
Information technology — Meta Object Facility (MOF)

Technologies de l'information — Facilité d'objet «méta» (MOF)

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

© ISO/IEC 2005

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Foreword viii

Introduction ix

1 Scope 1

2 Normative references 1

 2.1 Identical Recommendations | International Standards 1

 2.2 International Standards 1

3 Abbreviations and Conventions 2

4 List of Documents 2

5 MOF Usage Scenarios 3

 5.1 Overview 3

 5.2 Software Development Scenarios 4

 5.3 Type Management Scenarios 5

 5.4 Information Management Scenarios 6

 5.5 Data Warehouse Management Scenarios 7

6 MOF Conceptual Overview 9

 6.1 Overview 9

 6.2 Metadata Architectures 9

 6.2.1 Four Layer Metadata Architectures 9

 6.2.2 The MOF Metadata Architecture 10

 6.2.3 MOF Metamodeling Terminology 12

 6.3 The MOF Model - Metamodeling Constructs 13

 6.3.1 Classes 13

 6.3.2 Associations 16

 6.3.3 Aggregation 17

 6.3.4 References 18

 6.3.5 DataTypes 20

 6.3.6 Packages 20

 6.3.7 Constraints and Consistency 23

 6.3.8 Miscellaneous Metamodeling Constructs 24

 6.4 Metamodels and Mappings 25

 6.4.1 Abstract and Concrete Mappings 25

 6.4.2 The MOF Metamodel to IDL Mapping 26

 6.4.3 The MOF Metamodel to XML Mapping 26

 6.4.4 Mappings of the MOF Model 27

7 MOF Model and Interfaces 29

 7.1 Overview 29

 7.2 How the MOF Model is Described 29

 7.2.1 Classes 30

7.2.2 Associations	34
7.2.3 DataTypes	35
7.2.4 Exceptions	35
7.2.5 Constants	36
7.2.6 Constraints	36
7.2.7 UML Diagrams	36
7.3 The Structure of the MOF Model	36
7.3.1 The MOF Model Package	36
7.3.2 The MOF Model Service IDL	38
7.3.3 The MOF Model Structure	38
7.3.4 The MOF Model Containment Hierarchy	40
7.4 MOF Model Classes	41
7.4.1 ModelElement	(abstract) 41
7.4.2 Namespace	(abstract) 45
7.4.3 GeneralizableElement	(abstract) 48
7.4.4 TypedElement	(abstract) 52
7.4.5 Classifier	(abstract) 53
7.4.6 Class	54
7.4.7 DataType	(abstract) 55
7.4.8 PrimitiveType	56
7.4.9 CollectionType	57
7.4.10 EnumerationType	58
7.4.11 AliasType	59
7.4.12 StructureType	59
7.4.13 StructureField	60
7.4.14 Feature	(abstract) 60
7.4.15 StructuralFeature	(abstract) 62
7.4.16 Attribute	(idl_substitute_name "MofAttribute") 63
7.4.17 Reference	64
7.4.18 BehavioralFeature	(abstract) 66
7.4.19 Operation	67
7.4.20 Exception	(idl_substitute_name "MofException") 68
7.4.21 Association	69
7.4.22 AssociationEnd	71
7.4.23 Package	74
7.4.24 Import	76
7.4.25 Parameter	78
7.4.26 Constraint	79
7.4.27 Constant	82
7.4.28 Tag	83
7.5 MOF Model Associations	85
7.5.1 Contains	85
7.5.2 Generalizes	86
7.5.3 RefersTo	87
7.5.4 Exposes	(derived) 88
7.5.5 IsOfType	90
7.5.6 CanRaise	90
7.5.7 Aliases	91
7.5.8 Constrains	92
7.5.9 DependsOn	(derived) 93

7.5.10 AttachesTo	95
7.6 MOF Model Data Types	96
7.6.1 PrimitiveTypes used in the MOF Model	96
7.6.2 MultiplicityType.....	96
7.6.3 VisibilityKind.....	97
7.6.4 DirectionKind.....	98
7.6.5 ScopeKind	98
7.6.6 AggregationKind	98
7.6.7 EvaluationKind	98
7.7 MOF Model Exceptions	99
7.7.1 NameNotFound.....	99
7.7.2 NameNotResolved	99
7.8 MOF Model Constants	99
7.8.1 Unbounded	100
7.8.2 The Standard DependencyKinds	100
7.9 MOF Model Constraints	101
7.9.1 MOF Model Constraints and other M2 Level Semantics	101
7.9.2 Notational Conventions	101
7.9.3 OCL Usage in the MOF Model specification	103
7.9.4 The MOF Model Constraints	105
7.9.5 Semantic specifications for some Operations, derived Attributes and Derived Associations	125
7.9.6 OCL Helper functions	131
7.10 The PrimitiveTypes Package	134
7.10.1 Boolean.....	135
7.10.2 Integer	135
7.10.3 Long	135
7.10.4 Float.....	135
7.10.5 Double	135
7.10.6 String	135
7.10.7 IDL for the PrimitiveTypes Package	136
7.11 Standard Technology Neutral Tags	136
8 The MOF Abstract Mapping.....	139
8.1 Overview	139
8.2 MOF Values	139
8.3 Semantics of Data Types	139
8.4 Semantics of Equality for MOF Values	140
8.5 Semantics of Class Instances	141
8.6 Semantics of Attributes	141
8.6.1 Attribute name and type	142
8.6.2 Multiplicity	142
8.6.3 Scope	143
8.6.4 Is_derived	144
8.6.5 Aggregation.....	144
8.6.6 Visibility and is_changeable	144
8.7 Package Composition	144
8.7.1 Package Nesting	144
8.7.2 Package Generalization	145
8.7.3 Package Importation	145

8.7.4 Package Clustering	145
8.8 Extents	145
8.8.1 The Purpose of Extents	146
8.8.2 Class Extents	147
8.8.3 Association Extents	147
8.8.4 Package Extents	147
8.9 Semantics of Associations	149
8.9.1 MOF Associations in UML notation	149
8.9.2 Core Association Semantics	150
8.9.3 AssociationEnd Changeability	152
8.9.4 Association Aggregation	152
8.9.5 Derived Associations	152
8.10 Aggregation Semantics	152
8.10.1 Aggregation “none”	152
8.10.2 Aggregation “composite”	153
8.10.3 Aggregation “shared”	153
8.11 Closure Rules	153
8.11.1 The Reference Closure Rule.....	153
8.11.2 The Composition Closure Rule	155
8.12 Recommended Copy Semantics	156
8.13 Computational Semantics	157
8.13.1 A Style Guide for Metadata Computational Semantics	157
8.13.2 Access operations should not change metadata	158
8.13.3 Update operations should only change the nominated metadata	158
8.13.4 Derived Elements should behave like non-derived Elements	158
8.13.5 Constraint evaluation should not have side-effects	158
8.13.6 Access operations should avoid raising Constraint exceptions	159
9 MOF to IDL Mapping	161
9.1 Overview	161
9.2 Meta Objects and Interfaces	161
9.2.1 Meta Object Type Overview	161
9.2.2 The Meta Object Interface Hierarchy	163
9.3 Computational Semantics for the IDL Mapping	165
9.3.1 The CORBAIdl Types Package.....	165
9.3.2 Mapping of MOF Data Types to CORBA IDL Types.....	169
9.3.3 Value Types and Equality in the IDL Mapping	170
9.3.4 Lifecycle Semantics for the IDL Mapping	170
9.3.5 Association Access and Update Semantics for the IDL Mapping	173
9.3.6 Link Addition Operations	173
9.3.7 Attribute Access and Update Semantics for the IDL Mapping	176
9.3.8 Reference Semantics for the IDL Mapping	181
9.3.9 Cluster Semantics for the IDL Mapping	182
9.3.10 Atomicity Semantics for the IDL Mapping	182
9.3.11 The Supertype Closure Rule	182
9.3.12 Copy Semantics for the IDL Mapping	183
9.4 Exception Framework	183
9.4.1 Error_kind string values	185
9.4.2 Structural Errors	185
9.4.3 Constraint Errors	188

9.4.4 Semantic Errors	188
9.4.5 Usage Errors	189
9.4.6 Reflective Errors	190
9.5 Preconditions for IDL Generation	192
9.6 Standard Tags for the IDL Mapping	194
9.6.1 Tags for Specifying IDL #pragma directives	194
9.6.2 Tags for Providing Substitute Identifiers	195
9.6.3 Tags for Specifying IDL Inheritance	196
9.7 Generated IDL Issues	198
9.7.1 Generated IDL Identifiers	198
9.7.2 Generation Rules for Synthesized Collection Types	200
9.7.3 IDL Identifier Qualification	202
9.7.4 File Organization and #include statements	202
9.8 IDL Mapping Templates	202
9.8.1 Template Notation	203
9.8.2 Package Module Template	203
9.8.3 Package Factory Template	205
9.8.4 Package Template	206
9.8.5 Class Forward Declaration Template	209
9.8.6 Class Template	209
9.8.7 Class Proxy Template	210
9.8.8 Instance Template	212
9.8.9 Class Create Template	213
9.8.10 Association Template	214
9.8.11 Attribute Template	222
9.8.12 Reference Template	231
9.8.13 Operation Template	240
9.8.14 Exception Template	242
9.8.15 DataType Template	243
9.8.16 Constraint Template	245
9.8.17 Annotation Template	245
10 The Reflective Module	247
10.1 Introduction	247
10.2 The Reflective Interfaces	248
10.2.1 Reflective Argument Encoding Patterns	248
10.2.2 Reflective::RefBaseObject	(abstract) 250
10.2.3 Reflective::RefObject	(abstract) 254
10.2.4 Reflective::RefAssociation	(abstract) 265
10.2.5 Reflective::RefPackage	(abstract) 269
10.3 The CORBA IDL for the Reflective Interfaces	270
10.3.1 Introduction	270
10.3.2 Data Types	271
Annex A (normative) Conformance Issues	273
Annex B (normative) Legal Information	275
INDEX	279

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

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

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national 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 and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC 19502 was prepared by the Object Management Group (OMG) and was adopted, under the PAS procedure, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC.

ISO/IEC 19502 is related to

- ISO/IEC 19501, *Information technology — Open Distributed Processing — Unified Modeling Language (UML) Version 1.4.2*
- ISO/IEC 19503, *Information technology — XML Metadata Interchange (XMI)*
- ISO/IEC 14769, *Information technology — Open Distributed Processing — Type Repository Function*

.....

Introduction

This International Standard defines a metamodel (defined using MOF), a set of interfaces (defined using ODP IDL (ITU-T Recommendation X.920 (1997) | ISO/IEC 14750:1999), that can be used to define and manipulate a set of interoperable metamodels and their corresponding models. It 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, *Open Distributed Processing — Reference Model* (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, this International 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.

Document Reference: 123456789

Information technology — Meta Object Facility (MOF)

1 Scope

This International Standard specifies the following:

- a. An abstract language 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 which uses standard technology mappings to transform MOF metamodels into metadata APIs.

This International Standard also provides the following:

- 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

- ITU-T Recommendation X.902 (1996) | ISO/IEC 10746-2:1996, *Open Distributed Processing — Reference Model: Foundations*
- ITU-T Recommendation X.903 (1996) | ISO/IEC 10746-3:1996, *Open Distributed Processing — Reference Model: Architecture*

2.2 International Standards

- ISO/IEC 14769:2001, *Information technology — Open Distributed Processing — Type Repository Function*
- ISO/IEC 19501:2005, *Information technology — Open Distributed Processing — Unified Modeling Language (UML) Version 1.4.2*
- ISO/IEC 19503:2005, *Information technology — XML Metadata Interchange (XMI)*

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.

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 model¹ 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.

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 idea². 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.

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.

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.

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.

(Blank page)

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.

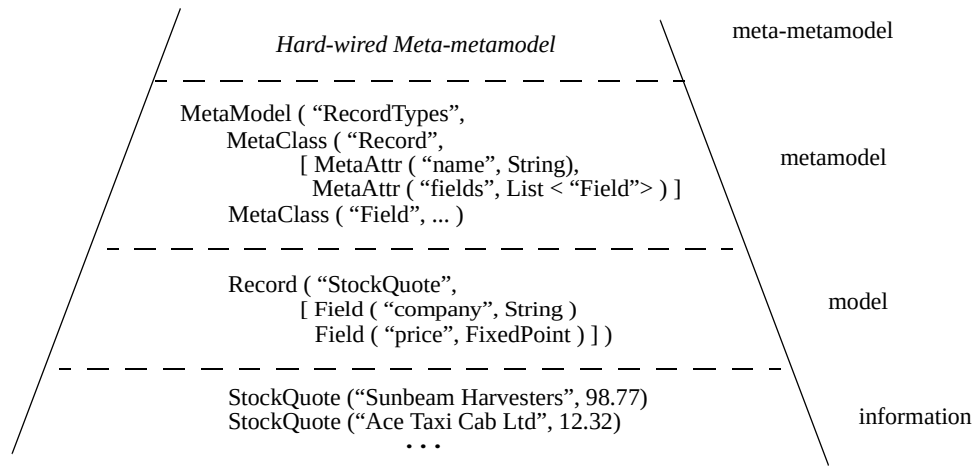


Figure 6.1 - Four Layer Metadata Architecture

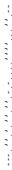
The classical four layer metadata architecture has a number of advantages over simple modeling approaches. If the framework is designed appropriately, it can:

- 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 architecture³, 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.

3. One could argue that the term “architecture” is 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.



1000

1000

- 1000

1000

- 1000

- 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.

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.

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.

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.

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.

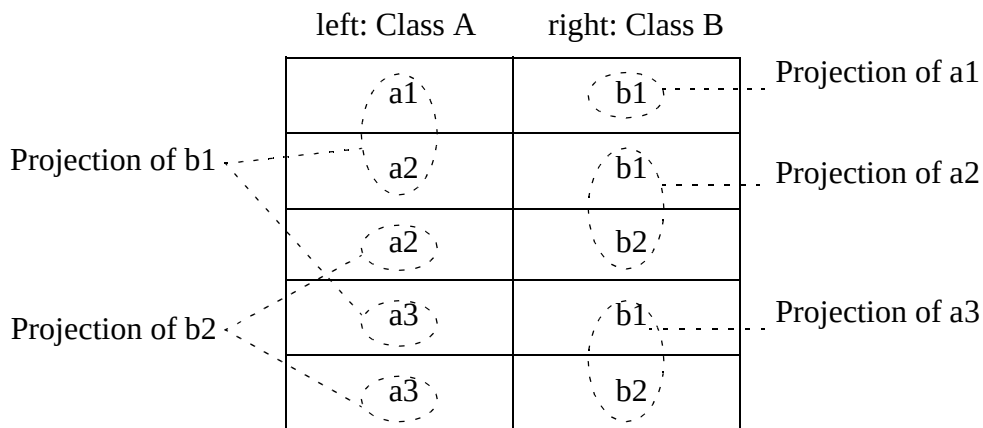


Figure 6.3 - The projections of a link set

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.

- 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.

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.

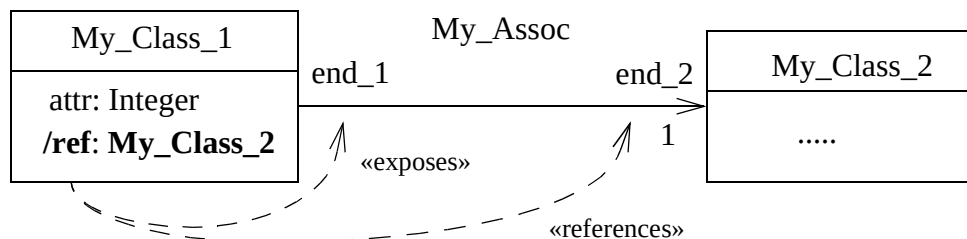


Figure 6.4 - An example of a Reference

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.

- 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).

- 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.

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 Package 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	$P1 \blacklozenge \text{---} P2$	$\underline{P1} \blacklozenge \text{---} \underline{P2}$
Generalization / Inheritance	P1 generalizes P2	$P2 - - \Rightarrow P1$	$P2 \text{---} \triangleright P1$
Importing	P1 imports P2	$P1 - - - \Rightarrow P2$	none
Clustering	P1 clusters P2	$P1 - - \Rightarrow P2$	$\underline{P1} \diamond \text{---} \underline{P2}$ 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.

- 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.

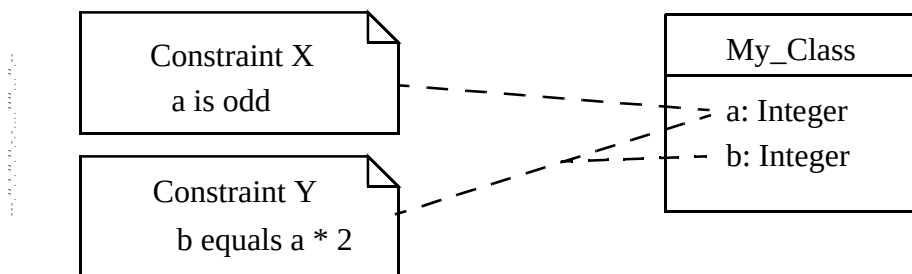


Figure 6.5 - Examples of Constraints

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

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.

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.

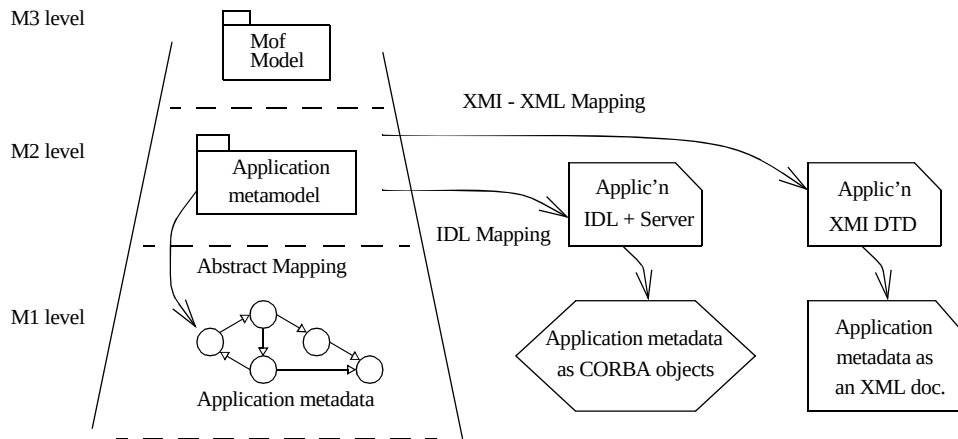


Figure 6.6 - The function of MOF Technology Mappings

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.

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.

(Blank page)

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.

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:

3.4.1 ModelElement

(abstract)

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

3.4.11 Attribute

(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.

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
<i>type:</i>	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.”
<i>multiplicity:</i>	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.
<i>changeable:</i>	This optional entry defines the “isChangeable” flag for the Attribute. If omitted, “isChangeable” is true.

Table 7.1 - Notation for Attribute Characteristics

<i>derived from:</i>	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.
<i>scope:</i>	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
<i>class:</i>	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.
<i>defined by:</i>	This entry defines the Association and AssociationEnd that the Reference is linked to via a RefersTo link.

Table 7.2 - Notation for Reference characteristics

<i>multiplicity:</i>	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.
<i>changeable:</i>	This optional entry defines the setting of the Reference’s “isChangeable” flag. If the entry is absent, the “isChangeable” flag is true.
<i>inverse:</i>	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
<i>return type:</i>	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.
<i>isQuery:</i>	This optional entry defines the Operation’s “isQuery” flag. If it is absent, the “isQuery” flag has the value false.
<i>scope:</i>	This optional entry defines the Operation’s “scope.” If it is absent, the Operation has a “scope” of “instance_level.”
<i>parameters:</i>	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.
<i>exceptions:</i>	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.
<i>operation semantics:</i>	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.

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.

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
<i>class:</i>	This entry specifies the Class whose instances are linked at this end of the Association. This is represented by the AssociationEnd’s “name” attribute.
<i>multiplicity:</i>	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.
<i>aggregation:</i>	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

- 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.

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

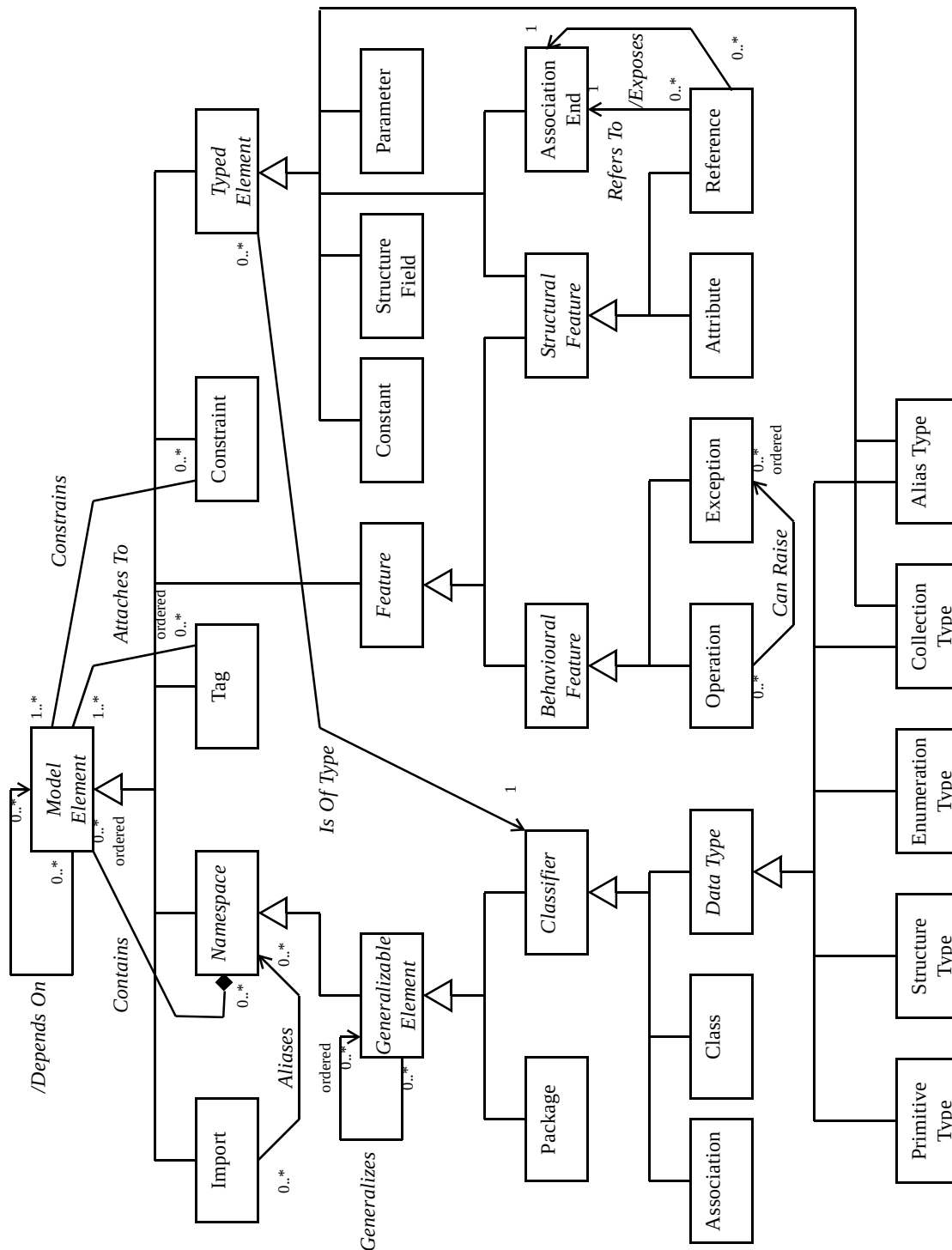


Figure 7.1 - The MOF Model Package

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.)

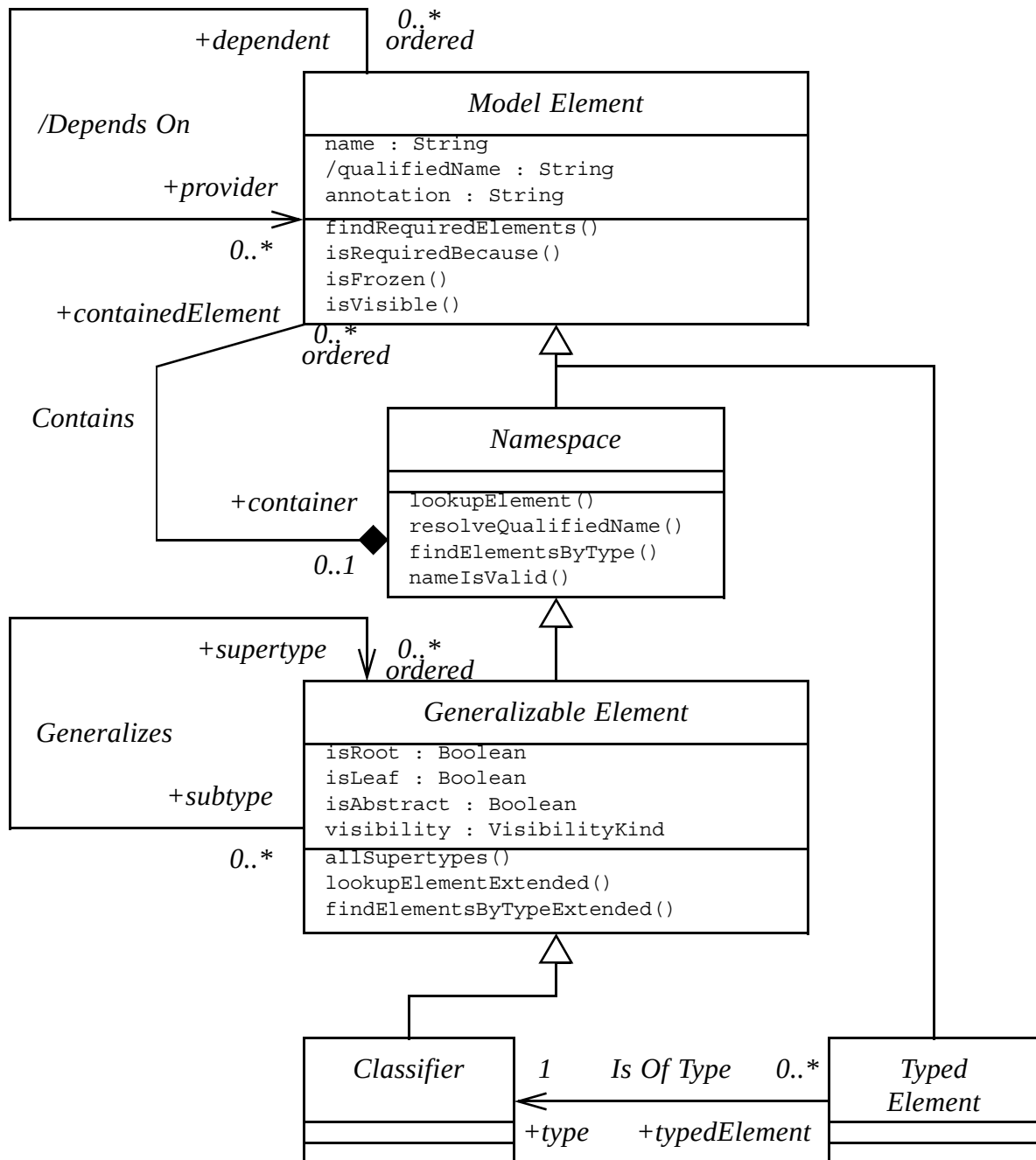


Figure 7.2 - The Key Abstractions of the MOF Model

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	Primitive Type	Structure Type	Collection Type	Enumeration Type	Alias Type	Association	Attribute	Reference	Operation	Exception	Parameter	AssociationEnd	Constraint	Constant	Import	StructureField	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.