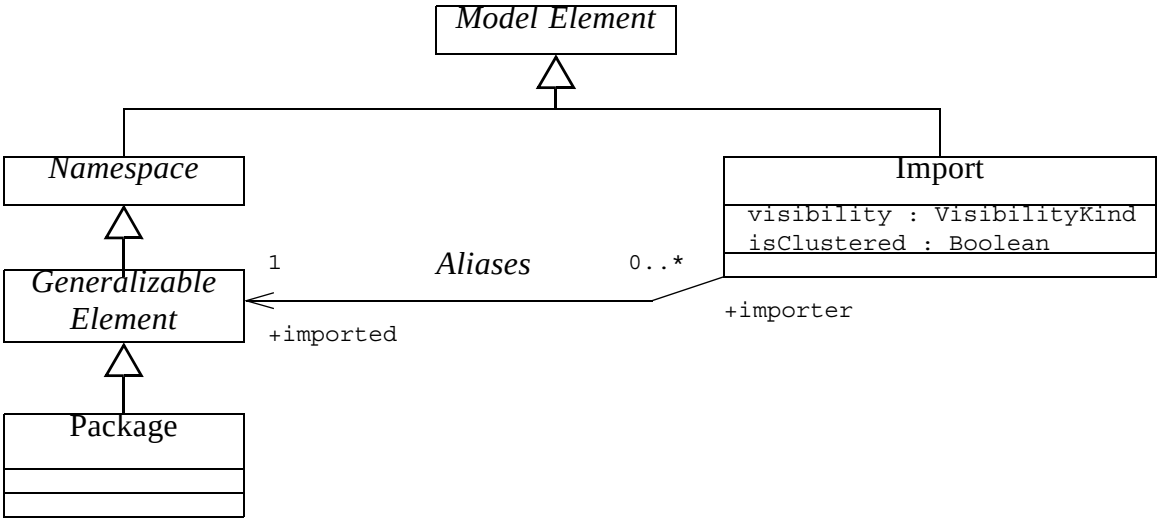

Figure 7.7- MOF Model Packaging

SuperClasses

GeneralizableElement

Contained Elements

Package, Class, Association, DataType, Exception, Import, Constraint, Constant; see PackageContainmentRules [C-43] on page 119.

Operations

none

Constraints

A Package may only contain Packages, Classes, DataTypes, Associations, Exceptions, Constants, Constraints, Imports, and Tags. [C-43] on page 119.

Packages cannot be declared as abstract. [C-44] on page 119.

IDL

```
interface PackageClass : GeneralizableElementClass {
    readonly attribute PackageSet all_of_type_package;
    readonly attribute PackageSet all_of_class_package;
    const string PACKAGE_CONTAINMENT_RULES =
        "org.omg.mof:constraint.model.package.package_containment_rules";
    const string PACKAGES_CANNOT_BE_ABSTRACT =
        "org.omg.mof:constraint.model.package.packages_cannot_be_abstract";
    Package create_package (
        /* from ModelElement */          in wstring name,
        /* from ModelElement */          in wstring annotation,
```

© ISO/IEC 2005 - All rights reserved

```
            /* from GeneralizableElement */        in boolean is_root,
            /* from GeneralizableElement */        in boolean is_leaf,
            /* from GeneralizableElement */        in boolean is_abstract,
            /* from GeneralizableElement */        in ::Model::VisibilityKind visibility)
        raises (Reflective::MofError);
}; // end of interface PackageClass

  interface Package : PackageClass, GeneralizableElement {
};
```

7.4.24 Import

An Import allows a Package to make use of ModelElements defined in some other Namespace. An Import object is related to another Namespace via the Aliases association. When a Package contains an Import object, it imports the associated Namespace. This means that ModelElements defined within the imported Namespace are visible in the importing Package.

An Import allows the visibility of the imported Package's contained ModelElements to be further restricted. An Import object represents either Package importing or Package clustering, depending on the "isClustered" attribute.

SuperClasses

ModelElement

Attributes

visibility

In the future, this Attribute will modify the visibility of imported ModelElements in the context of the importing Namespace. For a description of visibility kinds, see 7.6.3, "VisibilityKind," on page 97. The MOF rules of visibility are not currently specified.	
type:	VisibilityKind
multiplicity:	exactly one

isClustered

The isClustered flag determines whether the Import object represents simple Package importation, or Package clustering.	
type:	Boolean
multiplicity:	exactly one

© ISO/IEC 2005 - All rights reserved

References

importedNamespace

The Import knows about the Namespace that it references.	
class:	Namespace
defined by:	Aliases::imported
multiplicity:	exactly one

Constraints

The Namespace imported by an Import must be visible to the Import's containing Package. [C-45] on page 119.

It is only legal for a Package to import or cluster Packages or Classes. [C-46] on page 120.

Packages cannot import or cluster themselves. [C-47] on page 120.

Packages cannot import or cluster Packages or Classes that they contain. [C-48] on page 120.

Nested Packages cannot import or cluster other Packages or Classes. [C-49] on page 121, [C-9] on page 108.

IDL

```
interface ImportClass : ModelElementClass {
    readonly attribute ImportSet all_of_type_import;
    readonly attribute ImportSet all_of_class_import;
  const string IMPORTED_NAMESPACE_MUST_BE_VISIBLE =
    "org.omg.mof:constraint.model.import.imported_namespace_must_be_visible";
  const string CAN_ONLY_IMPORT_PACKAGES_AND_CLASSES =
    "org.omg.mof:constraint.model.import.can_only_import_packages_and_classes";
  const string CANNOT_IMPORT_SELF =
    "org.omg.mof:constraint.model.import.cannot_import_self";
  const string CANNOT_IMPORT_NESTED_COMPONENTS =
    "org.omg.mof:constraint.model.import.cannot_import_nested_components";
  const string NESTED_PACKAGES_CANNOT_IMPORT =
    "org.omg.mof:constraint.model.import.nested_packages_cannot_import";

   Import create_import (
        /* from ModelElement */          in wstring name,
        /* from ModelElement */          in wstring annotation,
        /* from Import */                in ::Model::VisibilityKind visibility,
        /* from Import */                in boolean is_clustered)
     raises (Reflective::MofError);
};// end of interface ImportClass

interface Import : ImportClass, ModelElement {
  VisibilityKind visibility ()
     raises (Reflective::MofError);
  void set_visibility (in VisibilityKind new_value)
     raises (Reflective::MofError);
  boolean is_clustered ()
     raises (Reflective::MofError);
  void set_is_clustered (in boolean new_value)
     raises (Reflective::MofError);
```

© ISO/IEC 2005 - All rights reserved

```
    Namespace imported_namespace ()
      raises (Reflective::MofError);
    void set_imported_namespace (in Namespace new_value)
      raises (Reflective::MofError);
};
```

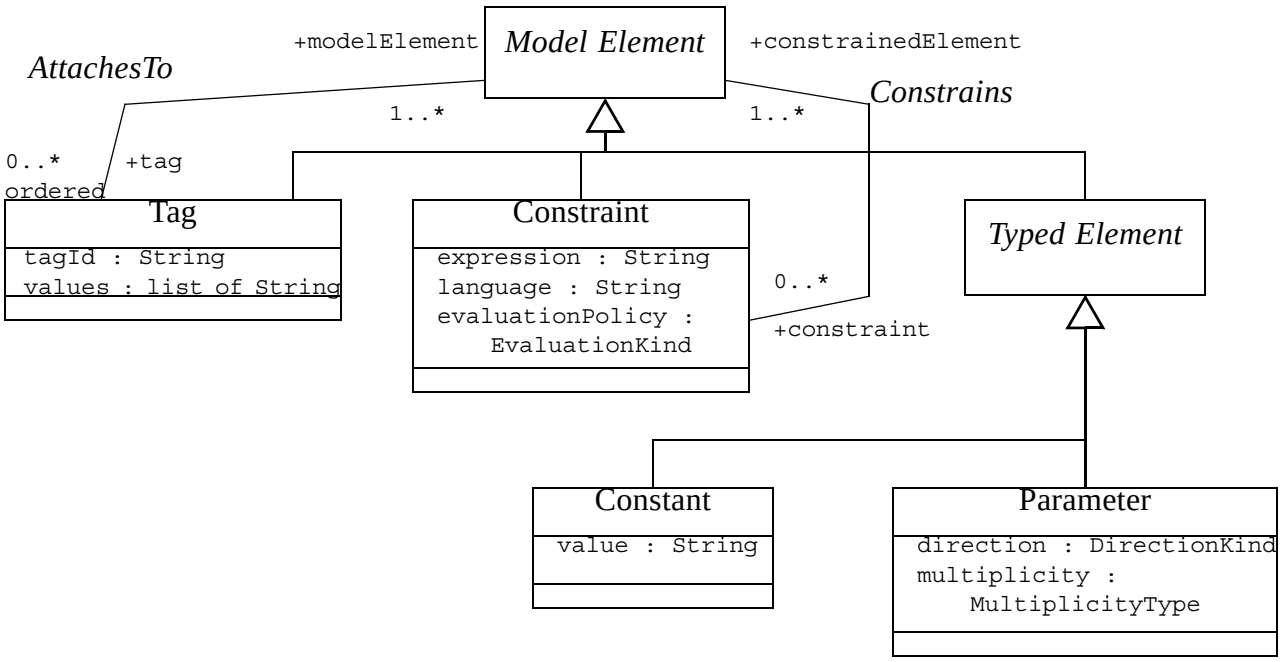

Figure 7.8 - MOF Model - Other Elements

7.4.25 Parameter

A parameter provides a means of communication with operations and other BehavioralFeatures. A parameter passes or communicates values of its defined type.

SuperClasses

TypedElement

Attributes

direction

This attribute specifies the purpose of the parameter; to input a value, to output a value, both purposes, or to provide an operation return value.	
type:	DirectionKind
multiplicity:	exactly one

© ISO/IEC 2005 - All rights reserved

multiplicity

Multiplicity defines cardinality constraints on the set of instances or values that a Parameter can hold. Multiplicity defines a lower and upper bound on the set, although the upper bound can be specified as Unbounded. Additionally, multiplicity defines two other characteristics of the set: 1) constraints on set member ordering, and 2) constraints on unique set elements. Specifically, Multiplicity contains an isOrdered field. When isOrdered is true, the ordering of the elements in the set are preserved. Multiplicity also has an isUnique field. When isUnique is true, the set is constrained to hold no more than one of any value or instance.	
type:	MultiplicityType
multiplicity:	exactly one

IDL

```
interface ParameterClass : TypedElementClass {
    readonly attribute ParameterSet all_of_type_parameter;
    readonly attribute ParameterSet all_of_class_parameter;

    Parameter create_parameter (
        /* from ModelElement */          in wstring name,
        /* from ModelElement */          in wstring annotation,
        /* from Parameter */             in ::Model::DirectionKind direction,
        /* from Parameter */             in ::Model::MultiplicityType multiplicity)
       raises (Reflective::MofError);
  }; // end of interface ParameterClass

interface Parameter : ParameterClass, TypedElement {
    DirectionKind direction ()
       raises (Reflective::MofError);
    void set_direction (in DirectionKind new_value)
      raises (Reflective::MofError);
    MultiplicityType multiplicity ()
      raises (Reflective::MofError);
    void set_multiplicity (in MultiplicityType new_value)
       raises (Reflective::MofError);
  };
```

7.4.26 Constraint

A Constraint defines a rule that restricts the state or behavior of one or more elements in the meta-model. When a Constraint is attached to a ModelElement, the rule it encodes applies to all relevant instances of the ModelElement in a model.

A Constraint rule, represented by the “expression” attribute, may be encoded in any form. The “language” attribute may be used to denote the language and encoding scheme used.

While some Constraints on a model may need to be treated as invariant, it is often convenient for other Constraints to be relaxed, for instance while a model is being edited. While, the “evaluationPolicy” attribute is used to represent these two cases, this information is at best advisory, since the MOF specification does not currently state how and when Constraints should be enforced.

© ISO/IEC 2005 - All rights reserved

NOTE: A Constraint cannot override structural integrity rules defined by other parts of a meta-model (e.g., multiplicity specifications) or the integrity rules defined by a particular mapping of the meta-model to implementation technology.

SuperClasses

ModelElement

Attributes

expression

The Constraint's expression attribute contains a textual representation of the constraint. The MOF has no specific requirement that an implementation must be able to interpret this expression, or to validate it against the language attribute. The meaning of the expression will depend on the language used.	
type	String
multiplicity:	exactly one

language

A Constraint's language attribute gives the name of textual language used in the constraint expression.	
type:	String
multiplicity:	exactly one

evaluationPolicy

Each constraint can be defined as immediate or deferred. For immediate Constraints, the constraint violation will be detected and reported within an operation in the chain of operations between the operation initiated by the MOF user and the operation that caused the constraint violation. The effect of an operation that violates an immediate constraint on the state of the object or objects being altered is implementation specific, and possibly undefined. However, if possible, an implementation should reverse the effects of the operation. For deferred Constraints, the constraint violation can only be detected when the Constraint is explicitly evaluated. A technology mapping will typically provide a verify operation. When a verify operation is invoked on instance of a constrained element, the Constraint will be checked and any violation will be reported.	
type:	EvaluationKind
multiplicity:	exactly one

© ISO/IEC 2005 - All rights reserved

References

constrainedElements

The Constraint has access to the ModelElements it constrains, through this reference. Note that the Constraint may depend on other ModelElements not represented in this reference. For instance, a Constraint may state that attribute A::x cannot exceed A::y in magnitude. The Constraint is on A::x, although it also depends on A::y. The relationship between the Constraint and A::y is not explicitly stated in the meta-model.	
class:	ModelElement
defined by:	Constrains::constrainedElement
multiplicity:	one or more

Constraints

Constraints, Tags, Imports, and Constants cannot be constrained. [C-50] on page 121.

A Constraint can only constrain ModelElements that are defined by or inherited by its immediate container. [C-51] on page 122.

IDL

```
interface ConstraintClass : ModelElementClass {
    readonly attribute ConstraintSet all_of_type_constraint;
    readonly attribute ConstraintSet all_of_class_constraint;
    const string CANNOT_CONSTRAIN_THIS_ELEMENT =
      "org.omg.mof:constraint.model.constraint.cannot_constrain_this_element";
    const string CONSTRAINTS_LIMITED_TO_CONTAINER =
      "org.omg.mof:constraint.model.constraint.constraints_limited_to_container";

   enum EvaluationKind {immediate, deferred};

   Constraint create_constraint (
           /* from ModelElement */  in wstring name,
           /* from ModelElement *  in wstring annotation,
           /* from Constraint */    in wstring expression,
           /* from Constraint */    in wstring language,
           /* from Constraint */    in ::Model::ConstraintClass::EvaluationKind evaluation_policy)
        raises (Reflective::MofError);
  }; // end of interface ConstraintClass

  interface Constraint : ConstraintClass, ModelElement {
    wstring expression ()
       raises (Reflective::MofError);
    void set_expression (in wstring new_value)
       raises (Reflective::MofError);
    wstring language ()
       raises (Reflective::MofError);
    void set_language (in wstring new_value)
      raises (Reflective::MofError);
    ConstraintClass::EvaluationKind evaluation_policy ()
       raises (Reflective::MofError);
    void set_evaluation_policy (in ConstraintClass::EvaluationKind new_value)
```

© ISO/IEC 2005 - All rights reserved

```
      raises (Reflective::MofError);
    ModelElementSet constrained_elements ()
      raises (Reflective::MofError);
     void set_constrained_elements (in ModelElementSet new_value)
       raises (Reflective::MofError);
    void add_constrained_elements (in ModelElement new_element)
       raises (Reflective::MofError);
    void modify_constrained_elements (in ModelElement old_element,
                          in ModelElement new_element)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove_constrained_elements (in ModelElement old_element)
      raises (Reflective::NotFound, Reflective::MofError);
 };
```

7.4.27 Constant

Constant provides a mechanism for defining constant values for use in meta-models. Constants are limited to values of types defined as PrimitiveType instances.

SuperClasses

TypedElement

Attributes

value

This Attribute gives the literal value of the constant encoded as text. The syntax for encoding literal values of the standard MOF primitive data types is given in 7.10, “The PrimitiveTypes Package,” on page 134. Other encodings are mapping, vendor or user specific.	
type	String
multiplicity:	exactly one

Constraints

The type of a Constant and its value must be compatible. [C-52] on page 122.

The type of a Constant must be a PrimitiveType. [C-53] on page 122.

IDL

```
interface ConstantClass : TypedElementClass {
   readonly attribute ConstantSet all_of_type_constant;
   readonly attribute ConstantSet all_of_class_constant;
   const string CONSTANTS_VALUE_MUST_MATCH_TYPE =
     "org.omg.mof:constraint.model.constant.constants_value_must_match_type";
   const string CONSTANTS_TYPE_MUST_BE_SIMPLE_DATA_TYPE =
     "org.omg.mof:constraint.model.constant.constants_type_must_be_simple_data_type";

   Constant create_constant (
       /* from ModelElement */      in wstring name,
       /* from ModelElement */      in wstring annotation,
```


© ISO/IEC 2005 - All rights reserved

```
      /* from Constant */              in wstring value)
      raises (Reflective::MofError);
}; // end of interface ConstantClass

  interface Constant : ConstantClass, TypedElement {
    wstring value ()
       raises (Reflective::MofError);
    void set_value (in wstring new_value)
      raises (Reflective::MofError);
  };
```

7.4.28 Tag

Tags provide a light-weight extension mechanism that allows mapping, vendor, and even customer specific information to be added to, or associated with a meta-model. In essence, Tags are arbitrary name / value pairs that can be attached to instances of most ModelElements.

A Tag has an attribute called “tagId” that denotes a category of meaning, and another attribute called “values” that parameterizes that meaning. Each Tag is related to one or more ModelElements by the AttachesTo Association. The Tag need not be contained within the meta-model of the ModelElement it “tags.”

The MOF specification does not generally define the values for the “tagId” or the application specific categories of meaning that they denote. The exception to this is:

- 9.6, “Standard Tags for the IDL Mapping,” on page 194 defines some Tags that tailor the IDL produced by the IDLmapping.

Since “tagId” values are not standardized, there is a risk that different vendors or user organizations will use the same values to denote different categories of meaning. If a “tagId” value is used to mean different things, problems can arise when meta-models using the value are exchanged.

To avoid such Tag collisions, it is recommended that “tagId” values should use the following scheme based on Java package naming. Each value should start with a prefix formed by reversing the Internet domain name of a “tagId” naming authority. This should be followed by a locally unique component. For instance, this might be a standard or product name followed by a name or names that denotes the meaning. Here are some examples:

```
"org.omg.mof.idl_prefix"
"org.omg.mof.some_tag"
"com.rational.rose.screen_position"
"au.edu.dstc.elvin.event_type"
```

It is also recommended that “tagId” values should be spelled in all lower case using the underscore (“_”) character as a word separator.

NOTE: In defining new Tag categories, the meta-modeler should take account of the fact that the MOF Model has no Reference for navigating from a ModelElement to its attached Tags. This allows one to attach Tags to elements of a “frozen” meta-model. On the other hand, it makes it harder for a “client” of the meta-model objects to find the Tags for an element. One option is to require relevant Tags to be Contained by the elements they AttachTo, or their parents.

SuperClasses

ModelElement

© ISO/IEC 2005 - All rights reserved

Attributes

tagId

Gives the category of meaning for the Tag. The values for this attribute and their associated meanings are not standardized here. See discussion above.	
type:	String
multiplicity:	exactly one

values

Carries additional information (e.g., “parameters”) associated with the Tag. The encoding of parameters as String values is tagId specific.	
type:	String
multiplicity:	zero or more, ordered.

References

elements

The ModelElement or ModelElements that this Tag is attached to.	
class:	ModelElement
defined by:	AttachesTo::modelElement
multiplicity:	one or more

IDL

```
interface TagClass : ModelElementClass {
    readonly attribute TagSet all_of_type_tag;
    readonly attribute TagSet all_of_class_tag;

    Tag create_tag (
        /* from ModelElement */in wstring name,
        /* from ModelElement */in wstring annotation,
       /* from Tag */          in wstring tag_id,
       /* from Tag */          in ::PrimitiveTypes::WStringList values)
      raises (Reflective::MofError);
}; // end of interface TagClass

  interface Tag : TagClass, ModelElement {
    wstring tag_id ()
        raises (Reflective::MofError);
    void set_tag_id (in wstring new_value)
        raises (Reflective::MofError);
    ::PrimitiveTypes::WStringList values ()
        raises (Reflective::MofError);
    void set_values (in ::PrimitiveTypes::WStringList new_value)
        raises (Reflective::MofError);
    void add_values (in wstring new_element)
```


© ISO/IEC 2005 - All rights reserved

```
        raises (Reflective::MofError);
    void add_values_at(in wstring new_element, in unsigned long position)
        raises (Reflective::BadPosition, Reflective::MofError);
    void modify_values (in wstring old_element, in wstring new_element)
        raises (Reflective::NotFound, Reflective::MofError);
    void modify_values_at(in wstring new_element, in unsigned long position)
        raises (Reflective::BadPosition, Reflective::MofError);
    void remove_values (in wstring old_element)
        raises (Reflective::NotFound, Reflective::MofError);
    void remove_values_at(in unsigned long position)
        raises (Reflective::BadPosition, Reflective::MofError);
    ModelElementUList elements ()
      raises (Reflective::MofError);
    void set_elements (in ModelElementUList new_value)
       raises (Reflective::MofError);
   void add_elements (in ModelElement new_element)
       raises (Reflective::MofError);
   void add_elements_before (in ModelElement new_element, in ModelElement before_element)
      raises (Reflective::NotFound, Reflective::MofError);
    void modify_elements (in ModelElement old_element, in ModelElement new_element)
      raises (Reflective::NotFound, Reflective::MofError);
  void remove_elements (in ModelElement old_element)
      raises (Reflective::NotFound, Reflective::MofError);
};
```

7.5 MOF Model Associations

7.5.1 Contains

A meta-model is defined through a composition of ModelElements. A Namespace defines a ModelElement that composes other ModelElements. Since Namespace has several subclasses, there is a sizable combinatorial set of potential Namespace-ModelElement pairings. However, some of these pairings are not appropriate for building an object-oriented meta-model, such as a Class containing a Package (see 7.3.4, “The MOF Model Containment Hierarchy,” on page 40). This approach factors the container mechanisms into one abstraction, and allows the greatest flexibility for future changes to the MOF Model.

Ends

container

Each Namespace is a composition of zero or more ModelElements.	
class:	Namespace
multiplicity:	zero or one
aggregation:	Namespace forms a composite aggregation of ModelElements.

© ISO/IEC 2005 - All rights reserved

containedElement

Each ModelElement, with the exception of top-level packages participates in the association as a containedElement.	
class:	ModelElement
multiplicity:	Zero or more; ordered.

IDL

```
interface Contains : Reflective::RefAssociation {
    ContainsLinkSet all_contains_links ()
        raises (Reflective::MofError);
    boolean exists (in Namespace container, in ModelElement contained_element)
     raises (Reflective::MofError);
    Namespace container (in ModelElement contained_element)
     raises (Reflective::MofError);
    ModelElementUList contained_element (in Namespace container)
     raises (Reflective::MofError);
    void add (in Namespace container, in ModelElement contained_element)
      raises (Reflective::MofError);
    void add_before_contained_element (in Namespace container,
                        in ModelElement contained_element,
                        in ModelElement before)
     raises (Reflective::NotFound, Reflective::MofError);
    void modify_container (in Namespace container,
                        in ModelElement contained_element,
                        in Namespace new_container)
     raises (Reflective::NotFound, Reflective::MofError);
    void modify_contained_element (in Namespace container,
                        in ModelElement contained_element,
                        in ModelElement new_contained_element)
     raises (Reflective::NotFound, Reflective::MofError);
    void remove (in Namespace container, in ModelElement contained_element)
     raises (Reflective::NotFound, Reflective::MofError);
 };
```

7.5.2 Generalizes

The Association defined on GeneralizableElement. A Link of this Association represents a supertype/subtype relationship (or a generalizes/specializes relationship).

Ends

supertype

The GeneralizableElement that is more general is the supertype.	
class:	GeneralizabelElement
multiplicity:	Zero or more (a GeneralizableElement may have zero or more supertypes); ordered.

© ISO/IEC 2005 - All rights reserved

subtype

The subtype is the GeneralizableElement that is more specific. The supertype Generalizes the subtype.	
class:	GeneralizableElement
multiplicity:	Zero or more (a GeneralizableElement may have zero or more subtypes).

IDL

```
interface Generalizes : Reflective::RefAssociation {
    GeneralizesLinkSet all_generalizes_links ()
          raises (Reflective::MofError);
    boolean exists (in GeneralizableElement supertype,
                in GeneralizableElement subtype)
      raises (Reflective::MofError);
    GeneralizableElementUList supertype (in GeneralizableElement subtype)
      raises (Reflective::MofError);
    GeneralizableElementSet subtype (in GeneralizableElement supertype)
      raises (Reflective::MofError);
     void add (in GeneralizableElement supertype,
           in GeneralizableElement subtype)
      raises (Reflective::MofError);
    void add_before_supertype (in GeneralizableElement supertype,
                        in GeneralizableElement subtype,
                        in GeneralizableElement before)
      raises (Reflective::NotFound, Reflective::MofError);
     void modify_supertype (in GeneralizableElement supertype,
                        in GeneralizableElement subtype,
                        in GeneralizableElement new_supertype)
      raises (Reflective::NotFound, Reflective::MofError);
    void modify_subtype (in GeneralizableElement supertype,
                        in GeneralizableElement subtype,
                        in GeneralizableElement new_subtype)
      raises (Reflective::NotFound, Reflective::MofError);
    void remove (in GeneralizableElement supertype,
                        in GeneralizableElement subtype)
      raises (Reflective::NotFound, Reflective::MofError);
  };
```

7.5.3 RefersTo

A Reference derives most of its state from the AssociationEnd that it is linked to, based on this Association. For a Class defined with a Reference, each of its instances can be used to access the referenced object or objects. Those referenced objects will be of the Class defined by this referencedEnd AssociationEnd, playing the defined end.

© ISO/IEC 2005 - All rights reserved

Ends

referent

The Reference that is providing the reference through which instances playing the end-defined by the AssociationEnd can be accessed.	
class:	Reference
multiplicity:	Zero or more; not ordered (an AssociationEnd may or may not be used by any number of References).

referencedEnd

The AssociationEnd which provides the majority of information for the Reference, including the LinkEnds that supply the referenced instances.	
class:	AssociationEnd
multiplicity:	exactly one

IDL

```
interface RefersTo : Reflective::RefAssociation {
    RefersToLinkSet all_refers_to_links ()
        raises (Reflective::MofError);
    boolean exists (in Reference referent, in AssociationEnd referenced_end)
     raises (Reflective::MofError);
    ReferenceSet referent (in AssociationEnd referenced_end)
     raises (Reflective::MofError);
    AssociationEnd referenced_end (in Reference referent)
     raises (Reflective::MofError);
    void add (in Reference referent, in AssociationEnd referenced_end)
     raises (Reflective::MofError);
    void modify_referent (in Reference referent,
                          in AssociationEnd referenced_end,
                          in Reference new_referent)
     raises (Reflective::NotFound, Reflective::MofError);
    void modify_referenced_end (in Reference referent,
                          in AssociationEnd referenced_end,
                          in AssociationEnd new_referenced_end)
     raises (Reflective::NotFound, Reflective::MofError);
    void remove (in Reference referent, in AssociationEnd referenced_end)
     raises (Reflective::NotFound, Reflective::MofError);
 };
```

7.5.4 Exposes (derived)

A Reference defines a reference for a Class. For an instance of that class, which holds one or more links to some object or objects conforming to the reference, the instance will be playing the role (end) defined by the AssociationEnd in this Association.

© ISO/IEC 2005 - All rights reserved

Ends

referrer

The Reference that is providing the exposedEnd's class instances within the Reference's Classifier.	
class:	Reference
multiplicity:	Zero or more; not ordered (an AssociationEnd may or may not be used by any number of References).
changeable:	yes

exposedEnd

The AssociationEnd representing the Reference's owning Classifier's end in the Association.	
class:	AssociationEnd
multiplicity:	exactly one
changeable:	yes

Derivation

See [S-13] on page 130. For a given Reference, the Link of this Association is derived as follows:

- The referrer's Reference is the given Reference.
- The exposedEnd's AssociationEnd is the given Reference's referent's container Association's other AssociationEnd.

IDL

```
interface Exposes : Reflective::RefAssociation {
    ExposesLinkSet all_exposes_links ()
        raises (Reflective::MofError);
    boolean exists (in Reference referrer, in AssociationEnd exposed_end)
     raises (Reflective::MofError);
    ReferenceSet referrer (in AssociationEnd exposed_end)
     raises (Reflective::MofError);
    AssociationEnd exposed_end (in Reference referrer)
     raises (Reflective::MofError);
    void add (in Reference referrer, in AssociationEnd exposed_end)
      raises (Reflective::MofError);
    void modify_referrer (in Reference referrer,
                          in AssociationEnd exposed_end,
                          in Reference new_referrer)
     raises (Reflective::NotFound, Reflective::MofError);
    void modify_exposed_end (in Reference referrer,
                          in AssociationEnd exposed_end,
                          in AssociationEnd new_exposed_end)
     raises (Reflective::NotFound, Reflective::MofError);
    void remove (in Reference referrer, in AssociationEnd exposed_end)
     raises (Reflective::NotFound, Reflective::MofError);
 };
```

© ISO/IEC 2005 - All rights reserved

7.5.5 IsOfType

A Link between a TypedElement subclass and a Classifier supports the definition of the TypedElement.

Ends

type

The type defining the TypedElement.	
class:	Classifier
multiplicity:	exactly one

typedElements

The set of typed elements supported by a Classifier.	
class:	TypedElement
multiplicity:	zero or more

IDL

```
interface IsOfType : Reflective::RefAssociation {
    IsOfTypeLinkSet all_is_of_type_links ()
          raises (Reflective::MofError);
     boolean exists (in Classifier type, in TypedElement typed_elements)
      raises (Reflective::MofError);
     Classifier type (in TypedElement typed_elements)
      raises (Reflective::MofError);
     TypedElementSet typed_elements (in Classifier type)
      raises (Reflective::MofError);
     void add (in Classifier type, in TypedElement typed_elements)
       raises (Reflective::MofError);
     void modify_type (in Classifier type,
                       in TypedElement typed_elements,
                       in Classifier new_type)
      raises (Reflective::NotFound, Reflective::MofError);
     void modify_typed_elements (in Classifier type,
                       in TypedElement typed_elements,
                       in TypedElement new_typed_elements)
      raises (Reflective::NotFound, Reflective::MofError);
     void remove (in Classifier type, in TypedElement typed_elements)
      raises (Reflective::NotFound, Reflective::MofError);
  };
```

7.5.6 CanRaise

Relates Operations to the Exceptions that they can raise.

© ISO/IEC 2005 - All rights reserved

Ends

operation

Given an Exception, the set of Operations which can Raise that Exception.	
class:	Operation
multiplicity:	Zero or more (an Exception may be defined that is not currently used by any Operation; an Exception may be raised by multiple Operations).

except

The set of Exceptions for an Operation.	
class:	Exception
multiplicity:	Zero or more (an Operation may be defined to raise no exception, or multiple exceptions); ordered (an Operation's Exceptions are ordered).

IDL

```
interface CanRaise : Reflective::RefAssociation {
    CanRaiseLinkSet all_can_raise_links ()
        raises (Reflective::MofError);
    boolean exists (in ::Model::Operation operation, in MofException except)
        raises (Reflective::MofError);
    OperationSet operation (in MofException except)
        raises (Reflective::MofError);
    MofExceptionUList except (in ::Model::Operation operation)
        raises (Reflective::MofError);
    void add (in ::Model::Operation operation, in MofException except)
        raises (Reflective::MofError);
    void add_before_except (in ::Model::Operation operation,
                in MofException except,
                in MofException before)
        raises (Reflective::NotFound, Reflective::MofError);
    void modify_operation (in Operation operation,
                in MofException except,
                in Operation new_operation)
        raises (Reflective::NotFound, Reflective::MofError);
    void modify_except (in ::Model::Operation operation,
                in MofException except,
                in MofException new_except)
        raises (Reflective::NotFound, Reflective::MofError);
    void remove (in ::Model::Operation operation, in MofException except)
        raises (Reflective::NotFound, Reflective::MofError);
 };
```

7.5.7 Aliases

An Import aliases or imports a single Namespace.

© ISO/IEC 2005 - All rights reserved

Ends

importer

A Namespace may be aliased by an Import, which is the importer.	
class:	Import
multiplicity:	Zero or more (a Namespace may not be aliased, or may be aliased by multiple Imports).

imported

The Namespace that an Import imports or aliases.	
class:	Namespace
multiplicity:	exactly one

IDL

```
interface Aliases : Reflective::RefAssociation {
    AliasesLinkSet all_aliases_links ()
        raises (Reflective::MofError);
    boolean exists (in Import importer, in Namespace imported)
        raises (Reflective::MofError);
    ImportSet importer (in Namespace imported)
        raises (Reflective::MofError);
    Namespace imported (in Import importer)
        raises (Reflective::MofError);
    void add (in Import importer, in Namespace imported)
        raises (Reflective::MofError);
    void modify_importer (in Import importer,
                in Namespace imported,
                in Import new_importer)
        raises (Reflective::NotFound, Reflective::MofError);
    void modify_imported (in Import importer,
                in Namespace imported,
                in Namespace new_imported)
        raises (Reflective::NotFound, Reflective::MofError);
    void remove (in Import importer, in Namespace imported)
        raises (Reflective::NotFound, Reflective::MofError);
 };
```

7.5.8 Constrains

Each Constraint constrains one or more ModelElements.

© ISO/IEC 2005 - All rights reserved

Ends

constraint

A Constraint that constrains a ModelElement.	
class:	Constraint
multiplicity:	Zero or more (a ModelElement need not be constrained, but could be constrained by more than one Constraint).

constrainedElement

The ModelElements that a Constraint holds its constraint against.	
class:	ModelElement
multiplicity:	One or more (a Constraint must constrain at least one ModelElement).

IDL

```
interface Constrains : Reflective::RefAssociation {
    ConstrainsLinkSet all_constrains_links ()
        raises (Reflective::MofError);
    boolean exists (in ::Model::Constraint constraint,
                    in ModelElement constrained_element)
        raises (Reflective::MofError);
    ConstraintSet constraint (in ModelElement constrained_element);
        raises (Reflective::MofError)
    ModelElementSet constrained_element (in ::Model::Constraint constraint)
        raises (Reflective::MofError);
    void add (in ::Model::Constraint constraint,
                    in ModelElement constrained_element)
        raises (Reflective::MofError);
    void modify_constraint (in ::Model::Constraint constraint,
                    in ModelElement constrained_element,
                    in Constraint new_constraint)
        raises (Reflective::NotFound, Reflective::MofError);
    void modify_constrained_element (in ::Model::Constraint constraint,
                    in ModelElement constrained_element,
                    in ModelElement new_constrained_element)
        raises (Reflective::NotFound, Reflective::MofError);
    void remove (in ::Model::Constraint constraint,
                    in ModelElement constrained_element)
        raises (Reflective::NotFound, Reflective::MofError);
};
```

7.5.9 DependsOn (derived)

DependsOn is a derived Association that allows a client to identify the collection of ModelElements on which a given ModelElement *structurally* depends. The Association is derived from a number of other Associations in the MOF Model, as described below.

© ISO/IEC 2005 - All rights reserved

NOTE: The model of dependency that is embodied in this Association is based solely on the structural relationships within a meta-model. In some cases, the structural dependencies have clear semantic parallels (e.g., the meaning of an Attribute depends on its type). In other cases the semantic parallel is more tenuous (e.g., a DataType only semantically depends on its container in the context of type identity).

Ends

dependent

This End is occupied by ModelElements that *structurally* depend on the ModelElement at the other End.	
class:	ModelElement
multiplicity:	Zero or more (a ModelElement can have no ModelElement depend on it, or many may depend on it).
changeable:	no

provider

This End is occupied by ModelElements that have other ModelElements that *structurally* depend on them.	
class:	ModelElement
multiplicity:	Zero or more (a ModelElement can depend on no other ModelElements or multiple ModelElements).
changeable:	no

Derivation

See [S-14] on page 131. A ModelElement (ME) depends on:

- “container” - its container Namespace from ModelElement::container
- “constraint” - any Constraints from ModelElement::constraints.
- “contents” - if ME is a Namespace, its contents from Namespace::contents.
- “specialization” - if ME is a GeneralizableElement, its supertypes from GeneralizableElement::supertypes.
- “import” if ME is an Import, the imported Package or Class from Import::importedNamespace.
- “signature” - if ME is an Operation, the Exceptions it raises from Operation::exceptions.
- “type definition” - if ME is a TypedElement, the Classifier from TypedElement::type.
- “referenced ends” - if ME is a Reference, the two AssociationEnds from Reference::referencedEnd and Reference::exposedEnd.
- “constrained elements” - if ME is a Constraint, the elements it constrains from Constraint::constrainedElements.
- “tagged elements” - if ME is a Tag, the elements it is attached to from Tag::elements.

© ISO/IEC 2005 - All rights reserved

IDL

```
interface DependsOn : Reflective::RefAssociation {
    DependsOnLinkSet all_depends_on_links ()
        raises (Reflective::MofError);
    boolean exists (in ModelElement dependent, in ModelElement provider)
        raises (Reflective::MofError);
    ModelElementSet dependent (in ModelElement provider)
        raises (Reflective::MofError);
    ModelElementSet provider (in ModelElement dependent)
        raises (Reflective::MofError);
};
```

7.5.10 AttachesTo

This association represents Tags attached to ModelElements. A ModelElement's Tags are ordered, although the ordering may not be of any significance, depending on the meaning of the Tags. Ordering is preserved in case some Tags, in conjunction with some defined semantics, requires an ordering.

Ends

modelElement

The ModelElements that an attached Tag describes, modifies, or otherwise associates.	
class:	ModelElement
multiplicity:	One or more (a Tag must be attached to at least one ModelElement).

tag

The set of Tags attached to a ModelElement.	
class:	Tag
multiplicity:	Zero or more (a ModelElement need not have a Tag), ordered.

IDL

```
interface AttachesTo : Reflective::RefAssociation {
    AttachesToLinkSet all_attaches_to_links ()
        raises (Reflective::MofError);
    boolean exists (in ModelElement model_element, in ::Model::Tag tag)
        raises (Reflective::MofError);
    ModelElementSet model_element (in ::Model::Tag tag)
        raises (Reflective::MofError);
    TagUList tag (in ModelElement model_element)
        raises (Reflective::MofError);
    void add (in ModelElement model_element, in ::Model::Tag tag)
        raises (Reflective::MofError);
    void add_before_tag (in ModelElement model_eleme
                         in ::Model::Tag tag,
                         in Tag before)
        raises (Reflective::NotFound, Reflective::MofError);
    void modify_model_element (in ModelElement model_element,
```

© ISO/IEC 2005 - All rights reserved

```
                    in ::Model::Tag tag,
                    in ModelElement new_model_element)
        raises (Reflective::NotFound, Reflective::MofError);
    void modify_tag (in ModelElement model_element,
                    in ::Model::Tag tag,
                    in Tag new_tag)
        raises (Reflective::NotFound, Reflective::MofError);
    void remove (in ModelElement model_element, in ::Model::Tag tag)
        raises (Reflective::NotFound, Reflective::MofError);
};
```

7.6 MOF Model Data Types

The following data types are part of the MOF Model. Each data type is represented in the MOF Model as an instance of the appropriate DataType subclass.

7.6.1 PrimitiveTypes used in the MOF Model

The only PrimitiveType instances used in the MOF Model are Boolean, Integer, and String. These are specified in “The PrimitiveTypes Package” on page 134.

NOTE: The PrimitiveTypes package defines 6 standard PrimitiveType instances in total. The other three instances (Long, Float, and Double) are not used in the specification of the MOF Model as an instance of itself.

7.6.2 MultiplicityType

MultiplicityType is a structure (record) type that is used to specify the multiplicity properties of an Attribute, Parameter, Reference, or AssociationEnd.

Fields

lower

This field gives the lower bounds on the number of elements allowed for the Attribute, Parameter, Reference, or AssociationEnd.	
type:	Integer

upper

This field gives the upper bounds on the number of elements allowed for the Attribute, Parameter, Reference, or AssociationEnd. A value of Unbounded (see 7.8.1, “Unbounded,” on page 100) indicates that there is no upper bound on the number of elements.	
type:	Integer

© ISO/IEC 2005 - All rights reserved

isOrdered

This flag indicates whether the order of the elements corresponding to the Attribute, Parameter, Reference, or AssociationEnd has any semantic significance.	
type:	Boolean

isUnique

This flag indicates whether or not the elements corresponding to the Attribute, Parameter, Reference, or AssociationEnd are required (or guaranteed) to be unique.	
type:	Boolean

Constraints

The “lower” bound of a MultiplicityType to be “Unbounded.” [C-54] on page 123.

The “lower” bound of a MultiplicityType cannot exceed the “upper” bound. [C-55] on page 123.

The “upper” bound of a MultiplicityType cannot be less than 1. [C-56] on page 123.

If a MultiplicityType specifies bounds of [0..1] or [1..1]), the “is_ordered” and “is_unique” values must be false. [C-57] on page 124.

IDL

```
struct MultiplicityType {
        long lower;
        long upper;
        boolean isOrdered;
        boolean isUnique;
    };

const string LOWER_CANNOT_BE_NEGATIVE_OR_UNBOUNDED =
    "org.omg:constraint.model.multiplicity_type.lower_cannot_be_negative_or_unbounded";
const string LOWER_CANNOT_EXCEED_UPPER =
    "org.omg:constraint.model.multiplicity_type.lower_cannot_exceed_upper";
const string UPPER_MUST_BE_POSITIVE =
    "org.omg:constraint.model.multiplicity_type.upper_must_be_positive";
const string MUST_BE_UNORDERED_NONUNIQUE =
    "org.omg:constraint.model.multiplicity_type.must_be_unordered_nonunique";
```

7.6.3 VisibilityKind

This data type enumerates the three possible kinds of visibility for a ModelElement outside of its container. These are:

1. “public_vis,” which allows anything that can use ModelElement’s container to also use the ModelElement.
2. “protected_vis,” which allows use of the ModelElement within containers that inherits from this one’s container.
3. “private_vis,” which denies all outside access to the ModelElement.

NOTE: The rules governing visibility of ModelElements in the MOF are yet to be specified. As an interim measure, all

© ISO/IEC 2005 - All rights reserved

ModelElements are deemed to be visible, irrespective of the “visibility” attribute settings. The IDL mapping specification includes minimal preconditions on visibility to ensure that generated IDL is compilable (see 9.5, “Preconditions for IDL Generation,” on page 192).

IDL

```
enum VisibilityKind {public_vis, private_vis, protected_vis};
```

7.6.4 DirectionKind

DirectionKind enumerates the possible directions of information transfer for Operation and Exception Parameters.

IDL

```
enum DirectionKind {in_dir, out_dir, inout_dir, return_dir};
```

7.6.5 ScopeKind

ScopeKind enumerates the possible “scopes” for Attributes and Operations.

IDL

```
enum ScopeKind {instance_level, classifier_level};
```

7.6.6 AggregationKind

AggregationKind enumerates the possible aggregation semantics for Associations (specified via AssociationEnds).

NOTE: Aggregation semantics in the MOF is intended to be aligned with UML. Unfortunately, the OMG UML specification does not define the meaning of “shared” aggregation for UML. As an interim measure, the use of “shared” aggregation in MOF meta-models is discouraged.

IDL

```
enum AggregationKind {none, shared, composite};
```

7.6.7 EvaluationKind

EvaluationKind enumerates the possible models for Constraint evaluation.

Container

Constraint

IDL

```
enum EvaluationKind {immediate, deferred};
```


© ISO/IEC 2005 - All rights reserved

7.7 MOF Model Exceptions

The following exceptions are contained in the MOF Model Package. The generated IDL interfaces for the MOF Model make use of more exceptions, which are defined in the Reflective Package (see the Reflective Type Packages clause) and assigned to operations based on criteria determinable during generation.

7.7.1 NameNotFound

The NameNotFound exception is raised when a lookup of a simple name has failed.

parameters

name : out String

The name parameter gives the string value that could not be found in the Namespace or extended Namespace searched by the operation.

Container

Namespace

7.7.2 NameNotResolved

The NameNotResolved exception is raised when resolution of a qualified name has failed.

parameters

explanation : out String
restOfName : out String (multiplicity: zero or more; ordered; not unique)

The restOfName parameter contains that part of the qualified name that was not resolved. The explanation parameter can have the following values with the corresponding interpretation:

- “InvalidName”: the first name in restOfName was malformed.
- “MissingName”: the first name in restOfName could not be resolved as no name binding exists for that name.
- “NotNameSpace”: the first name in restOfName did not resolve to a NameSpace when a NameSpace was expected.
- “CannotProceed”: the first name in restOfName could not be resolved (for any other reason).

Container

Namespace

7.8 MOF Model Constants

The following Constants form part of the MOF Model.

© ISO/IEC 2005 - All rights reserved

7.8.1 Unbounded

This constant is used in the context of MultiplicityType to represent an unlimited upper bound on a cardinality (see 7.6.1, “PrimitiveTypes used in the MOF Model,” on page 96). Its type is Integer.

Container

Model

IDL

```
const long UNBOUNDED = -1;
```

7.8.2 The Standard DependencyKinds

These constants (ContainerDep, ContentsDep, SignatureDep, ConstraintDep, ConstrainedElementsDep, SpecializationDep, ImportDep, TypeDefinitionDep, ReferencedEndsDep, TaggedElementsDep, IndirectDep, and AllDep) denote the standard dependency categories and pseudo-categories. Their types are all String.

When a ModelElement depends on a second model element under one kind of dependency; and the second model element depends on a third under some other kind of dependency; then the first ModelElement depends on the third ModelElement. However, the kind of dependency cannot be specified, based on the other two dependency kinds, except to categorize the dependency as indirect.

Refer to 7.4.1, “ModelElement (abstract),” on page 41 and 7.5.9, “DependsOn (derived),” on page 93 for detailed explanations.

Container

ModelElement

IDL

```
const wstring CONTAINER_DEP = "container";
const wstring CONTENTS_DEP = "contents";
const wstring SIGNATURE_DEP = "signature";
const wstring CONSTRAINT_DEP = "constraint";
const wstring CONSTRAINED_ELEMENTS_DEP = "constrained elements";
const wstring SPECIALIZATION_DEP = "specialization";
const wstring IMPORT_DEP = "import";
const wstring TYPE_DEFINITION_DEP = "type definition";
const wstring REFERENCED_ENDS_DEP = "referenced ends";
const wstring TAGGED_ELEMENTS_DEP = "tagged elements";
const wstring INDIRECT_DEP = "indirect";
const wstring ALL_DEP = "all";
```


© ISO/IEC 2005 - All rights reserved

7.9 MOF Model Constraints

7.9.1 MOF Model Constraints and other M2 Level Semantics

This sub clause defines the semantic constraints that apply to the MOF Model. These are expressed as M2-level Constraints and are formally part of the MOF Model (i.e., they are a required part of a representation of the MOF Model as MOF meta-objects or in the MOF Model / XMI interchange format).

The sub clause also provides OCL semantic specifications for most M2-level Operations, derived Attributes, and derived Associations in the MOF Model, and for a collection of “helper” functions used by them and the Constraints. These semantic specifications need not be present in a representation of the MOF Model. Indeed, this document does not specify how they should be represented.

NOTE: The use of OCL in the MOF Model specification does not imply a requirement to use OCL evaluation as part of a MOF Model server’s implementation. Furthermore, if that approach is used, it is anticipated that the implementor may rewrite the OCL rules to make evaluation more efficient. For example, the Constraint OCL could be rewritten as pre-conditions on the appropriate mapped update operations.

7.9.2 Notational Conventions

7.9.2.1 Notation for MOF Model Constraints

The M2-level Constraints on the MOF Model are described in the following notation:

[C-xxx]	ConstraintName
evaluation policy:	immediate or deferred
description:	brief english description

```
context SomeClassifierName
inv: ...
```

The meaning of the above is as follows:

- “[C-xxx]” is the cross reference tag for the Constraint used elsewhere in this document.
- “ConstraintName” is the name for the Constraint in the MOF Model. The IDL mapping uses this name to produce the MofError “kind” string for the Constraint. These strings appear in the generated IDL for the MOF Model, as described in 9.8.16, “Constraint Template,” on page 245.
- The “evaluation policy” states whether the Constaint should be checked on any relevant update operation, or whether checking should be deferred until full meta-model validation is triggered. It defines the Constraint’s “evaluationPolicy” value.
- The “description” is a brief non-normative synopsis of the Constraint. It could be used as the Constraint’s “annotation” value.
- The OCL for the Constraint is defined using the OCL syntax defined in UML 1.3.

The OCL for the Constraints start with a “context” clause that names a ModelElement in the MOF Model. This serves two purposes:

© ISO/IEC 2005 - All rights reserved

1. It defines the context in which the OCL constraint should be evaluated (i.e., the M3-level Class or DataType whose instances are constrained by the OCL).
2. It defines the "constrainedElements" and "container" for the Constraint.

While the OCL for the Constraints are mostly expressed as invariants, this should not be taken literally. Instead, the Constraint OCL should be viewed as:

- a pre-condition on the relevant IDL operations for "immediate" Constraints, or
- a part of the specification of ModelElement's "verify" Operation for "deferred" Constraints.

The Constraints in the MOF Model are expressed as restrictions on either Classes or DataTypes. Each one applies to ("Constrains") a single Classifier, and each one is defined to be contained by the Classifier that it applies. The "language" attribute of each Constraint is either "MOF-OCL" (for those with complete OCL specifications) or "Other." The "expression" attribute should be the normative OCL defined here, even if different (but equivalent) OCL is used in a MOF Model server's implementation.

7.9.2.2 Notation for Operations, derived Attributes and derived Association

The semantics of M2-level Operations, derived Attributes, and derived Associations on the MOF Model are described in the following notation:

[O-xxx]	ModelElementName
kind:	classification
description:	brief english description

```
context ClassifierName::OperationName(...) : ...
post: result = ...
```

or

```
context ClassName::AttributeName() : ...
post: result = ...
```

or

```
context ClassName::ReferenceName() : ...
post: result = ...
```

The meaning of the above is as follows:

- "[O-xxx]" is the cross reference tag for the semantic description that may be used elsewhere in this document.
- "ModelElementName" is the name of the Attribute, Operation, or Association in the MOF Model whose semantics is described.
- The "classification" describes the kind of the ModelElement (e.g., "readonly derived Attribute" or "query Operation").
- The "description" is a brief non-normative synopsis of the semantics.
- The OCL is expressed using the OCL syntax defined in the UML 1.4 specification (http://www.omg.org/technology/documents/formal/mof.htm). The "context" clause names an "abstract" operation or method on an M1 level interface whose semantics is specified. The name of the real operation(s) or method(s) will depend on the mapping. The semantics are expressed as post-conditions for these methods.

© ISO/IEC 2005 - All rights reserved

7.9.2.3 Notation for Helper Functions

OCL Helper Functions are described in the following notation:

[S-xxx]	HelperName
description:	brief english description

```
context ClassifierName::HelperName(...) : ...
post: result = ...
```

The meaning of the above is as follows:

- "[S-xxx]" is the cross reference tag for the helper function that may be used elsewhere in this document.
- "HelperName" is the name of the Helper Function.
- The "description" is a brief non-normative synopsis of the Helper's semantics.
- The OCL for the Helper is defined using the OCL syntax defined in UML 1.3. The "context" clause names a notional helper function on a ModelElement whose semantic is then specified. These notional functions are not intended to be callable by client code.

7.9.3 OCL Usage in the MOF Model specification

The OCL language was designed as a part of the UML specification. As such, the OCL semantics are specified in terms of UML concepts and constructs. Some of these concepts do not match MOF concepts exactly. Accordingly, it is necessary to reinterpret parts of the OCL specification so that it can be used in MOF Model's Constraints and other semantics aspects of the MOF Model.

7.9.3.1 UML AssociationEnds versus MOF References

In the UML version of OCL, the dot (".") and arrow ("->") operators are used to access Attribute values, and to navigate Associations. Consider an OCL expression of the form:

<expr> "." <identifier>

Assuming that "<expr>" evaluates to an object, the value of the expression is either the value of an Attribute named "<identifier>" for the object or another object obtained by navigating a link in a binary Association which has "<identifier>" as an Association End name.

In this context (i.e., the definition of the MOF Model), the "<identifier>" is interpreted differently. In the MOF Model, the interfaces for navigating Associations are specified using References rather than AssociationEnds. Thus in the MOF version of OCL, link navigation is expressed using the name of a Reference for the "<expr>" object as the "<identifier>." However, the overall meaning is analogous to the UML case.

7.9.3.2 Helper functions are not MOF Operations

In the UML version of OCL, object behavior is invoked by an expression of the form:

<expr> "." <identifier> "(" ... ")"

where "<identifier>" names a UML Operation or Method on the object obtained by evaluating "<expr>."

© ISO/IEC 2005 - All rights reserved

In the MOF Model specification, the above expression invokes behavior defined by either a MOF Operation, or a helper function. The distinction between conventional UML and its usage here is that helper functions have no defined connection with any internal or external interfaces in a MOF Model server. Indeed, they need not exist at all as implementation artifacts.

7.9.3.3 Post-conditions on MOF Model objects

Rules [C-2] on page 105, [C-3] on page 106, and [C-4] on page 106 are intended to define post-conditions on all operations on ModelElement objects. This is expressed in the MOF Model OCL by giving a Class rather than an Operation as the "context" for the OCL rules. It is not clear that this is allowed by UML OCL.

7.9.3.4 OCL evaluation order

The UML OCL specification does not define an evaluation order for OCL expressions in general, and for boolean operators in particular. This is OK when OCL is used as an abstract specification language, as it is in the UML specification. However it causes problems when OCL expressions may be directly evaluated. These problems arise in OCL that traverses cyclic graphs (e.g., [O-1] on page 131) or raises exceptions (e.g., [S-6] on page 127).

The MOF Model semantic specification touches on some of these issues (e.g., when traversing a cyclical Imports graph). Therefore, the MOF Model usage of OCL makes the following assumptions about OCL expression evaluation order:

- In general, a MOF OCL expression is assumed to be evaluated by evaluating its sub-expressions in order, starting with the leftmost sub-expression and ending with the rightmost. The sub-expressions are delimited according to the standard OCL operator precedence rules. If evaluation of one of the sub-expressions raises an exception, the remaining sub-expressions are not evaluated.
- The above does not apply to the boolean operators "**and**," "**or**," "**implies**," and "**if-then-else**." These are evaluated with short-circuiting as follows:
  - In the expression "<expr1> **and** <expr2>," "<expr2>" is only evaluated if "<expr1>" evaluates to true.
  - In the expression "<expr1> **or** <expr2>," "<expr2>" is only evaluated if "<expr1>" evaluates to false.
  - In the expression "<expr1> **implies** <expr2>," "<expr2>" is only evaluated if "<expr1>" evaluates to true.
  - In the expression "**if** <expr1> **then** <expr2> **else** <expr3> **endif**," "<expr2>" is only evaluated if "<expr1>" evaluates to true, and "<expr3>" is only evaluated if "<expr1>" evaluates to false.

7.9.3.5 "OclType::allInstances"

In UML OCL, the type.allInstances() is defined to return:

"The set of all instances of *type* and all of its subtypes in existence at the moment in time that the expression is evaluated."

In the MOF Model OCL, this expression is used to refer to the set of all instances that exist within a given outermost Package extent. (Any OCL expression that required the enumeration of all instances in existence "anywhere" would be problematical, since a MOF repository does not exist in a closed world.)

7.9.3.6 "OclType::references"

The MOF Model OCL in rule Clause [C-4] assumes that the signature of OclType (as defined in the UML OCL specification) is extended to include an operation called "references." This is assumed to behave like the "attributes" operation, except that it returns the names of an (M3-level) Classes' References.

© ISO/IEC 2005 - All rights reserved

7.9.3.7 Foreign types and operations

Some of the MOF Model OCL rules make use of types and operations that are not predefined in OCL, not defined as Operations in the MOF Model, and not defined as Helper functions. Examples include:

- Rule [C-3] on page 106 makes uses of the CORBA Object::non_existent operation to assert that an object must continue to exist. This would be expressed differently in other contexts.
- Rules [C-2] on page 105 and [C-4] on page 106 use operations defined in the RefObject and RefBaseObject interfaces to access the meta-objects that represent the MOF Model. It should be understood that this is not intended to imply that a MOF Model server is required to make these objects available at runtime.

7.9.4 The MOF Model Constraints

[C-1]	MustBeContainedUnlessPackage
format1:	MUST_BE_CONTAINED_UNLESS_PACKAGE
format2:	must_be_contained_unless_package
evaluation policy	deferred
description:	A ModelElement that is not a Package must have a container.

```
context ModelElement
inv:
   not self.oclIsTypeOf(Package) implies
          self.container -> size = 1
```

[C-2]	FrozenAttributesCannotBeChanged
format1:	FROZEN_ATTRIBUTES_CANNOT_BE_CHANGED
format2:	frozen_attributes_cannot_be_changed
evaluation policy	immediate
description:	The attribute values of a ModelElement that is frozen cannot be changed.

```
context ModelElement
inv:
   self.isFrozen() implies
          let myTypes = self.oclType() -> allSupertypes() ->
                includes(self.oclType()) in
          let myAttrs : Set(Attribute) =
                self.RefBaseObject::refMetaObject() ->
                      asOclType(Class) ->
                      findElementsByTypeExtended(Attribute) in
          myAttrs -> forAll(a |
                              self.RefObject::refValue@pre(a) =
                                  self.RefObject::refValue(a))
```

© ISO/IEC 2005 - All rights reserved

[C-3]	FrozenElementsCannotBeDeleted
format1:	FROZEN_ELEMENTS_CANNOT_BE_DELETED
format2:	frozen_elements_cannot_be_deleted
evaluation policy	immediate
description:	A frozen ModelElement that is in a frozen Namespace can only be deleted, by deleting the Namespace.

```
context ModelElement
post:
    (self.isFrozen@pre() and
     self.container@pre -> notEmpty and
     self.container.isFrozen@pre()) implies
    (self.container.Object::non_existent() or
     not self.Object::non_existent())
```

[C-4]	FrozenDependenciesCannotBeChanged
format1:	FROZEN_DEPENDENCIES_CANNOT_BE_CHANGED
format2:	frozen_dependencies_cannot_be_changed
evaluation policy	immediate
description:	The link sets that express dependencies of a frozen Element on other Elements cannot be explicitly changed.

```
context ModelElement
post:
    self.isFrozen() implies
           let myClasses = self.oclType() -> allSupertypes() ->
                  includes(self.oclType()) in
           let myRefs = Set(Reference) =
                  self.RefBaseObject::refMetaObject() ->
                         asOclType(Class) ->
                         findElementsByTypeExtended(Reference) in
           let myDepRefs = myRefs ->
                  select(r |
                                Set{"contents", "constraints", "supertypes",
                                 "type", "referencedEnd", "exceptions",
                                 "importedNamespace", "elements"} ->
                                includes(r.name)) in
           myDepRefs ->
                  forAll(r |
                                self.RefObject::refValue@pre(r) =
                                 self.RefObject::refValue(r))
```


© ISO/IEC 2005 - All rights reserved

[C-5]	ContentNamesMustNotCollide
format1:	CONTENT_NAMES_MUST_NOT_COLLIDE
format2:	content_names_must_not_collide
evaluation policy	immediate
description:	The names of the contents of a Namespace must not collide.

```
context Namespace
inv: self.contents.forAll(
          e1, e2 | e1.name = e2.name implies r1 = r2)
```

[C-6]	SupertypeMustNotBeSelf
format1:	SUPERTYPE_MUST_NOT_BE_SELF
format2:	supertype_must_not_be_self
evaluation policy	immediate
description:	A Generalizable Element cannot be its own direct or indirect supertype.

```
context GeneralizableElement
inv: self.allSupertypes() -> forAll(s | s <> self)
```

[C-7]	SupertypeKindMustBeSame
format1:	SUPERTYPE_KIND_MUST_BE_SAME
format2:	supertype_kind_must_be_same
evaluation policy	immediate
description:	A supertypes of a Generalizable Element must be of the same kind as the GeneralizableElement itself.

```
context GeneralizableElement
inv: self.supertypes -> forAll(s | s.oclType() = self.oclType())
```

© ISO/IEC 2005 - All rights reserved

[C-8]	ContentsMustNotCollideWithSupertypes
format1:	CONTENTS_MUST_NOT_COLLIDE_WITH_SUPERTYPES
format2:	contents_must_not_collide_with_supertypes
evaluation policy	immediate
description:	The names of the contents of a GeneralizableElement should not collide with the names of the contents of any direct or indirect supertype.

```
context GeneralizableElement
inv:
   let superContents = self.allSupertypes() ->
          collect(s | s.contents) in
   self.contents ->
          forAll(m1 |
                    superContents ->
                              forAll(m2 |
                                 m1.name = m2.name implies m1 = m2))
```

[C-9]	DiamondRuleMustBeObeyed
format1:	DIAMOND_RULE_MUST_BE_OBEYED
format2:	diamond_rule_must_be_obeyed
evaluation policy	immediate
description:	Multiple inheritance must obey the “Diamond Rule.”

```
context GeneralizableElement
inv:
   let superNamespaces =
          self.supertypes -> collect(s | s.extendedNamespace) in
   superNamespaces -> asSet -> isUnique(s | s.name)
```

[C-10]	NoSupertypesAllowedForRoot
format1:	NO_SUPERTYPES_ALLOWED_FOR_ROOT
format2:	no_supertypes_allowed_for_root
evaluation policy	immediate
description:	If a Generalizable Element is marked as a “root,” it cannot have any supertypes.

© ISO/IEC 2005 - All rights reserved

```
context GeneralizableElement
inv: self.isRoot implies self.supertypes -> isEmpty
```

[C-11]	SupertypesMustBeVisible
format1:	SUPERTYPES_MUST_BE_VISIBLE
format2:	supertypes_must_be_visible
evaluation policy	deferred
description:	A Generalizable Element’s immediate supertypes must all be visible to it.

```
context GeneralizableElement
inv: self.supertypes -> forAll(s | self.isVisible(s))
```

[C-12]	NoSubtypesAllowedForLeaf
format1:	NO_SUBTYPES_ALLOWED_FOR_LEAF
format2:	no_subtypes_allowed_for_leaf
evaluation policy	immediate
description:	A GeneralizableElement cannot inherit from a GeneralizableElement defined as a “leaf.”

```
context GeneralizableElement
inv: self.supertypes -> forAll(s | not s.isLeaf)
```

[C-13]	AssociationsCannotBeTypes
format1:	ASSOCIATIONS_CANNOT_BE_TYPES
format2:	associations_cannot_be_types
evaluation policy	immediate
description:	An Association cannot be the type of a TypedElement.

```
context TypedElement
inv: not self.type.oclIsKindOf(Association)
```

© ISO/IEC 2005 - All rights reserved

[C-14]	TypeMustBeVisible
format1:	TYPE_MUST_BE_VISIBLE
format2:	type_must_be_visible
evaluation policy	deferred
description:	A TypedElement and only have a type that is visible to it.

```
context TypedElement
inv: self.isVisible(self.type)
```

[C-15]	ClassContainmentRules
format1:	CLASS_CONTAINMENT_RULES
format2:	class_containment_rules
evaluation policy	immediate
description:	A Class may contain only Classes, DataTypes, Attributes, References, Operations, Exceptions, Constants, Constraints, and Tags.

```
context Class
inv:
   Set{Class, DataType, Attribute, Reference, Operation,
          Exception, Constant, Constraint, Tag} ->
                 includesAll(self.contentTypes())
```

[C-16]	AbstractClassesCannotBeSingleton
format1:	ABSTRACT_CLASSES_CANNOT_BE_SINGLETON
format2:	abstract_classes_cannot_be_singleton
evaluation policy	deferred
description:	A Class that is marked as abstract cannot also be marked as singleton.

```
context Class
inv: self.isAbstract implies not self.isSingleton
```


© ISO/IEC 2005 - All rights reserved

[C-17]	DataTypeContainmentRules
format1:	DATA_TYPE_CONTAINMENT_RULES
format2:	data_type_containment_rules
evaluation policy	immediate
description:	A DataType may contain only TypeAliases, Constraints, Tags (or in the case of StructureTypes) StructureFields.

```
context DataType
inv:
   if self.oclIsOfType(StructureType)
   then
          Set{TypeAlias, Constraint, Tag, StructureField} ->
                includesAll(self.contentTypes())
   else
          Set{TypeAlias, Constraint, Tag} ->
                includesAll(self.contentTypes())
```

[C-18]	<Placeholder for a deleted constraint>

[C-19]	DataTypesHaveNoSupertypes
format1:	DATA_TYPES_HAVE_NO_SUPERTYPES
format2:	data_types_have_no_supertypes
evaluation policy	immediate
description:	Inheritance / generalization is not applicable to DataTypes.

```
context DataType
inv: self.supertypes -> isEmpty
```

[C-20]	DataTypesCannotBeAbstract
format1:	DATA_TYPES_CANNOT_BE_ABSTRACT
format2:	data_types_cannot_be_abstract
evaluation policy	immediate
description:	A DataType cannot be abstract.

© ISO/IEC 2005 - All rights reserved

```
context DataType
inv: not self.isAbstract
```

[C-21]	ReferenceMultiplicityMustMatchEnd
format1:	REFERENCE_MULTIPLICITY_MUST_MATCH_END
format2:	reference_multiplicity_must_match_end
evaluation policy	deferred
description:	The multiplicity for a Reference must be the same as the multiplicity for the referenced AssociationEnd.

```
context Reference
inv: self.multiplicity = self.referencedEnd.multiplicity
```

[C-22]	ReferenceMustBeInstanceScoped
format1:	REFERENCE_MUST_BE_INSTANCE_SCOPED
format2:	reference_must_be_instance_scoped
evaluation policy	immediate
description:	Classifier scoped References are not meaningful in the current M1 level computational model.

```
context Reference
inv: self.scope = #instance_level
```

[C-23]	ChangeableReferenceMustHaveChangeableEnd
format1:	CHANGEABLE_REFERENCE_MUST_HAVE_CHANGEABLE_END
format2:	changeable_reference_must_have_changeable_end
evaluation policy	deferred
description:	A Reference can be changeable only if the referenced AssociationEnd is also changeable.

```
context Reference
inv: self.isChangeable = self.referencedEnd.isChangeable
```


© ISO/IEC 2005 - All rights reserved

[C-24]	ReferenceTypeMustMatchEndType
format1:	REFERENCE_TYPE_MUST_MATCH_END_TYPE
format2:	reference_type_must_match_end_type
evaluation policy	deferred
description:	The type attribute of a Reference and its referenced AssociationEnd must be the same.

```
context Reference
inv: self.type = self.referencedEnd.type
```

[C-25]	ReferencedEndMustBeNavigable
format1:	REFERENCED_END_MUST_BE_NAVIGABLE
format2:	referenced_end_must_be_navigable
evaluation policy	deferred
description:	A Reference is only allowed for a navigable AssociationEnd.

```
context Reference
inv: self.referencedEnd.isNavigable
```

[C-26]	ContainerMustMatchExposedType
format1:	CONTAINER_MUST_MATCH_EXPOSED_TYPE
format2:	container_must_match_exposed_type
evaluation policy	deferred
description:	The containing Class for a Reference must be equal to or a subtype of the type of the Reference's exposed AssociationEnd.

```
context Reference
inv:
   self.container.allSupertypes() -> including(self) ->
          includes(self.referencedEnd.otherEnd.type)
```

© ISO/IEC 2005 - All rights reserved

[C-27]	ReferencedEndMustBeVisible
format1:	REFERENCED_END_MUST_BE_VISIBLE
format2:	referenced_end_must_be_visible
evaluation policy	deferred
description:	The referenced AssociationEnd for a Reference must be visible from the Reference.

```
context Reference
inv: self.isVisible(self.referencedEnd)
```

[C-28]	OperationContainmentRules
format1:	OPERATION_CONTAINMENT_RULES
format2:	operation_containment_rules
evaluation policy	immediate
description:	An Operation may only contain Parameters, Constraints, and Tags.

```
context Operation
inv:
   Set{Parameter, Constraint, Tag} ->
          includesAll(self.contentTypes())
```

[C-29]	OperationsHaveAtMostOneReturn
format1:	OPERATIONS_HAVE_AT_MOST_ONE_RETURN
format2:	operations_have_at_most_one_return
evaluation policy	immediate
description:	An Operation may have at most one Parameter whose direction is “return.”

```
context Operation
inv:
   self.contents ->
          select(c | c.oclIsTypeOf(Parameter)) ->
                 select(p : Parameter | p.direction = #return_dir) ->
                        size < 2
```


© ISO/IEC 2005 - All rights reserved

[C-30]	OperationExceptionsMustBeVisible
format1:	OPERATION_EXCEPTIONS_MUST_BE_VISIBLE
format2:	operation_exceptions_must_be_visible
evaluation policy	deferred
description:	The Exceptions raised by an Operation must be visible to the Operation.

```
context Operation
inv: self.exceptions -> forAll(e | self.isVisible(e))
```

[C-31]	ExceptionContainmentRules
format1:	EXCEPTION_CONTAINMENT_RULES
format2:	exception_containment_rules
evaluation policy	immediate
description:	An Exception may only contain Parameters and Tags.

```
context Exception
inv: Set{Parameter, Tag}) -> includesAll(self.contentTypes())
```

[C-32]	ExceptionsHaveOnlyOutParameters
format1:	EXCEPTIONS_HAVE_ONLY_OUT_PARAMETERS
format2:	exceptions_have_only_out_parameters
evaluation policy	immediate
description:	An Exception’s Parameters must all have the direction “out.”

```
context Exception
inv:
   self.contents ->
          select(c | c.oclIsTypeOf(Parameter)) ->
                 forAll(p : Parameter | p.direction = #out_dir)
```

© ISO/IEC 2005 - All rights reserved

[C-33]	AssociationContainmentRules
format1:	ASSOCIATIONS_CONTAINMENT_RULES
format2:	associations_containment_rules
evaluation policy	immediate
description:	An Association may only contain AssociationEnds, Constraints, and Tags.

```
context Association
inv:
   Set{AssociationEnd, Constraint, Tag} ->
          includesAll(self.contentTypes())
```

[C-34]	AssociationsHaveNoSupertypes
format1:	ASSOCIATIONS_HAVE_NO_SUPERTYPES
format2:	associations_have_no_supertypes
evaluation policy	immediate
description:	Inheritance / generalization is not applicable to Associations.

```
context Association
inv: self.supertypes -> isEmpty
```

[C-35]	AssociationMustBeRootAndLeaf
format1:	ASSOCIATIONS_MUST_BE_ROOT_AND_LEAF
format2:	associations_must_be_root_and_leaf
evaluation policy	immediate
description:	The values for “isLeaf” and “isRoot” on an Association must be true.

```
context Association
inv: self.isRoot and self.isLeaf
```


© ISO/IEC 2005 - All rights reserved

[C-36]	AssociationsCannotBeAbstract
format1:	ASSOCIATIONS_CANNOT_BE_ABSTRACT
format2:	associations_cannot_be_abstract
evaluation policy	immediate
description:	An Association cannot be abstract.

```
context Association
inv: not self.isAbstract
```

[C-37]	AssociationsMustBePublic
format1:	ASSOCIATIONS_MUST_BE_PUBLIC
format2:	associations_must_be_public
evaluation policy	immediate
description:	Associations must have visibility of “public.”

```
context Association
inv: self.visibility = #public_vis
```

[C-38]	AssociationsMustNotUnary
format1:	ASSOCIATIONS_MUST_NOT_BE_UNARY
format2:	associations_must_be_not_unary
evaluation policy	immediate
description:	An Association must not be unary; that is, it must have at least two AssociationEnds.

```
context Association
inv: self.contents ->
   select(c | c.ocllstTypeOf(AssociationEnd)) -> size = 2
```

[C-39]	EndTypeMustBeClass
format1:	END_TYPE_MUST_BE_CLASS

© ISO/IEC 2005 - All rights reserved

format2:	end_type_must_be_class
evaluation policy	immediate
description:	The type of an AssociationEnd must be Class.

```
context AssociationEnd
inv: self.type.oclIsTypeOf(Class)
```

[C-40]	EndsMustBeUnique
format1:	ENDS_MUST_BE_UNIQUE
format2:	ends_must_be_unique
evaluation policy	immediate
description:	The “isUnique” flag in an AssociationEnd’s multiplicity must be true.

```
context AssociationEnd
inv:
    (self.multiplicity.upper > 1 or
     self.multiplicity.upper = UNBOUNDED) implies
            self.multiplicity.isUnique
```

[C-41]	CannotHaveTwoOrderedEnds
format1:	CANNOT_HAVE_TWO_ORDERED_ENDS
format2:	cannot_have_two_ordered_ends
evaluation policy	deferred
description:	An AssociationEnd cannot have two AssociationEnds marked as “ordered.”

```
context AssociationEnd
inv:
    self.multiplicity.isOrdered implies
           not self.otherEnd.multiplicity.isOrdered
```

[C-42]	CannotHaveTwoAggregateEnds
format1:	CANNOT_HAVE_TWO_AGGREGATE_ENDS

© ISO/IEC 2005 - All rights reserved

format2:	cannot_have_two_aggregate_ends
evaluation policy	deferred
description:	An AssociationEnd cannot have an aggregation semantic specified for both AssociationEnds.

```
context AssociationEnd
inv:
   self.aggregation <> #none implies self.otherEnd = #none
```

[C-43]	PackageContainmentRules
format1:	PACKAGE_CONTAINMENT_RULES
format2:	package_containment_rules
evaluation policy	immediate
description:	A Package may only contain Packages, Classes, DataTypes, Associations, Exceptions, Constants, Constraints, Imports, and Tags.

```
context Package
inv:
   Set{Package, Class, DataType, Association, Exception,
          Constant, Constraint, Import, Tag}) ->
                 includesAll(self.contentTypes)
```

[C-44]	PackagesCannotBeAbstract
format1:	PACKAGES_CANNOT_BE_ABSTRACT
format2:	packages_cannot_be_abstract
evaluation policy	immediate
description:	Packages cannot be declared as abstract.

```
context Package
inv: not self.isAbstract
```

[C-45]	ImportedNamespaceMustBeVisible
format1:	IMPORTED_NAMESPACE_MUST_BE_VISIBLE

© ISO/IEC 2005 - All rights reserved

format2:	imported_namespace_must_be_visible
evaluation policy	deferred
description:	The Namespace imported by an Import must be visible to the Import's containing Package.

```
context Import
inv: self.container.isVisible(self.importedNamespace)
```

[C-46]	CanOnlyImportPackagesAndClasses
format1:	CAN_ONLY_IMPORT_PACKAGES_AND_CLASSES
format2:	can_only_import_packages_and_classes
evaluation policy	immediate
description:	It is only legal for a Package to import or cluster Packages or Classes.

```
context Import
inv:
   self.imported.oclIsTypeOf(Class) or
   self.imported.oclIsTypeOf(Package)
```

[C-47]	CannotImportSelf
format1:	CANNOT_IMPORT_SELF
format2:	cannot_import_self
evaluation policy	deferred
description:	Packages cannot import or cluster themselves.

```
context Import
inv: self.container <> self.imported
```

[C-48]	CannotImportNestedComponents
format1:	CANNOT_IMPORT_NESTED_COMPONENTS

© ISO/IEC 2005 - All rights reserved

format2:	cannot_import_nested_components
evaluation policy	deferred
description:	Packages cannot import or cluster Packages or Classes that they contain.

```
context Import
inv: not self.container.allContents() -> includes(self.imported)
```

[C-49]	NestedPackagesCannotImport
format1:	NESTED_PACKAGES_CANNOT_IMPORT
format2:	nested_packages_cannot_import
evaluation policy	deferred
description:	Nested Packages cannot import or cluster other Packages or Classes.

```
context Import
inv:
   self.container -> notEmpty implies
          self.container -> asSequence -> first -> container -> isEmpty
```

[C-50]	CannotConstrainThisElement
format1:	CANNOT_CONSTRAIN_THIS_ELEMENT
format2:	cannot_constrain_this_element
evaluation policy	immediate
description:	Constraints, Tags, Imports, and Constants cannot be constrained.

```
context Constraint
inv:
   self.constrainedElements ->
          forAll(c |
                    not Set{Constraint, Tag, Imports, Constant} ->
                              includes(c.oclType())
```

© ISO/IEC 2005 - All rights reserved

[C-51]	ConstraintsLimitedToContainer
format1:	CONSTRAINTS_LIMITED_TO_CONTAINER
format2:	constraints_limited_to_container
evaluation policy	deferred
description:	A Constraint can only constrain ModelElements that are defined by or inherited by its immediate container.

```
context Constraint
inv:
   self.constrainedElements ->
          forAll(c | self.container.extendedNamespace() ->
                           includes(c))
```

[C-52]	ConstantsValueMustMatchType
format1:	CONSTANTS_VALUE_MUST_MATCH_TYPE
format2:	constants_value_must_match_type
evaluation policy	deferred
description:	The type of a Constant and its value must be compatible.

```
context Constant
inv: ...
```

[C-53]	ConstantsTypeMustBePrimitive
format1:	CONSTANTS_TYPE_MUST_BE_PRIMITIVE
format2:	constants_type_must_be_primitive
evaluation policy	immediate
description:	The type of a Constant must be a PrimitiveType.

```
context Constant
inv:
   self.type.oclIsOfType(PrimitiveType)
```


© ISO/IEC 2005 - All rights reserved

[C-54]	LowerCannotBeNegativeOrUnbounded
format1:	LOWER_CANNOT_BE_NEGATIVE_OR_UNBOUNDED
format2:	lower_cannot_be_negative_or_unbounded
evaluation policy	immediate
description:	The “lower” bound of a MultiplicityType to be “Unbounded.”

```
context MultiplicityType
inv: self.lower >= 0 and self.lower <> Unbounded
```

[C-55]	LowerCannotExceedUpper
format1:	LOWER_CANNOT_EXCEED_UPPER
format2:	lower_cannot_exceed_upper
evaluation policy	immediate
description:	The “lower” bound of a MultiplicityType cannot exceed the “upper” bound.

```
context MultiplicityType
inv: self.lower <= self.upper or self.upper = Unbounded
```

[C-56]	UpperMustBePositive
format1:	UPPER_MUST_BE_POSITIVE
format2:	upper_must_be_positive
evaluation policy	immediate
description:	The “upper” bound of a MultiplicityType cannot be less than 1.

```
context MultiplicityType
inv: self.upper >= 1 or self.upper = Unbounded
```

© ISO/IEC 2005 - All rights reserved

[C-57]	MustBeUnorderedNonunique
format1:	MUST_BE_UNORDERED_NONUNIQUE
format2:	must_be_unordered_nonunique
evaluation policy	immediate
description:	If a MultiplicityType specifies bounds of [0..1] or [1..1]), the “is_ordered” and “is_unique” values must be false.

```
context MultiplicityType
inv:
    self.upper = 1 implies
           (not self.isOrdered and not self.isUnique)
```

[C-58]	StructuredFieldContainmentRules
format1:	STRUCTURE_FIELD_CONTAINMENT_RULES
format2:	structure_field_containment_rules
evaluation policy	immediate
description:	A StructureField contains Constraints and Tags.

```
context StructureField
inv:
    Set{   Constraint, Tag}) -> includesAll(self.contentTypes)
```

[C-59]	MustHaveFields
format1:	MUST_HAVE_FIELDS
format2:	must_have_fields
evaluation policy	deferred
description:	A StructureType must contain at least one StructureField.

```
context StructureType
inv:
    self.contents -> exists(c | c.oclIsOfType(StructureField))
```


© ISO/IEC 2005 - All rights reserved

7.9.5 Semantic specifications for some Operations, derived Attributes and Derived Associations

[S-1]	allSupertypes
kind:	query Operation
description:	The value is the closure of the ‘Generalizes’ Association from the perspective of a subtype. Note that the sequence of all supertypes has a well defined order.

```
context GeneralizableElement::allSupertypes() :
                        Sequence(GeneralizableElement)
post: result = self.allSupertypes2(Set{})
```

[S-2]	otherEnd
kind:	query Operation
description:	The value is the other AssociationEnd for this Association.

```
context AssociationEnd::otherEnd() : AssociationEnd
post: result = self.container.contents ->
   select(c | c.oclIsKindOf(AssociationEnd) and c <> self)
```

[S-3]	isVisible
kind:	query Operation
description:	Determines whether or not “otherElement” is visible for the definition of this element. (Note: As an interim measure, the OCL states that everything is visible!)

```
context ModelElement::isVisible(
                        otherElement : ModelElement): boolean
post: result = true
```

[S-4]	findRequiredElements
kind:	query Operation
description:	Selects a subset of a ModelElement’s immediate or recursive dependents.

© ISO/IEC 2005 - All rights reserved

```
context ModelElement::isRequiredBecause(
                        kinds : Sequence(DependencyKind),
                        recursive : boolean) : Sequence(ModelElement)
post: result =
    if kinds -> includes("all")
    then
           self.findRequiredElements(
                  Set{"constraint", "container", "constrained elements",
                        "specialization", "import", "contents", "signature",
                        "tagged elements", "type definition",
                        "referenced ends"})
    else
           if recursive
           then
                  self.recursiveFindDeps(kinds, Set{self})
           else
                  kinds -> collect(k : self.findDepsOfKind(k)) -> asSet()
           endif
    endif
```

[S-5]	findRequiredBecause
kind:	query Operation
description:	Returns the DependencyKind that describes the dependency between this element and "other."

```
context ModelElement::findRequiredElements(
                        other : ModelElement,
                        reason : out DependencyKind) : boolean
post: -- NB, if there is more than one dependency between self
                   -- and 'other', the selection of the 'reason' is defined
                   -- to be non-deterministic ... not deterministic as a
                   -- left to right evaluation of the OCL implies.
    reason = (
           if self -> isDepOfKind("constraint", other)
           then
                  "constraint"
           else
                  if self -> isDepOfKind("container", other)
                  then
                        "container"
                  else
                        if self -> isDepOfKind("constrained elements", other)
                        then
                              "constrained elements"
                        else
                              if self -> isDepOfKind("specialization", other)
                              then
                               "specialization"
```


© ISO/IEC 2005 - All rights reserved

```
                                else
                                 if self -> isDepOfKind("import", other)
                                 then
                                     "import"
                                 else
                                     if self -> isDepOfKind("contents", other)
                                     then
                                        "contents"
                                     else
                                        if self -> isDepOfKind("signature", other)
                                        then
                                         "signature"
                                        else
                                         if self -> isDepOfKind("tagged elements", other)
                                         then
                                             "tagged elements"
                                         else
                                             if self -> isDepOfKind("type definition", other)
                                             then
                                               "type definition"
                                             else
                                               if self -> isDepOfKind("referenced ends", other)
                                               then
                                                 "referenced ends"
                                               else
                                                 if self -> dependsOn() -> notEmpty()
                                                 then
                                                     "indirect"
                                                 else
                                                     ""
                                                 endif
                                               endif
                                             endif
                                         endif
                                        endif
                                     endif
                                 endif
                                endif
                        endif
                endif
        endif) and
result = (reason <> "")
```

[S-6]	lookupElement
kind:	query Operation
description:	Returns the ModelElement in the Namespace whose name is equal to “name,” or raises an exception.

© ISO/IEC 2005 - All rights reserved

```
context Namespace::lookupElement(name : string) : ModelElement
post: result =
    let elems = self.contents -> select(m | m.name = name) in
    if elems -> size = 0
    then
           -- Raise exception NameNotFound
    else
           elems -> first  -- should only be one
    endif
```

[S-7]	resolveQualifiedName
kind:	query Operation
description:	Returns the ModelElement that “qualifiedName” resolves to or raises an exception.

```
context Namespace::resolveQualifiedName(
                      qualifiedName : Sequence(string)) : ModelElement
pre: qualifiedName -> size >= 1
post: result =
    let elems = self.contents ->
                 select(m | m.name = qualifiedName -> first) in
    if elems -> size = 0
    then
           -- Raise exception NameNotResolved ...
    else
           if qualifiedName -> size = 1
           then
                  elems -> first    -- there should only be one
           else
                  if not elems -> first -> oclIsOfKind(Namespace)
                  then
                         -- Raise exception NameNotResolved ...
                  else
                         let rest = qualifiedName ->
                                subSequence(2, qualifiedName -> size) in
                         elems -> first -> resolveQualifiedName(rest)
                  endif
           endif
    endif
```

[S-8]	NameIsValid
kind:	query Operation
description:	Returns true if “proposedName” is a valid name that could be used for a new containedElement of this Namespace.

© ISO/IEC 2005 - All rights reserved

```
context Namespace::nameIsValid(
                        proposedName : string) : boolean
post: result =
    self.extendedNamespace ->
           forAll(e | not e.name = proposedName)
```

[S-9]	findElementsByType
kind:	query Operation
description:	Returns a subset of the contained elements. If “includeSubtypes” is false, the result consists of instances of “ofType.” If it is true, instances of subClasses are included.

```
context Namespace::findElementsByType(
                        ofType : Class,
                        includeSubtypes : boolean) : Sequence(ModelElement)
post: result =
    if includeSubtypes
    then
           self.contents -> select(m | m.oclIsOfKind(ofType))
    else
           self.contents -> select(m | m.oclIsOfType(ofType))
    endif
```

[S-10]	lookupelementExtended
kind:	query Operation
description:	Returns the ModelElement whose name is equal to “name” in the extended namespace of this GeneralizableElement, or raises an exception.

```
context Namespace::lookupElementExtended(
                        name : string) : ModelElement
post: result =
    let elems = self -> extendedNamespace ->
                  select(m | m.name = name) in
    if elems -> size = 0
    then
           -- Raise exception NameNotFound
    else
           elems -> first  -- should only be one
    endif
```

© ISO/IEC 2005 - All rights reserved

[S-11]	findElementsByTypeExtended
kind:	query Operation
description:	Returns a subset of the contained, inherited, or imported elements. If “includeSubtypes” is false, the result consists of instances of “ofType.” If it is true, instances of subClasses are included.

```
context GeneralizeableElement::findElementsByTypeExtended(
                        ofType : Class,
                        includeSubtypes : boolean) : Sequence(ModelElement)
post: result =
    if includeSubtypes
    then
           self.extendedNamespace -> select(m | m.oclIsOfKind(ofType))
    else
           self.extendedNamespace -> select(m | m.oclIsOfType(ofType))
    endif
```

[S-12]	qualifiedName
kind:	readonly derived Attribute
description:	The qualified name gives the sequence of names of the containers of this ModelElement starting with the outermost.

```
context ModelElement::qualifiedName() : Sequence(ModelElement)
post: result =
    if self.container -> notEmpty
    then
           self.container.qualifiedName() -> append(self.name)
    else
           self.name
    endif
```

[S-13]	Exposes
kind:	derived Association
description:	This association relates a Reference to the exposed AssociationEnd of an Association that corresponds to its referencedEnd.

© ISO/IEC 2005 - All rights reserved

```
context Reference
inv: AssociationEnd.allInstances ->
   forAll(
          a |
          self.references = a implies self.exposes = a.otherEnd and
          not self.references = a implies self.exposes <> a.otherEnd)
```

[S-14]	DependsOn
kind:	derived Association
description:	This association relates a ModelElement to the other ModelElements whose definition it depends on.

```
context ModelElement
inv: self.findRequiredElements("all", true)
```

7.9.6 OCL Helper functions

[O-1]	allSupertypes2
description:	Helper function for the allSupertypes operation.

```
context GeneralizableElement::allSupertypes2(
                visited : Set(GeneralizableElement)) :
                       Sequence(GeneralizableElement)
post: result =
   if (visited -> includes(self))
   then
          Sequence{}
   else
          let mySupers : Sequence(GeneralizableElement) =
                self.supertypes ->
                       collect(s |
                              s.allSupertypes2(visited ->
                                            including(self))) in
          mySupers ->
                iterate(s2 : GeneralizableElement;
                       a : Sequence(GeneralizableElement) = Sequence{} |
                       if a -> includes(s2)
                       then
                              a
                       else
                              a -> append(s2)
                       endif)
```

© ISO/IEC 2005 - All rights reserved

[O-2]	extendedNamespace
description:	The extendedNamespace of a Namespace is its contents, the contents of its supertypes, and any Namespaces that it imports.

```
context Namespace::extendedNamespace() : Set(ModelElement)
post: result =
    self.contents
```

```
context GeneralizableElement::extendedNamespace : Set(ModelElement)
post: result =
    self.contents ->
           union(self.allSupertypes() -> collect(s | s.contents))
```

```
context Package::extendedNamespace : Set(ModelElement)
post: result =
    let ens = self.contents ->
           union(self.allSupertypes() -> collect(s | s.contents)) in
    let imports = ens -> select(e | e.oclKindOf(Import)) ->
           collect(i : Import | i.imported) in
    ens -> union(imports)
```

[O-3]	contentTypes
description:	The set of OCL types for a Namespace’s contents.

```
context Namespace::contentTypes() : Set(OCLType)
post: result = self.contents -> collect(m | m.oclType()) -> asSet
```

[O-4]	format1Name
description:	The simple name of the element converted to words and reassembled according to the “format1” rules; see “IDL Identifier Format 1” on page 5-39.

```
context ModelElement::format1Name() : string
post: result = ...
```


© ISO/IEC 2005 - All rights reserved

[O-5] repositoryId

description: The qualified name of the element converted into a standard CORBA repositoryId string.

```
context ModelElement::repositoryId() : string
post: result = ...
```

[O-6] recursiveFindDeps

description: The set of ModelElements that recursively depend on this one.

```
context ModelElement::recursiveFindDeps(
                       kinds : Sequence(string),
                       seen : Set(ModelElement)) : Set(ModelElement)
post: result =
   let seen2 = seen ->
          collect(m | kinds ->
                 collect(k | m.findDepsOfKind(k)) -> asSet) in
   if seen2 = seen
   then
          seen
   else
          self.recursiveFindDeps(kinds, seen2)
   endif
```

[O-7] isDepOfKind

description: Returns true if this element depends on ‘other’ with a dependency of ‘kind.’

```
context ModelElement::isDepOfKind(
                       kind : string,
                       other : ModelElement) : boolean
post: result = self -> findDepsOfKind(kind) -> includes(other)
```

[O-8] finalDepsOfKind

description: The set of Modelelements that this one Depends on with “kind” dependency.

© ISO/IEC 2005 - All rights reserved

```
context ModelElement::findDepsOfKind(
                                        kind : string) : Sequence(ModelElement)
post: result =
    if kind = "constraint"
    then self.constraints()
    else if kind = "container"
    then self.container()
    else if kind = "constrained elements" and
                                self -> isOclTypeOf(Constraint)
    then self -> oclAsType(Constraint) -> constrainedElements()
    else if kind = "specialization" and
                                self -> isOclKindOf(GeneralizableElement)
    then self -> oclAsType(GeneralizableElement) -> supertypes()
    else if kind = "import" and self -> isOclType(Import)
    then self -> oclAsType(Import) -> importedNamespace()
    else if kind = "contents" and self -> isOclKindOf(Namespace)
    then self -> oclAsType(Namespace) -> contents()
    else if kind = "signature" and self -> isOclTypeOf(Operation)
    then self -> oclAsType(Operation) -> exceptions()
    else if kind = "tagged elements" and
                                self -> isOclTypeOf(Tag)
    then self -> oclAsType(Tag) -> elements()
    else if kind = "type definition" and
                                self -> isOclKindOf(TypedElement)
    then self -> oclAsType(TypedElement) -> type()
    else if kind = "referenced ends" and
                                self -> isOclKindOf(Reference)
    then
           let ref = self -> asOclType(Reference) in
                        ref -> referencedEnd() -> union(ref -> exposedEnd())
    else
           Set{}
    endif endif endif endif endif endif endif endif endif endif
```

7.10 The PrimitiveTypes Package

The MOF defines a package called PrimitiveTypes that contains the MOF's standard collection of primitive data types. These types are available for use in all MOF meta-models. Some of them (Boolean, Integer, and String) are used in the specification of the MOF Model as an instance of itself.

The PrimitiveTypes Package is a MOF meta-model whose name is "PrimitiveTypes." The package contains the PrimitiveType instances defined below, and no other instances. These instances denote the MOF's standard technology-neutral primitive data types.

NOTE: Technology mappings shall recognize standard PrimitiveType instances based on their qualified names. Multiple PrimitiveType instances with the required qualified name shall be deemed to mean the same thing.

The sub clauses below define the 6 standard technology neutral primitive data types and their value domains. They also define the standard syntax used in the "value' attribute of a Constant instance; see "Constant" on page 82.

© ISO/IEC 2005 - All rights reserved

7.10.1 Boolean

This primitive data type is used to represent truth values.

value domain	The set of the boolean values ‘true’ and ‘false.’
constant value syntax	‘TRUE,’ ‘FALSE’ (case sensitive)

7.10.2 Integer

This primitive data type is the set of signed (two’s complement) 32 bit integers

value domain	The subset of the integers in the range -2^{31} to $+2^{31} - 1$
constant value syntax	CORBA IDL integer literal syntax with an optional leading ‘-’ character.

7.10.3 Long

This primitive data type is the set of signed (two’s complement) 64 bit integers

value domain	The subset of the integers in the range -2^{63} to $+2^{63} - 1$
constant value syntax	CORBA IDL integer literal syntax with an optional leading ‘-’ character.

7.10.4 Float

This primitive data type is the set of IEEE single precision floating point numbers (see ANSI/IEEE Standard 754-1985.

value domain	The subset of the rational numbers that correspond to the values representable as IEEE single precision floating point numbers.
constant value syntax	CORBA IDL floating point literal syntax with an optional leading ‘-’ character.

7.10.5 Double

This primitive data type is the set of IEEE double precision floating point numbers; see ANSI/IEEE Standard 754-1985.

value domain	The subset of the rational numbers that correspond to the values representable as IEEE double precision floating point numbers.
constant value syntax	CORBA IDL floating point literal syntax with an optional leading ‘-’ character.

7.10.6 String

This primitive data type is the set of all character strings that consist of 16 bit characters (excluding NUL).

value domain	The infinite set of all finite sequences of 16 bit characters (excluding the zero character value).
constant value syntax	A sequence of 16-bit characters. (Note: a Constant’s ‘value’ string for a String has no surrounding quotes and contains no character escape sequences.)

© ISO/IEC 2005 - All rights reserved

7.10.7 IDL for the PrimitiveTypes Package

The IDL for the “PrimitiveTypes” package is given below. It is generated by applying the MOF to IDL mapping to the PrimitiveTypes Package. This IDL would typically be “#included” by IDL for meta-models (e.g., the MOF Model IDL).

NOTE: This is *not* the IDL for the MOF objects that represent the primitive types.

```
#pragma prefix "org.omg.mof"
module PrimitiveTypes {
    // Collection types for the six standard MOF primitive data types
    typedef sequence < boolean > BooleanBag;
    typedef sequence < boolean > BooleanSet;
    typedef sequence < boolean > BooleanList;
    typedef sequence < boolean > BooleanUList;
    typedef sequence < long > LongBag;
    typedef sequence < long > LongSet;
    typedef sequence < long > LongList;
    typedef sequence < long > LongUList;
    typedef sequence < long long > LongLongBag;
    typedef sequence < long long > LongLongSet;
    typedef sequence < long long > LongLongList;
    typedef sequence < long long > LongLongUList;
    typedef sequence < float > FloatBag;
    typedef sequence < float > FloatSet;
    typedef sequence < float > FloatList;
    typedef sequence < float > FloatUList;
    typedef sequence < double > DoubleBag;
    typedef sequence < double > DoubleSet;
    typedef sequence < double > DoubleList;
    typedef sequence < double > DoubleUList;
    typedef sequence < wstring > WStringBag;
    typedef sequence < wstring > WStringSet;
    typedef sequence < wstring > WStringList;
    typedef sequence < wstring > WStringUList;

    // This interface would be inheritted by IDL for a Package declared as a sub-package of PrimitiveTypes
    interface PrimitiveTypesPackage : Reflective::RefPackage { };

    // This interface is present because we can’t declare PrimitiveTypes as an abstract Package.
    // There is no point instantiating it.
    interface PrimitiveTypesPackageFactory {
        PrimitiveTypesPackage create_primitive_types_package()
            raises (Reflective::MofError);
    };
};
```

7.11 Standard Technology Neutral Tags

This sub clause defines the standard Tags that apply to all meta-models, irrespective of the technology mappings used. Other Tags may be attached to the elements of a meta-model, but the meaning of these Tags is not specified here. Tags relevant to the IDL mapping are defined in 9.6, “Standard Tags for the IDL Mapping,” on page 194.

Table 7.5 shows the conventions used to describe the standard Tags and their properties.

© ISO/IEC 2005 - All rights reserved

Table 7.5 Notation for Describing Standard Tags

tag id:	A string that denotes the semantic category for the tag.
attaches to:	Gives the kind(s) of Model::ModelElement that this category of tag can be meaningfully attached to.
values:	Gives the number and types of the tag's values (i.e., parameters), if any. Parameters are expressed as an ordered sequence of Strings.
meaning:	Describes the meaning of the tag in this context.
idl generation:	Defines the tag's impact on the generated IDL.
restrictions:	Tag usage restrictions - for example: "at most one tag of this kind per element," or "tag must be contained by the meta-model."

NOTE: There are no standard technology neutral Tags at this time.

© ISO/IEC 2005 - All rights reserved

© ISO/IEC 2005 - All rights reserved

8 The MOF Abstract Mapping

8.1 Overview

This Clause describes the MOF's M1-level information model, and the common principles underlying mapping specific M1-level computational models. Since it is intended to be independent of any mapping to implementation technology, the material is rather abstract.

8.2 MOF Values

A MOF meta-model is an abstract language for defining the types of meta-data. The M2-level constructs used in a meta-model map onto M1-level representations as MOF values. The types of these M1-level values can be defined using either M2-level Classes or M2-level DataTypes.

An M2-level Class defines an M1-level Instance type with the following properties:

- Instance typed objects have full object identity; that is, it is always possible to reliably distinguish one instance (object) from another. Object identity is intrinsic and permanent, and is not dependent on other properties such as attribute values.
- Instance typed objects can be linked via an Association.
- Null is a valid instance of an M2-level Class, though there are limitations on its use.

By contrast, an M2-level DataType defines a type with the following properties:

- Data typed values do *not* have full object identity; see below.
- Data typed values *cannot* be linked via an Association.
- Null is *not* a valid instance of an M2-level DataType.

8.3 Semantics of Data Types

Data types in MOF meta-models fall into two groups:

- MOF standard technology neutral data types; that is, the 6 standard primitive data types (Boolean, Integer, Long, Float, Double, and String) and those produced using the constructors EnumerationType, CollectionType, StructureType, and AliasType.
- Native types (i.e., other data types. Native data types are not technology neutral).

Each data type in a MOF meta-model denotes a finite or infinite set of values. In the case of the standard technology neutral MOF data types and constructors, these sets are as follows:

- The type Boolean consists of two values that are conventionally called "true" and "false."
- The type Integer is the subrange of integers from -2^{31} to $+2^{31} - 1$.
- The type Long is the subrange of integers from -2^{63} to $+2^{63} - 1$.
- The type Float is the the subrange of the rational numbers whose values are AIEEE 764 single precision floating point numbers.

© ISO/IEC 2005 - All rights reserved

- The type Double is the subrange of the rational numbers whose values are IEEE 764 double precision floating point numbers.
- The type String is the set of all possible finite length sequences of the UTF16 characters (excluding the NUL or zero character).
- An enumeration type is the set of the EnumerationType labels or enumerators.
- A structure type is the set of all tuples whose field values are members of the types of the StructureType's respective StructureFields.
- A collection type is the set of all collections of the CollectionType's type, limited according to the CollectionType's multiplicity.
- An alias type is a subset of the AliasType's type.

NOTE: Every MOF technology mapping must define a mapping from the standard MOF technology neutral data types to a concrete type such that every value in the value domain has a distinct representation using the concrete type. A technology mapping that does not support all of the data types, or that maps them in a way that loses information is *not valid*.

It is not possible to define the sets of values that comprise native types in a technology independent way.

8.4 Semantics of Equality for MOF Values

Much of the detail of the MOF computational model depends on a notion of equality of values. For example, the precise formulation of the "no duplicates" rule for link sets depends on a definition for what it means for object type instances to be equal.

Equality of MOF Values is defined as follows:

1. Instances of Classes are equal if and only if they have the same identity. Equality does *not* take into account the values of Attributes for the instances or the links involving the instances. The null Class instance is only equal to itself.
2. Values of all MOF data types are incomparable if they do not have the same type.
3. Values of MOF standard primitive data types are equal if and only if they denote the same element of the set that defines the primitive type.
4. Values of MOF enumeration data types are equal if and only if they denote the same enumerator.
5. Values of MOF collection data types are equal if and only if they have the same number of elements. For a 'bag' or 'set,' there must be a 1-for-1 correspondence between the collection elements such that the corresponding elements be equal according to these rules. For a 'list' or 'ordered set,' the elements at each position in the collection must be equal according to these rules.
6. Values of MOF structure types are equal if and only if they have the same type and the corresponding structure fields are equal according to these rules.
7. Values of MOF alias types are equal if and only if they have the same type and their values compare as equal in the context of the alias type's base type.
8. The meaning of equality for native type values is not specified here. In some mappings, some native values may be incomparable.

© ISO/IEC 2005 - All rights reserved

NOTE: The meaning of equality for values of incomparable types is not defined. However, this is not an issue since the semantics of MOF metadata do not depend on being able to compare such values.

8.5 Semantics of Class Instances

One and only one M1-level Instance is a value whose type is described by an M2-level Class. An Instance has the following properties in the MOF computational model:

- It has object identity. This has different implications depending on the mapping, but in general it means that many conceptually distinct Instance values can exist whose component values are the same.
- It has a definite lifetime. An Instance value is created in response to particular events in the computational model, and continues to exist until it is deleted in response to other events.
- It is created in a computational context known as a Class extent, and remains in that extent for its lifetime; see 8.8, “Extents,” on page 145.
- It can have attribute values defined using M2-level Attributes; see 8.6, “Semantics of Attributes,” on page 141.
- It can be linked to other Instances; 8.9, “Semantics of Associations,” on page 149.

Not all M2-level Classes can have corresponding M1-level Instances. In particular, Instances can never be created for Classes that have “isAbstract” set to true. In addition, if an M2-level Class has “isSingleton” set to true, only one Instance of the class can exist within an extent for the Class.

The null instance of an M2-level Class has a conceptual identity that is distinct from other (non-null) instances. Null conceptually exists forever in all Class extents, but it does not have attribute values and cannot be related to other Instances (or itself) by an Association link.

NOTE: While null is currently a valid Class instance, some technology mappings do not support it. Therefore it is inadvisable to rely on being able to use the null instance value in a technology neutral metamodel.

8.6 Semantics of Attributes

Attributes are one of two mechanisms provided by the MOF Model for defining relationships between values at the M1-level. An Attribute of an M2-level Class defines a relation between each M1-level Instance of the Class, and values of some other type. The Attribute specification consists of the following properties:

- the Attribute’s “name,”
- the Attribute’s “type” which may be expressed using a Class or DataType,
- a “scope” specification,
- a “visibility” specification,
- a “multiplicity” specification,
- an “isChangeable” flag,
- an “isDerived” flag, and
- an (implicit) aggregation specification.

© ISO/IEC 2005 - All rights reserved

Many aspects of the M1-level computational semantics of Attributes depend on the mapping used. The following sub clauses describe those aspects of the semantics that are mapping independent.

8.6.1 Attribute name and type

The “name” and “type” of an Attribute define the basic signature of a notional binary relationship between a Class instance and an attribute value or values. For example, an Attribute declaration of the form:

```
class Class1 {
   attribute attr_1 AttrType;
};
```

defines a notional relation between an M1-level type corresponding to the Class1 and an M1-level type corresponding to the AttrType. The three main kinds of relation that can exist between a Class and an Attribute are illustrated below in Figure 8.1. The figure shows cases where an Attribute’s “multiplicity” bounds are “[1..1]” (single-valued), “[0..1]” (optional) and “[m..n]” (multi-valued) respectively. Each notional relation is distinguishable from others for that Class by the Attribute’s “name.”

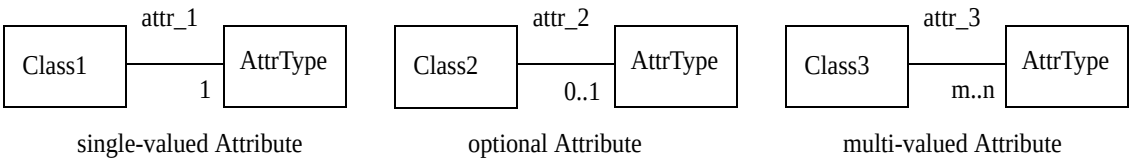

Figure 8.1 - Notional Class — Attribute Relations

An M2-level Attribute’s “type” can be either a Class or a DataType. In the former case, the Class — AttrType relation relates M1-level Instances corresponding to the two Classes. In the latter case, it relates M1-level Instances corresponding to the Class to M1-level Instances corresponding to the DataType.

In the following sub clauses, it is often necessary to talk about the type of the M1-level Instances on the AttrType end of a Class — AttrType relation. To make the text more readable, we will use the phrase “the Attribute’s M1-level *base type*” for this type rather than referring to it as “the M1-level type corresponding to the M2-level Attribute’s “type.” As we shall see, the phrase “the Attribute’s M1-level *type*” is best used for another purpose.

8.6.2 Multiplicity

The “multiplicity” property defines the cardinality, uniqueness, and orderedness of an Attribute as follows:

- The “lower” and “upper” fields set the bounds on the number of elements (i.e., cardinality) allowed in an Attribute value; that is, the “(collection of) AttrType” in Figure 8.1 and Figure 8.2 on page 143. Discussion of multiplicity usually needs to deal with three cases:
  - If the “lower” and “upper” are both 1, the Attribute is single-valued; that is, the “value” is a single instance belonging to the Attribute’s M1-level base type.
  - If the “lower” is 0 and “upper” is 1, the Attribute is optional; that is, the “value” is either an instance belonging to the Attribute’s M1-level base type, or nothing.

© ISO/IEC 2005 - All rights reserved

- Otherwise, the Attribute is multi-valued; that is, its “value” is a collection of instances belonging to the Attribute’s M1-level base type.
- The “isUnique” flag specifies whether or not a multi-valued Attribute is allowed to contain duplicates; that is, elements that are equal according to the definition in 8.4, “Semantics of Equality for MOF Values,” on page 140.
- The “isOrdered” flag specifies whether or not the order of the elements in a multi-valued Attribute are significant.

The “multiplicity” settings of an M2-level Attribute have considerable influence on the M1-level Attributes values. In particular, it determines whether the M1-level type of the Attribute is the M1-level base type, or a collection of that type. In addition, the “multiplicity” may also cause:

- runtime checks to ensure that a multi-valued Attribute’s cardinality lies within a given range,
- runtime checks to ensure that a multi-valued Attribute does not contain duplicate members, and
- mechanisms that allow the user to specify the order of the elements of a multi-valued Attribute.

The “multiplicity” may also have considerable impact on the APIs that a mapping provides for accessing and updating Attribute values.

It should be noted that when an M2-level Attribute has “isOrdered” set to true, the corresponding Class — AttrType relation has an associated partial ordering when viewed from the Class role.

8.6.3 Scope

The “scope” of an Attribute can be either “instance_level” or “classifier_level.” For an “instance_level” Attribute, independent relationships exist between instances of MyClass and instances of AttrType. For a “classifier_level” Attribute, a single instance of AttrType (or a collection of AttrType) is related to all instances of MyClass in the extent. This is illustrated in Figure 8.2.

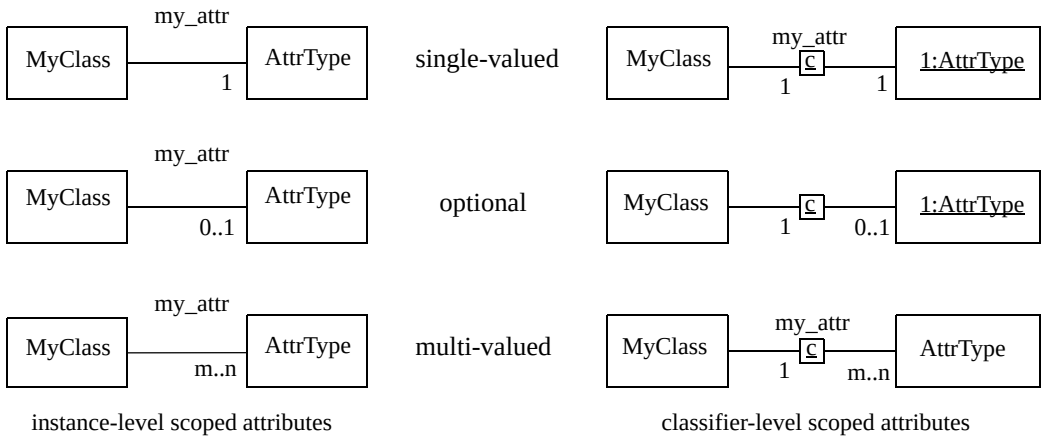

Figure 8.2 - Instance-level versus Classifier-level scoping

NOTE: For the classifier-level Attributes, the diagrams are intended to show that all MyClass instances are related to a single instance or collection of instances of AttrType.

© ISO/IEC 2005 - All rights reserved

8.6.4 Is_derived

The “isDerived” flag indicates whether the notional relationship between a Class instance and the Attribute type instances is stored or computed.

8.6.5 Aggregation

The possible aggregation semantics of an Attribute depend on its type:

- If an Attribute’s type is expressed as a DataType, it has “non-aggregate” semantics.
- If an Attribute’s type is expressed as a Class, it has “composite” semantics.

In cases where an Attribute has “composite” semantics, the Class instance that is the value of the Attribute is a component of the Class instance that contains the Attribute, not vice-versa.

NOTE: The above description reflects the fact that the Attribute model element does not have an “aggregation” attribute. A Class-valued Attribute with “non-aggregate” semantics is currently expressed by making the Attribute’s type a DataType, where the DataType’s “typeCode” is an object reference type that is linked to the Class via a TypeAlias.

8.6.6 Visibility and is_changeable

The “visibility” property of an Attribute determines whether or not any operations for the notional relation should be present. Similarly, the “isChangeable” property determines whether update operations are present. The presence or absence of these operations do not alter the semantics of the Attribute.

8.7 Package Composition

This sub clause summarizes the meta-model composition mechanisms supported by the MOF Model and discusses their impact on M1-level semantics.

8.7.1 Package Nesting

Package nesting is the simplest of the MOF’s Package composition mechanisms. At the M2-level, Package nesting is expressed by making the outer Package the “container” of the nested Package. The definition of the Contains association in the MOF Model means that Package nesting is a strict composition relationship.

The main intended function of Package nesting is information hiding. Placing a Class or DataType in an inner Package rather than an outer one notionally makes it less visible to other meta-models. When the MOF visibility rules are defined (in a future revision of this specification), this information hiding will be more strongly enforced.

Nesting of Packages also affects the M1-level interfaces and implementations. The meaning of any element of a meta-model is potentially dependent on its context in a variety of ways. Thus, when the element is defined in a nested Package, its meaning may depend on the outer Package; for example, on Constraints or Classifiers declared therein. This means that anything that uses a nested element will also implicitly depend on the context. To avoid potential M1-level anomalies caused by this kind of dependency, the MOF Model does not allow a meta-model to import a nested Package or a Classifier defined within a nested Package.

© ISO/IEC 2005 - All rights reserved

The M1-level semantics of Package nesting are as follows. The behavior of an M1-level instance of a Classifier declared in a nested Package depends on state in both its immediate Package, and its enclosing Packages. As a result, the M1-level instance of the nested Classifier is inextricably tied to other instances within the outermost enclosing Package extent; see 8.8.4, “Package Extents,” on page 147.

8.7.2 Package Generalization

Package generalization allows an M2-level Package to reuse all of the definitions of another M2-level Package. Package generalization is expressed at the M2-level by connecting the super-Package and sub-Package using a Generalizes link. (The MOF Model’s Constraints mean that Generalization is effectively an aggregation in the UML sense.)

The M1-level semantics of Package generalization are as follows. The behavior of M1-level instances of the elements of an M2-level Package typically depends on M1-level behavior for M2-level super-Package elements. Indeed, an M1-level Package “instance” is substitutable for M1-level Package instances for M2-level super-Packages.

Package inheritance does not create any relationship between an instance of the super-Package and an instance of the sub-Package. Therefore an M1-level Package extent is not related to M1-level super- or sub-Package extents; see 8.8.4, “Package Extents,” on page 147.

8.7.3 Package Importation

Package importing allows an M2-level Package to selectively reuse definitions from other M2-level Packages. Package importation is expressed at the M2-level by placing an Import in the importing Package that is related to the imported Package by an Aliases link. In this case, the M2-level Import object has its “isClustered” attribute set to false. Since Package importation can be cyclic, it is neither an aggregation or a composition in the UML sense.

NOTE: The MOF Model’s Constraints make it illegal for a Package to import itself, or for any Package to import a nested Package. Furthermore, while the MOF Model allows Package importation to be cyclic, the preconditions for the MOF Model to IDL mapping disallow most dependency cycles, including those between Packages that result from cyclic importation.

The M1-level semantics of Package importation are minimal. No substitutability or state relationships exist between the M1-level instances of an importing or imported Package, or between their respective extents. Indeed, an importing Package will typically not even share implementation code with the imported Package.

8.7.4 Package Clustering

Package clustering allows an M2-level Package to selectively reuse definitions from other M2-level Packages, and also share M1-level implementation infrastructure. The M2-level expression of Package clustering is similar to that for Package importation; see above. The difference is that the Import object has “isClustered” set to true.

The M1-level semantics of Package clustering are similar to those of Package nesting because a cluster Package instance has its clustered Package instances as its components. However, unlike nested Packages, it is still possible to have a free-standing M1-level instance of such a Package whose extent is unrelated to any extent of a cluster Package.

8.8 Extents

This sub clause introduces the concept of an “extent” in more detail, and then gives the formal definitions of the extent of a Class, an Association, and a Package.

© ISO/IEC 2005 - All rights reserved

8.8.1 The Purpose of Extents

Current generation middleware systems typically aim to allow clients to use objects without knowledge of their locations or context. However, groups of objects generally exist in the context of a "server," which has responsibility for managing them. The implementation of an object often uses knowledge of its shared context with other objects to optimize performance, and so forth.

While statements about object location have no place in the MOF specification, the MOF Computational Model assumes a notion of context in many areas:

- The classifier-scoped features of an M2-level Class are notionally common to "all instances" of the Class.
- Mappings typically allow a client to query over "all links" in an Association instance.

It is impractical to define "all instances" or "all links" as meaning all instances or links in the known universe. Therefore, the MOF specification defines logical domains of M1-level instances that are the basis of these and other "for all" quantifications. These domains of M1-level instances are called extents.

Figure 8.3 shows the extents defined by two "instances" (on the right) of the example meta-model on the left. Notice that the static nesting of Packages, Classes, and Associations inside other Packages is mirrored in the extents (i.e., the dotted ovals).

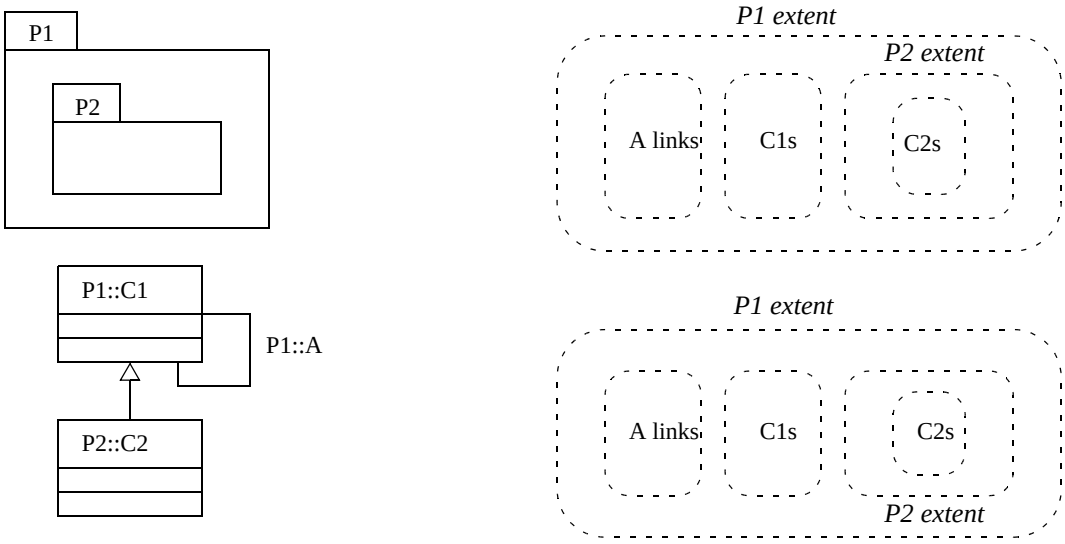

Figure 8.3 - Extents for two meta-model instances

Every Class instance or link belongs to precisely one Class or Association extent. These extents are part of Package extents, depending on the "lexical" structure of the meta-model. This means that extents are strictly hierarchical in nature. As we shall see in 9.2.1, "Meta Object Type Overview," on page 161 extents are related to the intrinsic container semantics of meta-objects.

NOTE: There is no requirement that extents have any manifestation in the partitioning of objects between physical MOF servers. However, there are clear performance advantages in implementing such a partitioning.

© ISO/IEC 2005 - All rights reserved

8.8.2 Class Extents

The extent of a Class is defined to be the complete set of M1-level instances of the Class that share classifier-scoped properties (e.g., Attribute values). A Class instance is created in the context of a Class extent and remains within that extent for its entire lifetime (i.e., until the instance is explicitly deleted).

8.8.3 Association Extents

The extent of an Association is defined to be the complete set of M1-level links for the Association. A link is created in the context of an Association extent and remains within that extent for its entire lifetime.

8.8.4 Package Extents

The extent of a Package is a conglomeration of the extents of Classes, Associations, and other Packages according to the following rules:

1. When an M2-level Package contains a Class or Association, an extent for the Package contains extents for the Classes and Associations.
2. When an M2-level Package contains nested Packages, an extent for the outer Package contains extents for the inner Packages.
3. When an M2-level Package clusters one or more other Packages, an extent for the cluster Package aggregates the extents for the clustered Packages.
4. When an M2-level Package inherits from another Package, an extent for the sub-Package:
    a. contains an extent for each nested Package, Class, or Association in the super-Package,
    b. aggregates an extent for each Package clustered by the super-Package, and
    c. aggregates or contains extents by recursive application of rule Clause 4 to the super-Package's super-Packages.

When a Package inherits from another Package by more than one route, the sub-Package extent will contain one extent for each directly or indirectly inherited Class, Association, or nested Package. This is illustrated in Figure 8.4. Notice that the extent for Package P4 contains only one C1 extent.

© ISO/IEC 2005 - All rights reserved

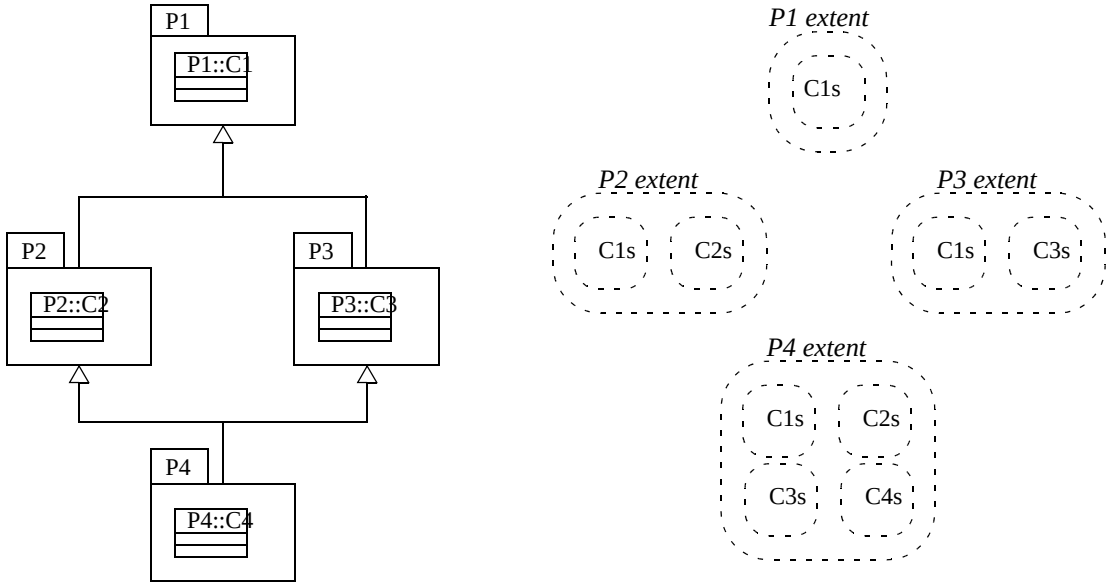

Figure 8.4 - Extents for Multiply Inheriting Packages

When a Package clusters other Packages by more than one route, the outer cluster Package will contain one extent for all directly or indirectly clustered Packages. This is illustrated in Figure 8.5. Notice that the relationship between the extents of a cluster Package and the extents of the clustered Packages is aggregation rather than strict containment. In particular, in the P4 case, the extent for P1 is not fully contained by either the P2 or P3 extents.

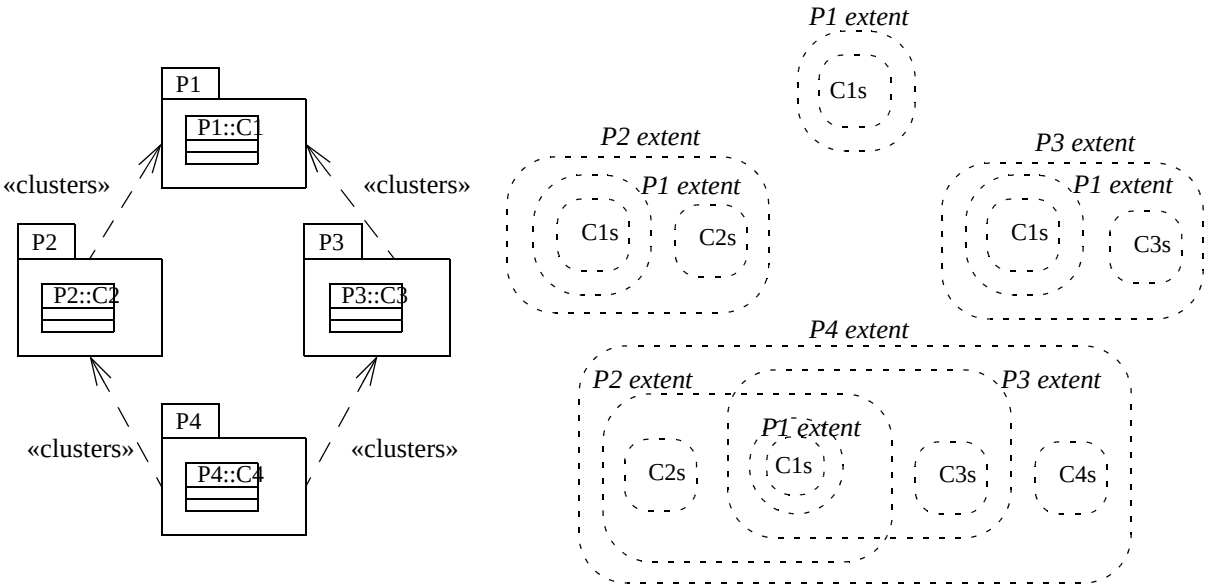

Figure 8.5 - Extents for Clusters of Clusters

© ISO/IEC 2005 - All rights reserved

NOTE: The extent for an M2 Package that imports (rather than clusters) other Packages does *not* contain extents for the imported Packages or their contents.

8.9 Semantics of Associations

Associations are the MOF Model's second mechanism for relating MOF values at the M1-level. A MOF M2-level Association defines a binary relation between pairs of M1-level Instances, where the relationships in the relation are called Links. The Links for a given M2-level Association conceptually belong to a Link set.

NOTE: While the MOF Model appears to support N-ary Associations, this is not so. There is a Constraint that states that an Association must have precisely 2 Association Ends; see "AssociationsMustNotUnary," on page 117.

An M2-level Association definition specifies the following properties:

- an Association "name,"
- a pair of AssociationEnds that each have:
  - a "name,"
    - a "type," which must be a Class,
    - a "multiplicity" specification,
    - an "isNavigable" flag, and
    - an "isChangeable" flag.
- an "isDerived" flag that determines whether the Association Links are stored explicitly or derived from other state.

8.9.1 MOF Associations in UML notation

A MOF Association is represented in UML notation as shown in Figure 8.6.

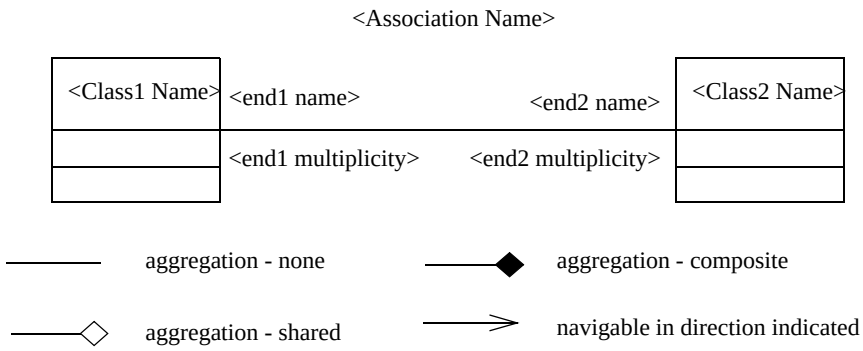

Figure 8.6 - An M2-level Association in UML notation

The connecting line denotes an Association between two Classes. The text of <Association Name>, <end1 name> and <end2 name> denote the "name" values for the respective Association and AssociationEnds. If the Association name is preceded by a forward slash, the Association has "isDerived" set to true.

© ISO/IEC 2005 - All rights reserved

The Class boxes denote the respective types of the two ends. If the two ends of an Association have the same type, the Association line loops around so that it connects a Class box to itself.

The <end1 multiplicity> and <end2 multiplicity> text give the multiplicity settings for the respective ends of the Association. The text that can appear here consists of an optional bounds specification with syntax:

```
<bounds>::= [<number> '..'] (<number> | '*')
```

and the optional keyword "ordered."

Finally, the navigability and aggregation of the ends of the Association are (partially) specified by the symbols at the respective ends of the line:

- An empty diamond indicates that the Instances at the labeled end "share" the Instances at other end.
- A filled diamond indicates that the Instances at the labeled end are "composed" of Instances at the other end.
- An arrow head indicates that the Association is navigable from the Instance at the other end to the Instance at the labeled end.

NOTE: There are a couple of anomalies in the mapping of UML Association notation to MOF Associations. First, while navigability and aggregation are orthogonal in the MOF, it is not possible to put both a diamond and an arrow head on the same end of a UML Association line. This means, for example, that it is not possible to express (the lack of) navigability from a component end to a composite end. Second, UML is imprecise about what an Association line with no arrowheads means. It can mean that the Association is not navigable, or alternatively that its navigability is not shown.

8.9.2 Core Association Semantics

This sub clause defines the core semantic model for M1-level Association instances in a rigorous, mapping independent fashion, and enumerates some important characteristics that follow from the definition.

8.9.2.1 A Mathematical Model of Association State

Given an M2 Association labeled as in Figure 8.6, the mapping to M1-level Link sets and Links can be modeled as follows:

1. The M1-level Instances of the M2-level Classes <Class1> and <Class2> belong to sets *Class1_Instances* and *Class2_Instances* that represent the sets of all possible instances of <Class1> and <Class2>, except for the respective null instances. (Note these sets are not restricted to current extant instances.)
2. The set *All_Links* is the Cartesian product of the sets *Class1_Instances* and *Class2_Instances*. Thus a *Link*, which is a member of *All_Links*, can be any tuple of the form "<c1, c2>" where "c1" and "c2" are members of *Class1_Instances* and *Class2_Instances* respectively.
3. The *Link_Set* is a subset of the set *All_Links* which consists of those *Links* that currently exist in the given M1-level Association.
4. If one or other of the AssociationEnds has "isOrdered" set to true, there is a partial ordering *Before* over the elements of *Link_Set* defined as follows. Assuming that <End1> of the Association is the one that is flagged as ordered:
    a. For each Instance "*i*" in *Class2_Instances*, we can define a subset $End2\_Links_i$ of *Link_Set* consisting of those *Links* in *Link_Set* for which the second tuple member is "*i*."

© ISO/IEC 2005 - All rights reserved

b. Given the *End2_Links*$_i$ sets as defined in item a. above, the *Before* ordering is defined between any pair of different *Links* in an *End2_Links*$_i$ set with 2 or more members. In other words, for any distinct *Link*$_j$ and *Link*$_k$ in *End2_Links*$_i$, we can say either *Link*$_j$ *Before Link*$_k$, or *Link*$_k$ *Before Link*$_j$.

c. The *Before* ordering is NOT defined between any pair of *Links* that belong to different *End2_Links* sets.

d. Where it is defined, the *Before* ordering is required to be:

i. transitive; i.e., L_i *Before* L_j and L_j *Before* L_k implies that L_i *Before* L_k, and

ii. anti-reflexive; i.e., L_i *Before* L_j implies *not* L_j *Before* L_i.

(If <End2> of the Association is ordered, substitute End2 for End1 and vice versa in the above.)

5. A *State* of an M1-level instance of an Association consists of the *Link_Set* and (if the Association is ordered) the *Before* ordering.

6. A *Well-formed State* is a *State* in which:

a. The *Links* set is a subset of *Valid_Links*, where *Valid_Links* is the subset of *All_Links* where the connected Instances currently exist.

b. The *End_Links*$_i$ sets as defined in item a. above conform to their respective Association End upper and lower bounds; that is,

i. the number of *Links* in each *End1_Links*$_i$ set must be greater than or equal to <End2.lower>, and less than or equal to <End2.upper>, and

ii. the number of *Links* in each *End2_Links*$_i$ set must be greater than or equal to <End1.lower>, and less than or equal to <End1.upper>.

Ideally, the computational semantics of M1-level Associations for a particular mapping should be describable as transformations from one *Well-formed State* to another. However, some mappings must be defined such that the *State* of an Association instances is not always a well-formed. For example, in the IDL mapping, deletion of an Instance may cause an *End_Links* set to contain too few *Links*.

The general model of an M1-level Association's *State* may be further constrained by M2-level Constraints on the Association or other elements of the meta-model. Other systematic restrictions may apply in some mappings; for example, 8.11.1, "The Reference Closure Rule," on page 153 and 8.11.2, "The Composition Closure Rule," on page 155.

8.9.2.2 Characteristics of M1-level Associations

The definitions of *Links* and *Link_Sets* above mean that M1-level Association instances have the following characteristics:

- *Links* only exist between existing Instances in a *Well-formed State*. When an Instance ceases to exist, any *Links* involving the Instance in any *Link_Set* cease to be universally meaningful.
- A *Link* "<a, b>" is distinct from a *Link* "<b, a>". In other words, *Links* are directed. (Whether or not the "direction" of a Link has a meaning depends on the underlying semantics of the reality that the M2-level Association describes.)
- *Links* do not have object identity, but are uniquely identified by the Instances at both ends.
- A *Link* cannot connect a null Class instance to any other instance (including itself).
- Since a *Link_Set* is defined to be a set, it cannot contain more than one copy of a given *Link*. In other words, M1-level Associations cannot contain duplicate links.

© ISO/IEC 2005 - All rights reserved

- The *Before* ordering on the *Links* in an *End_Links* set (where defined) can be represented by arranging the *Links* in a strictly linear sequence.
- There can be multiple *States* for a given M2-level Association, each corresponding to a different M1-level Association instance in separate Package instances. In this scenario:
  - a given *Link* can be a member of multiple *Link_Sets*, and
  - the *Before* orderings of different *States* will be independent.

8.9.3 AssociationEnd Changeability

The “isChangeable” flag for an AssociationEnd determines whether or not the APIs for the Association should allow clients to change Links in an M1-level Association instance. The precise interpretation of this flag is mapping specific.

8.9.3.1 AssociationEnd Navigability

The “isNavigable” flag for an AssociationEnd determines whether or not clients should be able to “navigate” the Links in an M1-level Association instance. The flag also determines whether or not the AssociationEnd can be used as a “key.” This flag’s interpretation (i.e., its impact on APIs) will depend on the mapping used.

8.9.4 Association Aggregation

The “aggregation” attributes of an Association’s two ends determines the aggregation semantics for the corresponding M1-level Association instances; see 8.10, “Aggregation Semantics,” on page 152. The impact of aggregation semantics are largely mapping specific. However, “composite” aggregation does place constraints on the *Link_Set* of a *Well-formed State*.

8.9.5 Derived Associations

When an M2-level Association has “isDerived” set to true, the resulting M1-level Association’s *Link_Set* is calculated from other information in the M1-level model. The M1-level semantics of derived Association instances is beyond the scope of the MOF specification.

8.10 Aggregation Semantics

As noted previously, the MOF Model provides two ways of relating MOF values; that is, Associations and Attributes. In both cases, a relation has a property known as aggregation that determines how strongly related values are tied together.

The MOF Model currently supports three aggregation semantics; that is, “none,” “shared,” and “composite” in order of increasing strength.

NOTE: In practice, the semantics of aggregation are mostly concerned with the life-cycles of related values. Since different mappings will use different strategies for managing the life-cycles of values, aggregation semantics are largely mapping specific.

8.10.1 Aggregation “none”

An Attribute or Association with aggregation of “none” has the weakest form of relation between values. This will typically correspond to independent life-cycles for both parties and the use of shallow copy semantics in a mapping.

© ISO/IEC 2005 - All rights reserved

8.10.2 Aggregation “composite”

An Attribute or Association with aggregation of “composite” has the strongest form of relation between values. A “composite” relation involving two types is asymmetric, with one “end” labeled as the “composition” type and the other end labelled the “component” type. An instance of the first type is “composed of” instances of the second type.

An M1-level “composite” relation is defined to have information model semantics that can be loosely described as containment semantics:

1. If a value “v1” is a component of some other value “v2” in a given composite relation, “v1” may not be a component of any other value “v3” in any composite relation. In short, a value can have *at most* one container in any “composite” relation. (This restriction does not apply when “v1” is a null instance.)

2. A value may not be related to itself in the closure of any “composite” relations. In short, a value may not directly or indirectly contain itself.

Other restrictions may apply to “composite” relations in some mappings (e.g., 8.11.2, “The Composition Closure Rule,” on page 155).

8.10.3 Aggregation “shared”

An Attribute or Association with aggregation of “shared” corresponds to a relation between values that is between “none” and “shared.”

NOTE: The semantics of “shared” aggregation should correspond to the semantics of an Aggregate in UML. Unfortunately, the OMG UML specification gives no clear guidance on what these semantics should be. As an interim measure, the use of “shared” aggregation in the MOF is discouraged.

8.11 Closure Rules

The MOF’s support for multiple Package “instances” introduces some potential anomalies into the computational model. These are resolved by three “closure” rules based on the definitions of extents in 8.8, “Extents,” on page 145.

8.11.1 The Reference Closure Rule

Recall that a Reference value is defined as a projection of an M1-level Class instance in an Association. Given that Association link sets are not global, a reference’s value must be a projection in a particular link set. There is an “obvious” candidate link set for typical M1-level Class instances, namely the link set belonging to the Package “instance” that contains the Class instance. This is shown in Figure 8.7.

© ISO/IEC 2005 - All rights reserved

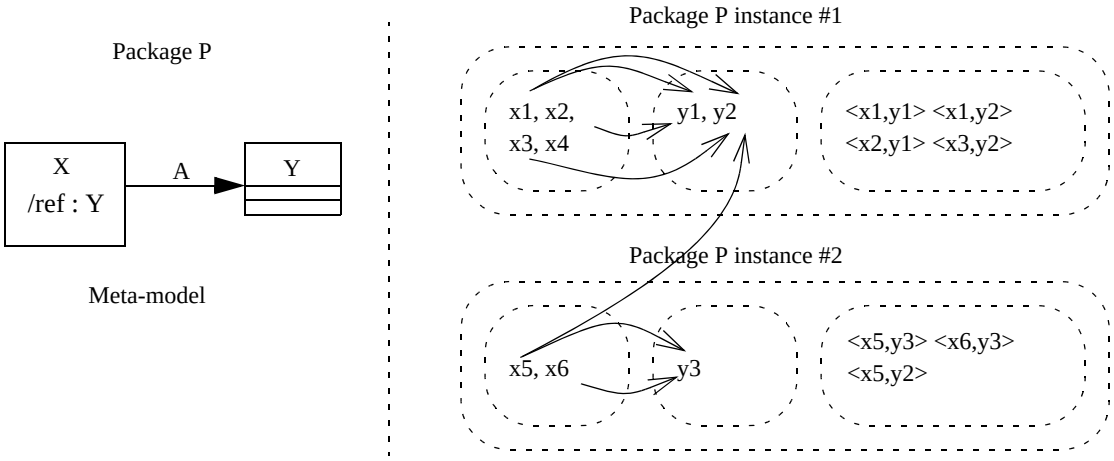

Figure 8.7 - References for multiple Package instances

Figure 8.7 shows the Y instances visible to each X instance in two Package instances. Notice that the link set in the second Package instance contains a link to a Y instance belonging to the first Package instance; that is, "<x5,y2>." This presents no particular problems, since the "x5" object can find the link to "y2" by looking in the A link set for its containing Package instance.

However, suppose that the "<x5,y2>" had been in the A link set for the first Package instance. Now an instance of the X Class has to look in the link sets of both (or in the general case, all) Package instances to find all of the links. Alternatively, an X instance might only look in the link set for its owning Package instance, leading to non-intuitive computational semantics for Reference values. (Consider the case where there are References for both Association Ends.)

To avoid such non-intuitive (and arguably anomalous) semantics, the computational semantics for Associations includes a runtime restriction that prevents the problematic links from being created. This restriction is called the *Reference Closure Rule*:

> "If Class C has a Reference R that exposes an Association End E in an Association A, then it is illegal to cause a link to be constructed such that an instance of C (or a sub-class of C) at the exposed End belongs to a different outermost extent to the A link set containing the link."

The *Reference Closure Rule* is shown graphically by Figure 8.8 for the case of an Association with a Reference to one end. The Reference Closure Rule is enforced by runtime checks on M1-level operations that construct links (e.g., the link add and modify operations). This can be achieved by using the "outermost_containing_package" operations on the respective meta-objects; see 10.2, "The Reflective Interfaces," on page 248.

© ISO/IEC 2005 - All rights reserved

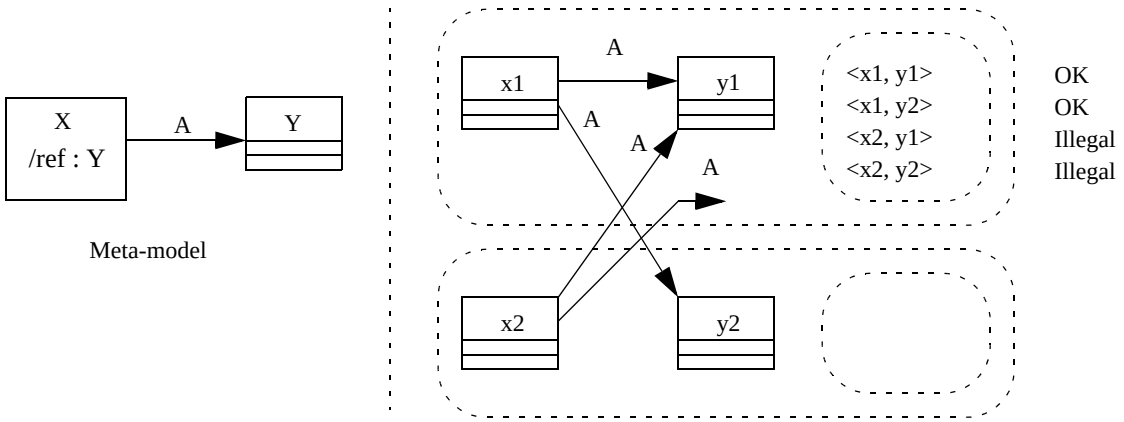

Figure 8.8 - The Reference Closure Rule

8.11.2 The Composition Closure Rule

The MOF Model provides constructs for declaring that the instances of one meta-model element are “composed of” instances of another; see 8.10, “Aggregation Semantics,” on page 152.

One of the key properties of composites is that a composite instance and its component instances have the same lifetime; that is, when a composite meta-object is deleted, all of its components are also deleted. This is not difficult to implement when the composite instance and its components all belong to the same Package instance. However, a range of problems can arise when a composition crosses one or more outermost Package extent boundaries. For instance:

- How do the server implementations for the respective extents ensure that deletion is reliable in the face of server crash, network partition, and so on?
- What are the access control implications of compositions? For example, should a client of one server / extent be able to implicitly delete components held in another server / extent?

To avoid having to deal with these difficult questions, the MOF computational model restricts the situations in which compositions may be formed. This restriction is called the *Composition Closure Rule*:

> “The composite and component instances in a composition along with any links that form the composition must all belong to the same outermost Package extent.”

The *Composition Closure Rule* is shown graphically by Figure 8.9. This shows the rule as it applies to both composite Attributes and composite Associations.

© ISO/IEC 2005 - All rights reserved

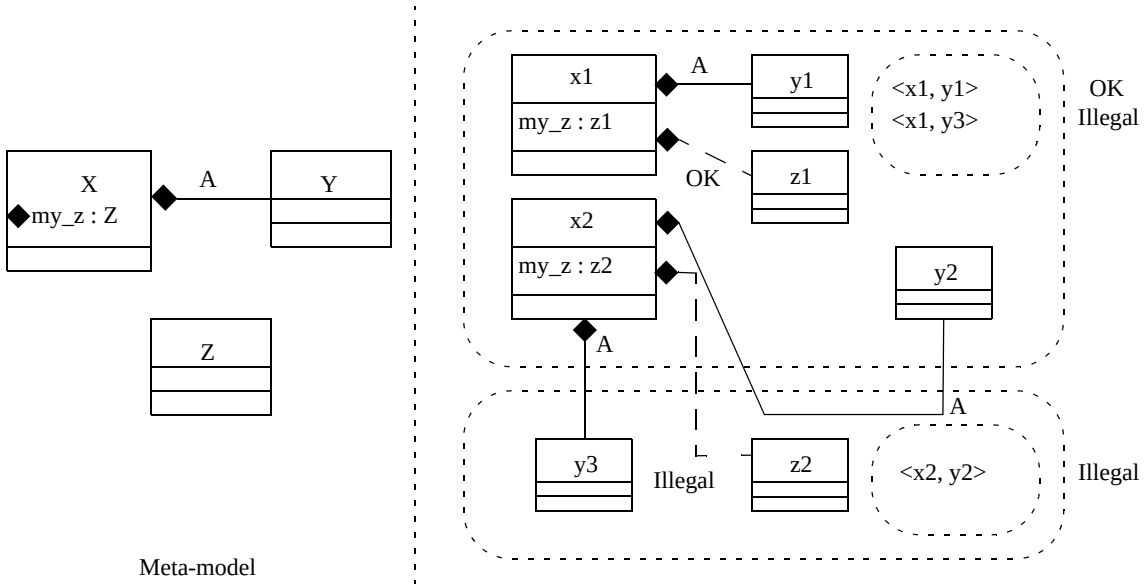

Figure 8.9 - The Composition Closure Rule

The Composite Closure Rule is enforced by runtime checks on M1-level operations that construct links in an Association with Composite semantics; e.g., the link add and modify operations. Similar checks are required for operations that update composite Attributes. The checks can be implemented by using the “immediate_container” and “outermost_containing_package” operations on the relevant meta-objects; see 10.2, “The Reflective Interfaces,” on page 248.

Since the null instance of a Class is defined to notionally belong to all extents for the Class, the Composition Closure Rule does not apply to Attributes with null values.

8.12 Recommended Copy Semantics

It is envisaged that some MOF mappings will provide APIs for copying metadata. The purpose of this sub clause is to recommend a semantic model for such copy operations. Suggested semantics are given for “shallow” and “deep” copying. (A shallow copy is one in which conceptual components of an object are copied and other connected objects are not. A deep copy is one in which both components and more loosely related objects are copied.)

© ISO/IEC 2005 - All rights reserved

The following table details what objects should and should not be copied. The semantics are defined from the perspective of an object being copied.

Table 8.1 - Copy semantics for different kinds of relationships

Construct	Target type	Aggregation	Shallow Copy	Deep Copy
Attribute	**Instance**	**none**	The Attribute value in the copy will be the same Instance value as in the original.	The Attribute value in the copy will be the same Instance value as in the original.
Attribute	**MOF data type**	**none**	The Attribute value in the copy will be the same data value as in the original. Embedded Instance values will be the same as in the original.	The Attribute value in the copy will be the same data value as in the original. Embedded Instance values will be the same as in the original.
Attribute	**Instance**	**composite**	The Attribute value in the copy will be a shallow copy of the Instance value as in the original.	The Attribute value in the copy will be a deep copy of the Instance value in the original.
Association	**Instance**	**none**	No link is created.	A link is created from the copy to the original link target.
Association	**Instance**	**shared**	A link is created from the copy to the original link target.	A link is created from the copy to a deep copy of the original link target.
Association	**Instance**	**composite**	A link is created from the copy to a shallow copy of the original link target.	A link is created from the copy to a deep copy of the original link target.

Unless otherwise stated, copying of a group of Instances related by Association or Attributes should give a 1-to-1 mapping between original Instances and copied Instances, and their respective relationships.

NOTE: The above suggested semantics do not cover copying of MOF values whose type is a native type. Those semantics will depend on whether or not the values in question are copyable.

8.13 Computational Semantics

8.13.1 A Style Guide for Metadata Computational Semantics

While the MOF specification gives the required computational semantics for M1-level metadata, it does not (and should not) state that these semantics constitute the only behavior. It is envisaged that vendor and end-user implementations of metadata servers may support additional semantics. In addition, the computational semantics of M2-level derived Attributes, derived Associations and Operations are not specified at all in the standardized part of the MOF Model.

In theory, the complete computational semantics of a meta-model server can include any behavior that the implementor chooses. The purpose of the sub clause is to set down some conventions to guide the implementor.

© ISO/IEC 2005 - All rights reserved

8.13.2 Access operations should not change metadata

Many operations on Instance and Associations are provided to support access to the public state of a model; e.g. the “get” operations for Attributes, the “query” operations for Associations. For normal (non-derived) Attributes and Associations, the standard computational semantics of an access operation are to simply return the corresponding value or collection. For derived Attributes and Associations, there are no standard semantics at all.

> In general, it is bad style for an access operation to have observable side-effects on the primary metadata. Similarly, it is bad style for an Operation with “isQuery” true to have such side-effects.

The rationale for this rule is that the user would not expect an access operation to have visible side-effects.

NOTE: It may be reasonable (for example) for an Attribute “get” operation to update a private counter Attribute that records the number of accesses. The legitimacy of this kind of behavior depends on whether or not the state modified can be classified as “primary” metadata.

8.13.3 Update operations should only change the nominated metadata

The standard semantics of metadata update operations define which metadata is expected to be modified by the operation. However, there is no explicit requirement that other metadata should not be changed.

> It is bad style for an update operation for a non-derived Attribute, Reference, or Association to change any primary metadata other than that which is identified by the standard semantics.

The rationale for this rule is that the user would not expect such changes to occur.

NOTE: This rule is not intended to apply to operations for derived Attributes, References or Associations, or to Operations with “isQuery” false.

8.13.4 Derived Elements should behave like non-derived Elements

M2-level Attributes and Associations can be defined as being derived from other information in a meta-model (i.e., by setting the respective “isDerived” flag to true). The required M1-level behavior of derived Elements is identical to that for equivalent non-derived Elements. Behavior that contradicts the semantics in this clause and in the relevant mapping specification is non-conformant.

However, since derived Attributes and Associations have to be implemented using mechanisms that are beyond the scope of the MOF Model, conformance is ultimately the responsibility of the meta-model implementor.

It is recommended that implementor defined M1-level operations for derived Elements should have MOF conformant behavior. The alternative (non-conformant behavior) tends to break the illusion that the Attribute or Association is “real,” and should be avoided. If the required semantics are unimplementable, the meta-model is incorrect.

8.13.5 Constraint evaluation should not have side-effects

The MOF specification does not define how Constraints defined in a meta-model should be evaluated. In particular, it does not define whether Constraint evaluation can change the metadata.

> It is bad style for the evaluation of a Constraint to change metadata.

© ISO/IEC 2005 - All rights reserved

The rationale is two fold. First, Constraints are provided as mechanism for specifying metadata correctness, not as a mechanism for defining behavior. Second, since the MOF specification does not say when Constraint evaluation should occur (in all cases), side-effects in Constraint evaluation could be a major source of interoperability problems.

8.13.6 Access operations should avoid raising Constraint exceptions

The MOF specification does not define when deferred Constraint evaluation should occur. In theory, it can occur at any time, including when the user invokes an access operation.

> It is bad style for an access operation on a non-derived Attribute, Reference, or Association to raise an exception to indicate that the metadata is structurally inconsistent or that a Constraint has been violated.

The rationale is that an application program that is reading metadata (rather than updating it) is typically not in a position to do anything about the violation of deferred structural constraints or model specific Constraint. Alternatively, an application may try to access the metadata, knowing that it is inconsistent, so that it can then correct it.

> It is bad style for an access operation on a derived Attribute, Reference, or Association to raise a similar exception *unless* the inconsistency makes it impossible to calculate the required derived value(s). The same rule applies to Operations with “isQuery” true.

The rationale being less prescriptive about derived access operations is that the formulae used to derive the value(s) will typically have certain assumptions about the consistency of the metadata.

© ISO/IEC 2005 - All rights reserved

© ISO/IEC 2005 - All rights reserved

9 MOF to IDL Mapping

9.1 Overview

This Clause defines the standard mapping from a model defined using the MOF Model onto CORBA IDL. The resulting interfaces are designed to allow a user to create, update, and access instances of the model using CORBA client programs. While the standard IDL mapping implies detailed functional semantics for an object server for a mapped model, it does not define the implementation.

Note that while the mapping specification is defined to be easy to automate, a conformant MOF implementation is not required to support automatic IDL generation.

9.2 Meta Objects and Interfaces

This sub clause describes the different kinds of meta-objects that represent MOF-based meta-data in a CORBA environment.

9.2.1 Meta Object Type Overview

The MOF to IDL mapping and the Reflective module share a common, object-centric model of meta-data with five kinds of M1-level meta-object; that is, “instance” objects, “class proxy” objects, “association” objects, “package” objects, and “package factory” objects. The relationships between M2-level concepts and M1-level objects is illustrated by the example in Figure 9.1.

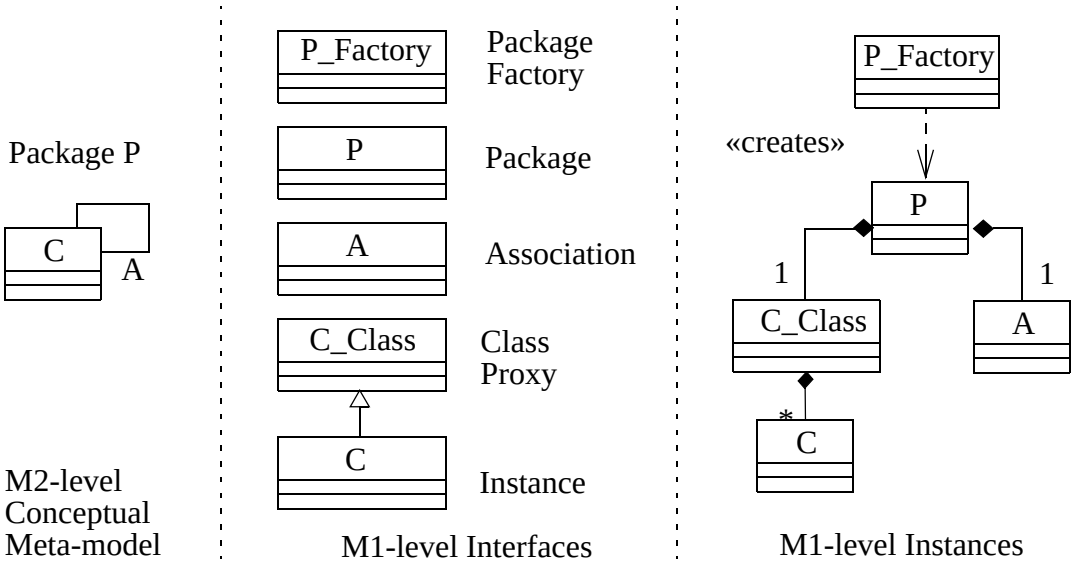

Figure 9.1 - Relationships between M1 and M2 level

© ISO/IEC 2005 - All rights reserved

The example shows how a simple M2-level meta-model (on the left) maps onto the five kinds of M1-level meta-object (in the center). The right of the diagram shows the intrinsic conglomeration relationships that exist between the meta-objects in a Package “instance.” (As noted, in 8.8, “Extents,” on page 145, these relationships do not always have strict containment semantics.)

NOTE: These intrinsic conglomeration relationships exist for all M1-level meta-objects. They have no explicit meaning in connection with the represented meta-data. Rather, they are provided to assist the management of meta-objects. (The intrinsic conglomeration relationships should not be confused with the M1-level composition relationships that correspond to M2-level composite Associations and Attributes.)

9.2.1.1 Package objects and Package Factory objects

The instances of an M2-level Package are represented as Package objects. A Package object is little more than a “directory” of read-only attributes that give access to a collection of meta-objects described by a meta-model. The attributes of a Package object refer to “static” objects. In particular, there is

- one Package attribute for each M2-level Package that is nested or clustered by the Package (none are present in the example above),
- one Class Proxy attribute for each M2-level Class in the Package, and
- one Association attribute for each M2-level Association in the Package.

The number and types of the static objects, and the corresponding attributes in an M1-level Package interface is determined by the M2-level Package specification. The objects cannot be directly created, destroyed, added, or removed by a client.

While there is usually a one-to-one correspondence between the Packages’ reference attributes and the static objects, this need not be the case. The correspondence is actually determined by the extent relationships as described in 8.8.4, “Package Extents,” on page 147. Thus, for example, when an M2-level Package is clustered by more than one route, there should be one M1-level Package object that is accessed via two attributes.

A Package object is typically obtained by invoking a “create” operation on Package Factory objects. This creates the Package object, and all of the necessary static objects. The arguments to the “create” operation are used to initialize any classifier-scoped Attributes defined within the M2-level Package.

9.2.1.2 Class Proxy objects

As stated above, a Package object contains one (and only one) Class Proxy object for each M2-level Class in the M2-level Package. A Class Proxy object serves a number of purposes:

- it is a factory object for producing Instance objects in the Package “instance,”
- it is the intrinsic container for Instance objects, and
- it holds the state of any classifier-scoped Attributes for the M2-level Class.

The interface of a Class Proxy object provides operations for accessing and updating the classifier-scoped attribute state. Other operations allow a client to invoke classifier-scoped Operations.

The interface also provides a factory operation that allows the client to create Instance objects. It also gives read-only access to the set of extant Instance objects contained by the Class Proxy object.

© ISO/IEC 2005 - All rights reserved

9.2.1.3 Instance objects

The instances of an M2-level Class are represented by Instance objects. An Instance object holds the state corresponding to the instance-scoped M2-level Attributes for the Class, and any other “hidden” state implied by the Class specification. Generally speaking, many Instance objects can exist within a given Package “instance.”

As described above, Instance objects are always contained by a Class Proxy object. The Class Proxy provides a factory operation for creating Instance objects that takes initial values for the instance-scoped Attributes as parameters. When an Instance object is created, it is automatically added to the Class Proxy container. An Instance is removed from the container when it is destroyed.

The interface for an Instance object inherits from the corresponding Class Proxy interface. In addition it provides:

- operations to access and update the instance-scoped Attributes,
- operations to invoke the instance-scoped Operations,
- operations to access and update Associations via Reference,
- operations that support object identity for the Instance, and
- an operation for deleting the Instance object.

9.2.1.4 Association objects

Links that correspond to M2-level Associations are not represented as meta-objects. Instead, an M1-level Association object holds a collection of links (i.e., the link set) corresponding to an M2-level Association. The Association object is a “static” object that is contained by a Package object, as described previously. Its interfaces provide:

- operations for querying the link set,
- operations for adding, modifying, and removing links from the set, and
- an operation that returns the entire link set.

9.2.2 The Meta Object Interface Hierarchy

This sub clause describes the patterns of interface inheritance in the CORBA IDL generated by the MOF to IDL mapping. The patterns are illustrated in Figure 9.2.

© ISO/IEC 2005 - All rights reserved

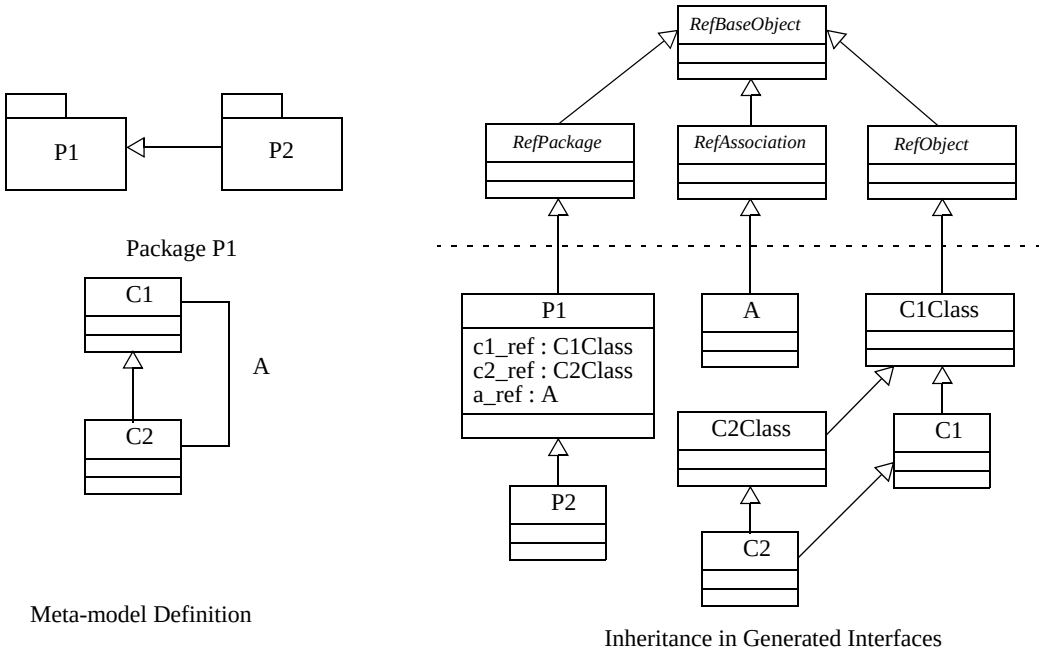

Figure 9.2 - Generated IDL Inheritance Patterns

Figure 9.2 shows an example MOF meta-model expressed in UML (on the left) that consists of two Packages P1 and P2. The first Package P1 contains Classes C1 and C2, where C2 is a subclass of C1 and an Association A that connects C1 and C2. The second Package P2 is then defined as a subpackage of P1.

The UML class diagram (on the right) shows the inheritance graph for the generated interfaces corresponding to the example meta-model.

The root of the inheritance graph is a group of four predefined interfaces that make up the Reflective module; see 10.2, “The Reflective Interfaces,” on page 248. These interfaces collectively provide:

- operations that implement meta-object identity,
- operations for finding a meta-object’s containing package instance(s),
- an operation for finding a meta-object’s M2-level description, and
- operations for exercising the functionality of a meta-object independent of its generated interface.

NOTE: The interfaces in the Reflective module are all designed to be “abstract;” that is, it is not anticipated that they should be the “most derived” type of any meta-object.

The interfaces for the Package objects, Association objects, Class Proxy objects, and Instance objects provide functionality as described previously. The inheritance patterns are as follows:

- All Package object interfaces inherit (directly or indirectly) from RefPackage.
- All Association object interfaces inherit from RefAssociation.

© ISO/IEC 2005 - All rights reserved

- All Class Proxy interfaces inherit (directly or indirectly) from RefObject.
- All Instance interfaces inherit from the corresponding Class Proxy interfaces.
- When an M2-level Package P2 inherits from another P1, the corresponding interface P2 inherits from P1.
- When an M2-level Class C2 inherits from another C1:
  - the Class Proxy interface for C2 inherits from the Class Proxy for C1, and
  - the Instance interface for C2 inherits from the Instance for C1.

The diamond pattern of interface inheritance is virtually unavoidable. The C2's Class Proxy needs to inherit the interface features for C1's classifier-scoped Attributes and Operations. Similarly, C2's Instance interface needs to inherit the instance-scoped interface features.

NOTE: The IDL mapping supports some Tags for specifying addition IDL supertypes of various generated interfaces; see "Tags for Specifying IDL Inheritance" on page 196. The effect of these Tags on the inheritance graph is defined by the relevant IDL templates; see "Package Template" on page 206, "Class Template" on page 209, "Class Proxy Template" on page 210, and "Association Template" on page 214.

9.3 Computational Semantics for the IDL Mapping

This sub clause specializes the MOF's general computational semantics (see Clause 8, "The MOF Abstract Mapping") for the MOF to IDL mapping.

9.3.1 The CorbaIdlTypes Package

The IDL mapping defines PrimitiveType instances that can be used to represent CORBA specific data types in a MOF meta-model. The IDL mapping maps each of these PrimitiveType instances into CORBA IDL data types, but other technology mappings typically will not map them.

Please note the following:

1. The types in the CorbaIdlTypes package are provided solely to ease migration of "legacy" meta-models and meta-data defined in the context of MOF 1.3 and earlier. Use of these types in new meta-models is discouraged as it will make them CORBA specific.
2. Implementations of the IDL mapping shall recognize the CORBA specific PrimitiveType instances based on their qualified names. Multiple PrimitiveType instances with the required qualified name shall be deemed to mean the same thing.

The CorbaIdlTypes package is a MOF package whose name is "CorbaIdlTypes." It contains the PrimitiveType instances defined below, and no other instances. The information below defines the value domain (set) for each type, and the syntax for encoding values for use in the Model::Constant "value" attribute.

CorbaOctet

This primitive data type represents the CORBA IDL 'octet' type.

value domain	The subset of integers in the range 0 to 255.
constant value syntax	CORBA IDL integer literal syntax.

© ISO/IEC 2005 - All rights reserved

CorbaShort

This primitive data type represents the CORBA IDL ‘short’ type.

value domain	The subset of the integers in the range -2^{15} to $+2^{15} - 1$
constant value syntax	CORBA IDL integer literal syntax with an optional leading ‘-’ character.

CorbaUnsignedShort

This primitive data type represents the CORBA IDL ‘unsigned short’ type.

value domain	The subset of the integers in the range 0 to $2^{16} - 1$
constant value syntax	CORBA IDL integer literal syntax.

CorbaUnsignedLong

This primitive data type represents the CORBA IDL ‘unsigned long’ type.

value domain	The subset of the integers in the range 0 to $2^{32} - 1$
constant value syntax	CORBA IDL integer literal syntax.

CorbaUnsignedLongLong

This primitive data type represents the CORBA IDL ‘unsigned long long’ type.

value domain	The subset of the integers in the range 0 to $2^{64} - 1$
constant value syntax	CORBA IDL integer literal syntax.

CorbaLongDouble

This primitive data type is the set of IEEE extended double precision floating point numbers (see ANSI/IEEE Standard 754-1985). This is the minimum requirement for the CORBA IDL ‘long double’ type.

value domain	The subset of the rational numbers that correspond to the values representable as IEEE extended double precision floating point numbers (96 bit).
constant value syntax	CORBA IDL floating point literal syntax with an optional leading ‘-’ character.

CorbaString

This primitive data type represents the CORBA IDL ‘string’ type.

value domain	The infinite set of all finite sequences of 8 bit characters (excluding the zero character value).
constant value syntax	A sequence of 8-bit characters. (Note: a Constant’s ‘value’ string for a CorbaString has no surrounding quotes and contains no character escape sequences.)

© ISO/IEC 2005 - All rights reserved

CorbaChar

This primitive data type represents the CORBA IDL ‘char’ type.

value domain	The set of 8 bit characters.
constant value syntax	One 8-bit character. NUL is represented as an empty String. (Note: a Constant’s ‘value’ string for a CorbaChar has no surrounding quotes and contains no character escape sequences.)

CorbaWChar

This primitive data type represents the CORBA IDL ‘wchar’ type.

value domain	The set of 16 bit characters.
constant value syntax	One 16-bit character. NUL is represented as an empty String. (Note: a Constant’s ‘value’ string for a CorbaWChar has no surrounding quotes and contains no character escape sequences.)

IDL for the CorbaIdlTypes module

The IDL for the CorbaIdlTypes package is given below. The IDL is produced by applying the IDL Mapping to the package. It would typically be “#included” by the IDL for meta-models that import the CorbaIdlTypes package.

```
#pragma prefix "org.omg.mof"

module CorbaIdlTypes {

    typedef sequence < octet > OctetBag;

    typedef sequence < octet > OctetSet;

    typedef sequence < octet > OctetList;

    typedef sequence < octet > OctetUList;

    typedef sequence < short > ShortBag;

    typedef sequence < short > ShortSet;

    typedef sequence < short > ShortList;

    typedef sequence < short > ShortUList;

    typedef sequence < unsigned short > UShortBag;

    typedef sequence < unsigned short > UShortSet;

    typedef sequence < unsigned short > UShortList;

    typedef sequence < unsigned short > UShortUList;

    typedef sequence < unsigned long > ULongBag;

    typedef sequence < unsigned long > ULongSet;

    typedef sequence < unsigned long > ULongList;

    typedef sequence < unsigned long > ULongUList;

    typedef sequence < unsigned long long > ULongLongBag;
```

© ISO/IEC 2005 - All rights reserved

```
    typedef sequence < unsigned long long > ULongLongSet;
    typedef sequence < unsigned long long > ULongLongList;
    typedef sequence < unsigned long long > ULongLongUList;
    typedef sequence < long double > ULongDoubleBag;
    typedef sequence < long double > ULongDoubleSet;
    typedef sequence < long double > ULongDoubleList;
    typedef sequence < long double > ULongDoubleUList;
    typedef sequence < string > StringBag;
    typedef sequence < string > StringSet;
    typedef sequence < string > StringList;
    typedef sequence < string > StringUList;
    typedef sequence < char > CharBag;
    typedef sequence < char > CharSet;
    typedef sequence < char > CharList;
    typedef sequence < char > CharUList;
    typedef sequence < wchar > WCharBag;
    typedef sequence < wchar > WCharSet;
    typedef sequence < wchar > WCharList;
    typedef sequence < wchar > WCharUList;
    // This interface would be inheritted by any the Package interface for any Package
    // that inheritted the CorbaIdlTypes package
    interface CorbaIdlTypesPackage : Reflective::RefPackage { };
    // This interface is here for completeness only. There is no point in instantiating
    // the CorbaIdlTypesPackage interface.
    interface CorbaIdlTypesPackageFactory {
        CorbaIdlTypesPackage create_corba_idl_types_package()
            raises (Reflective::MofError);
    };
};
```


© ISO/IEC 2005 - All rights reserved

9.3.2 Mapping of MOF Data Types to CORBA IDL Types

The following MOF PrimitiveType instances are mapped to CORBA IDL types in the IDL mapping. Other PrimitiveType instances have no defined mapping.

PrimitiveType instance	**Corresponding IDL type**
PrimitiveTypes::Boolean	boolean
PrimitiveTypes::Integer	long
PrimitiveTypes::Long	long long
PrimitiveTypes::Float	float
PrimitiveTypes::Double	double
PrimitiveTypes::String	wstring
CorbaIdlTypes::CorbaOctet	octet
CorbaIdlTypes::CorbaShort	short
CorbaIdlTypes::CorbaUnsignedShort	unsigned short
CorbaIdlTypes::CorbaUnsignedLong	unsigned long
CorbaIdlTypes::CorbaUnsignedLongLong	unsigned long long
CorbaIdlTypes::CorbaLongDouble	long double
CorbaIdlTypes::CorbaString	string
CorbaIdlTypes::CorbaChar	char
CorbaIdlTypes::CorbaWChar	wchar

NOTE: The MOF to IDL mapping *does not* define a standard mapping to the following CORBA IDL primitive data types: Principal, TypeCode, Any.

The MOF constructed data types are mapped to CORBA IDL types, as follows.

DataType subtype	**Corresponding IDL type**
StructureType(*name*, *fields*)	struct *name* { *fields*};.
CollectionType(*name*, *type*, ...)	typedef sequence < *type* > *name*;
EnumerationType(*name*, *labels*)	enum *name* { *labels* };
AliasType(*name*, *type*)	typedef *type name*;

NOTE: The MOF to IDL mapping *does not* define a standard mapping to the following CORBA IDL constructed types: arrays, bounded sequences, bounded strings, bounded wide strings, fixed types, union types, value types, boxed value types, interface types or abstract interface types.

© ISO/IEC 2005 - All rights reserved

9.3.3 Value Types and Equality in the IDL Mapping

The IDL mapping defines all MOF Instance types as CORBA object types that are descended from the “RefObject” interface; see 10.2.4, “Reflective::RefAssociation (abstract),” on page 265. Equality of Instance objects should be implemented as follows:

- Existing Instance objects are equal if and only if the “refMofId” operation defined by 10.2.3, “Reflective::RefObject (abstract),” on page 254 returns the same string for both objects.
- Non-existent Instance objects are deemed to be equal if and only if they have the same object reference; that is, when the “Object::_is_equivalent” operation returns true.

NOTE: An implementation must take care when comparing Instance object values to distinguish between non-existent (i.e., deleted) Instance objects and objects that may only be temporarily inaccessible. An operation should only raise an exception for a non-existent Instance object when it cannot be performed. In particular, an operation that replaces or removes defunct links or Instance values should not complain that the Instance being removed is defunct.

9.3.4 Lifecycle Semantics for the IDL Mapping

This sub clause defines the IDL mapping’s computational model for meta-object creation and deletion. It also gives definitions of copy semantics, though these should currently be viewed as indicative rather than normative.

9.3.4.1 Package object creation and deletion semantics

An M1-level Package object for a non-nested M2-level Package is created by invoking the create operation provided by the corresponding PackageFactory object. This create operation requires the caller to supply the values for all non-derived classifier-scoped Attributes. If the supplied initial values do not have the correct multiplicity or if they individually or collectively violate immediate Constraints defined in the metamodel, the create operation should raise an exception.

Instances of the following dependent M1-level objects are automatically created along with each M1-level Package object:

- An M1-level Package object is created for each nested Package within the outermost Package extent.
- An M1-level Package object is created for each clustered Package within the outermost Package extent.
- An M1-level Class Proxy object is created for each Class within the outermost Package extent.
- An M1-level Association object is created for each Association within the outermost Package extent.

The object references for the dependent Package and Class objects provide the “ref” attributes in the respective Package objects. The objects are initialized so that the outermost_package and enclosing_package operations return the appropriate M1-level Package objects.

NOTE: If an M2-level Package P2 clusters an existing top-level M2 Package P1, the above rules mean that two kinds of M1-level P1 Package objects can exist. If the user calls create on a P2 Package Factory object, the resulting P2 Package object will have its own dependent P1 Package object. On the other hand, if the user calls create on a P1 Package Factory, the resulting P1 Package object will be an outermost Package object. These two kinds of P1 Package objects behave identically, apart from their respective “refOutermostPackage” and “refOutermostPackage” operations; see 10.2.3, “Reflective::RefObject (abstract),” on page 254.

© ISO/IEC 2005 - All rights reserved

When an M1-level Class Proxy object is created, the values of the non-derived classifier-level Attributes are initialized from the corresponding create operation arguments. The “all_of_type” and “all_of_kind” collections will initially be empty, since no M1-level Instance objects will have been created in the Class Proxy extent.

NOTE: An implementation may support other mechanisms for creating or recreating outermost M1-level Package objects. Any such mechanism must also (re-)create and initialize the necessary dependent objects as above.

An outermost M1-level Package object can be destroyed using the “refDelete” operation; see 10.2.3, “Reflective::RefObject (abstract),” on page 254. The required computational semantics for deleting an outermost Package object are straightforward. The following things must occur (in an unspecified order):

- The binding between the outermost Package object and its object reference(s) must be revoked.
- The bindings between all dependent Package, Association, and Class Proxy objects and their object references must be revoked.
- All Instance objects within the extent of the outermost Package object must be destroyed as described below.

NOTE: A typical implementation will delete the metadata and reclaim the space used to store it. However, this behavior is not essential and in some situations it could be undesirable.

Dependent M1-level Package objects, M1-level Association objects and M1-level Class Proxy objects cannot be directly destroyed by the user. An implementation of the “refDelete” operation for these objects is required to raise an exception when called by client code. (The operations may be used to implement outermost Package deletion, but this is beyond the scope of this specification.)

9.3.4.2 Instance object lifecycle semantics

An M1-level Instance object can be created by invoking the appropriate create operation. Suitable create operations are present on both M1-level Class Proxy objects and M1-level Instance objects, depending on the M2-level Class inheritance graph. A create operation requires the caller to supply values for all non-derived instance-scoped Attributes for the Instance object. If any value does not conform to the Attribute’s multiplicity or if they individually or collectively violate any immediate Constraints on the meta-model, an exception is raised.

An Instance object is created within the extent of a Class Proxy object for the Instance’s M2-level Class. The Class Proxy can be found as follows:

1. Find the outermost Package extent containing the object on which the create operation was invoked.
2. Within that extent, find the one and only Class Proxy object for the M2 Class whose instance is being created.

If no Class Proxy can be found by the above, the create request violates the Supertype Closure Rule (see 9.3.11, “The Supertype Closure Rule,” on page 182) and an exception is raised.

Creation of an Instance object will also fail if the corresponding M2-level Class is abstract. Similarly, it will fail if the M2-level Class is a “singleton” Class and an Instance object for that Class already exists within the Class Proxy’s extent. In either case, an exception is raised.

When an Instance object is (successfully) created within the extent of a Class Proxy object, it becomes part of a collection returned by the Class Proxy object’s “all_of_kind” operation. The Instance object remains a member of that collection for its lifetime (i.e., until it is deleted).

An Instance object will be deleted in the following three situations:

© ISO/IEC 2005 - All rights reserved

1. When a client invokes the “refDelete” operation on the Instance object; see 10.2.3, “Reflective::RefObject (abstract),” on page 254.
2. When the Package object for the Instance object’s outermost Package extent is deleted (see above), and
3. When the Instance is a component of a “composite” Instance that is deleted. This applies to composites formed by both Associations and Attributes.

When an Instance object is deleted the following things must occur:

- The binding between the Instance object and its object reference(s) must be revoked.
- The Instance object must be removed from its Class Proxy object’s “all_of_type” collection.
- Any Instance objects that are components of the object being deleted must also be deleted.
- Links involving the deleted Instance object should be deleted as per the “Link lifecycle semantics” specification below.

An implementation will typically delete the state of an Instance object that has been deleted, and reclaim any associated space.

NOTE: When an Instance object is deleted, corresponding object reference values in non-composite Attributes of other objects become “dangling” references. These dangling references *should not* be automatically expunged or converted to nil object references, since doing so potentially destroys information and creates new structural errors. Instead, it is the user’s responsibility to ensure that dangling references in Attributes are tidied up in the most appropriate way.

9.3.4.3 Link lifecycle semantics

Links can be created and deleted in various ways. These include:

- by the user operations on M1-level Association objects; see 9.3.5, “Association Access and Update Semantics for the IDL Mapping,” on page 173,
- by the user operations corresponding to References on M1-level Instance objects; see 9.3.7, “Attribute Access and Update Semantics for the IDL Mapping,” on page 176,
- by the user copying metadata (using some vendor specific API); see 8.12, “Recommended Copy Semantics,” on page 156,
- by the user deleting one or other linked Instance objects; see 9.3.4.2, “Instance object lifecycle semantics,” on page 171, and
- when the server notices that a linked Instance object no longer exists.

A link is created within the extent of an Association object, and becomes part of the collection returned by the Association object’s “links()” operation. A link remains within the extent in which it was created for the lifetime of the link (i.e., until it is deleted). When a link is deleted, it is removed from the “links” collection. Removing a link does not affect the lifecycle of the linked Instance objects.

According to 8.9.2.2, “Characteristics of M1-level Associations,” on page 151, deletion of an Instance object causes any links for that object to become meaningless. Ideally, a well-formed M1-level Association instance should not contain such links. In practice, the immediate removal of meaningless links from an M1-level Association instance cannot always be implemented, in particular in the case of links that cross outermost Package extent boundaries.

© ISO/IEC 2005 - All rights reserved

Instead, a meta-object server is required to behave as follows. When an Instance object is deleted:

- all links referring to the Instance object that belong to Association instances within the same outermost Package extent as the Instance object *must* also be deleted, and
- any links referring to the Instance object that belong to Association instances in another outermost Package extent as the Instance object *may* also be deleted.

NOTE: The above semantics means that an Association instance can legally contain links that refer to defunct Instance objects in other extents.

9.3.5 Association Access and Update Semantics for the IDL Mapping

This sub clause describes the computational semantics of the Association object access and update operations defined in the MOF to IDL Mapping and the Reflective interfaces. With a couple of exceptions, these semantics transform one *Well-formed State* (as defined in 8.9.2.1, “A Mathematical Model of Association State,” on page 150) to another. The exceptions are as follows:

- Deletion of an Instance object in another outermost Package extent may cause an Association instance to contain links that are not members of *Valid_Links*.
- Deletion of an Instance object can cause an *End_Links* set to contain fewer links than is required.

M1-level Instance objects are passed as CORBA object reference values in IDL mapped operations. However, since the Association State model requires that Links connect Instances, it is not legal to pass the CORBA nil object reference value as a parameter to any operation on an M1-level Association.

NOTE: While the semantics of Associations are described (below) in terms of sets of pairs of M1-level Instance objects, this should not be read as implying any particular implementation approach.

9.3.5.1 Access Operations

There are three kinds of link access operations in the M1-level Association interface generated by the IDL mapping:

- The “all_links” operation returns the current *Link_Set* for an Association object.
- The “<end_name>” operations return a projection of the corresponding *End_Links* sets.
- The “exists” operation tests for the existence of a given *Link* in the *Link_Set*.

These operations are defined to be side-effect free; that is, they do not modify the *State* of the Association instance.

9.3.6 Link Addition Operations

The operations for adding links to an M1-level Association vary, depending on whether it has an ordered M2-level AssociationEnd:

- For an unordered Association, the “add” operation adds a *Link* to the *Link_Set*.
- For an ordered Association, the “add” and “add_before” operations both add a *Link* between a pair of Instances to the *Link_Set*. In the “add” case, the new Link is added after existing Links. In the “add_before” case, the new Link is added immediately before the link selected by the “before” argument.

© ISO/IEC 2005 - All rights reserved

More precisely, assuming that the first AssociationEnd is the ordered one and the new Link connects Instances *i* and *j*. The *Before* mapping is updated as follows:

- For "add," all *Links* that were in $End2\_Links_j$ prior to the operation are *Before* the new *Link* when it completes.
- For "add_before," the *Before_Link* connects the "*before*" and *j* Instances. For all *Links* that were in $End2\_Links_j$ and were *Before* the *Before_Link* prior to the operation, the pre-existing Link is *Before* the new *Link* after the operation. For all other Links that were in $End2\_Links_j$ prior to the operation, the new *Link* is *Before* the pre-existing *Link* after the operation.
- In both cases, the ordering of the other *End2_Links* sets are unchanged.

A number of constraints apply to the link addition operations:

- A new *Link* can only be added between extant Instances; that is, the new *Link* must be a member of *Valid_Links*.
- An operation cannot add a *Link* that is already a member of the *Link_Set*.
- An operation cannot add a *Link* if it would make the number of members of either $End1\_Links_i$ or $End2\_Links_j$ greater than the respective AssociationEnd's "upper" bound.
- An operation cannot add a *Link* that creates a Composition cycle, or that violates the Composition or Reference Closure rules.

9.3.6.1 Link Modification Operations

There are two "modify" operations for replacing an existing *Link* in the *Link_Set* of an M1-level Association. One operation (in effect) modifies the Instance at the first end of a *Link*, and the second modifies the Instance at the second end. While the operation signatures do not vary, the semantics of the "modify" operations depend on whether the M2-level Association has an ordered AssociationEnd.

- In the non-ordered case, a "modify" operation is almost identical to a "remove" operation followed by an "add" operation. The only difference is in the bounds checking; see below.
- In the ordered case, a "modify" operation can differ from an "add" followed by a "remove" in the way that the *Before* ordering is handled. Specifically, if we assume that the first AssociationEnd is the ordered one, the *Before* mapping is updated as follows:
  - For "modify_<end1_name>(*i*, *j*, *k*)", the new *Link* (between *k* and *j*) occupies the same position in the *Before* ordering of $End2\_Links_j$ as the *Link* (between *i* and *j*) that it replaces.
  - For "modify_<end2_name>(*i*, *j*, *k*)", the new *Link* (between *i* and *k*) becomes the last *Link* in the *Before* ordering of $End2\_Links_k$.
  - In both cases, the ordering of the other *End2_Links* sets are unchanged.

A number of constraints apply to the link modification operations:

- The *Link* that is replaced by the "modify" operation must be a member of *Link_Set*. However, it need not be a member of *Valid_Links*.
- The replacement *Link* that is created by a "modify" operation must be a member of *Valid_Links*.
- The replacement *Link* cannot already be a member of the *Link_Set*.
- A "modify" operation cannot produce a *Link* that would make the number of members in either the $End1\_Links_k$ or

© ISO/IEC 2005 - All rights reserved

End2_Links$_k$ sets greater than the respective AssociationEnd's "upper" bound.

- A "modify" operation cannot remove a *Link* if doing so would make the number of members of *End1_Links*$_i$ or *End2_Links*$_j$ less than the respective AssociationEnd's "lower" bound. (However, a *Link* **can** be produced in this situation.)
- A "modify" operation cannot produce a *Link* that creates a Composition cycle, or that violates the Composition or Reference Closure rules.

NOTE: A modify operation of the form "modify_<end1_name>(*i*, *j*, *i*)" is treated as a "no-op." In particular, it does not trigger checking of "lower" or "upper" bounds.

9.3.6.2 Link Removal Operations

The "remove" operation can be used to delete an exist *Link* (between *i* and *j*) from the *Link_Set* of an M1-level Association. The constraints that apply to the link removal operation are:

- The operation cannot remove a *Link* if doing so would make the number of members of *End1_Links*$_i$ or *End2_Links*$_j$ less than the respective AssociationEnd's "lower" bound.
- The operation cannot remove a *Link* that is not a member of the *Link_Set*. However, it should succeed if the *Link* is a member of *Link_Set* but not of *Valid_Links*.

9.3.6.3 Changeability, Navigability, and Derivedness

The operation descriptions given above assume that the AssociationEnds of the M2-level Association have been defined with "isChangeable" and "isNavigable" set to true. If this is not so, the main impact is that certain operations are suppressed:

- If an AssociationEnd of an Association is defined as non-changeable (i.e., when its "isChangeable" flag is set to false), the IDL mapping suppresses various link update operations. The "add," "add_before," and "remove" operations are suppressed if either AssociationEnd is non-changeable. Furthermore, the "modify_<end_name>" operation is suppressed for any AssociationEnd that is non-changeable, along with any related Reference-based operations.
- If an AssociationEnd of an Association is defined as non-navigable (i.e., when its "isNavigable" flag is set to false) the IDL mapping suppresses any link operations that depend on the ability to search based on that AssociationEnd. Specifically, it suppresses the "<assoc_end>", "add_before_<end>", "modify_<end>" operations.

Setting "isDerived" to be true for an M2-level Association is a "hint" that an M1-level Association's *Link_Set* and *Before* mapping should be computed from other M1-level information. Apart from this, the IDL mapping makes no distinction between derived and non-derived Associations. Equivalent IDL interfaces are generated in each case, and the semantics are defined to be equivalent. If a derived Association's operations are coded by hand, it is the programmer's responsibility to ensure that they implement the required semantics.

Some combinations of the Association and AssociationEnd flags result in generated interfaces that are of little use. For example:

- Setting "isChangeable" to be false on one AssociationEnd and not the other results in an M1-level Association that supports one "modify" operation but no "add" or "remove" operations.
- Setting "isChangeable" to be false on an Association that has "isDerived" set to false results in a "stored" Association with no operations to update the *Link_Set*.

© ISO/IEC 2005 - All rights reserved

9.3.7 Attribute Access and Update Semantics for the IDL Mapping

The IDL mapping maps M2-level Attributes to a variety of operations, depending on the Attribute's "multiplicity" settings. There are three major cases:

1. single-valued with bounds of [1..1]),
2. optional with bounds of [0..1], and
3. multi-valued.

Unlike Associations, the CORBA "nil" object reference is a legal (and logically distinct) value for any Class or object reference-valued Attribute. When an accessor operation returns a "nil" object reference, this does not necessarily mean that the Attribute has no value(s). In addition, the lifecycle semantics for Attributes in the IDL mapping mean that an accessor operation can return a reference for a non-existent object.

NOTE: While the semantics of Attributes are described (below) in terms of notional relations between M1-level values, this should not be read as implying any particular implementation approach.

9.3.7.1 Single-valued Attributes

The interfaces and semantics for single-valued Attributes are the simplest to describe. A single-valued Attribute (i.e., one whose "lower" and "upper" bounds are set to one) is mapped to these IDL operations:

- "<attr_name>"
- "set_<attr_name>".

The "<attr_name>" operation returns the current value of the named Attribute for an M1-level Instance object. In the single-valued case, this is a single Instance of the Attribute's M1-level base type as mapped by the IDL mapping. In the terminology of 8.6.1, "Attribute name and type," on page 142, the operation returns the M1-level value that is related to the Instance object by the notional "<attr_name>" Class — AttrType relation.

The "set_<attr_name>" operation replaces the current value of the named Attribute for an M1-level Instance with a new value. As before, the new value is a single Instance of the Attribute's M1-level base type as mapped by the IDL mapping. The operation replaces the existing Class — AttrType relationship with a new one between the Instance object and the new value.

The behavior of "set_<attr_name>" for a Class-valued Attribute (i.e., one with "composite" aggregation semantics) is constrained as follows:

- The new value supplied must be either a reference to an existing Instance object or a nil object reference.
- The new value (i.e., the component Instance) must not already be a component of another Instance object.
- The composite and component Instance objects must belong to the same outermost M1-level Package extent (i.e., the Composition Closure rule must not be violated).
- Creating the new Class — AttrType relationship must not create a composition cycle.

9.3.7.2 Optional Attributes

The interfaces and semantics for optional Attributes are also relatively straight-forward. An optional Attribute (i.e., one whose "lower" bound is 0 and whose "upper" bound is 1) maps to three operations:

© ISO/IEC 2005 - All rights reserved

1. “<attr_name>”
2. “set_<attr_name>”
3. “unset_<attr_name>”

The IDL mapping treats an M1-level optional Attribute as having two states. In the “set” state, the Attribute has a value that is an instance of the Attribute’s M1-level base type. In the “unset” state, the Attribute has no value.

In the single-valued case, “<attr_name>” simply returns the current M1-level value for the Attribute. In the optional case, the semantics depend on whether the Attribute is currently “set” or “unset.”

- If the Attribute is “set” (i.e., there is a Class — AttrType relationship between the Instance object and some other value), the “<attr_name>” operation returns the related value.
- If the Attribute is “unset” (i.e., there is no Class — AttrType relationship with the Instance object in the “class” role), the “<attr_name>” operation raises an exception.

The “set_<attr_name>” operation behaves exactly as in the single-valued case; it replaces the existing Class — AttrType relationship (if any) with a relationship with the new value. As a consequence, the Attribute enters the “set” state. The structural constraints for “set_<attr_name>” in the single-valued case apply here as well.

The “unset_<attr_name>” operation removes the Class — AttrType relationship, if it exists, leaving the Attribute in the “unset” state.

9.3.7.3 Multi-valued Attributes

The interfaces and semantics for multi-valued Attributes are relatively complicated, and depend to a considerable extent on the settings of the “isOrdered” and “isUnique” fields of the M2-level Attribute’s “multiplicity” property.

M1-level operations on multi-valued Attributes can be divided into two groups. The “<attr_name>” and “set_<attr_name>” operations access and update the Attribute’s state as a single value, transferring it as a CORBA sequence type. The other operations treat the Attribute’s state as a collection of values, and update it by adding, modifying, or removing individual elements of the collection.

The “<attr_name>” and “set_<attr_name>” operations transfer an Attribute’s M1-level state using a “collection” type. This is a named IDL sequence type whose base type is the Attribute’s M1-level base type, and whose name is determined by the “name” of the Attribute’s “type” and the settings of the “isOrdered” and “isUnique” flags. For details, see 9.7.1.5, “Literal String Values,” on page 200.

The “<attr_name>” operation returns the multi-valued Attribute’s value as a sequence using the IDL type described above. The contents of the result comprise the collection of base type instances related to the Instance object by the Class — AttrType relation. If “isOrdered” is true, the order of the Class — AttrType relationships determines the order of the elements in the sequence. If the collection is empty, the returned value is a zero length sequence.

The “set_<attr_name>” operation replaces the multi-valued Attribute’s value with a new collection of base type instances. If the Attribute is ordered, the order of the elements in the parameter value determines the order of the new Class — AttrType relationships.

A number of restrictions apply to the “set_<attr_name>” operation for multi-valued Attributes. These are as follows:

- If the Attribute’s “multiplicity” has the “isUnique” flag set to true, no two base type instances in the collection may be equal.
- If the Attribute’s “multiplicity” has a “lower” value greater than zero, there must be at least that many elements in the

© ISO/IEC 2005 - All rights reserved

collection.

- If the Attribute's "multiplicity" has an "upper" value other than the "UNBOUNDED" value (i.e., -1), there can be at most that many elements in the collection.

If the Attribute has composite semantics (i.e., the Attribute's "type" is expressed using a Class) the following restrictions also apply:

- Each element (i.e., Instance object) in the new value collection must be either a reference to an existing Instance object or a nil object reference.
- No element of the new value collection can already be a component of another Instance object.
- The composite and every component Instance object must belong to the same outermost M1-level Package extent (i.e., the Composition Closure rule must not be violated).
- Creating the new Class — AttrType relationships must not create any composition cycles.

The IDL mapping can define up to 7 additional operations for a multi-valued Attribute. There are up to 3 operations for adding new element values to an Attribute collection, up to 2 for modifying them and up to 2 for removing them. The subset that is available for a given Attribute depends on the "isUnique" and "isOrdered" flags in the M2-level Attribute's "multiplicity." This is shown in the table below.

isOrdered	**isUnique**	**Operations available**
false	false	add_<attr_name>, modify_<attr_name>, remove_<attr_name>
false	true	add_<attr_name>, modify_<attr_name>, remove_<attr_name>
true	false	add_<attr_name>, add_<attr_name>_before, add_<attr_name>_at, modify_<attr_name>, modify_<attr_name>_at, remove_<attr_name>, remove_<attr_name>_at
true	true	add_<attr_name>, add_<attr_name>_before, modify_<attr_name>, remove_<attr_name>

When "isOrdered" is set to false, the operations provided are the basic ones for adding, modifying, or removing element values. Given that the collection is unordered, there is no need to specify the position at which a new element value is added, or (in the false, false case) which of a number of equal element values should be modified or removed. The semantics of the operations for an unordered Attribute are as follows:

- The "add_<attr_name>" operation creates a new Class — AttrType relationship between the Instance object and the M1-level base type instance being added to the Attribute collection.
- The "modify_<attr_name>" operation replaces the Class — AttrType relationship between the Instance object and the M1-level base type instance being modified with another for the new element value.
- The "remove_<attr_name>" operation removes the Class — AttrType relationship between the Instance object and the M1-level base type instance being removed from the Attribute collection. Removing the instance decreases the Attribute collection's length rather than leaving a "hole."

These three operations must also respect the restrictions listed above for the multi-valued "set_<attr_name>" operation.

When "isOrdered" is set to true, the "add_<attr_name>," "modify_<attr_name>," and "remove_<attr_name>" operations take on additional semantics:

© ISO/IEC 2005 - All rights reserved

- The “add_<attr_name>” operation must ensure that the newly added element appears as the last element in the Attribute collection.
- The “modify_<attr_name>” operation must ensure that the replacement M1-level base type instance appears in the same position in the Attribute collection as the value that it replaces. When “isUnique” is set to false, the collection may contain duplicates. In this case, the operation should replace the first example of the instance in the ordered Attribute collection.
- When “isUnique” is set to false, the “remove_<attr_name>” operation should remove the first example of the instance in the ordered Attribute collection.

In addition, the client is provided with extra operations for order sensitive element update:

- The “add_<attr_name>_before” operation is similar to the “add_<attr_name>” operation, except that the new instance is added to the Attribute collection before an existing element designated by the caller. When “isUnique” is false, the operation is defined to replace the first example of the instance in the Attribute collection.
- When “isOrdered” is true and “isUnique” is false, the “add_<attr_name>_at,” “modify_<attr_name>_at,” and “remove_<attr_name>_at” are provided to allow the client to update the collection in the presence of duplicates. These operations specify an element insertion point or an element to be modified to be removed by giving a position index. For the purposes of these operations, the elements in an Attribute collection are numbered starting from zero according to the defined order of the members of the collection. The operations are as follows:
  - **add_<attr_name>_at** - inserts the new M1-level base type instance so that it appears at the position given. The instance originally at that position, and all instances will have their position indexes increased by one.
  - **modify_<attr_name>_at** - replaces the M1-level base type instance at the position.
  - **remove_<attr_name>_at** - removes the M1-level base type instance at the position given. Any instances in the collection that follow the removed instance will have their position indexes decreased by one (i.e., the operation does not leave a “hole” in the Attribute collection).

These five additional operations must also respect the restrictions listed above for the multi-valued “set_<attr_name>” operation.

9.3.7.4 Changeability and Derivedness

The previous semantic descriptions assume the M2-level Attribute has “isChangeable” set to true and “isDerived” set to false. This sub clause describes what happens if this is not the case.

If an Attribute has “isChangeable” set to false, the effect on the IDL mapping is that all generated operations for updating the Attribute’s state are suppressed. This does not preclude the existence of other mechanisms for updating the Attribute’s state.

Setting an Attribute’s “isDerived” flag to true, has no effect on the IDL mapping. The operations generated for the derived and non-derived cases are equivalent and they are defined to have equivalent semantics. If a derived Attribute’s operations are coded by hand, it is the programmer’s responsibility to ensure that they implement the required semantics.

9.3.7.5 Classifier scoped Attributes

The previous semantic descriptions assume the M2-level Attribute has “scope” set to “instance_level.” When an Attribute’s “scope” is “classifier_level,” we can model the notional relation that defines the M1-level Attribute state as a relation between the Class extent and the AttrType; see 8.6.3, “Scope,” on page 143. In the IDL mapping, this translates to a notional relation between a Class Proxy object and instances of the Attribute’s M1-level base type.

© ISO/IEC 2005 - All rights reserved

On this basis, an Attribute whose “scope” is “classifier_level” differs from one whose “scope” is “instance_level” in the following respects:

- The notional Class Proxy — AttrType relation supplies the value or values accessed and updated by “classifier_level” scoped Attribute operations.
- When the Attribute has aggregation semantics of “composite”:
  - the Composition Closure rule means that the Class Proxy object and M1-level Attribute value Instances must belong to the same extent, and
  - checking for composition cycles is unnecessary. The Class Proxy object that holds the Attribute value(s) is not an Instance, and thus cannot be a “component” in this sense.

9.3.7.6 Inherited Attributes

The previous semantic descriptions apply equally to Attributes defined within an M2-level Class, and Attribute inherited from supertypes of the Class.

9.3.7.7 Life-cycle Semantics for Attributes

The previous semantic descriptions say nothing about how an Attribute gets its initial value or values. (With the exception of the single-valued case of the “<attr_name>” operation, the semantic descriptions would “work” if no notional relationships existed initially.) In fact, the IDL mapping ensures that all M1-level Attributes get a client-supplied initial value:

- All “instance_level” scoped Attribute values for an M1-level Instance object are initialized from the parameters to the “create_<class_name>” operation.
- All “classifier_level” scoped Attribute values within the extent of an outermost M1-level Package are initialized from the parameters to the “create_<package_name>” operation.

An M1-level Attribute only exists while the M1-level Instance object or Class Proxy object that it belongs to exists. When the object is deleted, the notional relationships disappear as well.

Attributes with “composite” aggregation semantics have special life-cycle. When an object with a composite Attribute is deleted, the Instance object or objects that form its value are also deleted.

Note that unlike Associations, when an Instance object is deleted, the delete operation should make no attempt to tidy up “dangling references” to it.

© ISO/IEC 2005 - All rights reserved

9.3.8 Reference Semantics for the IDL Mapping

The IDL mapping maps References into a hybrid that combines an Attribute style interface with Association access and update semantics. In each case, a Reference operation maps fairly directly onto an Association operation as shown in the table below.

Multiplicity	**Reference Operation**	**Association Operation(s)** (assuming that the referenced AssociationEnd is the 2nd one)

optional	i.<*reference_name*>()	temp = a.<*referenced_end_name*>(i)
		if temp.size > 0 then
		  temp[0]
		else
		  raise NotSet

single- and multi-valued	i.<*reference_name*>()	a.<*referenced_end_name*>(i)

optional	i.set_<*reference_name*>(new)	old = a.<*reference_end_name*>(i)
		if old.size > 0 then
		  a.modify_<*reference_end_name*>(i, old[0], new)
		else
		  a.add(i, new)

optional	i.unset_<*reference_name*>()	old = a.<*reference_end_name*>(i) if old.size > 0 then a.remove(i, old[0])
single-valued	i.set_<*reference_name*>(new)	old = a.<*ref_end_name*>(i) a.modify_<*ref_end_name*>(i, old, new)

multi-valued	i.set_<*reference_name*>(new)	old = a.<*ref_end_name*>(i)
		for j in 0 .. (old.size - 1) do
		  a.remove(i, old[j])
		for j in 0 .. (old.size - 1) do
		  a.add(i, new[j])

multi-valued	i.add_<*reference_name*>(new)	a.add(i, new)
multi-valued	i.add_before_<*reference_name*>(new, before)	a.add_before_<*referenced_end_name*>(i, new, before)
multi-valued	i.modify_<*reference_name*>(old, new)	a.modify_<*referenced_end_name*>(i, old, new)
multi-valued	i.remove_<*reference_name*>(old)	a.remove_<*referenced_end_name*>(i, old)

In practice, an implementation also needs to transform exceptions reported for the Association operations into exceptions that apply from the Reference perspective. In addition, a “quality” implementation would ensure that Reference operations did not leave the Association object in a half way state following an exception.

NOTE: The above semantic mapping description is not intended as implying any particular implementation approach.

© ISO/IEC 2005 - All rights reserved

9.3.9 Cluster Semantics for the IDL Mapping

The impact of clusters on the IDL mapping semantics are largely described elsewhere. At the M1-level, a clustered Package behaves identically to a nested Package in terms of life-cycle and extent rules. The only significant difference is that clustering is not always a strict composition relationship at the M1-level; see 8.8.4, “Package Extents,” on page 147. In the IDL mapping, this means that two or more Package “ref” attributes point at the same clustered Package instance.

9.3.10 Atomicity Semantics for the IDL Mapping

All operations defined by the IDL mapping (including the Reflective versions) are required to be atomic and idempotent:

- If an operation succeeds, state changes required by the specification should be made, except as noted below:
  - When an Instance object is deleted, deletion of any component Instance objects may occur asynchronously.
  - When an Instance object is deleted, removal of links to the deleted Instance object may occur asynchronously.
- If an operation fails (e.g., by raising an exception), no externally visible changes should be caused by the failed operation.
- When the invocation of two or more operations overlap in time, the resultant behavior should be semantically equivalent to the sequential invocation of the operations in some order.

NOTE: The IDL mapping specification does not require a transactional or persistent implementation of a meta-data server.

9.3.11 The Supertype Closure Rule

The inheritance pattern for Instance and Class Proxy interfaces has an important consequence when one M2-level Class is a sub-Class of a second one.

Recall that each Class Proxy interface defines a factory operation for the corresponding Instance object, and that it also inherits from the Class Proxy interfaces for any M2-level super-Classes. Taken together, this means that any Class Proxy object has operations for creating Instance objects for both the M2-level Class, and all of its M2-level super-Classes.

Normally, this artifact of the IDL inheritance hierarchy is just a convenience. However, problems arise when an M2-level Class (e.g., P2::C2) has a super-Class that is imported from another M2-level Package (e.g., P1::C1); see Figure 9.3 on page 183. The Class Proxy interface corresponding to the C2 Class now has a factory operation to create instances of a Class from another Package, and therefore would appear to require all of the mechanisms for creating, accessing, updating, and deleting these instances. This is not what Package importing is defined to mean.

The adopted solution to this problem is to add an extra restriction to the MOF computational semantics. This restriction is known as the *Supertype Closure Rule*.

Supertype Closure Rule

Suppose that the Package extent for a non-nested M2-level Package P contains a Class Proxy object, which has a create operation for instances of Class C. This create operation can be used if and only if the M2-level closure of the Package P under generalization and clustering includes the M2-level Class C.

© ISO/IEC 2005 - All rights reserved

In other words, a factory operation for instances of an M2-level Class will only work within a Package instance with the machinery for supporting the Class. The Supertype Closure Rule is illustrated in Figure 9.3.

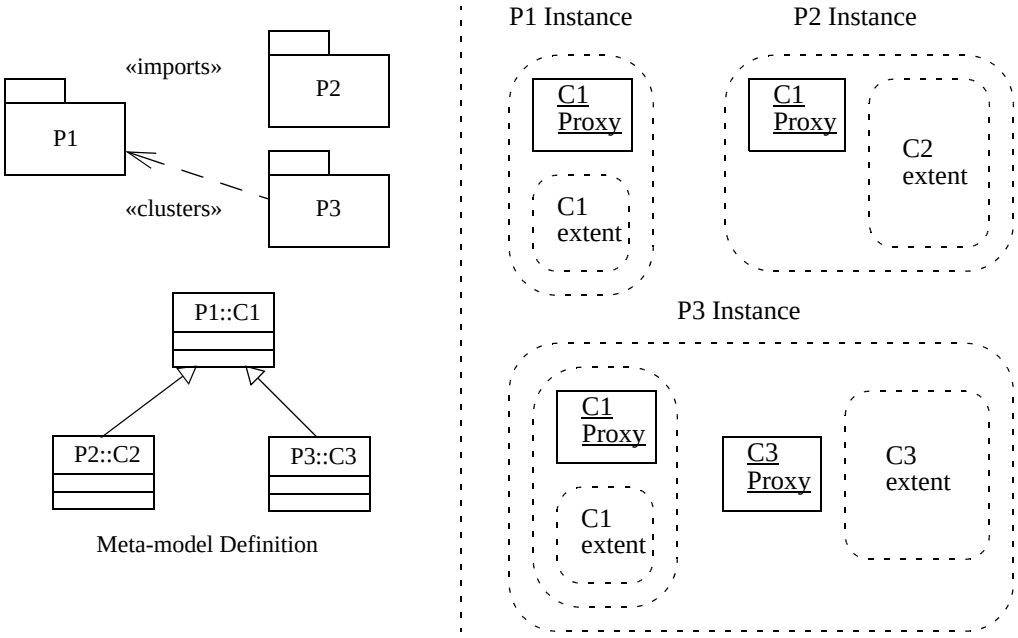

Figure 9.3 - Supertype Closure Rule

9.3.12 Copy Semantics for the IDL Mapping

The IDL mapping currently defines no APIs for copying meta-data. Copy semantics are therefore beyond the scope of this clause.

9.4 Exception Framework

This sub clause describes the way that Exceptions are organized in the MOF to IDL mapping. These exceptions are raised in a variety of CORBA interfaces, including:

- Reflective interfaces: (see 10.2.2, “Reflective::RefBaseObject (abstract),” on page 250, 10.2.3, “Reflective::RefObject (abstract),” on page 254, 10.2.4, “Reflective::RefAssociation (abstract),” on page 265, 10.2.5, “Reflective::RefPackage (abstract),” on page 269.
- Model interfaces (see 7.4, “MOF Model Classes,” on page 41 and 7.5, “MOF Model Associations,” on page 85).
- Specific interfaces produced by the mapping templates (see 9.8, “IDL Mapping Templates,” on page 202).

The exceptional conditions that arise in the context of the MOF to IDL mapping are classified into 5 groups:

1. Structural errors - this group covers those situations where the basic structural consistency rules for the metadata are (or would be) violated. For example, when there are too many or too few elements in a collection value.
2. Constraint errors - this group covers violations of metadata consistency rules specified in the metamodel using

© ISO/IEC 2005 - All rights reserved

Constraints.

3. Usage errors - this group covers those situations where a client tries to use the MOF interfaces in a meaningless way. For example, giving a ‘position’ for a collection element that is outside of the collection bounds.

4. Reflective errors - this group covers errors that can only occur when using the Reflective interfaces. For example, calling “refInvokeOperation” on an Attribute. These errors are the notional equivalent of runtime type errors.

5. Semantic errors - this group covers errors not covered above (i.e., implementation specific errors).

The complexity of the MOF means that the number of exceptional conditions is (at least in theory) unbounded. The precise set of possible exceptional conditions for just one operation in the mapped interfaces can be very hard to define. Constraint and Semantic errors are particularly difficult to tie down. Furthermore, including lots of exceptions in an IDL operation signature can make client code inordinately complex.

To solve these problems, the MOF IDL mapping defines the MofError exception that covers most of the exceptional conditions that might arise.

```
struct NamedValueType {
        wstring name;
        any value;
    };
    typedef sequence < NamedValueType > NamedValueList;
    exception MofError {
        wstring error_kind;
        RefBaseObject element_in_error;
        NamedValueList extra_info;
        wstring error_description;
    };
```

The fields of the MofError exception are defined as follows:

- **error_kind** is a wide string that denotes the particular kind of exceptional condition that is being raised. The formation of values for this field is discussed below.
- **element_in_error** is the DesignatorType for the object or feature that is deemed to be in error for this error condition. The detailed specifications of the error conditions below define which meta-object should be returned in each case. In situations where no M2-level meta-objects are available, this field may contain a nil object reference.
- **extra_info** is a list of name / value pairs that provides the client with extra information about the error condition.

  The list consists of zero or more standardized name / value pairs, followed by any implementation specific pairs. For the standardized part of the list, the sequence of the pairs and the values (including casing) of the names are mandatory. This allows clients to extract list elements by position or by matching names. It is recommended that implementors take the same approach for the implementation specific part of the list.
- **error_description** is a human readable diagnostic message in a wide string. The contents of this field are not specified by this document.

© ISO/IEC 2005 - All rights reserved

NOTE: The standardized name / value pairs for the **extra_info** field represent a compromise between the anticipated cost of implementation and the provision of useful information to the caller. Implementors are encouraged to provide additional information. Similarly, implementors are encouraged to provide detailed and informative diagnostics in the **error_description** field.

9.4.1 Error_kind string values

The values of the **error_kind** field or MofError are structured using Java's reversed domain name syntax:

"org.omg.mof:structural.composition_cycle"

"au.edu.dstc.mofamatic:botched_assertion"

The values for each group of errors are as follows:

Structural and Reflective errors: the prefix **"org.omg.mof:"** followed by either **"structural." or "reflective."** and then the specific error name in lowercase with underscores between words. These values are defined as constants in the IDL for the Reflective module.

Constraint errors: the IDL prefix for the metamodel (if any), followed by **":constraint."** followed by the qualified constraint name using the Format2 convention. For example, a Constraint named "MyConstraint" declared in "PackageA::ClassB," the error kind string value is:

":constraint.package_a.class_b.my_constraint"

or with an IDL prefix of **"com.acme"**

it is:

"com.acme:constraint.package_a.class_b.my_constraint"

See "Constraint Template" on page 245 for the definitive specification.

- Usage errors: not applicable. None of these error conditions are signalled using MofError.
- Semantic errors: an implementation specific prefix, followed by **":semantic."** followed by an implementation specific string. It is strongly recommended that the implementation specific part follow the conventions above (i.e., reverse domain names, all lowercase, periods for qualification and underscores between words).

9.4.2 Structural Errors

All structural errors are signalled using MofError. With the exception of "Underflow," the consistency rules covered by the structural errors are either pre- or post-conditions on operations.

© ISO/IEC 2005 - All rights reserved

The MOF IDL mapping defines the structural errors as defined in the table below.

Structural error	“Element_in_error”	Standard “extra_info”	Description
Underflow	Attribute, Parameter, or Association End defining the Multiplicity that is violated.	none	"Underflow" arises when a collection or projection contains fewer values than is required by the corresponding Multiplicity.lower. Note that the evaluation “underflow” is context dependent. For an operation that takes a collection value as a parameter, or whose net effect is to decrease the number of elements in a multi-valued Attribute or a projection of a Link set, “underflow” is treated as an immediate constraint. In other cases, “underflow” is treated as a deferred constraint.
Overflow	Attribute, Parameter, or Association End defining the Multiplicity that is violated.	none	"Overflow" arises when a collection or projection contains more values than is allowed by the corresponding Multiplicity.upper.
Duplicate	Attribute, Parameter, or Association End defining the Multiplicity that is violated.	“duplicate” : Any(<Value>) A value that appears more than once in the unique collection / projection.	"Duplicate" arises when a collection or projection whose corresponding Multiplicity.is_unique is true contains duplicate values. For example, when two or more values at different positions in the collection or projection that are “equal” according to the definitions in 8.4, “Semantics of Equality for MOF Values,” on page 140.
Reference Closure	Reference for which the closure rule is violated.	“external” : Any(<Instance>) An Instance that violates a closure rule with respect to the Association being updated.	"Reference Closure" can arise when an Association extent contains a link for an Instance object belonging to another outermost Package extent. More particularly, this happens when the Instance object's M2-level Class (or a super-Class ancestor) has a Reference to the M2-level Association. See 8.11.1, “The Reference Closure Rule,” on page 153.
Composition Closure	Attribute or Association for which the closure rule is violated.	“external” : Any(<Instance>) An Instance that was passed as or within in an operation parameter that violates the closure rule.	"Composition Closure" arises when an Instance object is member of a composite that crosses an outermost Package extent boundary. See 8.11.2, “The Composition Closure Rule,” on page 155.

© ISO/IEC 2005 - All rights reserved

Structural error	“Element_in_error”	Standard “extra_info”	Description
Supertype Closure	Class of the object that cannot be created.	none	"Supertype Closure" arises when a client attempts to create an Instance object in a Package extent that does not support its M2-level Class. See 9.3.11, “The Supertype Closure Rule,” on page 182.
Composition Cycle	Attribute, Reference or Association that is being updated to form the cycle.	“cyclic” : Any(<Instance>) A composite Instance passed as or within a parameter that would become cyclic as a result of this operation.	"Composition Cycle" arises when an Instance object is a component of itself via one or more relationships defined by composite Associations or composite Attributes.
Nil Object	Reference or Association End for which the nil object reference was supplied.	none	"Nil Object" arises when an Association operation is passed a CORBA nil object reference.
Inaccessible Object	Attribute, Parameter, Reference, or Association End for which the inaccessible object was detected.	“inaccessible” : Any(<RefObject>) An Instance object that was inaccessible.	"Inaccessible Object" arises when an operation tries to use an Instance object only to find that it is currently inaccessible.
Invalid Object	Attribute, Parameter, Reference, or Association End for which the invalid object was detected.	“invalid” : Any(<RefBaseObject>) An object reference for a MOF meta-object that does not exist.	"Invalid Object" can arise when an object operation detects a reference for a non-existent (i.e.,deleted) object.
Already Exists	Class of the object that already exists.	“existing” : Any(<Instance>) The pre-existing singleton Instance object for the extent.	“Already Exists” arises when a client attempts to create a second Instance object for an M2-level Class with “isSingleton” of true.

NOTE: There are no mandatory **‘extra_info’** pairs for "Overflow" and "Underflow" because the error conditions occur in such a wide range of contexts that it is difficult to come up with a set that is universally applicable. Vendors are encouraged to innovate by defining non-standard pairs.

The following IDL constants define the corresponding **error_kind** strings.

```
const string UNDERFLOW_VIOLATION =
    "org.omg.mof:structural.underflow";
const string OVERFLOW_VIOLATION =
    "org.omg.mof:structural.overflow";
const string DUPLICATE_VIOLATION =
    "org.omg.mof:structural.duplicate";
const string REFERENCE_CLOSURE_VIOLATION =
    "org.omg.mof:structural.reference_closure";
```

© ISO/IEC 2005 - All rights reserved

```
const string SUPERTYPE_CLOSURE_VIOLATION =
    "org.omg.mof:structural.supertype_closure";
const string COMPOSITION_CYCLE_VIOLATION =
    "org.omg.mof:structural.composition_cycle";
const string COMPOSITION_CLOSURE_VIOLATION =
    "org.omg.mof:structural.composition_closure";
const string NIL_OBJECT_VIOLATION =
    "org.omg.mof:structural.nil_object";
const string INACCESSIBLE_OBJECT_VIOLATION =
    "org.omg.mof:structural.inaccessible_object";
const string INVALID_OBJECT_VIOLATION =
    "org.omg.mof:structural.invalid_object";
const string ALREADY_EXISTS_VIOLATION =
    "org.omg.mof:structural.already_exists";
```

9.4.3 Constraint Errors

Constraint errors occur when a consistency rule is defined as a Constraint in the metamodel.

All Constraint errors are signalled by raising MofError. The fields of the MofError exception are defined as follows:

- The **error_kind** string is defined by the IDL mapping rules (see 9.8.16, “Constraint Template,” on page 245).
- The **element_in_error** is the designator for the ModelElement whose Constraint has been violated. In the case of Constraint on a DataType, the designator is the Parameter or Attribute for the context in which the erroneous DataType instance occurred.
- The value of the **extra_info** field is implementation specific. Where possible, the implementation should provide the constrained object(s) or value(s) for which the constraint is violated.

Constraints can be defined with an “evaluationPolicy” of “immediate” or “deferred.” In the former case, violations of the rule are likely to be reported when a constrained object is created or updated. In the latter case, violations are likely to be reported when deferred Constraint checking is triggered.

NOTE: The above statements assume that constraint checking is implemented according to the spirit of 8.8, “Extents,” on page 145.

9.4.4 Semantic Errors

The Semantic error group is the “catch all” for otherwise unclassified implementation specific errors. Semantic errors are signaled by raising the MofError exception when appropriate. Possible sources of this error include:

- additional metadata consistency rules that are not specified in the metamodel,
- implementation specific access control violations,

© ISO/IEC 2005 - All rights reserved

- resource limitations in a metadata server, and
- internal errors in a metadata server.

The values of the MofError exception fields for a Semantic error are implementation specific:

- Implementors should define a unique string for the **error_kind** field to distinguish the different kinds of Semantic error. These values should conform to the pattern described in 9.4.1, “Error_kind string values,” on page 185.”
- The values and meanings of the **element_in_error** and **extra_info** fields should be defined as appropriate.

9.4.5 Usage Errors

The Usage error group indicates inappropriate use of the MOF IDL interfaces. They can arise when a client is using either the Reflective interfaces, or the interfaces generated by the IDL mapping.

The Usage errors are signalled using their own exceptions.

Table 9.1 - Usage Exceptions

Usage Exception	Arguments	Description
NotFound	none	NotFound is raised by modify and remove operations on multi-valued Attributes, References, and Associations when the argument that should identify the member or link to be removed does not match any value that is currently there.
NotSet	none	NotSet is raised when a client attempts to read the element value of an optional collection (i.e., one with bounds of [0..1]) when the collection is empty.
BadPosition	none	BadPosition is raised when a positional add, modify, or remove operation is supplied with a ‘**position**’ argument whose value is out of range. The collection’s current size is returned in the exception’s ‘**current_size**’ field. This will be 0 if the collection is empty, 1 if it contains a single member, and so on.

NOTE: The members of a collection value containing *size* elements are numbered {*0, 1,... size - 1*} for the purposes of the positional update operations. The positional modify / remove operations are defined to modify or remove the member indexed by the **position** (i.e., position values in the range *0 to size - 1* inclusive are valid). The positional add operation is defined to insert a member before the member indicated by the **position**. In this case, position values in the range *0 to size* inclusive are valid, with *size* meaning “insert at the end.”

The IDL declarations for the Usage error exception are as follows:

```
exception NotFound {};

exception NotSet {};

exception BadPosition {
    unsigned long current_size;
};
```

© ISO/IEC 2005 - All rights reserved

9.4.6 Reflective Errors

Reflective error conditions occur exclusively in operations in the Reflective interfaces. They occur when a Reflective operation is invoked with parameters that contradict the target object's description in the metamodel. When the client uses interfaces generated by the IDL mapping, the static type checking based on the specific IDL signatures should prevent the equivalent errors from occurring.

In most cases, the MofError exception is used to signal reflective errors. Table 9.2 lists the Reflective errors that are signalled using MofError, along with the MofError field specifications and descriptions. All are pre-conditions for the respective operations.

Table 9.2 - Reflective Errors signalled using MofError

Reflective error	"Element_in_error"	Standard "extra_info"	Description
Invalid Designator	ModelElement that is invalid	none	"Invalid Designator" arises when a "feature" parameter: • is not a Model::ModelElement, or • does not denote an accessible, non nil CORBA object.
Wrong Designator Kind	ModelElement that has the wrong kind	none	"Wrong Designator Kind" arises when the supplied designator has an inappropriate most-derived type. For example, when a Model::Attribute is supplied where a Model::Operation is required.
Unknown Designator	ModelElement that is not known.	none	"Unknown Designator" arises when the supplied designator does not belong in this context. For example, when a Model::Attribute is not a member of this Instance's Class or its superClasses.
Abstract Class	Class that is abstract.	none	"Abstract Class" arises when a client calls "refCreateInstance" for a Class that is defined as abstract.
Not Changeable	ModelElement that has "isChangeable" = false	none	"Not Changeable" arises when an update operation is attempted on something that is defined by the metamodel to be not changeable.
Not Navigable	AssociationEnd that has "isNavigable" = false	none	"Not Navigable" arises when RefAssociation operations are attempted for an AssociationEnd that is defined by the meta-model to be not navigable.
Not Public	ModelElement that has "visibility" = "private_vis" or "protected_vis"	none	"Not Public" arises when an operation is attempted for a "private" or "protected" feature.
Wrong Scope	Attribute or Operation with "scope" = "instance_level"	none	"Wrong Scope" arises when an attempt is made to use an instance-level Attribute or Operation from a Class proxy object.

© ISO/IEC 2005 - All rights reserved

Table 9.2 - Reflective Errors signalled using MofError

Reflective error	"Element_in_error"	Standard "extra_info"	Description
Wrong Multiplicity	Reference or Attribute used in error.	none	"Wrong Multiplicity" arises when a reflective operation is requested where the corresponding specific operation does not exist for this feature's multiplicity. For example: • a member update on a [0..1] or [1..1] feature, • a unset on a feature that is not [0..1], • an add_value_at on an unordered feature.
Wrong Type	Attribute, Reference, AssociationEnd, or Parameter for the value that is in error.	"invalid_value" : Any The value or object whose type is incorrect in this context. (The first version is used when the value in error was passed as an Any, and the second when it was passed as a RefObject.) "expected_type" : Any(TypeCode) The CORBA TypeCode that the value should have been.	"Wrong Type" arises when a RefObject or an Any value has the wrong type for context in which it was supplied. For example; • A RefObject whose most derived type is incorrect; for example, has the wrong M2-level Class or is a Class proxy instead of Instance, or vice versa. • An Any value that contains a single value where a sequence is required, or vice versa. • An Any value that contains a single value or sequence of values of the wrong CORBA type.
Wrong Number Parameters	Class or Operation for which the wrong number of actual parameters was supplied.	"number_expected" : Any(Unsigned Long) The expected number of actual parameters.	"Wrong Number Parameters" arises when a client calls "refCreateInstance" or "refInvokeOperation" with too few or too many parameters.
Invalid Deletion	A nil object reference	none	"Invalid Deletion" arises when a client calls "refDelete" on a meta-object that cannot be deleted this way; that is, an Association object, a Class Proxy object, or a dependent Package object.

The following IDL defines the **error_kind** strings for the above Reflective errors:

```
const string INVALID_DESIGNATOR_VIOLATION =
    "org.omg.mof:reflective.invalid_designator";
const string WRONG_DESIGNATOR_DESIGNATOR_VIOLATION =
    "org.omg.mof:reflective.wrong_designator_kind";
const string UNKNOWN_DESIGNATOR_VIOLATION =
    "org.omg.mof:reflective.unknown_designator";
const string ABSTRACT_CLASS_VIOLATION =
    "org.omg.mof:reflective.abstract_class";
const string NOT_CHANGEABLE_VIOLATION =
    "org.omg.mof:reflective.not_changeable";
```

© ISO/IEC 2005 - All rights reserved

```
const string NOT_NAVIGABLE_VIOLATION =
    "org.omg.mof:reflective.not_navigable";
const string NOT_PUBLIC_VIOLATION =
    "org.omg.mof:reflective.not_public";
const string WRONG_SCOPE_VIOLATION =
    "org.omg.mof:reflective.wrong_scope";
const string WRONG_MULTIPLICITY_VIOLATION =
    "org.omg.mof:reflective.wrong_multiplicity";
const string WRONG_TYPE_VIOLATION =
    "org.omg.mof:reflective.wrong_type";
const string WRONG_NUMBER_PARAMETERS_VIOLATION =
    "org.omg.mof:reflective.wrong_number_parameters";
const string INVALID_DELETION_VIOLATION =
    “org.omg.mof:reflective.invalid_deletion”;
```

Other Exception

There is one exception to this — when an Operation defined in the metamodel raises an Exception that is also defined in the metamodel; see below.

The OtherException exception is raised when a call to “refInvokeOperation” results in an error condition that corresponds to an M2-level Exception defined for the Operation in the metamodel.

```
exception OtherException {
    DesignatorType exception_designator;
    ValuesType exception_args;
};
```

The arguments to the OtherException exception are as follows:

- **exception_designator** gives the designator for the M2-level Exception raised.
- **exception_args** is an ordered list of CORBA Any values that represent the arguments of the Exception raised. The encoding of this field is defined in the specification of the “refInvokeOperation” on page 263.

NOTE: When an error condition could be expressed as either a Reflective error or a Structural error, the latter takes precedence. For example, if one end of Link in a call to “refAddLink” is a nil object reference, this should be signalled as “Nil Object” rather than “Wrong Type.”

9.5 Preconditions for IDL Generation

The IDL mapping may not produce valid CORBA IDL if any of the following preconditions on the input meta-model is not satisfied:

© ISO/IEC 2005 - All rights reserved

- The MOF Model constraints, as defined above, must all be satisfied for the input meta-model.
- The input meta-model must be structurally consistent.
- The visible names within a NameSpace must conform to the standard CORBA IDL identifier syntax:

  "An identifier is an arbitrarily long sequence of ASCII alphabetic, digit and underscore ("_") characters. The firstd character must be an ASCII alphabetic character."

Names of Model Elements that have a valid "idl_substitute_name" Tag are excepted from this precondition; (see 9.6.2.1, "Substitute Name," on page 196).

NOTE: No such requirement applies to Model Elements such as Tags whose names are not visible in the IDL mapping. However, for these "invisible" elements it is advisable to use a naming convention that minimizes the risk of name collision within the Namespace itself.

- The visible ModelElement names must map to contextually unique IDL identifiers after name substitution (see 9.6.2.1, "Substitute Name," on page 196), the application of the Format1, Format2 or Format3 name rewriting algorithms (see 9.7.1, "Generated IDL Identifiers," on page 198) and other name mangling as specified in the mapping templates.
- An AliasType, CollectionType, or StructureType may not depend on itself via a chain of 'IsOfType' and 'Contains' links between DataType or StructureField instances. For example, MOF DataTypes that map to IDL recursive data types are not supported.
- A nested Package may not be used as a subtype or supertype.
- A nested Package may not import or be imported by another Package.
- The following interim visibility definitions and constraints apply to the IDL mapping:
  - A ModelElement is visible to another ModelElement only if the former has visibility of "public_vis."
  - A ModelElement declared within another top-level Package is visible within a top-level Package only if the former Package is imported, clustered, or inherited by the latter Package.
  - One ModelElement can only depend on another (in the sense of the M2-level DependsOn Association) if the latter is visible from the former within the definition of visibility immediately above.
- After name substitution (see "Substitute Name" on page 196), the name of an Import must equal the name of its "importedNamespace."
- A Class may not be nested within another Class.
- A Class may not be imported.
- If a Constraint is contained by a DataType or Operation, its name must also be unique in the DataType or Operation's container Namespace.
- Model Elements in a meta-model cannot be cyclically dependent except as follows:
  - A dependency cycle consisting of one or more Classes is legal, provided they all have the same container.
  - A dependency cycle consisting of one or more Classes and one or more DataTypes or Exceptions, is legal provided they all have the same container.

NOTE: This precludes circular importing and circular clustering. It also precludes recursion between "pure" DataTypes. (The two exceptions correspond to cases that can be expressed in OMG IDL using forward interface declarations.)

© ISO/IEC 2005 - All rights reserved

CORBA 2.3 adds an additional IDL restriction: “The name of an interface or a module may not be redefined within the immediate scope of the interface of the module.” For example:

```
module M {
    typedef short M;      // Error: M is the name of the module
                          // in the scope of which the typedef is
    interface I {
        void i (in short j);
                          // Error: i clashes with the interface
    };
};
```

The IDL templates in this specification do not contain any patterns of this form. However, poor choice of names in a meta-model may generate IDL that violates this restriction. In particular, the same name should not be used for both a container and its contents. For example, a Package should not have the same name as one of its Classes, DataTypes, or Associations. A Class should not have the same name as one of its Attributes or References. An Association should not have the same name as one of its AssociationEnds.

9.6 Standard Tags for the IDL Mapping

This sub clause defines the standard Tags that apply to the Model to IDL mapping. Other Tags may be attached to the elements of a meta-model, but the meaning of these Tags is not specified. Similarly, this sub clause does not specify the meaning of the Tags below in contexts apart from the Model to IDL mapping.

All standard Tag identifiers for the IDL mapping start with the prefix string:

“org.omg.mof.idl_”

The notation used below for defining the Tags is described in Table 7.5 on page 137.

NOTE: Many of the IDL mapping Tags significantly alter the interface signatures of the generated IDL. It is prudent for an IDL generator to only respect IDL mapping Tags when they are contained within the respective meta-model. Otherwise, it may not be possible to determine which Tags were in effect when the meta-data server was generated. This would make it hard for a client to infer the meaning of generated IDL at runtime. It would also make problems for automatic server and client generators.

9.6.1 Tags for Specifying IDL #pragma directives

9.6.1.1 IDL Prefix

This tag allows the meta-modeler to specify the CORBA Interface Repository Identifier prefix for the generated IDL. This is essential when a MOF meta-model is used as the authoritative source for IDL for some other OMG standard.

tag id:	“org.omg.mof.idl_prefix”
attaches to:	Model::Package
values:	one String
meaning:	This tag supplies a RepositoryId prefix that is used for the entire module generated for the Package.

© ISO/IEC 2005 - All rights reserved

tag id:	"org.omg.mof.idl_prefix"
idl generation:	A #pragma prefix is inserted into the IDL before the "module" declaration for the Package.
restrictions:	[1] A Prefix tag should only be attached to a non-nested Package. [2] A Prefix tag should have a value that is a valid OMG IDL prefix, consisting of ASCII letters, digits, underscore ('_'), hyphen ('-') and period ('.') as specified in the CORBA Core specification. [3] A Prefix tag contained by a Package takes precedence over one that is not contained.

9.6.1.2 IDL Version

When a MOF metamodel is modified it will often result in generated IDL that has the same module and interface names but different interface signatures. In such cases, it is strictly necessary to use different IDL version numbers for all types, interfaces, and exceptions whose signatures have changed. In MOF 1.4, this can be done by attaching an Version tag to the appropriate model elements.

tag id:	"org.omg.mof.idl_version"
attaches to:	Model::Package, Model::Class, Model::Association, Model::Attribute, Model::Operation, Model::Reference, Model::StructureType, Model::AliasType, Model::CollectionType, Model::EnumerationType, Model::Exception, Model::Constant, Model::Constraint
values:	one String
meaning:	This tag supplies a version number that is used for selected IDL declarations corresponding to the tagged element.
idl generation:	A #pragma version is inserted into or following selected IDL declaration for modules, interfaces, data types, constants, and exceptions generated from the tagged model element. Version tags on Attributes, Operations, and References result in version tags for the corresponding IDL operations. Refer to the respective templates for details.
restrictions:	[1] A Version tag value must have the form "<major>.<minor>" where <major> and <minor> are unsigned 16 bit decimal integers. [2] It is not meaningful to attach a Version tag to an AssociationEnd, Import, Parameter, Tag, StructureField, or PrimitiveType.

9.6.2 Tags for Providing Substitute Identifiers

There are some situations when the IDL identifiers produced by the IDL mapping templates will result in name collisions. The following tag allows a meta-modeler to provide a substitute for a model element's name that will be used in IDL generation.

© ISO/IEC 2005 - All rights reserved

9.6.2.1 Substitute Name

tag id:	"org.omg.mof.idl_substitute_name"
attaches to:	Model::ModelElement
values:	one String
meaning:	The value is a name to be used in place of the model element's name.
idl generation:	Wherever the IDL mapping makes use of a model element's name, the substitute name will be used in its place. This substitution occurs before the application of Format1, Format2, or Format3 rewriting and other name mangling.
restrictions:	The preconditions described in "Preconditions for IDL Generation" on page 192 apply to the substitute name. For example: [1] The identifier formatting rules must produce a syntactically valid OMG IDL identifier from the value. [2] All identifiers produced from it must be unique in their respective scopes after formatting and name mangling, as per the IDL mapping specification. [3] There can be at most one Substitute Name tag for any given ModelElement.

9.6.3 Tags for Specifying IDL Inheritance

The following tags allow the meta-modeler to specify that a generated interface inherits from one or more additional IDL interfaces. These tags allow the definition of MOF-based meta-models that are upwards compatible with pre-existing meta-data interfaces expressed in CORBA IDL.

9.6.3.1 Instance Supertypes

tag id:	"org.omg.mof.idl_instance_supertypes"
attaches to:	Model::Class
values:	one or more Strings (order is significant)
meaning:	The values give the fully qualified OMG IDL identifiers for additional interfaces that the "instance" interface for this Class should inherit from.
idl generation:	The specified interfaces are added to the "instance" interface's inheritance list following the other supertypes defined by the templates. They appear in the order given.
restrictions:	[1] The values must be fully qualified identifiers for OMG IDL interfaces. [2] There can be at most one Instance Supertypes tag per Class.

© ISO/IEC 2005 - All rights reserved

9.6.3.2 Instance Supertypes

tag id:	"org.omg.mof.idl_instance_supertypes"
attaches to:	Model::Class
values:	one or more Strings (order is significant)
meaning:	The values give the fully qualified OMG IDL identifiers for additional interfaces that the "instance" interface for this Class should inherit from.
idl generation:	The specified interfaces are added to the "instance" interface's inheritance list following the other supertypes defined by the templates. They appear in the order given.
restrictions:	[1] The values must be fully qualified identifiers for OMG IDL interfaces. [2] There can be at most one Instance Supertypes tag per Class.

9.6.3.3 Class Proxy Supertypes

tag id:	"org.omg.mof.idl_class_proxy_supertypes"
attaches to:	Model::Class
values:	one or more Strings (order is significant)
meaning:	The values give the fully qualified OMG IDL identifiers for additional interfaces that the "class proxy" interface for this Class should inherit from.
idl generation:	The specified interfaces are added to the "class proxy" interface's inheritance list following the other supertypes defined by the templates. They appear in the order given.
restrictions:	[1] The values must be fully qualified identifiers for OMG IDL interfaces. [2] There can be at most one Class Proxy Supertypes tag per Class.

9.6.3.4 Association Supertypes

tag id:	"org.omg.mof.idl_association_supertypes"
attaches to:	Model::Association
values:	One or more Strings (order is significant).
meaning:	The values give the fully qualified OMG IDL identifiers for additional interfaces that the interface for this Association should inherit from.
idl generation:	The specified interfaces are added to the "association" interface's inheritance list following the other supertypes defined by the templates. They appear in the order given.
restrictions:	[1] The values must be fully qualified identifiers for OMG IDL interfaces. [2] There can be at most one Association Supertypes tag per Association.

© ISO/IEC 2005 - All rights reserved

9.6.3.5 Package Supertypes

tag id:	“org.omg.mof.idl_package_supertypes”
attaches to:	Model::Package
values:	One or more Strings (order is significant).
meaning:	The values give the fully qualified OMG IDL identifiers for additional interfaces that the interface for this Package should inherit from.
idl generation:	The specified interfaces are added to the “package” interface’s inheritance list following the other supertypes defined by the templates. They appear in the order given.
restrictions:	[1] The values must be fully qualified identifiers for OMG IDL interfaces. [2] There can be at most one Package Supertypes tag per Package.

9.7 Generated IDL Issues

During the design of the MOF Model to IDL mapping, several design decisions were made that are explained in this sub clause.

9.7.1 Generated IDL Identifiers

Identifier naming is an important issue for automatically generated IDL, especially when that IDL is intended to be used by applications written by human programmers. The mapping has to reach a balance between conflicting requirements:

- Syntactic correctness - all identifiers in the mapped IDL must conform to the defined CORBA IDL syntax, and they must all conform to the CORBA scoping and upper/lower casing restrictions.
- User friendliness - identifiers should convey as much information as possible without being overly long.
- Conformance to existing conventions - identifiers should conform to existing stylistic conventions.

The OMG conventions for IDL identifiers (see “OMG IDL Style Guide: ab/98-06-03”) are based on the notion that an identifier is formed from one or more words in some natural language. The conventions allow digits to be used in words and take account of acronyms. The Style Guide then specifies three different styles for putting some words together as an identifier. In particular:

- Identifiers for IDL module, interface, and types are capitalized. If the name consists of multiple words, each word is capitalized in the identifier.
- Identifiers for IDL operations, attributes, formal parameters, struct, and exception members are all lower-case. If the name consists of multiple words, the words are separated by underscores (“_”) in the identifier.
- Identifiers for IDL constant and enumerator names are all upper-case. If the name consists of multiple words, the words are separated by underscores (“_”) in the identifier.

9.7.1.1 Rules for splitting MOF Model::ModelElement names into "words"

The MOF Model represents the “name” of a ModelElement using the MOF String data type; that is, as UTF-16 strings. The IDL mapping typically needs to convert these names (or alternates provided using an “idl_substitute_name” tag) into OMG IDL identifiers for use in a variety of IDL contexts.

© ISO/IEC 2005 - All rights reserved

Since the MOF Model (like the UML meta-model) does not restrict ModelElement name strings, not all names can be mapped to legal OMG IDL identifiers. For example, names that include graphic characters or accented letters do not map to IDL identifiers.

Names that are subject to mapping must consist only of ASCII letters, digits, hyphens ('-'), underscores ('_') and white-space characters, and must conform to the following syntax:

```
word       ::=    [A-Z][A-Z0-9]*[a-z0-9]*
                | [a-z][a-z0-9]*
white-space::=  SP, CR, LF, HT, VT
non-sig    ::=    { '_' | '-' | white-space }*
identifier::=    [non-sig] word { non-sig word }* [non-sig]
```

The above syntax defines a heuristic for splitting names into a sequence of words. They can then be reassembled into OMG IDL identifiers using the 3 formats below. The "non-sig" characters are non-significant and are discarded.

NOTE: The behavior of the IDL mapping for names that do not match the above syntax is not specified.

9.7.1.2 IDL Identifier Format 1

In Format 1, the first letter of each word is converted into upper case, and other letters remain the same case as input. The words are not separated by other characters. The table below lists some examples of Format 1 identifiers.

Name	Name split into words	Identifier in Format 1
foo	"foo"	Foo
foo_bar	"foo" "bar"	FooBar
ALPHAbeticalOrder	"ALPHAbetical" "Order"	ALPHAbeticalOrder
-a1B2c3-d4-	"a1" "B2c3" "d4"	A1B2c3D4
DSTC pty ltd	"DSTC" "pty" "ltd"	DSTCPtyLtd

Format 1 is used by the IDL mapping to produce the names of modules and interfaces.

9.7.1.3 IDL Identifier Format 2

In Format 2, all letters in each word are converted into lower case. Each word is separated by an underscore "_." The table below lists some examples of Format 2 identifiers.

Name	Name split into words	Identifier in Format 2
foo	"foo"	foo
foo_bar	"foo" "bar"	foo_bar
ALPHAbeticalOrder	"ALPHAbetical" "Order"	alphabetical_order
-a1B2c3_d4_	"a1" "B2c3" "d4"	a1_b2c3_d4
DSTC pty ltd	"DSTC" "pty" "ltd"	dstc_pty_ltd

© ISO/IEC 2005 - All rights reserved

Format 2 is used by the IDL mapping for identifiers for IDL operations, exceptions, attributes, formal parameters, exception members, and members of generated struct types.

9.7.1.4 IDL Identifier Format 3

In Format 3, all letters in each word are converted into upper case. Each word is separated by an underscore "_." The table below lists some examples of Format 3 identifiers.

Name	Name split into words	Identifier in Format 3
foo	"foo"	FOO
foo_bar	"foo" "bar"	FOO_BAR
ALPHAbeticalOrder	"ALPHAbetical" "Order"	ALPHABETICAL_ORDER
-a1B2c3_d4_	"a1" "B2c3" "d4"	A1_B2C3_D4
DSTC pty ltd	"DSTC" "pty" "ltd"	DSTC_PTY_LTD

Format 3 is used by the IDL mapping for identifiers for IDL constants.

9.7.1.5 Literal String Values

Literal string values (in String valued Constants) are not subject to word splitting and reformatting. They should be output in the generated IDL as wide-string literals with character escape sequences as required to express the String value as legal IDL.

9.7.2 Generation Rules for Synthesized Collection Types

The MOF Model allows Attributes, AssociationEnds, References, and Parameters to being single-, optional-, or multi-valued depending on the ModelElement's base type and its multiplicity.

At various places in the mapped interfaces, it is necessary to pass collections that represent values for the optional- or multi-valued cases. The IDL mapping synthesizes an IDL type for representing these collections. This type is a **typedef** (i.e., an alias) for an unbounded CORBA sequence of the collection's base type, where the name of the typedef depends on the corresponding ModelElement's multiplicity specification.

For example, for an ordered unique collection, the synthesized collection type would be a "unique list" (or UList) type. The typedef name for a unique list takes the form <*ClassifierType*>UList (i.e., the name of the collection base type followed by the characters "UList"). For example, if an M2-level Operation returns an ordered, unique list of some Class called "Foo," then the IDL type for the corresponding operation's result is declared as follows:

```
interface Foo;
typedef sequence <Foo> FooUList;
```

Similar collection kind naming conventions are used for constructed data types. Thus for a non-ordered unique collection of an enumeration type, the mapping would produce the following:

```
enum SomeEnum {e1, e2};
typedef sequence <SomeEnum> SomeEnumSet;
```


© ISO/IEC 2005 - All rights reserved

There are four distinct collection type suffixes corresponding to the combinations of the “isOrdered” and “isUnique” flags for an element’s “multiplicity” attribute. The appropriate suffix should be generated whenever <*CollectionKind*> appears in the IDL templates below.

Multiplicity Flags	**Collection**	**Kind Suffix**
none	bag	Bag
ordered	list	List
unique	set	Set
ordered, unique	unique list (ordered set)	UList

Note that the MOF Model specification includes a relevant Constraint on multiplicity values; see the “MustBeUnorderedNonunique” constraint in 7.9.4, “The MOF Model Constraints,” on page 105. This states that when a feature’s multiplicity bounds are [0..1], both the “isOrdered” and “isUnique” are set to false. As a consequence, the <*CollectionKind*> suffix for a [0..1] collection type is always “Bag.”

When the base DataType maps to a built-in CORBA data type, the base name for the synthesized collection type is defined as shown in Table 9.3.

Table 9.3 - Base Names for Synthesized Collections of built-in IDL types

MOF PrimitiveType	**CORBA data type**	**Base name**
PrimitiveTypes::Boolean	boolean	Boolean
PrimitiveTypes::Integer	long	Long
PrimitiveTypes::Long	long long	LongLong
PrimitiveTypes::Float	float	Float
PrimitiveTypes::Double	double	Double
PrimitiveTypes::String	wstring	WString
CorbaIdlTypes::CorbaShort	short	Short
CorbaIdlTypes::CorbaUnsignedShort	unsigned short	UShort
CorbaIdlTypes::CorbaUnsignedLong	unsigned long	ULong
CorbaIdlTypes::CorbaUnsignedLongLong	unsigned long long	ULongLong
CorbaIdlTypes::CorbaLongDouble	long double	LongDouble
CorbaIdlTypes::CorbaOctet	octet	Octet
CorbaIdlTypes::CorbaChar	char	Char
CorbaIdlTypes::CorbaString	string	String
CorbaIdlTypes::CorbaWChar	wchar	WChar

© ISO/IEC 2005 - All rights reserved

Declarations for the collection types appear following the IDL declaration for a DataType or the forward IDL declaration for a Class. Declarations for synthesized collection types for technology neutral and CORBA specific primitive data types appear in the “PrimitiveTypes” and “CorbaIdlTypes” modules respectively. The corresponding IDL files will typically be “#included” as required.

Operations produced by the IDL mapping with collection parameters must ensure that the sequence values supplied and returned have an appropriate number of elements. When collection parameters are sets or unique lists, operations must also ensure that the sequence values contain no duplicates.

9.7.3 IDL Identifier Qualification

To avoid scoping errors within the mapped IDL, identifier names must be either fully qualified, or partially qualified to an appropriate level. This specification leaves the choice between the use of fully or partially qualified identifiers to the implementor.

9.7.4 File Organization and #include statements

This specification does not prescribe how the generated IDL is organized into files. Therefore, the generation rules do not contain any “#include” statements. An implementor must decide how to organize the generated IDL into files, and must generate appropriate “#include” statements to ensure that the resultant IDL can compile. Similarly, the implementor must generate “#ifndef” guards as required by the OMG style rules.

9.8 IDL Mapping Templates

Model specific IDL is produced by traversing the containment hierarchy of a top-level M2-level Package. The CORBA module structure of the resulting IDL directly reflects the containment hierarchy of the source Package. If element X contains element Y in the source model, then the IDL corresponding to X will have the IDL corresponding to Y embedded in it (assuming that IDL is produced for Y).

The IDL mapping supports the containment hierarchy for ModelElements as described in 7.3.4, “The MOF Model Containment Hierarchy,” on page 40, except as stated in 9.5, “Preconditions for IDL Generation,” on page 192. Further restrictions on meta-models that can be successfully mapped are described in the same sub clause.

The mapping rules are described in terms of IDL templates. Each Template describes the maximum IDL that could be generated when mapping MOF Model objects. In any specific case, the actual IDL generated will depend on the properties of the corresponding MOF Model object.

Throughout the following Template descriptions, the IDL is said to be “generated by” the Templates. Clearly the Templates do not generate IDL in a literal sense. Instead, the reader should imagine that each Template is a parameter to a hypothetical generator function. When it is called with the appropriate kind of MOF ModelElement object as a second parameter, the function “elaborates” the template to produce an appropriate fragment of CORBA IDL. A similar “elaboration” process gives the required semantics for the IDL from the descriptions following the templates and the specifications given earlier in 9.3, “Computational Semantics for the IDL Mapping,” on page 165.

NOTE: The Template approach used here is a notational convenience, not a required or suggested implementation strategy.

© ISO/IEC 2005 - All rights reserved

9.8.1 Template Notation

The following table is a guide to interpreting the IDL generation templates.

Appearance (by example)	Meaning
typedef	The literal characters in bold font should be generated.
<*AttributeType*>	The characters should be substituted for the described identifier using Identifier Format 1. The <> do not appear in the generated IDL.
<*attribute_name*>	The characters should be substituted for the described identifier using the Identifier Format 2. The <> do not appear in the generated IDL.
<*CONSTANT_NAME*>	The characters should be substituted for the described identifier using the Identifier Format 3. The <> do not appear in the generated IDL.
<*CONSTANTVALUE*>	The characters should be substituted for the described identifier without formatting (i.e., as is). Typically, these are literal values. The <> do not appear in the generated IDL.
<<XYZ TEMPLATE>>	Apply the named template. The <<>> do not appear in the generated IDL.
some phrase **,** . . .	The ellipsis characters ". . ." following the "**,**" indicate that this generates a comma separated list of "some phrase." It is implicit that there is no comma at the end of the list.
[some phrase]	The square bracket characters "[]" surrounding a phrase in a template indicate that the phrase may or may not be required, depending on context.
// for each parameter	Gives the rules on when and how to perform the IDL generation, or some general commentary on the process. The rules themselves do not appear in the generated IDL.

9.8.2 Package Module Template

This sub clause describes the rules for mapping a MOF Package object to a CORBA IDL module as expressed in the Package Module Template.

The Package Module Template generates a CORBA IDL module that contains the IDL for each of the M2-level Constants, DataTypes, Exceptions, Constraints, Imports, Classes, and Associations in an M2-level Package. It also contains the IDL for the M1-level Package and Package Factory interfaces, and type declarations for various collection types. Most of this is defined in subsidiary templates. IDL generation is suppressed if the Package "visibility" is not "public_vis."

Template

```
<<ANNOTATION TEMPLATE>>
// if this Package has visibility of private or protected, no IDL is
// generated for it
module <PackageName> {
// if this Package has an idl_version Tag
#pragma version <PackageName> <version>
```

© ISO/IEC 2005 - All rights reserved

```
interface <PackageName>Package;    // forward declaration

// for each Class contained in the Package
<<CLASS FORWARD DECLARATION TEMPLATE>>

// for each Package, DataType, Exception, Class, Association, Constraint,
// and Constant contained by the Package, generate the appropriate IDL
<<PACKAGE MODULE TEMPLATE>>
<<DATATYPE TEMPLATE>>
<<EXCEPTION TEMPLATE>>
<<CLASS TEMPLATE>>
<<ASSOCIATION TEMPLATE>>
<<CONSTRAINT TEMPLATE>>
<<CONSTANT TEMPLATE>>

// Generate the Package Factory interface
<<PACKAGE FACTORY TEMPLATE>>

// Generate the Package interface
<<PACKAGE TEMPLATE>>

}; // end of module <PackageName>
```

Description

The Package Module Template starts by rendering the M2-level Package's "annotation" attribute as a comment using the Annotation Template. This is followed by the IDL module header for the Package's module. The module name is *<PackageName>*.

The template generates forward declarations for some IDL interfaces. First, it forward declares the M1-level Package interface, giving it the name *<PackageName>***Package**. Then, it forward declares the Class proxy and Instance interfaces for all M2-level Classes in the current M2-level Package's "contents" using the template defined in 9.8.5, "Class Forward Declaration Template," on page 209.

Next, IDL must be generated for the current M2-level Package's "contents" as follows:

- For nested Packages, use the template defined in 9.8.2, "Package Module Template," on page 203.
- For Classes, use the template defined in 9.8.6, "Class Template," on page 209.
- For Associations, use the template defined in 9.8.10, "Association Template," on page 214.
- For Constants, use the template defined in 9.8.13, "Operation Template," on page 240.
- For Exceptions, use the template defined in 9.8.14, "Exception Template," on page 242.
- For DataTypes, use the template defined in 9.8.15, "DataType Template," on page 243.

© ISO/IEC 2005 - All rights reserved

- For Constraints, use the template defined in 9.8.16, “Constraint Template,” on page 245.

The IDL for the contained ModelElements must be generated in an order that reflects their dependencies. For example, the IDL for a DataType should appear before the IDL for other ModelElements that use it.

Finally, the Package Module Template generates the Package Factory and Package interfaces for the current M2-level Package using the templates respectively defined in 9.8.3, “Package Factory Template,” on page 205 and 9.8.4, “Package Template,” on page 206.

9.8.3 Package Factory Template

The Package Factory Template defines the IDL generation rules for the Package Factory interface; see 9.2.1.1, “Package objects and Package Factory objects,” on page 162 and 9.2.2, “The Meta Object Interface Hierarchy,” on page 163.

A Package Factory interface is generated for top-level M2 Packages only. The interface is named <*PackageName*>**PackageFactory** and it contains a single “factory” operation, as described below.

Template

```
// if the this Package is top-level
interface <PackageName>PackageFactory
{
// if this Package has an idl_version Tag
#pragma version <PackageName>PackageFactory <version>

<PackageName>Package create_<package_name>_package (
    // for each non-derived class-level Attribute of any directly or
    // indirectly contained Class within this Package and its closure
    // under Package generalization and clustering.
    in <AttributeType>[<CollectionKind>]
        <qualified_attribute_name>, ...
    )
raises (Reflective::MofError);
};
```

IDL Supertypes

none

Operations

create_<package_name>_package

The “create_<package_name>_package” operation creates a new Package object that is an instance of this M2-level Package.	
reflective analog:	none

© ISO/IEC 2005 - All rights reserved

return type:	<PackageName>Package
parameters:	<qualified_attribute_name> : in <AttributeType>[<CollectionKind>], ...
exceptions:	MofError (Overflow, Underflow, Duplicate)

The parameters for “create_<package_name>_package” give the initial values for any non-derived classifier-scoped Attributes for all Classes that belong to this M2-level Package’s extent.

As Attributes in different Classes can have the same name, the parameter name <*qualified_attribute_name*> is qualified relative to the Package (e.g., “class1_attribute1”).

When the Attribute multiplicity is not [1..1], the <*AttributeType*> has an appropriate CollectionKind suffix appended; see 9.7.1.5, “Literal String Values,” on page 200.

The parameters are declared in a sequence defined by a recursive depth-first traversal of the Package’s ancestors clusters and components, visiting a Package’s supertypes before its contents. The following ordering rules apply:

1. A Package’s supertype Packages are processed before the “contents” of the Package.
2. The supertype Packages are processed in the order defined by the “Generalizes” association.
3. Classes, Imports (with “isClustered” set to true) and nested Packages within a Package are processed in the order of the “Contains” association.
4. A Class’s superclasses are processed before the “contents” of the Class.
5. Any Class superclasses are processed in the order of the “Generalizes” association.
6. An Import with “isClustered” set to true is processed by processing the clustered Package.
7. Attributes within a Class are processed in the order of the “Contains” association.
8. When an Attribute is encountered that has already been encountered during the traversal, generation of another initialization parameter is suppressed.

The MofError exception can be raised if there is a Structural, Constraint, or Semantic error. In particular, “Overflow,” “Underflow,” and “Duplicate” occur if an Attribute initialization parameter does not conform to the respective Attribute’s multiplicity specification.

9.8.4 Package Template

The Package Factory Template defines the IDL generation rules for the Package interface; see 9.2.1.1, “Package objects and Package Factory objects,” on page 162 and 9.2.2, “The Meta Object Interface Hierarchy,” on page 163.

A Package interface is named <*PackageName*>**Package** and it contains read-only IDL attributes giving the dependent Package, Association, and Class proxy objects for a Package object.

Template

```
interface <PackageName>Package :
   // if Package has no super-Packages
```


© ISO/IEC 2005 - All rights reserved

```
    Reflective::RefPackage
    // else for each public super-Package (in order)
    <SuperPackage>Package, ...
    // if Package has a "Package Supertypes" Tag
    //     for each supertype defined by the Tag (in order)
    , <PackageSupertypeName>, ...
{
    // if this Package has an idl_version Tag
    #pragma version <PackageName>Package <version>

    // for each Package for an Import where:
    //    is_clustered == true and
    //    Import.visibility == public and
    //    importedNamespace.visibility == public
    readonly attribute <ClusteredPackageName>Package
       <clustered_package_name>_ref;
    // for each public contained Package
    readonly attribute <NestedPackageName>Package
          <nested_package_name>_ref;
    // for each public contained Class
    readonly attribute <ClassName>Class <class_name>_ref;
    // for each public contained Association
    readonly attribute <AssociationName> <association_name>_ref;
     };
```

Supertypes

If the M2-level Package inherits from other M2-level Packages with "visibility" of "public_vis," the Package interface inherits from the interfaces corresponding super-Packages. Otherwise, the Package interface inherits from **Reflective::RefPackage**.

If the M2-level Package has a "Package Supertypes" Tag (see 9.6.3, "Tags for Specifying IDL Inheritance," on page 196), the generated Package interface also inherits from the IDL interfaces specified by the Tag.

Attributes

clustered_package_name>_ref

An attribute of this form is generated for each public clustered Package of the current M2-level Package. The attribute is generated if and only if:

1. the Import's "isClustered" flag is true,
2. the Import's "visibility" is "public_vis,"
3. the Import's "importedNamespace" is a Package, and

© ISO/IEC 2005 - All rights reserved

4. the clustered Package has “visibility” of “public_vis.”

The attribute holds the object reference for the M1-level Package’s M1-level clustered Package object.

reflective analog:	ref_package_ref(<*clustered_package_designator*>); 10.2.5, “Reflective::RefPackage (abstract),” on page 269.
type:	<*ClusteredPackageName*>**Package**
multiplicity:	exactly one
changeable:	no

<nested_package_name>_ref

An attribute of this form is generated for each nested Package in the current M2-level Package whose “visibility” is “public_vis.” The attribute holds the object reference for the M1-level Package’s M1-level nested Package object.

reflective analog:	ref_package_ref(<*nested_package_designator*>); 10.2.5, “Reflective::RefPackage (abstract),” on page 269.
type:	<*NestedPackageName*>**Package**
multiplicity:	exactly one
changeable:	no

<class_name>_ref

An attribute of this form is generated for each Class in the current Package whose “visibility” is “public_vis.” The attribute holds the object reference for the M1-level Package’s M1-level Class Proxy object.

reflective analog:	ref_class_ref(<*class_designator*>)
type:	<*ClassName*>**Class**
multiplicity:	exactly one
changeable:	no

<association_name>_ref

An attribute of this form is generated for each Association in the current Package whose “visibility” is “public_vis.” The attribute holds the object reference for the M1-level Package’s M1-level Association object.

reflective analog:	ref_package_ref(<*association_designator*>);
type:	<*AssociationName*>
multiplicity:	exactly one
changeable:	no

Operations

none

© ISO/IEC 2005 - All rights reserved

9.8.5 Class Forward Declaration Template

The Class Forward Declaration Template defines the IDL generation rules for the forward interface declarations for an M2-level Class whose “visibility” is “public_vis.” It also produces any Class collection type declarations required by the IDL of the containing Package(s).

Template

```
// if the Class has visibility of protected or private, no IDL
// is generated.

interface <ClassName>Class;
interface <ClassName>;

    typedef sequence < <ClassName> > <ClassName>Set;
        // if this Class has an idl_version Tag
        #pragma version <ClassName>Set <version>

typedef sequence < <ClassName> > <ClassName>Bag;
        // if this Class has an idl_version Tag
        #pragma version <ClassName>Bag <version>

typedef sequence < <ClassName> > <ClassName>List;
        // if this Class has an idl_version Tag
        #pragma version <ClassName>List <version>

typedef sequence < <ClassName> > <ClassName>UList;
        // if this Class has an idl_version Tag
        #pragma version <ClassName>UList <version>
```

Description

The Class Forward Declaration Template generates a forward declaration for the Instance and Class proxy interfaces for an M2-level Class. These have IDL identifiers <*ClassName*> and <*ClassName*>**Class** respectively. The synthesized collection type declarations for the Class follow the forward declarations.

9.8.6 Class Template

The Class Template defines the IDL generation rules for an M2-level Class whose “visibility” is “public_vis.” The IDL is generated within the module for the Class’s containing Package and consists of a comment followed by the complete Classes Class Proxy and Instance interfaces.

Template

```
// if the Class has visibility of protected or private, no IDL
// is generated
```

© ISO/IEC 2005 - All rights reserved

```
<<ANNOTATION TEMPLATE>>
<<CLASS PROXY TEMPLATE>>
<<INSTANCE TEMPLATE>>
```

Description

See 9.8.7, “Class Proxy Template,” on page 210 and 9.8.6, “Class Template,” on page 209.

9.8.7 Class Proxy Template

The Class Proxy Template defines the IDL generation rules for the <*ClassName*>**Class** interface for an M2-level Class whose “visibility” is “public_vis.” This interface has operations for any classifier-scoped Attributes and Operations, along with a factory operation and IDL attributes that give access to the extant Instance objects. It also contains IDL declarations corresponding to any DataTypes, Exceptions, Constants, and Constraints in the M2-level Class.

Template

```
interface <ClassName>Class :
   // if Class has no super-Classes
   Reflective::RefObject
   // else for each super-Class
   <SuperClass>Class, ...
   // if Class has a "Class Proxy Supertypes" Tag
   //     for each supertype defined by the Tag (in order)
   , <ClassProxySupertypeName>, ... {
    // if this Class has an idl_version Tag
           #pragma version <ClassName>Class <version>

    // all <ClassName> including subclasses of <ClassName>
   readonly attribute <ClassName>Set all_of_type_<class_name>;

   // if the Class is not abstract
   // all <ClassName> excluding subclasses of <ClassName>
   readonly attribute <ClassName>Set all_of_class_<class_name>;

   // for each Constant, DataType, Exception, Constraint,
   // classifier-scoped Attribute and classifier-scoped Operation
   // in the Class, generate the appropriate IDL
   <<DATATYPE TEMPLATE>>
   <<CONSTRAINT TEMPLATE>>
   <<CONSTANT TEMPLATE>>
   <<EXCEPTION TEMPLATE>>
   <<ATTRIBUTE TEMPLATE>>          // public classifier-level only
   <<OPERATION TEMPLATE>>          // public classifier-level only
```


© ISO/IEC 2005 - All rights reserved

```
    // if the Class is not abstract
    <<CLASS CREATE TEMPLATE>>

}; // end of interface <ClassName>Class
```

Supertypes

If the M2-level Class inherits from other M2-level Classes, the generated Class Proxy interface inherits from the corresponding supertype Class Proxy interfaces. Otherwise, the Class Proxy interface inherits from **Reflective::RefObject**.

If the M2-level Class has a “Class Proxy Supertypes” Tag (see 9.6.3, “Tags for Specifying IDL Inheritance,” on page 196), the generated Class Proxy interface also inherits from the IDL interfaces specified by the Tag.

Attributes

all_of_class_<class_name>

The “all_of_class_<class_name>” attribute gives all Instance objects in the current extent for the corresponding M2-level Class. The attribute is only generated if “isAbstract” is false for the Class.

reflective analog:	ref_all_objects(<*class_designator*>, false)
type:	<*ClassName*> (multiplicity zero or more, unique, non ordered)
multiplicity:	exactly one
changeable:	no

The value of this attribute mirrors the definition of Instance object lifetimes; see 9.3.4.2, “Instance object lifecycle semantics,” on page 171. It does not include any deleted Instance objects.

all_of_type_<class_name>

The “all_of_type_<class_name>” attribute gives all Instance objects in the current extent for the corresponding M2-level Class or for any M2-level subClasses.

reflective analog:	ref_all_objects(<*class_designator*>, true)
type:	<*ClassName*> (multiplicity zero or more, unique, non ordered)
multiplicity:	exactly one
changeable:	no

The value of this attribute mirrors the definition of Instance object lifetimes; see 9.3.4.2, “Instance object lifecycle semantics,” on page 171. It does not include any deleted Instance objects.

Operations

The operations for a <*ClassName*>**Class** interface are produced by the Attribute, Operation, and Class Create Templates. Note that the operations for the M2-level Classes instance-scoped features do not appear in this interface.

© ISO/IEC 2005 - All rights reserved

9.8.8 Instance Template

The Instance Template defines the IDL generation rules for the <*ClassName*> interface for an M2-level Class whose "visibility" is "public_vis." This interface contains operations for the M2-level Classes instance-scoped Attributes and Operations, along with any References.

Template

```
interface <ClassName> :
    // (The Instance interface inherits the Class Proxy interface
    // for the Class and Instance interfaces for any super-Classes)
    <ClassName>Class
    // for each super-Class of this Class (in order)
    , <SuperClassName>, ...
    // if Class has an "Instance Supertypes" Tag
    //     for each supertype defined by the Tag (in order)
    , <InstanceSupertypeName>, ... {
     // if this Class has an idl_version Tag
            #pragma version <ClassName> <version>

    // for each Attribute, Reference, Operation contained in
    // this Class, generate the appropriate IDL
    <<ATTRIBUTE TEMPLATE>> // public instance-level only
    <<REFERENCE TEMPLATE>> // public only
    <<OPERATION TEMPLATE>> // public instance-level only

}; // end of interface <ClassName>
```

Supertypes

The Instance interface for an M2-level Class inherits from the Class's Class Proxy interface, along with the Instance interfaces for all of its M2-level super-Classes.

If the M2-level Class has an "Instance Supertypes" Tag (see 9.6.3, "Tags for Specifying IDL Inheritance," on page 196), the generated Instance interface also inherits from the IDL interfaces specified by the Tag.

Attributes

none

Operations

The operations for an Instance interface are generated by the Attribute, Reference, and Operation Templates. Note that the operations for instance-scoped Attributes and Operations only appear here.

© ISO/IEC 2005 - All rights reserved

9.8.9 Class Create Template

The Class Create Template defines the IDL generation rules for the Instance factory operation for a non-abstract M2-level Class whose “visibility” is “public_vis.”

Template

```
<ClassName> create_<class_name> (
    // for each non-derived direct or inherited attribute
    in <AttributeType>[<CollectionKind>] <attribute_name>, ...
)
raises (Reflective::MofError);
```

Operations

create_<class_name>

The “create_<class_name>” operation creates new Instance objects for the M2-level Class (i.e., instances of the Class's <*ClassName*> interface).

reflective analog:	ref_create_instance(<*class_designator*>, <*attr_name*>,...)
return type:	<ClassName>
parameters:	in <*AttrTypeName*>*[*<*CollectionType*>*]* <*attr_name*>, ...
exceptions:	MofError (Overflow, Underflow, Duplicate, Composition Closure, Supertype Closure, Already Created)

The parameters to “create_<class_name>” provide initial values for the M2-level Class’s non-derived attributes. Parameter declarations are generated in an order defined by a recursive depth-first traversal of the inheritance graph. More precisely:

1. A Classes’ super-Classes are processed before the Classes’ Attributes.
2. Super-Classes are processed in the order of the “Generalizes” association.
3. The Attributes of each Class or super-Class are processed in the order of the “Contains” association.
4. When an Attribute is encountered with a “scope” value of “classifier_level” or an “isDerived” value of true no parameter is generated.
5. When an Attribute is encountered a second or subsequent time, no additional parameter is generated.

When an Attribute has multiplicity bounds other than [1..1], the type of the corresponding initial value parameter’s type will be a collection type; see 9.7.1.5, “Literal String Values,” on page 200.

“Overflow,” “Underflow,” and “Duplicate” occur if an argument that gives the initial value for an Attribute does not match the Attribute’s multiplicity specification.

“Composition Closure” occurs if the initial value for a “composite” Attribute contains an Instance object in another extent; see 8.11.2, “The Composition Closure Rule,” on page 155.

© ISO/IEC 2005 - All rights reserved

"Supertype Closure" occurs if the extent for the current object cannot create Instance objects for this super-Class; see 9.3.11, "The Supertype Closure Rule," on page 182.

"Already Created" occurs if the M2-level Class has "isSingleton" set to true, and this object's extent already includes an Instance object for the Class.

9.8.10 Association Template

The Association Template defines the generation rules for the Association interface corresponding to an M2-level Association whose "visibility" is "public_vis." This interface contains the IDL operations for accessing and updating the Association's M1-level link set. It also contains IDL declarations corresponding to any Constraints in the M2-level Association.

Template

```
// If the Association has visibility of protected or private,
// no IDL is generated

// data types for Association <AssociationName>
struct <AssociationName>Link {
    <AssociationEnd1ClassName> <associationend1_name>;
    <AssociationEnd2ClassName> <associationend2_name>;
};
// if this Association has an idl_version Tag
        #pragma version <AssociationName>Link <version>
typedef sequence < <AssociationName>Link >
    <AssociationName>LinkSet;
// if this Association has an idl_version Tag
        #pragma version <AssociationName>LinkSet <version>

<<ANNOTATION TEMPLATE>>

interface <AssociationName> : Reflective::RefAssociation
    // if Association has an "Association Supertypes" Tag
    //     for each supertype defined by the Tag (in order)
    , <AssociationSupertypeName>, ... {
     // if this Association has an idl_version Tag
            #pragma version <AssociationName> <version>

     // for each Constraint in the Association, generate the IDL
     <<CONSTRAINT TEMPLATE>>

    // list of associated elements
    <AssociationName>LinkSet all_<association_name>_links ()
```


© ISO/IEC 2005 - All rights reserved

```
    raises (Reflective::MofError);

boolean exists (
    in <AssociationEnd1Class> <association_end1_name>,
    in <AssociationEnd2Class> <association_end2_name>)
    raises (Reflective::MofError);

// if association_end1 is_navigable
<AssociationEnd1Class>[<CollectionKind>] <association_end1_name> (
    in <AssociationEnd2Class> <association_end2_name>)
    raises (Reflective::MofError);

// if association_end2 is_navigable
<AssociationEnd2Class>[<CollectionKind>] <association_end2_name> (
    in <AssociationEnd1Class> <association_end1_name>)
    raises (Reflective::MofError);

// if association_end1 is_changeable
// and association_end2 is_changeable
void add (
    in <AssociationEnd1Class> <association_end1_name>,
    in <AssociationEnd2Class> <association_end2_name>)
    raises (Reflective::MofError);

// if association_end1 is_changeable and is_navigable
// and association_end2 is_changeable
// and association_end1 has upper > 1 and is_ordered
void add_before_<association_end1_name> (
    in <AssociationEnd1Class> <association_end1_name>,
    in <AssociationEnd2Class> <association_end2_name>,
    in <AssociationEnd1Class> before)
    raises (Reflective::NotFound, Reflective::MofError);

// if association_end1 is_changeable
// and association_end2 is_changeable and is_navigable
// and association_end2 has upper > 1 and is_ordered
void add_before_<association_end2_name> (
    in <AssociationEnd1Class> <association_end1_name>,
    in <AssociationEnd2Class> <association_end2_name>,
    in <AssociationEnd2Class> before)
    raises (Reflective::NotFound, Reflective::MofError);
```

© ISO/IEC 2005 - All rights reserved

```
// if association_end1 is_navigable and is_changeable
void modify_<association_end1_name> (
    in <AssociationEnd1Class> <association_end1_name>,
    in <AssociationEnd2Class> <association_end2_name>,
    in <AssociationEnd1Class> new_<association_end1_name>)
    raises (Reflective::NotFound, Reflective::MofError);
// if association_end2 is_navigable and is_changeable
void modify_<association_end2_name> (
    in <AssociationEnd1Class> <association_end1_name>,
    in <AssociationEnd2Class> <association_end2_name>,
    in <AssociationEnd2Class> new_<association_end2_name>)
    raises (Reflective::NotFound, Reflective::MofError);

// if association_end1 is_changeable
// and association_end2 is_changeable
void remove (
    in <AssociationEnd1Class> <association_end1_name>,
    in <AssociationEnd2Class> <association_end2_name>)
    raises (Reflective::NotFound, Reflective::MofError);
};
```

Data Types

The Association Template generates data type declarations that are used to pass a link set for the M2-level Association. The <*AssociationName*>**Link** and <*AssociationName*>**LinkSet** type declarations precede the Association interface declaration.

Supertypes

Every generated Association interface inherits from **Reflective::RefAssociation**. If the M2-level Association has an “Association Supertypes” Tag (see 9.6.3, “Tags for Specifying IDL Inheritance,” on page 196), the generated Association interface also inherits from the IDL interfaces specified by the Tag.

Attributes

none

Operations

all_<association_name>_links

© ISO/IEC 2005 - All rights reserved

The “all_<association_name>_links” operation creates new Instance objects for the M2-level Class (i.e., instances of the Class’s <*ClassName*> interface).

reflective analog:	ref_all_links()
return type:	<AssociationName>**LinkSet**
parameters:	none
query:	yes
exceptions:	MofError()

The “all_<association_name>_links” operation returns the current link set for this Association expressed using the <AssociationName>**LinkSet** type.

While the definitions in 8.9.2.1, “A Mathematical Model of Association State,” on page 150 state that an ordered Association implies a partial ordering over the LinkSet, the result of the “all_<association_name>_links” operation is defined to be a Set. A client should not draw any conclusions from the ordering of the returned links.

The operation’s signature raises **Reflective::MofError** to allow Constraint error and Semantic error conditions to be signalled.

exists

The “exists” operation queries whether a link currently exists between a given pair of Instance objects in the current M1-level Association extent.

reflective analog:	ref_link_exists(Link{<*assoc_end1_name*>, <*assoc_end2_name*>})
return type:	boolean
parameters:	in <*AssocEnd1ClassName*> <*assoc_end1_name*> in <*AssocEnd2ClassName*> <*assoc_end2_name*>
query:	yes
exceptions:	MofError (Invalid Object, Nil Object, Inaccessible Object)

The parameters to the “exists” operation are a pair of Instance values of the appropriate type for the Association. Since MOF link relationships are implicitly directional, the order of the parameters is significant.

“Invalid Object,” “Nil Object,” and “Inaccessible Object” occurs if either of the parameters is a non-existent, nil, or inaccessible Instance object.

<association_end1_name>

© ISO/IEC 2005 - All rights reserved

The “<association_end1_name>” operation queries the Instance object or objects that are related to a particular Instance object by a link in the current M1-level Association extent. When “isNavigable” is set to false for the AssociationEnd, the “<association_end1_name>” operation is suppressed.

reflective analog:	ref_query(*<assoc_end1_designator>*, *<assoc_end1_name>*)
return type:	*<AssocEnd2ClassName>[<CollectionType>]*
parameters:	in *<AssocEnd1ClassName>* *<assoc_end1_name>*
query:	yes
exceptions:	MofError (Invalid Object, Nil Object, Inaccessible Object, Underflow)

The *<association_end1_name>* parameter is the Instance object from which the caller wants to “navigate.” “Invalid Object,” “Nil Object,” and “Inaccessible Object” occur when the parameter is a non-existent, nil object or inaccessible Instance object.

The result type of the operation depends on the multiplicity of *<AssociationEnd2>*. If it has bounds of [1..1], the result type is the Instance type corresponding to the AssociationEnd’s “type.” Otherwise, it is a collection of the same Instance type, as described in “Literal String Values” on page 200.

“Underflow” occurs when *<AssociationEnd2>* has bounds [0..1] or [1..1] and the Instance object given by the parameter is not related in the current Association extent. It should not occur in other cases where the result type is a collection type. (If there is a multiplicity underflow, it is signalled by returning a collection value with too few elements as opposed to raising an exception.)

The result of the operation gives the object or collection of objects related to the parameter object by a Link or Links in this Association. If the multiplicity is [1..1], the result will be a non-nil object reference. If the multiplicity is [0..1], the operation will return a non-nil object reference if there is a related object, and a nil object reference otherwise. In other cases, the operation will return a (possibly zero length) sequence of non-nil object references. If the association end being queried is ordered, the order of the contents of the returned collection is significant.

<association_end2_name>

This operation is the equivalent of “<association_end1_name>,” with the “end1” and “end2” interchanged.

add

The “add” operation creates a link in this Association between a pair of Instance objects. When “isChangeable” is set to false for either of the M2-level Association’s AssociationEnd, the “add” operation is suppressed.

reflective analog:	ref_add_link(Link{*<assoc_end1_name>*, *<assoc_end2_name>*})
return type:	none
parameters:	in *<AssocEnd1ClassName>* *<assoc_end1_name>* in *<AssocEnd2ClassName>* *<assoc_end2_name>*
exceptions:	MofError (Invalid Object, Nil Object, Inaccessible Object, Overflow, Duplicate, Reference Closure, Composition Closure, Composition Cycle)

© ISO/IEC 2005 - All rights reserved

The two parameters to the “add” operation give the Instance objects at the two ends of the new link. “Invalid Object,” “Nil Object,” and “Inaccessible Object” occur if either of the parameter values is a non-existent, nil or inaccessible Instance object.

If one or other end of the Association has “isOrdered” set to true, the new link must be added so that it is the last member of the projection for the ordered AssociationEnd. The operation must also preserve ordering of the existing members of the ordered projection.

“Overflow” occurs when adding the new link would cause the size of the projection of either the first or second parameter object to exceed the upper bound for the opposite AssociationEnd. “Duplicate” occurs when the link set for the current Association extent already contains the link whose creation is requested.

“Reference Closure” occurs when either (or both) of the AssociationEnds has References, and the corresponding Instance object parameter does not belong to the same outermost Package extent as the Association object; see 8.11.1, “The Reference Closure Rule,” on page 153.

“Composition Closure” occurs when either AssociationEnd has “aggregation” set to “composite,” and either of the Instance object parameters does not belong to the same outermost Package extent as this Association object; see 8.11.2, “The Composition Closure Rule,” on page 155.

“Composition Cycle” occurs when adding the new link would create a cycle of composite / component relationships such that one of the Instance object parameters is a (ultimately) component of itself; see 8.10.2, “Aggregation “composite”,” on page 153.

add_before_<association_end1_name>

The “add_before_<association_end1_name>” operation creates a link between a pair of Instance objects at a given place in this Association. This operation is only generated when “isChangeable” is true for both AssociationEnds, and when the first AssociationEnd is multi-valued, ordered, and navigable.

reflective analog:	ref_add_link_before(Link{<*assoc_end1_name*>, <*assoc_end2_name*>}, <*assoc_end1_designator*>, before); (see 10.2.4, “Reflective::RefAssociation (abstract),” on page 265).
return type:	none
parameters:	in <*AssocEnd1ClassName*> <*assoc_end1_name*> in <*AssocEnd2ClassName*> <*assoc_end2_name*> in <*AssociationEnd1ClassName*> before
exceptions:	NotFound, MofError (Invalid Object, Nil Object, Inaccessible Object, Overflow, Duplicate, Reference Closure, Composition Closure, Composition Cycle)

The first two parameters to the “add_before_<association_end1_name>” operation give the Instance objects at the two ends of the new link. “Invalid Object,” “Nil Object,” and “Inaccessible Object” occur if either of the parameter values is a non-existent, nil, or inaccessible Instance object.

The third parameter (“before”) gives an Instance object that determines the point at which the new link is inserted. “Invalid Object,” “Nil Object,” and “Inaccessible Object” also apply to the “before” parameter value.

© ISO/IEC 2005 - All rights reserved

The “before” value should be present in the projection of the “<*assoc_end2_name*>” parameter value. If this is so, the insertion point for the new link is immediately before the “before” value, otherwise the “NotFound” error occurs.

“Overflow,” “Duplicate,” “Reference Closure,” “Composition Closure,” and “Composition Cycle” occur as described for the “add” operation above.

add_before_<association_end2_name>

This operation is the equivalent of “add_before_<association_end1_name>,” with the “end1” and “end2” interchanged.

NOTE: The preconditions for generating the “add_before_<association_end1_name>” and “add_before_<association_end2_name>” operations are such that at most one of them may appear in an Association interface.

modify_<association_end1_name>

The “modify_<association_end1_name>” operation updates a link between a pair of Instance objects, replacing the Instance at AssociationEnd1 with a new Instance object. When AssociationEnd1 has “isChangeable” or “isNavigable” set to false, this operation is suppressed.

reflective analog:	ref_modify_link(Link{<*assoc_end1_name*>, <*assoc_end2_name*>}, <*assoc_end1_designator*> new_<*assoc_end1_name*>); (see 10.2.4, “Reflective::RefAssociation (abstract),” on page 265).
return type:	none
parameters:	in <*AssocEnd1ClassName*> <*assoc_end1_name*> in <*AssocEnd2ClassName*> <*assoc_end2_name*> in <*AssocEnd2ClassName*> new_<*assoc_end1_name*>
exceptions:	NotFound, MofError (Invalid Object, Nil Object, Inaccessible Object, Overflow, Underflow, Duplicate, Reference Closure, Composition Closure, Composition Cycle)

The first two parameters to the “modify_<association_end1_name>” operation should give the Instance objects at the ends of an existing link. “Invalid Object,” “Nil Object,” and “Inaccessible Object” occur if either of the parameter values are non-existent, nil, or inaccessible Instance objects. “NotFound” occurs if the link does not exist in the current extent.

The third parameter (“new_<*assoc_end1_name*>”) gives the Instance object that is to replace the Instance at AssociationEnd1 for the selected link. “Invalid Object,” “Nil Object,” and “Inaccessible Object” also occurs if this parameter’s value is a non-existent, nil, or inaccessible Instance object.

If the “<*assoc_end1_name*>” and “new_<*assoc_end1_name*>” parameters give the same Instance object, this operation is required to have no effect on the Association’s link set.

NOTE: The following error conditions apply to the state of the M1-level Association after the completion of the operation, not to any intermediate states.

“Underflow” occurs if completion of the operation would leave the M1-level Association in a state where

```
size(Projection(<assoc_end1_name>)) less than <AssocEnd2>.lower
```

“Overflow” occurs if completion of the operation would leave the M1-level Association in a state where

© ISO/IEC 2005 - All rights reserved

```
size(Projection(new_<assoc_end1_name>)) greater than
    <AssocEnd2>.upper
```

Note that the “Underflow” condition for the “new_<*assoc_end1_name*>” Instance should be treated as a deferred constraint.

“Duplicate” occurs if the operation would create a duplicate link in this M1-level Association extent. Similarly, “Composition Cycle” occurs if the operation creates a link that (on completion of the operation) would make the “<*assoc_end2_name*>” or “new_<*assoc_end1_name*>” objects components of themselves.

“Reference Closure” and “Composition Closure” occur if the operation would create a link that violates the corresponding closure rules; see 8.11.1, “The Reference Closure Rule,” on page 153 and 8.11.2, “The Composition Closure Rule,” on page 155.

- If either AssociationEnd has “isOrdered” set to true, this operation must preserve ordering of the remaining members in the relevant projections of the ordered end.

In addition:

- If AssociationEnd1 is ordered, the projection of “<*assoc_end2_name*>” must have “new_<*assoc_end1_name*>” in the position taken by “<*assoc_end1_name*>.”
- If AssociationEnd2 is ordered, the projection of “new_<*assoc_end1_name*>” must have “<*assoc_end2_name*>” as the last member.

modify_<association_end2_name>

This operation is the equivalent of “modify_<association_end1_name>” with the “end1” and “end2” interchanged.

remove

The “remove” operation removes a link between a pair of Instance objects in the current Association extent. When either AssociationEnd or AssociationEnd2 has “isChangeable” set to false, the “remove” operation is suppressed.

reflective analog:	ref_remove_link(Link{<*assoc_end1_name*>, <*assoc_end2_name*>}); (see 10.2.4, “Reflective::RefAssociation (abstract),” on page 265).
return type:	none
parameters:	in <*AssocEnd1ClassName*> <*assoc_end1_name*> in <*AssocEnd2ClassName*> <*assoc_end2_name*>
exceptions:	NotFound, MofError (Nil Object, Underflow)

The two parameters to this operation give the Instance objects at both ends of the link that is to be removed from the current Association object’s link set. “Nil Object” occurs if either parameter value is a nil object reference.

“NotFound” occurs if the link to be deleted does not exist in the current Association extent.

NOTE: “Invalid Object” and “Inaccessible Object” does occur here. The “remove” operation needs to be capable of deleting links that involve Instance objects that have been deleted or that are inaccessible. In the latter case, an implementation can usually fall back on local comparison of object references. If that fails (e.g., because there are multiple references for an Instance object), the implementation will typically be unable to distinguish the case from “NotFound.”

© ISO/IEC 2005 - All rights reserved

"Underflow" occurs if deleting the link would cause the size of the projection of either the "*<assoc_end1_name>*" or "*<assoc_end2_name>*" parameter value to be less than the corresponding "lower" bound.

If either AssociationEnd1 or AssociationEnd2 has "isOrdered" set to true, the "remove" operation must preserve the ordering of the remaining members of the corresponding projection.

9.8.11 Attribute Template

The Attribute Template defines the generation rules for M2-level Attributes whose "visibility" is "public_vis." The Attribute Template declares operations to query and update the value of an Attribute. These operations appear on different interfaces, depending on the Attribute's "scope."

- IDL operations for instance-scoped Attributes appear in the Instance ("*<ClassName>*") interface for the Attribute's containing Class.
- IDL operations for classifier-scoped Attributes appear in the Class Proxy ("*<ClassName>***Class**") interface for the Attribute's containing Class, and are inherited by the Instance interface.

The operations generated for an Attribute and their signatures depend heavily on the Attribute's properties. For the purposes of defining the generated IDL, Attribute multiplicities fall into three groups:

- single-valued Attributes: multiplicity bounds are [1..1],
- optional-valued Attributes: multiplicity bounds are [0..1], and
- multi-valued Attributes: any other multiplicity.

Template

```
// if Attribute visibility is private or protected no IDL
// is generated
<<ANNOTATION TEMPLATE>>

// if lower = 0 and upper = 1
<AttributeType> <attribute_name> ()
   raises (Reflective::NotSet, Reflective::MofError);
// if this Attribute has an idl_version Tag
       #pragma version <attribute_name> <version>

// if lower = 1 and upper = 1
<AttributeType> <attribute_name> ()
   raises (Reflective::MofError);
// if this Attribute has an idl_version Tag
       #pragma version <attribute_name> <version>

// if upper > 1
<AttributeType><CollectionKind> <attribute_name> ()
   raises (Reflective::MofError);
```


© ISO/IEC 2005 - All rights reserved

```
// if this Attribute has an idl_version Tag
       #pragma version <attribute_name> <version>

// if upper = 1 and is_changeable
void set_<attribute_name> (in <AttributeType> new_value)
   raises (Reflective::MofError);
// if this Attribute has an idl_version Tag
       #pragma version set_<attribute_name> <version>

// if upper > 1 and is_changeable
void set_<attribute_name> (
      in <AttributeType><CollectionKind> new_value)
   raises (Reflective::MofError);
// if this Attribute has an idl_version Tag
       #pragma version set_<attribute_name> <version>

// if lower = 0 and upper = 1 and is_changeable
void unset_<attribute_name> ()
   raises (Reflective::MofError);
// if this Attribute has an idl_version Tag
       #pragma version unset_<attribute_name> <version>

// if upper > 1 and is_changeable
void add_<attribute_name> (in <AttributeType> new_element)
   raises (Reflective::MofError);
// if this Attribute has an idl_version Tag
       #pragma version add_<attribute_name> <version>

// if upper > 1 and is_changeable and is_ordered
void add_<attribute_name>_before (
   in <AttributeType> new_element,
   in <AttributeType> before_element)
   raises (Reflective::NotFound, Reflective::MofError);
// if this Attribute has an idl_version Tag
       #pragma version add_<attribute_name>_before <version>

// if upper > 1 and is_changeable and is_ordered and not is_unique
void add_<attribute_name>_at (
   in <AttributeType> new_element,
   in unsigned long position)
   raises (Reflective::BadPosition, Reflective::MofError);
```

© ISO/IEC 2005 - All rights reserved

```
// if this Attribute has an idl_version Tag
       #pragma version add_<attribute_name>_at <version>

// if upper > 1 and is_changeable
void modify_<attribute_name> (
   in <AttributeType> old_element,
   in <AttributeType> new_element)
   raises (Reflective::NotFound, Reflective::MofError);
// if this Attribute has an idl_version Tag
       #pragma version modify_<attribute_name> <version>

// if upper > 1 and is_changeable and is_ordered and not is_unique
void modify_<attribute_name>_at (
       in <AttributeType> new_element,
          in unsigned long position)
   raises (Reflective::BadPosition, Reflective::MofError);
// if this Attribute has an idl_version Tag
       #pragma version modify_<attribute_name>_at <version>

// if upper > 1 and upper != lower and is_changeable
void remove_<attribute_name> (
       in <AttributeType> old_element)
   raises (Reflective::NotFound, Reflective::MofError);
// if this Attribute has an idl_version Tag
       #pragma version remove_<attribute_name> <version>

// if upper > 1 and upper != lower and is_changeable and
// is_ordered and not is_unique
void remove_<attribute_name>_at (in unsigned long position)
   raises (Reflective::BadPosition, Reflective::MofError);
// if this Attribute has an idl_version Tag
       #pragma version remove_<attribute_name>_at <version>
```

Operations

<attribute_name>

© ISO/IEC 2005 - All rights reserved

The “<attribute_name>” operation returns the value of the named Attribute.

reflective analog:	ref_value(<*attribute_designator*>); (see 10.2.3, “Reflective::RefObject (abstract),” on page 254).
return type:	[0..1] - <*AttributeType*> [1..1] - <*AttributeType*> other - <*AttributeType*><*CollectionKind*>
parameters:	none
query:	yes
exceptions:	[0..1] - Unset, MofError [1..1] - MofError other - MofError

The signature of the “<attribute_name>” operation depends on the Attribute’s multiplicity as indicated above. Its behavior is as follows:

- In the [0..1] case, the operation either returns the Attribute’s optional value, or raises the NotSet exception to indicate that the optional value is not present.
- In the [1..1] case, the operation simply returns the Attribute’s single value.
- In other cases, the operation returns the Attribute’s collection value. In the case where the collection is empty the result value will be a sequence with length zero. No exception is raised in this case.

If the Attribute is instance-scoped, the operation is only available on Instance objects, and invoking it returns a value that is related to this Instance object. If the Attribute is classifier-scoped, the operation can be invoked on both Class Proxy and Instance objects. In both cases, the operation returns a value that is related to all Instances for the Attribute’s Class in the current extent. For a more detailed comparison of classifier versus instance-scoped Attributes, see 8.6, “Semantics of Attributes,” on page 141.

The MofError exception may be raised to signal meta-model defined Constraint errors and implementation specific Semantic errors. However, an implementation generally should avoid doing this, for the reasons given in 8.13.6, “Access operations should avoid raising Constraint exceptions,” on page 159.

set_<attribute_name>

The “set_<attribute_name>” operation sets the value of the named Attribute.

reflective analog:	ref_set_value(<*attribute_designator*>, new_value); (see 10.2.3, “Reflective::RefObject (abstract),” on page 254).

© ISO/IEC 2005 - All rights reserved

return type:	none
parameters:	[0..1] - in <*AttributeType*> new_value [1..1] - in <*AttributeType*> new_value other - in <*AttributeType*><*CollectionKind*> new_value
exceptions:	[0..1] - MofError (Invalid Object, Inaccessible Object, Composition Closure, Composition Cycle) [1..1] - MofError (Invalid Object, Inaccessible Object, Composition Closure, Composition Cycle) other - MofError (Overflow, Underflow, Duplicate, Invalid Object, Inaccessible Object, Composition Closure, Composition Cycle)

The signature of the "set_<attribute_name>" operation depends on the Attribute's multiplicity as indicated above. Its behavior is as follows:

- In the single and optional-valued cases, the operation assigns the single value given by "new_value" to the named Attribute.
- In the multi-valued case, the operation assigns the collection value given by "new_value" parameter to the named Attribute.

When the Attribute has a lower bound of 0, its value can legally be empty:

- In the optional-valued case, the Attribute's value is set to "empty" by invoking the "unset_<attribute_name>" operation described below.
- In the [0..*N*] case (where *N* is not 1), the Attribute's value is set to empty by invoking "set_<attribute_name>" with a zero length sequence as the parameter.

"Composition Closure" and "Composition Cycle" are only possible when the type of the Attribute is a Class, and the Attribute has "composite" aggregation semantics:

- "Composition Closure" occurs when "new_value" or one of its members (in the multi-valued case) belongs to a different outermost Package extent to this object.
- "Composition Cycle" occurs when the operation would result in this object having itself as a direct or indirect component.

"Overflow," "Underflow," and "Duplicate" can only occur in the case of a multi-valued Attribute:

- "Overflow" occurs if the number of members in the "new_value" collection is greater than the Attribute's upper bound.
- "Underflow" occurs if the number of members in the "new_value" collection is less than the Attribute's lower bound.
- "Duplicate" occurs if the Attribute has "isUnique" set to true and the "new_value" collection contains duplicate values.

"Invalid Object" and "Inaccessible Object" occur when some Instance object is found to be non-existent or inaccessible. An implementation should only signal one of these conditions when it prevents other consistency checking (e.g., testing for composition cycles).

© ISO/IEC 2005 - All rights reserved

unset_<attribute_name>

The “unset_<attribute_name>” operation sets the value of an optional-valued Attribute to empty. This operation is suppressed in the single-valued and multi-valued cases.

reflective analog:	ref_unset_value(<*attribute_designator*>); (see 10.2.3, “Reflective::RefObject (abstract),” on page 254).
return type:	none
parameters:	none
exceptions:	MofError

The “unset_<attribute_name>” operation is the only way to update an optional-valued Attribute to the “empty” state.

The MofError exception may be raised to signal meta-model defined Constraint errors and implementation specific Semantic errors.

add_<attribute_name>

The “add_<attribute_name>” operation updates a multi-valued Attribute by adding a new member value to its collection value. This operation is suppressed for optional and single-valued Attributes and for Attributes with “isChangeable” set to false.

reflective analog:	ref_add_value(<*attribute_designator*>, new_element); (see 10.2.3, “Reflective::RefObject (abstract),” on page 254).
return type:	none
parameters:	in <*AttributeType*> new_element
exceptions:	MofError (Overflow, Duplicate, Invalid Object, Inaccessible Object, Composition Closure, Composition Cycle)

The “add_<attribute_name>” operation adds “new_element” to the collection of a changeable multi-valued Attribute. If the Attribute’s “multiplicity” has “isOrdered” set to true, the “new_element” is added after that current last element of the collection.

“Overflow” occurs if adding another element to the collection makes the number of elements it contains greater than the Attribute’s upper bound.

“Duplicate” occurs if the Attribute’s “multiplicity” has “isOrdered” set to true, and the “new_element” value is equal to an element of the Attribute’s current value.

“Composition Closure” and “Composition Cycle” are only possible when the type of the Attribute is a Class, and the Attribute has “composite” aggregation semantics:

- “Composition Closure” occurs when “new_element” belongs to a different outermost Package extent to this object.
- “Composition Cycle” occurs when the operation would result in this object being a direct or indirect component of itself.

© ISO/IEC 2005 - All rights reserved

“Invalid Object” and “Inaccessible Object” occur when some Instance Object is found to be non-existent or inaccessible. An implementation should only signal one of these conditions when it prevents other consistency checking (e.g., testing for composition cycles).

add_<attribute_name>_before

The “add_<attribute_name>_before” operation updates a multi-valued Attribute by adding a new element at given position in its current collection value. The operation is suppressed for optional and single-valued Attributes, and for Attributes with “isChangeable” or “isOrdered” set to false.

reflective analog:	ref_add_value_before(<*attribute_designator*>, new_element, before_element); (see 10.2.3, “Reflective::RefObject (abstract),” on page 254).
return type:	none
parameters:	in <*AttributeType*> new_element in <*AttributeType*> before_element
exceptions:	NotFound, MofError (Overflow, Duplicate, Invalid Object, Inaccessible Object, Composition Closure, Composition Cycle)

The “add_<attribute_name>_before” operation adds “new_element” to the collection at a given place within the collection value of an ordered, changeable, multi-valued Attribute. Insertion is required to preserve the initial order of the collection’s elements.

The “new_element” is inserted before the first occurrence in the Attribute’s collection of the value supplied as the “before_element” parameter (i.e., the occurrence with the smallest index). “NotFound” occurs when the “before_element” value is not present in the collection.

“Overflow,” “Duplicate,” “Composition Closure,” and “Composition Cycle” occur in equivalent situations to those for “add_<attribute_name>” above.

“Invalid Object” and “Inaccessible Object” occur when some Instance object is found to be non-existent or inaccessible. An implementation should only signal one of these conditions when it prevents other consistency checking (e.g., testing for composition cycles).

add_<attribute_name>_at

The “add_<attribute_name>_at” operation updates a multi-valued Attribute by adding a new element at a given position in its current collection value. It is provided for non-unique Attributes where an insertion point must be specified using an index. The operation is suppressed for optional and single-valued Attributes, for Attributes with “isChangeable” or “isOrdered” set to false, and for Attributes with “isUnique” set to true.

reflective analog:	ref_add_value_at(<*attribute_designator*>, new_element, position); (see 10.2.3, “Reflective::RefObject (abstract),” on page 254).

© ISO/IEC 2005 - All rights reserved

return type:	none
parameters:	in <*AttributeType*> new_element in unsigned long position
exceptions:	BadPosition, MofError (Overflow, Duplicate, Invalid Object, Inaccessible Object, Composition Closure, Composition Cycle).

The “add_<attribute_name>_at” operation adds “new_element” at a given point within the collection value of an ordered, non-unique changeable, multi-valued Attribute. Insertion is required to preserve the initial order of the collection’s elements.

The insertion point is given by the value of the “position” parameter. This is the index of the collection member before which “new_element” should be inserted, with zero being the index for the first member. A “position” value equal to the current number of elements means that “new_element” should be added to the end of the collection. “BadPosition” occurs when the “position” value is greater than the number of elements in the collection. (It is not possible to create a collection value with “gaps” by adding elements with “position” values larger that the collection size.)

“Overflow,” “Duplicate,” “Composition Closure,” and “Composition Cycle” occur in equivalent situations to those described for “add_<attribute_name>” above.

“Invalid Object” and “Inaccessible Object” occur when some Instance object is found to be non-existent or inaccessible. An implementation should only signal one of these conditions when it prevents other consistency checking (e.g., testing for composition cycles).

modify_<attribute_name>

The “modify_<attribute_name>” operation updates a multi-valued Attribute by replacing an existing member of its collection value. This operation is suppressed for optional and single-valued Attributes and for Attributes with “isChangeable” set to false.

reflective analog:	ref_modify_value(<*attribute_designator*>, old_element, new_element); (see 10.2.3, “Reflective::RefObject (abstract),” on page 254).
return type:	none
parameters:	in <*AttributeType*> old_element in <*AttributeType*> new_element
exceptions:	NotFound, MofError (Duplicate, Invalid Object, Inaccessible Object, Composition Closure, Composition Cycle)

The “modify_<attribute_name>” operation replaces an occurrence of the value passed in the “old_element” parameter with the value of “new_element.” “NotFound” occurs if the “old_element” value is not present in the Attribute’s initial collection value.

If the Attribute has “isOrdered” set to true, the operation is required to preserve the initial order of the collection’s elements. If it also has “isUnique” set to false, then the operation is defined to replace the first occurrence (i.e., the one with the smallest index).

© ISO/IEC 2005 - All rights reserved

“Duplicate,” “Composition Closure,” and “Composition Cycle” occur in similar situations to those described for “add_<attribute_name>” above.

“Invalid Object” and “Inaccessible Object” occur when some Instance object is found to be non-existent or inaccessible. An implementation should only signal one of these conditions when it prevents other consistency checking (e.g., testing for composition cycles).

modify_<attribute_name>_at

The “modify_<attribute_name>_at” operation updates a multi-valued Attribute by replacing a member of its collection value at a given position. It is provided for non-unique Attributes where the member to be modified must be specified using an index. This operation is suppressed for optional and single-valued Attributes and for Attributes with “isChangeable” set to false.

reflective analog:	ref_modify_value_at(<*attribute_designator*>, new_element, position); (see 10.2.3, “Reflective::RefObject (abstract),” on page 254).
return type:	none
parameters:	in <*AttributeType*> new_element in unsigned long position
exceptions:	BadPosition, MofError (Duplicate, Invalid Object, Inaccessible Object, Composition Closure, Composition Cycle)

The “modify_<attribute_name>_at” operation replaces the value whose index in the collection is given by the “position” parameter. “BadPosition” occurs if the “position” parameter is greater than or equal to the number of elements in the Attribute collection.

The replacement value is given by the “new_value” parameter. The operation is required to preserve the order of the collection’s elements.

“Duplicate,” “Composition Closure,” and “Composition Cycle” occur in similar situations to those described for “add_<attribute_name>” above.

“Invalid Object” and “Inaccessible Object” occur when some Instance object is found to be non-existent or inaccessible. An implementation should only signal one of these conditions when it prevents other consistency checking (e.g., testing for composition cycles).

remove_<attribute_name>

The “remove_<attribute_name>” operation removes an existing member from a multi-valued Attribute. This operation is suppressed for optional and single-valued Attributes and for Attributes with “isChangeable” set to false. It is also suppressed when the lower and upper bounds are equal.

reflective analog:	ref_remove_value(<*attribute_designator*>, old_element); (see 10.2.3, “Reflective::RefObject (abstract),” on page 254).

© ISO/IEC 2005 - All rights reserved

return type:	none
parameters:	in <*AttributeType*> old_element
exceptions:	NotFound, MofError(Underflow)

The “remove_<attribute_name>” operation removes an occurrence of the value passed in the “old_element” parameter. “NotFound” occurs if the “old_element” value is not present in the Attribute’s collection value.

If the Attribute has “isOrdered” set to true, the operation is required to preserve the initial order of the collection’s elements. If it also has “isUnique” set to false, then the operation is defined to remove the first occurrence (i.e., the one with the smallest index).

“Underflow” occurs if removing an element makes the number of elements in the collection less than the Attribute’s lower bound.

NOTE: The “remove_<attribute_name>” operation should not signal an exception if it finds that some Instance object is non-existent or inaccessible. If the object in question is the object to be removed from the Attribute, it should be removed. Otherwise, the condition should be silently ignored.

remove_<attribute_name>_at

The “remove_<attribute_name>_at” operation removes the member at a given position from a multi-valued Attribute. This operation is suppressed for optional and single-valued Attributes, and for Attributes with “isChangeable” or “isOrdered” set to false or “isUnique” set to true. It is also suppressed when the lower and upper bounds are equal.

reflective analog:	ref_remove_value_at(<*attribute_designator*>, position); (see 10.2.3, “Reflective::RefObject (abstract),” on page 254).
return type:	none
parameters:	in unsigned long position
exceptions:	BadPosition, MofError(Underflow)

The “remove_<attribute_name>_at” operation removes the element of an Attribute’s collection value whose (zero based) index is given by the “position” parameter. “BadPosition” occurs if the “position” value is greater than or equal to the number of elements in the Attribute’s collection value.

“Underflow” occurs if removing an element makes the number of elements in the collection less than the Attribute’s lower bound.

9.8.12 Reference Template

The Reference Template defines the IDL generation rules for a Reference whose “visibility” is “public_vis.” The IDL generated for a Reference is declared within the scope of <*ClassName*>**Class** interface definition. The IDL generated by the Reference Template provides the operations to return the value of the Reference as well as operations to modify it. The IDL generated is dependent upon the multiplicity, mutability, and ordering of the specified Reference.

The Reference Template defines the IDL generation rules for References. It declares operations on the Instance interface to query and update links in the Association object for the current extent.

© ISO/IEC 2005 - All rights reserved

The operations generated for a Reference and their signatures depend heavily on the properties of the referenced AssociationEnd, which are also mirrored on the Reference itself. For the purposes of defining the generated IDL, Reference multiplicities fall into three groups:

- single-valued References: multiplicity bounds are [1..1],
- optional-valued References: multiplicity bounds are [0..1], and
- multi-valued References: any other multiplicity.

The generated operations for a Reference are designed to have similar signatures and behaviors to those for an instance-scoped Attribute with the same multiplicity and changeability settings.

NOTE: Recall that Reference is only well formed if the referenced AssociationEnd has “isNavigable” set to true. Similarly, a Reference’s “isChangeable” can only be true if the referenced AssociationEnd’s “isChangeable” is also true.

Template

```
// If the Reference has visibility of protected or private, no IDL
// is generated

<<ANNOTATION TEMPLATE>>

// operations to return the Reference value
// if lower = 0 and upper = 1
<ReferenceClass> <reference_name> ()
   raises (Reflective::NotSet, Reflective::MofError);
// if this Reference has an idl_version Tag
       #pragma version <reference_name> <version>

// if lower = 1 and upper = 1
<ReferenceClass> <reference_name> ()
   raises (Reflective::MofError);
// if this Reference has an idl_version Tag
       #pragma version <reference_name> <version>

// if upper > 1
<ReferenceClass><Multiplicity> <reference_name> ()
   raises (Reflective::MofError);
// if this Reference has an idl_version Tag
       #pragma version <reference_name> <version>

// operations to modify the Reference value
// if upper = 1 and is_changeable
void set_<reference_name> (
   in <ReferenceClass> new_value)
```


© ISO/IEC 2005 - All rights reserved

```
    raises (Reflective::MofError);
// if this Reference has an idl_version Tag
        #pragma version set_<reference_name> <version>

// if upper > 1 and is_changeable
void set_<reference_name> (
    in <ReferenceClass><Multiplicity> new_value)
    raises (Reflective::MofError);
// if this Reference has an idl_version Tag
        #pragma version set_<reference_name> <version>

// if lower = 0 and upper = 1 and is_changeable
void unset_<reference_name> ()
    raises (Reflective::MofError);
// if this Reference has an idl_version Tag
        #pragma version unset_<reference_name> <version>

// if upper > 1 and is_changeable
void add_<reference_name> (
    in <ReferenceClass> new_element)
    raises (Reflective::MofError);
// if this Reference has an idl_version Tag
        #pragma version add_<reference_name> <version>

// if upper > 1 and is_changeable and is_ordered
void add_<reference_name>_before (
    in <ReferenceClass> new_element,
    in <ReferenceClass> before_element)
    raises (Reflective::NotFound, Reflective::MofError);
// if this Reference has an idl_version Tag
        #pragma version add_<reference_name>_before <version>

// if upper > 1 and is_changeable
void modify_<reference_name> (
    in <ReferenceClass> old_element,
    in <ReferenceClass> new_element)
    raises (Reflective::NotFound, Reflective::MofError);
// if this Reference has an idl_version Tag
        #pragma version modify_<reference_name> <version>

// if upper > 1 and lower != upper and is_changeble
```

© ISO/IEC 2005 - All rights reserved

```
void remove_<reference_name> (
    in <ReferenceClass> old_element)
    raises (Reflective::NotFound, Reflective::MofError);
// if this Reference has an idl_version Tag
        #pragma version remove_<reference_name> <version>
```

<reference_name>

The "<reference_name>" operation reads the value of Reference. The signature of the operation depends on the multiplicity of the Reference.

reflective analog:	ref_value(<*reference_designator*>); (see 10.2.3, "Reflective::RefObject (abstract)," on page 254).
return type:	[0..1] - <*ReferenceClass*> [1..1] - <*ReferenceClass*> other - <*ReferenceClass*><*CollectionKind*>
parameters:	none
exceptions:	[0..1] - NotSet, MofError [1..1] - MofError (Underflow) other - MofError

The "<reference_name>" operation's signature is determined by the multiplicity of the Reference, and hence the referenced AssociationEnd, as shown above.

In each case, the operation calculates and returns the projection of "this" object in the link set of the referenced AssociationEnd's Association for the current outermost extent:

- In the [0..1] case, the operation returns the projected Instance object if there is one, and raises the Reflective::NotSet exception if there is not.
- In the [1..1] case, the operation normally returns a single Instance object. However, if the projection contains no elements, this is signalled as a Reflective::MofError exception with "error_kind" of "Underflow."
- In other cases, the operation returns the projection using a sequence value. If the projection is empty, the result is a sequence of length zero. If it contains fewer elements than the Reference's lower bound, those that it does contain are returned.

NOTE: Under no circumstances should the "<reference_name>" operation return a nil object reference or a sequence that includes a nil object reference.

set_<reference_name>

The "set_<reference_name>" operation assigns a new value to a Reference. The signature of the operation depends on the multiplicity of the Reference. If "isChangeable" is set to false for the Reference, this operation is suppressed.

© ISO/IEC 2005 - All rights reserved

reflective analog:	ref_set_value(<*reference_designator*>, new_value); (See 10.2.3, “Reflective::RefObject (abstract),” on page 254.)
return type:	none
parameters:	[0..1] - in <*ReferenceClass*> new_value [1..1] - in <*ReferenceClass*> new_value other - in <*ReferenceClass*><*CollectionKind*> new_value
exceptions:	[0..1] - MofError (Overflow, Underflow, Invalid Object, Nil Object, Inaccessible Object, Composition Closure, Composition Cycle, Reference Closure) [1..1] - MofError (Overflow, Underflow, Invalid Object, Nil Object, Inaccessible Object, Composition Closure, Composition Cycle, Reference Closure) other - MofError (Overflow, Underflow, Duplicate, Nil Object, Inaccessible Object, Invalid Object, Composition Closure, Composition Cycle, Reference Closure)

The “set_<reference_name>” operation’s signature is determined by the multiplicity of the Reference, and hence the referenced AssociationEnd, as shown above.

In each case, the operation replaces the set of links in the extent for the referenced AssociationEnd’s Association. The behavior is as follows:

- In the [0..1] and [1..1] case, the caller passes a single Instance object in the “new_value” parameter that is used to create the replacement link.
- In other cases, the “new_value” parameter is a sequence of Instance objects that are used to create the replacement links. If the sequence is empty, no replacement links will be created.

The projection for an optional-valued Reference can only be set to “empty” using the “unset_<reference_name>” operation; see below.

The ordering semantics of the “set_<reference_name>” operation depend on the setting of “isOrdered” in the “multiplicity” for the Reference’s “referencedEnd” and “exposedEnd” AssociationEnds:

- If neither of the AssociationEnds has “isOrdered” set to true, the Association has no ordering semantics.
- If the “referencedEnd” AssociationEnd has “isOrdered” set to true, the order of the elements of the projection of “this” Instance after the operation has completed must be the same as the order of the elements of the “new_value” parameter.
- If the “exposedEnd” AssociationEnd has “isOrdered” set to true, the order of the elements of the “new_value” parameter (if it is multi-valued) are irrelevant. Instead, the operation is required to preserve the ordering of the projections that contained “this” object, both before and after the update, as follows:
  - If “this” object is in a projection of some other Instance object before the operation but not afterwards, the order of

© ISO/IEC 2005 - All rights reserved

the projection must be preserved, with “this” object removed.

- If “this” object is in a projection of some other Instance object after the operation but not before, the order of the projection must be preserved, *and* “this” object must be added at the end of the projection.
- If “this” object is in a projection of some other Instance object both before *and* after the operation, the “before” and “after” versions of the projection must be identical.

- It is impossible for both of the AssociationEnds to have “isOrdered” set to true.

A large number of error conditions can occur, depending on “new_value,” the current state of the Association and the multiplicity of the Reference’s “referencedEnd” and “exposedEnd” AssociationEnds:

- “Invalid Object,” “Nil Object,” and “Inaccessible Object” occur if any of the supplied Instance objects is a non-existent, nil, or inaccessible Instance object.
- “Overflow” can occur in two cases. First, it occurs when the “new_value” parameter contains more elements than is allowed by the “referencedEnd”’s upper bound. Second, it occurs when the projection of an element of “new_value” after completion of the operation would have more elements than is allowed by the “exposedEnd”’s upper bound.
- “Duplicate” occurs for a multi-valued Reference when the “new_value” parameter collection contains two or more occurrences of the same Instance object.
- “Underflow” can also occur in two cases. First it occurs when the “new_value” parameter contains fewer elements than is allowed by the “referencedEnd”’s lower bound. Second, it occurs when the projection of an element of “new_value” after completion of the operation would have fewer elements than is allowed by the “exposedEnd”’s lower bound, *and* fewer elements than it had before the operation commenced.
- “Reference Closure” occurs when “new_value” (in the [0..1], [1..1] case) or one of its elements (in the “other” case) belongs in a different outermost extent to “this” object.
- “Composition Closure” occurs in the same situation as “Reference Closure,” where the referenced Association has composite aggregation semantics.
- “Composition Cycle” occurs when the referenced Association has composite aggregation semantics, and the update would make “this” object a component of itself.

unset_<reference_name>

The “unset_<reference_name>” operation sets an optional-valued Reference to empty. If “isChangeable” is set to false for the Reference, or if the bounds are not [0..1], this operation is suppressed.

reflective analog:	ref_reset_value(<*reference_designator*>); (See 10.2.3, “Reflective::RefObject (abstract),” on page 254.)
return type:	none
parameters:	none
exceptions:	MofError

The “unset_<reference_name>” operation removes the link for this object from the link set of the referenced Association, should it exist. If no such link exists, the operation does nothing.

If the “exposedEnd” AssociationEnd has “isOrdered” set to true, the operation preserves the ordering of the projection that initially contains “this” Instance object.

© ISO/IEC 2005 - All rights reserved

add_<reference_name>

The "add_<reference_name>" operation adds an Instance object to a Reference collection. If "isChangeable" is set to false for the Reference, or the Reference's upper bound is 1, this operation is suppressed.	
reflective analog:	ref_add_value(<*reference_designator*>, new_element); (See 10.2.3, "Reflective::RefObject (abstract)," on page 254.)
return type:	none
parameters:	in <*ReferenceClass*> new_element
exceptions:	MofError (Overflow, Duplicate, Invalid Object, Nil Object, Inaccessible Object, Reference Closure, Composition Closure, Composition Cycle)

The "add_<reference_name>" operation adds the "new_element" Instance to a multi-valued Reference collection by creating a link in the corresponding Association's link set. "Invalid Object," "Nil Object," or "Inaccessible Object" occur if the "new_element" parameter is a non-existent, nil, or inaccessible Instance object.

If the "referencedEnd" AssociationEnd has "isOrdered" set to true, the new link should be created so that "new_element" is the last element of the projection of "this" object. Alternatively, if the "exposedEnd" AssociationEnd has "isOrdered" set to true, the new link should be created so that "this" object is the last element of the projection of the "new_element" object. In either case, the operation should preserve the order of other elements in the respective ordered projections.

"Overflow" occurs if the number of elements in the projections of either the "this" object or the "new_element" object already equals the respective AssociationEnd's upper bound.

"Duplicate" occurs if the operation would create a duplicate link in the link set for the referenced Association. For example, when the "new_element" value is a duplicate of a value in the current Reference collection.

"Reference Closure," "Composition Closure," and "Composition Cycle" all occur in similar situations to those described above for the "set_<reference_name>" operation.

add_<reference_name>_before

The "add_<reference_name>_before" operation adds an Instance object at a particular place in an ordered Reference collection. If "isChangeable" or "isOrdered" is set to false for the Reference, this operation is suppressed.	
reflective analog:	ref_add_value_before(<*reference_designator*>, new_element, before_element); (see 10.2.3, "Reflective::RefObject (abstract)," on page 254).
return type:	none
parameters:	in <*ReferenceClass*> new_element in <*ReferenceClass*> before_element
exceptions:	NotFound, MofError (Overflow, Duplicate, Invalid Object, Nil Object, Inaccessible Object, Reference Closure, Composition Closure, Composition Cycle).

© ISO/IEC 2005 - All rights reserved

The “add_<reference_name>_before” operation is a more specialized version of the “add_<reference_name>” operation described previously. It creates a link between “this” object and the “new_element” Instance object so that it appears in a designated place in “this” object’s projection.

The “before_element” parameter gives the Instance object in the projection of “this” before which the “new_element” object should be inserted. “Invalid Object,” “Nil Object,” and “Inaccessible Object” occur if either “new_element” or “before_element” is a non-existent, nil or inaccessible Instance object. “Not Found” occurs if “before_element” is not present in the projection of “this” object.

The new link is created such that the “new_element” object appears immediately before the “before_element” value in the projection of “this” object. Apart from this, the order of the projection’s elements is unchanged.

“Overflow,” “Duplicate,” “Reference Closure,” “Composition Closure” and “Composition Cycle” all occur in equivalent situations to those described above for the “add_<reference_name>” and “set_<reference_name>” operations.

modify_<reference_name>

The “modify_<reference_name>” operation updates a Reference collection, replacing one element with another. If the Reference is not multi-valued or its “isChangeable” multiplicity flag is set to false, this operation is suppressed.

reflective analog:	ref_modify_value(*<reference_designator>*, old_element, new_element); (see 10.2.3, “Reflective::RefObject (abstract),” on page 254).
return type:	none
parameters:	in <*ReferenceClass*> old_element in <*ReferenceClass*> new_element
exceptions:	NotFound, MofError (Underflow, Overflow, Duplicate, Invalid Object, Nil Object, Inaccessible Object, Reference Closure, Composition Closure, Composition Cycle)

The “modify_<reference_name>” operation updates the link set so that the projection of “this” object has “new_element” in place of “old_element.” The operation is notionally equivalent to either

```
<the_association>.modify_<association_end1>(
    old_element, <this>, new_element)
```

or

```
<the_association>.modify_<association_end2>(
    <this>, old_element, new_element)
```

where *<the_association>* is the current outermost extent’s M1-level Association object for the referenced M2-level Association.

The “old_element” and “new_element” parameters must both give usable Instance objects. “Invalid Object,” “Nil Object,” or “Inaccessible Object” occur if either is a non-existent, nil or inaccessible object.

The “old_element” object must be an element of the projection of “this” object; that is, a link must already exist between “this” and “old_element.” “NotFound” occurs if this is not the case. If “old_element” and “new_element” are the same Instance object, the operation is required to do nothing at all.

If the referenced Association is ordered, the operation is required to preserve ordering as follows:

© ISO/IEC 2005 - All rights reserved

- If the “referencedEnd” AssociationEnd has “isOrdered” set to true, the order of the elements in the projection of “this” object should be preserved, with “new_element” occupying the same position as “old_element” did before the update.
- If the “exposedEnd” AssociationEnd has “isOrdered” set to true, the order of the elements in the projections of “old_element” and “new_element” should be preserved, except that “this” is removed from the former projection and added to the end of the latter projection.

“Overflow” occurs when the number of elements in the projection of “new_element” would be greater than the upper bound for the “exposedEnd” AssociationEnd.

“Underflow” occurs when the number of elements in the projection of “old_element” would be decreased, and it would be less than the lower bound of the “exposedEnd” AssociationEnd. (In the case where “old_element” and “new_element” are the same object, the operation does not alter the number of elements in the projection. Hence “Overflow” cannot be signalled, even if the number of elements is less than the bound.)

“Duplicate” occurs if the “modify_<reference_name>” operation would introduce a duplicate into the projection. (Care should be taken to avoid signalling “Duplicate” in the case where “old_element” and “new_element” are the same object.)

“Reference Closure,” “Composition Closure,” and “Composition Cycle” all occur in equivalent situations to those described above for the “add_<reference_name>” and “set_<reference_name>” operations.

remove_<reference_name>

The “remove_<reference_name>” operation updates a Reference collection by removing an element. If the Reference is not multi-valued or its “isChangeable” multiplicity flag is set to false, this operation is suppressed. It is also suppressed if the Reference’s lower and upper bounds are equal.

reflective analog:	ref_remove_value(<*reference_designator*>, old_element); (see 10.2.3, “Reflective::RefObject (abstract),” on page 254).
return type:	none
parameters:	in <*ReferenceClass*> old_element
exceptions:	NotFound, MofError (Underflow)

The “remove_<reference_name>” operation updates the link set (i.e., by removing a link) so that the projection of “this” object no longer contains “old_element.” “NotFound” occurs if there is no link to be deleted.

NOTE: The “remove_<reference_name>” operation should be able to cope with removal of a link when the object at the other end of a link is non-existent or inaccessible.

If the referenced Association is ordered, the operation is required to preserve the ordering of the projection with the ordered collection value.

“Underflow” occurs when the number of elements in the projections of “old_element” and “this” would be less than the lower bounds of the respective AssociationEnds.

© ISO/IEC 2005 - All rights reserved

9.8.13 Operation Template

The Operation Template defines the IDL generation rules for M2-level Operations whose “visibility” is “public_vis.” It generates an IDL operation within the scope of an Instance or Class Proxy interface, depending on the scope of the M2-level Operation.

Template

```
// If the Operation has visibility of protected or private, no IDL
// is generated

<<ANNOTATION TEMPLATE>>

// if Operation contains no "return" Parameter
void <operation_name>(
// else
<ReturnParamType>[<CollectionKind>] <operation_name>(
   // for each contained "in", "out" or "inout" Parameter
   <direction> <ParamType>[<CollectionKind>] <param_name>, ...
    )
    raises (
     // for each Exception raised by the Operation
   <ExceptionName>, ... // (a trailing comma is required)
   Reflective::MofError);
// if this Operation has an idl_version Tag
       #pragma version <operation_name> <version>

// for each Constraint contained by this Operation
<<CONSTRAINT_TEMPLATE>>
```

<operation_name>

An “<operation_name>” operation invokes an implementation specific method to perform the behavior implied by the M2-level Operation model element.

© ISO/IEC 2005 - All rights reserved

reflective analog:	ref_invoke_operation(<*reference_designator*>, old_element); (see 10.2.3, "Reflective::RefObject (abstract)," on page 254).
return type:	no return param - void [0..1] return param - <*ParamType*>Bag <*param_name*> [1..1] return param - <*ParamType*> <*param_name*> other return param - <*ParamType*><*CollectionKind*> <param_name>
parameters:	<*direction*> <*ParamType*>[<*CollectionKind*>], ...
exceptions:	<*ExceptionName*>, ... MofError (Overflow, Underflow, Duplicate, Invalid Object, Semantic Error)

An "<operation_name>" operation invokes an implementation specific method. While the behavior of the method itself is beyond the scope of the IDL mapping, the signature of the IDL operation is defined by the mapping, along with some parameter checking semantics.

The return type for an "<operation_name>" operation is generated from the M2-level Operation's (optional) return Parameter. For example, the contained Parameter object whose "direction" attribute has the value "return_dir." The return type is as follows:

- If there is no return Parameter, the return type is "void."
- If the return Parameter has "multiplicity" bounds of "[1..1]," the return type is the "type" of the Parameter; i.e., <*ParameterType*>.
- If the return Parameter some other "multiplicity" bounds, the return type is a collection type determined by the bounds. For example, <*ParameterType*><*CollectionKind*>, as described in "Literal String Values" on page 200.

The parameter declarations for an "<operation_name>" operation are generated from the M2-level Operation's Parameter, excluding the return Parameter (if any). For each non-return Parameter of the Operation, in the defined order, the "<operation_name>" declaration has a parameter declaration consisting of the following:

- The "<*direction*>" is produced by rendering the Parameter's "direction" as "in," "out," or "inout" as appropriate.
- The "<*ParameterType*>*[*<*CollectionKind*>*]*" is produced from the Parameter's "type" and "multiplicity" as follows:
  - If the Parameter has "multiplicity" bounds of "[1..1]", the <*CollectionKind*> is omitted.
  - If the Parameter has "multiplicity" bounds other than "[1..1]", <*CollectionKind*> is generated according to 9.7.1.5, "Literal String Values," on page 200.
- The "<*parameter_name*>" is produced by rendering the Parameter's name.

The list of exceptions raised by an "<operation_name>" operation is generated from the M2-level Operation's "exceptions." The generated "raises" list consists of an appropriately qualified identifier for each M2-level Exception in the Operation's "exceptions" list, followed by the qualified identifier for the MofError exception. The "raises" list should of course be comma separated as required by the syntax for OMG IDL.

© ISO/IEC 2005 - All rights reserved

While meta-model specific error conditions should be signalled by raising exceptions corresponding to the Operation's "exceptions" list, MofError is used to signal the following structural errors relating to the values supplied by the caller for "in" and "inout" parameters.

- "Overflow" occurs when the supplied collection value for a multi-valued parameter has more elements than is allowed by the M2-level Parameter's upper bound.
- "Underflow" occurs when the supplied collection value for a multi-valued parameter has fewer elements than is allowed by the M2-level Parameter's lower bound.
- "Duplicate" occurs when a multi-valued M2-level Parameter has "isUnique" set to true, and the supplied collection value contains members that are equal according to the definitions in 8.11, "Closure Rules," on page 153.
- "Invalid Object" can occur if an Instance object typed parameter value or element is a reference to a non-existent (i.e., deleted) or inaccessible object. This condition will occur if duplicate checking finds an Instance object that it cannot test for equality. It can also occur if the semantics of the Operation require an Instance object reference to be usable.

Like all other operations that have MofError in their signature, an "<operation_name>" operation can use MofError to signal Constraint errors and Semantic errors.

If an Operation has a multi-valued 'out,' 'inout,' or 'return' Parameter, or if it can raise an Exception with a multi-valued field, its implementation shall ensure that the returned collection or collections satisfy the Parameters' multiplicity constraints. If they do not, the implementation shall use MofError to signal a Semantic Error. (Since this case is deemed to be an error in the implementation, "Overflow," "Underflow," "Duplicate," or "Invalid" Object must *not* be signalled here.)

9.8.14 Exception Template

The Exception template defines the IDL generation rules for M2-level Exceptions whose "visibility" is "public_vis."

Template

```
// If the Exception has visibility of protected or private, no IDL
// is generated

<<ANNOTATION TEMPLATE>>
exception <ExceptionName> {
   // for each Parameter
   <ParameterType>[<CollectionKind>] <parameter_name>; ...
};
// if this Exception has an idl_version Tag
       #pragma version <ExceptionName> <version>
```

Description

The generated IDL for an M2-level Exception is an IDL exception. The declaration appears within an IDL interface or module corresponding to the Exception's M2-level container. In the case of an M2-level Class, this is the Class Proxy interface so that the IDL exception is available to be raised by classifier-scoped Operations.

The fields of the IDL exception are generated from the Exception's Parameters in a way that is similar to Operation Parameters:

© ISO/IEC 2005 - All rights reserved

- An Exception Parameter whose multiplicity has a “[1..1]” bound is mapped to a field whose type is “*<ParameterType>*.”
- An Exception Parameter whose multiplicity has any other bound is mapped to a field whose type is of the form “*<ParameterType><CollectionKind>*,” generated according to the rules in 9.7.1.5, “Literal String Values,” on page 200.

Constant Template

The Constant Template defines the rules for generating IDL constant declarations from M2-level Constants.

Template

```
<<ANNOTATION TEMPLATE>>
const <ConstantType> <CONSTANT_NAME> = <CONSTANTVALUE>;
// if this Constant has an idl_version Tag
       #pragma version <CONSTANT_NAME> <version>
```

The generated IDL for an M2-level Constant is an IDL constant declaration. The IDL appears an interface or module corresponding to the Constant’s M2-level container. In the container is a Class, the declaration appears within the Class Proxy interface.

The IDL generation process needs to produce a valid IDL literal value of the appropriate type from the Constant’s “value.”

9.8.15 DataType Template

The DataType Template defines the rules for generating IDL for an M2-level DataType subtypes whose “visibility” is “public_vis.” This typically consists of an IDL type declaration for the data type, followed by one or more collection type declarations, as required.

NOTE: If the IDL mapping preconditions are strictly observed, the template will only generate IDL declarations for the DataType’s type in cases where this is appropriate.

Template

```
// If the DataType’s visibility is protected or private, no IDL
// is generated


<<ANNOTATION TEMPLATE>>
// generate the DataType’s type declaration
// if the DataType is an AliasType
typedef <type.TYPESPEC> <DatatypeName>;
// if this DataType has an idl_version Tag
       #pragma version <DatatypeName> <version>

// else if the DataType is a CollectionType
typedef sequence < <type.TYPESPEC> > <DatatypeName>;
// if this DataType has an idl_version Tag
```

© ISO/IEC 2005 - All rights reserved

```
        #pragma version <DatatypeName> <version>

// else if the DataType is an EnumerationType
enum <DatatypeName> { <label> , ... };
// if this DataType has an idl_version Tag
        #pragma version <DatatypeName> <version>

// else if the DataType is a StructureType
struct <DatatypeName> {
    // for each StructureField of the StructureType
    <StructField.type.TYPESPEC> <struct_field.name> ; ...
};
// if this DataType has an idl_version Tag
        #pragma version <DatatypeName> <version>

// else no declaration. The PrimitiveTypes do not require IDL
// declarations.  All supported PrimitiveTypes map to builtin
// CORBA IDL types.
// For each Constraint contained by this DataType
<<CONSTRAINT_TEMPLATE>>

typedef sequence < <DataTypeName> > <DataTypeName>Bag;
// if this DataType has an idl_version Tag
        #pragma version <DatatypeName>Bag <version>

typedef sequence < <DataTypeName> > <DataTypeName>Set;
// if this DataType has an idl_version Tag
        #pragma version <DatatypeName>Set <version>

typedef sequence < <DataTypeName> > <DataTypeName>List;
// if this DataType has an idl_version Tag
        #pragma version <DatatypeName>List <version>

typedef sequence < <DataTypeName> > <DataTypeName>UList;
// if this DataType has an idl_version Tag
        #pragma version <DatatypeName>UList <version>
```

Description

A DataType template generates IDL M2-level DataType subclass instances for the technology neutral and CORBA specific data types. The template produces the following declarations:

- If the DataType is a CollectionType, AliasType, StructureType, or EnumerationType, the template produces an IDL

© ISO/IEC 2005 - All rights reserved

declaration for the type. (The PrimitiveTypes that are supported for the IDL mapping do not require declarations.)

- If the DataType has Constraints, the template generates the corresponding constraint string declarations.
- Finally, the template also generates synthesized collection types according to 9.7.2, “Generation Rules for Synthesized Collection Types,” on page 200.

9.8.16 Constraint Template

The Constraint template defines the rules for generating the requisite error kind string declaration for an M2-level Constraint.

Template

```
<<ANNOTATION TEMPLATE>>
const string <CONSTRAINT_NAME> = "<constraint.string>";
// if this Constraint has an idl_version Tag
       #pragma version <CONSTRAINT_NAME> <version>
```

Description

The Constraint template generates an IDL string constant whose name is based on the M2-level Constraint name. If the Constraint is contained by an M2-level DataType or Operation, the constant declaration is generated within the scope of the Constraint container’s container. If this results in a name collision, the meta-modeler can solve the problem using a substitute name tag as described in 9.6.2.1, “Substitute Name,” on page 196.

The “*<constraint.string>*” value is generated to match the following syntax (expressed in EBNF):

```
<constraint.string> ::= [ <IDL prefix> ] ':constraint.'
          ( <container_name> '.' )+ <constraint_name>
```

The components of the error kind string value are as follows:

- If the meta-model has an IDL prefix (see “IDL Prefix” on page 194), the string starts with the value of this prefix.
- Next there is a colon (“:”) to separate the prefix from the rest of the string.
- Next there is the fixed string “constraint” to indicate the class of error, followed by a period (“.”).
- Next there are a series of Format 2 renderings of the names of the Constraint’s enclosing containers. These are separated by period (“.”) characters, and followed by another period.
- The value ends with the Format 2 rendering of the name of the Constraint itself.

9.8.17 Annotation Template

The Annotation template optionally generates IDL comments for an M2-level ModelElement’s “annotation.” This template should be regarded as indicative rather than normative.

Template

```
// Annotation comments may optionally be suppressed by the IDL
// generator
```

© ISO/IEC 2005 - All rights reserved

```
// Annotation comments may use the "/*…*/" style

/* <line 1 of the ANNOTATION>
   <line 2 of the ANNOTATION>
   . . .
   <line N of the ANNOTATION> */

// or the "//" style

// <line 1 of ANNOTATION>
// <line 2 of ANNOTATION>
// . . .
// <line N of the ANNOTATION>
```

Description

The Annotation template optionally includes the “annotation” for a ModelElement in the generated IDL as an IDL comment. It is anticipated that a vendor’s IDL generator would give some control over the way that these comments are generated. For example, allowing the user to

- suppress the comments completely,
- choose between the two styles of comments, and
- choose whether or not to respect embedded line breaks and other markup.

© ISO/IEC 2005 - All rights reserved

10 The Reflective Module

10.1 Introduction

One of the advantages of meta-objects (in the general sense) is that they allow a program to use objects without prior knowledge of the objects' interfaces. In the MOF context, an object's M2-level meta-object allows a program to "discover" the nature of any M1-level MOF object, both at a syntactic level and at a deeper level. With this information in hand, the MOF's Reflective interfaces allow a program to:

- create, update, access, navigate and invoke operations on M1-level Instance objects,
- query and update links using M1-level Association objects, and
- navigate an M1-level Package structure

without using meta-model specific interfaces.

NOTE: The functionality above is all available through the "model specific" interfaces defined by the IDL mapping described in this clause. The Reflective interfaces do not allow a program to access or update MOF objects contrary to their meta-object descriptions. For example, they cannot be used to create, access or update Attributes that do not exist, or to bypass Constraint checking.

In addition, the Reflective interfaces allow the program to:

- find a M1-level object's M2-level meta-object,
- find a MOF object's container(s) and enclosing Package(s),
- test for MOF object identity, and
- delete a MOF object.

NOTE: While many of these capabilities are correctly described as reflective, the MOF does not offer the full repertoire of reflective programming features. Since it does not define object behavior, the MOF does not define interfaces for reflective behavior modification. Even if it did, these interfaces could not be implemented in many CORBA contexts.

The CORBA Interface Repository (IR) and the Dynamic Invocation Interface (DII), provide similar capabilities in the context of a CORBA object's Interface. However, using the IR and DII for this purpose means that the user cannot make use of the richer semantic information in models defined using the MOF meta-model. For example, the IR can tell the user that the "Model::Contains" IDL interface has an operation called "exists;" however, it is only by using MOF meta-objects that the user knows that the "exists" operation tests whether one object "contains" another one.

The MOF Reflective module contains four "abstract" interfaces that are inherited by the M1-level interfaces for a model that are generated from a meta-model by the IDL mapping.

1. The Reflective::RefObject interface provides common operations for M1-level Instance objects and Class Proxy objects.
2. The Reflective::RefAssociation interface provides common operations for M1-level Association objects.
3. The Reflective::RefPackage interface provides a common operations for M1-level Package objects.
4. The Reflective::RefBaseObject interface provides common operations for all MOF objects.

© ISO/IEC 2005 - All rights reserved

Since the M2-level interfaces for the MOF Model are generated by this means, they also inherit from the Reflective interfaces.

10.2 The Reflective Interfaces

This sub clause describes the interfaces defined in the “Reflective” module. These interfaces are modeled on the interfaces that are produced by the IDL mapping. However, there are some important differences:

- Reflective operations pass the values of Attributes and References, and of the Parameters to Operations and Exceptions as CORBA Any values. The mapped versions of these operations pass the values using precise types according to the meta-model.
- Reflective operations on Associations pass Instance objects with the type RefObject. The mapped versions of these operations pass Instance objects using their true types.
- The “target” feature for a Reflective operation is passed as a “designator” parameter whose type is a MOF meta-object. In the mapped case, the target is implicit in the mapped operation name.

As stated previously, the Reflective versions of operations, which are defined in the mapped IDL, do not allow a program to violate the information and computational models implied by the meta-model definition. This includes not allowing operations that, while meaningful for a model, are not possible using the mapped interfaces. For example, while it might be meaningful to call “refSetValue” on an optional Attribute passing an "empty" argument (encoded appropriately), this is not allowed: the program must use “refSetValue.”

This sub clause consists of a sub clause that explains some common patterns that are used for encoding parameters used by many Reflective operations. The remaining four sub clauses describe each Reflective interface in turn.

10.2.1 Reflective Argument Encoding Patterns

The Reflective module make heavy use of the CORBA Any type to provide meta-model independent interfaces. This sub clause defines some common patterns used throughout the Reflective interfaces for encoding parameter values in Anys.

NOTE: It is important that the type information (expressed as CORBA TypeCodes) in the encoded Anys be precisely as specified below. In particular, collection type aliases and their names are mandatory.

If the base type of the value-defining feature is a DataType, the TypeCode in the encoded Any must be the full TypeCode for the base type. Type aliases must not be optimized away, and all optional names (e.g., of struct types, fields, and so on) must be present. (Optimization of type information in Anys should be done at the ORB level if at all.)

© ISO/IEC 2005 - All rights reserved

10.2.1.1 The Standard Value Encoding Pattern

This pattern is used for encoding complete values as CORBA Anys. It is used in most cases where a reflective operation requires or provides a complete value for an element that may be collection valued (depending on the multiplicity). Examples that use this pattern are values for Operation arguments and results, values for Exception fields and Attribute initial values in a create operation.

Table 10.1- Standard Value Encoding Pattern

Bounds	CORBA Any Encoding	Notes
[0..1]	Any(alias(seq(<*type*>, 0))) where the alias name is <*typeName*>Bag	An “optional” feature value with no elements is encoded as zero length sequence.
[1..1]	Any(<*type*>)	
others	Any(alias(seq(<*type*>, 0))) where the alias name is <*typeName*><*CollectionKind*>	A “multi-valued” feature value with no elements is encoded as a zero length sequence.

10.2.1.2 The Alternate Value Encoding Pattern

The standard pattern for encoding complete values (above) does not fit well with the IDL templates for the specific “get” and “set” operations. To improve the alignment between the reflective and specific interfaces, the following alternative pattern is used for the “refValue” and “refSet Value” operations for Attributes and References.

“The operation fetches the current value of the Attribute or Reference denoted by the “feature” argument. If this object is a Class proxy, only classifier scoped Attributes can be fetched.”

“The “refSetValue” operation assigns a new value to an Attribute or Reference for an object. The assigned value must be a single value or a collection value depending on the feature’s multiplicity.” .

Table 10.2- Alternate Value Encoding Pattern

Bounds	CORBA Any Encoding	Notes
[0..1]	Any(<*type*>)	An “optional” feature value with no elements is handled as follows: • the ref_get_value() raises Unset when the value is empty • the ref_unset_value() is used to set value to no elements
[1..1]	Any(<*type*>)	
others	Any(alias(seq(<*type*>, 0))) where the alias name is <*typeName*><*CollectionKind*>	A “multi-valued” feature value with no elements is encoded as a zero length sequence.

10.2.1.3 The Value Member Encoding Pattern

The following pattern is used in the reflective versions of the add, modify and remove operations that operate on the individual members of a multi-valued Attribute or Reference. The pattern is simply to encode the member as an Any containing an instance of the feature’s base type. For example:

Any(<*type*>)

© ISO/IEC 2005 - All rights reserved

10.2.1.4 The Link Encoding Pattern

Some of the operations in the RefAssociation interface use the “generic” Link type to pass link values; see 10.3.2, “Data Types,” on page 271. While the Link type uses RefObject rather than Any, a pattern is still required to describe the encoding.

The “generic” Link type is declared as a sequence of RefObject values with an upper bound of 2. The standard encoding of a link for a given Association is:

```
Link(<assocEnd1Type>, <assocEnd2Type>)
```

In other words, the sequence value contains precisely two elements, and the elements appear in the order of the corresponding AssociationEnds in the Association.

10.2.2 Reflective::RefBaseObject (abstract)

The RefBaseObject interface is inherited by the other three reflective interfaces. It provides common operations for testing for object identity, returning an object’s meta-object, and returning its “repository container” as required for implementing structural constraints such as the MOF’s type closure rule and composition restrictions.

Supertypes

none (root object)

Operations

refMofId

The “refMofId” operation returns this object’s permanent unique identifier string.

specific analog:	none
return type:	string
isQuery:	yes
parameters:	none
exceptions:	none

Every MOF object has a permanent, unique MOF identifier associated with it. This identifier is generated and bound to the object when it is created and cannot be changed for the lifetime of the object. The primary purpose of the MOF identifier is to serve as a label that can be compared to definitively establish an object’s identity.

A MOF implementation must ensure that no two distinct MOF objects within the extent of an outermost Package object ever have the same MOF identifier. This invariant must hold for the lifetime of the extent.

A group of outermost Package extents can only be safely federated if the respective implementations can ensure the above invariant applies across the entire federation. A federation of extents in which the invariant does not hold is not MOF compliant.

The MOF specification does not mandate a scheme for achieving this. Instead, the following approach is recommended:

1. Choose an appropriate scheme (or schemes) for allocating unique identifiers. This will depend on the nature of the federation.

© ISO/IEC 2005 - All rights reserved

2. Define a textual syntax for MOF identifier strings of the form:

   ```
   <scheme-prefix> ":" <scheme-specific-part>
   ```

   where <scheme-prefix> is either standardized elsewhere, or a vendor or user specific string that is unlikely to clash with other prefixes.

In the absence of a more appropriate identifier generation scheme, it is recommended that the following scheme based on the DCE UUID algorithm and textual encoding be used. The recommended DCE UUID-based identifier syntax is:

```
"DCE" ":" <printable-form-of-dce-uuid> [":" <decimal-digits>]
```

For example:

"DCE:d62207a2-011e-11ce-88b4-0800090b5d3e"

"DCE:d62207a2-011e-11ce-88b4-0800090b5d3e:1234"

The first case would be used when it is acceptable to generate a new DCE UUID for each MOF object. The second case might be used when the overheads of doing this are too large, or the required rate of UUID generation is too high. In this case, the UUID would denote an extent incarnation, and the suffix would be a local object sequence number for the extent incarnation does not repeat during the latter's lifetime.

refMetaObject

The "refMetaObject" operation returns the Model::ModelElement object that describes this object in its metamodel specification.

specific analog:	none
return type:	DesignatorType
isQuery:	yes
parameters:	none
exceptions:	none

If the object's meta-object is unavailable, the return value may be a CORBA nil object reference.

refItself

The "refItself" operation tests whether this object and another RefBaseObject provided as an argument are the same CORBA object.

specific analog:	none
return type:	boolean
isQuery:	yes
parameters:	otherObject : in RefBaseObject
exceptions:	MofError (Invalid Object)

"Invalid Object" occurs if the "otherObject" is not a valid object, or if it is inaccessible.

© ISO/IEC 2005 - All rights reserved

refImmediatePackage

The “RefImmediatePackage” operation returns the RefPackage object for the Package that most immediately contains or aggregates this object.

specific analog:	none
return type:	RefPackage
isQuery:	yes
parameters:	none
exceptions:	none

If this object has no containing or aggregating Package (i.e., it is the RefPackage object for an outermost Package), then the return value is a CORBA nil object reference. In complex cases where there is more than one immediate aggregating Package (see 8.8, “Extents,” on page 145 and 9.2.1, “Meta Object Type Overview,” on page 161, the return value may be any of them.

refOutermostPackage

The “refOutermostPackage” operation returns the RefPackage object for the Package that ultimately contains this object.

specific analog:	none
return type:	RefPackage
isQuery:	yes
parameters:	none
exceptions:	none

If this object is the RefPackage object for an outermost Package, then the return value is this object.

refDelete

The “refDelete” operation destroys this object, including the objects it contains directly or transitively (see 9.3.4, “Lifecycle Semantics for the IDL Mapping,” on page 170 and 8.10, “Aggregation Semantics,” on page 152).

specific analog:	none
return type:	none
parameters:	none
exceptions:	MofError (Invalid Deletion)

The semantics of this operation depend on this RefBaseObject’s most derived type; see 9.2.1, “Meta Object Type Overview,” on page 161. Five sub-cases of RefBaseObject need to be considered here:

- outermost (i.e., non-nested, non-dependent) Package objects,
- nested or dependent Package objects,
- Association objects,

© ISO/IEC 2005 - All rights reserved

- Class proxy objects, and
- Instance objects.

Ordinary clients may only use “refDelete” to delete instances of outermost Package objects and Instance objects.

- Deletion of an outermost Package causes all objects within its extent to be deleted; see 9.3.4.1, “Package object creation and deletion semantics,” on page 170.
- Deletion of an Instance object deletes it and its component closure; see 9.3.4.2, “Instance object lifecycle semantics,” on page 171.

“Invalid Deletion” occurs if an ordinary client invokes “refDelete” on a nested or dependent Package object, an Association object, or a Class proxy object.

As part of the deletion of an outermost Package, a Package object’s implementation may use the “refDelete” operation to delete nested or dependent Package objects, Association objects, and Class proxy objects as well as Instance objects.

refVerifyConstraints

The “refVerifyConstraints” operation triggers verification of the Constraints attached to the target object’s meta-object.

specific analog:	none
return type:	boolean
parameters:	deepVerify : in boolean maxProblems : in long problems : out ViolationTypeSet
exceptions:	MofError (Semantic Error)

The operation should attempt to check any Constraints for the target object that may possibly fail, including any deferred structural constraints. If all constraints hold for the target, the result is true, and an empty set will be returned in ’problems.’ Otherwise, the result is false.

If ’deepVerify’ is false, just the Constraints attached to this object’s meta-object are evaluated. Otherwise any Constraints attached to the meta-objects of this object’s component objects (direct or indirect) are also evaluated. For example, if a Class is defined with Constraints attached to one of its Attributes, these will only be evaluated if ’deepVerify’ is true.

The ’maxProblems’ parameter allows the caller to specify the maximum number of ViolationType instances to be returned in ’problems.’ If ’maxProblems’ is zero, no ViolationType instances will be returned. When an implementation detects more constraint violations that can be reported, it should cease checking and return the results that it can. If ’maxProblems’ is UNBOUNDED (-1), or any other negative number, the implementation should return ViolationType instances for all constraint violations detected.

The ’problems’ parameter is used to return ViolationType descriptors for any constraints that do not currently hold. The ’maxProblems’ parameter says how many descriptors may be returned by the call. Refer to 10.3.2, “Data Types,” on page 271 for description of ViolationType and how it is used.

Interface

```
interface RefBaseObject {
        string ref_mof_id ();
        DesignatorType ref_meta_object ();
```

© ISO/IEC 2005 - All rights reserved

```
        boolean ref_itself (in RefBaseObject other_object);
        RefPackage ref_immediate_package ();
        RefPackage ref_outermost_package ();
        void ref_delete ()
            raises (MofError);
        boolean refVerifyConstraints(
                in boolean deepVerify,
                in long maxProblems,
                out ViolationTypeSet problems)
            raises (MofError)
    }; // end of RefBaseObject
```

10.2.3 Reflective::RefObject (abstract)

The RefObject interface provides the meta-object description of an object that inherits from it, provides generic operations for testing for object identity and type membership, and a range of operations for accessing and updating the object in a model independent way.

The model assumed by the interface is that an object has structural features and operations. The model allows structural features to have single values or collection values. In the latter case, the collection values may have ordering or uniqueness semantics. There is provision for creation of new object instances, and for obtaining the set of objects that exist in a context.

Supertypes

RefBaseObject

Operations

refIsInstanceOf

This operation tests whether this RefObject is an instance of the Class described by the “someClass” meta-object. If the “considerSubtypes” argument is TRUE, an object whose Class is a subclass of the Class described by “someClass” will be considered as an instance of the Class.

specific analog:	none
return type:	boolean
isQuery:	yes
parameters:	someClass : in DesignatorType considerSubtypes : in boolean
exceptions:	MofError (Invalid Designator, Wrong Designator Kind)

refCreateInstance

This operation creates a new instance of the Class for the RefObject’s most derived interface. The operation can be called on a Class proxy object or an Instance object. The “args” list gives the initial values for the new Instance object’s instance scoped, non-derived Attributes.

© ISO/IEC 2005 - All rights reserved

specific analog:	create_<*class_name*>**(...)**; (see 9.8.9, “Class Create Template,” on page 213).
return type:	RefObject
parameters:	args : in Any (multiplicity: zero or more; ordered)
exceptions:	MofError (Overflow, Underflow, Duplicate, Composition Closure, Supertype Closure, Already Created, Abstract Class, Wrong Type, Wrong Number Parameters)

The members of the "args" list correspond 1-to-1 to the parameters for the specific create operation. They must be encoded as per 10.2.1.1, “The Standard Value Encoding Pattern,” on page 249. “Wrong Type” and “Wrong Number Parameters” when the “args” list has the wrong length or is incorrectly encoded.

“Abstract Class” occurs when “refCreateInstance” is called to create an instance of an “abstract Class.” The remaining error conditions are directly equivalent to error conditions for the specific “create” operation.

refAllObjects

The “refAllObjects” operation returns the set of all Instances in the current extent whose type is given by this object’s Class. The operation can be called on a Class proxy object or an Instance object.

specific analog:	**attribute** all_of_type_<*class_name*>; **attribute** all_of_class_<*class_name*>; (See 9.8.6, “Class Template,” on page 209).
return type:	RefObject (multiplicity zero or more; unique; unordered)
isQuery:	yes
parameters:	includeSubtypes : in boolean
exceptions:	none

If “includeSubtypes” is true, the Instance objects for any subClasses of the M2 level Class are also included in the result set. This case is equivalent to the specific “all_of_type_<*class_name*>.”

If the M2 level Class has “isAbstract” set to true, the result of

ref_all_objects(false)

is an empty set.

refValue

The “refValue” operation fetches the current value of the Attribute or Reference denoted by the “feature” argument. If this object is a Class proxy, only classifier scoped Attributes can be fetched.

© ISO/IEC 2005 - All rights reserved

.

specific analog:	<reference_name>**()**; (see 9.8.13, “Operation Template,” on page 240). <attribute_name>**()**; (see 9.8.11, “Attribute Template,” on page 222).
return type:	Any
isQuery:	yes
parameters:	feature : in DesignatorType
exceptions:	NotSet, MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator, Not Public, Wrong Scope, Underflow)

The result for the “refValue” operation is encoded as per 10.2.1.2, “The Alternate Value Encoding Pattern,” on page 249.

“NotSet” occurs when the feature’s multiplicity is [0..1] and its value is unset (i.e., an empty collection). This should not occur with other multiplicities.

“Invalid Designator,” “Wrong Designator Kind,” “Unknown Designator,” “Not Public,” and “Wrong Scope” all occur in cases where the “feature” argument does not denote an Attribute or Reference accessible from this object.

“Underflow” occurs when the feature is a Reference with multiplicity is [1..1] and its value has not been initialized. This should not occur for an Attribute or with other multiplicities.

refSetValue

The “refSetValue” operation assigns a new value to an Attribute or Reference for an object. The assigned value must be a single value or a collection value depending on the feature’s multiplicity.

specific analog:	set_<reference_name>**(**newValue**)**; (see 9.8.12, “Reference Template,” on page 231). set_<attribute_name>**(**newValue**)**; (see 9.8.11, “Attribute Template,” on page 222).
return type:	none
parameters:	feature : in DesignatorType newValue : in Any
exceptions:	MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator, Not Public, Wrong Scope, Not Changeable, Underflow, Overflow, Duplicate, Reference Closure, Composition Closure, Composition Cycle, Invalid Object, Nil Object, Inaccessible Object, Wrong Type)

The “newValue” parameter must be encoded as per 10.2.1.2, “The Alternate Value Encoding Pattern,” on page 249. “Wrong Type” occurs when this parameter is incorrectly encoded.

“Invalid Designator,” “Wrong Designator Kind,” “Unknown Designator,” “Not Public,” “Wrong Scope,” and “Not Changeable” all occur in situations where the “feature” parameter does not denote a changeable Attribute or Reference that is accessible from this object.

The remaining error conditions are directly equivalent to error conditions for the “set_<*feature_name*>” operation.

© ISO/IEC 2005 - All rights reserved

refUnsetValue

The “refUnsetValue” operation resets an optional Attribute or Reference to contain no elements. This operation can only be used when the feature’s multiplicity is [0..1].

specific analog:	unset_<reference_name>**()**; (see 9.8.12, “Reference Template,” on page 231). unset_<attribute_name>**()**; (see 9.8.12, “Reference Template,” on page 231).
return type:	none
parameters:	feature : in DesignatorType
exceptions:	MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator, Not Public, Wrong Scope, Not Changeable, Wrong Multiplicity, Underflow)

“Invalid Designator,” “Wrong Designator Kind,” “Unknown Designator,” “Not Public,” “Wrong Scope,” “Not Changeable,” and “Wrong Multiplicity” all occur in situations where the “feature” parameter does not denote an Attribute or Reference for which “unset_<*feature_name*>” is allowed.

“Underflow” occurs in the same situation as for the “unset_<*feature_name*>” operation. For example, when “feature” is a Reference whose exposed Association End has a non-zero lower bound.

refAddValue

The “refAddValue” operation adds a new element to the current value of an Attribute or Reference with multiplicity that allows multiple values. If the Attribute or Reference is ordered, the new element is added at the end of the current value.

specific analog:	add_<reference_name>**(**newElement**)**; (see 9.8.12, “Reference Template,” on page 231). add_<attribute_name>**(**newElement**)**; (see 9.8.11, “Attribute Template,” on page 222).
return type:	none
parameters:	feature : in DesignatorType newElement : in Any
exceptions:	MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator, Not Public, Wrong Scope, Not Changeable, Wrong Multiplicity, Overflow, Duplicate, Invalid Object, Nil Object, Inaccessible Object, Reference Closure, Composition Closure, Composition Cycle, Wrong Type)

The “newElement” parameter should contain a single value of the feature’s base type. “Wrong Type” occurs when it does not.

“Invalid Designator,” “Wrong Designator Kind,” “Unknown Designator,” “Not Public,” “Wrong Scope,” “Not Changeable,” and “Wrong Multiplicity” all occur when the “feature” parameter does not designate a Reference or Attribute for which the “add_<*feature_name*>” operation is allowed.

The remaining error conditions are directly equivalent to error conditions for the “add_<*feature_name*>” operation.

refAddValueBefore

The “refAddValueBefore” operation is similar to “refAddValue” except that the caller specifies an existing element before which the new element is to be added.

© ISO/IEC 2005 - All rights reserved

This operation can only be used for Attributes and References that are multi-valued and ordered. If the feature is non-unique (and therefore an Attribute), the insertion is made before the first element that matches, starting from the beginning of the collection.

specific analog:	add_<ref_name>_before**(**newElement, beforeElement**)**; (see 9.8.12, “Reference Template,” on page 231) add_<attr_name>_before**(**newElement, beforeElement**)**; (see 9.8.11, “Attribute Template,” on page 222)
return type:	none
parameters:	feature : in DesignatorType newElement : in Any beforeElement : in Any
exceptions:	NotFound, MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator, Not Public, Wrong Scope, Not Changeable, Wrong Multiplicity, Overflow, Duplicate, Invalid Object, Nil Object, Inaccessible Object, Reference Closure, Composition Closure, Composition Cycle, Wrong Type)

The “newElement” and “beforeElement” parameters should each contain a single value of the feature’s base type. “Wrong Type” occurs when it does not.

“Invalid Designator,” “Wrong Designator Kind,” “Unknown Designator,” “Not Public,” “Wrong Scope,” “Not Changeable,” and “Wrong Multiplicity” all occur when the “feature” parameter does not designate a Reference or Attribute for which the “add_<*feature_name*>_before” operation is allowed.

The remaining error conditions are directly equivalent to error conditions for the “add_<*feature_name*>_before” operation.

refAddValueAt

The “refAddValueAt” operation is similar to “refAddValueBefore” except that the caller explicitly gives the position of the insertion. The operation is only applicable to multi-valued ordered, non-unique Attributes.

specific analog:	add_<ref_name>_at**(**newElement, position**)**; (see 9.8.12, “Reference Template,” on page 231) add_<attr_name>_at**(**newElement, position**)**; (see 9.8.11, “Attribute Template,” on page 222)
return type:	none
parameters:	feature : in DesignatorType newElement : in Any position : in unsigned long
exceptions:	BadPosition, MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator, Not Public, Wrong Scope, Not Changeable, Wrong Multiplicity, Overflow, Duplicate, Invalid Object, Nil Object, Inaccessible Object, Reference Closure, Composition Closure, Composition Cycle, Wrong Type)

© ISO/IEC 2005 - All rights reserved

The “newElement” parameter should contain a single value of the Attribute’s base type. “Wrong Type” occurs if it is not.

The “position” parameter is interpreted the same way as for the corresponding specific operation. “Bad Position” occurs if the position parameter’s value is out of range, as defined for the “add_<*feature_name*>_at” operation (i.e., if it is greater than the size of the collection before the operation is invoked).

“Invalid Designator,” “Wrong Designator Kind,” “Unknown Designator,” “Not Public,” “Wrong Scope,” “Not Changeable,” and “Wrong Multiplicity” all occur when “feature” does not designate an Attribute for which the “add_<*feature_name*>_at” operation is allowed.

The remaining error conditions are directly equivalent to error conditions for the specific “add_<*feature_name*>_at” operation.

refModifyValue

The “refModifyValue” operation replaces one element of a multi-valued Attribute or Reference with a new value. If the feature is an ordered and non-unique (and therefore an Attribute), the element modified is the first one that matches, starting from the beginning of the collection.

specific analog:	modify_<ref_name>**(**oldElement, newElement**)**; (see 9.8.12, “Reference Template,” on page 231) modify_<attr_name>**(**oldElement, newElement**)**; (see 9.8.11, “Attribute Template,” on page 222)
return type:	none
parameters:	feature : in DesignatorType oldElement : in Any newElement : in Any
exceptions:	NotFound, MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator, Not Public, Wrong Scope, Not Changeable, Wrong Multiplicity, Underflow, Overflow, Duplicate, Invalid Object, Nil Object, Inaccessible Object, Reference Closure, Composition Closure, Composition Cycle, Wrong Type)

The “newElement” and “oldElement” parameters should contain a single value of the feature’s base type. “Wrong Type” occurs if it is not.

The “oldElement” parameter should be an existing element of the collection being updated. “Not Found” occurs if it is not.

“Invalid Designator,” “Wrong Designator Kind,” “Unknown Designator,” “Not Public,” “Wrong Scope,” “Not Changeable,” and “Wrong Multiplicity” all occur when the “feature” parameter does not designate a Reference or Attribute that supports the “modify_<*feature_name*>” operation.

The remaining error conditions are directly equivalent to error conditions for the “modify_<*feature_name*>” operation.

© ISO/IEC 2005 - All rights reserved

refModifyValueAt

The “refModifyValueAt” operation is similar to the “refModifyValue” operation, except that the element to be modified is specified by position. The operation is only applicable to multi-valued, ordered, non-unique Attributes.

specific analog:	modify_<ref_name>_at**(**newElement, position**)**; (see 9.8.12, “Reference Template,” on page 231) modify_<attr_name>_at**(**newElement, position**)**; (see 9.8.11, “Attribute Template,” on page 222)
return type:	none
parameters:	feature : in DesignatorType newElement : in Any position : in unsigned long
exceptions:	BadPosition, MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator, Not Public, Wrong Scope, Not Changeable, Wrong Multiplicity, Underflow, Overflow, Duplicate, Invalid Object, Nil Object, Inaccessible Object, Reference Closure, Composition Closure, Composition Cycle, Wrong Type)

The “newElement” parameter should contain a single value of the Attribute’s base type. “Wrong Type” occurs if it is not.

The “position” parameter is interpreted in the same way as for the corresponding specific operation. “Bad Position” occurs if the position parameter’s value is out of range, as defined for the “modify_<*feature_name*>_at” operation (i.e., if it is greater than or equal to the size of the collection).

“Invalid Designator,” “Wrong Designator Kind,” “Unknown Designator,” “Not Public,” “Wrong Scope,” “Not Changeable,” and “Wrong Multiplicity” all occur when “feature” does not designate an Attribute for which the “modify_<*feature_name*>_at” operation is allowed.

The remaining error conditions are directly equivalent to error conditions for the specific “modify_<*feature_name*>_at” operation.

refRemoveValue

The “refRemoveValue” operation removes an element of a multi-valued Attribute or Reference. The operation is only applicable when the upper bound is not equal to the lower bound. When the feature is ordered and non-unique (and therefore an Attribute) the element removed is the first one in the collection that matches, starting from the beginning of the collection..

specific analog:	remove_<reference_name>**(**oldElement**)**; (see 9.8.12, “Reference Template,” on page 231) remove_<attribute_name>**(**oldElement**)**; (see 9.8.11, “Attribute Template,” on page 222)

© ISO/IEC 2005 - All rights reserved

return type:	none
parameters:	feature : in DesignatorType oldElement : in Any
exceptions:	NotFound, MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator, Not Public, Wrong Scope, Not Changeable, Wrong Multiplicity, Underflow, Duplicate, Invalid Object, Nil Object, Inaccessible Object, Reference Closure, Composition Closure, Composition Cycle, Wrong Type)

The “oldElement” parameter should contain a single value of the Attribute’s base type. “Wrong Type” occurs if it is not.

“Not Found” occurs if the value in the “oldElement” parameter is not a member of the collection.

“Invalid Designator,” “Wrong Designator Kind,” “Unknown Designator,” "Not” Public,” “Wrong Scope,” “Not Changeable,” and “Wrong Multiplicity” all occur when “feature” does not designate an Attribute or Reference for which the “remove_<*feature_name*>” operation is allowed.

The remaining error conditions are directly equivalent to error conditions for the specific “remove_<*feature_name*>” operation.

refRemoveValueAt

The “refRemoveValueAt” operation is similar to the “refRemoveValue” operation except that the element to be modified is specified by position. Furthermore, the operation is only applicable to ordered, non-unique Attributes.

specific analog:	remove_<reference_name>_at**(**position**)**; (see 9.8.12, “Reference Template,” on page 231) remove_<attribute_name>_at**(**position**)**; (see 9.8.11, “Attribute Template,” on page 222)
return type:	none
parameters:	feature : in DesignatorType position : in unsigned long
exceptions:	BadPosition, MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator, Not Public, Wrong Scope, Not Changeable, Wrong Multiplicity, Underflow, Duplicate, Invalid Object, Nil Object, Inaccessible Object, Reference Closure, Composition Closure, Composition Cycle, Wrong Type)

The “position” parameter is interpreted in the same way as for the corresponding specific operation. “Bad Position” occurs if the position parameter’s value is out of range, as defined for the “remove_<*feature_name*>_at” operation (i.e., if it is greater than or equal to the size of the collection before the operation is called).

“Invalid Designator,” “Wrong Designator Kind,” “Unknown Designator,” “Not Public,” “Wrong Scope,” “Not Changeable,” and “Wrong Multiplicity” all occur when “feature” does not designate an Attribute for which the “remove_<*feature_name*>_at” operation is allowed.

The remaining error conditions are directly equivalent to error conditions for the specific “remove_<*feature_name*>_at” operation.

© ISO/IEC 2005 - All rights reserved

refImmediateComposite

The “refImmediateComposite” operation returns the “immediate composite” object for this Instance as specified below.

specific analog:	none
return type:	RefObject
isQuery:	yes
exceptions:	none

The immediate composite object C returned by this operation is an Instance object such that:

- C is related to this object via a relation R defined by an Attribute or Association,
- the aggregation semantics of the relation R are “composite,” and
- this object fills the role of “component” in its relationship with C.

If the immediate object C does not exist, or if “this” object is a Class proxy object rather than an Instance object, a CORBA nil object reference is returned.

NOTE: If the composite relationship R corresponds to a “classifier-level” scoped M2-level Attribute, the immediate composite object C will be the Class Proxy object that holds the Attribute value.

refOutermostComposite

The “refOutermostComposite” operation returns the “outermost composite” for this object as defined below.

specific analog:	none
return type:	RefObject
isQuery:	yes
exceptions:	none

The outermost composite object C returned by this operation is an Instance object such that:

- There is a chain of *zero or more* immediate composite relationships (as described for “The “refImmediateComposite”) connecting “this” object to C, and
- C does not have an immediate composite.

The above definition is such that if “this” object is not a component of any other object, it will be returned.

If “this” object is a Class proxy object, a CORBA nil object reference is returned.

NOTE: As with “refImmediateComposite” if the last composite relationship in the chain corresponds to a “classifier-level” scoped M2 level Attribute, the outermost composite object C will be the Class Proxy object that holds the Attribute value.

© ISO/IEC 2005 - All rights reserved

refInvokeOperation

The “refInvokeOperation” operation invokes a metamodel defined Operation on the Instance or Class proxy object with the arguments supplied.

specific analog:	none
return type:	Any (multiplicity: zero or more; ordered; not unique)
parameters:	requestedOperation : in DesignatorType args : inout Any (multiplicity: zero or more; ordered; non-unique)
exceptions:	OtherException, MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator, Not Public, Wrong Scope, Overflow, Underflow, Duplicate, Wrong Number Parameters, Wrong Type, Semantic Error)

The “args” parameter is used to pass the values of all of the Operation’s Parameters that have directions “in,” “out,” or “inout” but not the “return” Parameter. There must be a distinct parameter value (real or dummy) in the “args” list for every “in,” “out,” and “inout” Parameter. “Wrong Number Parameters” occurs if this is not so.

The parameter values in “args” must appear in the order of the Operation’s “in,” “out,” and “inout” Parameters as defined in the metamodel.

The “args” member values provided by the caller for “in” and “inout” Parameter positions must be encoded depending on the Parameter’s type and multiplicity as per the “The Standard Value Encoding Pattern” on page 249. “Wrong Type” occurs if any of these values have the wrong type for the corresponding Parameter. “Underflow,” “Overflow,” or “Duplicate” occur when one of the supplied values does not fit the multiplicity specified by the corresponding Parameter.

The caller must provide a dummy “args” member value in each “out” Parameter position. This value may be any legal CORBA Any value.

The “args” member values passed back to the caller for “out” and “inout” Parameter positions are likewise encoded depending on the Parameter’s type and multiplicity as per 10.2.1.1, “The Standard Value Encoding Pattern,” on page 249. Note that the values passed back to the caller the “in” Parameter positions of the “args” list are dummies whose content is undefined.

If the Operation defines a result (i.e., a Parameter with direction “return”), the result for a “The “refInvokeOperation” operation invokes a metamodel defined Operation on the Instance or Class proxy object with the arguments supplied.” call gives the result value. This is encoded depending on the “return” Parameter’s type and multiplicity as per 10.2.1.1, “The Standard Value Encoding Pattern,” on page 249. When the Operation does not define a result, the result of a “The “refInvokeOperation” operation invokes a metamodel defined Operation on the Instance or Class proxy object with the arguments supplied.” call is a dummy value whose content is undefined.

NOTE: In the cases above where dummy values are used, it is recommended that “light weight” Any values are used. (We would recommend the use of an Any value whose type kind is tk_null. However, there is currently some question as to whether the CDR standard defines an encoding for this value.)

“OtherException” occurs when a “refInvokeOperation” invocation needs to signal an Operation specific Exception. The “exception_designator” field of “OtherException” will denote the Exception raised, and the “exception_args” list will give the values for any Exception fields. The “exception_args” list will have one member value for each Parameter of the Exception in the order defined by the meta-model. The member values will be encoded depending on the corresponding Exception Parameter’s type and multiplicity as per 10.2.1.1, “The Standard Value Encoding Pattern,” on page 249.

© ISO/IEC 2005 - All rights reserved

A “Semantic Error” will occur if the invoked Operation tries to return a collection value as a result, out or inout parameter that violates the Parameter’s structural constraints. This will also occur if an Exception Parameter’s value is similarly incorrect.

“Invalid Designator,” “Wrong Designator Kind,” “Unknown Designator,” “Not Public,” and “Wrong Scope” all occur when “requestedOperation” does not designate an Operation that can be invoked using this object.

Interface

```
interface RefObject : RefBaseObject {
   boolean ref_is_instance_of (in DesignatorType some_class,
                        in boolean consider_subtypes);
   RefObject ref_create_instance (in AnyList args)
     raises (MofError);
   RefObjectSet ref_all_objects (in boolean include_subtypes);
   void ref_set_value (in DesignatorType feature,
                  in any new_value)
     raises (MofError);
   any ref_value (in DesignatorType feature)
     raises (NotSet, MofError);
void ref_unset_value ()
     raises (MofError);
void ref_add_value (in DesignatorType feature,
                  in any new_element)
     raises (MofError);
   void ref_add_value_before (in DesignatorType feature,
                        in any new_element,
                        in any before_element)
     raises (NotFound, MofError);
void ref_add_value_at (in DesignatorType feature,
                     in any new_element,
                     in unsigned long position)
     raises (BadPosition, MofError);
void ref_modify_value (in DesignatorType feature,
                     in any old_element,
                     in any new_element)
     raises (NotFound, MofError);
   void ref_modify_value_at (in DesignatorType feature,
                        in any new_element,
                        in unsigned long position)
     raises (BadPosition, MofError);
void ref_remove_value (in DesignatorType feature,
                     in any old_element)
     raises (NotFound, MofError);
   void ref_remove_value_at (in DesignatorType feature,
                        in unsigned long position)
     raises (BadPosition, MofError);
RefObject ref_immediate_composite ();
    RefObject ref_outermost_composite ();
any ref_invoke_operation (
           in DesignatorType requested_operation,
           inout AnyList args)
      raises (OtherException, MofError);
}; // end of interface RefObject
```


© ISO/IEC 2005 - All rights reserved

10.2.4 Reflective::RefAssociation (abstract)

The RefAssociation interface provides the meta-object description of an association that inherits from it. It also provides generic operations querying and updating the links that belong to the association.

The model of association supported by this interface is of collection of two ended asymmetric links between objects. The links may be viewed as ordered on one or other of the ends, and there may be some form of cardinality constraints on either end.

The RefAssociation interface is designed to be used with associations that contain no duplicate links, though this is not an absolute requirement. There is no assumption that different association objects for a given association type are mutually aware. Links are modeled as having no object identity.

A data model that required “heavy weight” links with object identity (e.g., so that attributes could be attached to them) would need to represent them as RefObject instances.The RefAssociation interface could be used to manage light weight links between heavy weight link objects and the objects they connect. Similar techniques could be used to represent N-ary associations. However, in both cases better performance would be achieved using a purpose built reflective layer.

Supertypes

RefBaseObject

Operations

refAllLinks

The “refAllLinks” operation returns all links in the link set for this Association object.

specific analog:	all_links(); (see 9.8.10, “Association Template,” on page 214)
return type:	Link (multiplicity zero or more, unordered, unique)
isQuery:	yes
parameters:	none
exceptions:	MofError

This operation returns the current link set for the current Association extent as defined for the specific version of this operation. The links are encoded as per 10.2.1.4, “The Link Encoding Pattern,” on page 250. MofError may be raised to signal a semantic error.

refLinkExists

The “refLinkExists” operation returns true if and only if the supplied link is a member of the link set for this Association object.

specific analog:	link_exists**(**someLink**)**; (see 9.8.10, “Association Template,” on page 214)
return type:	boolean
isQuery:	yes
parameters:	someLink : in Link
exceptions:	MofError(WrongType)

© ISO/IEC 2005 - All rights reserved

The “someLink” parameter should be encoded as per “The Link Encoding Pattern” on page 250. “Wrong Type” occurs if the link encoding is not correct.

refQuery

The “refQuery” operation returns a list containing all Instance objects that are linked to the supplied “queryObject” by links in the extent of this Association object, where the links all have the “queryObject” at the "queryEnd."

specific analog:	<*endName*> **(**queryObject**)**; (see 9.8.10, “Association Template,” on page 214)
return type:	RefObject (Multiplicity zero or more; ordered; unique)
isQuery:	yes
parameters:	queryEnd : in DesignatorType queryObject : in RefObject
exceptions:	MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator, Wrong Type, Invalid Object, Nil Object, Inaccessible Object, Not Navigable)

The “queryEnd” parameter must designate an AssociationEnd for this Association object. “Invalid Designator,” “Wrong Designator Kind,” and “Unknown Designator” occur in cases where this is not so. “Not Navigable” is raised if the “queryEnd” parameter designates an AssociationEnd that has “isNavigable” set to false.

The “queryObject” parameter must be an Instance object whose type is compatible with the type of the “queryEnd” of the Association. “Wrong Type” is raised if the parameter has the wrong type.

“Invalid Object,” “Nil Object,” or “Inaccessible Object” is raised if the “queryObject” parameter is a non-existent, nil, or inaccessible Instance object.

While the result of this operation is declared as an ordered set of links, the ordering only has meaning if the other AssociationEnd (i.e., not the “queryEnd”) is defined ordered.

refAddLink

The “refAddLink” operation adds “newLink” into the set of links in the extent of this Association object. If one or other of the Association’s Ends is ordered, the link is inserted after the last link with respect to that ordering.

specific analog:	add**(**newLink[0], newLink[1]**)**; (see 9.8.10, “Association Template,” on page 214)
return type:	none
parameters:	newLink : in Link
exceptions:	MofError (Not Changeable, Overflow, Duplicate, Reference Closure, Composition Closure, Composition Cycle, Wrong Type, Invalid Object, Nil Object, Inaccessible Object)

The “newLink” parameter should be encoded as per 10.2.1.4, “The Link Encoding Pattern,” on page 250. “Wrong Type” occurs if the link encoding is not correct.

Both RefObject members of the “newLink” parameter should be valid Instance objects. “Invalid Object,” “Nil Object,” or “Inaccessible Object” is raised if either one is a non-existent, nil, or inaccessible Instance object.

“Not Changeable” occurs if this operation is invoked on an Association that has “isChangeable” set to false on either Association End.

© ISO/IEC 2005 - All rights reserved

"Overflow," "Duplicate," "Reference Closure," "Composition Closure," and "Composition Cycle" are directly equivalent to error conditions for the corresponding specific "add" operation.

refAddLinkBefore

The "refAddLinkBefore" operation adds "newLink" into the link set of an ordered Association object. The link insertion point is immediately before the link whose "positionEnd" matches the "before" Instance.

specific analog:	add_before_<*endName*>**(**newLink[0], newLink[1], before**)**; (see 9.8.10, "Association Template," on page 214)
return type:	none
parameters:	newLink : in Link positionEnd : in DesignatorType before : in RefObject
exceptions:	NotFound, MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator, Not Changeable, Not Navigable, Overflow, Duplicate, Reference Closure, Composition Closure, Wrong Type, Invalid Object, Nil Object, Inaccessible Object)

The "newLink" parameter should be encoded as per "The Link Encoding Pattern" on page 250. "Wrong Type" occurs if the link's encoding is not correct.

The "positionEnd" parameter should denote an AssociationEnd of this object's Association. One of "Invalid Designator," "Wrong Designator Kind," or "Unknown Designator" occurs if thus is not the case.

"Not Changeable" occurs if this operation is invoked on an Association that has "isChangeable" set to false on either Association End. "Not Navigable" occurs if the "positionEnd" AssociationEnd has "isNavigable" set to FALSE.

The "before" parameter should be an Instance object that is type compatible with the type of the AssociationEnd denoted by "positionEnd." "Wrong Type" occurs if this is not the case.

The remaining error conditions are directly equivalent to error conditions for the corresponding "add_before_<*endName*>" operation.

refModifyLink

The "refModifyLink" operation updates the "oldLink" in the Association object's link set, replacing the Instance object at "positionEnd" with "newObject."

specific analog:	modify_<*endName*>**(**oldLink[0], oldLink[1], newObject**)**; (see 9.8.10, "Association Template," on page 214)
return type:	none
parameters:	oldLink : in Link positionEnd : in DesignatorType newObject : in RefObject
exceptions:	NotFound, MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator, Not Changeable, Not Navigable, Underflow, Overflow, Duplicate, Reference Closure, Composition Closure, Wrong Type, Invalid Object, Nil Object, Inaccessible Object)

© ISO/IEC 2005 - All rights reserved

The “oldLink” parameter should be encoded as per 10.2.1.4, “The Link Encoding Pattern,” on page 250. “Wrong Type” occurs if the link’s encoding is not correct.

The “positionEnd” parameter should denote an AssociationEnd of this object’s Association. One of “Invalid Designator,” “Wrong Designator Kind,” or “Unknown Designator” occurs if thus is not the case.

“Not Changeable” occurs if the "positionEnd" AssociationEnd that has “isChangeable” set to false. “Not Navigable” occurs if it has “isNavigable” set to false.

The “newObject” parameter should be an Instance object that is type compatible with the type of the AssociationEnd denoted by “positionEnd.” “Wrong Type” occurs if this is not the case.

The remaining error conditions are directly equivalent to error conditions for the corresponding “modify_<*endName*>” operation. Note that any structural constraints notionally apply to the final state following the operation, and not to any intermediate states.

refRemoveLink

The “refRemoveLink” operation removes the “oldLink” from the association.

specific analog:	remove**(**oldLink[0], oldLink[1]**)**; (see 9.8.11, “Attribute Template,” on page 222)
return type:	none
parameters:	oldLink : in Link
exceptions:	NotFound, MofError (Not Changeable, Underflow, Wrong Type, Nil Object)

“Not Changeable” occurs if this operation is invoked on an Association that has “isChangeable” set to false for either AssociationEnd.

The “oldLink” parameter should be encoded as per “The Link Encoding Pattern” on page 250. “Wrong Type” occurs if the link’s encoding is not correct.

“NotFound,” “Nil Object,” and “Underflow” are directly equivalent to error conditions for the corresponding specific “remove” operation. “Invalid Object” and “Inaccessible Object” cannot occur, as in the specific operation.

Interface

```
interface RefAssociation : RefBaseObject {
   LinkSet ref_all_links ()
     taises (MofError);
boolean ref_link_exists (in Link some_link)
    raises (MofError);
RefObjectUList ref_query (in DesignatorType query_end,
                          in RefObject query_object)
    raises (MofError);
   void ref_add_link (in Link new_link)
    raises (MofError);
   void ref_add_link_before (in Link new_link,
                          in DesignatorType position_end,
                          in RefObject before)
    raises (NotFound, MofError);
   void ref_modify_link (in Link old_link,
                      in DesignatorType position_end,
```


© ISO/IEC 2005 - All rights reserved

```
                in RefObject new_object)
    raises (NotFound, MofError);
  void ref_remove_link (in Link old_link)
    raises (NotFound, MofError);
}; // end of interface RefAssociation
```

10.2.5 Reflective::RefPackage (abstract)

The RefPackage interface is an abstraction for accessing a collection of objects and their associations. The interface provides an operation to access the meta-object description for the package, and operations to access the package instance's class proxy objects (one for each Class) and its association objects.

Supertypes

RefBaseObject

Operations

refClassRef

The “refClassRef” operation returns the Class proxy object for a given Class.

specific analog:	**readonly attribute** <ClassName>_class_ref; (see 9.8.10, “Association Template,” on page 214)
return type:	RefObject
isQuery:	yes
parameters:	class : in DesignatorType
exceptions:	MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator)

The “class” parameter should designate the M2 level Class whose Class proxy object is to be returned. “Invalid Designator,” “Wrong Designator Kind,” “Unknown Designator” occur in various situations where this is not the case.

refAssociationRef

The “refAssociationRef” operation returns an Association object for a given Association.

specific analog:	**readonly attribute** <AssociationName>_ref; (see 9.8.10, “Association Template,” on page 214)
return type:	RefAssociation
isQuery:	yes
parameters:	association : DesignatorType
exceptions:	MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator)

The “association” parameter should designate the M2 level Association whose Association object is to be returned. “Invalid Designator,” “Wrong Designator Kind,” “Unknown Designator” occur in various situations where this is not the case.

© ISO/IEC 2005 - All rights reserved

refPackageRef

The “refPackageRef” operation returns a Package object for a nested or clustered Package.

specific analog:	**readonly attribute** <PackageName>_ref; (see 9.8.10, “Association Template,” on page 214)
return type:	RefPackage
isQuery:	yes
parameters:	package : DesignatorType
exceptions:	MofError (Invalid Designator, Wrong Designator Kind, Unknown Designator)

The “package” parameter should designate the M2 level Package whose Package object is to be returned. It must either be nested within the Package for this Package object, or imported with “isClustered” set to true. “Invalid Designator,” “Wrong Designator Kind,” “Unknown Designator” occur in the situations where this is not the case.

Interface

```
interface RefPackage : RefBaseObject {

  RefObject ref_class_ref (in DesignatorType type)
    raises (MofError);

  RefAssociation ref_association_ref (
      in DesignatorType association)
    raises (MofError);

   RefPackage ref_package_ref (in DesignatorType package)
    raises (InvalidDesignator)

}; // end of interface RefPackage
```

10.3 The CORBA IDL for the Reflective Interfaces

This sub clause describes the relevant excerpts of the CORBA IDL for the Reflective module.

10.3.1 Introduction

The Reflective module starts with forward declarations of the three object types RefObject, RefAssociation, and RefPackage.

```
module Reflective {
  interface RefBaseObject;

  interface RefObject;
  typedef sequence < RefObject > RefObjectUList;

  interface RefAssociation;

  interface RefPackage;
```


© ISO/IEC 2005 - All rights reserved

10.3.2 Data Types

Operations on the Reflective interfaces need to identify the elements (e.g., attributes, operations, roles, classes, etc.) that they apply to. Some exceptions have similar requirements. The type DesignatorType is used to denote uses of RefObject with this meaning.

```
typedef RefObject DesignatorType;
```

Links are expressed as bounded sequences of (two) RefObject values.

```
typedef sequence <RefObject, 2> Link;
typedef sequence <Link> LinkSet;
```

The following type is used to pass multiple CORBA Any values.

```
typedef sequence < any > AnyList;
```

The 'refVerifyConstraints' operation uses an instance of ViolationTypeSet to return descriptions of any constraints that do not hold. This type and the ViolationType type are defined below. The fields called 'error_kind,' 'element_in_error,' 'extra_info,' and 'error_description' of ViolationType are equivalent to fields of the MofError exception; see 9.4, "Exception Framework," on page 183. The 'object_in_error' field gives the MOF object (if any) whose state violates the constraint reported. When a Constraint on a DataType is violated, 'object_in_error' will refer to the MOF object, which has the erroneous DataType instance as an attribute value.

```
struct ViolationType {
    wstring error_kind;
    RefBaseObject object_in_error;
    RefObject element_in_error;
    NamedValueList extra_info;
    wstring error_description;
};

typedef sequence < ViolationType > ViolationTypeSet;
```

© ISO/IEC 2005 - All rights reserved

© ISO/IEC 2005 - All rights reserved

Annex A
(normative)

Conformance Information

The MOF specification has the following conformance points. These points are independent of each other.

A.1 MOF Model and its IDL Interfaces

This compliance point defines a MOF compliant metamodel service. It has the following components:

- The service must support the interfaces of the Model module as defined in Clause 7.
- The service must implement the semantics of the Model module defined by elaborating the MOF to IDL mapping semantic specifications in Clause 9 for the MOF Model.
- The service must implement the interfaces and semantics of the Reflective module as defined in Clause 10, in conjunction with the Model module semantics.

A.2 MOF to IDL Mapping

This compliance point defines the compliance of a CORBA MOF implementation for a specific metamodel. It has one component:

- A MOF implementation that supports CORBA for a specific metamodel must support IDL Interfaces that conform to the MOF to IDL mapping templates and semantics as defined in Clause 9.

NOTE: This mapping enables interoperability of conformant implementations. Automation of this mapping by a product (for example, IDL or code generation) is not required for conformance to the MOF to IDL mapping.

A.3 MOF Metamodel Interchange

This compliance point relates to the ability of a metamodel service or some other tool to interchange metamodels using XMI. It has two components:

- The tool must be able to externalize a metamodel as an XML document that is equivalent to the result of elaborating the metamodel according to the XMI production rules in the context of the MOF Model.
- The tool must be able to internalize a metamodel elaborated as above.

© ISO/IEC 2005 - All rights reserved

© ISO/IEC 2005 - All rights reserved

Annex B
(normative)

Legal Information

B.1 Copyright Information

Copyright © 1998, 1999, 2000, 2001 IBM Corporation
Copyright © 2003, Object Management Group
Copyright © 1998, 1999, 2000, 2001 Softeam
Copyright © 1998, 1999, 2000, 2001 Unisys Corporation

B.2 Use Of Specification - Terms, Conditions & Notices

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

B.3 Licenses

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

B.4 Patents

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

© ISO/IEC 2005 - All rights reserved

B.5 General Use Restrictions

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

B.6 Disclaimer Of Warranty

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.

IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

B.7 Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 250 First Avenue, Needham, MA 02494, U.S.A.

B.8 Trademarks

The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, UML®, XMI® and IIOP® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBAservices™, CORBAfacilities™, CORBAmed™, CORBAnet™, Integrate 2002™, Middleware That's Everywhere™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ and the XMI Logo™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

B.9 Compliance

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

© ISO/IEC 2005 - All rights reserved

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

B.10 Issue Reporting

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents & Specifications, Report a Bug/Issue.

© ISO/IEC 2005 - All rights reserved

© ISO/IEC 2005 - All rights reserved

© ISO/IEC 2005 - All rights reserved

© ISO/IEC 2005 - All rights reserved