# Common Warehouse Metamodel (CWM) Specification

**Version 1.0, 2 February 2001**

# Preface 1

## 1.1 Co-submitting Companies and Supporters

The following companies are co-submitters of the Common Warehouse Metamodel specification (hereafter referred to as CWM):

- International Business Machines Corporation
- Unisys Corporation
- NCR Corporation
- Hyperion Solutions Corporation
- Oracle Corporation
- UBS AG
- Genesis Development Corporation
- Dimension EDI

The following companies are supporters of CWM:

- Deere & Company
- Sun Microsystems Inc.
- Hewlett-Packard Company
- Data Access Technologies
- InLine Software
- Aonix
- Hitachi, Ltd
- SAS Institute Inc.
- Meta Integration Technology, Inc.

- Adaptive Ltd
- Cognos Inc.

## 1.2 Introduction

The main purpose of CWM is to enable easy interchange of warehouse and business intelligence metadata between warehouse tools, warehouse platforms and warehouse metadata repositories in distributed heterogeneous environments. CWM is based on three key industry standards:

- UML - Unified Modeling Language, an OMG modeling standard
- MOF - Meta Object Facility, an OMG metamodeling and metadata repository standard
- XMI - XML Metadata Interchange, an OMG metadata interchange standard

These three standards form the core of the OMG metadata repository architecture as illustrated in Figure 1-1.

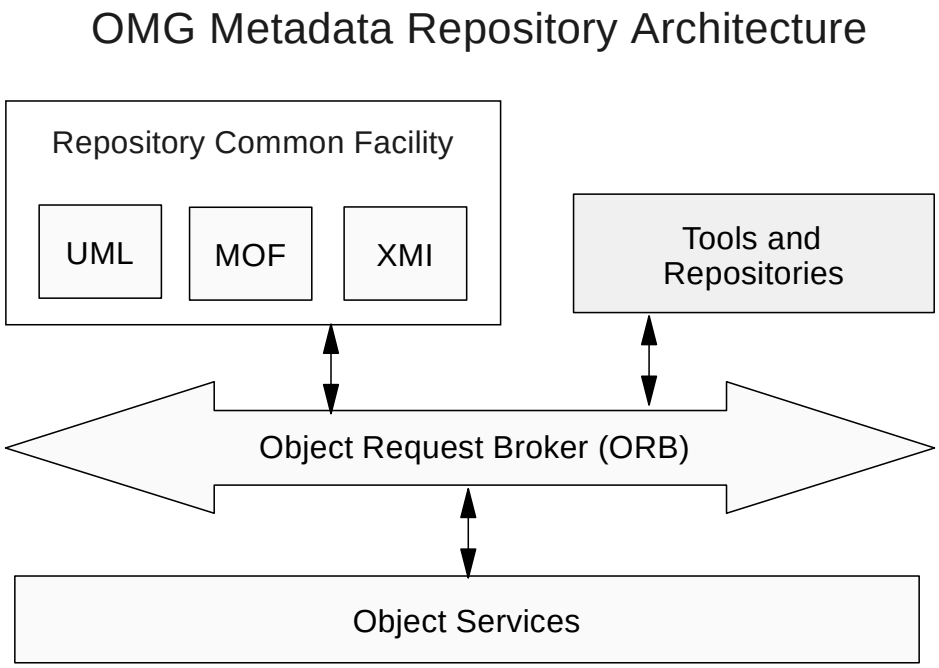


*Figure 1-1* OMG Metadata Repository Architecture

The UML standard defines a rich, object oriented modeling language that is supported by a range of graphical design tools. The MOF standard defines an extensible framework for defining models for metadata, and providing tools with programmatic interfaces to store and access metadata in a repository. The XMI standard allows

metadata to be interchanged as streams or files with a standard format based on XML. The complete architecture offers a wide range of implementation choices to developers of tools, repositories and object frameworks. XMI in particular lowers the barrier to entry for the use of OMG metadata standards.

Key aspects of the architecture include:

- A four layered metamodeling architecture for general purpose manipulation of metadata in distributed object repositories. See the MOF and UML specifications for more details
- The use of UML notation for representing metamodels and models
- The use of standard information models (UML) to describe the semantics of object analysis and design models
- The use of MOF to define and manipulate metamodels programmatically using fine grained CORBA interfaces. This approach leverages the strength of CORBA distributed object infrastructure.
- The use of XMI for stream based interchange of metadata

This specification mainly consists of definitions of metamodels in the following domains:

- Object model (a subset of UML)
- CWM foundation
- Relational data resources
- Record data resources
- Multidimensional data resources
- XML data resources
- Data transformations
- OLAP (On-line Analytical Processing)
- Data mining
- Information visualization
- Business nomenclature
- Warehouse process
- Warehouse operation

This specification defines these metamodels and provides proof of concept that covers key aspects of CWM. The specification represents the integration of work currently underway by the submitters and supporters in the areas of warehouse metadata management in distributed object environments. The submitters intend to commercialize the CWM technology within the guidelines of the OMG.

The adoption of the UML and MOF specifications in 1997 was a key step forward for the OMG and the industry in terms of achieving consensus on modeling technology and repositories. The adoption of XMI in 1999 reduced the plethora of proprietary

metadata interchange formats into one. The adoption of CWM in 2000 has solidified these core technologies by demonstrating their applicability in data warehousing and business intelligence - a major industry domain, as well as solving the most critical problem facing data warehousing and business intelligence today - metadata interchange and management.

## 1.3 Specification contact points

Please send comments on this specification to cwm-feedback@omg.org.

All questions about this specification should be directed to:

Daniel T. Chang
IBM Corporation
555 Bailey Ave., DRYA/D164
San Jose, CA 95141
Phone: +1 408 463-2319
E-mail: dtchang@us.ibm.com

Sridhar Iyengar
Unisys Corporation
25725 Jeronimo Rd.
Mission Viejo, CA 92691
Phone: +1 949 380-5692
E-mail: sridhar.iyengar2@unisys.com

Contact information for representatives of other co-submitting companies is:

Vilhelm Rosenqvist
NCR Corporation
SE-Copenhagen
Vibevej 20
DK-2400 Copenhagen NV, Denmark
Phone: +45 38 15 75 43
E-mail: Vilhelm.Rosenqvist@Copenhagen.ncr.com

John D. Poole
Hyperion Solutions Corporation
900 Long Ridge Road
Stamford, CT 06902-1135
Phone: +1 203 703-4359
E-mail: john_poole@hyperion.com

Gordon Callan
Oracle Corporation
500 Oracle Parkway, M/S 2op7
Redwood Shores, CA 94065
Phone: +1 650 506-2757
E-mail: gordon.callan@oracle.com

Hans-Peter Hoidn
UBS AG
P.O. Box, CH-8098
Zurich, Switzerland
Phone: +41 1 238 29 38
E-mail: hans-peter.hoidn@ubs.com

David S. Frankel
Genesis Development Corporation
741 Santiago Court
Chico, CA 95973-8781
Phone: +1 530 893-1100
E-mail: dfrankel@gendev.com

Chris Nelson
Dimension EDI
High Trees, Elmbridge Road
Cranleigh, Surrey GU6 8JX
England
Phone: +44 1483 271443
E-mail: chris@dimension-edi.com

Contact information for other members of the co-submitting companies is:

J. J. Daudenarde
IBM Corporation
Phone: +1 408 463-3470
E-mail: jjd@us.ibm.com

Debra LaVergne
IBM Corporation
Phone: +1 408 463-2428
E-mail: lavergne@us.ibm.com

Christoph Lingenfelder
IBM Corporation
Phone: +49 7031 16 4065
E-mail: lin@de.ibm.com

Doug Tolbert
Unisys Corporation
Phone: +1 949 380-6606
E-mail: doug.tolbert@unisys.com

Don Baisley
Unisys Corporation
Phone: +1 949 380-6382
E-mail: donald.baisley@unisys.com

David Zhang
Hyperion Solutions Corporation
Phone: +1 203 703-4875

E-mail: david_zhang@hyperion.com

David Last
Oracle Corporation
Phone: +44 118 924 6218
E-mail: david.last@oracle.com

David Mellor
Oracle Corporation
Phone: +1 781 684-5663
E-mail: david.mellor@oracle.com

Mark Hornick
Oracle Corporation
Phone: +1 781 684-7564
E-mail: mark.hornick@oracle.com

Jeffrey Peckham
UBS AG
Phone: +41 61 288 1575
E-mail: jeffrey.peckham@ubs.com

Phil Longden
Genesis Development Corporation
Phone: +44 1276 513990
E-mail: plongden@gendev.com

Steve Allman
Dimension EDI
Phone: +34 93 454 2287
E-mail: sallman@vo.lu

Anders Tornqvist
Dimension EDI
Phone: +46 31 69 30 45
E-mail: anders.tornqvist@comfact.com

Contact information for the supporting companies is:

David C. Smith
Deere & Company
E-mail: ds60162@deere.com

Chuck Mosher
Sun Microsystems, Inc.
E-mail: chuck.mosher@sun.com

Jishnu Mukerji
Hewlett-Packard Company
E-mail: jis@fpk.hp.com

Cory B. Casanave
Data Access Technologies

E-mail: cory-c@dataaccess.com

Jack J. Greenfield
InLine Software
E-mail: jack@inline-software.com

Charles E. Simon
Aonix
E-mail: chaz@aonix.com

Yuichi Yagawa
Hitachi Ltd.
E-mail: yagawa@sdl.hitachi.co.jp

Barbara Walters
SAS Institute Inc.
E-mail: Barbara.Walters@sas.com

Christian Bremeau
Meta Integration Technology, Inc.
E-mail: bremeau@metaintegration.com

Pete Rivett
Adaptive Ltd.
E-mail: pete.rivett@adaptive.com

Tina Groves
Cognos Inc.
E-mail: tina.groves@cognos.com

The submitters and supporters of the CWM specification appreciate the contributions of the following individuals during the CWM specification development process:

Ravi Dirckze, Susan Donahue, Giuseppe Facchetti, James Jonas, Robert Kemper, Suresh Kumar, Joanne Lamb, Don Lind, Tony Maresco, Bruce McLean, Karel Pagrach, William Perlman, Jeff Pinard, Curtis Sojka, Robin Noble-Thomas, Chris de Vaney, Robert Vavra, Adriaan Veldhuisen

## 1.4 Status of this Document

This document is the final draft of CWM 1.0. Refer to the OMG web site, http://www.omg.org for additional information and the status of the finalization process.

## 1.5 Guide to the Specification

This specification is presented in the following chapters:

Chapter 1 Preface

Introduces the specification and provides the context for the CWM technology within the OMG repository architecture

Chapter 2 Proof of Concept

Describes proof of concept efforts and results, in demonstration of the specification's technical viability.

Chapter 3 Response to RFP Requirements

Identifies the specific CWMI RFP requirements and this specification's response to each requirement.

Chapter 4 Design Rationale

Describes the design goals and rationale of this specification, giving an overview of the provided solution and insight into the motivation and design forces.

Chapter 5 Usage Scenarios

Describes how CWM is expected to be used by customers and tool vendors

Chapter 6 CWM

Describes the overall organization of the CWM metamodel and how it is specified.

Chapter 7 ObjectModel

Describes the ObjectModel package which contains classes and associations that serve as the base metamodel for CWM. This package is a subset of UML.

Chapter 8 Foundation

Describes the CWM Foundation package which contains classes and associations that are used by more than one CWM package.

Chapter 9 Relational

Describes the Relational package which contains classes and associations that represent metadata of relational data resources.

Chapter 10 Record

Describes the Record package which contains classes and associations that represent metadata of record data resources.

Chapter 11 Multidimensional

Describes the Multidimensional package which contains classes and associations that represent metadata of multidimensional data resources.

Chapter 12 XML

Describes the XML package which contains classes and associations that represent metadata of XML data resources.

Chapter 13 Transformation

Describes the Transformation package which contains classes and associations that represent metadata of data transformation tools.

Chapter 14 OLAP

Describes the OLAP package which contains classes and associations that represent metadata of on-line analytical processing tools.

Chapter 15 Data Mining

Describes the Data Mining package which contains classes and associations that represent metadata of data mining tools.

Chapter 16 Information Visualization

Describes the Information Visualization package which contains classes and associations that represent metadata of information visualization tools.

Chapter 17 Business Nomenclature

Describes the Business Nomenclature package which contains classes and associations that represent business metadata.

Chapter 18 Warehouse Process

Describes the Warehouse Process package which contains classes and associations that represent metadata of warehouse processes.

Chapter 19 Warehouse Operation

Describes the Warehouse Operations package which contains classes and associations that represent metadata of results of warehouse operations.

Chapter 20 Compatibility with Other Standards

Discusses how CWM is related to other industry standards.

Chapter 21 Conformance Points

Discusses compliance points in the CWM specification.

Chapter 22 CWM Data Types

Describes some widely supported data types the use of which can facilitate the interchange of metadata based on CWM. These are given as examples. This chapter is not a normative part of the CWM specification.

References

Lists the references used in this specification.

Glossary

Describes a glossary of terms relevant to CWM.

### 1.5.1 Other Parts of the Submission

Volume 2 Extensions

Contains the CWM Extensions (CWMX), which consist of: *Entity Relationship, COBOL Data Division, DMS II, IMS, Essbase, Express, InformationSet,* and *Information Reporting*. This volume is not a normative part of the CWM specification.

CWM XML, IDL and DTD files

Contain the *CWM XML*, *CWM IDL* and *CWM DTD.*

In the generation of CWM XML and CWM DTD files:

a. The CWM metamodel identifies the XML namespaces using MOF Tags.

b. No data type model is used for the metadata beyond what is directly supported by MOF.

c. Any special string encodings are described in the documentation of string valued attributes where such encodings apply.

CWMX XML, IDL and DTD files

Contain the *CWMX XML*, *CWMX IDL* and *CWMX DTD*. These files are not a normative part of the CWM specification.

In the generation of CWMX XML and CWMX DTD files:

a. The CWMX metamodel identifies the XML namespaces using MOF Tags.

b. No data type model is used for the metadata beyond what is directly supported by MOF.

c. Any special string encodings are described in the documentation of string valued attributes where such encodings apply.

CWM/CWMX MDL files

Contain the *CWM/CWMX MDL*. These files are not a normative part of the CWM specification.

# Proof of Concept 2

## 2.1 Copyright Waiver

Upon adoption by OMG, the CWM submitters grant to the OMG, a non-exclusive, royalty-free, paid-up, worldwide license to copy and distribute this specification document and to modify the document and distribute copies of the modified version. For more detailed information, see the disclaimer on the inside of the cover page of this submission.

## 2.2 Proof of Concept

CWM submitters and supporters have extensive experience in the areas of databases, data warehousing, metadata repositories, data modeling tools, CORBA and the related problems of interchange of metadata and data across tools and databases in distributed heterogeneous environments. Representative portions of their experience and ongoing proofs of concept are highlighted below:

- IBM, Hyperion, Oracle and NCR have extensive experience in implementing enterprise class data warehouses which have a critical need for metadata management. Current implementations of metadata use proprietary formats and interfaces. A fundamental goal of CWM is to open up this proprietary metadata using OMG standard metadata interfaces and interchange formats.
- Unisys, IBM and Oracle have extensive experience in implementing metadata repositories. Their contributions to OMG Meta Object Facility (MOF) and XML Metadata Interchange (XMI) form a solid foundation on which this specification has been designed. Unisys and IBM have shipped commercial implementations of object repositories and XMI based technologies.
- Oracle, NCR, Hyperion and IBM have commercial implementations of products in the areas of OLAP and Multidimensional databases - a key area addressed by this submission.

- Dimension EDI has experience in the area of statistical analysis and use of statistical metadata.
- The CWM submission is based on three key available metadata standards - MOF, UML and XMI - that have been implemented by several vendors. In addition the CWM metamodel extends UML and reuses the core concepts in UML.
- IBM, Unisys, NCR, Oracle, Dimension EDI, SAS Institute, and Meta Integration have implemented XMI for a subset of the CWM metamodel and verified interoperability in the context of database and data warehouse modeling tools.
- The CWM specification has been analyzed by end user organizations implementing data warehouses. The end user organizations that have participated in the design include UBS, John Deere and Sun. Three companies have participated in evaluation and feedback: InLine, Aonix, and Hitachi.

The submitters have demonstrated some proofs of concept as they relate to the CWM submission at the November, 1999. A more complete Enablement Showcase demonstrating cross-vendor exchange of metadata with CWM using production-quality software was provided at the December, 2000 OMG meeting.

# Response to RFP Requirements 3

## 3.1 Mandatory Requirements

The table below describes how the submission meets the mandatory requirements, as put forth in Section 6.5, Mandatory Requirements, of the Common Warehouse Metadata Interchange RFP.

| RFP Mandatory Requirement | How the submission meets the requirement |
|---|---|
| Proposals shall use the MOF as the meta-metamodel. | The CWM metamodel uses the MOF Model as its meta-metamodel. |
| Proposals shall provide a complete specification of the syntax and semantics needed to export/import warehouse metadata and the common warehouse metamodel. This may consist of a specification for common warehouse metamodel, APIs (in IDL), and/or interchange formats. | The submission includes the complete specification of the CWM metamodel and detailed descriptions of all types and associations. Also included are both IDL and DTD specifications that have been directly generated from the metamodel. These specifications define the API and interchange formats, respectively. |
| Proposals shall address the interchange of all warehouse metadata including both technical metadata and business metadata. | The CWM metamodel includes comprehensive definitions of both the technical metadata (e.g., Software Deployment, Transformation, Warehouse Process) and business metadata (e.g., OLAP, Business Information in CWM Foundation) required to represent a fully-functional data warehouse. |

| RFP Mandatory Requirement | How the submission meets the requirement |
|---|---|
| Proposals shall address the interchange of metadata that describes all warehouse data elements including data sources, transformations, and data targets. | The CWM metamodel contains a generic and powerful Transformation package that leverages core metaclasses in such a manner that transformation mappings and processes can be specified between any conceivable data source and target. A number of widely used data source/target packages are also defined, such as Relational, Record, Multidimensional, XML, and OLAP. In addition, a number of tool-specific extensions of these generic data source/target packages (e.g., IMS, DMSII, COBOLData, Essbase, Express) are included as substantive examples in Volume 2, Extensions. |
| Proposals shall address the interchange of metadata that describes all warehouse processing elements including scheduling, status reporting, and history recording. | The submission includes a complete metamodel of all warehouse processing elements. These are defined collectively by the Software Deployment, Warehouse Process and Warehouse Operation metamodels. |
| Proposals shall address the interchange of metadata that describes informational data and the use of major types of informational data models (e.g., relational, multidimensional, and hierarchical classification) for representing informational data. | The submission defines Relational, Multidimensional, XML and OLAP metamodels as the primary representations of informational data models. In addition, it includes Essbase and Express, both extensions of the Multidimensional metamodel, and Information Set, an extension of the OLAP metamodel encompassing hierarchical classifications, as substantive examples. |
| Proposals shall demonstrate support for import/export of warehouse metadata and the common warehouse metamodel. This demonstration shall include demonstration of a round-trip metadata exchange without information loss. | The submission includes XML, DTD and IDL definitions that have been generated directly from the CWM metamodel. These definitions support the import/export of the CWM metamodel and its metadata instances. The CWM Enablement Showcase at the December 2000 OMG Technical Conference demonstrated metadata interchange between tool sets from six different vendors. |
| Proposals shall support use of international standard codesets. | The CWM metamodel supports internationalization features required by specific, widely used domains (e.g., SQL-99). Use of XMI as the stream-based metadata interchange format means that the CWM metamodel and its metadata instances can be exchanged using international standard codesets supported by XML. Use of IDL as the API means that programmatic access to CWM metadata can be achieved using international standard codesets supported by IDL. |

## 3.2 Optional Requirements

The table below describes how the submission satisfies any optional requirements described in Section 6.6, Optional Requirements, of the Common Warehouse Metadata Interchange RFP.

| RFP Optional Requirement | How the submission satisfies the requirement |
|---|---|
| Proposals may address the interchange of metadata that describes operational data and the use of major types of operational data models (e.g., relational, object-oriented, and hierarchical) for representing operational data. | The submission defines Relational and Record metamodels as the primary representations of operational data models. In addition, it includes IMS, DMSII, and COBOLData, as substantive examples of extensions of the Record metamodel. |
| Proposals may address the administrative aspects of metadata interchange such as security (authorization and authentication). | The submission does not address the administrative aspects of metadata interchange such as security (authorization and authentication). These should be addressed at the MOF and XMI levels since they are applicable to all types of metadata, not just common warehouse metadata. |
| In order to preserve the investments of OMG members, proposals may be upward-compatible with MDIS, MDAPI, and/or OIM. This does not imply downward-compatibility. The CWMI specification may contain constructs unsupported by MDIS, MDAPI, or OIM. | The CWM metamodel is generally upward-compatible or aligned to some extent with these specific standards. See Chapter 20, Compatibility with Other Standards, of the submission for a more detailed discussion. |

## 3.3 Issues to be Discussed

The table below describes how the submission addresses issues described in Section 6.7, Issues to be discussed, of the Common Warehouse Metadata Interchange RFP.

| RFP Issue | How the submission addresses the issue |
|---|---|
| Proposals in response to this RFP may discuss the usage and relevance of related technologies such as MDIS. | The CWM metamodel was designed using MDIS, MDAPI and MDC-OIM as design references. See Chapter 20, Compatibility with Other Standards, of the submission for a more detailed discussion. |
| Proposals should include information on how to perform conformance tests. | Conformance points are defined in Chapter 21. |

## 3.4 Evaluation Criteria

The submission satisfies the evaluation criteria described in Section 6.8. Evaluation Criteria, of the Common Warehouse Metadata Interchange RFP. The CWM metamodel has been designed independently of any particular data warehouse implementation. The CWM uses the MOF as its meta-metamodel, the UML Notation as its graphical notation, and the XMI as its stream-based interchange format.

# Design Rationale 4

## 4.1 Design Overview

This submission proposes that XML Metadata Interchange (XMI) is used to interchange data warehouse metadata based on the CWM metamodel. The CWM metamodel is specified using the Meta Object Facility (MOF) Model, allowing XMI to be used

- to transform the CWM metamodel into a CWM Document Type Definition (DTD),
- to transfer instances of warehouse metadata that conform to the CWM metamodel as XML documents, based on the CWM DTD, and
- to transform the CWM metamodel itself into an XML document, based on the MOF DTD, for interchange between MOF-compliant repositories.

Thus these specifications work together to allow warehouse metadata and the CWM metamodel to be interchanged using W3C's Extensible Markup Language (XML).

This submission additionally proposes that IDL is used for specifying programmatic access to data warehouse metadata based on the CWM metamodel. Other programming language APIs may be generated based on the CWM IDL and specific IDL-programming language mappings (e.g. IDL-Java, CORBA-COM).

This submission specifically defines the CWM metamodel. The CWM DTD, CWM XML and CWM IDL specifications are automatically generated from the CWM metamodel, as defined by the MOF and XMI specifications.

## 4.2 CWM and the MOF

### 4.2.1 An Overview of the MOF

The Meta Object Facility (MOF) is the OMG's adopted technology for defining metadata and representing it as CORBA objects. ***Metadata*** is a general term for data

that in some sense describes information. The information so described may be information represented in a computer system; e.g. in the form of files, databases, running program instances and so on. Alternatively, the information may be embodied in some system, with the metadata being a description of some aspect of the system such as a part of its design.

The MOF supports any kind of metadata that can be described using Object Modeling techniques. This metadata may describe any aspect of a system and the information it contains, and may describe it to any level of detail and rigour depending on the metadata requirements.

The term ***model*** is generally used to denote a description of something from the real world. The concept of a model is highly fluid, and depends on one's point of view. To someone who is concerned with building or understanding an entire system, a model would include all of the metadata for the system. On the other hand, most people are only concerned with certain components (e.g. programs A and B) or certain kinds of detail (e.g. record definitions) of the system.

In the MOF context, the term ***model*** has a broader meaning. Here, a model is any collection of metadata that is related in the following ways:

- The metadata conforms to rules governing its structure and consistency; i.e. it has a common ***abstract syntax***.
- The metadata has meaning in a common (often implied) semantic framework.

Metadata is itself a kind of information, and can accordingly be described by other metadata. In MOF terminology, metadata that describes metadata is called ***meta-metadata***, and a model that consists of meta-metadata is called a ***metamodel***.

One kind of metamodel plays a central role in the MOF. A ***MOF metamodel*** defines the abstract syntax of the metadata in the MOF representation of a model. Since there are many kinds of metadata in a typical system, the MOF framework needs to support many different MOF metamodels. The MOF integrates these metamodels by defining a common abstract syntax for defining metamodels. This abstract syntax is called the ***MOF Model*** and is a model for metamodels; i.e. a meta-metamodel. The MOF metadata framework is typically depicted as a four layer architecture as shown in Table 1 below.

**Table 4-1 OMG Metadata Architecture**

| Meta-level | MOF terms | Examples |
|---|---|---|
| M3 | meta-metamodel | The "MOF Model" |
| M2 | metamodel, meta-metadata | UML Metamodel,<br>CWM Metamodel |
| M1 | model, metadata | UML models,<br>CWM metadata |
| M0 | object, data | Modeled systems,<br>Warehouse data |

Some points on OMG and MOF metadata terminology:

- To make things easier to understand, we often describe things in terms of their level in the meta-stack; e.g. the MOF Model is an M3-level model in a 4 level stack.
- The “meta-” prefix should be viewed in a relative rather than absolute sense. Similarly, the numbering of meta-levels is not absolute.
- While there are typically four layers in a MOF-based metadata stack, the number of layers can be more or less than this.

The MOF specification has three core parts; i.e. the specification of the MOF Model, the MOF IDL Mapping and the MOF’s interfaces.

## *The MOF Model*

The “MOF Model” is the MOF’s built-in meta-metamodel. One can think of it as the “abstract language” for defining MOF metamodels. This is analogous to the way that the UML metamodel is an abstract language for defining UML models. While the MOF and UML are designed for two different kinds of modeling (i.e. metadata versus object modeling), the MOF Model and the core of the UML metamodel are closely aligned in their modeling concepts. (The alignment of the two models is close enough to allow UML notation to be used to express MOF-based metamodels!)

The three main metadata modeling constructs provided by the MOF are the Class, Association and Package. These are similar to their counterparts in UML, with some simplifications:

- Classes can have Attributes and Operations at both “object” and “class” level. Attributes have the obvious usage; i.e. representation of metadata. Operations are provided to support metamodel specific functions on the metadata. Both Attributes and Operation Parameters may be defined as “ordered”, or as having structural constraints on their cardinality and uniqueness. Classes may multiply inherit from other Classes.
- Associations support binary links between Class “instances”. Each Association has two AssociationEnds that may specify “ordering” or “aggregation” semantics, and structural constraints on cardinality or uniqueness. When a Class is the type of an AssociationEnd, the Class may contain a Reference that allows navigability of the Association’s links from a Class “instance”.
- Packages are collections of related Classes and Associations. Packages can be composed by importing other Packages or by inheriting from them. Packages can also be nested, though this provides a form of information hiding rather than reuse.

The other significant MOF Model constructs are DataTypes and Constraints. DataTypes allow the use of non-object types for Parameters or Attributes. In the OMG MOF specification, these must be data types or interface types expressible in CORBA IDL.

Constraints are used to associate semantic restrictions with other elements in a MOF metamodel. This defines the well-formedness rules for the metadata described by a

metamodel. Any language may be used to express Constraints, though there are obvious advantages in using a formal language like OCL.

## The MOF IDL Mapping

The MOF's "IDL Mapping" is a standard set of templates that map a MOF metamodel onto a corresponding set of CORBA IDL interfaces. If the input to the mapping is the metamodel for a given kind of metadata, then the resulting IDL interfaces are for CORBA objects that can represent that metadata. The mapped IDL are typically used in a repository for storing the metadata.

The IDL mapping is too large to describe here, and indeed it is largely irrelevant to the problem of model interchange. Instead, we will simply note the main correspondences between elements in a MOF metamodel (M2-level entities) and the CORBA objects that represent metadata (M1-level entities):

- A Class in the metamodel maps onto an IDL interface for metadata objects and a metadata class proxy. These interfaces support the Operations, Attributes and References defined in the metamodel, and in the case of class proxy, provide a factory operation for metadata objects.
- An Association maps onto an interface for a metadata association proxy that supports association queries and updates.
- A Package maps onto an interface for a metadata package proxy. A package proxy acts as a holder for the proxies for the Classes and Associations contained by the Package, and therefore serves to define a logical extent for metadata associations, classifier level attributes and the like.

The IDL that is produced by the mapping is defined in precise detail so that different vendor implementations of the MOF can generate compatible repository interfaces from a given MOF metamodel. Similarly, the semantic specification of the mapped interfaces allows metadata objects be interoperable.

In addition to the metamodel specific interfaces for the metadata (defined by the IDL mapping), MOF metadata objects share a common set of Reflective base interfaces. These interfaces allow a 'generic' client program to access and update metadata without either being compiled against the metamodel's generated IDL or having to use the CORBA DII.

## The MOF Interfaces

The final component of the MOF specification is the set of IDL interfaces for the CORBA objects that represent a MOF metamodel. These are not of interest to the meta-modeller who will typically use vendor supplied graphical editors, compilers and generator tools to access a MOF Model repository. However, they are of interest to MOF-based tool vendors, and to programmers who need to access metadata using the Reflective interfaces.

In fact, there is not a lot to say about these interface, except to explain how they were derived. In the MOF specification, the MOF Model is defined using the MOF Model as

its own modeling language; i.e. it is the “fixed point” of the metadata stack Conceptually, the MOF Model is M3 level metadata conforming to an M4 level metamodel that is isomorphic to the MOF Model. The IDL mapping is then applied to this metamodel (or strictly speaking meta-metamodel) to produce the MOF Model’s IDL interfaces. Likewise, the MOF Model IDL’s operational semantics are largely defined by the mapping and the OCL constraints in the MOF Model specification.

### 4.2.2 The relationship between CWM and MOF

The MOF has been adopted as OMG’s standard for representing metamodels. The CWM metamodel has been designed to conform to this standard. This allows CWM to use other OMG specifications that are dependent on the MOF. In particular, it allows use of XMI to interchange warehouse metadata that is represented using the CWM metamodel, and it allows use of IDL (and other programming languages) for programmatic access to warehouse metadata based on the CWM metamodel.

## 4.3 CWM and UML

### 4.3.1 An Overview of UML

The Unified Modeling Language (UML) is a graphical language for modeling discrete systems. Although the UML is not necessarily tied to any particular application area or modeling process, its greatest applicability is in the area of object-oriented software design.

UML is the synthesis, or unification, of three preceding modeling languages that had previously dominated the field of object-oriented software development: The Booch (Grady Booch), OMT (James Rumbaugh) and OOSE (Ivar Jacobson) notational systems were combined together by their authors into the Unified Modeling Language, at Rational Software Corporation, in the 1994-1995 time frame.

The UML definition was subsequently submitted by Rational and a number of other OMG member companies, as a proposal to the Object Management Group in September, 1997, in response to an OMG RFP (OA&DTF RFP-1), requesting a standard approach to object-oriented modeling. The UML proposal was created by a team consisting of both its original authors and representatives from the various OMG submitters. The UML proposal was subsequently ratified by the OMG in November 1997. Today, UML, along with the Meta Object Facility and XML Meta Data Interchange specifications, serves as one of the cornerstones of the OMG metadata architecture (of which CWM is a proposed, domain-specific extension).

The various modeling elements of UML support the specification of both static and behavioral aspects of discrete, object-oriented systems. UML static models include the definition of classes, their attributes, operations and interfaces. Standard relationships between classes, such as inheritance/generalization, association, dependency and containment, can be specified under UML and are used in the construction of class diagrams. The behavioral semantics of the system being modeled can be specified using UML conventions for expressing time-ordered inter-object message sequencing

(sequence diagrams) and spatially-oriented collaborations between instances (collaboration diagrams). Support for the specification of state-machines is also provide for detailed modeling of object internals. UML also supports object-oriented analysis and the modeling of external system behavior through use case diagrams. Finally, UML provides notations for specifying the packaging of a logical design into components and the deployment and allocation of those components to nodes in a distributed computing architecture.

The UML language is formally defined by a metamodel (or semantic model) which is itself defined recursively, using UML. This meta-circular definition enables the entire UML to be based on a small number of elementary terms.

### *4.3.2 The relationship between CWM and UML*

A primary objective of the CWM is to define a metamodel (or, equivalently, a "metadata model") of a generic data warehouse architecture. Thus, the CWM metamodel defines formal rules for modeling instances of data warehouses. However, there is also a requirement for the CWM metamodel to be expressed in MOF (and thus enabled for interchange via either CORBA interfaces or XMI).

The CWM metamodel includes an Object Model package which is based on the UML metamodel. It consists of a version of the UML metamodel in which those aspects that are not relevant in a data warehouse scenario have been removed. This Object Model serves two purposes:

- as the base on which the CWM metamodel is built, and
- as the metamodel for object-oriented data resources.

The CWM metamodel is effectively an extension of the UML-based Object Model. Any metaclass within CWM ultimately (if not directly) inherits from some metaclass of the Object Model. For example, consider the CWM Relational Package. The Relational metamodel defines a metaclass called "Table" that represents any relational database table. This metaclass derives from the Object Model metaclass "Class". Similarly, the Relational metaclass "Column" derives from the Object Model metaclass "Attribute". This formally establishes the semantic relationship between the relational concepts of Table and Column that it is well-understood intuitively, i.e. that a Table is "something" that has properties (or attributes) and serves as a template for a collection of "things" (i.e. rows) that all share that same set of properties but supply their own "values" of those properties. The semantic equivalent in UML is the notion of a Class and its Attributes, and this equivalence is established by defining Table as a specialization of the notion of Class, and Column as a specialization of Attribute.

The UML specification is also used in the following ways:

- The UML notation is used in the diagrammatic representations of the CWM metamodel.
- Additional constraints on the CWM metamodel are represented in Object Constraint Language (OCL), as defined in the UML specification.

## 4.4 CWM and XMI

### 4.4.1 An Overview of XMI

The purpose of XMI is to allow the interchange of models in a serialised form. Since the MOF is the OMG's adopted technology for representing metadata, it is natural that XMI focuses on the interchange of MOF metadata; i.e. metadata conforming to a MOF metamodel. In fact, XMI is really a pair of parallel mappings: one between MOF metamodels and XML DTDs, and another between MOF metadata and XML documents.

XMI can be viewed as a common metadata interchange format that is independent of middleware technology. Any metadata repository or tool that can encode and decode XMI streams can exchange metadata with other repositories or tools with the same capability. There is no need for products to implement the MOF-defined CORBA interfaces, or even to "speak" CORBA at all.

XMI provides a possible route for interchange of metadata with repositories whose metamodels are not MOF based. This interchange can be realised by ad hoc mappings between an XMI document and the repository's native metamodel.

XMI is based on the W3C's Extensible Markup Language (XML), and has two major components:

- The *XML DTD Production Rules* for producing XML Document Type Definitions (DTDs) for XMI encoded metadata. XMI DTDs serve as syntax specifications for XMI documents, and allow generic XML tools to be used to compose and validate XMI documents.
- The *XML Document Production Rules* for encoding metadata into an XML compatible format. The production rules can be applied in reverse to decode XMI documents and reconstruct the metadata.

XMI supports the interchange of any kind of metadata that can be expressed using the MOF specification. It supports the encoding of metadata consisting of both complete models and model fragments, as well as tool-specific extension metadata. XMI has optional support for interchange of metadata in differential form, and for metadata interchange with tools that have incomplete understanding of the metadata.

### 4.4.2 The relationship between CWM and XMI

CWM uses XMI as its interchange mechanism. This means that the full power and flexibility of XMI is available for interchanging both warehouse metadata and the CWM metamodel itself. CWM does not require any extensions to XMI.

A standard DTD for the CWM metamodel is generated using XMI's DTD Production Rules. Warehouse metadata can then be encoded as an XML document using XMI's Document Production Rules.

A standard XML document for the CWM metamodel is also generated using XMI's Document Production Rules, based on the MOF DTD.

## 4.5 Major Design Goals and Rationale

This section describes the major design goals that the CWM developers are aiming to meet, and explains some of the more significant design choices that have been made.

### 4.5.1 Reuse of UML concepts

**Design Goal: The CWM model should reuse concepts in the UML metamodel where applicable.**

The CWM metamodel has as its base an Object Model based on a version of the UML metamodel in which those aspects that are not relevant in a data warehouse scenario have been removed. The CWM metamodel is built on top of and extends this Object Model.

Many of the core UML object types and associations are reflected by the CWM Object Model. Wherever appropriate, Object Model types are subtyped to provide more specific object types in the CWM metamodel, normally with additional attributes or associations. All CWM object types are direct or indirect subtypes of appropriate Object Model types, and so inherit their attributes and associations.

This approach has many advantages. It allows the CWM specification to capitalise on the substantial investment in developing and refining the UML metamodel. The general awareness of UML concepts should aid understanding of the CWM specification and its base Object Model. It also enables easy inclusion of UML models as part of the data warehouse metadata.

### 4.5.2 Modularity

**Design Goal: The CWM model should be subdivided into packages allowing implementation of relevant subsets of the model.**

The CWM metamodel is split up into a set of packages. This should aid comprehension of the metamodel, by splitting it up into smaller units, and also allow users and implementors to ignore packages that are not relevant to their needs.

The CWM metamodel has a layered structure:

- The foundation consists of the UML-based Object Model and the CWM Foundation, which supports additional concepts and structures that are shared by other packages. Additionally, the Software Deployment package supports the deployment information for the data sources and targets in the next layer.
- The Relational, Record, Multidimensional and XML packages support the definition of various types of data sources and data targets.
- The Transformation, OLAP, Data Mining, Information Visualization and Business Nomenclature packages define the transformations and analytical processing that takes place on these data sources.

- Finally, the Warehouse Process package supports scheduling information, and the Warehouse Operation package is used to record operational details such as the results of transformation runs.

### 4.5.3 Generic model

**Design Goal: The CWM metamodel should be independent of any specific data warehouse implementation. Yet it should contain features that are effective in, and mappable to, a broad range of representative warehouse configurations based on specific tools.**

Much attention has been taken to ensure that the CWM metamodel has been made as generic as possible, and that only information that is shareable between different implementations has been included in the metamodel. Shareability of information has been checked and refined by examining the metadata needs of several different, but representative, implementations as well as a broad range of representative warehouse configurations.

# Usage Scenarios 5

## 5.1 Overview

This chapter describes some of the problems that data warehousing users, administrators, developers and vendors face today and illustrates how CWM helps to address these problems.

As stated in Section 4.5.3, a design goal of CWM is to be independent of any specific data warehouse implementation and to contain features that are effective and easy to use in a broad range of representative warehouse configurations based on specific tools. The usage scenarios contained in this chapter are provided to demonstrate that this design goal is met.

In addition these usage scenarios illustrate problem domains in which CWM is applicable.

## 5.2 Users of CWM

CWM is targeted at six categories of users:

- Warehouse platform and tool vendors
- Professional service providers
- Warehouse developers
- Warehouse administrators
- End users
- Information technology managers

These users participate in one or more of the following four stages in the development and usage of CWM-based data warehouses:

- Establishment
  Implementing and deploying CWM, including a Repository Common Facility (as shown in Fig. 1-1. OMG Metadata Repository Architecture).
- Build
  Exercising CWM to define a baseline data warehouse configuration (establishing the exchange paths between known data sources and targets).
- Operation
  Operating the CWM-based data warehouse.
- Maintenance
  Exercising CWM to define changes in data warehouse configuration (to cover changes as small as the addition of more elements of a type already in the configuration and as large as merger with or replacement by another configuration).

In this chapter, we present usage scenarios that illustrate activities in the Build and Maintenance steps.

The following table shows how CWM benefits users in data warehouse development and usage.

**Table 5-1 Value of CWM to Users**

| | | *Value of CWM to Users* | | |
|---|---|---|---|---|
| ***User Category*** | **Stage** | **Problem or Need** | **Tools and Repositories** | **How CWM promotes better Data Warehouse utilization** |
| **Warehouse platform and tool vendors** | Build | Must subscribe to standards for inter-vendor interconnect | · CWM<br>· OMG Repository Common Facility<br>· Tools for modeling, development, deployment and system management | CWM provides a common backplane for pluggable subsystems. It is a globally usable notation for metadata exchange protocols which enables flexible distribution of enterprise services over a heterogeneous collection of systems. |
| **Professional service providers** | Build | Must accumulate and reuse objects from service engagement | Third party and in-house tools that apply CWM metadata to concrete database catalogs and vice versa | Reusable, editable, and extensible CWM metadata provides an asset base that builds value. This base of reusable objects starts a self-reinforcing feedback loop with continually increasing returns (improved engagement productivity). |

**Table 5-1 Value of CWM to Users**

| | | *Value of CWM to Users* | | |
|---|---|---|---|---|
| ***User Category*** | **Stage** | **Problem or Need** | **Tools and Repositories** | **How CWM promotes better Data Warehouse utilization** |
| **Professional service providers** | Maintenance | Must modify configuration: knowing what and where to modify; knowing dependency closure | Third party or in-house tools to manage reconfiguration editing of a warehouse model | CWM exposes the information required to modify a model. Context definition and self-describing features of CWM are used to isolate dependency relationships. |
| **Professional service providers, warehouse administrators** | Maintenance | Must integrate existing tools and data which adhere to standards other than CWM into a data warehouse configuration. | Tools based on CWM's ability to incorporate metamodels of legacy, web, proprietary, and alternate metadata definition practices and standards. | CWM provides submodels supporting details of information held in a variety of different formats, including XML, Relational SQL and conventional file formats. |
| **Warehouse administrators** | Build | Must establish and manage expressions, relationships, and lineage over multiple database schemata. | Tools that use built-in facilities of CWM to define schema content, relationships, and lineage. | CWM design is based on need to manage such information at multiple levels. The Transformation and Warehouse Operation packages are designed to allow navigation of metadata correlated to schemata. |
| **Warehouse administrators** | Maintenance | Must add, subtract, re-partition, reallocate, or merge resources in deployment configuration. | System management tools. | CWM consists of models of metadata that assist in making such changes and allow impact of these changes to be assessed. |

**Table 5-1 Value of CWM to Users**

| | | *Value of CWM to Users* | | |
|---|---|---|---|---|
| ***User Category*** | **Stage** | **Problem or Need** | **Tools and Repositories** | **How CWM promotes better Data Warehouse utilization** |
| **Warehouse developers** | All | Must view source, target, application descriptions (including interfaces). | Tools to facilitate development with ability to refer to information in metadata repository | CWM includes containers for description at fine and coarse grain levels. |
| **End users** | All | Must know<br>- refresh state of inputs and outputs of queries,<br>- mapping between models for transfer of data sets between tools, and<br>- transforma-tion rules. | Query and presentation tools | CWM presents models of metadata to be exploited by query and presentation tools. |
| **Information technology managers** | All | Must have visibility into warehouse deployment state. | System management and report tools | CWM presents models of metadata to be exploited by system management and report tools. |

## 5.3 Usage Scenarios

This section identifies four application scenarios and six tool scenarios outlining likely usages of CWM. The application scenarios cover key data warehousing activities. These are summarized in sections 5.3.1 through 5.3.4 below and illustrated in Table 5-2. The tool scenarios in section 5.3.5 and Table 5-3 cover some significant data warehousing tools from the submitters used in present day practice.

The purpose of these scenarios is purely to illustrate potential usage of CWM.

In warehouse operations, two common categories of data movement are (a) loading data into a data store, and (b) accessing data for analysis and presentation from the data store. The ETL Scenario addresses the first category. The OLAP Scenario addresses the second category.

### 5.3.1 ETL Scenario

ETL, Extract-Transform-Load, is a common term for the warehouse load process comprising a set of data movement operations, each from a data source to a data target with some transforming or restructuring logic applied.

The ETL Scenario starts by defining a CWM Transformation model for movement from a data source to a data target. Parameters of the source data, target data, and transformation logic are assigned values in the model. Source data parameters depend on the type of the data source (object-oriented, relational, record-oriented, multidimensional, or XML). Target data parameters are similarly chosen. Transformation logic parameters include identification of a transformation component and of data sources and data targets. The transformation component is a method composed of a possibly large hierarchy of components (commercial tools, commercial libraries, custom scripts) whose detailed structure is defined elsewhere.

An ETL process is realized by a number of components across several CWM packages. A CWM warehouse process may launch an ETL process as a scheduled operation consisting of a number of transformation steps executed in sequence.

For example, the first transformation consists of the extraction and filtering of data from any of a number of possible data sources. A second transformation cleanses, combines, or otherwise reduces the data and then stores it in a normalized format in some primary relational database of the warehouse. A third transformation selects certain rows from the primary relational database and loads their values into the input cells of a multidimensional database. Finally, the CWM warehouse process might instruct the multidimensional database to re-calculate its aggregated cells based on the new input data.

### 5.3.2 OLAP Scenario

An end user of a data warehouse engages in an analytic session with a multidimensional or relational database through the OLAP layer. The user navigates cubes and dimensions and selectively launches OLAP queries. At some point, the user decides to drill-down from a consolidated value to lower levels of detail in an OLAP hierarchy, possibly down to the lowest level input value(s) in the hierarchy.

Leveraging CWM's inherent ability to preserve data lineage, the user may view the operational detail which formed the input value(s). The original data sources can be identified from the CWM Warehouse Operation that recorded the production of the input value(s).

### 5.3.3 Questionnaire Scenario

An important aspect of data warehousing is the collection of raw data from external resources using for example application-generated reports, questionnaires or surveys. To allow for inter-operability of tools supporting raw data collection, the metadata identifying the data to be collected must be defined, together with metadata that can be used to ensure accuracy and validity of data.

Questionnaires are commonly used as a means of collecting data about real-world activities, processes, and opinions. Most of us experience questionnaires as paper documents. However, technological advances are making possible on-line acquisition of questionnaire data and generation of questionnaires from automated sources, such as application systems.

Once assimilated, questionnaire data can be stored in data warehouses for further statistical processing and analysis. The inherent multi-category, hierarchical nature of questionnaire responses makes them ideal candidates for multidimensional analysis. Once questionnaire data has been transformed by an ETL process into a multidimensional data store, it becomes available for analysis with OLAP tools.

### *5.3.4 Warehouse Administration Scenario*

A warehouse administrator needs access to all the necessary information to control and monitor the state of the data warehouse. To accomplish this, ETL processes need to be scheduled to update information in the data warehouse. Monitoring ETL operations and journalizing changes to data warehouses must be performed for a variety of data integrity, organizational and regulatory reasons. In the event of problems arising, the administrator needs to be able to take appropriate action (such as initiating a rerun of a set of warehouse processes).

For information held in the data warehouse, the administrator may need to determine its source, derivation, and update history. This involves identifying transformations that created the information and determining when they last ran. Because the source of a transformation may itself be another transformation, it may be necessary for the administrator to track backward through several transformations to identify the original source(s) of the information.

**Table 5-2 Application Scenarios**

| | ***Application Scenario*** | | | |
|---|---|---|---|---|
| ***CWM Package*** | **ETL (Extract, Transform, Load)** | **OLAP** | **Questionnaire** | **Warehouse Administration** |
| **Software Deployment** | X | X | | X |
| | | | | |
| **Object-Oriented (UML)** | X | | | |
| **Relational** | X | X | X | |
| **Record** | X | | X | |
| **Multi-dimensional** | X | X | | |
| **XML** | X | | X | |

**Table 5-2 Application Scenarios**

| | *Application Scenario* | | | |
|---|---|---|---|---|
| ***CWM Package*** | **ETL (Extract, Transform, Load)** | **OLAP** | **Questionnaire** | **Warehouse Administration** |
| | | | | |
| **Transformation** | X | X | X | X |
| **OLAP** | | X | X | |
| **Data Mining** | | | | X |
| **Information Visualization** | | X | X | X |
| **Business Nomenclature** | X | X | X | X |
| | | | | |
| **Warehouse Process** | X | X | | X |
| **Warehouse Operation** | X | X | | X |

### *5.3.5 Tool Scenarios*

The following tool scenarios cover some significant data warehousing tools from the submitters used in present day practice:

- Dimension EDI

  Polyval XML Mediator, Polyval XML Questionnaire

- Hyperion

  Hyperion Essbase OLAP Server, Hyperion Integration Server, Hyperion Application Link, Hyperion Analytical Reporting

- IBM

  Visual Warehouse, DB2 Family, DB2 OLAP Server, IMS, Team Connection

- NCR

  Teradata Warehouse Suite

- Oracle

  Oracle Express, Oracle 8i, Oracle Discoverer, Oracle Warehouse Builder, Oracle Repository

- Unisys

  Unisys Universal Repository (UREP)

**Table 5-3 Tools Scenarios**

| | *Tools Scenario* | | | | | |
|---|---|---|---|---|---|---|
| ***CWM Package*** | **Dimension EDI** | **Hyperion** | **IBM** | **NCR** | **Oracle** | **Unisys** |
| **CWM and Metadata Repository Facility** | | X | X | | X | X |
| | | | | | | |
| **Software Deployment** | | | X | X | X | |
| | | | | | | |
| **Relational** | X | X | X | X | X | X |
| **Record** | | | X | X | | X |
| **Multi-dimensional** | | X | | | X | |
| **XML** | X | | X | | X | X |
| | | | | | | |
| **Transforma-tion** | X | X | X | X | X | |
| **OLAP** | | X | X | X | X | |
| **Data Mining** | | X | X | X | X | |
| **Information Visualization** | X | X | X | | X | |
| **Business Nomenclature** | X | X | X | | X | |
| | | | | | | |
| **Warehouse Process** | | | X | X | X | |
| **Warehouse Operation** | | | X | X | X | |

# CWM 6

## 6.1 Overview

The amount of data in a given organization doubles every five years. Most organizations suffer from an overabundance of redundant and inconsistent data that is difficult to manage effectively, to access, and to use for decision making purposes. *Data warehousing* provides an excellent approach for transforming data into useful and reliable information to support the business decision making process and to achieve *business intelligence*. One of the most important aspects of data warehousing is *metadata*. Metadata is used for building, maintaining, managing, and using the data warehouse. Unfortunately, the proliferation of data management and analysis tools has resulted in almost as many different representations and treatments of metadata as there are tools.

Since every data management and analysis tool requires different metadata and a different metadata model (known as a metamodel) to solve the data warehouse metadata problem, it is simply not possible to have a single metadata repository that implements a single metamodel for all the metadata in an organization. Instead, what is needed is a standard for interchange of warehouse metadata.

The CWM is a response to these needs. It provides a framework for representing metadata about data sources, data targets, transformations and analysis, and the processes and operations that create and manage warehouse data and provide lineage information about its use.

The CWM Metamodel consists of a number of sub-metamodels which represent common warehouse metadata in the following major areas of interest to data warehousing and business intelligence (see Figure 6-1 on page 52):

- Data Resources

  These include metamodels that represent object-oriented, relational, record, multidimensional, and XML data resources. In the case of object-oriented data resource, CWM reuses the base object model.

- Data Analysis

  These include metamodels that represent data transformations, OLAP (On-line Analytical Processing), data mining, information visualization, and business nomenclature.

- Warehouse Management

  These include metamodels that represent warehouse processes and results of warehouse operations.

## The CWM Metamodel

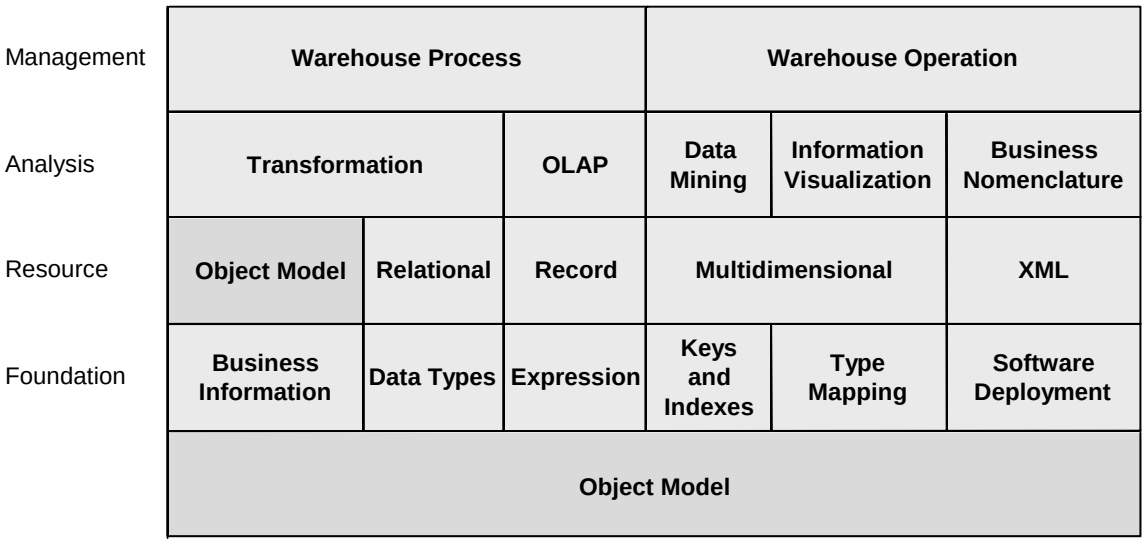


*Figure 6-1* The CWM Metamodel.

The CWM Metamodel is designed to maximize the reuse of Object Model (a subset of UML) and the sharing of common modeling constructs where possible. The most prominent example is that CWM reuses/depends on Object Model for representing object-oriented data resources. In addition, where applicable, key elements of the metamodels for other types of data resources all subclass from the same model elements in Object Model, as shown in Table 6-1 below. (The entries listed under Software System and Deployed Software System are examples.)

**Table 6-1 CWM Data Resources**

| | Software System | Deployed Software System | Package | Class | Attribute |
|---|---|---|---|---|---|
| **Object Model** | Java | Java installation | Package | Class | Attribute |
| **Relational** | DB2 UDB, Oracle 8i, Teradata | DB2 UDB, Oracle 8i, Teradata installations | Catalog/ Schema | Table | Column |
| **Record** | IMS, DMS II | IMS, DMS II installations | RecordFile | RecordDef | Field |
| **Multidimensional** | Essbase, Express | Essbase, Express installations | Schema | Dimension | Dimensioned Object |
| **XML** | XML4J | XML4J installation | Schema | ElementType | Attribute |

### *6.1.1 The Roles of UML in CWM*

UML is used in CWM in three different critical roles:

- UML is used as the MOF-equivalent meta-metamodel.
  UML, or the part that corresponds to the MOF Model, UML Notation, and OCL (Object Constraint Language) are used as the modeling language, graphical notation, and constraint language, respectively, for defining and representing CWM.
- UML is used as the foundation metamodel.
  UML, specifically a subset as represented by the Object Model packages, is used as the foundation of CWM from which other metamodels inherit classes and associations
- UML is used as the object-oriented metamodel.
  UML, specifically the Object Model package, is relied on for representing object-oriented data resources.

## 6.2 Organization of the CWM

The CWM Metamodel uses packages and a hierarchical package structure to control complexity, promote understanding, and support reuse. The model elements are contained in the following packages:

- ObjectModel package
  - Core package
    Contains classes and associations that form the core of the CWM object model, which are used by all other CWM packages including other ObjectModel packages.
  - Behavioral package
    Contains classes and associations that describe the behavior of CWM objects and provide a foundation for describing the invocations of defined behaviors.
  - Relationships package
    Contains classes and associations that describe the relationships between CWM object.
  - Instance package
    Contains classes and associations that represents instances of CWM classifiers.
- Foundation package
  - Business Information package
    Contains classes and associations that represent business information about model elements.
  - Data Types package
    Contains classes and associations that represent constructs that modelers can use to create the specific data types they need.
  - Expressions package
    Contains classes and associations that represent expression trees.
  - Keys and Indexes package
    Contains classes and associations that represent keys and indexes.
  - Software Deployment package
    Contains classes and associations that represent how software is deployed in a data warehouse.
  - Type Mapping package
    Contains classes and associations that represent mapping of data types between different systems.
- Resource package
  - Relational package
    Contains classes and associations that represent metadata of relational data resources.
  - Record package
    Contains classes and associations that represent metadata of record data resources.
  - Multidimensional package
    Contains classes and associations that represent metadata of multidimensional data resources.

  - XML package
    Contains classes and associations that represent metadata of XML data resources.

- Analysis package
  - Transformation package
    Contains classes and associations that represent metadata of data transformation tools.
  - OLAP package
    Contains classes and associations that represent metadata of on-line analytical processing tools.
  - Data Mining package
    Contains classes and associations that represent metadata of data mining tools.
  - Information Visualization package
    Contains classes and associations that representing metadata of information visualization tools.
  - Business Nomenclature package
    Contains classes and associations that represent metadata on business taxonomy and glossary.

- Management package
  - Warehouse Process package
    Contains classes and associations that represent metadata of warehouse processes.
  - Warehouse Operation package
    Contains classes and associations that represent metadata of results of warehouse operations.

### 6.2.1 Modeling Conventions

To promote clearer understanding of the contents of the CWM metamodels, this specification contains a number of UML representations of portions of the CWM model packages. The CWM design team has used several conventions in the construction of CWM metamodel packages and accompanying diagrams. These conventions are outlined here and apply to the remainder of the specification.

#### *Names*

The names of UML *packages, classifiers,* and *associations* are shown with the initial letter of their names in upper case; these names must be unique within a package. *Features* (attributes and operations), *references* and *association ends* are shown with the initial letter of their names in lower case; these names must be unique within their containing classifier or association. Internal upper case letters are used in both types of names to separate words; intervening spaces, hyphens, or underscores have been avoided. Double colon delimiters ("::") are used to connect individual names into qualified names.

#### *Classes*

Conforming to standard UML notation, classes are represented in diagrams as rectangular boxes with three horizontal sections containing the class name, attributes, and operations, respectively, from the top. CWM itself defines no operations, but extension packages are permitted to do so.

Classes defined in the current CWM package are shown with all their attributes and operations visible. Classes imported from UML or other CWM packages show only the class name and a notation in parentheses identifying the source package. Attributes and operations of imported classes are not displayed; refer to the package where they are defined to see their complete definition.

In diagrams, classes defined in any CWM package are shown with lightly shaded background fill, whether imported or local. Classes imported from a UML package are shown with no background fill.

## *Attributes*

Unless specified otherwise in the specification, attributes have a multiplicity of *exactly one*; attribute multiplicity is not shown in diagrams. Attributes are shown diagrammatically following standard UML notation:

*<<stereotype>> name : type = initvalue.*

Attribute stereotypes and initial values are suppressed in diagrams if they are not defined.

## *Data Types*

Metamodel (M2) data types are placed in the most specific package possible and have a stereotype of *<<primitive>>*, *<<datatype>>* or *<<enumeration>>*.

Enumerations are used infrequently within the CWM. In diagrams, the names of enumerations appear only as the types of attributes; their individual values are not displayed. The defined values for an enumeration begin with a lower case letter and can be found in the text where the enumeration is used as the type of an attribute. For example, the values of the *WeekDay* enumeration used as the type of an attribute named *dayOfWeek* would appear in the text as follows:

***dayOfWeek***

The day of the week on which something interesting happened.

| | |
|---|---|
| *type:* | WeekDay ( sunday \| monday \| tuesday \| wednesday \| thursday \| friday \| saturday ) |
| *multiplicity:* | exactly one |

Data types required by CWM extension packages, such as the types of a programming language, are generally represented as instances of the UML DataType class or as instances of other classes that are subclasses of UML's Classifier class. Refer to the Foundation and Data Types chapters for additional details.

## Associations

All CWM associations are named. However, to improve readability, their names usually do not appear in diagrams.

Associations declared in UML and other CWM packages can be reused in two ways: inheritance or derivation. *Inherited associations* are reuses, without modification, of associations declared elsewhere in the metamodel. In contrast, *derived associations* are "filtered" by OCL statements so that only a subset of the source association's instances are available in the derived association.

Inherited associations are never renamed and are added to the diagrams only when they clarify the relationships between types appearing in the diagram. They can be identified in diagrams by leading forward slash characters ("/") on the names of their association ends. For example, the association between a relational table and its columns can be drawn between the Table and Column classes with end names of "/owner" and "/feature", indicating that the association is an application of the UML association between the Classifier and Feature classes.

Derived associations are separately named and have a real presence in the metamodel. They do not have slashes on the names of association ends. One "filtering" OCL statement on at least one association end is required. (Note, however, that OCL statements that simply restrict the multiplicity of inherited association ends are not sufficient to turn them into derived associations.)

*Shared* (open diamond) *aggregation associations* have been avoided unless there was not other way of representing the required semantics. UML *association classes* have been avoided because MOF 1.3 does not support them.

## Association Ends

All association ends are named in CWM. By default, the names of association ends are the same as the names of the classes to which they connect. Association end names are defined only within the scope of the association whose ends they name. The names of association ends appear in the diagrams only when they have some name other than the default or when their presence is required to clarify the meaning or identity of the association (as with inherited associations appearing on diagrams).

Generally, all CWM association ends are navigable. In the diagrams, navigable association ends are marked with an arrowhead when the opposite end is non-navigable for some specific semantic reason. Such situations are considered rare, occur only when associations cross package boundaries and are dependent on the specific semantics of each situation.

## References

Because it is based on the MOF, CWM distinguishes *references* and *association ends*. References appear as attributes of classes and indicate related instances of the class that is the attribute's stated type. The names of references are preceded by forward slashes ("/") in diagrams. Association ends, in contrast, appear as labels on the ends of lines representing associations.

It is appropriate to think of a reference as being "implemented" by a corresponding association end of an association between the reference's class and the class represented by the reference's type. Reference names are generally identical to their corresponding association end's name. However, reference names may differ from end names when doing so improves the clarity of the model.

References may be omitted if traversal to the associated class is either not possible, as is often the case when crossing package boundaries, or not desirable for some other semantic reason. For example, references should be omitted when the association end they correspond to resides in another package or when they would interfere with federation across network metadata repositories (refer to the MOF specification for details).

Examples of these relationships are illustrated in the following figure.

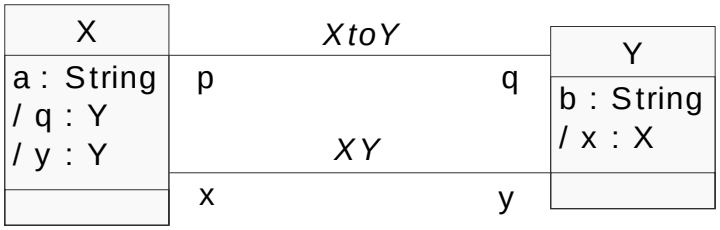


*Figure 6-2* References and Association Ends

In the figure, *X.a* and *Y.b* are attributes of type *String* in classes *X* and *Y*, respectively. *X.y* is a reference from class *X* to class *Y*, and *Y.x* is a reference from *Y* to *X*. Although CWM does not specify implementation details, this pair of references can be thought of as being implemented by the *XY* association, with the *XY.x* association end implementing the reference *Y.x* and *XY.y* implementing *X.y*. Consequently, *X.y* and *Y.x* are mutually inverse references. Similarly, *X.q* is implemented by *XtoY.q* but has no inverse reference. Because the inverse reference is not defined, instances of *X* cannot be directly accessed from *Y*. However, related instances of *X* still can be found by examining the *XtoY* association itself. This technique is commonly used when an association crosses a package boundary, and a reference cannot be added to the class in the other package (*Y*, in this case).

If traversal from *Y* to *X* were not semantically valid, the *XtoY* association could be so-marked with an open arrowhead at the *q* association end, pointing to *Y* (but not shown in the figure).

## Constraints

Constraints are statements of “facts” assumed to be true always and are core parts of any expressive metamodel.

CWM constraints are expressed in two ways. Some constraints are represented in the structure of the model itself. These constraints take the form of multiplicity statements for attributes and association ends, positioning of containment and inheritance relationships, and the abstractness of some metaclasses. Other constraints are, following OMG requirements, expressed as OCL statements. Within the body of a chapter, these constraints are described in text, and corresponding OCL statements are referenced by number and enclosed in square brackets. For example, a reference to the third OCL statement in a chapter would appear as “[C-3]”. OCL statements within a chapter are numbered sequentially from C-1 and collected together in a section at the end of chapter. Because the Foundation chapter contains an additional layer of subsections, constraint numbers in it include the subsection number; for example, “[C-2-1]” is the first constraint in the second subsection of the chapter.

A complete description of CWM includes both structural constraints and their accompanying OCL statements. Structural constraints are used to capture general features of CWM and are generally preferred over OCL statements. OCL statements are used when capturing a constraint structurally would overly complicate or otherwise obscure the basic intent and understanding of the metamodel. OCL statements are written to capture specific situational constraints (such as uniqueness, filters for derived associations, and dependencies between attribute values) or to express relationships between instances that cannot be inferred from the metamodel itself (such as “or-ed” or “xor-ed” associations and attributes, references to superclasses or other related instances, subsets, and implied transitivity).

Following the ground rules of OCL, CWM does not specify the nature of actions taken when constraints fail. Rather, specification of failure actions and recognition of failure modes are left to individual implementations of CWM.

Unless otherwise stated for a particular OCL constraint, the evaluation policy for all CWM constraints is “deferred” meaning that constraint checking should occur at the end of bulk operations, such as a load, or as part of a model validation operation.

## *Instance Diagrams*

The specification contains examples of metamodel usage in a graphical presentation similar in appearance to UML collaboration diagrams. These instance diagrams should not, however, be confused with UML collaboration diagrams. Individual instances are represented as rectangular boxes containing the instance’s name (if any) followed by the instance’s type. Lines represent links between instance rectangles and are labeled only when required for clarity. Refer to the specification text for precise definition of the identity and semantics of individual lines. Attribute values are shown, when necessary, in separate boxes linked to their owning instance with text in the form *<attribute name> = <value>*.

## *Modularity*

As much as possible, different modeling areas have been placed in different packages, and dependencies between packages have been kept to a minimum. This has been done so that CWM packages can be deployed in various combinations rather than as one enormous model.

## 6.3 How the CWM Metamodel is Described

The following topics briefly describe the conventions this specification uses to define the metamodel elements and their characteristics. This section is extracted from the MOF specification.

### 6.3.1 Classes

Classes are the fundamental building blocks of CWM metamodels. 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.

The CWM uses the term Class with a meaning that is identical to that of Class in UML. A Class is an abstract specification 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 Class specification does not include any methods to implement meta-object behavior.

Each Class is defined in terms of its name(s), super-Classes, the Classes whose instances it can contain, its attributes, its references, its operations, its constraints and whether it is abstract or concrete. This specification uses a hybrid textual and tabular notation to define the important characteristics of each Class. 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 defining Classes and their characteristics.

#### *Class Heading*

Each Class is introduced by a section heading. The heading defines the standard ModelElement name for the Class. The Class's name on the heading line can be followed by the word "*abstract*" or by a "*substitute_name*" for some mapping.

#### *Superclasses*

This heading lists the Classes which generalizes the Class being described. Generalization is another term for inheritance. Multiple inheritance is permitted in CWM.

#### *Contained Elements*

If presented, the heading lists the Classes whose instances may be contained by an instance of this container Class. Instances of Classes may act as containers of other elements by means of composite aggregation associations. Only Classes that are in the current metamodel package or in other packages upon which it is dependent are listed in this section. Omission of a Class from this list does not necessarily preclude instances of that Class from being contained by this container Class.

## *Attributes*

This heading lists the Attributes for a Class. Attributes that are inherited from the super-Classes are not listed. If the "Attributes" heading is absent, the Class has no Attributes.

The following text explains the notation used for defining variable characteristics of Attributes.

| | |
|---|---|
| *type:* | This entry defines the base type for the Attribute. |
| *multiplicity:* | This entry defines the "multiplicity" for the Attribute, consisting of its "lower" and "upper" bounds, and "isOrdered" flag, and an "isUnique" flag.<br><br>The multiplicity for an Attribute is expressed as follows:<br>(1) The "lower" and "upper" bounds are expressed as "exactly one", "zero or one", "zero or more" and "one or more".<br>(2) If the word "ordered" appears, "isOrdered" should be true. If it is absent, "isOrdered" should be false.<br>(3) If the word "unique" appears, "isUnique" should be true. If it is absent, "isUnique" should be false. |
| *changeable:* | This optional entry defines the "isChangeable" flag for the Attribute. If omitted, "isChangeable" is true. |
| *derived from:* | This optional entry describes the derivation of a derived Attribute. If the entry is present, the Attribute's "isDerived" flag will be true. If it is absent, the flag will be false. |
| *scope*: | This optional entry defines the "scope" of an Attribute as either "instance_level" or "class_level". If the entry is absent, the Attribute's "scope" is "instance_level". |

## *References*

This heading lists the References for a Class. References that are inherited from the super-Classes are not listed If the "References" heading is absent, the Class has no References.

A Reference connects its containing Class to an AssociationEnd 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.

The following text explains the notation used for defining variable characteristics of References.

| | |
|---|---|
| *class:* | This entry defines the base type of the Reference. Note the "type" of a Reference must be the same as the "type" of the referenced AssociationEnd. |
| *defined by:* | This entry defines the Association and AssociationEnd that the Reference is linked to. |
| *multiplicity:* | This entry defines the "multiplicity" for the Reference. These are defined in the same way as Attribute "multiplicity" characteristics, except that "unique" is omitted. Note the "multiplicity" settings for an AssociationEnd and its corresponding Reference(s) must be the same. |
| *changeable:* | This optional entry defines the "isChangeable" flag for the Reference. If omitted, "isChangeable" is true. |
| *inverse:* | This optional entry defines the "inverse" Reference for this Reference. If this entry is absent, the Reference does not have an inverse Reference. |

## *Operations*

This heading lists the Operations for a Class. Operations that are inherited from the super-Classes are not listed. If the "Operations" heading is absent, the Class has no Operations.

The following text explains the notation used for defining variable characteristics of Operations.

| | |
|---|---|
| *return type:* | This optional entry defines the "type" and "multiplicity" of the Operations's return Parameter. If this entry is absent, the Operation does not have a return Parameter. |
| *isQuery:* | This optional entry defines the Operation's "isQuery" flag. If it is absent, "isQuery' has the value false. |
| *scope:* | This optional entry defines the Operation's "scope". If it is absent, the Operation has a "scope" of "instance_level". |
| *parameters:* | This entry defines the Operation's non-return Parameter list in the order that they appear in the Operation's signature. The "name", "direction", "type" and "multiplicity" are defined for each Parameter. If the entry simple says "none", the Operation has no non-return Parameters. |
| *exceptions:* | This optional entry defines the list of Exceptions that this Operation may raise in the order that they appear in the Operation's signature. If it is absent, the Operation raises no Exception. |
| *operation semantics:* | This optional entry simple gives a cross reference to the OCL defining the Operation's semantics. |

#### *Constraints*

This heading lists the Constraints that are attached to this Class. If the "Constraints" heading is absent, the Class has no Constraints.

### *6.3.2 Associations*

Associations describe relationships between instances of Classes. The properties of an Association rests mostly in its two AssociationEnds.

The following text explains the notation used for defining Associations and their characteristics.

#### *Association Heading*

Each Association is introduced by a section heading. The heading defines the standard ModelElement name for the Association. The Association's name on the heading line can be followed by the word "*derived*", and "protected" or "private".

### *Ends*

This heading defines the two AssociationEnds for an Association. They are defined by giving their names and defining the remaining characteristics in tabular form.

The following text explains the notation used for defining variable characteristics of AssociationEnds.

| | |
|---|---|
| *class:* | This entry specifies the Class whose instances are linked at this end of the Association. |
| *multiplicity:* | This entry defines the "multiplicity" for the AssociationEnd. These are defined in the same way as Attribute "multiplicity" characteristics, except that "unique" is omitted. Note the "multiplicity" settings for an AssociationEnd and its corresponding Reference(s) must be the same. |
| *aggregation:* | This optional entry defines the AssociationEnd's "aggregation" attribute as one of "composite", "shared" or "none". If the entry is absent, the AssociationEnd's "aggregation" attribute takes the value "none". |

### *Derivation*

This heading defines how a derived Association should be computed. If the "Derivation" heading is absent, the Association is not derived.

# ObjectModel 7

## 7.1 Overview

The CWM ObjectModel provides basic constructs for creating and describing metamodel classes in all other CWM packages. The ObjectModel is a subset of UML that includes only those features that are needed for creating and describing the CWM. Defining a subset of UML containing only those things needed by CWM allows the CWM to leverage UML's concepts and modeling power without burdening implementations with the full breadth of UML's capabilities.

The specification defined in this chapter, where applicable, is based on and taken from the UML specification.

## 7.2 Organization of the ObjectModel Package

The CWM uses packages to control complexity and create groupings of logically interrelated classes. The ObjectModel is a collection of packages that are described together because they all provide basic metamodel constructs to other CWM packages. A subsection of this chapter is devoted to each of the ObjectModel packages. Because it relies on no other package, the Core package is described first, followed by the Behavioral, Instance, and Relationships packages. Each of the subsequent packages depends only on the Core package; there are no other dependencies between the ObjectModel packages. The relationship between the ObjectModel and each of its constituent packages is shown diagrammatically in Figure 7-2-1.

Organizing the ObjectModel in this fashion allows the individual metamodel packages to be understood and used independently of each other without sacrificing their common purpose. For example, the CWM Record metamodel depends only on the ObjectModel's Core and Instance packages for its definition; other ObjectModel packages are not needed for defining records.

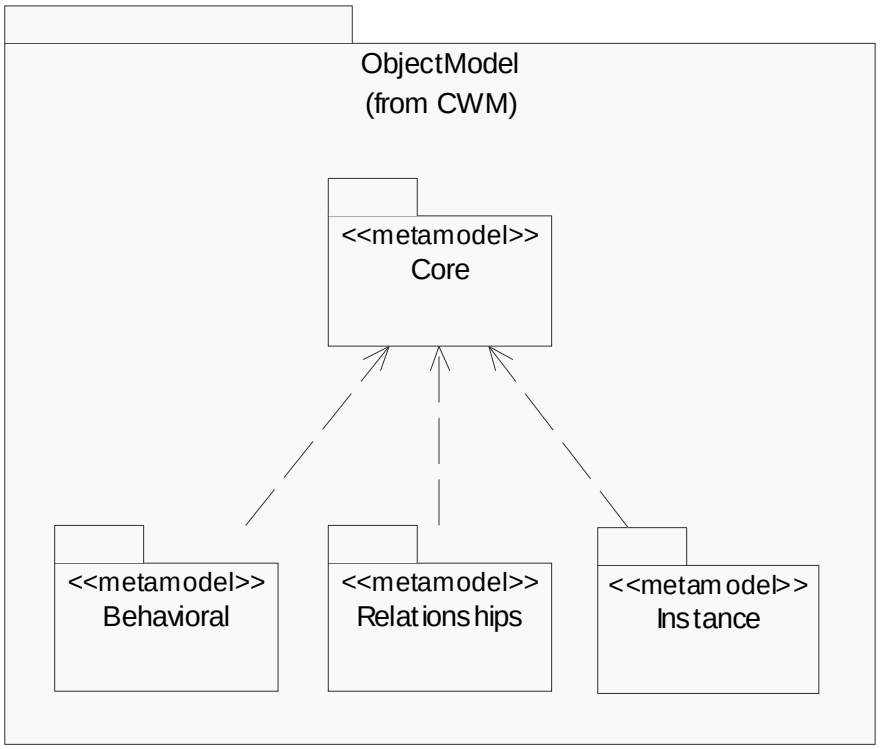


*Figure 7-2-1* ObjectModel metamodel packages

## 7.3 Core Metamodel

The Core metamodel depends on no other packages.

The ObjectModel Core metamodel contains basic metamodel classes and associations used by all other CWM metamodel packages, including other ObjectModel packages. The classes and associations that make up the Core metamodel are shown in Figure 7-3-1. Figure 7-3-2 contains supporting classes within the Core metamodel that are generally used as the types of attributes. Figure 7-4 contains classes and associations that provide generic extension mechanisms to existing metamodels.

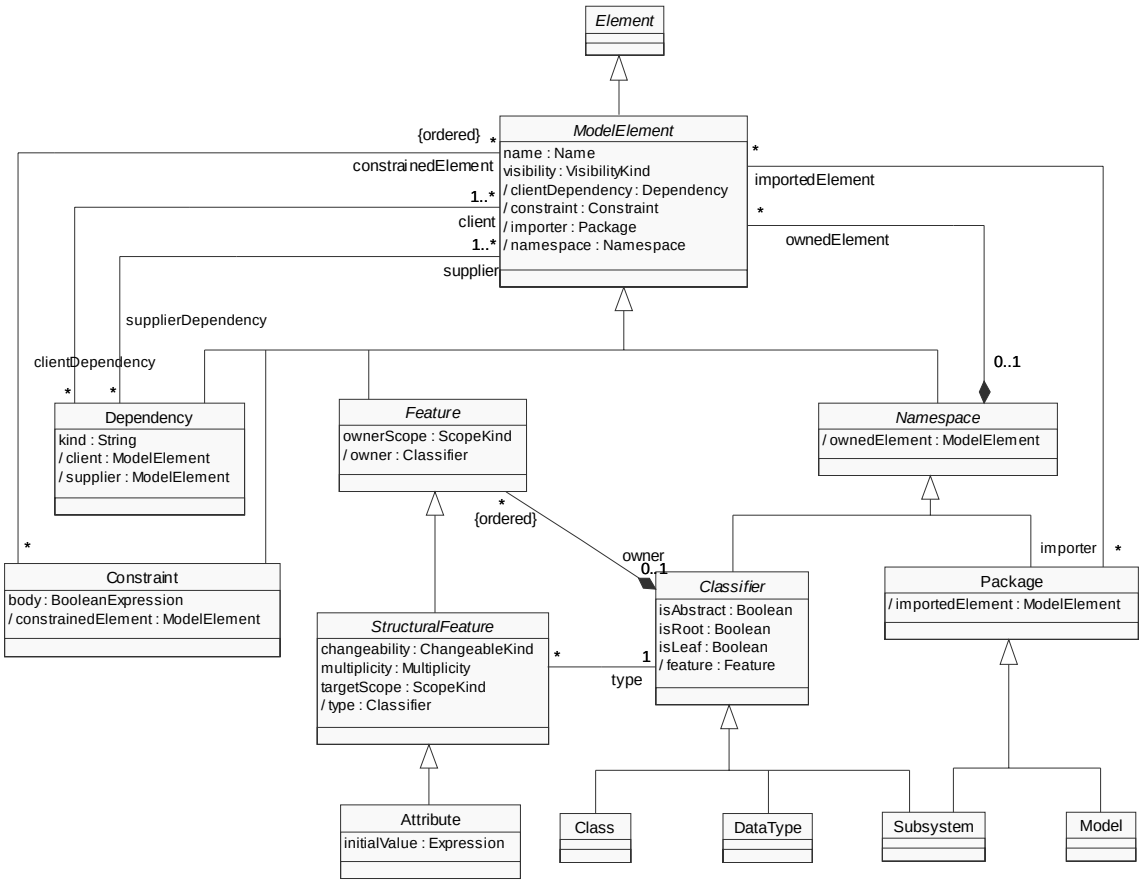


*Figure 7-3-1* Core metamodel.

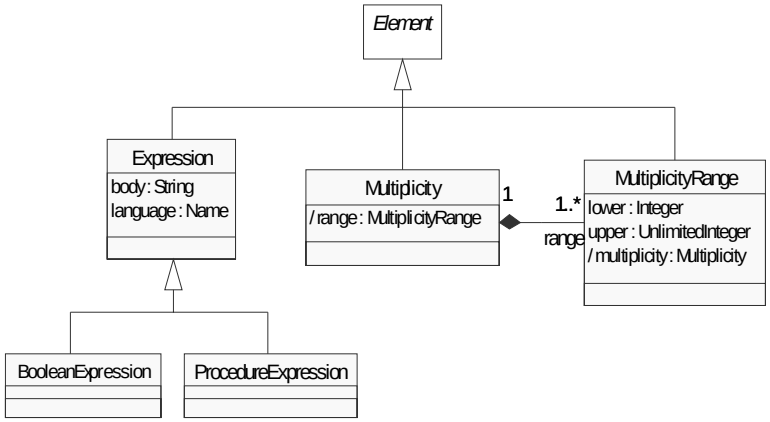


*Figure 7-3-2* Core metamodel supporting classes.

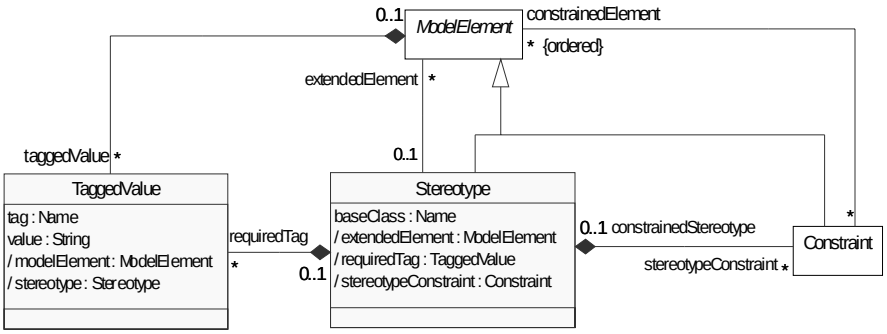


*Figure 7-3-3* ObjectModel extension mechanisms.

## *7.3.1 Core Data Types*

The ObjectModel metamodel contains the data types, shown below in alphabetical order. Each of these data types is an instance of the DataType class.

Some of these data types have default values. These default values only apply for **mandatory** attributes or parameters of the relevant data type where an explicit value is not supplied.

- Any

  The Any data type is used to indicate that an attribute or parameter may take values from any of the available data types. In CWM, the set of data types an Any attribute or parameter may assume includes the data types and enumerations described in this chapter plus any available instances of the Classifier class.

  There is no default value for data type Any.

- Boolean

Boolean defines an enumeration that denotes a logical condition. Its enumeration literals are:

*true* The Boolean condition is satisfied.

*false* The Boolean condition is not satisfied.

The default for data type Boolean is *false.*

- Float

  The Float data type is used to indicate that an attribute or parameter may take on floating point numeric values. The number of significant digits and other representational details are implementation defined.

  The default for the Float data type is the value 0.0.

- Integer

  Integer represents the predefined type of integers. An instance of Integer is an element in the (infinite) set of integers (…-2, -1, 0, 1, 2…).

  The default for Integer is 0.

- Name

  Name defines a token which is used for naming ModelElements and similar usages. Each Name has a corresponding String representation. For purposes of exchange a name should be represented as a String.

  The default for the Name data type is an empty string.

- String

  String defines a piece of text. Strings do not normally have a defined length; rather, they are considered to be arbitrarily long (practical limits on the length of Strings exist, but are implementation dependent). When String is used as the type of an Attribute, string length sometimes can be specified (see the Relational and Record packages for examples).

  The default for the String data type is an empty string.

- Time

  Time defines a statement which will define the time of occurrence of an event. The specific format of time expressions is not specified here and is subject to implementation considerations.

  There is no default for the Time data type.

- UnlimitedInteger

  UnlimitedInteger defines a data type whose range is the nonnegative integers augmented by the special value “unlimited”. It is used for the upper bound of multiplicities.

  The default for an UnlimitedInteger is the special value "unlimited".

The ObjectModel metamodel contains the enumerated data types shown below in alphabetical order. Enumeration literals defined for each enumerated type are described as well.

- ChangeableKind

In the metamodel ChangeableKind defines an enumeration that denotes how an attribute link may be modified. Its values are:

*ck_changeable* No restrictions on modification.

*ck_frozen* The value may not be changed from the source end after the creation and initialization of the source object. Operations on the other end may change a value.

*ck_addOnly* If the multiplicity is not fixed, values may be added at any time from the source object, but once created a value may not be removed from the source end. Operations on the other end may change a value.

The default value is *ck_changeable.*

- OrderingKind

  In the metamodel OrderingKind defines an enumeration that specifies how the elements of a set are arranged. Used in conjunction with elements that have a multiplicity in cases when the multiplicity value is greater than one. The ordering must be determined and maintained by operations that modify the set. Its values are:

  *ok_unordered* The elements of the set have no inherent ordering.

  *ok_ordered* The elements of the set have a sequential ordering.

  The default value is *ok_unordered.*

- ScopeKind

  In the metamodel ScopeKind defines an enumeration that denotes whether a feature belongs to individual instances or an entire classifier. Its values are:

  *sk_instance* The feature pertains to instances of a Classifier. For example, it is a distinct attribute in each instance or an operation that works on an instance.

  *sk_classifier* The feature pertains to an entire Classifier. For example, it is an attribute shared by the entire Classifier or an operation that works on the Classifier, such as a creation operation.

  The default value is *sk_instance*.

- VisibilityKind

  In the metamodel VisibilityKind defines an enumeration that denotes how the element to which it refers is seen outside the enclosing name space. Its values are:

  *vk_public* Other elements may see and use the target element.

  *vk_protected* Descendants of the source element may see and use the target element.

  *vk_private* Only the source element may see and use the target element.

  *vk_package* Elements declared in the same package as the target element may see and use the target element.

  *vk_notapplicable* May be used where namespaces do not support the concept of visibility.

  The default value is *vk_public.*

## *7.3.2 Core Classes*

### 7.3.2.1 Attribute

An Attribute describes a named slot within a Classifier that may hold a value.

#### Superclasses

StructuralFeature

#### Attributes

***initialValue***

An Expression specifying the value of the attribute upon initialization. It is meant to be evaluated at the time the object is initialized. (Note that an explicit constructor may supersede an initial value.)

| | |
|---|---|
| *type:* | Expression |
| *multiplicity:* | zero or one |

### 7.3.2.2 BooleanExpression

In the metamodel BooleanExpression defines a statement which will evaluate to an instance of Boolean when it is evaluated.

#### Superclasses

Expression

### 7.3.2.3 Class

A class is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment. In the metamodel, a Class describes a set of objects sharing a collection of Features that are common to the set of objects.

The purpose of a Class is to declare a collection of Features that fully describe the structure and behavior of objects. Some Classes may not be directly instantiated. These Classes are said to be abstract and exist only for other Classes to inherit and reuse the Features declared by them. No object may be a direct instance of an abstract Class, although an object may be an indirect instance of one through a subclass that is non-abstract.

A Class acts as the namespace for various kinds of contained elements defined within its scope, including classes, interfaces and associations (note that this is purely a

scoping construction and does not imply anything about aggregation). The contained classes can be used as ordinary classes in the container class. If a class inherits another class, the contents of the ancestor are available to its descendents if the visibility of an element is public or protected; however, if the visibility is private, then the element is not visible and therefore not available in the descendant.

### *Superclasses*

Classifier

## *7.3.2.4 Classifier* *Abstract*

A classifier is an element that describes structural and behavioral features; it comes in several specific forms, including class, data type, interface, component, and others that are defined in other metamodel packages.

Classifier is often used as a type.

In the metamodel, a Classifier may declare a collection of Features, such as Attributes, Operations and Methods. It has a name, which is unique in the Namespace enclosing the Classifier. Classifier is an abstract metaclass.

Classifier is a child of Namespace. As a Namespace, a Classifier may declare other Classifiers nested in its scope. Nested Classifiers may be accessed by other Classifiers only if the nested Classifiers have adequate visibility. There are no data value or state consequences of nested Classifiers, i.e., it is not an aggregation or composition.

### *Superclasses*

Namespace

### *Contained Elements*

Feature

#### ***isAbstract***

An abstract Classifier is not instantiable.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

### *References*

***feature***

An ordered list of Features owned by the Classifier.

| | |
|---|---|
| *class:* | Feature |
| *defined by:* | ClassifierFeature::feature |
| *multiplicity:* | zero or more; ordered |
| *inverse:* | Feature::owner |

### 7.3.2.5 Constraint

A constraint is a semantic condition or restriction expressed in text.

In the metamodel, a Constraint is a BooleanExpression on an associated ModelElement(s) which must be true for the model to be well formed. This restriction can be stated in natural language, or in different kinds of languages with well-defined semantics. Certain Constraints are predefined, others may be user defined. Note that a Constraint is an assertion, not an executable mechanism.

The specification is written as an expression in a designated constraint language. The language can be specially designed for writing constraints (such as OCL), a programming language, mathematical notation, or natural language. If constraints are to be enforced by a model editor tool, then the tool must understand the syntax and semantics of the constraint language. Because the choice of language is arbitrary, constraints can be used as an extension mechanism.

The constraint concept allows new semantics to be specified linguistically for a model element. In the metamodel a Constraint directly attached to a ModelElement describes semantic restrictions that this ModelElement must obey.

#### *Superclasses*

ModelElement

#### *Attributes*

***body***

A BooleanExpression that must be true when evaluated for an instance of a system to be well-formed. A boolean expression defining the constraint. Expressions are written as strings in a designated language. For the model to be well formed, the expression must always yield a true value when evaluated for instances of the constrained elements at any time when the system is stable (i.e., not during the execution of an atomic operation).

| | |
|---|---|
| *type:* | BooleanExpression |
| *multiplicity:* | exactly one |

### *References*

***constrainedElement***

A ModelElement or list of ModelElements affected by the Constraint.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | ElementConstraint::constrainedElement |
| *multiplicity:* | zero or more |
| *inverse:* | ModelElement::constraint |

### *Constraints*

A Constraint cannot be applied to itself. [C-3-1]

## *7.3.2.6 DataType*

A data type is a type whose values have no identity (i.e., they are pure values). Data types include primitive built-in types (such as integer and string) as well as definable enumeration types.

In the metamodel, a DataType defines a special kind of Classifier in which operations are all pure functions (i.e., they can return data values but they cannot change data values, because they have no identity). For example, an “add” operation on a number with another number as an argument yields a third number as a result; the target and argument are unchanged.

A DataType is a special kind of Classifier whose instances are primitive values, not objects. For example, integers and strings are usually treated as primitive values. A primitive value does not have an identity, so two occurrences of the same value cannot be differentiated. Usually, DataTypes are used for specification of the type of an attribute or parameter.

### *Superclasses*

Classifier

### *Constraints*

A DataType cannot contain any other ModelElements. [C-3-2]

## *7.3.2.7 Dependency*

A dependency states that the implementation or functioning of one or more elements requires the presence of one or more other elements.

In the metamodel, a Dependency is a directed relationship from a client (or clients) to a supplier (or suppliers) stating that the client is dependent on the supplier (i.e., the client element requires the presence and knowledge of the supplier element).

A dependency specifies that the semantics of a set of model elements requires the presence of another set of model elements. This implies that if the source is somehow modified, the dependents probably must be modified. The reason for the dependency can be specified in several different ways (e.g., using natural language or an algorithm) but is often implicit.

Whenever the supplier element of a dependency changes, the client element is potentially invalidated. After such invalidation, a check should be performed followed by possible changes to the derived client element. Such a check should be performed after which action can be taken to change the derived element to validate it again.

## *Superclasses*

ModelElement

## *Attributes*

### ***kind***

Contains a description of the nature of the dependency relationship between the client and supplier. The list of possible values is open-ended. However, CWM predefines the values "Abstraction" and "Usage".

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | zero or one |

## *References*

### ***client***

The element that is affected by the supplier element. In some cases the direction is unimportant and serves only to distinguish the two elements.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | DependencyClient::client |
| *multiplicity:* | one or more |
| *inverse:* | ModelElement::clientDependency |

***supplier***

Inverse of client. Designates the element that is unaffected by a change. In a two-way relationship this would be the more general element. In an undirected situation the choice of client and supplier may be irrelevant.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | DependencySupplier::supplier |
| *multiplicity:* | one or more |
| *inverse:* | ModelElement::supplierDependency |

### *7.3.2.8 Element* *Abstract*

An element is an atomic constituent of a model. In the metamodel, an Element is the top metaclass in the metaclass hierarchy. Element is an abstract metaclass.

### *7.3.2.9 Expression*

In the metamodel an Expression defines a statement which will evaluate to a (possibly empty) set of instances when executed in a context. An Expression does not modify the environment in which it is evaluated. An expression contains an expression string and the name of an interpretation language with which to evaluate the string.

#### *Superclasses*

Element

#### *Attributes*

***body***

The text of the expression expressed in the given language.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

***language***

Names the language in which the expression body is represented.
The interpretation of the expression depends on the language. If the language name is omitted, no interpretation for the expression can be assumed.
In general, a language name should be spelled and capitalized exactly as it appears in the document defining the language. For example, use COBOL, not Cobol; use Ada, not ADA; use PostScript, not Postscript.

| | |
|---|---|
| *type:* | Name |
| *multiplicity:* | zero or one |

### *7.3.2.10 Feature* *Abstract*

A feature is a property, like attribute or operation, which is encapsulated within a Classifier.

In the metamodel, a Feature declares a structural or behavioral characteristic of an instance of a Classifier or of the Classifier itself. Feature is an abstract metaclass.

#### *Superclasses*

ModelElement

#### *Attributes*

***ownerScope***

Specifies whether the Feature appears in every instance of the Classifier or whether it appears only once for the entire Classifier.

| | |
|---|---|
| *type:* | ScopeKind |
| *multiplicity:* | zero or one |

#### *References*

***owner***

The Classifier declaring the Feature.

| | |
|---|---|
| *class:* | Classifier |
| *defined by:* | ClassifierFeature::owner |
| *multiplicity:* | zero or more |
| *inverse:* | Classifier::feature |

### 7.3.2.11 Model

A model captures a view of a physical system. It is an abstraction of the physical system, with a certain purpose. The model completely describes those aspects of the physical system that are relevant to the purpose of the model, at the appropriate level of detail.

In the metamodel, Model is a subclass of Package. It contains a containment hierarchy of ModelElements that together describe the physical system. A Model also contains a set of ModelElements that represents the environment of the system.

Different Models can be defined for the same physical system, where each model represents a view of the physical system defined by its purpose and abstraction level, e.g. an analysis model, a design model, an implementation model. Typically different models are complementary and defined from the perspectives (viewpoints) of different system stakeholders.

#### Superclasses

Package

### 7.3.2.12 ModelElement *Abstract*

A model element is an element that is an abstraction drawn from the system being modeled.

In the metamodel, a ModelElement is a named entity in a Model. It is the base for all modeling metaclasses in the CWM. All other modeling metaclasses are either direct or indirect subclasses of ModelElement.

#### Superclasses

Element

#### Contained Elements

TaggedValue

#### Attributes

***name***

An identifier for the ModelElement within its containing Namespace.

| | |
|---|---|
| *type:* | Name |
| *multiplicity:* | exactly one |

#### *visibility*

Specifies extent of the visibility of the ModelElement within its owning Namespace.

| | |
|---|---|
| *type:* | VisibilityKind |
| *multiplicity:* | exactly one |

### *References*

#### *clientDependency*

Inverse of client. Designates a set of Dependency in which the ModelElement is a client.

| | |
|---|---|
| *class:* | Dependency |
| *defined by:* | DependencyClient::clientDependency |
| *multiplicity:* | zero or more |
| *inverse:* | Dependency::client |

#### *constraint*

A set of Constraints affecting the element. A constraint that must be satisfied by the model element. A model element may have a set of constraints. The constraint is to be evaluated when the system is stable (i.e., not in the middle of an atomic operation).

| | |
|---|---|
| *class:* | Constraint |
| *defined by:* | ElementConstraint |
| *multiplicity:* | zero or more |
| *inverse:* | Constraint::constrainedElement |

#### *importer*

References the set of Package instances that import the ModelElement.

| | |
|---|---|
| *class:* | Package |
| *defined by:* | ImportedElements::importer |
| *multiplicity:* | zero or more |
| *inverse:* | Package::importedElement |

***namespace***

Designates the Namespace that contains the ModelElement. Every ModelElement except a root element must belong to exactly one Namespace or else be a composite part of another ModelElement (which is a kind of virtual namespace). The pathname of Namespace or ModelElement names starting from the root package provides a unique designation for every ModelElement. The association attribute visibility specifies the visibility of the element outside its namespace (see ElementOwnership).

| | |
|---|---|
| *class:* | Namespace |
| *defined by:* | ElementOwnership::namespace |
| *multiplicity:* | zero or one |
| *inverse:* | Namespace::ownedElement |

### *Constraints*

Tags associated with a model element (directly via a property list or indirectly via a stereotype) must not clash with any meta attributes associated with the model element. [C-3-3]

A model element must have at most one tagged value with a given tag name. [C-3-4]

A stereotype cannot extend itself. [C-3-5]

## *7.3.2.13 Multiplicity*

In the metamodel a Multiplicity defines a non-empty set of non-negative integers. A set which only contains zero ({0}) is not considered a valid Multiplicity. Every Multiplicity has at least one corresponding String representation.

### *Superclasses*

Element

### *Contained Elements*

MultiplicityRange

### *References*

***range***

References the set of MultiplicityRange instances that describe the cardinality of the Multiplicity instance.

| | |
|---|---|
| *class:* | MultiplicityRange |
| *defined by:* | RangeMultiplicity |
| *multiplicity:* | one or more |
| *inverse:* | MultiplicityRange::multiplicity |

### 7.3.2.14 MultiplicityRange

In the metamodel a MultiplicityRange defines a range of integers. The upper bound of the range cannot be below the lower bound. The lower bound must be a nonnegative integer. The upper bound must be a nonnegative integer or the special value *unlimite*d, which indicates there is no upper bound on the range.

#### Superclasses

Element

#### Attributes

***lower***

Specifies the positive integer lower bound of the range.

| | |
|---|---|
| *type:* | Integer |
| *multiplicity:* | exactly one |

***upper***

Specifies the upper bound of the range, which is a positive integer or the special value 'unlimited' indicating no upper bound is defined.

| | |
|---|---|
| *type:* | UnlimitedInteger |
| *multiplicity:* | exactly one |

#### References

***multiplicity***

References the Multiplicity instance that owns the MultiplicityRange.

| | |
|---|---|
| *class:* | Multiplicity |
| *defined by:* | RangeMultiplicity::multiplicity |
| *multiplicity:* | exactly one |
| *inverse:* | Multiplicity::range |

### *7.3.2.15 Namespace* *Abstract*

A namespace is a part of a model that contains a set of ModelElements each of whose names designates a unique element within the namespace.

In the metamodel, a Namespace is a ModelElement that can own other ModelElements, such as Classifiers. The name of each owned ModelElement must be unique within the Namespace. Moreover, each contained ModelElement is owned by at most one Namespace. The concrete subclasses of Namespace may have additional constraints on which kind of elements may be contained.

Namespace is an abstract metaclass.

Note that explicit parts of a model element, such as the features of a Classifier, are not modeled as owned elements in a namespace. A namespace is used for unstructured contents such as the contents of a package, or a class declared inside the scope of another class.

#### *Superclasses*

ModelElement

#### *Contained Elements*

ModelElement

#### *References*

***ownedElement***

A set of ModelElements owned by the Namespace. The ModelElement's visibility attribute states whether the element is visible outside the namespace.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | ElementOwnership::ownedElement |
| *multiplicity:* | zero or more |
| *inverse:* | ModelElement::namespace |

### 7.3.2.16 Package

A package is a grouping of model elements.

In the metamodel, Package is a subclass of Namespace. A Package contains ModelElements such as Packages and Classifiers. A Package may also contain Constraints and Dependencies between ModelElements of the Package.

The purpose of the *package* construct is to provide a general grouping mechanism. In fact, its only semantics is to define a namespace for its contents. The package construct can be used for organizing elements for any purpose; the criteria to use for grouping elements together into one package are not defined.

A package owns a set of model elements, with the implication that if the package is removed from the model, so are the elements owned by the package. Elements with names, such as classifiers, that are owned by the same package must have unique names within the package, although elements in different packages may have the same name.

There may be relationships between elements contained in the same package, and between an element in one package and an element in a surrounding package at any level. In other words, elements “see” all the way out through nested levels of packages. Elements in peer packages, however, are encapsulated and are not a priori visible to each other. The same goes for elements in contained packages, i.e. packages do not see “inwards”.

Elements owned by a Package can be made available to other Packages by importing them. Although any ModelElement may be imported by a Package, imported ModelElements are typically other Packages. When an element is imported by a package it extends the namespace of that package. Thus the elements available in a Package consists of its owned and imported ModelElements.

#### Superclasses

Namespace

#### References

***importedElement***

The namespace defined by the package is extended by model elements imported from other packages.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | ImportedElements::importedElement |
| *multiplicity:* | zero or more |
| *inverse:* | ModelElement::importer |

### 7.3.2.17 ProcedureExpression

In the metamodel ProcedureExpression defines a statement which will result in a change to the values of its environment when it is evaluated.

#### Superclasses

Expression

### 7.3.2.18 Stereotype

The stereotype concept provides a way of branding (classifying) model elements so that they behave as if they were instances of new virtual metamodel constructs. These model elements have the same structure (attributes, associations, operations) as similar non-stereotyped model elements of the same kind. The stereotype may specify additional constraints and required tagged values that apply to model elements. In addition, a stereotype may be used to indicate a difference in meaning or usage between two model elements with identical structure.

In the metamodel the Stereotype metaclass is a subclass of ModelElement. Tagged Values and Constraints attached to a Stereotype apply to all ModelElements branded by that Stereotype.

A stereotype keeps track of the base class to which it may be applied. The base class is a class in the metamodel (not a user-level modeling element) such as Class, Association, etc. If a model element is branded by an attached stereotype, then the CWM base class of the model element must be the base class specified by the stereotype or one of the subclasses of that base class.

#### Superclasses

ModelElement

#### Contained Elements

Constraint

TaggedValue

#### Attributes

#### *baseClass*

Specifies the name of a modeling element to which the stereotype applies, such as Class, Association, Constraint, etc. This is the name of a metaclass, that is, a class from the metamodel itself rather than a user model class.

| | |
|---|---|
| *type:* | Name |
| *multiplicity:* | exactly one |

### *References*

#### *extendedElement*

Designates the model elements affected by the stereotype. Each one must be a model element of the kind specified by the baseClass attribute.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | StereotypedElement::extendedElement |
| *multiplicity:* | zero or more |
| *inverse:* | ModelElement::stereotype |

#### *requiredTag*

Specifies a set of TaggedValues, each of which specifies a tag that an element classified by the Stereotype is required to have. The value part indicates the default value for the tagged value, that is, the tagged value that an element will be presumed to have if it is not overridden by an explicit tagged value on the element bearing the stereotype. If the value is unspecified, then the element must explicitly specify a tagged value with the given tag.

| | |
|---|---|
| *class:* | TaggedValue |
| *defined by:* | StereotypeTaggedValues::requiredTag |
| *multiplicity:* | zero or more |
| *inverse:* | TaggedValue::stereotype |

#### *stereotypeConstraint*

Designates constraints that apply to all model elements branded by this stereotype. These constraints are defined in the scope of the full metamodel.

| | |
|---|---|
| *class:* | Constraint |
| *defined by:* | StereotypeConstraints::stereotypeConstraint |
| *multiplicity:* | zero or more |
| *inverse:* | Constraint::constrainedStereotype |

### Constraints

The base class name must be provided. [C-3-6]

## 7.3.2.19 StructuralFeature *Abstract*

A structural feature refers to a static feature of a model element.

In the metamodel, a StructuralFeature declares a structural aspect of a Classifier that is typed, such as an attribute. For example, it specifies the multiplicity and changeability of the StructuralFeature. StructuralFeature is an abstract metaclass.

### Superclasses

Feature

### Attributes

#### *changeability*

Specifies whether the value may be modified after the object is created.

| | |
|---|---|
| *type:* | ChangeabilityKind |
| *multiplicity:* | exactly one |

#### *multiplicity*

The possible number of data values for the feature that may be held by an instance. The cardinality of the set of values is an implicit part of the feature. In the common case in which the multiplicity is 1..1, then the feature is a scalar (i.e., it holds exactly one value).

| | |
|---|---|
| *type:* | Multiplicity |
| *multiplicity:* | zero or one |

#### *ordering*

Specifies whether the set of instances is ordered. The ordering must be determined and maintained by Operations that add values to the feature. This property is only relevant if the multiplicity is greater than one.

| | |
|---|---|
| *type:* | OrderingKind |
| *multiplicity:* | zero or one |

***targetScope***

Specifies whether the targets are ordinary Instances or are Classifiers.

| | |
|---|---|
| *type:* | ScopeKind |
| *multiplicity:* | zero or one |

### *References*

***type***

Designates the Classifier whose instances are values of the feature. It must be a Class, DataType or Interface.

| | |
|---|---|
| *class:* | Classifier |
| *defined by:* | StructuralFeatureType::type |
| *multiplicity:* | exactly one |

## *7.3.2.20 Subsystem*

A subsystem is a grouping of model elements that represents a behavioral unit in a physical system. A subsystem offers interfaces and has operations.

In the metamodel, Subsystem is a subclass of both Package and Classifier. As such it may have a set of Features.

The purpose of the *subsystem* construct is to provide a grouping mechanism for specifying a behavioral unit of a physical system. Apart from defining a namespace for its contents, a subsystem serves as a specification unit for the behavior of its contained model elements.

The contents of a subsystem is defined in the same way as for a package, thus it consists of owned elements and imported elements, with unique names within the subsystem.

### *Superclasses*

Classifier

Package

## *7.3.2.21 TaggedValue*

A tagged value allows information to be attached to any model element in the form of a "tagged value" pair (i.e., name = value). The interpretation of tagged value semantics is intentionally beyond the scope of CWM. It must be determined by user or tool conventions. It is expected that tools will define tags to supply information needed for their operations beyond the basic semantics of CWM. Such information could

include code generation options, model management information, or user-specified semantics.

Even though TaggedValues are a simple and straightforward extension technique, their use restricts semantic interchange of metadata to only those tools that share a common understanding of the specific tagged value names.

## Superclasses

Element

## Attributes

### *tag*

Contains the name of the TaggedValue. This name determines the semantics that are applicable to the contents of the value attribute.

| | |
|---|---|
| *type:* | Name |
| *multiplicity:* | exactly one |

### *value*

Contains the current value of the TaggedValue.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

## References

### *modelElement*

References the ModelElement to which the TaggedValue pertains.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | TaggedElement::modelElement |
| *multiplicity:* | zero or one |
| *inverse:* | ModelElement::taggedValue |

### *stereotype*

References a Stereotype that uses the TaggedValue.

| | |
|---|---|
| *class:* | Stereotype |

| | |
|---|---|
| *defined by:* | StereotypeTaggedValues |
| *multiplicity:* | zero or one |
| *inverse:* | Stereotype::requiredTag |

### 7.3.3 Core Associations

#### 7.3.3.1 ClassifierFeature *Protected*

The ClassifierFeature association provides a composite aggregation containment relationship between Classifiers and the Features they own. The feature end of the association is ordered allowing preservation of the ordering of Features within their owning Classifier. A Feature can be owned by at most one Classifier. The optional character of ownership is intended as a convenience to tools, allowing them to create Features prior to linking them to their owning Classifier.

*Ends*

***owner***

Identifies the Classifier instance that owns the Feature.

| | |
|---|---|
| *class:* | Classifier |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

***feature***

Identifies the Features owned by a Classifier instance and provides their ordering.

| | |
|---|---|
| *class:* | Feature |
| *multiplicity:* | zero or more; ordered |

#### 7.3.3.2 DependencyClient *Protected*

The DependencyClient association links Dependency instances with ModelElements that act as clients in the represented dependency relationship.

*Ends*

***client***

Identifies the ModelElements that are clients of the Dependency instance.

| | |
|---|---|
| *class:* | ModelElement |
| *multiplicity:* | one or more |

***clientDependency***

Identifies Dependency instances in which the ModelElement acts as a client.

| | |
|---|---|
| *class:* | Dependency |
| *multiplicity:* | zero or more |

### *7.3.3.3 DependencySupplier*

The DependencySupplier association links Dependency instances with ModelElements that act as suppliers in the represented dependency relationship.

#### *Ends*

***supplier***

Identifies the ModelElements that are suppliers of the Dependency instance.

| | |
|---|---|
| *class:* | ModelElement |
| *multiplicity:* | one or more |

***supplierDependency***

The DependencySupplier association links Dependency instances with ModelElements that act as suppliers in the represented dependency relationship.

| | |
|---|---|
| *class:* | Dependency |
| *multiplicity:* | zero or more |

### *7.3.3.4 ElementConstraint* *Protected*

The ElementConstraint association provides linkages between ModelElements and the Constraint instances that constrain their state. Note that a Constraint instance may not simultaneously participate in both the ElementConstraint and the StereotypeConstraint associations.

#### *Ends*

***constrainedElement***

Identifies the ModelElements whose state is constrained by the Constraint instance.

| | |
|---|---|
| *class:* | ModelElement |
| *multiplicity:* | zero or more; ordered |

***constraint***

Identifies the Constraint instances that restrict the possible states that a ModelElement may take.

| | |
|---|---|
| *class:* | Constraint |
| *multiplicity:* | zero or more |

### *7.3.3.5 ElementOwnership* *Protected*

The ElementOwnership association identifies ModelElements owned by Namespaces. ModelElements may be owned by at most one Namespace. Refer to the above discussion of the Namespace class for more information on the nature of the ownership relationship between Namespaces and ModelElements.

#### *Ends*

***ownedElement***

Identifies the ModelElements owned by a Namespace.

| | |
|---|---|
| *class:* | ModelElement |
| *multiplicity:* | zero or more |

***namespace***

Identifies the Namespace, if any, that owns the ModelElement.

| | |
|---|---|
| *class:* | Namespace |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

### *7.3.3.6 ImportedElements* *Protected*

The ImportedElements association identifies ModelElements that a Package instance imports from other Namespaces. Although any ModelElement may be imported by a Package, imported ModelElements are typically other Packages or aggregate Classifiers, such as Class instances.

*Ends*

***importedElement***

Identifies ModelElements imported by a Package.

| | |
|---|---|
| *class:* | ModelElement |
| *multiplicity:* | zero or more |

***importer***

Identifies the Packages that import a ModelElement.

| | |
|---|---|
| *class:* | Package |
| *multiplicity:* | zero or more |

### *7.3.3.7 RangeMultiplicity* *Protected*

The RangeMultiplicity association identifies the set of MultiplicityRange instances that specify the lower and upper bounds of individual cardinality ranges defined by a Multiplicity instance. A MultiplicityRange instance must be owned by a single Multiplicity instance.

*Ends*

***multiplicity***

Identifies the Multiplicity instance that owns the MultiplicityRange.

| | |
|---|---|
| *class:* | Multiplicity |
| *multiplicity:* | exactly one |
| *aggregation:* | composite |

***range***

Identifies the set of MultiplicityRange instances owned by a Multiplicity.

| | |
|---|---|
| *class:* | MultiplicityRange |
| *multiplicity:* | one or more |

### *7.3.3.8 StereotypeConstraints*

The StereotypeConstraints association links Stereotypes with Constraints that further restrict the states that a stereotyped ModelElement may assume. A Constraint instance may not simultaneously participate in both the StereotypeContraints association and the ElementConstraint association.

### Ends

***stereotypeConstraint***

Identifies the set of Constraint instances defined for the Stereotype instance.

| | |
|---|---|
| *class:* | Constraint |
| *multiplicity:* | zero or more |

***constrainedStereotype***

Identifies the Stereotype owning a Constraint instance.

| | |
|---|---|
| *class:* | Stereotype |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

## 7.3.3.9 StereotypedElement

The StereotypedElement association links Stereotypes with the ModelElements to which they apply.

### Ends

***extendedElement***

Identifies the set of ModelElements to which the Stereotype instance applies.

| | |
|---|---|
| *class:* | ModelElement |
| *multiplicity:* | zero or more |

***stereotype***

Identifies the Stereotype instance that further defines the semantics of the ModelElement.

| | |
|---|---|
| *class:* | Stereotype |
| *multiplicity:* | zero or one |

## 7.3.3.10 StereotypeTaggedValues *Protected*

The StereotypeTaggedValues association links Stereotypes with the set of TaggedValues they require.

TaggedValues cannot simultaneously participate in both the TaggedElement and StereotypeTaggedValues associations.

*Ends*

***requiredTag***

Specifies a set of TaggedValues, each of which specifies a tag that an element classified by the Stereotype is required to have.

| | |
|---|---|
| *class:* | TaggedValue |
| *multiplicity:* | zero or more |

***stereotype***

Identifies a Stereotype instance that owns the TaggedValue instance.

| | |
|---|---|
| *class:* | Stereotype |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

### *7.3.3.11 StructuralFeatureType*

The StructuralFeatureType association identifies the Classifier instance that defines the type of particular StructuralFeatures. A StructuralFeature instance must have a Classifier instance that defines its type.

*Ends*

***structuralFeature***

Identifies the set of StructuralFeatures for which the Classifier defines the type.

| | |
|---|---|
| *class:* | StructuralFeature |
| *multiplicity:* | zero or more |

***type***

Identifies the Classifier defining the type of a StructuralFeature.

| | |
|---|---|
| *class:* | Classifier |
| *multiplicity:* | exactly one |

### *7.3.3.12 TaggedElement*

The TaggedElement association links TaggedValues with the ModelElements that own them.

TaggedValues cannot simultaneously participate in both the TaggedElement and StereotypeTaggedValues associations.

#### *Ends*

***modelElement***

Identifies the ModelElement instance that owns the TaggedValue instance.

| | |
|---|---|
| *class:* | ModelElement |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

***taggedValue***

Identifies the set of TaggedValue instances that extend a ModelElement.

| | |
|---|---|
| *class:* | TaggedValue |
| *multiplicity:* | zero or more |

### *7.3.4 OCL Representation of Core Constraints*

#### *Operations*

The operation **allFeatures** results in a Set containing all Features of the Classifier itself and all its inherited Features.

```
allFeatures : Set(Feature);
allFeatures = self.feature->union(self.parent.oclAsType(Classifier).allFeatures)
```

The operation **allAttributes** results in a Set containing all Attributes of the Classifier itself and all its inherited Attributes.

```
allAttributes : set(Attribute);
allAttributes = self.allFeatures->select(f | f.oclIsKindOf(Attribute))
```

The operation **specification** yields the set of Classifiers that the current Classifier realizes.

```
specification: Set(Classifier)
specification = self.clientDependency->
select(d | d.stereotype.name = "realization"
  and d.supplier.oclIsKindOf(Classifier)).supplier.oclAsType(Classifier)
```

The operation **parent** returns a Set containing all direct parents of a Classifier.

```
parent : Set(Classifier);
parent = self.generalization.parent
```

The operation **allParents** returns a Set containing all the Classifiers inherited by this Classifier (the transitive closure), excluding the Classifier itself.

```
allParents : Set(Classifier);
allParents = self.parent->union(self.parent.allParents)
```

The operation **allContents** returns a Set containing all ModelElements contained in the Classifier together with the contents inherited from its parents.

```
allContents : Set(ModelElement);
allContents = self.contents->union(self.parent.allContents->
  select(e | e.elementOwnership.visibility = #public or
        e.elementOwnership.visibility = #protected))
```

The operation **supplier** results in a Set containing all direct suppliers of the ModelElement.

```
supplier : Set(ModelElement);
supplier = self.clientDependency.supplier
```

The operation **allSuppliers** results in a Set containing all the ModelElements that are suppliers of this ModelElement, including the suppliers of these Model Elements. This is the transitive closure.

```
allSuppliers : Set(ModelElement);
allSuppliers = self.supplier->union(self.supplier.allSuppliers)
```

The operation **contents** results in a Set containing all ModelElements contained by the Namespace.

```
contents : Set(ModelElement)
contents = self.ownedElement -> union(self.namespace.contents)
```

The operation **allContents** results in a Set containing all ModelElements contained by the Namespace.

```
allContents : Set(ModelElement);
allContents = self.contents
```

The operation **allVisibleElements** results in a Set containing all ModelElements visible outside of the Namespace.

```
allVisibleElements : Set(ModelElement)
allVisibleElements = self.allContents -> select(e |
  e.elementOwnership.visibility = #public)
```

The operation **allSurroundingNamespaces** results in a Set containing all surrounding Namespaces.

```
allSurroundingNamespaces : Set(Namespace)
allSurroundingNamespaces =
self.namespace->union(self.namespace.allSurroundingNamespaces)
```

The operation **contents** results in a Set containing the ModelElements owned by or imported by the Package.

```
contents : Set(ModelElement)
contents = self.ownedElement->union(self.importedElement)
```

The operation **allImportedElements** results in a Set containing the ModelElements imported by the Package.

```
allImportedElements : Set(ModelElement)
allImportedElements = self.importedElement
```

The operation **allContents** results in a Set containing the ModelElements owned by or imported by the Package.

```
allContents : Set(ModelElement)
allContents = self.contents
```

## *Constraints*

[C-3-1] A Constraint cannot be applied to itself.

**context** Constraint **inv:**

**not** self.constrainedElement->includes (self)

[C-3-2] A DataType cannot contain any other ModelElements.

**context** DataType **inv:**

self.ownedElement->isEmpty

[C-3-3] Tags associated with a model element (directly via a property list or indirectly via a stereotype) must not clash with any meta attributes associated with the model element.

**context** ModelElement **inv:**

-- cannot be specified with OCL

[C-3-4] A model element must have at most one tagged value with a given tag name.

**context** ModelElement **inv:**

self.taggedValue->forAll(t1, t2 : TaggedValue |
t1.tag = t2.tag **implies** t1 = t2)

[C-3-5] A stereotype cannot extend itself.

**context** ModelElement **inv:**

self.stereotype->excludes(self)

[C-3-6] The base class name must be provided.

**context** Stereotype **inv:**

Set {self.baseClass}->notEmpty

## 7.4 Behavioral Metamodel

The Behavioral metamodel depends on the following package:

- org.omg::CWM::ObjectModel::Core

The Behavioral metamodel collects together classes and associations that describe the behavior of CWM types and provides a foundation for recording the invocations of defined behaviors. The elements of the Behavioral metamodel are shown in the following figure.

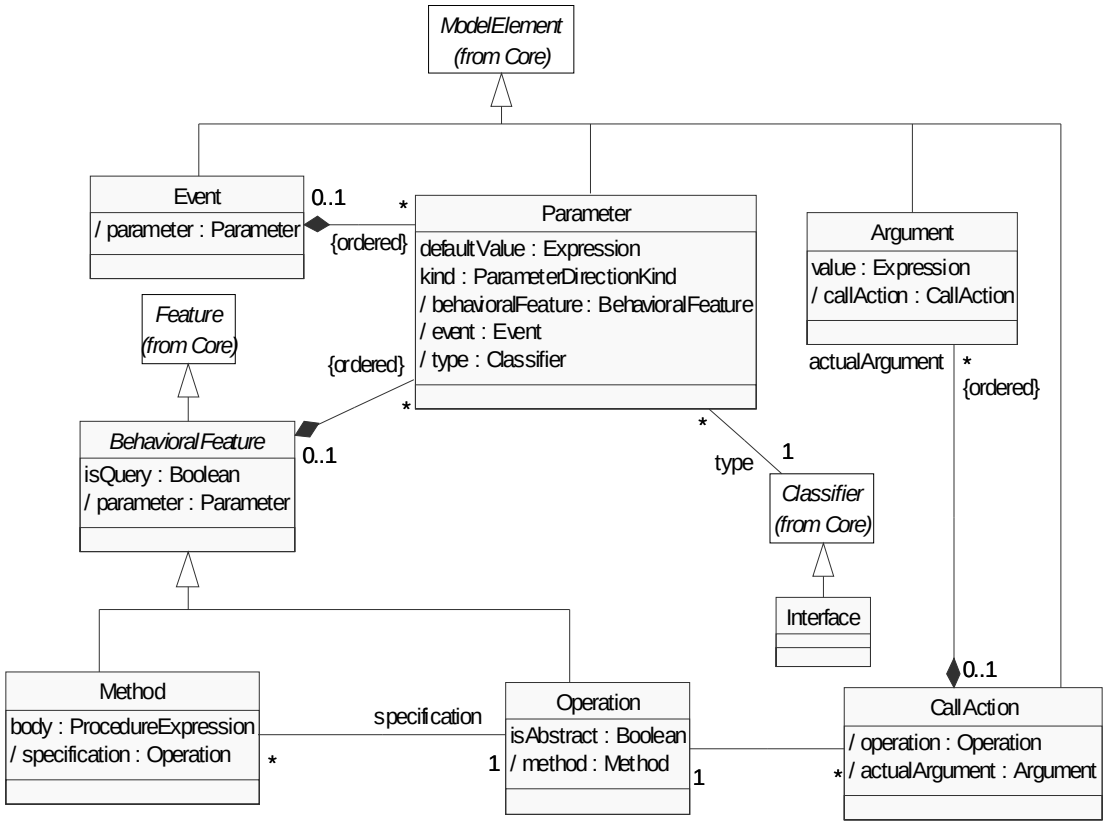


*Figure 7-4-1* Behavioral metamodel.

### 7.4.1 Behavioral Data Types

The Behavioral package provides the following enumerated type:

- ParameterDirectionKind

  In the metamodel ParameterDirectionKind defines an enumeration that denotes if a Parameter is used for supplying an argument and/or for returning a value. The enumeration values are:

*pdk_in*: An input Parameter (may not be modified).

*pdk_out*: An output Parameter (may be modified to communicate information to the caller).

*pdk_inout*: An input Parameter that may be modified.

*pdk_return*: A return value of a call.

The default value is *pdk_in.*

## 7.4.2 Behavioral Classes

### 7.4.2.1 Argument

Argument is an expression describing how to determine an actual value passed in a CallAction.

In the metamodel an Argument is a composite part of a CallAction and contains a meta-attribute, value, of type Expression. It states how the actual argument is determined when the owning CallAction is executed.

#### Superclasses

ModelElement

#### Attributes

***value***

An expression determining the actual Argument instance when executed.

| | |
|---|---|
| *type:* | Expression |
| *multiplicity:* | exactly one |

#### References

***callAction***

Identifies the CallAction which uses the Argument.

| | |
|---|---|
| *class:* | CallAction |
| *defined by:* | CallArguments::action |
| *multiplicity:* | zero or one |
| *inverse:* | CallAction::actualArgument |

### 7.4.2.2 BehavioralFeature *Abstract*

A behavioral feature refers to a dynamic feature of a model element, such as an operation or method. In the metamodel, BehavioralFeature specifies a behavioral aspect of a Classifier. All different kinds of behavioral aspects of a Classifier, such as Operation and Method, are subclasses of BehavioralFeature.

BehavioralFeature is an abstract metaclass.

#### *Superclasses*

Feature

#### *Contained Elements*

Parameter

#### *Attributes*

***isQuery***

Specifies whether an execution of the BehavioralFeature leaves the state of the system unchanged. True indicates that the state is unchanged; false indicates that side-effects may occur.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

#### *References*

***parameter***

An ordered list of Parameters for the BehavioralFeature. To call the BehavioralFeature, the caller must supply a list of values compatible with the types of the Parameters.

| | |
|---|---|
| *class:* | Parameter |
| *defined by:* | BehavioralFeatureParameter::parameter |
| *multiplicity:* | zero or more; ordered |
| *inverse:* | Parameter::behavioralFeature |

#### *Constraints*

All Parameters should have a unique name. [C-4-1]

The type of the Parameters should be included in the Namespace of the Classifier. [C-4-2]

### 7.4.2.3 CallAction

A call action is an action resulting in an invocation of an operation.

The purpose of a CallAction is to identify the actual Arguments used in a specific invocation of an Operation.

#### Superclasses

ModelElement

#### References

***operation***

The Operation which will be invoked when the CallAction is executed.

| | |
|---|---|
| *class:* | Operation |
| *defined by:* | CalledOperation::operation |
| *multiplicity:* | exactly one |

***actualArgument***

The Argument(s) supplied to the CallAction.

| | |
|---|---|
| *class:* | Argument |
| *defined by:* | CallArguments::actualArgument |
| *multiplicity:* | zero or more; ordered |
| *inverse:* | Argument::callAction |

#### Constraints

The number of arguments must be the same as the number of the Operation. [C-4-3]

### 7.4.2.4 Event

Event is a specification of an observable occurrence. The occurrence that generates an event instance is assumed to take place at an instant in time.

#### Superclasses

ModelElement

### *Contained Elements*

Parameter

### *References*

***parameter***

References the set of ordered Parameter instances that comprise the signature of the Event.

| | |
|---|---|
| *class:* | Parameter |
| *defined by:* | EventParameter::parameter |
| *multiplicity:* | zero or more; ordered |
| *inverse:* | Parameter::event |

## *7.4.2.5 Interface*

Interface is a named set of operations that specify the behavior of an element.

In the metamodel, an Interface contains a set of Operations that together define a service offered by a Classifier realizing the Interface. A Classifier may offer several services, which means that it may realize several Interfaces, and several Classifiers may realize the same Interface.

### *Superclasses*

Classifier

### *Constraints*

An Interface can only contain Operations. [C-4-4]

An Interface cannot contain any ModelElements. [C-4-5]

All Features defined in an Interface are public. [C-4-6]

## *7.4.2.6 Method*

Method is the implementation of an Operation. It specifies the algorithm or procedure that effects the results of an Operation.

### *Superclasses*

BehavioralFeature

### Attributes

#### *body*

A specification of the Method in some appropriate form (such as a programming language). The exact form of a Method's specification and knowledge of the language in which it is described is outside the scope of the CWM.

| | |
|---|---|
| *type:* | ProcedureExpression |
| *multiplicity:* | exactly one |

### References

#### *specification*

References the Operation that the Method implements.

| | |
|---|---|
| *class:* | Operation |
| *defined by:* | OperationMethod::specification |
| *multiplicity:* | exactly one |
| *inverse:* | Operation::method |

### Constraints

If the realized Operation is a query, then so is the Method. [C-4-7]

The signature of the Method should be the same as the signature of the realized Operation. [C-4-8]

The visibility of the Method should be the same as for the realized Operation. [C-4-9]

The realized Operation must be a feature (possibly inherited) of the same Classifier as the Method. [C-4-10]

If the realized Operation has been overridden one or more times in the ancestors of the owner of the Method, then the Method must realize the latest overriding (that is, all other Operations with the same signature must be owned by ancestors of the owner of the realized Operation). [C-4-11]

There may be at most one Method for a given Classifier (as owner of the Method) and Operation (as specification of the Method) pair. [C-4-12]

## 7.4.2.7 Operation

Operation is a service that can be requested from an object to effect behavior. An Operation has a signature, which describes the parameters that are possible (including possible return values).

In the metamodel, an Operation is a BehavioralFeature that can be applied to instances of the Classifier that contains the Operation.

Operation is the specification, while Method is the implementation.

### *Superclasses*

BehavioralFeature

### *Attributes*

***isAbstract***

If true, then the Operation does not have an implementation, and one must be supplied by a descendant. If false, the Operation must have an implementation in the class or inherited from an ancestor.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

### *References*

***method***

References the set of Method instances defined for the Operation.

| | |
|---|---|
| *class:* | Method |
| *defined by:* | OperationMethod::method |
| *multiplicity:* | zero or more |
| *inverse:* | Method::specification |

## *7.4.2.8 Parameter*

Parameters are used in the specification of operations, methods and events. A Parameter may include a name, type, and direction of communication.

### *Superclasses*

ModelElement

### *Attributes*

***defaultValue***

An Expression whose evaluation yields a value to be used when no argument is supplied for the Parameter.

| | |
|---|---|
| *type:* | Expression |
| *multiplicity:* | zero or one |

***kind***

Specifies what kind of a Parameter is required.

| | |
|---|---|
| *type:* | ParameterDirectionKind |
| *multiplicity:* | exactly one |

## *References*

***behavioralFeature***

References the BehavioralFeature instance for which the Parameter instance describes a parameter.

| | |
|---|---|
| *class:* | BehavioralFeature |
| *defined by:* | BehavioralFeatureParameter::behavioralFeature |
| *multiplicity:* | zero or one |
| *inverse:* | BehavioralFeature::parameter |

***event***

References the Event instance for which the Parameter instance describes a parameter.

| | |
|---|---|
| *class:* | Event |
| *defined by:* | EventParameter::event |
| *multiplicity:* | zero or one |
| *inverse:* | Event::parameter |

***type***

Designates a Classifier to which an argument value must conform.

| | |
|---|---|
| *class:* | Classifier |
| *defined by:* | ParameterType::type |
| *multiplicity:* | exactly one |

### 7.4.3 Behaviorial Associations

#### 7.4.3.1 BehavioralFeatureParameter *Protected*

The BehavioralFeatureParameter association identifies and orders Parameter instances describing the parameters of a BehavioralFeature. Parameters may be owned by at most one BehavioralFeature instance. The set of parameters of a BehavioralFeature, together with its name and return value, are said to constitute the BehavioralFeature's "signature".

*Ends*

***behavioralFeature***

Identifies the BehavioralFeature instance owner of a Parameter instance.

| | |
|---|---|
| *class:* | BehavioralFeature |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

***parameter***

Identifies the Parameter instances that describe the parameters of the BehavioralFeature.

| | |
|---|---|
| *class:* | Parameter |
| *multiplicity:* | zero or more; ordered |

#### 7.4.3.2 CallArguments *Protected*

Identifies the Argument instances representing the actual argument values passed to an Operation during the particular invocation indicated by the CallAction instance. The ordering of actual argument values is assumed to correspond to the ordering of the Operation's parameters as represented by the ordering of the BehavioralFeatureParameter association.

*Ends*

***actualArgument***

Identifies the Argument instances representing the actual arguments passed during Operation invocation.

| | |
|---|---|
| *class:* | Argument |
| *multiplicity:* | zero or more; ordered |

***callAction***

Identifies the CallAction instance representing a particular invocation of an Operation.

| | |
|---|---|
| *class:* | CallAction |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

### *7.4.3.3 CalledOperation*

The CalledOperation association identifies the CallAction instance representing a particular invocation of an Operation.

***Ends***

***callAction***

Identifies the CallAction instance representing a particular invocation of an Operation.

| | |
|---|---|
| *class:* | CallAction |
| *multiplicity:* | zero or more |

***operation***

Identifies the Operation instance for which the CallAction instance records an invocation.

| | |
|---|---|
| *class:* | Operation |
| *multiplicity:* | exactly one |

### *7.4.3.4 EventParameter* *Protected*

The EventParameter association identifies the set of Parameter instances owned by an Event instance.

#### *Ends*

***event***

Identifies the Event owning a set of Parameter instances.

| | |
|---|---|
| *class:* | Event |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

***parameter***

Identifies the ordered set of Parameter instances owned by an Event that describe the Event's parameters.

| | |
|---|---|
| *class:* | Parameter |
| *multiplicity:* | zero or more; ordered |

### *7.4.3.5 OperationMethod* *Protected*

The OperationMethod association links an Operation with the Method instance(s) that realize it. The various Method instances represent alternative implementations (usually in different programming languages or environments) of the Operation.

*Ends*

***specification***

Identifies the Operation that a Method implements.

| | |
|---|---|
| *class:* | Operation |
| *multiplicity:* | exactly one |

***method***

Identifies the set of Methods defined for an Operation.

| | |
|---|---|
| *class:* | Method |
| *multiplicity:* | zero or more |

### *7.4.3.6 ParameterType*

The ParameterType association links a Parameter instance with the Classifier that defines the parameter's type.

*Ends*

***parameter***

Identifies the set of Parameter instances for which a particular Classifier acts as a type definition.

| | |
|---|---|
| *class:* | Parameter |
| *multiplicity:* | zero or more |

***type***

Identifies the Classifier instance the defines the type of a Parameter.

| | |
|---|---|
| *class:* | Classifier |
| *multiplicity:* | exactly one |

## *7.4.4 OCL Representation of Behavioral Constraints*

### *Operations*

The operation **hasSameSignature** checks if the argument has the same signature as the instance itself.

```
hasSameSignature ( b : BehavioralFeature ) : Boolean;
hasSameSignature (b) =
  (self.name = b.name) and
  (self.parameter->size = b.parameter->size) and
  Sequence{ 1..(self.parameter->size) }->forAll( index : Integer |
    b.parameter->at(index).type =
      self.parameter->at(index).type and
    b.parameter->at(index).kind =
      self.parameter->at(index).kind
  )
```

The operation **allOperations** results in a Set containing all Operations of the Classifier itself and all its inherited Operations.

```
allOperations : Set(Operation);
allOperations = self.allFeatures->select(f | f.ockIsKindOf(Operations))
```

The operation **allMethods** results in a Set containing all Methods of the Classifier itself and all its inherited Methods.

```
allOperations : Set(Method);
allMethods = self.allFeatures->select(f | f.ockIsKindOf(Method))
```

## Constraints

[C-4-1] All Parameters should have a unique name.

**context** BehavioralFeature **inv:**

self.parameter->forAll(p1, p2 | p1.name = p2.name **implies** p1 = p2)

[C-4-2] The type of the Parameters should be included in the Namespace of the Classifier.

**context** BehavioralFeature **inv:**

self.parameter->forAll( p | self.owner.namespace.allContents->includes (p.type) )

[C-4-3] The number of arguments must be the same as the number of parameters of the Operation.

**context** CallAction **inv:**

self.actualArgument->size = self.operation.parameter->size

[C-4-4] An Interface can only contain Operations.

**context** Interface **inv:**

self.allFeatures->forAll( f | f.oclIsKindOf( Operation ) )

[C-4-5] An Interface cannot contain any ModelElements.

**context** Interface **inv:**

self.allContents->isEmpty

[C-4-6] All Features defined in an Interface are public.

**context** Interface **inv:**

self.allFeatures->forAll( f | f.visibility = #public )

[C-4-7] If the realized Operation is a query, then so is the Method.

**context** Method **inv:**

self.specification->isQuery **implies** self.isQuery

[C-4-8] The signature of the Method should be the same as the signature of the realized Operation.

**context** Method **inv:**

self.hasSameSignature( self.specification )

[C-4-9] The visibility of the Method should be the same as for the realized Operation.

**context** Method **inv:**

self.visibility = self.specification.visibility

[C-4-10] The realized Operation must be a feature (possibly inherited) of the same Classifier as the Method.

**context** Method **inv:**

self.owner.allOperations->includes( self.specification )

[C-4-11] If the realized Operation has been overridden one or more times in the ancestors of the owner of the Method, then the Method must realize the latest overriding (that is, all other operations with the same signature must be owned by ancestors of the owner of the realized Operation).

**context** Method **inv:**

self.specification.owner.allOperations->includesAll(
( self.owner.allOperations->select( op |
self.hasSameSignature( op ) ) )

[C-4-12] There may be at most one method for a given classifier (as owner of the method) and operation (as specification of the method) pair.

**context** Method **inv:**

self.owner.allMethods->select( operation = self.operation )->size = 1

## 7.5 Relationships Metamodel

The Relationships metamodel depends on the following package:

- org.omg::CWM::ObjectModel::Core

The Relationships metamodel collects together classes and associations that describe the relationships between object within a CWM information store. The Relationships metamodel describes to types of relationships: association and generalization.

Association relationships record linkages between model elements. These linkages may represent simple linkages between model elements or aggregation ("is part of") relationships between model elements; aggregation relationships come in two forms -- shared and composite. Associations have two or more named ends that link them to instances of the classes connected by the association.

Generalization relationships record arrangements of model elements into type hierarchies in a parent/child (or "is type of") fashion. Child types are said to "specialize", "subclass" or "subtype" their parental types, represent a subset of parental instances that fulfill the definition of the child type, and inherit the structural features (Attributes, AssociationEnd) and behavioral features (Operations, Methods) of their parents. Parental types are said to "generalize" their child types or to be "superclasses" or "supertypes" of their children.

CWM generalization hierarchies support multiple inheritance; that is, child types may have more than one parental type and inherit the union of the features of all their parental types. Although called "hierarchies", multiple inheritance actually represents a directed acyclic graph of parental and child types.

The classes and associations of the Relationships metamodel are shown in the following figure.

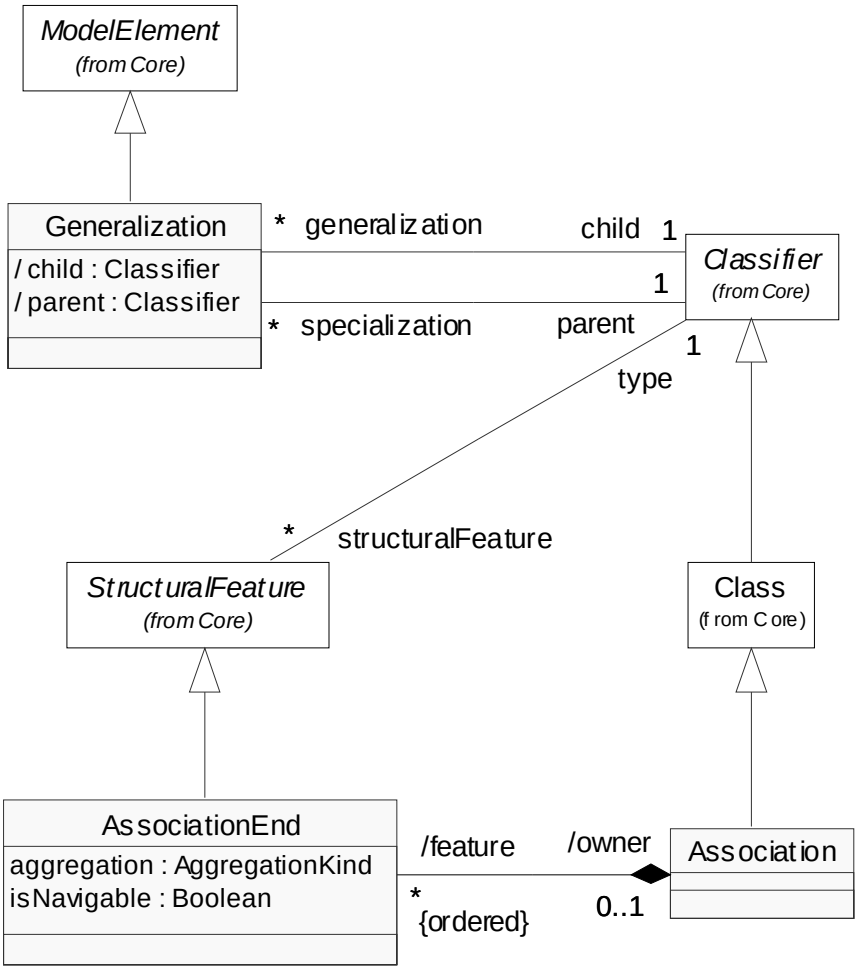


*Figure 7-5-1* Relationship metamodel.

## *7.5.1 Relationships Data Types*

The Relationships metamodel contains the following enumerated type:

- AggregationKind

  An enumeration that denotes what kind of aggregation an Association defines. When placed on a target end, specifies the relationship of the target end to the source end. AggregationKind defines an enumeration whose values are:

  *ak_none* The end is not an aggregate.

  *ak_aggregate* The end is an aggregate; therefore, the other end is a part and must have the aggregation value of none. The part may be contained in other aggregates.

*ak_composite* The end is a composite; therefore, the other end is a part and must have the aggregation value of none. The part is strongly owned by the composite and may not be part of any other composite.

The default value is *ak_none*.

## 7.5.2 Relationships Classes

### 7.5.2.1 Association

An association defines a semantic relationship between classifiers. Associations have two or more named ends. Associations with two or more ends are called "n-ary" whereas associations with exactly two ends are called "binary". Each end, depending upon its multiplicity, connects to zero or more instances of some classifier.

In the metamodel, an Association is a declaration of a semantic relationship between Classifiers, such as Classes. Associations must have two, and may have more, association ends. Each end is connected to a Classifier; the same Classifier may be connected to more than one association end in the same association. (Refer to the ObjectModel's Instance package, below, for a description of how Associations are instantiated.)

Because Associations are classifiers, they own and order their association ends (which are Attributes) via the ClassifierFeature association. In addition, because Associations are Classes, they can also own more traditional StructuralFeatures such as Attributes. Consequently, they may act in a manner similar to "association classes" described by some other object models.

An association may represent an aggregation (i.e., a whole/part relationship). In this case, the association end attached to the whole element is designated, and the other association end represents the parts of the aggregation.

Associations can be of three different kinds: (1) ordinary association, (2) composite aggregate, and (3) shareable aggregate. Since the aggregate construct can have several different meanings depending on the application area, CWM gives a more precise meaning to two of these constructs (i.e., association and composite aggregate) and leaves the shareable aggregate more loosely defined in between. Only binary Associations can have composite or sharable aggregation semantics.

Composite aggregation is a strong form of aggregation which requires that a part instance be included in at most one composite at a time and that the composite object has sole responsibility for the disposition of its parts. This means that the composite object is responsible for the creation and destruction of the parts. In implementation terms, it is responsible for their memory allocation. If a composite object is destroyed, it must destroy all of its parts. It may remove a part and give it to another composite object, which then assumes responsibility for it. If the multiplicity from a part to composite is zero-to-one, the composite may remove the part and the part may assume responsibility for itself, otherwise it may not live apart from a composite.

A consequence of these rules is that a composite aggregation implies propagation semantics (i.e., some of the dynamic semantics of the whole is propagated to its parts).

For example, if the whole is copied or destroyed, then so are the parts as well (because a part may belong to at most one composite).

A classifier on the composite end of an association may have parts that are classifiers and associations. At the instance level, an instance of a part element is considered "part of" the instance of a composite element. If an association is part of a composite and it connects two classes that are also part of the same composite, then an instance of the association will connect objects that are part of the same composite object of which the instance is part.

A shareable aggregation denotes weak ownership (i.e., the part may be included in several aggregates) and its owner may also change over time. However, the semantics of a shareable aggregation does not imply deletion of the parts when an aggregate referencing it is deleted. Both kinds of aggregations define a transitive, antisymmetric relationship (i.e., the instances form a directed, non-cyclic graph). Composition instances form a strict tree (or rather a forest).

#### *Superclasses*

Class

#### *Constraints*

An Association must have at least two AssociationEnds. [C-5-1]

The AssociationEnds must have a unique name within the association. [C-5-2]

At most one AssociationEnd may be an aggregation or composition. [C-5-3]

If an Association has three or more AssociationEnds, then no AssociationEnd may be an aggregation or composition. [C-5-4]

The connected Classifiers of the AssociationEnds should be included in the Namespace of the Association, or be Classifiers with public visibility in other Namespaces to which the Association has access. [C-5-5]

### *7.5.2.2 AssociationEnd*

An association end is an endpoint of an association, which connects the association to a classifier. Each association end is part of one association. The association ends of each association are ordered.

In the metamodel, an AssociationEnd is part of an Association and specifies the connection of an Association to some other Classifier. Because AssociationEnds are a kind of StructuralFeature, they are owned and ordered by Association instances via the ClassifierFeature association. The StructuralFeatureType association is used to identify the Classifier to which the AssociationEnd is attached. Each AssociationEnd has a name and defines a set of properties of the connection.

The multiplicity property of an association end specifies how many instances of the classifier at a given end (the one bearing the multiplicity value) may be associated with

a single instance of the classifier at the other end. The association end also states whether or not the connection may be traversed towards the instance playing that role in the connection (the isNavigable attribute), that is, if the instance is directly reachable via the association.

## *Superclasses*

StructuralFeature

## *Attributes*

***aggregation***

When placed on one end (the “target” end), specifies whether the class on the target end is an aggregation with respect to the class on the other end (the “source”end). Only one end of an association can be an aggregation.

| | |
|---|---|
| *type:* | AggregationKind |
| *multiplicity:* | exactly one |

***isNavigable***

When placed on a target end, specifies whether traversal from a source instance to its associated target instances is possible. A value of true means that the association can be navigated by the source class and the target rolename can be used in navigation expressions. Specification of navigability for each direction is defined independently.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

## *Constraints*

An AssociationEnd must have an owning Association. [C-5-6]

The Classifier of an AssociationEnd cannot be an Interface or a DataType if the association is navigable away from that end. [C-5-7]

An Instance may not belong by composition to more than one composite Instance. [C-5-8]

An AssociationEnd with composite or shared aggregation semantics must be owned by an Association. [C-5-9]

### 7.5.2.3 Generalization

A generalization is a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element (it has all of its properties, members, and relationships) and may contain additional information.

In the metamodel, a Generalization is a directed inheritance relationship, uniting a Classifier with a more general Classifier in a hierarchy. Generalization is a subtyping relationship; that is, an instance of the more general ("parent") Classifier may be substituted by an instance of the more specific ("child") Classifier.

To understand inheritance fully, it is necessary to understand the concept of a full descriptor and a segment descriptor. A full descriptor is the full description needed to describe an instance of a metamodel object. It contains a description of all of the attributes, associations, and operations that the object contains.

In a pre-object-oriented language, the full descriptor of a data structure was declared directly in its entirety. In an object-oriented language, the description of an object is built out of incremental segments that are combined using inheritance to produce a full descriptor for an object. The segments are the modeling elements that are actually declared in a model. Each classifier contains a list of features and other relationships that it adds to what it inherits from its ancestors. The mechanism of inheritance defines how full descriptors are produced from a set of segments connected by generalization. The full descriptors are implicit, but they define the structure of actual instances. Features of a classifier that have private visibility are not visible to descendants of the classifier.

If a classifier has no parent, then its full descriptor is the same as its segment descriptor. If a classifier has one or more parents, then its full descriptor contains the union of the features from its own segment descriptor and the segment descriptors of all of its ancestors. No attribute, operation, or association end with the same signature may be declared in more than one of the segments (in other words, they may not be redefined). A method may be declared in more than one segment. A method declared in any segment supersedes and replaces a method with the same signature declared in any ancestor. If two or more methods nevertheless remain, then they conflict and the model is ill-formed. The constraints on the full descriptor are the union of the constraints on the segment itself and all of its ancestors. If any of them are inconsistent, then the model is ill-formed.

In any full descriptor for a classifier, each method must have a corresponding operation. In a concrete classifier, each operation in its full descriptor must have a corresponding method in the full descriptor.

#### *Superclasses*

ModelElement

#### *References*

***child***

Designates a Classifier that occupies the child or specialization position of the Generalization relationship.

| | |
|---|---|
| *class:* | Classifier |
| *defined by:* | ChildElement::child |
| *multiplicity:* | exactly one |

***parent***

Designates a Classifier that occupies the parent or generalization position of the Generalization relationship.

| | |
|---|---|
| *class:* | Classifier |
| *defined by:* | ParentElement::parent |
| *multiplicity:* | exactly one |

## *7.5.3 Relationships Associations*

### *7.5.3.1 ChildElement*

The ChildElement association links Classifiers with the Generalization instances that describe where they participate as children in the inheritance hierarchy.

#### *Ends*

***child***

Identifies the Classifier instance that acts as a child in the Generalization relationship.

| | |
|---|---|
| *class:* | Classifier |
| *multiplicity:* | exactly one |

***generalization***

Identifies the set of Generalization instances in which the Classifier acts as a child in the inheritance hierarchy.

| | |
|---|---|
| *class:* | Generalization |
| *multiplicity:* | zero or more |

#### 7.5.3.2 ParentElement

The ParentElement association links Classifiers with the Generalization instances that describe where the Classifiers participate as parents in the inheritance hierarchy.

##### Ends

***parent***

Identifies the Classifier instance that acts as a parent in an inheritance hierarchy.

| | |
|---|---|
| *class:* | Classifier |
| *multiplicity:* | exactly one |

***specialization***

Identifies the set of Generalization instances in which the Classifier acts a parent in the inheritance hierarchy.

| | |
|---|---|
| *class:* | Generalization |
| *multiplicity:* | zero or more |

### 7.5.4 OCL Representation of Relationships Constraints

#### 7.5.4.1 Association

##### Operations

The operation **allConnections** results in the set of all AssociationEnds of the Association.

```
allConnections : Set(AssociationEnd);
allConnections = self.feature.oclIsKindOf(AssociationEnd)
```

##### Constraints

[C-5-1] An Association must have at least 2 AssociationEnds.

**context** Association **inv:**

```
self.allConnections->size > 1
```

[C-5-2] The AssociationEnds must have a unique name within the association.

**context** Association **inv:**

```
self.allConnections->forAll( r1, r2 | r1.name = r2.name implies r1 = r2)
```

[C-5-3] At most one AssociationEnd may be an aggregation or composition.

**context** Association **inv:**

```
self.allConnections->select(aggregation <> #ak_none)->size <= 1
```

[C-5-4] If an Association has three or more AssociationEnds, then no AssociationEnd may be an aggregation or composition.

**context** Association **inv:**

```
self.allConnections->size >=3 implies
self.allConnections->forall(aggregation = #ak_none)
```

[C-5-5] The connected Classifiers of the AssociationEnds should be included in the Namespace of the Association, or be Classifiers with public visibility in other Namespaces to which the Association has access.

**context** Association **inv:**

```
self.allConnections->forAll(r | self.namespace.allContents->includes (r.type) ) or
self.allConnections->forAll(r | self.namespace.allContents->excludes (r.type))
  implies
self.namespace.clientDependency->exists (d |
  d.supplier.oclAsType(Namespace).ownedElement->select (e |
    e.elementOwnership.visibility = #ak_public)->includes (r.type) or
  d.supplier.oclAsType(Classifier).allParents.
    oclAsType(Namespace).ownedElement->select (e |
    e.elementOwnership.visibility = #ak_public)->includes (r.type) or
  d.supplier.oclAsType(Package).allImportedElements->select (e |
    e.elementImport.visibility = #ak_public) ->includes (r.type) ) )
```

### *7.5.4.2 AssociationEnd*

#### *Constraints*

[C-5-6] An AssociationEnd must have an owning Association.

**context** AssociationEnd **inv:**

```
self.owner.oclIsKindOf(Association)
```

[C-5-7] The Classifier of an AssociationEnd cannot be an Interface or a DataType if the association is navigable away from that end.

**context** AssociationEnd **inv:**

(self.type.oclIsKindOf (Interface) **or**
self.type.oclIsKindOf (DataType)) **implies**
self.owner->select (ae | ae <self)->forAll(ae | ae.isNavigable = #false)

[C-5-8] An instance may not belong by composition to more than one composite Instance.

**context** AssociationEnd **inv:**

self.aggregation = #ak_composite **implies** self.multiplicity.max <= 1

[C-5-9] An AssociationEnd with composite or shared aggregation semantics must be owned by an Association.

**context** AssociationEnd **inv:**

self.aggregation = #ak_composite **or** self.aggregation = #ak_shared **implies**
self.owner.oclIsKindOf(Association)

## 7.6 Instance Metamodel

The Instance metamodel depends on the following package:

- org.omg::CWM::ObjectModel::Core

In addition to the metadata normally interchanged with CWM, it is sometimes useful to interchange specific data instances as well. The ObjectModel's Instance metamodel allows the inclusion of data instances with the metadata.

The Instance metamodel is shown in Figure 7-6-1.

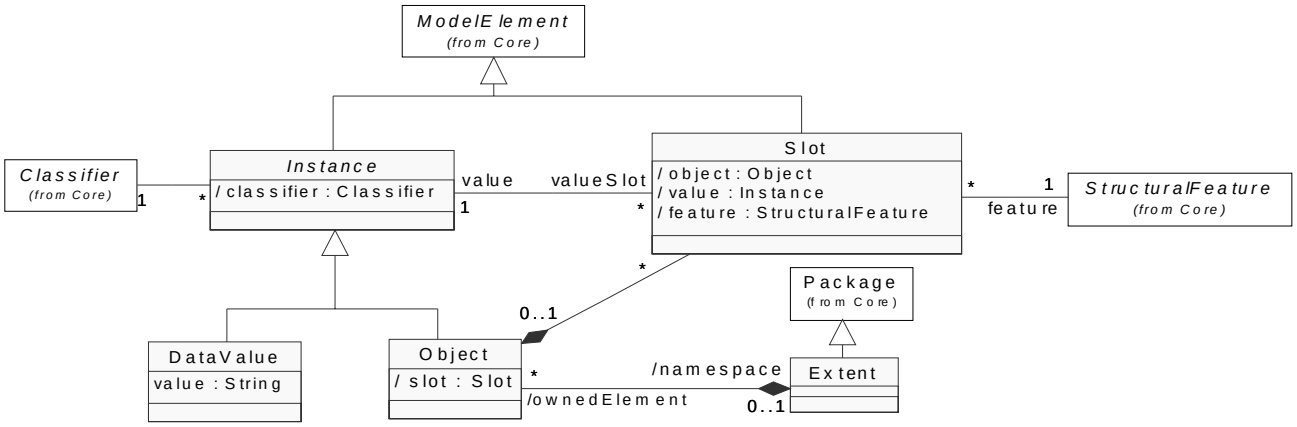


*Figure 7-6-1* Instance metamodel.

To aid understanding of the appropriate use of Instance metamodel classes and associations, a full example is presented in Figure 7-6-3, below, showing how Instance metamodel objects are used to represent the model, shown in Figure 7-6-2, and its instances.

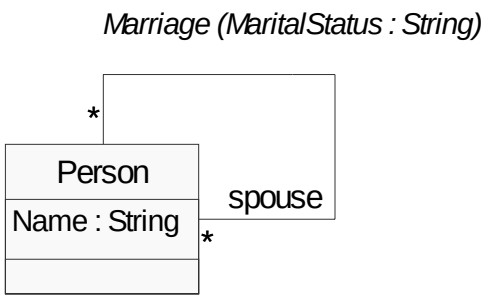


*Figure 7-6-2* Instance metamodel example model

The example model describes people and their marital relationships to other people. Marital relationships are represented by the reflective Marriage association between two separate people. The Marriage association has two association ends named "person" and "spouse". Notice that each instance of the Marriage association has a

string-valued attribute describing the current status of the marital relationship it represents. Valid values for the MaritalStatus attribute are "Married", "Divorced", and "Widowed". People who have never been married have no instances of the Marriage association.

Figure 7-6-3 shows how the example model is represented as instances of the CWM ObjectModel metaclasses Class, Attribute, DataType, Association, and AssociationEnd. In addition, Instance metamodel classes are used to capture two kinds of data values that might be exchange using the CWM DTDs: valid values of the MaritalStatus attribute, and the marital relationship between the people George and Martha Custis Washington. In the figure, instances of the Instance metamodel are shown with a shaded background and labelled with an uppercase letter near their upper right corner to facilitate discussion. Lines in the figure represent ObjectModel associations that capture relationships between instances and are labelled with the asociation's name.

George is represented by Object A, and Martha, by Object B. These person objects own Slots C and D, respectively. Slot C holds DataValue E whose value attribute records George's name. Similarly, Slot D holds Martha's name in DataValue F.

The valid values of the MaritalStatus attribute are recorded by DataValue instances K, L, and M.

The marital relationship between George and Martha is represented, from George's perspective, by Object H which is an instance of the Marriage association. Object H owns Slots G, J, and I. Slot G holds the person association end and references Object A (George), whereas Slot I holds the spouse association end, referencing Object B (Martha). Slot J holds a DataValue instance describing the current value ("Married") of the MaritalStatus attribute for Object H.

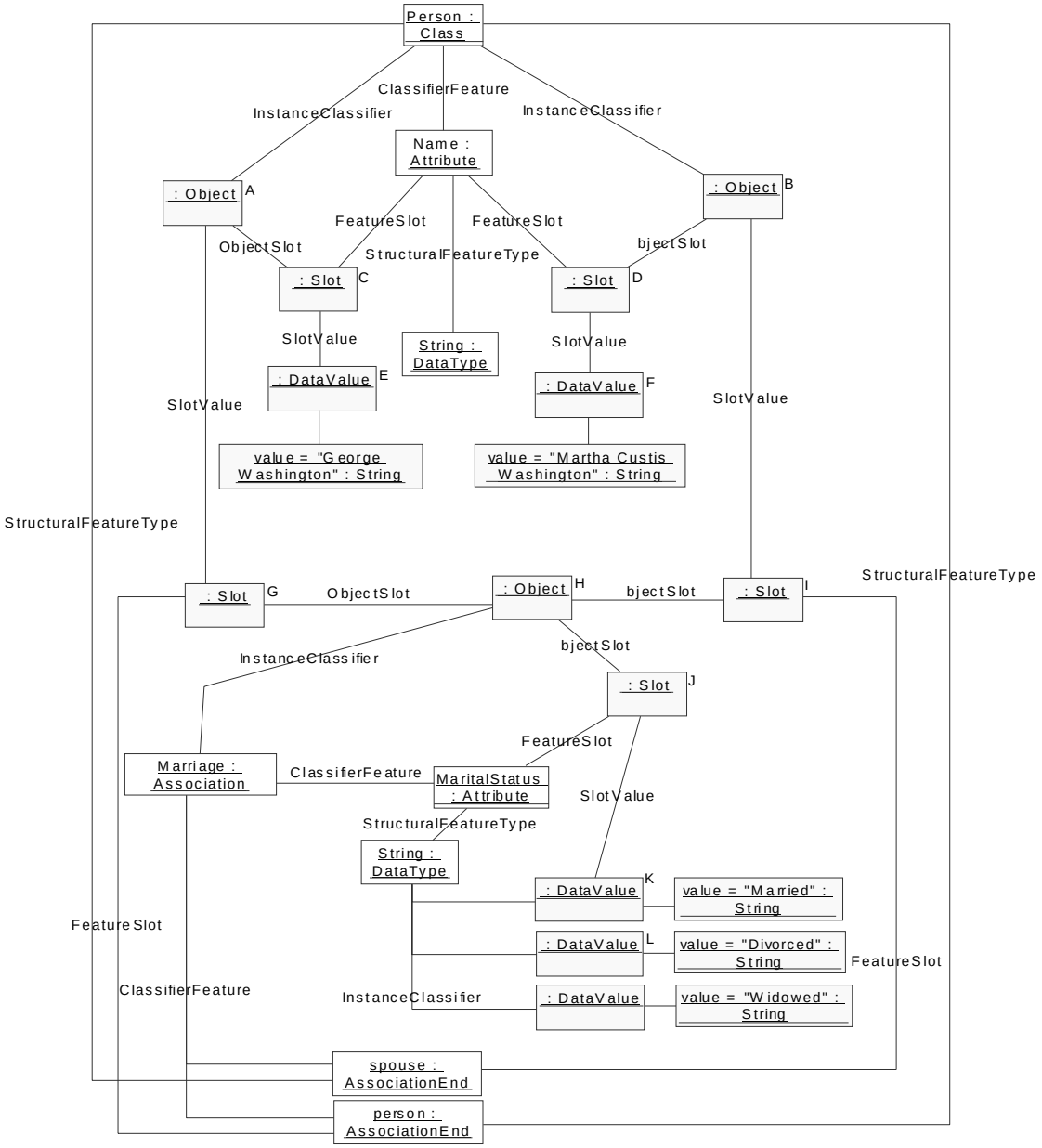


*Figure 7-6-3* Instance metamodel example instances

## *7.6.1 Instance Classes*

### 7.6.1.1 DataValue

A data value is an instance with no identity. In the metamodel, DataValue is a child of Instance that cannot change its state, i.e. all operations that are applicable to it are pure functions or queries that do not cause any side effects. DataValues are typically used as attribute values.

Since it is not possible to differentiate between two data values that appear to be the same, it becomes more of a philosophical issue whether there are several data values representing the same value or just one for each value. In addition, a data value cannot change its data type and it does not have contained instances.

#### Superclasses

Instance

#### Attributes

***value***

A string representation of the value.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

#### Constraints

A DataValue originates from a Classifier that is a DataType. [C-6-1]

A DataValue has no Slots. [C-6-2]

### 7.6.1.2 Extent

Each instance of Extent owns a collection of instances and is used to link such collections to their structural and behavioral definitions in CWM Resource packages. Because Extent is a subclass of package, it owns member instances via the ElementOwnership associaton.

#### Superclasses

Package

#### Contained Elements

Object

### 7.6.1.3 Instance — Abstract

The instance construct defines an entity to which a set of operations can be applied and which has a state that stores the effects of the operations. In the metamodel Instance is connected to a Classifier that declares its structure and behavior. It has a set of attribute values matching the definition of its Classifier. The set of attribute values implements the current state of the Instance.

Because Instance is an abstract class, all Instances are either Object or DataValue instances.

The data content of an Instance comprises one value for each attribute in its full descriptor (and nothing more). The value must be consistent with the type of the attribute. An instance must obey any constraints on the full descriptor of the Classifier of which it is an instance (including both explicit constraints and built-in constraints such as multiplicity).

#### *Superclasses*

ModelElement

#### *References*

***classifier***

The Classifier that declares the structure of the Instance.

| | |
|---|---|
| *class:* | Classifier |
| *defined by:* | InstanceClassifier::classifier |
| *multiplicity:* | exactly one |

### 7.6.1.4 Object

An object is an instance that originates from a class.

In the metamodel, Object is a subclass of Instance originating from a Class. The Class may be modified dynamically, which means that the set of features of the Object may change during its life-time.

An object is an instance that originates from a class; it is structured and behaves according to its class. All objects originating from the same class are structured in the same way, although each of them has its own set of attribute slots. Each attribute slot references an instance, usually a data value or possibly, another object. The number of attribute slots with the same name fulfills the multiplicity of the corresponding attribute in the class. The set may be modified according to the specification in the corresponding attribute, e.g. each referenced instance must originate from (a specialization of) the type of the attribute, and attribute slots may be added or removed according to the changeable property of the attribute.

An Object instance's slots may contain either DataValue instances or other Object instances. Owned Object instances occur as side-effects of either of two metamodel situations: First, the Classifier of the owning instance contains features (via the ClassifierFeature association) whose types are non-DataType Classifiers. Second, the StructuralFeature describing the attribute slot is an AssociationEnd.

An Object instance may own other Object instances. This occurs when the Classifier describing the owning Object contains the Classifier(s) describing the owned object through namespace containment via the ElementOwnership association. Namespace rules imply that an Object instance contained in another Object instance has access to all names that are accessible to its container instance.

### *Superclasses*

Instance

### *Contained Elements*

Slot

### *References*

***slot***

The set of Slot instances owned by the Object.

| | |
|---|---|
| *class:* | Slot |
| *defined by:* | ObjectSlot::slot |
| *multiplicity:* | zero or more |
| *inverse:* | Slot::object |

### *Constraints*

An Object may only own Objects and DataValues. [C-6-3]

If an Object represents an association, at least two of its ends must be not be empty. [C-6-4]

## *7.6.1.5 Slot*

A slot is a named location in an Object instance that holds the current value of the StructuralFeature associated with the Slot instance. Normally, the StructuralFeature associated with the slot will be either an Attribute instance or an AssociationEnd instance. Slots are owned by Objects; DataValues do not have slots.

### *Superclasses*

ModelElement

### *References*

#### ***feature***

References the StructuralFeature instance that describes the value held by the Slot instance.

| | |
|---|---|
| *class:* | StructuralFeature |
| *defined by:* | FeatureSlot::feature |
| *multiplicity:* | exactly one |

#### ***object***

References the Object instance that owns the Slot.

| | |
|---|---|
| *class:* | Object |
| *defined by:* | ObjectSlot::object |
| *multiplicity:* | zero or one |
| *inverse:* | Object::slot |

#### ***value***

References the DataValue or Object instance that contains the current value held by the Slot.

| | |
|---|---|
| *class:* | Instance |
| *defined by:* | SlotValue::value |
| *multiplicity:* | exactly one |

### *Constraints*

If the StructuralFeature describing a Slot is an AssociationEnd, the Classifier associated with the Object owning the Slot must be an Association. [C-6-5]

## *7.6.2 Instance Associations*

### *7.6.2.1 FeatureSlot*

The FeatureSlot association connects Slot instances with the StructuralFeature instance (usually either an Attribute or AssociationEnd instance) describing the structure of the value held by the Slot.

*Ends*

***feature***

Identifies the StructuralFeature instance for which the Slot instance contains the current value.

| | |
|---|---|
| *class:* | StructuralFeature |
| *multiplicity:* | exactly one |

***slot***

Identifies the set of Slot instances containing values of the  which the StructuralFeature instance.

| | |
|---|---|
| *class:* | Slot |
| *multiplicity:* | zero or more |

### *7.6.2.2 InstanceClassifier*

The InstanceClassifier association links Instances with Classifiers that describe them.

*Ends*

***instance***

Identifies the set of Instances described by the Classifier.

| | |
|---|---|
| *class:* | Instance |
| *multiplicity:* | zero or more |

***classifier***

Identifies the Classifier that describes the structure of the Instance.

| | |
|---|---|
| *class:* | Classifier |
| *multiplicity:* | exactly one |

### *7.6.2.3 ObjectSlot* *Protected*

The ObjectSlot association connects Slot instances with their owning Object instances.

*Ends*

***object***

Identifies the Object instance that owns the Slot instance.

| | |
|---|---|
| *class:* | Object |
| *multiplicity:* | exactly one |
| *aggregation:* | composition |

***slot***

Identifies the set of Slot instances owned by the Object instance.

| | |
|---|---|
| *class:* | Slot |
| *multiplicity:* | zero or more |

#### *7.6.2.4 SlotValue*

The SlotValue association connects slot instances with the DataValue or Object instance that contains the current value held by the slot.

##### *Ends*

***value***

Identifies the Instance subtype (either a DataValue or an Object) that holds the current value represented by the Slot instance.

| | |
|---|---|
| *class:* | Instance |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

***valueSlot***

Identifies the set of Slot instances for which the DataValue or Object instance contains the current value.

| | |
|---|---|
| *class:* | Slot |
| *multiplicity:* | zero or more |

### *7.6.3 OCL Representation of Instance Constraints*

## Constraints

[C-6-1] A DataValue originates from a Classifier that is a DataType.

**context** DataValue **inv:**

self.classifier.oclIsKindOf(DataType)

[C-6-2] A DataValue has no Slots.

**context** DataValue **inv:**

self.valueSlot->isEmpty

[C-6-3] An Object may only own Objects and DataValues.

**context** Object **inv:**

self.contents->forAll(c | c.oclIsKindOf(Object) **or** c.oclIsKindOf(DataValue))

[C-6-4] If an Object represents an association, at least two of its ends must be not be empty.

**context** Object **inv:**

```
self.classifier.oclIsKindOf(Association) implies
self.slot.feature->iterate( ae ; cnt : Integer = 0 |
   if ae.oclIsKindOf(AssociationEnd) and ae.value.notEmpty then
      cnt + 1
   else
      cnt
   end if ) > 1
```

[C-6-5] If the StructuralFeature describing a Slot is an AssociationEnd, the Classifier associated with the Object owning the Slot must be an Association.

**context** Slot **inv:**

```
self.feature.oclIsKindOf(AssociationEnd) implies
self.value.classifier.oclIsKindOf(Association)
```

# Foundation 8

## 8.1 Overview

The Foundation is a collection of metamodel packages that contain model elements representing concepts and structures that are shared by other CWM packages. Consequently, Foundation model elements often have a more general-purpose nature than model elements found in packages at higher CWM organizational levels.

Foundation model elements in a particular metamodel package are not necessarily intended to describe fully all aspects of concepts and structures they represent. Rather, they are meant to provide a common foundation which other packages can extend as necessary to meet their specific needs.

Foundation model elements differ from ObjectModel elements because they are specific to the goals and purposes of CWM. ObjectModel elements, in contrast, are of a general purpose nature and applicable in diverse areas.

## 8.2 Organization of the Foundation

The CWM uses packages to control complexity and create groupings of logically interrelated classes. The Foundation is a collection of packages that are described together because they all provide metamodel services to other CWM packages. A subsection of this chapter is devoted to each of the Foundation packages, presented in alphabetical order. The relationship between the Foundation and each of its constituent packages is shown diagrammatically in .

Organizing the Foundation in this fashion allows the individual metamodel packages to be understood and used independently of each other without sacrificing their common purpose. For example, a CWM extension package supporting a programming language might need the DataTypes, Expressions, TypeMapping and SoftwareDeployment packages but not need the KeysIndexes or BusinessInformation packages.

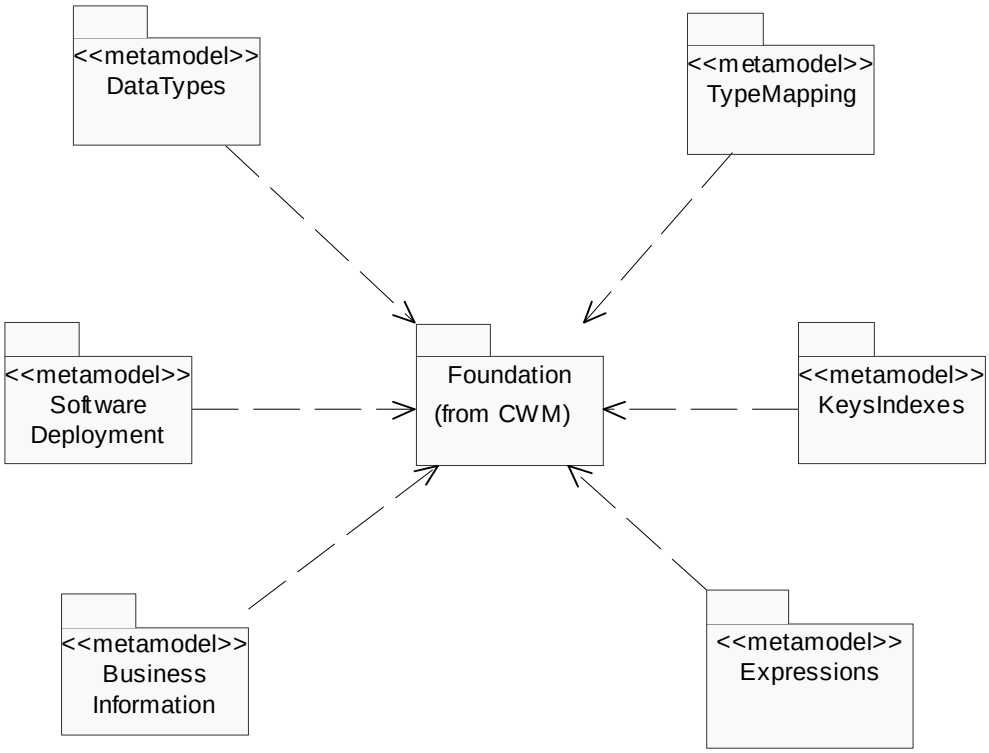


*Figure 8-2-1* Foundation Top Level Packages.

## 8.3 Business Information Metamodel

The Business Information package depends on the following package:

- org.omg::CWM::ObjectModel::Core

The Business Information Metamodel provides general purpose services available to all CWM packages for defining business-oriented information about model elements. The business-oriented services described here are designed to support the needs of data warehousing and business intelligence systems; they are not intended as a complete representation of general purpose business intelligence metamodel.

Business Information Metamodel services support the notions of responsible parties and information about how to contact them, identification of off-line documentation and support for general-purpose descriptive information. Three CWM classes “anchor” these services: ResponsibleParty, Document and Description, respectively.

The metamodel is shown in .

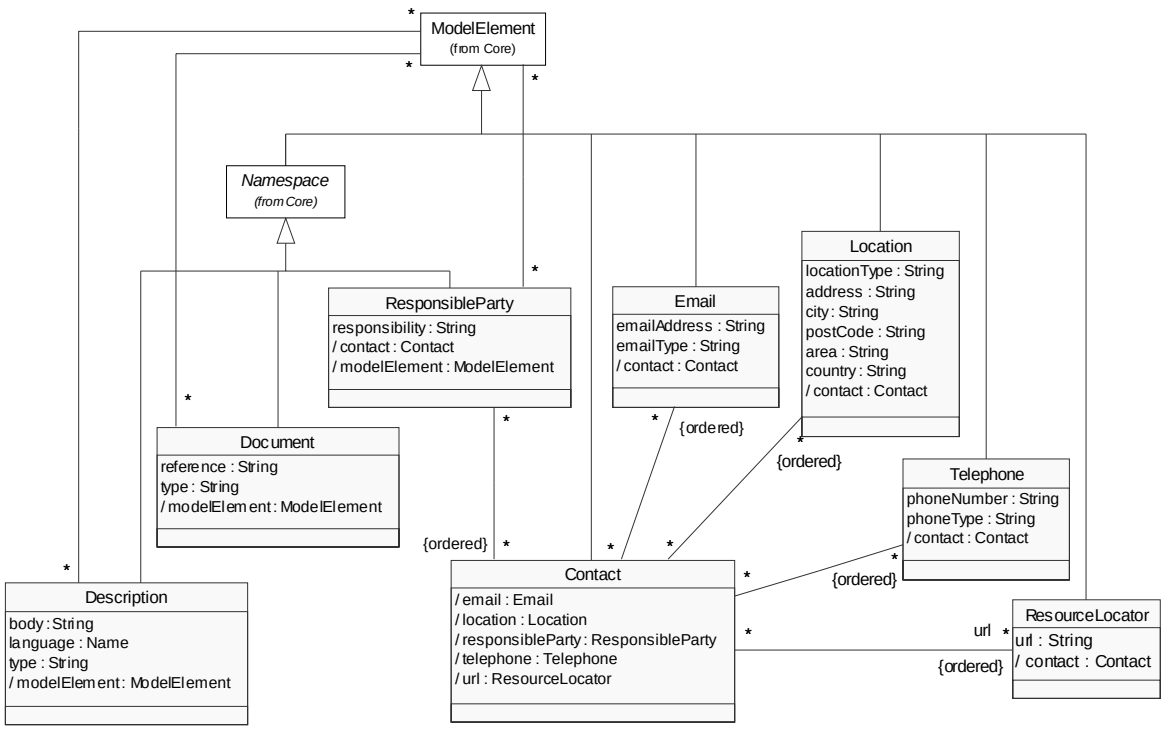


*Figure 8-3-1* BusinessInformation metamodel.

To aid in representing the diversity of organizational structures and documentation relationships that may be encountered in a business intelligence system, the metamodel provides robust relationships between the anchor classes and every model element in the CWM metamodel. The necessary robustness is achieved in several ways.

First, every CWM model element may have zero or more instances of each anchor class associated with it. This means, for example, that a single Description instance can be used to describe many different model elements. Conversely, a single model element may be described by many different Description instances. Likewise, Document and ResponsibleParty instances can be associated in completely *ad hoc* ways with any model element. Extending this idea means, for example, that Description instances could be used to further describe ResponsibleParty and Document instance, if needed.

Second, because they are Namespaces, the anchor classes can be organized into hierarchies using the ElementOwnership association. For instance, an organizational structure can be represented by a hierarchy of ResponsibleParty instances. Also, the internal structure of a document (i.e., its chapters, sections, subsections, etc.) might be represented by a hierarchy of Document instances.

Finally, instances of the three anchor classes can be associated with any model element (via their individual associations to ModelElement) and referenced by multiple instances of any of the three anchor classes. Because of the strong containment of the ElementOwnership association in the ObjectModel, anchor class instances can only participate in one hierarchy, but there are no restrictions preventing anchor class instances embedded in a hierarchy from referencing, or being referenced by, other model elements (even other members of the same hierarchy).

To illustrate some of the ways that the metamodel can be used, the following figure shows a simple document hierarchy with responsibility assignments and descriptive comments (boxes represent instances of metamodel classes and labelled lines represent metamodel associations connecting instances). In the example, the product plan document for the Widget product is composed of three subplans: a marketing plan, an engineering plan, and a resource plan. The relationships between the subplans documents is shown as a hierarchy with the product plan owning the three subordinate plans via the ElementOwnership association. Each part of the plan is assigned to a reponsible party using the ModelElementResponsibility association. Finally, Description instances are used to record roles for the responsible parties.

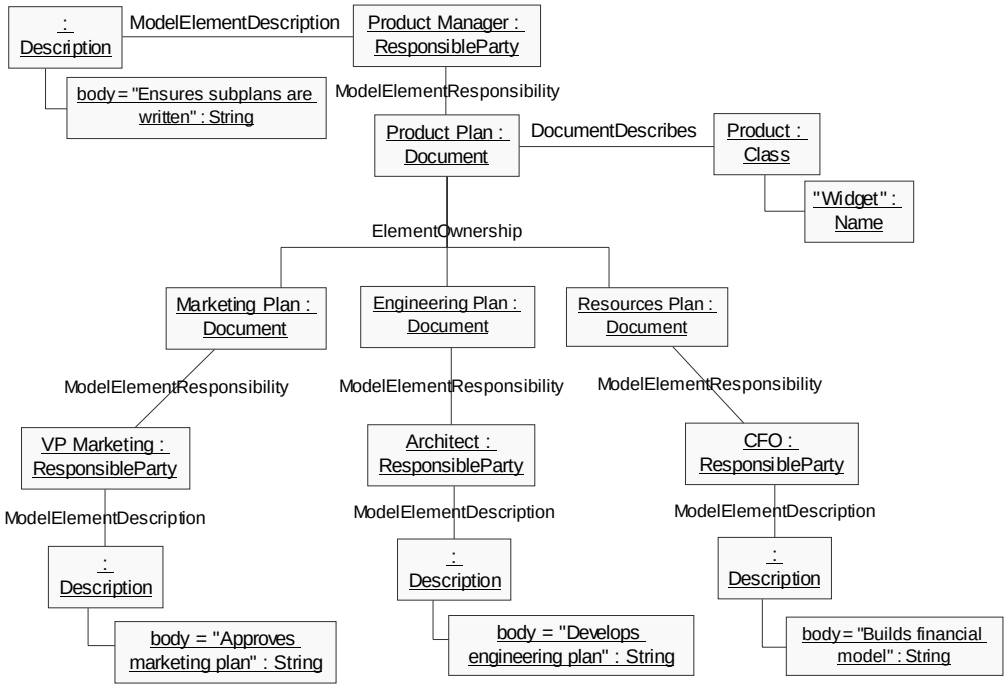


*Figure 8-3-2* Document hierarchy with assigned ResponsibleParties

Similar robustness is provided for structuring relationships between ResponsibleParty instances and the means of contacting them. Each ResponsibleParty can have multiple, ordered sets of contact information (the Contact class) and a single set of contact information can service multiple ResponsibleParties. Also, because they are not owned by any particular Contact instance, Telephone, Email, Location, and ResourceLocator instances can be reused elsewhere in the CWM metamodel. An example of the use of Business Information classes to find the ChiefEngineer at three times (Weekday, Weekend, Emergency) is shown in the following figure.

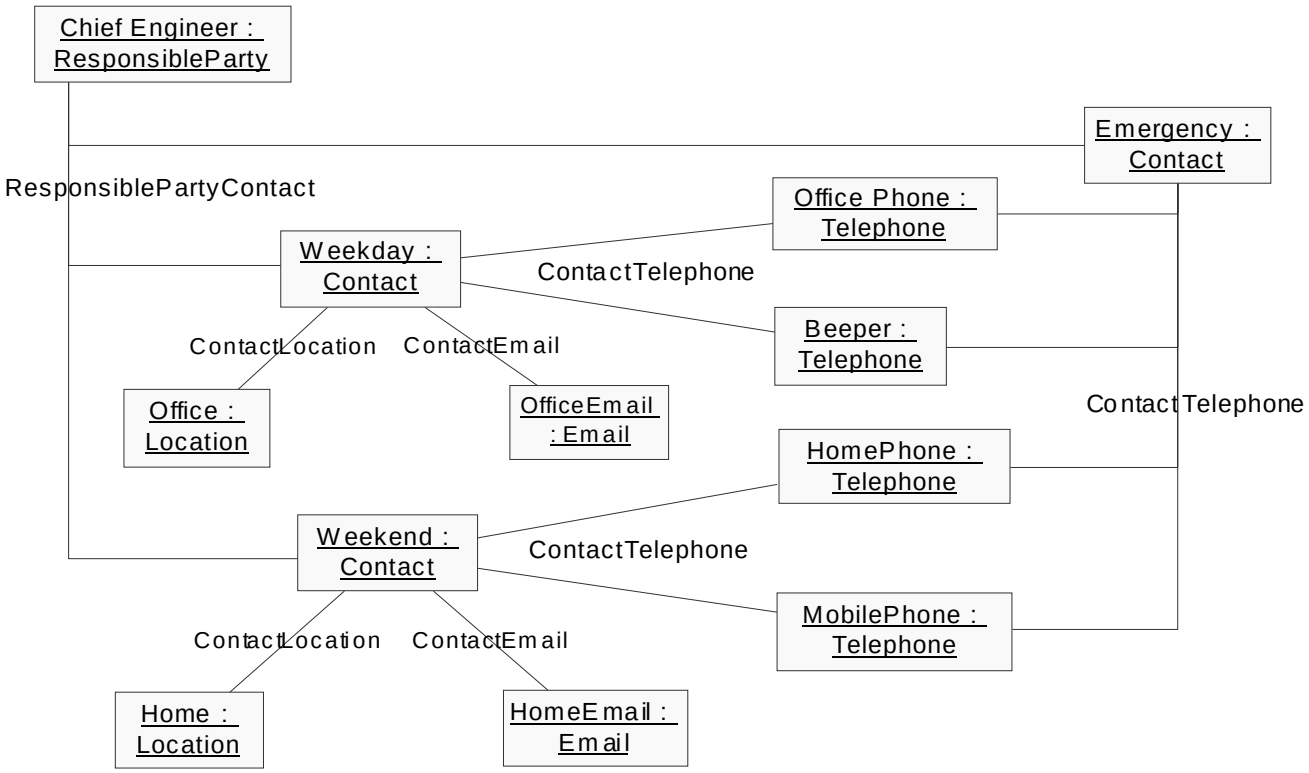


*Figure 8-3-3* Using Contact information to find the ChiefEngineer.

## 8.3.1 BusinessInformation Classes

### 8.3.1.1 Contact

Each Contact instance collects together the various types of related contact information. Each Contact instance can be associated with multiple Email, Location and Telephone instances. Conversely, each Email, Location, ResourceLocator and Telephone instance can be associated with many Contact instances. The ordering constraints on the associations between these classes and the Contact class can be used to represent a prioritized sequence in which the various types of contact information should be used.

A particular ResponsibleParty instance may have multiple instances of Contact associated with it via the ResponsiblePartyContact association. This association is ordered to support representation of the sequence in which Contact instances should be used. For example, a ResponsibleParty instance representing an employee might be associated with Contact instances representing their office, home, and mobile contact information with an indication that the employee should be contacted first at the office, then at home, and finally via their mobile phone.

To maximize flexibility of the metamodel, Contact instances may provide contact information for multiple ResponsibleParty instances.

## Superclasses

ModelElement

## References

### *email*

Identifies the Email instances associated with this Contact instance. The ordered constraint may be used to identify the order in which Email instances should be contacted.

| | |
|---|---|
| *class:* | Email |
| *defined by:* | ContactEmail::email |
| *multiplicity:* | zero or more; ordered |
| *inverse:* | Email::contact |

### *location*

Identifies the Location instances associated with this Contact instance. The ordered constraint may be used to identify the order in which Location instances should be contacted.

| | |
|---|---|
| *class:* | Location |
| *defined by:* | ContactLocation::location |
| *multiplicity:* | zero or more; ordered |
| *inverse:* | Location::contact |

### *responsibleParty*

Identifies the ResponsibleParty instances associated with this Contact instance.

| | |
|---|---|
| *class:* | ResponsibleParty |
| *defined by:* | ResponsiblePartyContact::responsibleParty |
| *multiplicity:* | zero or more |
| *inverse:* | ResponsibleParty::contact |

***telephone***

Identifies the Telephone instance associated with this Contact instance. The ordered constraint may be used to identify the order in which Telephone instances should be contacted.

| | |
|---|---|
| *class:* | Telephone |
| *defined by:* | ContactTelephone::telephone |
| *multiplicity:* | zero or more; ordered |
| *inverse:* | Telephone::contact |

***url***

Identifies the ResourceLocator instances associated with this Contact instance. The ordered constraint on the ResourceLocator association may by be used to identify the order in which ResourceLocator instances should be contacted.

| | |
|---|---|
| *class:* | ResourceLocator |
| *defined by:* | ContactResourceLocator::url |
| *multiplicity:* | zero or more; ordered |
| *inverse:* | ResourceLocator::contact |

### *8.3.1.2 Description*

Instances of the Description class contain arbitrary textual information relevant to a particular ModelElement. While Description instances may contain any desired textual information, they will typically contain documentation or references to external reference information about the owning ModelElement.

Any ModelElement may have multiple Description instances associated with it. Indeed, a ModelElement instance that is a Description instance may itself have multiple Description instances linked to it. Also, a hierarchies of Description instances can be constructed.

Description instances are meant to hold descriptive textual information that will be stored in the metamodel itself. In contrast, Document instances are meant to describe the location documentary information stored outside the metamodel.

#### *Superclasses*

Namespace

#### *Attributes*

***body***

Contains a textual description of information pertaining to the owning ModelElement.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

***language***

Contains an identification of the language in which the content of the ***body*** attribute is specified. If desired, the language specification may be applied to the ***name*** attribute derived from ModelElement as well.

| | |
|---|---|
| *type:* | Name |
| *multiplicity:* | exactly one |

***type***

Contains a textual description of the type of information the Description represents. Specific contents are usage defined.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### *References*

***modelElement***

Identifies the ModelElement for which this Description instance is relevant.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | ModelElementDescription::modelElement |
| *multiplicity:* | zero or more |

### *Constraints*

A Description instance may not describe itself [C-3-1].

## *8.3.1.3 Document*

The Document class represents externally stored descriptive information about some aspect of the modeled system. An instance of Document might be associated with one or more ModelElements. The name of a Document instance is derived from its superclasses.

Although the purposes of the Description and Document types may overlap somewhat, their chief distinction is that Description instances are stored with the CWM metadata whereas Documentation instances are stored externally to the CWM metadata. Although there is an implication here that Documentation instances might represent more voluminous information than Description instances, there is no particular requirement that this be so.

Because Documentation instances are themselves Namespace instances, hierarchical relationships between various externally stored documents can be represented.

## *Superclasses*

Namespace

## *Attributes*

### ***reference***

Contains a textual representation of the identification, and perhaps the physical location, of externally maintained documentary information about some aspect of the ModelElement(s) with which the Document instance is associated.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### ***type***

Contains a textual description of the type of information the Document represents. Specific contents are usage defined.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

## *References*

### ***modelElement***

Identifies the ModelElement(s) for which this Document instance is relevant.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | DocumentDescribes::modelElement |
| *multiplicity:* | zero or more |

## *Constraints*

A Document instance may not describe itself [C-3-2].

### 8.3.1.4 Email

An Email instance identifies a single email address. Via a Contact instance, an email address can be associated with one or more ResponsibleParty instances. Email instances might, for example, be used by an automated tool to send an automatically generated email message to a ResponsibleParty instance responsible about some change of state for a particular ModelElement. Multiple Email instances may be associated with a single Contact instance and the ordering of the association between them may be used to represent the sequence in which the Email instances should be contacted.

Because email addresses are first class objects within the CWM, they can be used for purposes beyond those associated with the CWMFoundation's Business Information concepts.

#### Superclasses

ModelElement

#### Attributes

***eMailAddress***

A textual representation of an email address.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

***eMailType***

Contains a textual representation of the type of the email address. Interesting values might include location information such as "home" or "work", or perhaps an indication of the type of email system for which the eMailAddress is formatted, such as "SMTP" or "X.400".

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

#### References

***contact***

Identifies the Contact instance(s) for which this Email instance is relevant.

| | |
|---|---|
| *class:* | Contact |

| | |
|---|---|
| *defined by:* | ContactEmail::contact |
| *multiplicity:* | zero or more |
| *inverse:* | Contact::email |

### 8.3.1.5 Location

Instances of the Location class represent physical locations. Note that the ***name*** of a Location is derived from its superclass, ModelElement.

Because Locations are first class objects within the CWM, they can be used for purposes beyond those associated with the CWM Foundation's Business Information concepts. If additional attributes about Location instances are required, they should be added by creating subtypes of the Location class and placing the additional attributes therein.

#### Superclasses

ModelElement

#### Attributes

***locationType***

Descriptive information about the character or identity of the Location instance.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

***address***

The address of the Location instance. The precise content of this string is usage-defined.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

***city***

The name of the city in which the Location instance is found. The precise content of this string is usage-defined.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one. |

***postCode***

The postal code of the Location instance. The precise content of this string is usage-defined.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

***area***

The area in which the Location instance is found. The precise content of this string is usage-defined, but a common usage would be to refer to a geographical subdivision such as a state or province.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

***country***

The name of the country in which the Location instance is found. The precise content of this string is usage-defined.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### *References*

***contact***

Identifies the Contact instance(s) with which this Location instance is associated.

| | |
|---|---|
| *class:* | Contact |
| *defined by:* | ContactLocation::contact |
| *multiplicity:* | zero or more |
| *inverse:* | Contact::location |

## *8.3.1.6 ResourceLocator*

Instances of the ResourceLocator class provide a general means for describing the resources whose location is not defined by a traditional mailing address. For example, a ResourceLocator instance could refer to anything from a location within a building (“Room 317, third file cabinet, 2nd drawer”) to a web location (“www.omg.org”).

Because they are first class objects in the CWM, ResourceLocator instances may also be used for purposes beyond those associated with the CWM Foundation's Business Information concepts.

### *Superclasses*

ModelElement

### *Attributes*

***url***

Contains the text of the resource location. For Internet locations, this will be a web URL (Uniform Resource Locator) but there is no requirement that this be so. In fact, the string can contain any text meaningful to its intended use in a particular environment.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### *References*

***contact***

Identifies the Contact instance(s) for which the ResourceLocator instance is relevant.

| | |
|---|---|
| *class:* | Contact |
| *defined by:* | ContactResourceLocator::contact |
| *multiplicity:* | zero or more |
| *inverse:* | Contact::url |

## *8.3.1.7 ResponsibleParty*

The ResponsibleParty class allows representation of entities within an information system that are in some way interested in receiving information about, or are otherwise responsible for, particular ModelElements. Each ResponsibleParty may own multiple sets of contact information, and a single ResponsibleParty may be associated with many ModelElements.

ResponsibleParty instances may represent any entity appropriate to the system being modeled and need not be limited to people. For example, a ResponsibleParty instance might represent an individual such as "George Washington", a role (the "President"), or an organization ("Congress"), depending upon the needs of the system being modeled. Similarly, the precise semantics of the ***responsibility*** attribute are open to intrepretation and may be adapted on a system-by-system basis.

Because ResponsibleParty instances are Namespaces, they can be organized into hierarchies of ResponsibleParty instances, capturing organizational structures or similar relationships.

### Superclasses

Namespace

### Attributes

***responsibility***

Textual identification or description of the ResponsibleParty in a usage-dependent format.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### References

***contact***

Identifies the Contact instance(s) associated with a ResponsibleParty instance. The ordered constraint on this reference allows retention of the sequence in which multiple Contact should be employed.

| | |
|---|---|
| *class:* | Contact |
| *defined by:* | ResponsiblePartyContact::contact |
| *multiplicity:* | zero or more; ordered |
| *inverse:* | Contact::responsibleParty |

***modelElement***

Identifies the model elements for which this ResponsibleParty instance has some interest or responsibility.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | ModelElementResponsibility::modelElement |
| *multiplicity:* | zero or more |

### Constraints

A ResponsibleParty instance may not be responsible for itself. [C-3-3]

## 8.3.1.8 Telephone

Instances of the Telephone class represent telephone contact information.

Because telephones are first class objects within the CWM, they can be used for purposes beyond those associated with the CWM Foundation's Business Information concepts.

### *Superclasses*

ModelElement

### *Attributes*

***phoneNumber***

A textual representation of the telephone's number.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

***phoneType***

A textual representation of the telephone's type, such as "multi-line", or its usage, such as "home", "work", "fax", or "mobile".

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### *References*

***contact***

Identifies the Contact instance(s) for which the Telephone instance is relevant.

| | |
|---|---|
| *class:* | Contact |
| *defined by:* | ContactTelephone::contact |
| *multiplicity:* | zero or more |
| *inverse:* | Contact::telephone |

## *8.3.2 BusinessInformation Associations*

## *8.3.2.1 ContactEmail* *Protected*

The ContactEmail association indicates the Email instances relevant used by Contact instances.

### *Ends*

***contact***

Identifies the Contact instance(s) for which this Email instance is relevant.

*class:* Contact

*multiplicity:* zero or more

***email***

Identifies the Email instances associated with this Contact instance. The ordered constraint may be used to identify the order in which Email instances should be contacted.

*class:* Email

*multiplicity:* zero or more; ordered

### *8.3.2.2 ContactLocation* *Protected*

The ContactLocation association relates Contact instances to relevant Location instances.

*Ends*

***contact***

Identifies the Contact instance(s) with which this Location instance is associated.

*class:* Contact

*multiplicity:* zero or more

***location***

Identifies the Location instances associated with this Contact instance. The ordered constraint may be used to identify the order in which Location instances should be contacted.

*class:* Location

*multiplicity:* zero or more; ordered

### *8.3.2.3 ContactResourceLocator* *Protected*

The ContactResourceLocator association relates ResourceLocator instances to the Contact instances in which they participate.

*Ends*

***contact***

Identifies the Contact instances for which a ResourceLocator instance is relevant.

| | |
|---|---|
| *class:* | Contact |
| *multiplicity:* | zero or more |

***url***

Identifies the ResourceLocator instances related to this ContactInfo instance. Note that the ordered constraint on this role can be used to indicate the sequence in which ResourceLocator should be contacted.

| | |
|---|---|
| *class:* | Telephone |
| *multiplicity:* | zero or more; ordered |

### *8.3.2.4 ContactTelephone* *Protected*

The ContactTelephone association relates telephones to the Contact instances that reference them.

*Ends*

***contact***

Identifies the Contact instance(s) for which the Telephone instance is relevant.

| | |
|---|---|
| *class:* | Contact |
| *multiplicity:* | zero or more |

***telephone***

Identifies the Telephone instance associated with this Contact instance. The ordered constraint may be used to identify the order in which Telephone instances should be contacted.

| | |
|---|---|
| *class:* | Telephone |
| *multiplicity:* | zero or more; ordered |

### *8.3.2.5 DocumentDescribes*

The DocumentDescribes association connects a Document instance with the ModelElement instances to which it pertains.

*Ends*

***modelElement***

Identifies the ModelElement instances for which this Document instance is relevant.

| | |
|---|---|
| *class:* | ModelElement |
| *multiplicity:* | zero or more |

***document***

Identifies the Document instances relevant to a particular ModelElement.

| | |
|---|---|
| *class:* | Document |
| *multiplicity:* | zero or more |

### 8.3.2.6 ModelElementDescription

The ModelElementDescription association connects a Description instance with the ModelElement instances to which it applies.

*Ends*

***modelElement***

Identifies the ModelElement instances for which this Description instance is relevant.

| | |
|---|---|
| *class:* | ModelElement |
| *multiplicity:* | zero or more |

***description***

Identifies the Description instances relevant for a particular ModelElement instance.

| | |
|---|---|
| *class:* | Description |
| *multiplicity:* | zero or more |

### 8.3.2.7 ModelElementResponsibility

The ModelElement Responsibility association identifies the ResponsibleParty instances for each ModelElement and allows determination of the ModelElements for which a ResponsibleParty instance is responsible.

*Ends*

***modelElement***

Identifies the model elements for which this ResponsibleParty instance has some interest or responsibility.

| | |
|---|---|
| *class:* | ModelElement |
| *multiplicity:* | zero or more |

***responsibleParty***

Identifies the ResponsibleParty instances relevant for a particular ModelElement instance.

| | |
|---|---|
| *class:* | ResponsibleParty |
| *multiplicity:* | zero or more |

### *8.3.2.8 ResponsiblePartyContact* *Protected*

The ResponsiblePartyContact association allows a ResponsibleParty to have multiple sets of contact information. The ordered constraint can be used to determine the sequence in which the sets of contact information should be used.

*Ends*

***contact***

Identifies the Contact instance(s) associated with a ResponsibleParty instance. The ordered constraint on this reference allows retention of the sequence in which multiple Contact should be employed.

| | |
|---|---|
| *class:* | Contact |
| *multiplicity:* | zero or more; ordered |

***responsibleParty***

Identifies the ResponsibleParty instances associated with this Contact instance.

| | |
|---|---|
| *class:* | ResponsibleParty |
| *multiplicity:* | zero or more |

## *8.3.3 OCL Representation of BusinessInformation Constraints*

[C-3-1] A Description may not describe itself.

**context** Description **inv:**

self.modelElement->forAll(p | p <> self)

[C-3-2] A Document may not describe itself.

**context** Document **inv:**

self.modelElement->forAll(p | p <> self)

[C-3-3] A ResponsibleParty may not describe itself.

**context** ResponsibleParty **inv:**

self.modelElement->forAll(p | p <> self)

## 8.4 DataTypes Metamodel

The DataTypes package depends on the following packages:

- org.omg::CWM::ObjectModel::Core

The CWM DataTypes metamodel supports definition of metamodel constructs that modelers can use to create the specific data types they need. Although the CWM Foundation itself does not contain specific data type definitions, a number of data type definitions for widely used environments are provided (in the CWM Data Types chapter) as examples of the appropriate usage of CWM Foundation classes for creating data type definitions.

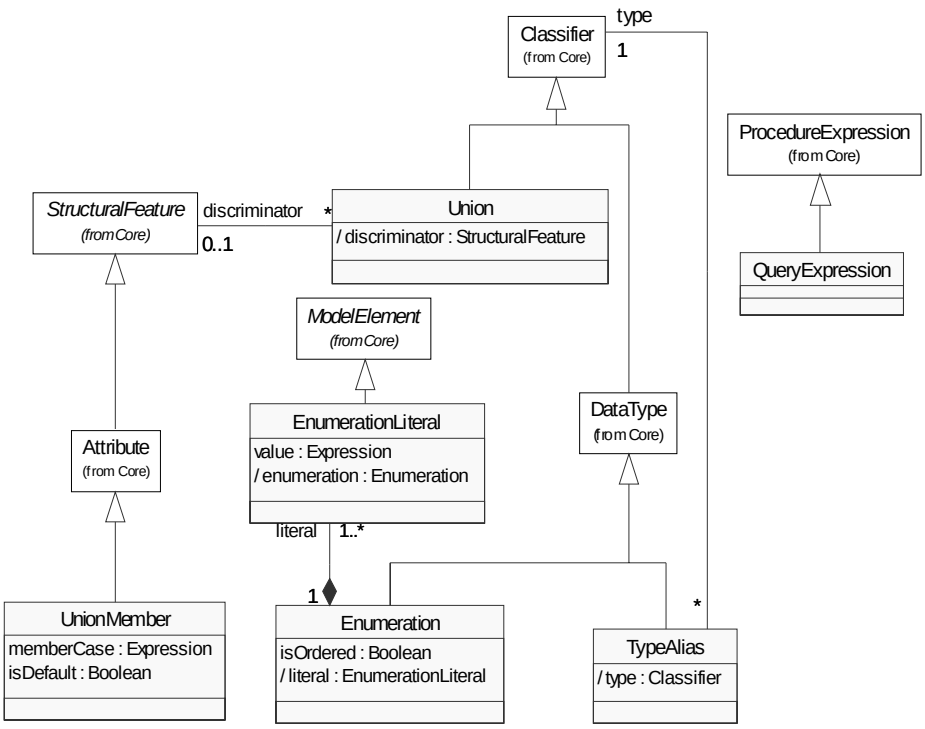


*Figure 8-4-1* DataTypes metamodel.

### 8.4.1 DataTypes Classes

#### 8.4.1.1 Enumeration

The Enumeration class is intended as a starting point from which enumerated data types can be created. An enumerated data type is a collection of identifiers often used as the permitted states that some other attribute or property of the enumerated type may take.

The ***isOrdered*** attribute of an Enumeration instance is used to determine if the ordered constraint on the EnumerationLiterals association is relevant for the enumeration. The particular ordering of EnumerationLiteral instances is obtained from the ordered constraint on the association even if the ***value*** attributes of the EnumerationLiteral instances contain non-null values that might be used to determine ordering. This is done to provide more flexible ordering semantics.

An instance of Enumeration is also required to create a range data type. Refer to the EnumerationLiteral class for details.

## *Superclasses*

DataType

## *Contained Elements*

EnumerationLiteral

## *Attributes*

### *isOrdered*

If True, the ordered constraint on the EnumerationLiterals association is relevant. Otherwise, the ordering of EnumerationLiteral instances is considered unspecified.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

## *References*

### *literal*

Identifies the EnumerationLiteral instances relevant for a particular Enumeration instance. If the Enumeration's isOrdered attribute is True, the ordering constraint on this reference end can be used to determine a logical ordering for the EnumerationLiteral instances. Otherwise, ordering is ignored.

| | |
|---|---|
| *class:* | EnumerationLiteral |
| *defined by:* | EnumerationLiterals::literal |
| *multiplicity:* | one or more; ordered |
| *inverse:* | EnumerationLiteral::enumeration |

### 8.4.1.2 EnumerationLiteral

EnumerationLiteral instances describe the enumeration identifiers, and possibly the values, associated with an enumerated data type. Enumeration identifiers are contained in the ***name*** attribute derived from the EnumerationLiteral instance's ModelElement superclass.

EnumerationLiteral instances may also be used to define expression-based values such as ranges. To do so, simply state the membership expression in the instance's value. For example, a range literal can be created by setting the ***value*** attribute to "*m..n*", where *m* represents the lower bound of the range, and *n*, the upper bound. In this way, ranges and other more complicated expressions can be intermixed with simple enumeration literals. For example, an enumeration might contain the literals "1", "2", "4..7", and "> 10".

Consequently, a simple range data type can be created with an Enumeration instance that owns a single EnumerationLiteral instance. For example, a data type for positive integers could be created as shown in the following instance diagram. A model attribute of this data type might then be declared as "posInt : PositiveInteger".

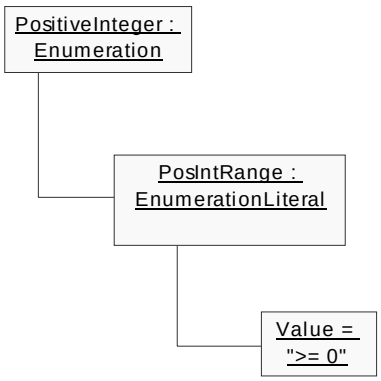


*Figure 8-4-2* Using Enumeration and EnumerationLiteral instances to create range data types.

#### *Superclasses*

ModelElement

#### *Attributes*

***value***

The value associated with an enumeration identifier can be stored here. The attribute is optional because enumeration literals are not required to have a specific, displayable value. This is indicated by either an empty ***value*** attribute or a ***value*** attribute value whose expression ***body*** attribute is a zero-length string.

| | |
|---|---|
| *type:* | Expression |
| *multiplicity:* | zero or more |

### *References*

***enumeration***

Identifies the Enumeration instance for which this enumeration literal is relevant.

| | |
|---|---|
| *class:* | Enumeration |
| *defined by:* | EnumerationLiterals::enumeration |
| *multiplicity:* | exactly one |
| *inverse:* | Enumeration::literal |

## *8.4.1.3 QueryExpression*

QueryExpression instances contain query statements in language-dependent form.

### *Superclasses*

ProcedureExpression

## *8.4.1.4 TypeAlias*

The TypeAlias class is intended to provide a renaming capability for Classifier instances. This class is required to support situations in which creation of an alias for a class effectively creates a new class. For example, CORBA IDL type aliases have different typeCodes than their base types and are therefore treated as distinct types.

### *Superclasses*

DataType

### *References*

***type***

Identifies the Classifier instance for which this TypeAlias instance acts as an alias.

| | |
|---|---|
| *class:* | Classifier |
| *defined by:* | ClassifierAlias::type |
| *multiplicity:* | exactly one |

### *Constraints*

A TypeAlias instance cannot alias itself. [C-4-1]

## *8.4.1.5 Union*

The Union class represents programming language unions and similarly structured data types. Because of the diversity of union semantics found across software systems, the Union and UnionMember classes are likely candidates for specialization to better capture union semantics in specific language extension packages.

A discriminated Union has a collection of UnionMembers that determine the sets of contents that the Union may contain. Such Unions have an attribute called the discriminator that identifies the memberCase value of the UnionMember that the Union currently contains. The discriminator is found via the UnionDiscriminator association to StructuralFeature. The discriminator may be embedded within UnionMembers or may be located outside the discriminator. If it is located within UnionMembers, the discriminator should occur in every UnionMember at the same location (often, the first).

Undiscriminated unions (for example, a C language union) are also supported, but have an empty discriminator reference, and the memberCase attribute of the UnionMembers it contains is ignored.

Undiscriminated Unions are often used to represent alternate views of a single physical storage area. A fine degree of control over this aspect of Unions may be obtained by creating a class that derives from both UnionMember and FixedOffsetField (in the CWM Record package) and setting the offset attribute instances of that class accordingly.

### *Superclasses*

Classifier

### *Contained Elements*

UnionMember

### *References*

***discriminator***

Identifies the StructuralFeature instance that serves as the discriminator for the Union.

| | |
|---|---|
| *class:* | StructuralFeature |
| *defined by:* | UnionDiscriminator::discriminator |
| *multiplicity:* | zero or more |

### *Constraints*

A Union can have at most one default UnionMember instance. [C-4-2]

## *8.4.1.6 UnionMember*

UnionMembers are described as features of a Union and each represents one of the members of a Union. Note, however, that multiple case values can map to a single UnionMember. If isDefault is true, the union member is the default member. UnionMember instances are allowed to have a memberCase and be the default case.

UnionMember instances often represent structured storage areas. A particular UnionMember may be associated with a Classifier that describes its internal structure via the StructuralFeatureType association (defined in the ObjectModel::Core package). For example, the Record::Group class, itself a Classifier, can be used as the type of a UnionMember in a manner completely analogous to how it is used to describe the type of a structured field (see the instance diagrams in the Record metamodel chapter for details).

### *Superclasses*

Attribute

### *Attributes*

***memberCase***

Contains the value of the Union's discriminator for this UnionMember.

| | |
|---|---|
| *type:* | Expression |
| *multiplicity:* | exactly one |

***isDefault***

Indicates if this UnionMember is the default member of the Union (implying that when unstated, the Union's discriminator would assume this instance's ***memberCase*** value).

*type:* Boolean

*multiplicity:* exactly one

## 8.4.2 DataTypes Associations

### 8.4.2.1 ClassifierAlias

The ClassifierAlias association connects TypeAlias instances with the Classifier instances which they rename.

*Ends*

***type***

Identifies the Classifier instance for which this TypeAlias instance acts as an alias.

*class:* Classifier

*multiplicity:* exactly one

***alias***

Identifies the TypeAliases that have be defined for a particular Classifier instance.

*class:* TypeAlias

*multiplicity:* zero or more

### 8.4.2.2 EnumerationLiterals *Protected*

The EnumerationLiterals association links enumeration literals to the Enumeration instances that contain them.

If the Enumeration's ***isOrdered*** attribute is True, the ordering constraint on the association is relevant. Otherwise, it is ignored.

*Ends*

***enumeration***

Identifies the Enumeration instance for which this enumeration literal is relevant.

| | |
|---|---|
| *class:* | Enumeration |
| *multiplicity:* | exactly one |

***literal***

Identifies the EnumerationLiteral instances relevant for a particular Enumeration instance. If the Enumeration's ***isOrdered*** attribute is True, the ordering constraint on this association end can be used to determine a logical ordering for the EnumerationLiteral instances. Otherwise, ordering is ignored.

| | |
|---|---|
| *class:* | EnumerationLiteral |
| *multiplicity:* | one or more; ordered |
| *aggregation:* | composite |

### 8.4.2.3 UnionDiscriminator

The UnionDiscriminator association connects a Union instance with the StructuralFeature instance that can be used to determine which UnionMember instance is currently present in the Union instance. This "discriminating" attribute may be a feature of the UnionMembers themselves or may be a feature of some Classifier that contains the Union instance as one of its Features. In the former case, the discriminating feature will usually be present at the same offset in each UnionMember instance. If the discriminator reference is empty for a particular Union instance, it is considered to be an "undiscriminated" Union and determination of the current UnionMember residing in the Union is usage-defined.

#### *Ends*

***discriminator***

Identifies the StructuralFeature instance that serves as the discriminator for the Union.

| | |
|---|---|
| *class:* | StructuralFeature |
| *multiplicity:* | zero or one |

***discriminatedUnion***

Identifies the Union instances in which a particular StructuralFeature acts as the discriminator.

| | |
|---|---|
| *class:* | Union |
| *multiplicity:* | zero or more |

## 8.4.3 OCL Representation of DataTypes Constraints

[C-4-1] A TypeAlias instance cannot alias itself.

**context** TypeAlias **inv:**

self.type <> self

[C-4-2] A Union can have at most one default UnionMember instance.

**context** Union **inv:**

self.allFeatures->select(isDefault)->size <= 1

## 8.5 Expressions Metamodel

The Expressions package depends on the following packages:

- org.omg::CWM::ObjectModel::Core

The CWM Expressions metamodel provides basic support for the definition of expression trees within the CWM. The intent of the Expressions metamodel is to provide a place for other CWM packages (such as Transformation) and CWM compliant tools to record shared expressions in a common form that can be used for interchange and lineage tracking.

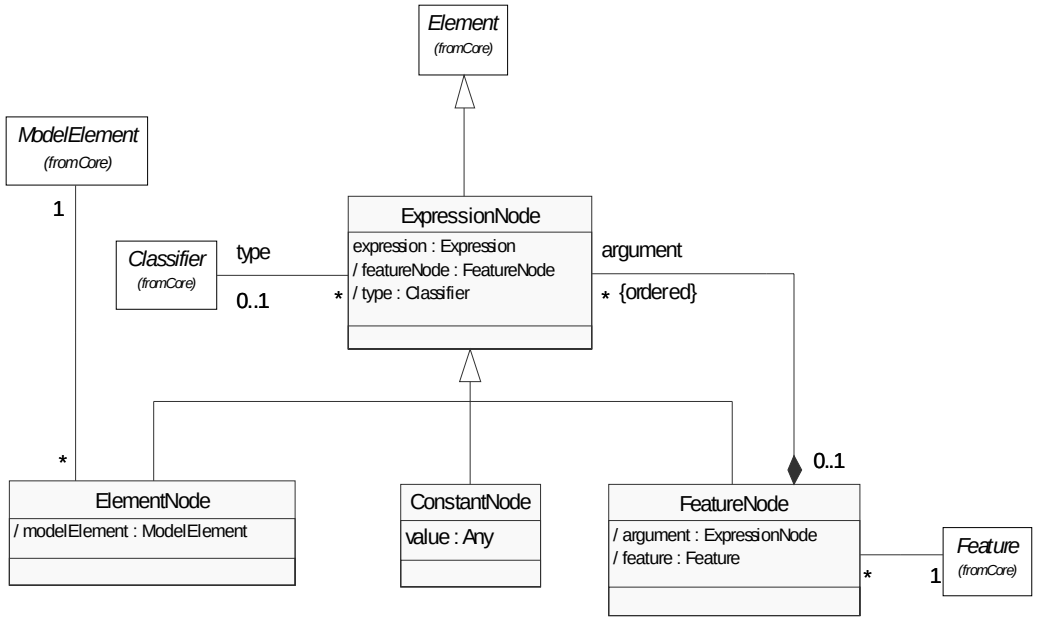


*Figure 8-5-1* Expressions metamodel.

The expression concept in the CWM Foundation takes a functional view of expression trees, resulting in the ability of relatively few expression types to represent a broad range of expressions. Every function or traditional mathematical operator that appears in an expression hierarchy is represented as a FeatureNode. For example, the arithmetic plus operation “a + b” can be thought of as the function “sum(a, b).” The semantics of a particular function or operation are left to specific tool implementations and are not captured by the CWM.

The hierarchical nature of the CWM’s representation of expressions is achieved by the recursive nature of the OperationArgument association. This association allows the sub-hierarchies within an expression to be treated as actual parameters of their parent nodes.

By way of example, the following instance diagram shows one representation of a CWM expression tree for the well-known Einstein equation $E = mc^2$. To better

understand how the equation is mapped into the expression tree, the formula can be rewritten in a functional notation as

Assign(*E*, Multiply(*m*, Power(*c*, *2*))).

This functional form of the equation is then mapped into a set of expression tree instances as shown in the following figure.

Alternatively, if sharing and lineage tracking of elements within the expression are not required, the expression could be stored using an Attribute of type ExpressionNode by assigning the string "E = $mc^2$" as the Attribute's ***expression::body*** value. For flexibility, use of the ***expression*** attribute within an expression hierarchy is allowed, but the precise semantics of such situations are not defined by CWM.

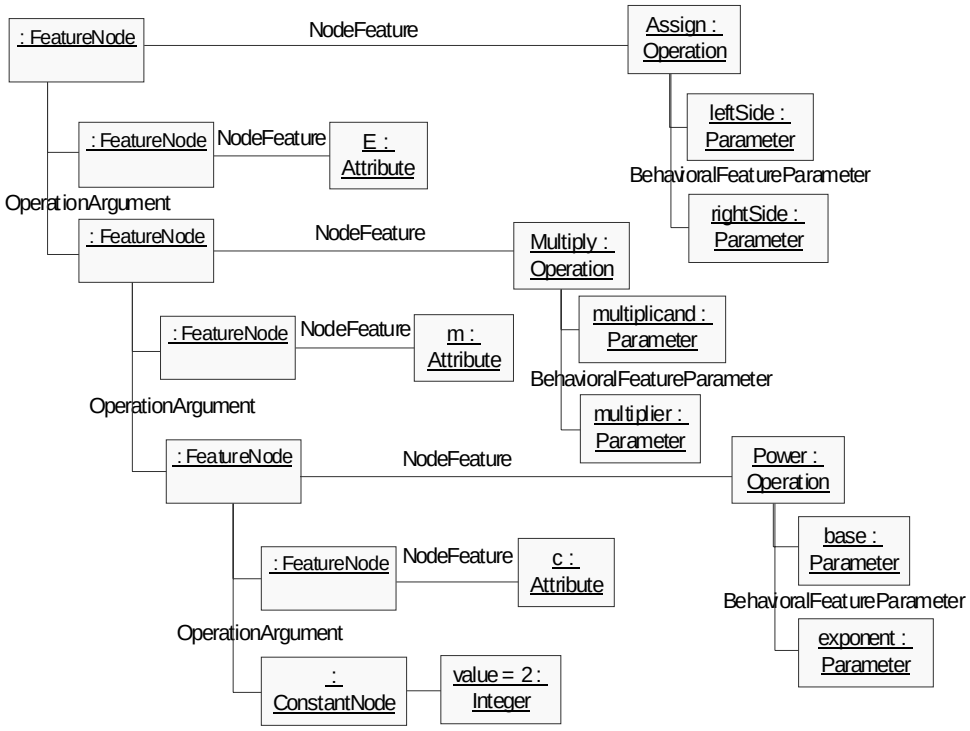


*Figure 8-5-2* A CWM expression tree for the formula $E = mc^2$.

## *8.5.1 Expressions Classes*

### *8.5.1.1 ConstantNode*

Instances of the ConstantNode class are ExpressionNodes that represent constant values within expressions. Appropriate uses of the ConstantNode class place the values of constants in the ***value*** attribute, rather than in the ***expression::body*** attribute inherited from ExpressionNode. The latter attribute is intended for a different purpose; see the description of the ExpressionNode class for details.

### *Superclasses*

ExpressionNode

### *Attributes*

***value***

The value of a constant in an expression tree.

| | |
|---|---|
| *type:* | Any |
| *multiplicity:* | exactly one |

## *8.5.1.2 ElementNode*

An ElementNode is a node in an expression that references some ModelElement instance. This subclass of ExpressionNode allows an expression to reference any CWM model element that is not a Feature and cannot, therefore, be represented as a FeatureNode.

Typically, use of this subclass of ExpressionNode implies that a tool attempting to evaluate the expression will be able to determine if the referenced ModelElement instance is also an instance of some interesting subclass of ModelElement that contains a value of interest in the expression.

### *Superclasses*

ExpressionNode

### *References*

***modelElement***

Identifies the ModelElement instance which this ElementNode references.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | ReferencedElement::modelElement |
| *multiplicity:* | exactly one |

## *8.5.1.3 ExpressionNode*

All node types within an expression are derived from the ExpressionNode type.

An expression is stored as a collection of instances of the subtypes of ExpressionNode arranged in a hierarchical fashion. The ExpressionNode instance at the top (or "root") of the hierarchy represents the value of the expression and serves as a starting point for expression evaluation. Refer to the descriptions of the subtypes of ExpressionNode (ElementNode, ConstantNode, and FeatureNode) for additional information about the representation of expressions.

One important purpose for providing storage of expressions as a general feature of the CWM is to promote sharing them between tools and to provide a means for recording lineage relationships between components within expressions. Specific details of the implementation of expression operators are left to the implementing tools.

When ExpressionNode is used as the type of an Attribute, an instance of the Attribute can contain either an expression tree as described here or a textual representation of the expression in *body* and *language* values of in an attribute of type Expression (defined ObjectModel). The ***expression*** attribute is provided for the latter usage. To obtain CWM's sharing and lineage tracking features for elements within an expression, the expression must be represented as an expression hierarcy.

## *Superclasses*

Element

## *Attributes*

### ***expression***

Contains a textual representation of the expression relevant for this ExpressionNode instance.

| | |
|---|---|
| *type:* | Expression |
| *multiplicity:* | zero or one |

## *References*

### ***featureNode***

Identifies the FeatureNode for which this ExpressionNode instance represents the value of an argument. Because arguments can themselves represent entire expression sub-trees, this reference is used to create hierarchies of expression nodes, permitting representation of entire expression trees within the CWM.

| | |
|---|---|
| *class:* | FeatureNode |
| *defined by:* | OperationArgument::featureNode |
| *multiplicity:* | zero or one |
| *inverse:* | FeatureNode::argument |

***type***

Identifies the Classifier instance that represents the type of the expression at this level in the expression hierarchy. Although, formally, each node within an expression tree is capable of having a value and therefore, a data type, this reference is optional because modeling the data type of intermediate nodes in an expression tree is not always interesting, thereby reducing the effort required to create expression trees.

| | |
|---|---|
| *class:* | Classifier |
| *defined by:* | ExpressionNodeClassifier::type |
| *multiplicity:* | zero or one |

### *8.5.1.4 FeatureNode*

The FeatureNode class represents ExpressionNode instances that are features (i.e., attributes or operations) of some Classifier instance within the CWM.

A FeatureNode with a null OperationArgument association represents either a parameter-less operation or an attribute value obtained from some StructuralFeature instance.

#### *Superclasses*

ExpressionNode

#### *Contained Elements*

ExpressionNode

#### *References*

***argument***

Identifies the ExpressionNode instances that represent the actual arguments for this FeatureNode. By convention, the first actual argument is a reference to the object itself. If the ***argument*** reference is null, the FeatureNode is an attribute or parameter-less function or operation.

| | |
|---|---|
| *class:* | ExpressionNode |
| *defined by:* | OperationArgument::argument |
| *multiplicity:* | zero or more; ordered |
| *inverse:* | ExpressionNode::featureNode |

***feature***

Identifies the Feature (attribute or operation) which this FeatureNode instance represents.

| | |
|---|---|
| *class:* | Feature |
| *defined by:* | NodeFeature::feature |
| *multiplicity:* | exactly one |

### *Constraints*

A FeatureNode that has parameters other than the “this” parameter represents a Feature that is also an Operation. [C-5-1]

If the FeatureNode represents an instance-scope feature, the first argument is a “this” or “self” argument; that is, the object invoking the feature. The convention is enforced by checking that the type of the first argument is the same as the type of the feature. [C-5-2]

If the FeatureNode represents a BehavioralFeature, the number of arguments must be equal to the number of the BehavioralFeature’s parameters, plus one for the “this” parameter if the BehavioralFeature is of instance scope. [C-5-3]

If the FeatureNode represents a BehavioralFeature, the types of the arguments must match, in order, the types of the parameters, allowing for the optional presence of a leading “this” parameter. [C-5-4]

## *8.5.2 Expressions Associations*

## *8.5.2.1 ExpressionNodeClassifier*

The ExpressionNodeClassifier association identifies the type of an ExpressionNode instance.

### *Ends*

***expressionNode***

Identifies the ExpressionNode instances for which this Classifier acts as the type.

| | |
|---|---|
| *class:* | ExpressionNode |
| *multiplicity:* | zero or more |

***type***

Identifies the Classifier instance that represents the type of the expression at this level in the expression hierarchy. Although, formally, each node within an expression tree is capable of having a value and therefore, a data type, this reference is optional because modeling the data type of intermediate nodes in an expression tree is not always interesting, thereby reducing the effort required to create expression trees.

| | |
|---|---|
| *class:* | Classifier |
| *multiplicity:* | zero or one |

### 8.5.2.2 NodeFeature

The NodeFeature association identifies the Feature (usually, an Attribute or Operation subtype) that FeatureNode represents.

#### Ends

***feature***

Identifies the Feature (attribute or operation) which this FeatureNode instance represents.

| | |
|---|---|
| *class:* | Feature |
| *multiplicity:* | exactly one |

***featureNode***

Identifies the FeatureNode instances that use a particular Feature.

| | |
|---|---|
| *class:* | FeatureNode |
| *multiplicity:* | zero or more |

### 8.5.2.3 OperationArgument *Protected*

The OperationArgument association identifies and orders the actual arguments of an Operation indicated by the FeatureNode end of the association. This association is meaningful only if the FeatureNode references, via the NodeFeature association, a Feature that is also an Operation. The association is not meaningful under other circumstances.

#### Ends

***argument***

Identifies the ExpressionNode instances that represent the actual arguments for this FeatureNode. If the argument reference is null, the FeatureNode is an attribute or parameter-less function or operation.

| | |
|---|---|
| *class:* | ExpressionNode |
| *multiplicity:* | zero or more; ordered |

***featureNode***

Identifies the FeatureNode for which this ExpressionNode instance represents the value of an argument. Because arguments can themselves represent entire expression sub-trees, this reference is used to create hierarchies of expression nodes, permitting representation of entire expression trees within the CWM.

| | |
|---|---|
| *class:* | FeatureNode |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

### *8.5.2.4 ReferencedElement*

The ReferencedElement association links the ElementNode instances of an expression with the ModelElement instances to which they refer.

#### *Ends*

***elementNode***

Identifies the ElementNode instances that represent a particular ModelElement in expressions.

| | |
|---|---|
| *class:* | ElementNode |
| *multiplicity:* | zero or more |

***modelElement***

Identifies the ModelElement instance which this ElementNode references.

| | |
|---|---|
| *class:* | ModelElement |
| *multiplicity:* | exactly one |

### 8.5.3 OCL Representation of Expressions Constraints

[C-5-1] A FeatureNode that has parameters other than the “this” parameter represents a Feature that is also an Operation.

**context** FeatureNode **inv:**

```
if self.feature.ownerScope = #instance
   then self.argument->size > 1 implies
         self.feature.oclIsKindOf(Operation)
    else self.argument->size > 0 implies
          self.feature.oclIsKindOf(Operation)
endif
```

[C-5-2] If the FeatureNode represents an instance-scope feature, the first argument is a “this” or “self” argument; that is, the object invoking the feature. The convention is enforced by checking that the type of the first argument is the same as the type of the feature.

**context** FeatureNode **inv:**

```
self.feature.ownerScope = #instance implies
self.argument->first.type.allFeatures->includes(self.feature)
```

[C-5-3] If the FeatureNode represents a BehavioralFeature, the number of arguments must be equal to the number of the BehavioralFeature’s parameters, plus one for the “this” parameter if the BehavioralFeature is of instance scope.

**context** FeatureNode **inv:**

```
self.feature.oclIsKindOf(BehavioralFeature) implies
(if self.feature.ownerScope = #instance
     then self.argument->size =
        self.feature.oclAsType(BehavioralFeature).parameters->size + 1
     else self.argument->size =
        self.feature.oclAsType(BehavioralFeature).parameters->size
endif)
```

[C-5-4] If the FeatureNode represents a BehavioralFeature, the types of the arguments must match, in order, the types of the parameters, allowing for the optional presence of a leading “this” parameter.

**context** FeatureNode **inv:**

```
self.feature.oclIsKindOf(BehavioralFeature) implies
(if self.feature.ownerScope = #instance
    then self.argument->forAll(arg : Integer  |
            self.argument->at(arg + 1)
                .allSuperTypes.union(self.argument.oclType)->
                includes(self.feature.oclAsType(BehavioralFeature)
                .parameters->at(arg))
    else self.argument->forAll(arg : Integer  |
            self.argument->at(arg)
               .allSuperTypes.union(self.argument.oclType)->
               includes(self.feature.oclAsType(BehavioralFeature)
               .parameters->at(arg))
endif)
```

## 8.6 KeysIndexes Metamodel

The KeysIndexes package depends on the following package:

- org.omg::CWM::ObjectModel::Core

Keys and indexes as means for specifying instances and for identifying alternate sortings of instances are represented in the CWMFoundation so that they can be shared among the various data models that employ them. The CWM Foundation defines the base concepts (uniqueness and relationships implemented as keys) upon which more specific key structures can be built by other CWM and tool-specific packages.

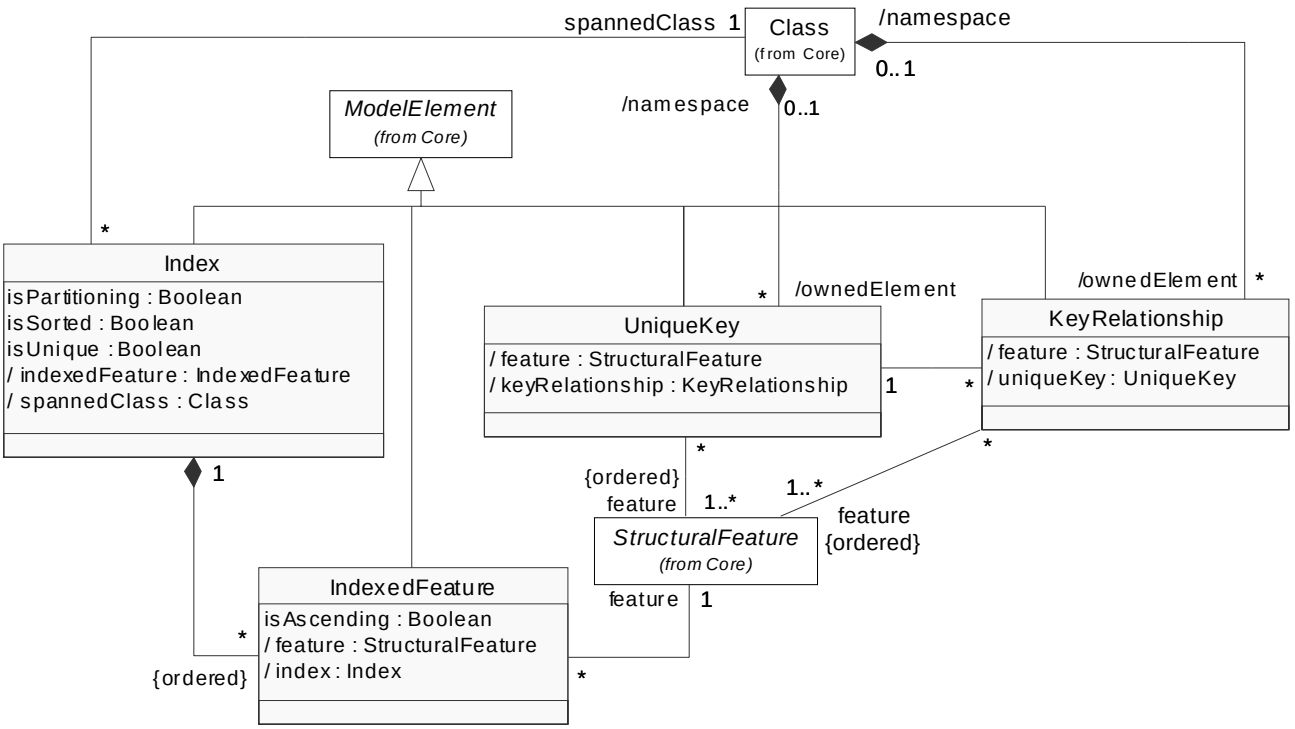


*Figure 8-6-1* KeysIndexes metamodel.

The concepts of key and index have been placed in the CWM Foundation because they are available in many types of data resources. In the CWM Foundation class and association descriptions that follow, relational model examples are frequently used when discussing the definition and usage of key and index types. This is done because of the wide-spread availability of relational systems and is thought to promote an understanding of the underlying concepts. These concepts, however, are no less applicable to other data models as well.

The two central classes for representing the concept of keys are UniqueKey and KeyRelationship. UniqueKey instances correspond to the notion that keys represent the identity of instances -- similar to the relational model's concept of a primary key or an object model's concept of an object identity. In contrast, KeyRelationship instances

correspond to the notion that keys embedded in an instance can be used to determine the identity of other related instances -- similar to the relational model concept of foreign key and the object model concept of a reference. Consequently, UniqueKey and KeyRelationship are best thought of as representing roles that collections of Features of Classifiers play rather than Classifiers describing the internal structure of keys. Representing keys as roles rather than structural entities provides greater flexibility and allows the reuse of Features in multiple keys and in differing relationships to each other. Associations within the KeysIndexes package describe how UniqueKey and KeyRelationship instances describe the roles they play for various Class instances and the StructuralFeature instances they contain.

An example of the usage of Index, KeyRelationship and UniqueKey instances to implement a simple foreign key relationship between stars and the constellations in which they are found can be seen in the following figure. Also, Indexes are used to implement the ordering of constellation and star IDs.

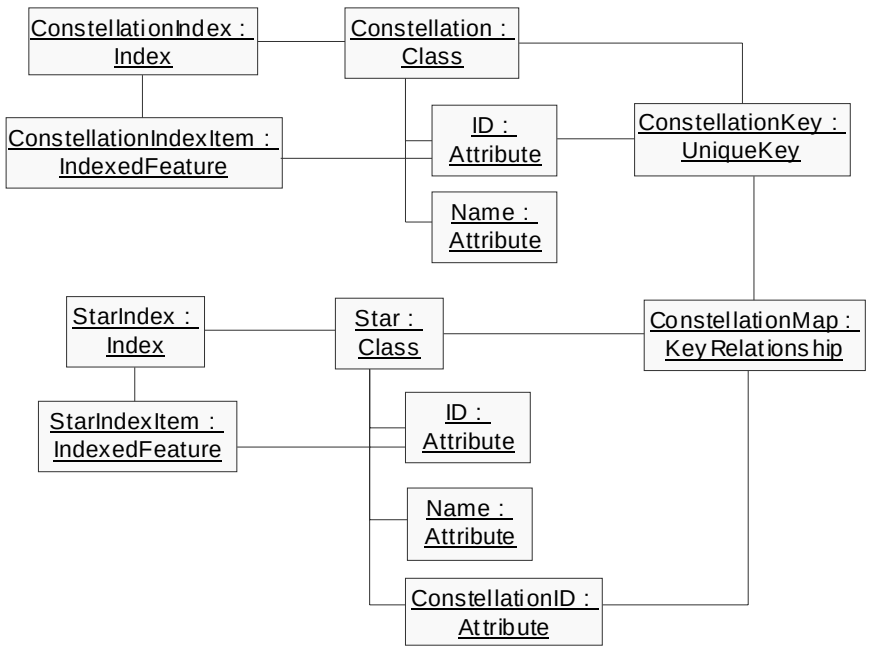


*Figure 8-6-2* Star and constellation keys and index example.

## 8.6.1 KeysIndexes Classes

### 8.6.1.1 Index

Instances of the Index class represent the ordering of the instances of some other Class, and the Index is said to “span” the Class. Indexes normally have an ordered set of attributes of the Class instance they span that make up the “key” of the index; this set

of relationships is represented by the IndexedFeature class that indicates how the attributes are used by the Index instance.

The Index class is intended primarily as a starting point for tools that require the notion of an index.

## Superclasses

ModelElement

## Contained Elements

IndexedFeature

## Attributes

### *isUnique*

The isUnique attribute is True if the Index instance guarantees all of its instances have a unique key value.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

### *isSorted*

If True, the Index instance is maintained in a sorted order.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

### *isPartitioning*

If True, this Index instance is used as a partitioning index.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

## References

***indexedFeature***

Identifies the IndexedFeature instance that associates this Index with one of the StructuralFeature elements of the Index's key. The ordered constraint on this reference can be used to represent the sequential order of elements of the Index's key.

| | |
|---|---|
| *class:* | IndexedFeature |
| *defined by:* | IndexedFeatureInfo::indexedFeature |
| *multiplicity:* | one or more; ordered |
| *inverse:* | IndexedFeature::index |

***spannedClass***

Identifies the Class instance spanned by the Index instance.

| | |
|---|---|
| *class:* | Class |
| *defined by:* | IndexSpansClass::spannedClass |
| *multiplicity:* | exactly one |

### *8.6.1.2 IndexedFeature*

Instances of the IndexedFeature class map StructuralFeature instances of the spanned Class instance to the Index instances that employ them as (part of) their key. Attributes of IndexedFeature instances indicate how specific StructuralFeature instance are used in Index keys.

#### *Superclasses*

ModelElement

#### *Attributes*

***isAscending***

The isAscending attribute is ***true*** if the feature is sorted in ascending order and ***false***, if descending order.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | Zero or one |

#### *References*

***index***

Identifies the Index instance for which this IndexedFeature instance is relevant.

| | |
|---|---|
| *class:* | Index |
| *defined by:* | IndexedFeatureInfo::index |
| *multiplicity:* | exactly one |
| *inverse:* | Index::indexedFeature |

***feature***

Identifies the StructuralFeature instance for which this IndexedFeature instance is relevant.

| | |
|---|---|
| *class:* | StructuralFeature |
| *defined by:* | IndexedFeatures::feature |
| *multiplicity:* | exactly one |

### *Constraints*

The ***isAscending*** attribute is valid only if the ***isSorted*** attribute is True. [C-6-1]

## *8.6.1.3 KeyRelationship*

KeyRelationship instances represent relationships between UniqueKey instances and the Class(es) that reference them. This class is intended as a starting point for the creation of “foreign key” and other associative relationships.

### *Superclasses*

ModelElement

### *References*

***feature***

Identifies StructuralFeature instances that participate as (part of) the key of this KeyRelationship instance.

| | |
|---|---|
| *class:* | StructuralFeature |
| *defined by:* | KeyRelationshipFeatures::feature |
| *multiplicity:* | one or more; ordered |

***uniqueKey***

Identifies the UniqueKey instance that serves as the “primary key” for this KeyRelationship instance.

| | |
|---|---|
| *class:* | UniqueKey |
| *defined by:* | UniqueKeyRelationship::uniqueKey |
| *multiplicity:* | exactly one |
| *inverse:* | UniqueKey::keyRelationship |

### *Constraints*

A KeyRelationship instance must be owned by one and only one Class instance [C-6-2].

## *8.6.1.4 UniqueKey*

A UniqueKey represents a collection of features of some Class that, taken together, uniquely identify instances of the Class. Instances of UniqueKey for which all features are required to have non-null values are candidates for use as primary keys such as those defined by relational DBMSs.

### *Superclasses*

ModelElement

### *References*

***feature***

Identifies the StructuralFeature instances that make up the unique key. The ordered constraint is used to represent the sequence of StructuralFeature instances that make up the UniqueKey instance’s key.

| | |
|---|---|
| *class:* | StructuralFeature |
| *defined by:* | UniqueFeature::feature |
| *multiplicity:* | one or more; ordered |

***keyRelationship***

Identifies the KeyRelationship instances that reference this UniqueKey instance.

| | |
|---|---|
| *class:* | KeyRelationship |

| | |
|---|---|
| *defined by:* | UniqueKeyRelationship::keyRelationship |
| *multiplicity:* | zero or more |
| *inverse:* | KeyRelationship::uniqueKey |

### *Constraints*

An UniqueKey instance must be owned by one and only one Class instance. [C-6-3]

## *8.6.2 KeysIndexes Associations*

### *8.6.2.1 IndexedFeatureInfo* *Protected*

The IndexedFeatureInfo association connects an Index instance to information about how the StructuralFeature instances that are constituents of the Index's key are used by the Index.

### *Ends*

***index***

Identifies the Index instance for which this IndexedFeature instance is relevant.

| | |
|---|---|
| *class:* | Index |
| *multiplicity:* | exactly one |

***indexedFeature***

Identifies the IndexedFeature instance that associates this Index with one of the StructuralFeature elements of the Index's key. The ordered constraint on this reference can be used to represent the sequential order of elements of the Index's key.

| | |
|---|---|
| *class:* | IndexedFeature |
| *multiplicity:* | zero or more; ordered |
| *aggregation:* | composite |

### *8.6.2.2 IndexedFeatures*

The IndexedFeatures association links StructuralFeature instances to information about how they participate in the keys of Index instances.

### *Ends*

***feature***

Identifies the StructuralFeature instance for which this IndexedFeature instance is relevant.

| | |
|---|---|
| *class:* | StructuralFeature |
| *multiplicity:* | exactly one |

***indexedFeature***

Identifies the IndexedFeature instances that describe how a particular StructuralFeature is used by the keys of Index instances.

| | |
|---|---|
| *class:* | IndexedFeature |
| *multiplicity:* | zero or more |

### *8.6.2.3 IndexSpansClass*

Associates indexes with the classes they span. This relationship is separate from the ownership of indexes, to allow modeling of systems where an index is NOT owned by the object it spans. In most situations, however, the spanned and owning Class instances will be the same.

*Ends*

***index***

Identifies Index instances that span this Class instance.

| | |
|---|---|
| *class:* | Index |
| *multiplicity:* | zero or more |

***spannedClass***

Identifies the Class instance the Index instance spans.

| | |
|---|---|
| *class:* | Class |
| *multiplicity:* | exactly one |

### *8.6.2.4 KeyRelationshipFeatures*

The KeyRelationshipFeatures association links KeyRelationship instances with the StructuralFeature instances that comprise their key.

*Ends*

***feature***

Identifies StructuralFeature instances that participate as (part of) the key of this KeyRelationship instance. In the relational case, this reference indicates the columns that make up the foreign key.

| | |
|---|---|
| *class:* | StructuralFeature |
| *multiplicity:* | one or more; ordered |

***keyRelationship***

Identifies the KeyRelationship instances that employ a particular StructuralFeature as part of their key.

| | |
|---|---|
| *class:* | KeyRelationship |
| *multiplicity:* | zero or more |

### *8.6.2.5 UniqueFeature*

The UniqueFeature association identifies the Feature instances of a Class instance that confer uniqueness. The ordered constraint is used to determine the order of StructuralFeature instances in the UniqueKey instance.

#### *Ends*

***feature***

Identifies the StructuralFeature instances that make up the unique key. The ordered constraint is used to represent the sequence of StructuralFeature instances that make up the UniqueKey instance's key. In the relational model case, these StructuralFeature instances identify the columns that make up a table's primary key.

| | |
|---|---|
| *class:* | StructuralFeature |
| *multiplicity:* | one or more; ordered |

***uniqueKey***

Identifies the UniqueKey instances in which a particular StructuralFeature participates.

| | |
|---|---|
| *class:* | UniqueKey |
| *multiplicity:* | zero or more |

### 8.6.2.6 UniqueKeyRelationship *Protected*

The UniqueKeyRelationship association links a KeyRelationship with the UniqueKey with which it is paired. For example, in relational model terms, this association links a foreign key -- the KeyRelationship instance -- with the primary key -- the UniqueKey instance -- with which it is paired.

*Ends*

***keyRelationship***

Identifies the KeyRelationship instances with which a particular UniqueKey instance is paired.

| | |
|---|---|
| *class:* | KeyRelationship |
| *multiplicity:* | zero or more |

***uniqueKey***

Identifies the KeyRelationship instances that reference this UniqueKey instance. In the relational case, this reference identifies the foreign keys that reference this primary key.

| | |
|---|---|
| *class:* | UniqueKey |
| *multiplicity:* | exactly one |

## 8.6.3 OCL Representation of KeysIndexes Constraints

[C-6-1]The ***isAscending*** attribute is valid only if the ***isSorted*** attribute is True.

**context** IndexedFeature **inv:**

self.isAscending->notEmpty **implies** self.index.isSorted

[C-6-2] A KeyRelationship instance must be owned by one and only one Class instance.

**context** KeyRelationship **inv:**

(self.namespace->size = 1) and self.namespace.oclIsKindOf(Class)

[C-6-3] An UniqueKey instance must be owned by one and only one Class instance.

**context** UniqueKey **inv:**

(self.namespace->size = 1) and self.namespace.oclIsKindOf(Class)

## 8.7 SoftwareDeployment Metamodel

The Software Deployment package depends on the following packages:

- org.omg::CWM::ObjectModel::Core
- org.omg::CWM::Foundation::BusinessInformation
- org.omg::CWM::Foundation::TypeMapping

The Software Deployment package contains classes to record how the software in a data warehouse is used.

A software package is represented as a SoftwareSystem object, which is a subtype of Subsystem. A SoftwareSystem may reference one or more TypeSystems that define the datatypes supported by the SoftwareSystem. The mappings between datatypes in different TypeSystems may be recorded as TypeMappings, as described in section 8.8, TypeMapping metamodel.

The separate components of a software package are each represented as Components that are either owned or imported by the SoftwareSystem. When a SoftwareSystem is installed, the deployment is recorded as a DeployedSoftwareSystem and a set of DeployedComponents.

A DeployedComponent represents the deployment of a specific Component on a specific computer. Dependencies between DeployedComponents on the same computer may be documented as Usage dependencies between them.

Individual computers are represented as Machine objects, located at a Site. A Site represents a geographical location, which may be recorded at any relevant level of granularity, e.g. a country, a building, or a room in a building. Hierarchical links between Sites at different levels of granularity may be documented.

A DataManager is a DeployedComponent such as a DBMS or file management system that provides access to data. It may be associated with one or more data Packages identifying the Schema, Relational Catalog, Files or other data containers that it provides access to.

A DataProvider is a DeployedComponent that acts as a client to provide access to data held by a DataManager. For example, an ODBC or JDBC client on a specific Machine would be represented as a DataProvider. A DataProvider may have several ProviderConnections, each identifying a DataManager that may be accessed using the DataProvider.

If a DataProvider uses a name for a data Package that is different from the actual name used by the DataManager, a PackageUsage object can be added to record this.

As a DataProvider is a subtype of DataManager, it is possible for a DataProvider to access data from a DataManager which is actually a DataProvider acting as a client to yet another DataManager.

The model for the Software Deployment package is shown in three diagrams. The first diagram shows the objects related to software deployment, while the second diagram displays the DataManager and DataProvider area of the model. The third diagram shows the inheritance structure for all the classes in the Software Deployment package.

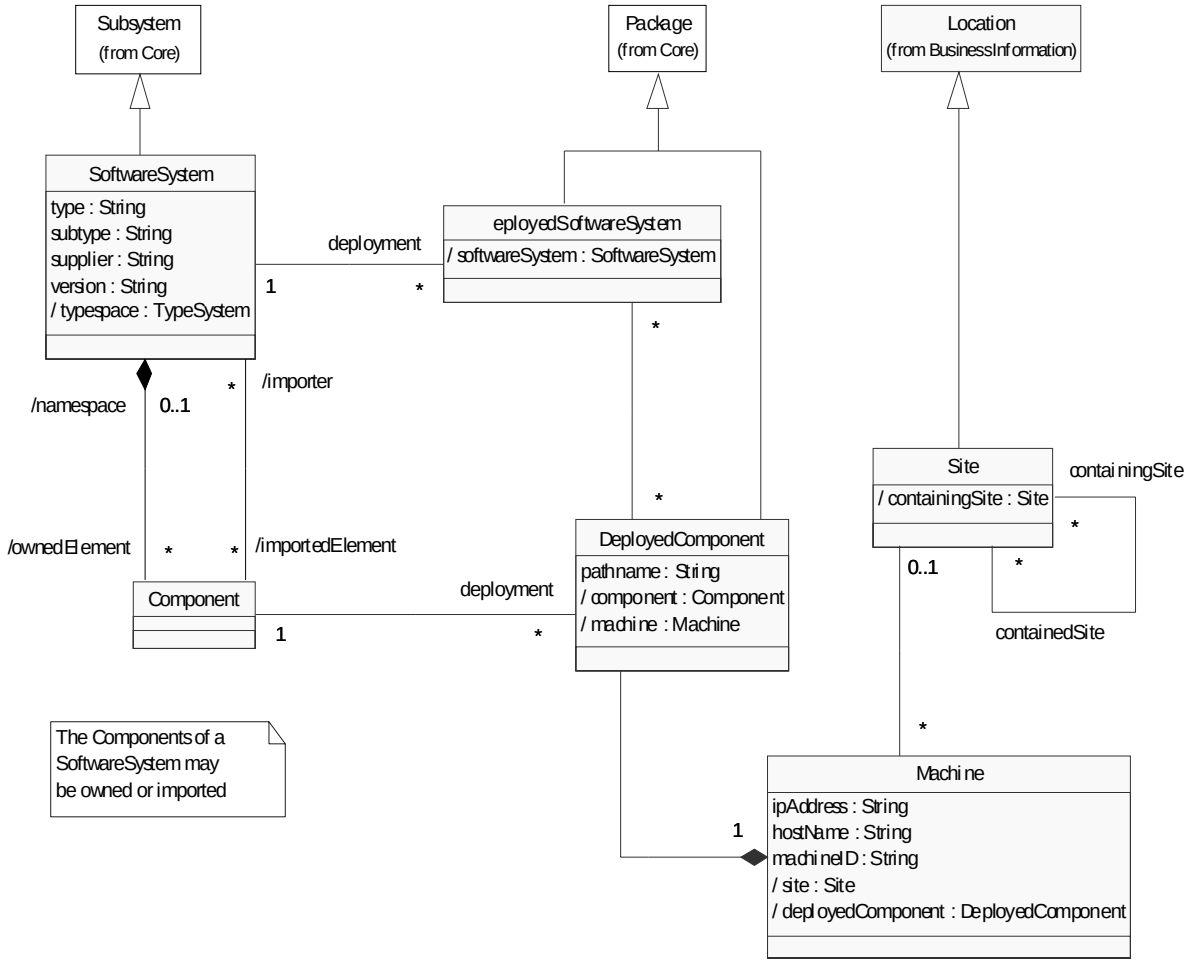


*Figure 8-7-1* Software Deployment

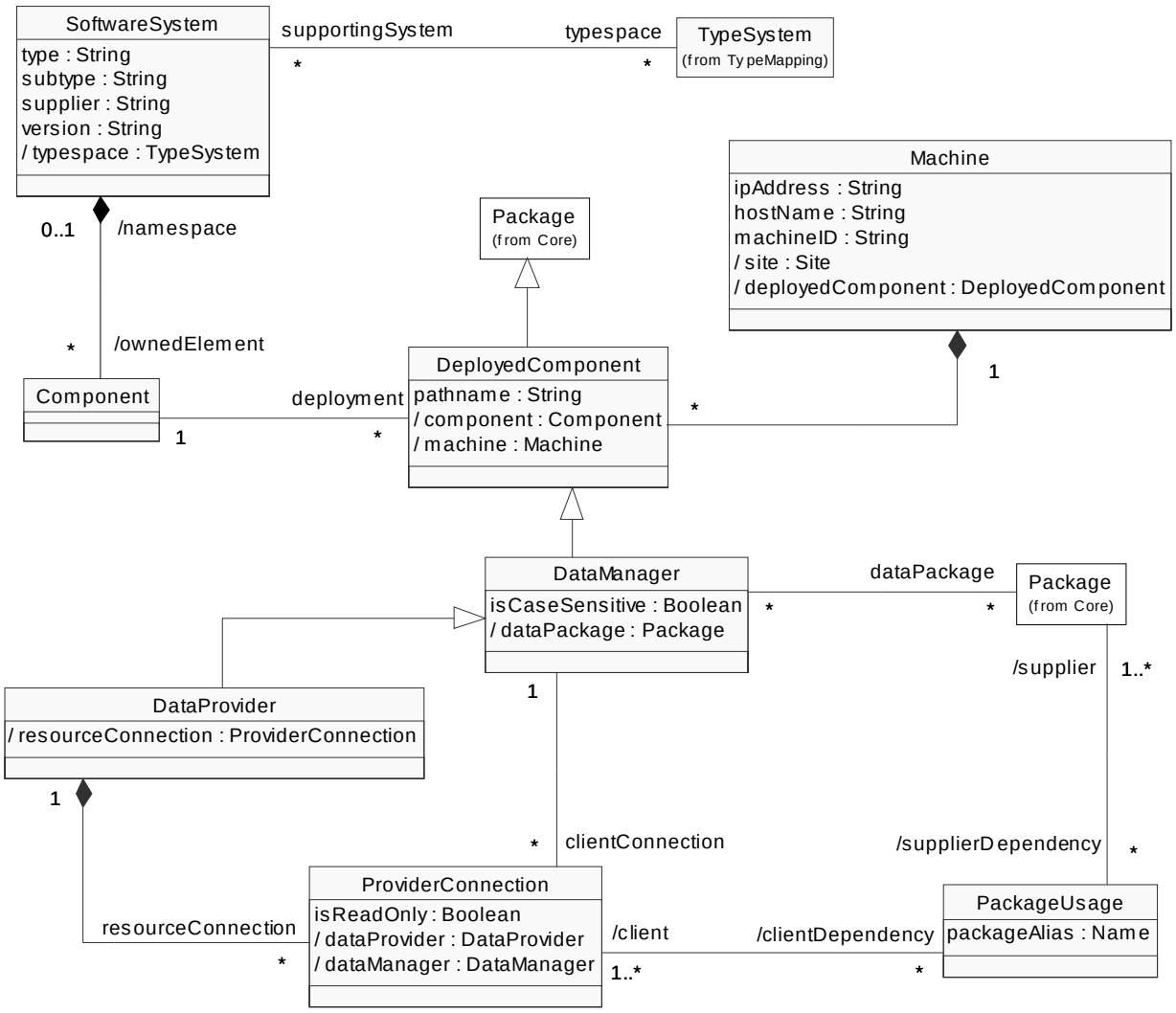


*Figure 8-7-2* DataManager and DataProvider

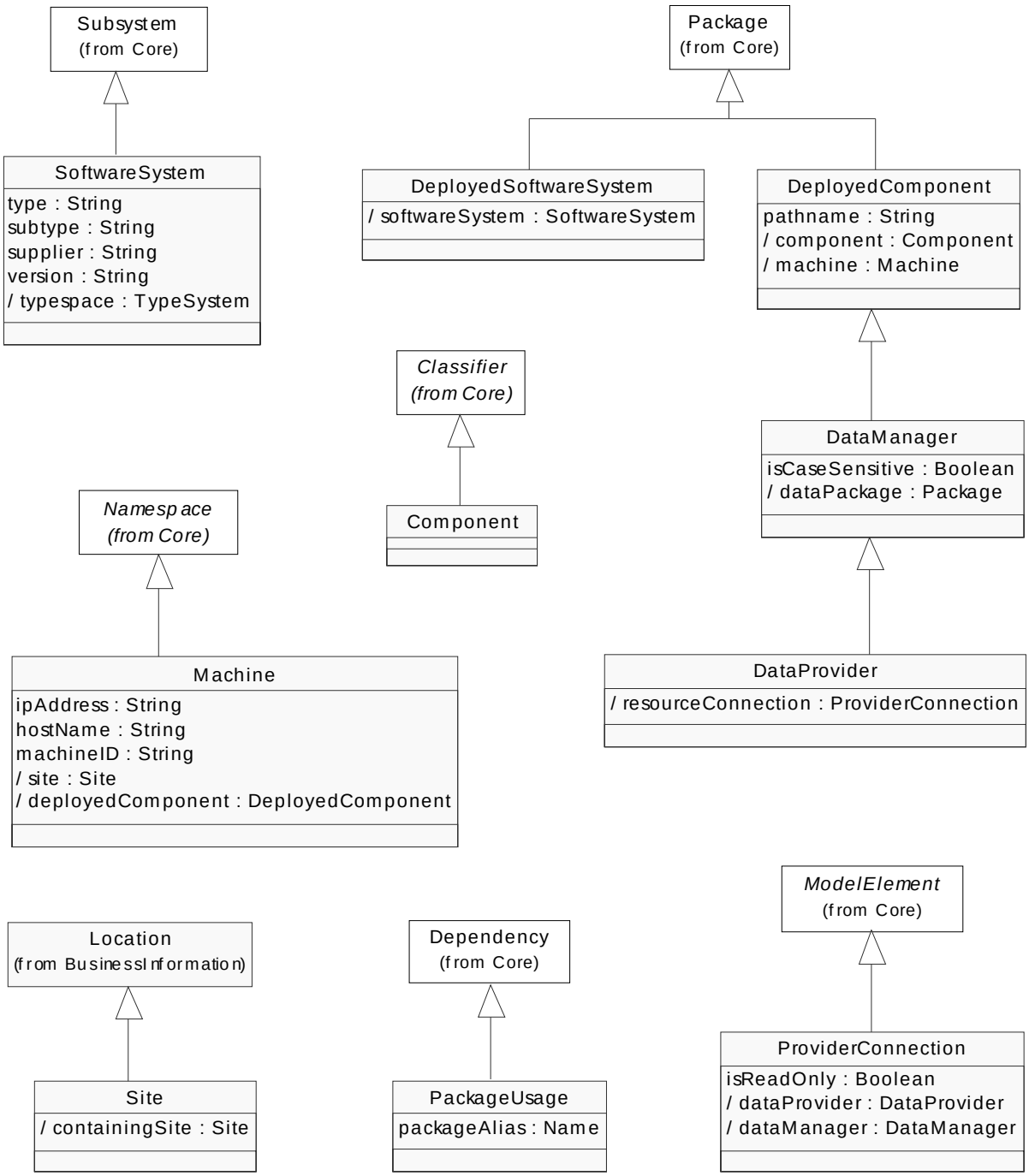


*Figure 8-7-3* Software Deployment Inheritances

## 8.7.1 SoftwareDeployment Classes

### 8.7.1.1 Component

A Component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files. A Component is a subtype of Classifier, and so may have its own Features, such as Attributes and Operations.

Deployment of a Component on a specific Machine is represented as a DeployedComponent.

#### *Superclasses*

Classifier

### 8.7.1.2 DataManager

A DataManager represents a DeployedComponent that manages access to data. For example, a deployed DBMS or File Management System would be represented as a DataManager.

The DataManager may be associated with one or more data Packages identifying the Schema, Relational Catalog, Files, or other data container that it provides access to.

#### *Superclasses*

DeployedComponent

#### *Attributes*

***isCaseSensitive***

Indicates whether or not the DataManager treats lower case letters within object names as being different from the corresponding upper case letters.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

#### *References*

***dataPackage***

Identifies the Package(s) containing the definition of the data made available by the DataManager.

| | |
|---|---|
| *class:* | Package |
| *defined by:* | DataManagerDataPackage::dataPackage |
| *multiplicity:* | zero or more |

### 8.7.1.3 DataProvider

A DataProvider is a deployed software Component that acts as a client to provide access to data that is managed by another product. For instance, a DataProvider might represent a deployed ODBC or JDBC product.

The DataProvider may have resourceConnection references to ProviderConnections identifying the DataManagers to which it provides access.

#### *Superclasses*

DataManager

#### *Contained Elements*

ProviderConnection

#### *References*

***resourceConnection***

Identifies the ProviderConnections that the DataProvider may use to access data resources.

| | |
|---|---|
| *class:* | ProviderConnection |
| *defined by:* | DataProviderConnections::resourceConnection |
| *multiplicity:* | zero or more |
| *inverse:* | ProviderConnection::dataProvider |

### 8.7.1.4 DeployedComponent

A DeployedComponent represents the deployment of a Component on a specific Machine.

It may represent the deployment of any type of Component. However, if the Component is part of a SoftwareSystem, the DeployedComponent should be part of a DeployedSoftwareSystem.

Usage dependencies may be used to document that one DeployedComponent uses another DeployedComponent.

### *Superclasses*

Package

### *Attributes*

***pathname***

A pathname for the DeployedComponent within the Machine's file system.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### *References*

***component***

Identifies the Component deployed.

| | |
|---|---|
| *class:* | Component |
| *defined by:* | ComponentDeployments::component |
| *multiplicity:* | exactly one |

***machine***

Identifies the Machine on which the DeployedComponent is deployed.

| | |
|---|---|
| *class:* | Machine |
| *defined by:* | ComponentsOnMachine::machine |
| *multiplicity:* | exactly one |
| *inverse:* | Machine::deployedComponent |

## *8.7.1.5 DeployedSoftwareSystem*

A DeployedSoftwareSystem represents a deployment of a SoftwareSystem.

Its associated DeployedComponents identify the individual Component deployments that constitute the DeployedSoftwareSystem. These DeployedComponents are not necessarily all deployed on the same Machine.

### *Superclasses*

Package

### *References*

#### ***softwareSystem***

Identifies the SoftwareSystem deployed.

| | |
|---|---|
| *class:* | SoftwareSystem |
| *defined by:* | SoftwareSystemDeployments::softwareSystem |
| *multiplicity:* | exactly one |

## *8.7.1.6 Machine*

A Machine represents a computer. The Site at which the Machine is located and the Components deployed on the Machine may be recorded.

### *Superclasses*

Namespace

### *Contained Elements*

DeployedComponent

### *Attributes*

#### ***ipAddress***

A fixed IP address for the Machine.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | zero or more; ordered |

#### ***hostName***

A Host Name for the Machine. This may be used to identify the Machine on the network when IP addresses are dynamically allocated.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | zero or more; ordered |

***machineID***

An identification code for the Machine.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | zero or one |

### *References*

***site***

Identifies the Site at which the Machine is located.

| | |
|---|---|
| *class:* | Site |
| *defined by:* | SiteMachines::site |
| *multiplicity:* | zero or one |

***deployedComponent***

Identifies the DeployedComponents on the Machine.

| | |
|---|---|
| *class:* | DeployedComponent |
| *defined by:* | ComponentsOnMachine::deployedComponent |
| *multiplicity:* | zero or more |
| *inverse:* | DeployedComponent::machine |

## *8.7.1.7 PackageUsage*

A PackageUsage represents a usage of a Package. It is particularly relevant in situations where a specific usage uses an alternative name for the Package, as this alternative name can be recorded using the packageAlias attribute.

For example, if a DataProvider representing an ODBC or JDBC client uses a name for a relational database that is different from the dataPackage name used by the RDBMS server, a PackageUsage that has the relevant ProviderConnection as client and the server's data Package as supplier can be added. Its packageAlias attribute can be used to record the name by which the data Package is known to the DataProvider.

### *Superclasses*

Dependency

### *Attributes*

***packageAlias***

If this attribute is present, it identifies the name by which the Package is known to the client.

| | |
|---|---|
| *type:* | Name |
| *multiplicity:* | zero or one |

### *Constraints*

A PackageUsage must have a single Package (or subtype of Package) as its supplier. [C-8-1]

## *8.7.1.8 ProviderConnection*

A ProviderConnection represents a connection that allows a DataProvider acting as a client to access data from a specific DataManager. For example a ProviderConnection could represent a connection from an ODBC or JDBC client to a DBMS.

### *Superclasses*

ModelElement

### *Attributes*

***isReadOnly***

Indicates whether the ProviderConnection only allows read access to the DataManager.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

### *References*

***dataProvider***

Identifies the DataProvider that is the client of the ProviderConnection.

| | |
|---|---|
| *class:* | DataProvider |
| *defined by:* | DataProviderConnections::dataProvider |
| *multiplicity:* | exactly one |
| *inverse:* | DataProvider::resourceConnection |

***dataManager***

Identifies the DataManager that is accessed by the ProviderConnection.

| | |
|---|---|
| *class:* | DataManager |
| *defined by:* | DataManagerConnections::dataManager |
| *multiplicity:* | exactly one |

### *Constraints*

A ProviderConnection must not associate a DataProvider with itself. [C-8-2]

## *8.7.1.9 Site*

A Site represents a geographical location. It provides a grouping mechanism for a group of machines at the same location.

Sites may be documented at different levels of granularity; containment links may be used to record hierarchical relationships between Sites.

### *Superclasses*

Location

### *References*

***containingSite***

Identifies a Site of which the current Site forms a part.

| | |
|---|---|
| *class:* | Site |
| *defined by:* | RelatedSites::containingSite |
| *multiplicity:* | zero or more |

### *Constraints*

A Site must not have a containingSite reference that refers to itself. [C-8-3]

## *8.7.1.10 SoftwareSystem*

A SoftwareSystem represents a specific release of a software product. It consists of a set of software Components.

### *Superclasses*

Subsystem

## Contained Elements

Component

## Attributes

### *type*

Identifies the type of the software product. One of the following predefined values should be used if appropriate:
OS, DBMS, MDDB, FileSystem, ODBC, JDBC or Application.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | zero or one |

### *subtype*

This is used in conjunction with the type attribute to provide additional information about the type of the software product.
For some of the predefined types, suggested subtype values are listed below:
For an Operating System product (type OS):
AIX, Linux, MVS, NT, Solaris, SunOS, VMS or Windows.
For a Database Management System product (type DBMS):
DB2, DMS II, IMS, Informix, Oracle, SQLServer or Sybase.
For a Multidimensional Database product (type MDDB):
Essbase or Express.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | zero or one |

### *supplier*

The supplier of the software product.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### *version*

The version identification for the software product.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

*References*

***typespace***

Identifies the TypeSystem(s) containing the datatypes supported by the SoftwareSystem.

| | |
|---|---|
| *class:* | TypeSystem |
| *defined by:* | SystemTypespace::typespace |
| *multiplicity:* | zero or more |

## *8.7.2 SoftwareDeployment Associations*

### *8.7.2.1 ComponentDeployments*

This association identifies the deployments of a Component.

*Ends*

***component***

Identifies the Component deployed.

| | |
|---|---|
| *class:* | Component |
| *multiplicity:* | exactly one |

***deployment***

Identifies the DeployedComponent.

| | |
|---|---|
| *class:* | DeployedComponent |
| *multiplicity:* | zero or more |

### *8.7.2.2 ComponentsOnMachine* *Protected*

Identifies the Machine on which a DeployedComponent is deployed.

*Ends*

***deployedComponent***

Identifies the DeployedComponents on the Machine.

| | |
|---|---|
| *class:* | DeployedComponent |
| *multiplicity:* | zero or more |

***machine***

Identifies the Machine on which a DeployedComponent is deployed.

| | |
|---|---|
| *class:* | Machine |
| *multiplicity:* | exactly one |
| *aggregation:* | composite |

### 8.7.2.3 DataManagerConnections

Identifies the DataManager that is accessed by a ProviderConnection.

*Ends*

***DataManager***

Identifies the DataManager accessed by the ProviderConnection.

| | |
|---|---|
| *class:* | DataManager |
| *multiplicity:* | exactly one |

***clientConnection***

Identifies the ProviderConnections that may be used by clients to access the data provided by this DataManager.

| | |
|---|---|
| *class:* | ProviderConnection |
| *multiplicity:* | zero or more |

### 8.7.2.4 DataManagerDataPackage

This associates the Package(s) containing the definition of the data with the DataManager that is used to access it.

For example, it may be used to associate a Schema, Relational Catalog or File with the DataManager that manages access to it.

*Ends*

***dataPackage***

Identifies a Package containing the definition of the data made available by the DataManager.

| | |
|---|---|
| *class:* | Package |
| *multiplicity:* | zero or more |

***dataManager***

Identifies a DataManager that provides access to the data defined in the Package.

| | |
|---|---|
| *class:* | DataManager |
| *multiplicity:* | zero or more |

### *8.7.2.5 DataProviderConnections* *Protected*

Identifies the ProviderConnections that a DataProvider acting as a client may use.

*Ends*

***dataProvider***

Identifies the DataProvider that uses the ProviderConnection.

| | |
|---|---|
| *class:* | DataProvider |
| *multiplicity:* | exactly one |
| *aggregation:* | composite |

***resourceConnection***

Identifies the ProviderConnections that the DataProvider may use to access DataManagers.

| | |
|---|---|
| *class:* | ProviderConnection |
| *multiplicity:* | zero or more |

### 8.7.2.6 DeployedSoftwareSystemComponents

This association identifies the DeployedComponents that constitute a DeployedSoftwareSystem.

#### Ends

***deployedSoftwareSystem***

Identifies the DeployedSoftwareSystem.

| | |
|---|---|
| *class:* | DeployedSoftwareSystem |
| *multiplicity:* | zero or more |

***deployedComponent***

Identifies the DeployedComponent.

| | |
|---|---|
| *class:* | DeployedComponent |
| *multiplicity:* | zero or more |

### 8.7.2.7 RelatedSites

This may be used to record hierarchical relationships between Sites.

#### Ends

***containingSite***

Identifies other Sites of which the current Site forms a part.

| | |
|---|---|
| *class:* | Site |
| *multiplicity:* | zero or more |

***containedSite***

Identifies other Sites that are part of the current Site.

| | |
|---|---|
| *class:* | Site |
| *multiplicity:* | zero or more |

### 8.7.2.8 SiteMachines

Identifies the Site on which a Machine is located.

#### Ends

***site***

Identifies the Site on which the Machine is located.

| | |
|---|---|
| *class:* | Site |
| *multiplicity:* | zero or one |

***machine***

Identifies the Machines located at the Site.

| | |
|---|---|
| *class:* | Machine |
| *multiplicity:* | zero or more |

### 8.7.2.9 SoftwareSystemDeployments

Identifies the deployments of a SoftwareSystem.

#### Ends

***softwareSystem***

Identifies the SoftwareSystem deployed.

| | |
|---|---|
| *class:* | SoftwareSystem |
| *multiplicity:* | exactly one |

***deployment***

Identifies the deployments of the SoftwareSystem.

| | |
|---|---|
| *class:* | DeployedSoftwareSystem |
| *multiplicity:* | zero or more |

#### 8.7.2.10 SystemTypespace

A SoftwareSystem's typespace identifies the TypeSystem(s) containing the datatypes supported by the SoftwareSystem.

*Ends*

***supportingSystem***

Identifies a SoftwareSystem that supports the datatypes defined by the TypeSystem.

| | |
|---|---|
| *class:* | SoftwareSystem |
| *multiplicity:* | zero or more |

***typespace***

Identifies a TypeSystem containing datatypes supported by the SoftwareSystem.

| | |
|---|---|
| *class:* | TypeSystem |
| *multiplicity:* | zero or more |

### 8.7.3 OCL Representation of SoftwareDeployment Constraints

[C-8-1] A PackageUsage must have a single Package (or subtype of Package) as its supplier

**context** PackageUsage **inv:**

self.supplier->size=1 and
self.supplier->at(1).oclIsKindOf(Package)

[C-8-2] A ProviderConnection must not associate a DataProvider with itself

**context** ProviderConnection **inv:**

self.dataManager <> self.dataProvider

[C-8-3] A Site must not have a containingSite reference that refers to itself.

**context** Site **inv:**

self.containingSite -> forAll (c | c <> self)

## 8.8 TypeMapping Metamodel

The TypeMapping package depends on the following packages:

- org.omg::CWM::ObjectModel::Core

The TypeMapping package supports the mapping of data types between different systems. The purpose of these mappings is to indicate data types in different systems that are sufficiently compatible that data values can be interchanged between them. Multiple mappings are allowed between any pair of types and a means of identifying the preferred mapping is provided.

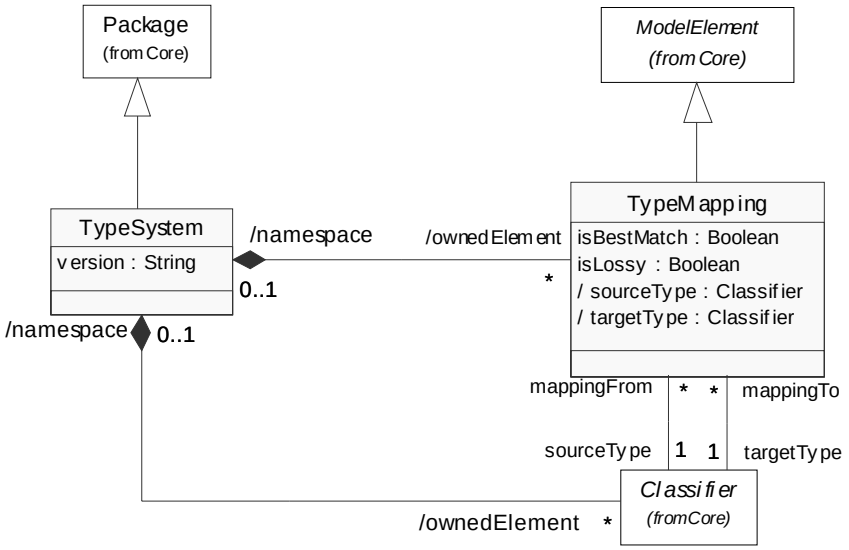


*Figure 8-8-1* TypeMapping metamodel.

The following instance diagram provides a simple example of the use of the TypeMapping package to map the CORBA IDL v2.2 *long* data type and the Java 2 *int* data type to each other.

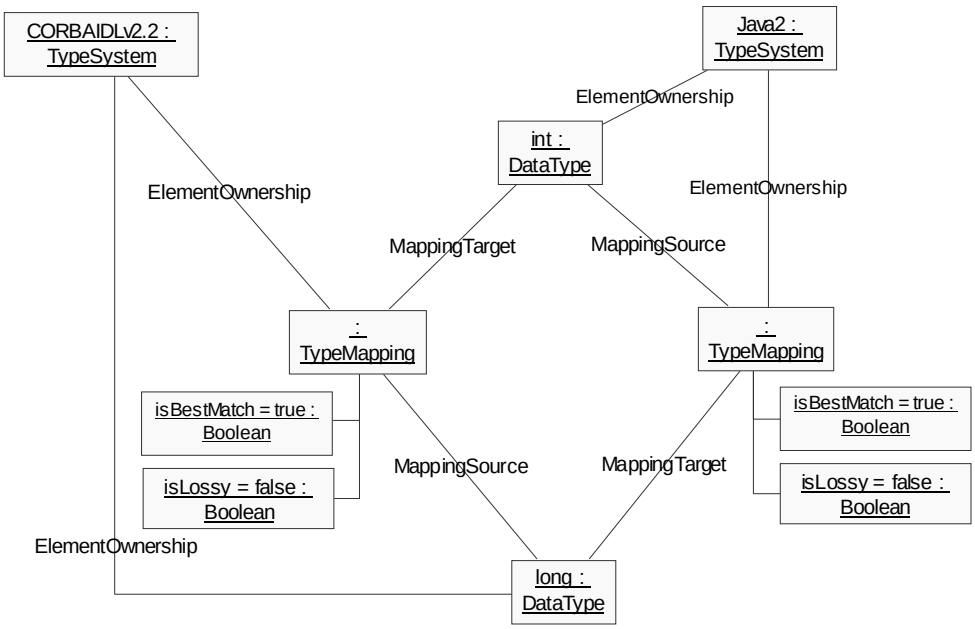


*Figure 8-8-2* Mapping the CORBA IDL *long* and Java *int* data types.

## *8.8.1 TypeMapping Classes*

### *8.8.1.1 TypeMapping*

TypeMapping instances permit the creation of mappings between data types defined within different environments and are used to indicate data type compatibilities that permit direct assignment of values from one environment (the “source” type) into equivalent values in another environment (the “target” type). For example, an integer field data type in a record-oriented DBMS (the source type) might be mapped to a compatible integer data type in a relational DBMS (the target type).

Whereas the actual transfer of data values between environments is accomplished using the CWM’s Transformation package, TypeMapping instances can be used the identify both the *permissible* and *preferred* mappings between data types. Value interchange between a pair of data types is considered permissible if a TypeMapping instance is defined for the pair. A TypeMapping instance is considered the preferred mapping if the instance’s ***isBestMatch*** attribute has the value ***true***.

Typically, there will be one TypeMapping Instance between a pair of data types that is considered the preferred mapping. To promote flexible use of this feature, there is no requirement that a preferred TypeMapping instance must be identified between a pair of data types nor are multiple preferred instances prohibited. In these latter cases, however, the precise semantics are usage-defined.

Interchange between data types defined by non-preferred mappings may often function as intended. However, the ***isLossy*** boolean may be set to indicate that such interchanges may be subject to validity restrictions in certain circumstances. For example, it may be valid to move data values from a 32-bit integer data type to a 16-bit integer data type as long as the actual values in the 32-bit underlying data type do not exceed the range permitted for 16-bit integers. The CWM Foundation leaves the understanding and handling of such differences to individual tools. If such differences must be modeled, consider using the CWM Transformation package to filter data values during interchange.

TypeMapping instances are unidirectional, so two instances are required to show that a data type pair can be mutually interchanged between environments.

## *Superclasses*

ModelElement

## *Attributes*

### *isBestMatch*

True if this TypeMapping instance represents the best available mapping between a pair of data types in different software systems.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

### *isLossy*

True if this TypeMapping instance may result in a data conversion error if the source data is within certain ranges. For example, storing a 32-bit unsigned integer value into a 16-bit unsigned integer container will result in a data conversion error only when the source data has a value greater than 65535.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

## *References*

### *sourceType*

Identifies the Classifier instance that is the source of information exchange.

| | |
|---|---|
| *class:* | Classifier |
| *defined by:* | MappingSource::sourceType |
| *multiplicity:* | exactly one |

***targetType***

Identifies the Classifier instance that is the target of information exchange.

| | |
|---|---|
| *class:* | Classifier |
| *defined by:* | MappingTarget::targetType |
| *multiplicity:* | exactly one |

### *Constraints*

The targetType and sourceType references may not refer to the same Classifier instance. [C-8-1]

## *8.8.1.2 TypeSystem*

Instances of the TypeSystem class collect together the data types (subclasses of Classifier) defined by a software system and the TypeMapping instances defining how they are mapped to data types in other TypeSystems. TypeMapping instances collected by a TypeSystem instance include both those in which the software system's data types act as sources and as targets of mappings. Classifiers and TapeMappings are maintained in a single collection via the ElementOwnership association but can be distinguished by their respective types.

Because it is a Package, a TypeSystem can also serve to collect together the Classifier instances for a particular software system.

### *Superclasses*

Package

### *Contained Elements*

TypeMapping

### *Attributes*

***version***

A string describing the version of the TypeSystem represented.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### *Constraints*

A TypeSystem may own only Classifiers and TypeMappings. [C-8-2]

## 8.8.2 TypeMapping Associations

### 8.8.2.1 MappingSource

The MappingSource association indicates the underlying Classifier instance of a particular TypeMapping.

#### Ends

***sourceType***

Identifies the Classifier instance that is the source of information exchange.

| | |
|---|---|
| *class:* | Classifier |
| *multiplicity:* | exactly one |

***mappingFrom***

Identifies the TypeMapping instances in which a particular Classifier participates.

| | |
|---|---|
| *class:* | TypeMapping |
| *multiplicity:* | zero or more |

### 8.8.2.2 MappingTarget

The MappingTarget association indicates the exposed data type for a particular TypeMapping instance.

#### Ends

***targetType***

Identifies the Classifier instance that is the target of information exchange.

| | |
|---|---|
| *class:* | Classifier |
| *multiplicity:* | exactly one |

***mappingTo***

Identifies the TypeMapping instance of a particular Classifier instance.

| | |
|---|---|
| *class:* | TypeMapping |
| *multiplicity:* | zero or more |

### 8.8.3 OCL Representation of TypeMapping Constraints

[C-8-1] The sourceType and targetType references may not refer to the same Classifier instance.

**context** TypeMapping **inv:**

self.sourceType <> self.targetType

[C-8-2] A TypeSystem may own only Classifiers and TypeMappings.

**context** TypeSystem **inv:**

self.ownedElement->forAll( e | e.oclIsKindOf(Classifier) **or** e.oclIsKindOf(TypeMapping))

# Relational 9

## 9.1 Overview

The Relational package describes data accessible through a relational interface such as a native RDBMS, ODBC, or JDBC. The Relational package is based on the [SQL] standard section concerning RDBMS catalogs.

The scope of the top level container, Catalog, is intended to cover all the tables a user can use in a single statement. A catalog is also the unit which is managed by a data resource. A catalog contains schemas which themselves contain tables. Tables are made of columns which have an associated data type.

The Relational package uses constructs in the ObjectModel package to describe the object extensions added to SQL by the [SQL] standards.

The Relational package also addresses the issues of indexing, primary keys and foreign keys by extending the corresponding concepts from the Foundation packages.

## 9.2 Organization of the Relational package

### 9.2.1 Inheritance

The Relational package depends on the following packages:

- org.omg::CWM::ObjectModel::Behavioral
- org.omg::CWM::ObjectModel::Core
- org.omg::CWM::ObjectModel::Instance
- org.omg::CWM::Foundation::DataTypes
- org.omg::CWM::Foundation::KeysIndexes

The Relational package references the ObjectModel and Foundation packages.

Figure 9-1 shows the Relational package classes and their inheritance from the ObjectModel and Foundation classes. The Relational package, as do the other data packages, define top-level containers (Catalog, Schema) that extend the ObjectModel Package class. ColumnSet and SQLStructuredType extend Class. The Columns contained in the ColumnSet are extensions of the ObjectModel Attribute. The data type of a column (SQLDataType) inherits from ObjectModel Classifier. This structuring of the classes will be particularly useful to describe the object extensions of SQL.

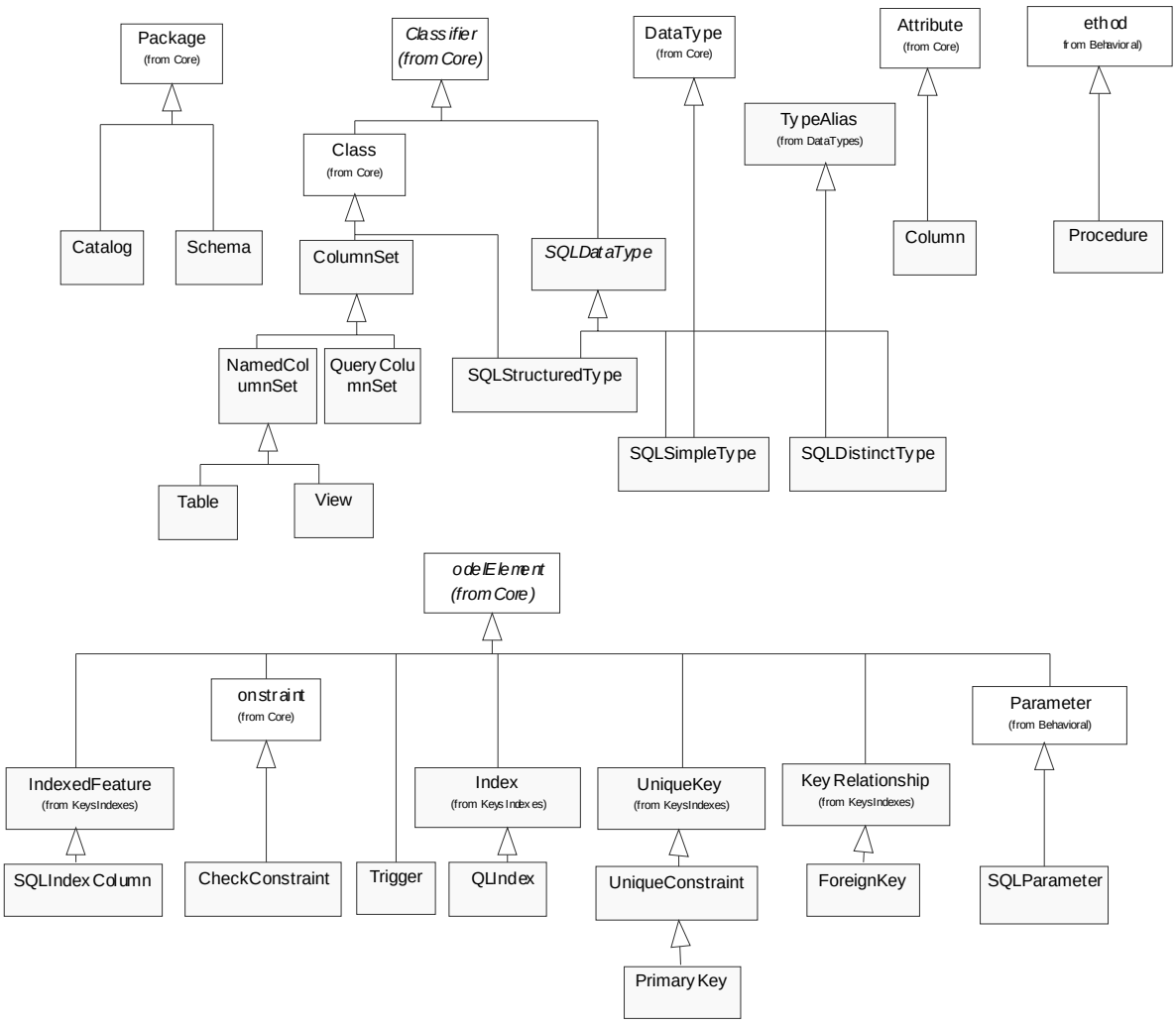


*Figure 9-1* Relational Package Inheritances

### 9.2.2 Containers

In addition to owning Tables and/or Views, Schemas also own Procedures and Triggers.

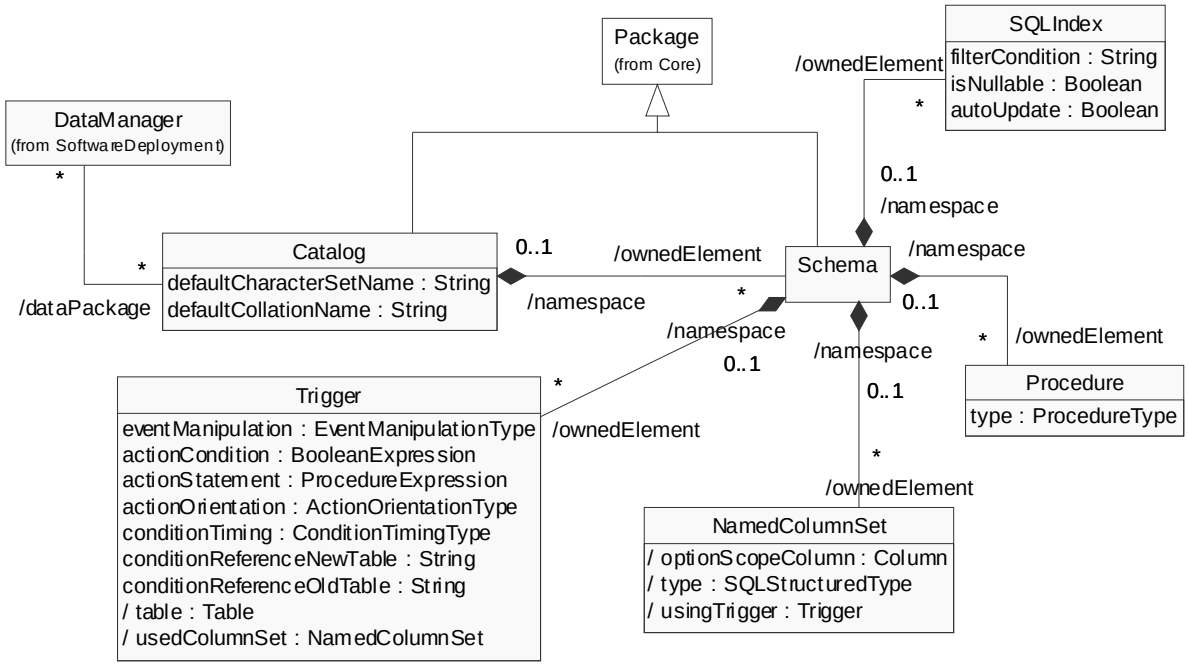


*Figure 9-2* Schemas and owned objects

### 9.2.3 Tables, columns and data types

A ColumnSet represents any form of relational data. A NamedColumnSet is a cataloged version of a ColumnSet, which is owned by a Schema. A NamedColumnSet can be a logical View or a physical Table. Instead of being a NamedColumnSet, a ColumnSet can be a QueryColumnSet, which is the result of an SQL query.

Columns are associated with an SQLDataType, using the type association from StructuralFeature to Classifier inherited from ObjectModel Core.

The following figure shows the original two data types: simple type and distinct type. Simple types are defined by the [SQL] standards, however, some RDBMS implementations use additional types. An SQL distinct type is defined from a simple type.

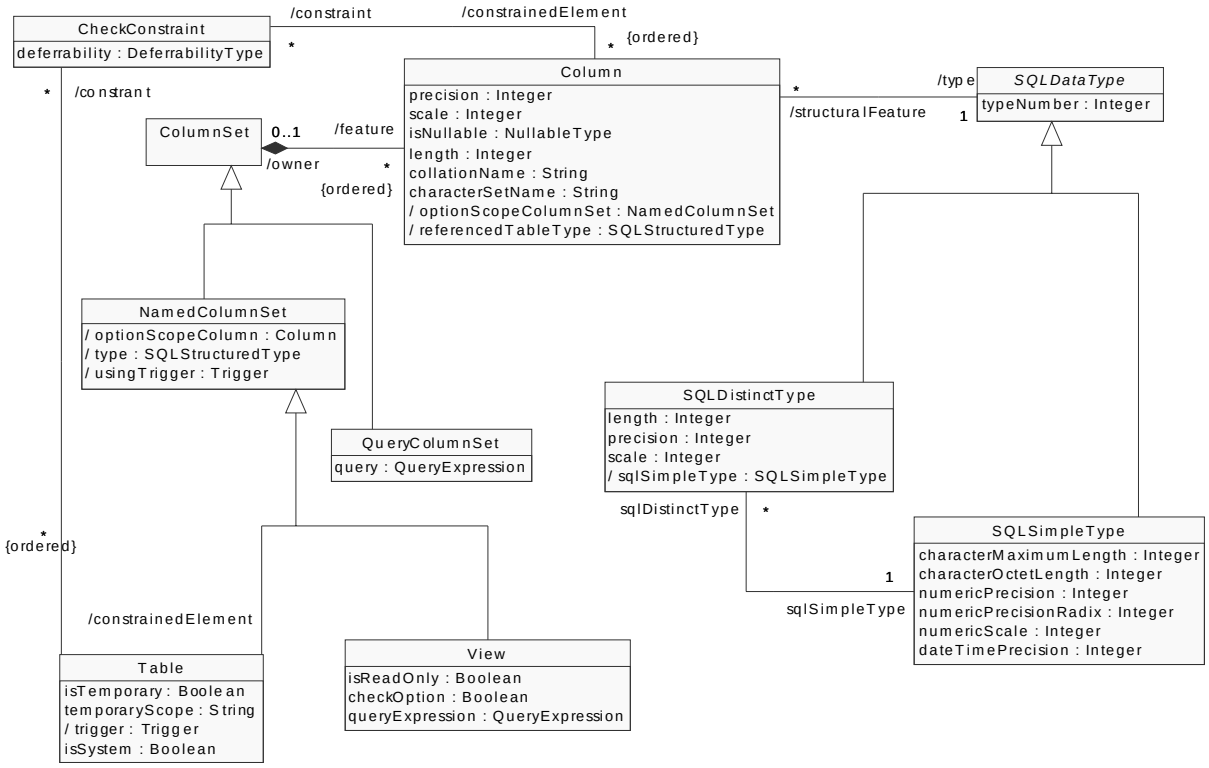


*Figure 9-3* Tables, columns and data types

## *9.2.4 Structured types and object extensions*

The [SQL] standard adds object-oriented notions to SQL with structured types.

A structured type is defined in term of columns, as illustrated in the following example: *CREATE TYPE person_t AS(name varchar(20), birthyear integer).* Since a SQLStructuredType is a Classifier which owns Attributes, it is natural to associate a SQLStructuredType to a set of Columns. Similarly, to represent a type created by CREATE TYPE emp_t UNDER person_t AS(salary integer). we use the ObjectModel Generalization to associate the two types. As a result, the following instances are created to represent the above two examples:

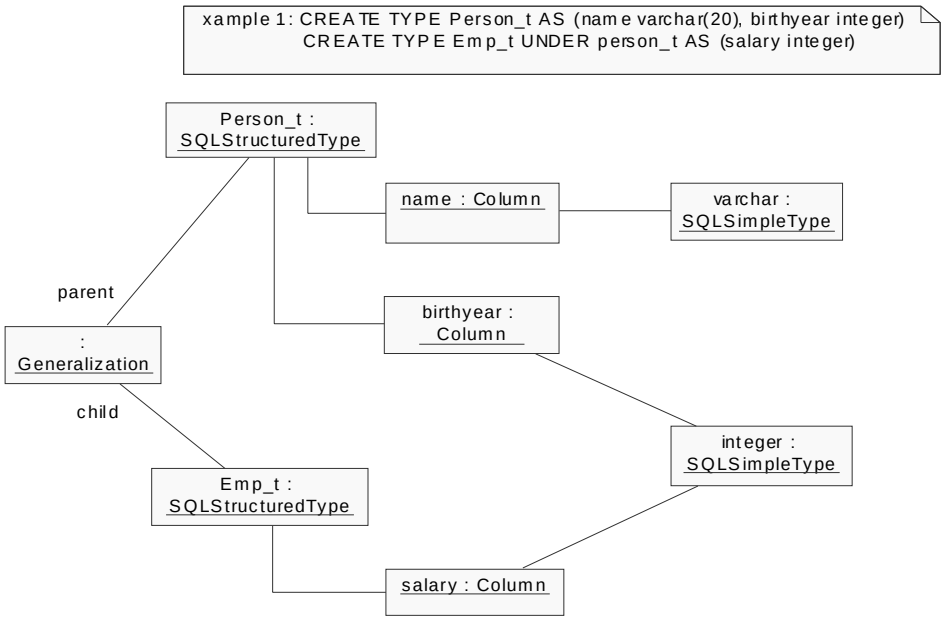


*Figure 9-4* Instance diagram for two structured types

An association between Column and SQLStructuredType (ColumnRefStructuredType) has been added to represent structured type attributes which reference another type, as in *CREATE TYPE dept_t AS (name varchar(40), mgr REF (emp_t)*. This leads to the following instance diagram:

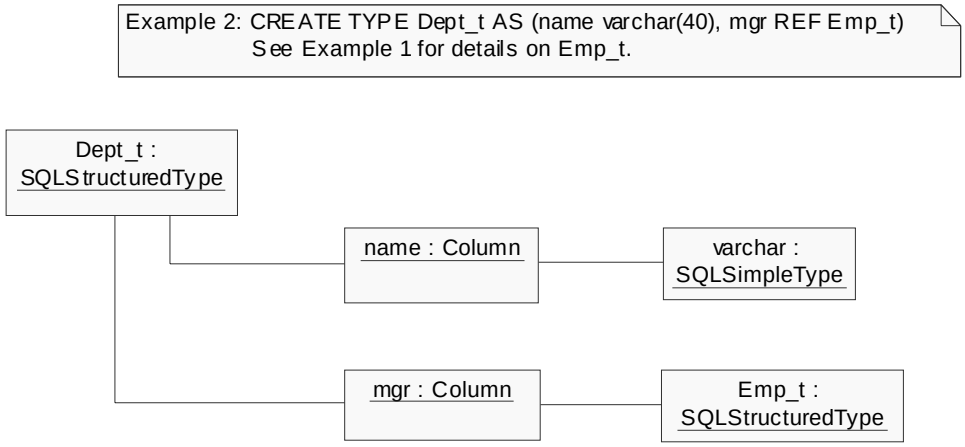


*Figure 9-5* Instance diagram for a structured type containing a REF clause.

A structured type can be used as the data type of a column, but also as a template for a table, as in *CREATE TABLE person OF person_t(ref is oid user generated)* or *CREATE TABLE emp OF emp_t UNDER person.* In these cases, the table will be created with columns which copy the content of the structured type, as described in the [SQL] standard. This allows programs which do not understand the object extensions to still work with the table, both at the data and metadata level. However, an association between the Table (this applies to views as well) and the SQLStructuredType allows the user of the model to remember which template was used to create the table. It is the responsibility of the application using the model to keep the SQLStructuredType and the Table list of columns synchronized. The following instance diagram represents the examples above:

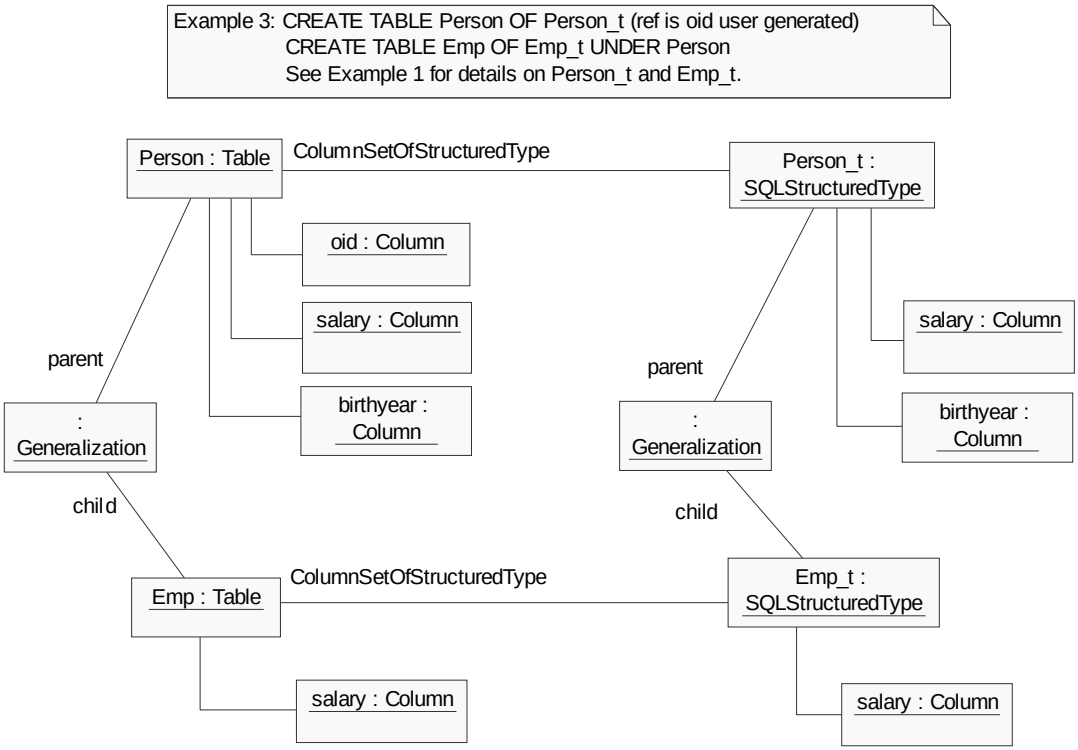


*Figure 9-6* Instance diagram for typed tables.

Finally, when a table (or a column) uses a structured type with a reference to another structured type, the reference is mapped to a table or view of the corresponding structured type, using the options scope clause. This represent an association between the column of the table or view with another table or view. This is modeled by the ColumnOptionsTable between a Column and a NamedColumnSet in CWM. For example, the statement *CREATE TABLE dept OF dept_t (ref is oid user generated, mgr WITH OPTIONS SCOPE emp)* would be represented by the following:

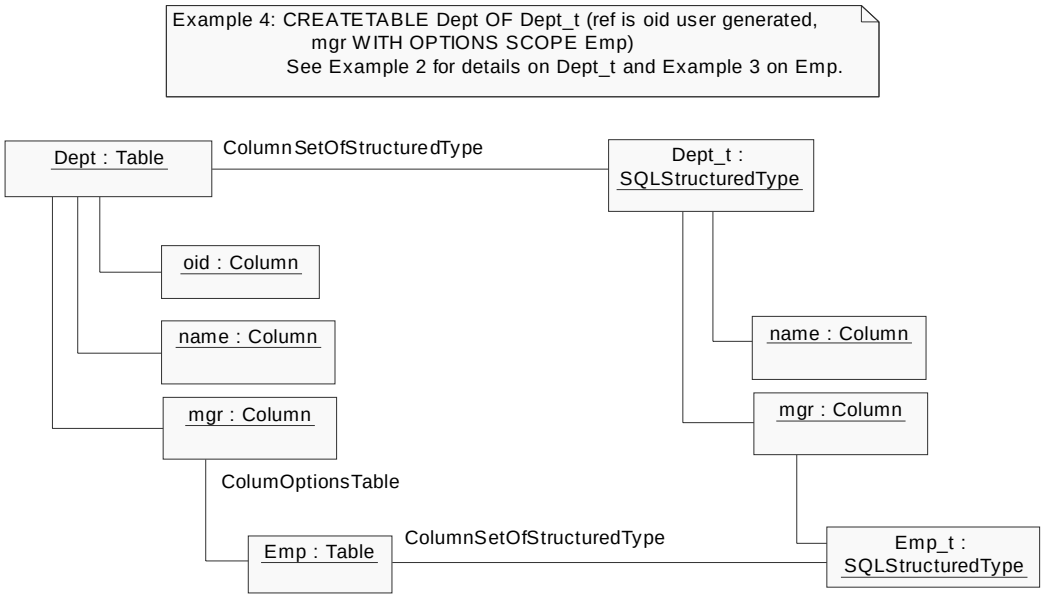


*Figure 9-7* Instance diagram showing the use of Options Scope clause.

In summary, the SQLStructuredType has the following associations:

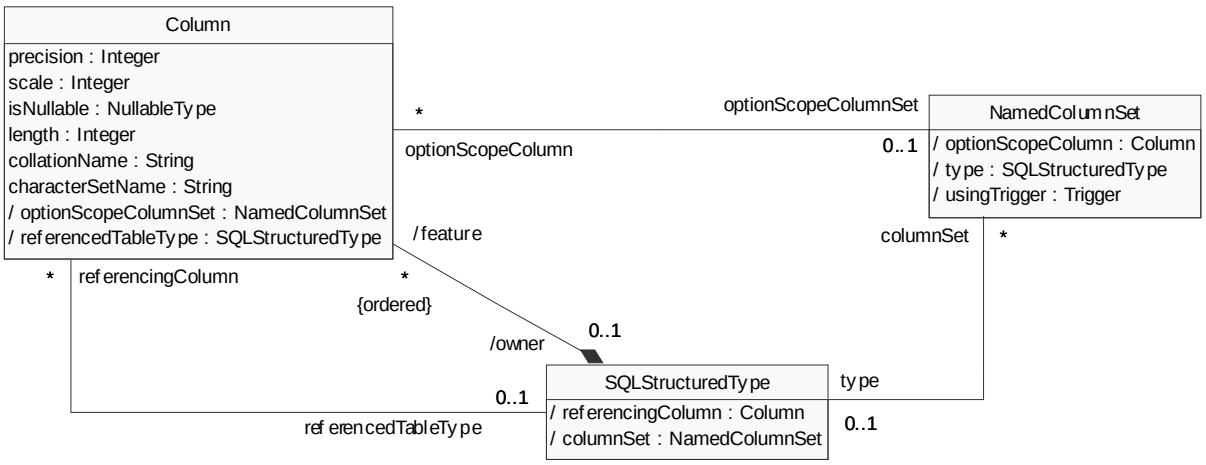


*Figure 9-8* SQLStructuredType and its associations

## *9.2.5 Keys*

The concept of a key, a set of attributes which defines uniqueness among the instances of a class, is already introduced in the Foundation Keys&Indexes package by the UniqueKey class. The Relational model extends the UniqueKey class to

UniqueConstraint. Similarly, the Relational package uses KeyRelationship from the Foundation package as the basis of a ForeignKey. The generic associations of the Foundation's UniqueKey and KeyRelationship between themselves, Class and StructuralFeatures are inherited by associations between UniqueConstraint, ForeignKey, Table and Columns in the Relational package.

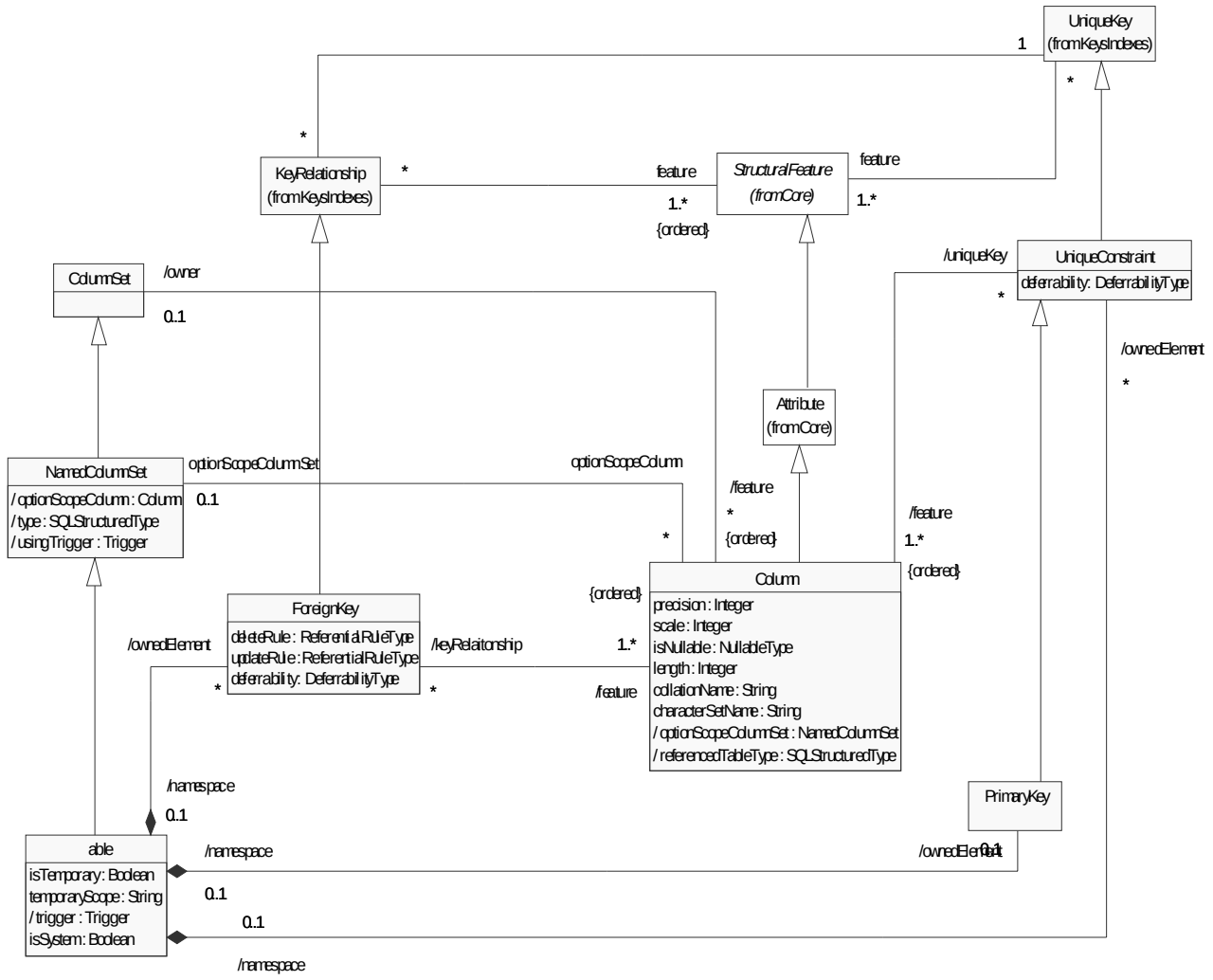


*Figure 9-9* UniqueConstraint and ForeignKey

## *9.2.6 Index*

Similarly to the keys, indexing is part of the Foundation and is extended in the Relational package.

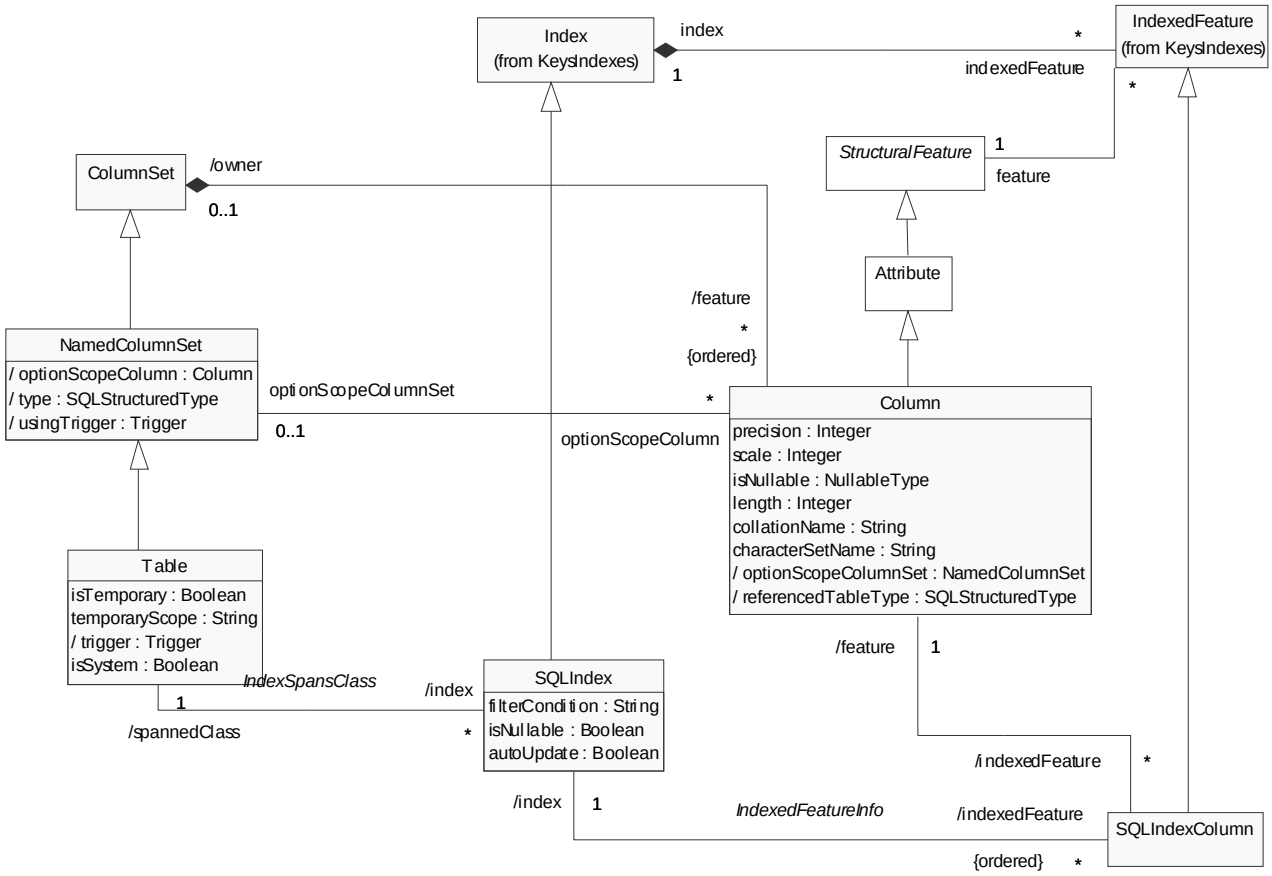


*Figure 9-10* Indexing

## *9.2.7 Triggers*

Triggers represent an action performed by the RDBMS when a certain table is changed. Triggers are associated to the Table they monitor and are owned by a Schema, which may or may not be the same as the Schema owning the table. In addition, Triggers that use tables in their expressions are associated with them.

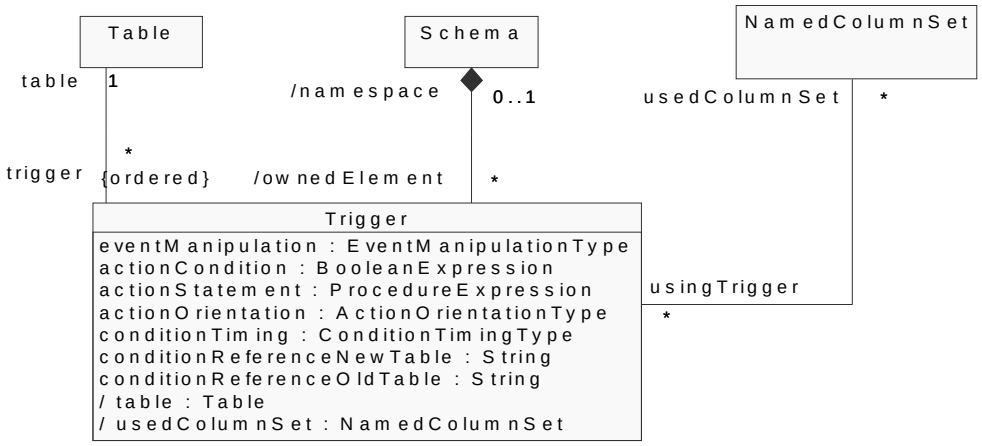


*Figure 9-11* Triggers

## *9.2.8 Procedures*

Procedures extend the ObjectModel Method class and are owned by a Schema (see Figure 9-2). The parameter and other information about the Procedure are illustrated in Figure 9-12.

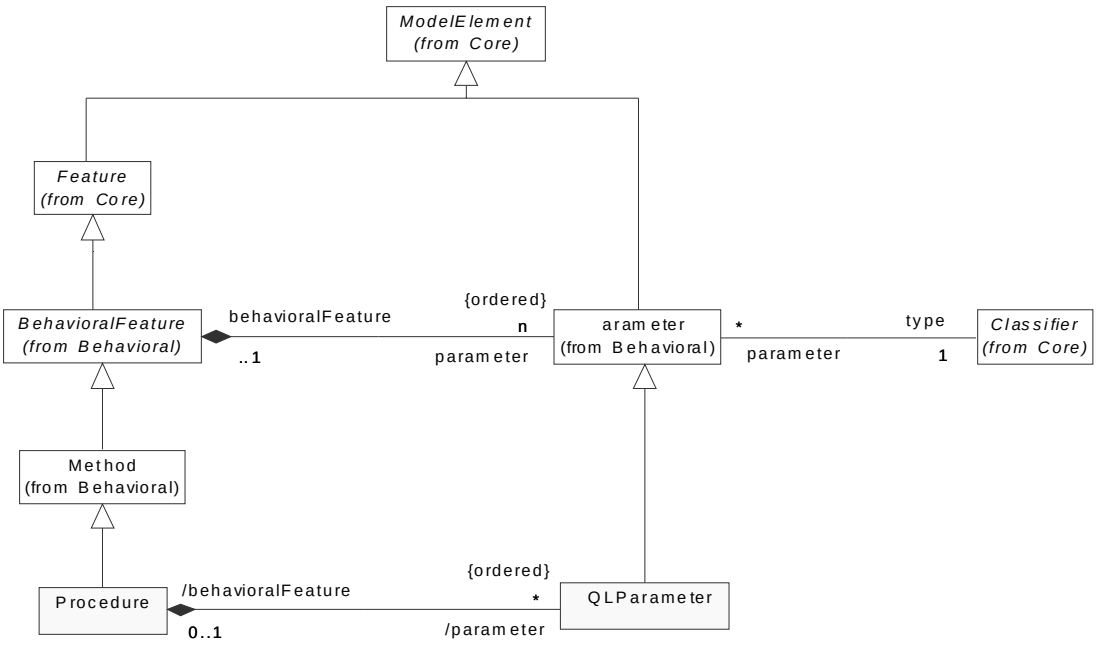


*Figure 9-12* Stored Procedures

### 9.2.9 Instances

It is sometimes necessary to provide either a copy or a sample of the data as part of the metadata. For example, one may want to specify during the design phase what will be the content of a Gender table. This is similar to the use of Collaboration diagrams in UML.

Figure 9-13 shows how a Rowset inherits from Extent, from the Foundation package. It represents all the data comprised in a ColumnSet. A RowSet can only be owned by a ColumnSet or any derived class. A RowSet contains Rows. Row inherits from Object. Its structure is defined by the corresponding ColumnSet and its Columns. Each Row is divided into ColumnValues, which match the value of a relational table, at the intersection of a row and a column. ColumnValue inherits from DataValue from ObjectModel.

Figure 9-14 shows a collaboration diagram, we show how the instances for the two column, two row Gender table are represented, and how they are associated with the Gender table definition. Two kinds of Instances are instantiated: Row and ColumnValue. The Row is associated with the AttributeLink through the instance/slot association. The ColumnValue is associated with the AttributeLink through the value association. While not shown on the diagram to keep it readable, each Instance is associated with a Class: the Row would be associated with the ColumnSet, and the ColumnValue with the SQLType of the corresponding Column.

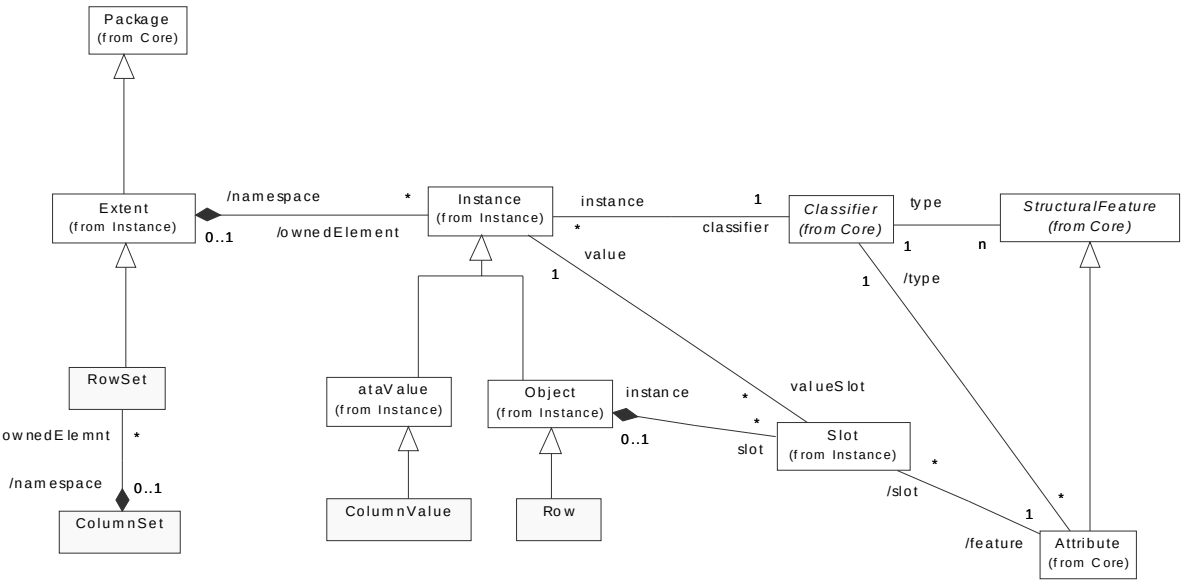


*Figure 9-13* Relational Instance classes

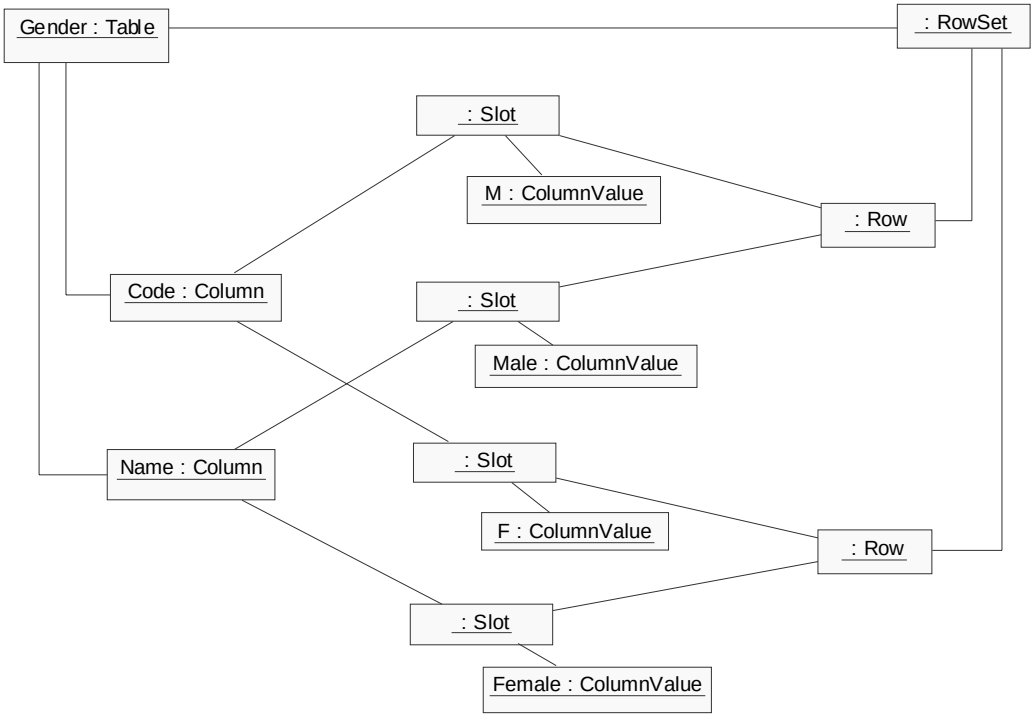


*Figure 9-14* Collaboration diagram showing use of instance classes

## 9.3 Relational Classes

### 9.3.1 Catalog

A Catalog is the unit of logon and identification. It also identifies the scope of SQL statements: the tables contained in a catalog can be used in a single SQL statement.

#### *Superclasses*

Package

#### *Contained Elements*

Schema

#### *Attributes*

***defaultCharacterSetName***

The name of the default character set used for the values in the column. This field applies only to columns whose datatype is a character string.

| | |
|---|---|
| type: | String |
| multiplicity: | exactly one |

***defaultCollationName***

The name of the default collation sequence used to sort the data values in the column.
This applies only to columns whose datatype is a form of character string.

| | |
|---|---|
| type: | String |
| multiplicity: | exactly one |

## 9.3.2 CheckConstraint

A rule that specifies the values allowed in one or more columns of every row of a table.

### Superclasses

Constraint

### Attributes

***deferrability***

Indicates the timing of the constraint enforcement during multiple-user updates.

| | |
|---|---|
| type: | DeferrabilityType ( initiallyDeferred \| initiallyImmediate \| notDeferrable ) |
| multiplicity: | exactly one |

## 9.3.3 Column

A column in a result set, a view, a table, or an SQLStructuredType.

### Superclasses

Attribute

### Attributes

***characterSetName***

The name of the character set used for the values in the column.
This field applies only to columns whose datatype is a character string.

| | |
|---|---|
| type: | String |
| multiplicity: | exactly one |

***collationName***

The name of the collation sequence used to sort the data values in the column.
This applies only to columns whose datatype is a form of character string.

| | |
|---|---|
| type: | String |
| multiplicity: | exactly one |

***isNullable***

Indicates if null values are valid in this column.

| | |
|---|---|
| type: | NullableType ( columnNoNulls \| columnNullable \| columnNullableUnknown ) |
| multiplicity: | exactly one |

***length***

The length of fixed length character or byte strings. Maximum length if length is variable.

| | |
|---|---|
| type: | Integer |
| multiplicity: | zero or one |

***precision***

The total number of digits in the field

| | |
|---|---|
| type: | Integer |
| multiplicity: | zero or one |
| constraints: | Scale must be specified when precision is specified |

***scale***

The number of digits on the right of the decimal separator.

| | |
|---|---|
| type: | Integer |
| multiplicity: | zero or one |

### References

#### *referencedTableType*

The column, used in an SQLStructuredType is a REF to a type. This reference the REF'ed SQLStructuredType.

| | |
|---|---|
| class: | SQLStructuredType |
| defined by: | ColumnRefStructuredType::referencedTableType |
| multiplicity: | zero or one |
| inverse: | SQLStructuredType::referencingColumn |

#### *optionScopeColumnSet*

Reference to the NamedColumnSet (Table or View) indicated in the SCOPE clause of the Column definition.

| | |
|---|---|
| class: | NamedColumnSet |
| defined by: | ColumnOptionsColumnSet::optionScopeColumnSet |
| multiplicity: | zero or one |
| inverse: | NamedColumnSet::optionScopeColumn |

### Constraints

The ***scale*** attribute is valid only if the ***precision*** attribute is specified. [C-3]

## 9.3.4 ColumnSet

A set of columns, representing either the result of a query, a view or a physical table.

### Superclasses

Class

### Contained Elements

Column

## 9.3.5 ColumnValue

The value in a column instance.

### *Superclasses*

DataValue

## *9.3.6 ForeignKey*

A Foreign Key associates columns from one table with columns of another table.

### *Superclasses*

KeyRelationship

### *Attributes*

***deleteRule***

An enumerated type. Indicates the disposition of the data records containing the foreign key value when the record of the matching primary key is deleted.

| | |
|---|---|
| type: | ReferentialRuleType ( importedKeyNoAction \| importedKeyCascade \| importedKeySetNull \| importedKeyRestrict \| importedKeySetDefault ) |
| multiplicity: | exactly one |

***updateRule***

Same as deleteRule for updates of the primary key data record

| | |
|---|---|
| type: | ReferentialRuleType ( importedKeyNoAction \| importedKeyCascade \| importedKeySetNull \| importedKeyRestrict \| importedKeySetDefault ) |
| multiplicity: | exactly one |

***deferrability***

Indicates if the validity of the ForeignKey is to be tested at each statement or at the end of a transaction.

| | |
|---|---|
| type: | DeferrabilityType ( initiallyDeferred \| initiallyImmediate \| notDeferrable ) |
| multiplicity: | exactly one |

## *9.3.7 NamedColumnSet*

A catalogued set of columns, which may be Table or View.

Note for typed tables: It is assumed that the typed table will own a set of columns conforming to the type they are OF. This set of columns allows the manipulation of the table by products which ignore this [SQL] extension. It also allows the columns of type REF, to be copied to a column with a SCOPE reference.

### *Superclasses*

ColumnSet

### *References*

***usingTrigger***

A Trigger which references this NamedColumnSet in its expression

| | |
|---|---|
| class: | Trigger |
| defined by: | TriggerUsingColumnSet::usingTrigger |
| multiplicity: | zero or more |
| inverse: | Trigger::usedColumnSet |

***type***

For typed Tables and Views, reference the base SQLStructuredType.

| | |
|---|---|
| class: | SQLStructuredType |
| defined by: | ColumnSetOfStructuredType::type |
| multiplicity: | zero or one |
| inverse: | SQLStructuredType::columnSet |

***optionScopeColumn***

This NamedColumnSet is referenced in a SCOPE clause of the referenced Column.

| | |
|---|---|
| class: | Column |
| defined by: | ColumnOptionsColumnSet::optionScopeColumn |
| multiplicity: | zero or more |
| inverse: | Column::optionScopeColumnSet |

## *9.3.8 PrimaryKey*

There is only one UniqueConstraint of type PrimaryKey per Table. It is implemented specifically by each RDBMS.

### Superclasses

UniqueConstraint

## 9.3.9 Procedure

This class describes Relational DBMS Stored procedures and functions.

### Superclasses

Method

### Attributes

***type***

A Procedure can be either a Function or a true Procedure. This indicates whether this object returns a value or not.

| | |
|---|---|
| type: | ProcedureType ( procedure \| function ) |
| multiplicity: | exactly one |

## 9.3.10 QueryColumnSet

The result set of a query.

### Superclasses

ColumnSet

### Attributes

***query***

The query expression generating this result. The language attribute of the expression should generally begin with "SQL"

| | |
|---|---|
| type: | QueryExpression |
| multiplicity: | exactly one |

## 9.3.11 Row

An instance of a ColumnSet.

### *Superclasses*

Object

## *9.3.12 RowSet*

Each instance of RowSet owns a collection of Row instances. The inherited association between Namespace (a superclass of Package) and ModelElement is used to contain Instances.

### *Superclasses*

Extent

### *Contained Elements*

Row

## *9.3.13 Schema*

A schema is a named collection of tables

### *Superclasses*

Package

### *Contained Elements*

NamedColumnSet

Trigger

Procedure

SQLIndex

CheckConstraint

## *9.3.14 SQLDataType* *abstract*

A SQLDataType is used to reference any datatype associated with a column

### *Superclasses*

Classifier

### *Attributes*

***typeNumber***

The number assigned to the datatype by the owning RDBMS

| | |
|---|---|
| type: | Integer |
| multiplicity: | zero or one |

## *9.3.15 SQLDistinctType*

A datatype defined as a Distinct Type, per [SQL] standard.

### *Superclasses*

SQLDataType

TypeAlias

### Attributes

***length***

The length of fixed length character or byte strings. Maximum length if length is variable.

| | |
|---|---|
| type: | Integer |
| multiplicity: | zero or one |

***precision***

The total number of digits in the field

| | |
|---|---|
| type: | Integer |
| multiplicity: | zero or one |

***scale***

The number of digits on the right of the decimal separator.

| | |
|---|---|
| type: | Integer |
| multiplicity: | zero or one |

### References

***sqlSimpleType***

The SQLSimpleType used to define the SQLDstinctType.

| | |
|---|---|
| class: | SQLSimpleType |
| definedBy: | SQLDistinctTypeWithSQLSimpleType |
| multiplicity: | exactly one |

## 9.3.16 SQLIndex

An Index on a table.

### Superclasses

Index

### Contained Elements

SQLIndexColumn

### Attributes

***filterCondition***

Which subset of the table is indexed

| | |
|---|---|
| type: | String |
| multiplicity: | exactly one |

***isNullable***

Entries in this index can be null

| | |
|---|---|
| type: | Boolean |
| multiplicity: | exactly one |

***autoUpdate***

The index is updated automatically

| | |
|---|---|
| type: | Boolean |
| multiplicity: | exactly one |

## 9.3.17 SQLIndexColumn

Associates an index with its columns.

This is really an association (link) class. It is associated with one index and one column.

### Superclasses

IndexedFeature

## 9.3.18 SQLParameter

Parameters of stored procedures.

### Superclasses

Parameter

### 9.3.19 SQLSimpleType

A simple datatype used with an SQL column. Examples are Integer, Varchar, LOB, CLOB, etc...

#### Superclasses

DataType

SQLDataType

### Attributes

***characterMaximumLength***

See [SQL], corresponding field in DATA_TYPE_DESCRIPTOR

type: Integer

multiplicity: zero or one

***characterOctetLength***

See [SQL], corresponding field in DATA_TYPE_DESCRIPTOR

type: Integer

multiplicity: zero or one

***numericPrecision***

See [SQL], corresponding field in DATA_TYPE_DESCRIPTOR

type: Integer

multiplicity: zero or one

***numericPrecisionRadix***

See [SQL], corresponding field in DATA_TYPE_DESCRIPTOR

type: Integer

multiplicity: zero or one

***numericScale***

See [SQL], corresponding field in DATA_TYPE_DESCRIPTOR

type: Integer

multiplicity: zero or one

***dateTimePrecision***

See [SQL], corresponding field in DATA_TYPE_DESCRIPTOR

type: Integer

multiplicity: zero or one

## 9.3.20 SQLStructuredType

A Datatype defined as Structured Type, per [SQL] standard.

### *Superclasses*

Class

SQLDataType

### *Contained Elements*

Column

### *References*

***columnSet***

A NamedColumnSet created as of this type.

| | |
|---|---|
| class: | NamedColumnSet |
| defined by: | ColumnSetOfStructuredType::columnSet |
| multiplicity: | zero or more |
| inverse: | NamedColumnSet::type |

***referencingColumn***

Reference a column of an SQLStructuredType (otherType) which is created with a REF clause referencing this SQLStructuredType (thisType). Note that in general, otherType and thisType are two different instances of SQLStructuredType.

| | |
|---|---|
| class: | Column |
| defined by: | ColumnRefStructuredType::referencingColumn |
| multiplicity: | zero or more |
| inverse: | Column::referencedTableType |

## *9.3.21 Table*

A materialized NamedColumnSet.

### *Superclasses*

NamedColumnSet

### *Contained Elements*

UniqueConstraint

ForeignKey

## Attributes

### *isSystem*

Indicates that the Table is a System Table (generally part of or view on the system catalog).

| | |
|---|---|
| type: | Boolean |
| multiplicity: | exactly one |

### *isTemporary*

Indicates that the table content is temporary. SQL92 standards provide two types of temporary tables (local Temporary and Global Temporary). However, RDBMS products have implemented variations on this theme. It is recommended that the product manufacturers provide specific temporary information (besides the temporaryScope attribute) in their extensions.

| | |
|---|---|
| type: | Boolean |
| multiplicity: | exactly one |

### *temporaryScope*

This attribute is meaningful only when the isTemporary flag is True [C-1]. The scope indicates when the data of this table are available. "SESSION", "APPLICATION" are examples of possible values. Look at the Scope attribute for Global Temporary tables in the SQL standards for more details.

| | |
|---|---|
| type: | String |
| multiplicity: | zero or one |
| constraints: | May not be specified if isTemporary is set to false |

## References

### *trigger*

Associates triggers executed during changes to the table.

| | |
|---|---|
| class: | Trigger |
| defined by: | TableOwningTrigger::trigger |
| multiplicity: | zero or more; ordered |
| inverse: | Trigger::table |

## Constraints

Attribute ***temporaryScope*** is meaningful only when the ***isTemporary*** flag is True [C-1]

### 9.3.22 Trigger

An action run by the DBMS when specified events occur on the table owning the Trigger

#### Superclasses

ModelElement

## Attributes

### *eventManipulation*

Indicates what types of events are using the current Trigger.

| | |
|---|---|
| type: | EventManipulationType ( insert \| delete \| update ) |
| multiplicity: | exactly one |

### *actionCondition*

A boolean expression which defines when the trigger has to be executed

| | |
|---|---|
| class: | BooleanExpression |
| multiplicity: | exactly one |

### *actionStatement*

The Trigger action itself

| | |
|---|---|
| class: | ProcedureExpression |
| multiplicity: | exactly one |

### *actionOrientation*

It indicates if the trigger is called once per statement execution or before or after each row of the table is modified.

| | |
|---|---|
| class: | ActionOrientationType ( row \| statement ) |
| multiplicity: | exactly one |

### *conditionTiming*

It indicates if the trigger activity is run before or after the statement or row is modified.

| | |
|---|---|
| class: | ConditionTimingType ( before \| after ) |
| multiplicity: | exactly one |

### *conditionReferenceNewTable*

The alias for the owning table name, used in the actionStatement, to represent the state of the table after the insert/delete/update

| | |
|---|---|
| class: | String |
| multiplicity: | exactly one |

***conditionReferenceOldTable***

The alias for the name of the owning table, used in the actionStatement, to represent the state of the table before the update/delete/insert.

| | |
|---|---|
| class: | String |
| multiplicity: | exactly one |

### *References*

***usedColumnSet***

Tables referenced by the actionStatement or the actionCondition.

| | |
|---|---|
| class: | NamedColumnSet |
| defined by: | TriggerUsingColumnSet::usedColumnSet |
| multiplicity: | zero or more |
| inverse: | NamedColumnSet::usingTrigger |

***table***

The table which owns the Trigger

| | |
|---|---|
| class: | Table |
| defined by: | TableOwningTrigger::table |
| multiplicity: | exactly one |
| inverse: | Table::trigger |

## *9.3.23 UniqueConstraint*

A condition to define uniqueness of rows in a table. An example of UniqueConstraint is a primary key

### *Superclasses*

UniqueKey

### Attributes

***deferrability***

Indicates if the validity of the UniqueConstraint is to be tested at each statement or at the end of a transaction.

| | |
|---|---|
| type: | DeferrabilityType ( initiallyDeferred \| initiallyImmediate \| notDeferrable ) |
| multiplicity: | exactly one |

## 9.3.24 View

A view is a non-materialized set of rows, defined by the associated query.

### Superclasses

NamedColumnSet

### Contained Elements

QueryExpression

### *Attributes*

***isReadOnly***

Indicates whether the underlying tables can be updated through an update to this View.

| | |
|---|---|
| type: | Boolean |
| multiplicity: | exactly one |

***queryExpression***

The query associated with the View. -
The query result must match the set of Columns associated with the View (in parent class ColumnSet)

| | |
|---|---|
| type: | QueryExpression |
| multiplicity: | exactly one |

***checkOption***

This field is meaningful only if the view is not ReadOnly. CheckOption indicates that the RDBMS will validate that changes made to the data verify the view filtering condition and belong to the view result set.

| | |
|---|---|
| type: | Boolean |
| multiplicity: | exactly one |
| constraints: | only used when isReadOnly=false |

### *Constraints*

checkOption is valid only if isReadOnly is False. [C-2]

## *9.4 Relational Associations*

### *9.4.1 ColumnOptionsColumnSet* *protected*

Associates Columns with NamedColumnSets they reference in their OPTIONS clause.

*Ends*

***optionScopeColumn***

Reference to the Column which contains theSCOPE clause.

| | |
|---|---|
| class: | Column |
| multiplicity: | zero or more |

***optionScopeColumnSet***

Reference to the NamedColumnSet indicated in the SCOPE clause of the Column definition.

| | |
|---|---|
| class: | NamedColumnSet |
| multiplicity: | zero or one |

### 9.4.2 ColumnRefStructuredType *protected*

Associates Columns of a StructuredType with the Type they reference in the REF clause

*Ends*

***referencedTableType***

The column, used in an SQLStructuredType is a REF to a type. This references the REF'ed SQLStructuredType.

| | |
|---|---|
| class: | SQLStructuredType |
| multiplicity: | zero or one |

***referencingColumn***

Reference to a column of an SQLStructuredType (otherType) which is created with a REF clause referencing this SQLStructuredType (thisType). Note that in general, otherType and thisType are two different instances of SQLStructuredType.

| | |
|---|---|
| class: | Column |
| multiplicity: | zero or more |

### 9.4.3 ColumnSetOfStructuredType *protected*

Associates structured types with NamedColumnSets defined of this type.

*Ends*

***type***

For typed Tables and Views, reference to the base SQLStructuredType.

| | |
|---|---|
| class: | SQLStructuredType |
| multiplicity: | zero or one |

***columnSet***

A NamedColumnSet created as of this type.

| | |
|---|---|
| class: | NamedColumnSet |
| multiplicity: | zero or more |

### *9.4.4 DistinctTypeHasSimpleType*

*Ends*

***sqlDistinctType***

Distinct types that use this simple type.

| | |
|---|---|
| class: | SQLDistinctType |
| multiplicity: | zero or more |

***sqlSimpleType***

The Simple type used to define the distinct class.

| | |
|---|---|
| class: | SQLSimpleType |
| multiplicity: | exactly one |

### *9.4.5 TableOwningTrigger* *protected*

Associates a Table with its Triggers. The Trigger will be activated when an action is performed on the Table.

*Ends*

***table***

The table which owns the Trigger

class: Table

multiplicity: exactly one

***trigger***

Associates triggers executed during changes to the table.

class: Trigger

multiplicity: zero or more; ordered

### *9.4.6 TriggerUsingColumnSet* *protected*

This associates a Trigger with the NamedColumnSets it uses in its expressions.

*Ends*

***usedColumnSet***

NamedColumnSets referenced by the actionStatement or the actionCondition.

class: NamedColumnSet

multiplicity: zero or more

***usingTrigger***

A Trigger which references this table in its expression

class: Trigger

multiplicity: zero or more

## 9.5 OCL Representation of Relational Constraints

[C-1] ***temporaryScope*** is valid only if the ***isTemporary*** is True.
**context** Table **inv:**
self.temporaryScope.notEmpty **implies** self.isTemporary=True

[C-2] ***checkOption*** is valid only if ***isReadOnly*** is False.
**context** View **inv:**
self.checkOption **implies** self.isReadOnly=False

[C-3] ***scale*** is valid only if ***precision*** is specified.
**context** Column **inv:**
self.scale.nonEmpty **implies** self.precision.notEmpty

# Record 10

## 10.1 Overview

The Record package covers the basic concept of a record and its structure. The package takes a broad view of the notion of record, including both traditional data records such as those stored in files and databases, as well as programming language structured data types. In fact, the concepts described here can be used as a foundation for extension packages describing any information structure that is fundamentally hierarchical, or "nested," in nature such as documents, questionnaires, and organizational structures.

## 10.2 Organization of the Record Package

The Record package depends on the following packages:

- org.omg::CWM::ObjectModel::Core
- org.omg::CWM::ObjectModel::Instance

Because of the antiquity of many record-based models, individual system implementations employing record models may have unusual features (such as occurs-depending arrays, various COBOL rename/remapping semantics, etc.) that are not shared with other implementations. When such features are limited to single implementations or languages, they have been purposefully left out of the Record metamodel. Rather, unusual features of this sort should be placed into extension packages designed to meet the needs of those implementations or languages. For example, record structuring features endemic to the COBOL language have been placed in the COBOLData metamodel in the CWMX package described in Volume 2 and do not appear here. In this way, COBOL-only features do not burden other record oriented implementations unnecessarily.

The Record metamodel appears in Figure 10-2-1.

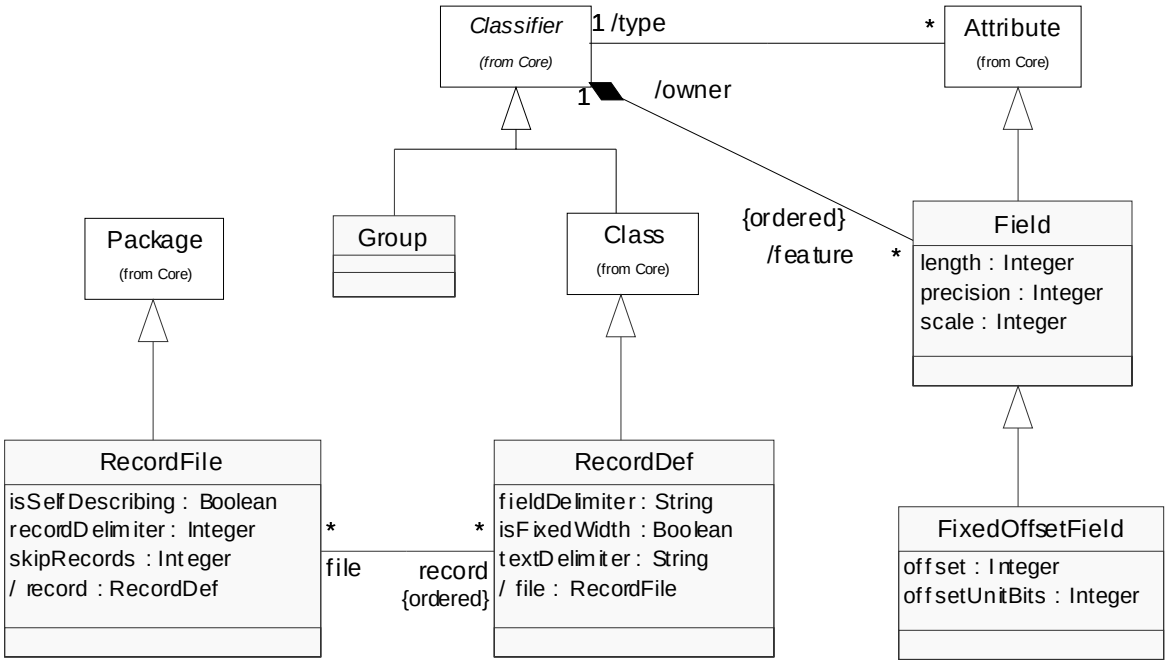


*Figure 10-2-1* Record Package

The instance diagram in Figure 10-2-2 shows how a record description is represented in this model. The record contains three fields, one of which is a group item that itself has embedded fields. The main RecordDef is named Customer. It contains three Fields – account, custName and custAddress.

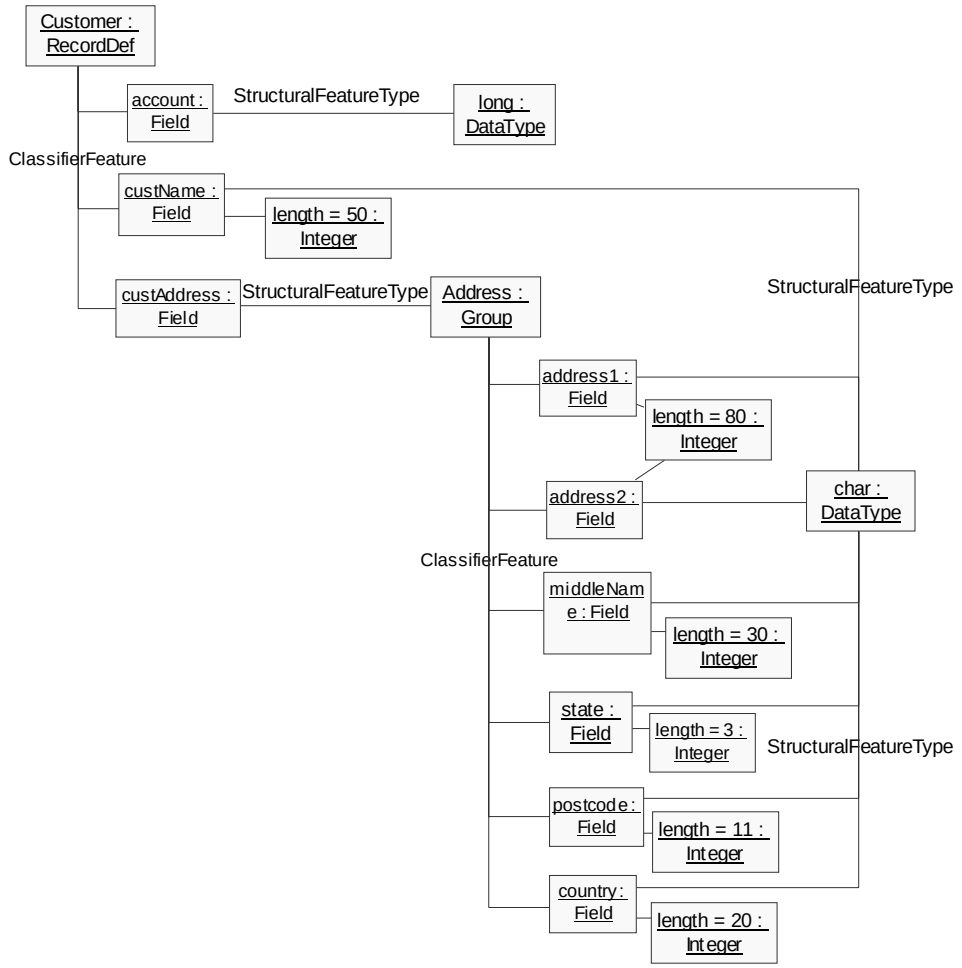


*Figure 10-2-2* Record metamodel instance example

The *account* is a numeric field with a type of *long*, which is an instance of DataType. Size information about the field -- its *length*, *precision*, and *scale* -- are not relevant for the *long* data type.

The field *custName* has a type of *char*, which is another instance of DataType. The field is 50 characters in *length* but needs no *precision* or *scale* information.

Field *custAddress* is a single field; its internal structure is determined from its type *Address*, an instance of Group containing six fields. *address1* and *address2* have type of *char* and are 80 characters long. *city* is also of type of *char* but is 30 characters long. *state*, *postcode* and *country* are of the type *char* as well but are 3, 11 and 20 characters long, respectively.

Table 10-0 shows how the example RecordDef would be described in three widely used programming languages.

**Table 10-0 Language representations of Record metamodel example**

| C | COBOL | PL/I |
|---|---|---|
| typedef struct Address {<br>  char address1[80];<br>  char address2[80];<br>  char city[30];<br>  char state[3];<br>  char postcode[11];<br>  char country[20];<br>} Address;<br>typedef struct Customer {<br>  long  account;<br>  char  custName[50];<br>  Address custAddress;<br>} Customer;<br>Customer cust; | 01 Customer.<br>  05 account PIC 999999<br>  USAGE BINARY.<br>  05 custName PIC X(50).<br>  05 custAddress.<br>  10 address1 PIC X(80).<br>  10 address2 PIC X(80).<br>  10 city PIC X(30).<br>  10 state PIC X(3).<br>  10 postcode PIC X(11).<br>  10 country PIC X(20). | DECLARE<br>  1 CUSTOMER ,<br>  2 ACCOUNT FIXED BIN(31,0),<br>  2 CUSTNAME CHAR(50),<br>  2 CUSTADDRESS,<br>  3 ADDRESS1 CHAR(80),<br>  3 ADDRESS2 CHAR(80),<br>  3 CITY CHAR(30),<br>  3 STATE CHAR(3),<br>  3 POSTCODE CHAR(11),<br>  3 COUNTRY CHAR(20); |

## *10.2.1 Instances*

Instances of records are created by extending the ObjectModel's Instance package as shown in Figure 10-2-3.

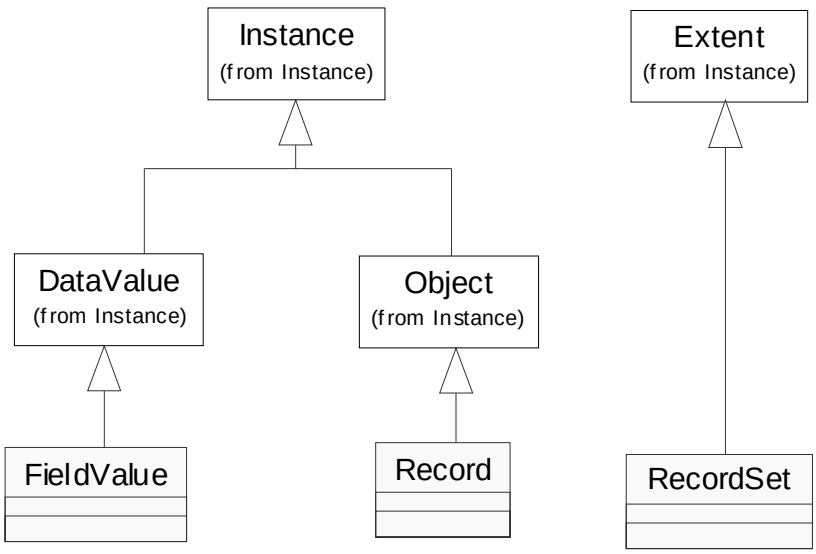


*Figure 10-2-3* Record metamodel instances

Figure 10-2-4 shows an example of how record instances are created using the Record, FieldValue, and RecordSet classes. The example uses the metamodel instances in Figure 10-2-2 to store the address of the President of the United States.

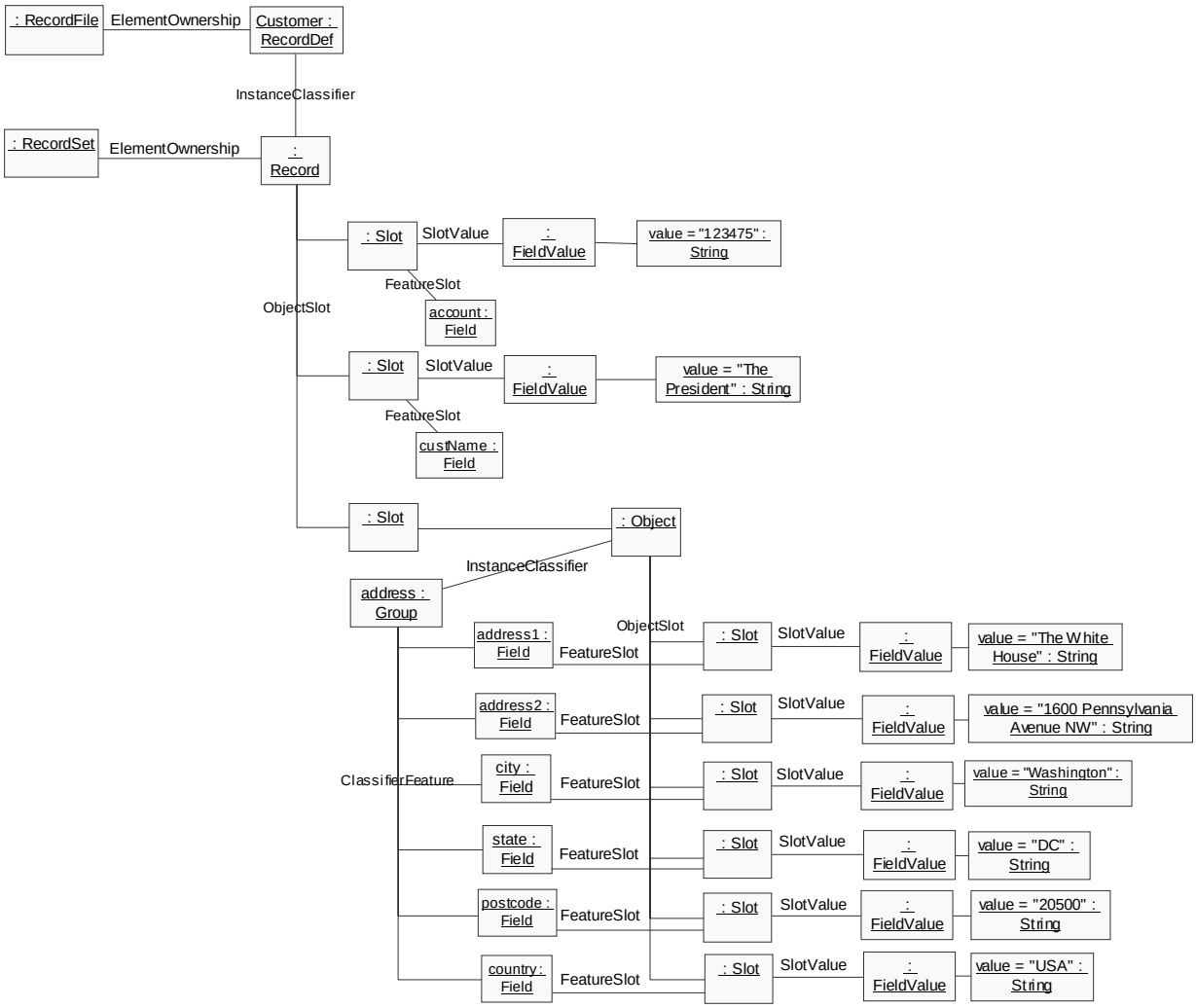


*Figure 10-2-4* Record instance example

## 10.3 Record Classes

### 10.3.1 Field

A Field is the fundamental information container within a RecordDef. It holds one piece of information, which may itself have structure. The inherited associations StructuralFeatureType and ElementOwnership provide access to a Field instance's type and owning classifier, respectively.

### *Superclasses*

Attribute

### *Attributes*

***length***

The length of a fixed length character or byte string field.

| | |
|---|---|
| type: | Integer |
| multiplicity: | zero or one |

***precision***

The total number of digits in a numeric field.

| | |
|---|---|
| type: | Integer |
| multiplicity: | zero or one |

***scale***

The number of digits on the right of the decimal separator in a numeric field.

| | |
|---|---|
| type: | Integer |
| multiplicity: | zero or one |

### *Constraints*

Owner and type cannot refer to the same Classifier. [C-1]

The scale attribute is valid only if the precision attribute is specified. [C-2]

The precision attribute is valid only if the length attribute is not specified. [C-3]

## *10.3.2 FieldValue*

The value currently held in a Field instance.

### *Superclasses*

DataValue

## *10.3.3 FixedOffsetField*

Instances of FixeOffsetField represent fields that have a fixed location in a record.

FixedOffsetFields can be used as a foundation for recording details of physical record layouts and as a means of representing the internal structure of undiscriminated (ie, C-type) unions.

### *Superclasses*

Field

### *Attributes*

***offset***

Specifies the offset of the field within its container in units of the number of bits indicated in the offsetUnitBits attribute.

| | |
|---|---|
| type: | Integer |
| multiplicity: | exactly one |

***offsetUnitBits***

The number of bits making up one record offset unit. For example, for a byte-relative offset, the value of this attribute would typically be 8.

| | |
|---|---|
| type: | Integer |
| multiplicity: | exactly one |

## *10.3.4 Group*

A Group is a structured data type and is used to collect together Field instances within a Record. Groups can be used in RecordDef instances as shown in the foregoing example.

### *Superclasses*

Classifier

## *10.3.5 Record*

A Record, a subclass of Object, represents a single data record. Each Record is described by a RecordDef instance found via the Object's InstanceClassifier association.

### *Superclasses*

Object

### 10.3.6 RecordDef

A RecordDef is an ordered collection of Fields representing the structure of a Record. Examples of RecordDefs include definitions of

- language-specific data structures
- database records
- IMS segments

The internal structure of a RecordDef instance is constructed by adding Field instances as features (using the ElementOwnership association) and pointing each Field instance's inherited type reference to the Classifier instance representing the Field's data type. The referenced instance can be either a primitive data type (an instance of DataType, such as "integer") or a structured data type (such as a Group instance).

Refer to the foregoing example for more details of the relationships between RecordDefs, Fields, Records, and their values.

#### Superclasses

Class

#### Contained Elements

Field

#### Attributes

***fieldDelimiter***

The value of a fieldDelimiter used to separate field values in an input stream.

| | |
|---|---|
| type: | String |
| multiplicity: | zero or one |

***isFixedWidth***

True if the record is fixed length. Otherwise, the record can be of variable length.

| | |
|---|---|
| type: | Boolean |
| multiplicity: | exactly one |

***textDelimiter***

The delimiter of a text string in the record, such as a quote.

| | |
|---|---|
| type: | String |
| multiplicity: | zero or one |

### *References*

***file***

Identifies files containing Records described by the RecordDef.

| | |
|---|---|
| class: | RecordFile |
| defined by: | RecordToFile::file |
| multiplicity: | zero or more |
| inverse: | RecordFile::record |

## *10.3.7 RecordFile*

A RecordFile is the definition of a file. It may have one or more RecordDefs, defining the structure of the records in the file. Each of these RecordDefs defines a valid structure for records in the file. Subclasses of RecordFile in extensions to support specific languages and systems may be used to represent specific types of files such as COBOL CopyLib files and C-language header files.

Physical deployments of a RecordFile can be found via the DataManagerDataPackage association in the SoftwareDeployment package .

### *Superclasses*

Package

### *Attributes*

***isSelfDescribing***

True if the contents of fields in the first record of the file contain field names applicable to subsequent records.

| | |
|---|---|
| type: | Boolean |
| multiplicity: | exactly one |

***recordDelimiter***

Contains the value that serves as a logical end-of-record indication in a stream-oriented file. A common examples include the usage of carriage-return characters and carriage-return/line-feed character pairs as new-line characters in ASCII text files.

| | |
|---|---|
| type: | String |
| multiplicity: | zero or one |

***skipRecords***

The number of records to ignore at the beginning of a file. The specific semantics of records that are skipped may be beyond the scope of CWM.

| | |
|---|---|
| type: | Integer |
| multiplicity: | zero or one |

### *References*

***record***

The record definitions used to describe the layout of individual record instances stored in the file. The ordering of these RecordDefs may be used to indicate the physical sequence in which records of various types are expected.

| | |
|---|---|
| class: | RecordDef |
| defined by: | RecordToFile::record |
| multiplicity: | zero or more; ordered |
| inverse: | RecordDef::file |

## *10.3.8 RecordSet*

A RecordSet represents a collection of Record instances.

### *Superclasses*

Extent

### *Contained Elements*

Record

## 10.4 Record Associations

### 10.4.1 RecordToFile *Protected*

A Record definition can appy to records stored in a RecordFile.

*Ends*

***file***

Identifies the set of files in which a record is stored.

| | |
|---|---|
| class: | RecordFile |
| multiplicity: | zero or more |

***record***

Identifies the set of records stored in the file. The ordering may indicate the physical ordering of records with different layouts.

| | |
|---|---|
| class: | RecordDef |
| multiplicity: | zero or more; ordered |

## 10.5 OCL Representation of Record Constraints

[C-1] The owner of a Field and the type of a Field may not refer to the same Classifier instance.

**context** Field **inv:**

self.owner <> self.type

[C-2] The scale attribute is valid only if the precision attribute is specified.

**context** Field **inv:**

self.scale->notEmpty **implies** self.precision->notEmpty

[C-3] The precision attribute is valid only if the length attribute is not specified.

**context** Field **inv:**

self.precision->notEmpty **implies** self.length->isEmpty

# Multidimensional 11

## 11.1 Overview

The CWM Multidimensional metamodel is a generic representation of a multidimensional database.

Multidimensional databases are OLAP databases that are directly implemented by multidimensional database systems. In a multidimensional database, key OLAP constructs (dimensions, hierarchies, etc.) are represented by the internal data structures of a multidimensional database server, and common OLAP operations (consolidation, drill-down, etc.) are performed by the server acting on those data structures. Multidimensional databases are often classified as “physical OLAP” or “MOLAP" (memory-based OLAP) databases.

Multidimensional databases offer enhanced performance and flexibility over OLAP systems that simulate multidimensional functionality using other technologies (e.g., relational database or spreadsheet):

- Performance: Multidimensional databases provide rapid consolidation times and formula calculations, and consistent query response times regardless of query complexity. This is accomplished, in part, through the use of efficient cell storage techniques and highly-optimized index paths.
- Flexibility: The specification and use of multidimensional schemas and queries (including the design of cubes, dimensions, hierarchies, member formulas, the manipulation of query result sets, etc.) can be accomplished in a relatively straight-forward manner, since the server directly supports (and exposes) the multidimensional paradigm.

The CWM Multidimensional metamodel does not attempt to provide a complete representation of all aspects of commercially available, multidimensional databases. Unlike relational database management systems, multidimensional databases tend to be proprietary in structure, and there are no published, widely agreed upon, standard representations of the logical schema of a multidimensional database. Therefore, the CWM Multidimensional Database metamodel is oriented toward complete generality

of specification. Tool-specific extensions to the metamodel are relatively easy to formulate, and several examples are provided in Volume 2, Extensions, of the CWM Specification.

## 11.2 Organization of the Multidimensional Package

### 11.2.1 Dependencies

The Multidimensional package depends on the following packages:

- org.omg::CWM::ObjectModel::Core
- org.omg::CWM::ObjectModel::Instance

### 11.2.2 Major Classes and Associations

The major classes and associations of the Multidimensional metamodel are shown in Figure 11-1.

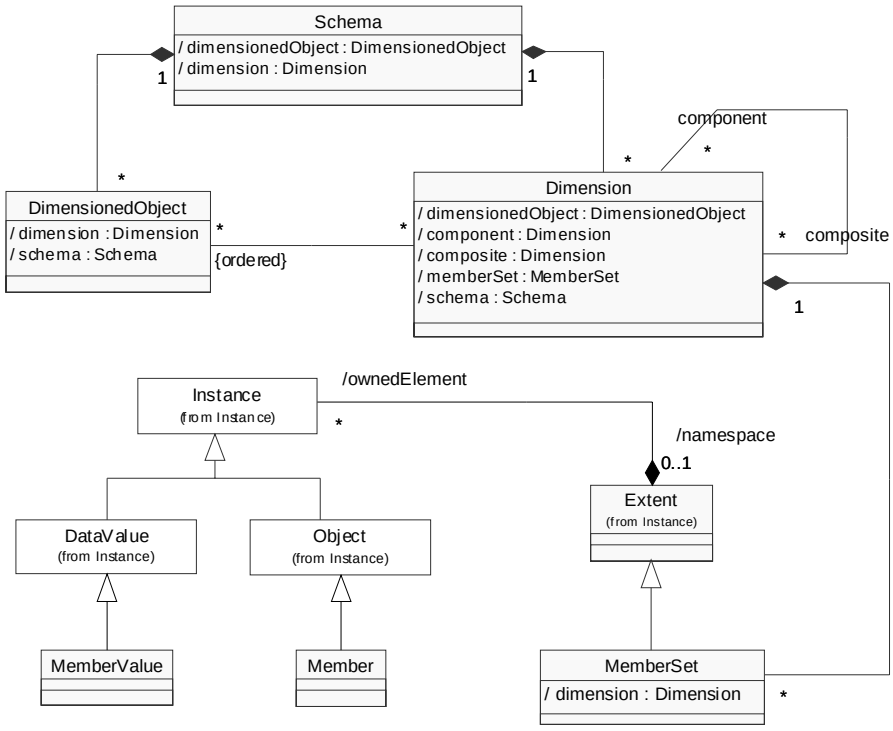


*Figure 11-1* Multidimensional Metamodel: Classes and Associations

Schema is the container of all elements comprising a Multidimensional model. It also represents the logical unit of deployment of a Multidimensional database instance.

Dimension represents a physical dimension in a Multidimensional database. Whereas the OLAP metamodel defines “dimension” as a purely conceptual entity, this Dimension represents the dimension object exposed by the programming model of a Multidimensional database.

A Dimension may reference other instances of Dimension to form arbitrarily complex dimensional structures (e.g., hierarchies with varying levels of detail).

DimensionedObject represents an attribute of Dimension. Examples of DimensionedObjects include measures (variables), formulas, consolidation functions, member alias names, etc. DimensionedObjects are contained by the Schema and referenced by the Dimensions that use them.

MemberSet represents the collection of Members associated with an instance of Dimension, and MemberValue represents an instance value of a Member. MemberSet, Member and MemberValue enable the specification and interchange of both M1-level Multidimensional models and associated M0-level data values.

### 11.2.3 Inheritance from the ObjectModel

Figure 11-2 illustrates the inheritance of the Multidimensional classes from metaclasses of the Object Model.

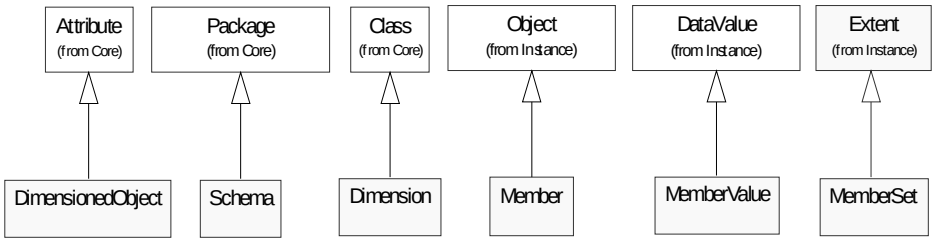


*Figure 11-2* Multidimensional Metamodel: Inheritance from Object Model

## 11.3 Multidimensional Classes

### 11.3.1 Dimension

Dimension represents physical dimension in a multidimensional database (e.g., a dimension object defined by the programming model/API of an OLAP database server). Tool-specific extensions to the Multidimensional package will generally contain classes that derive from Dimension.

## Superclasses

Class

## Contained Elements

MemberSet

## References

***dimensionedObject***

References the collection of DimensionedObjects associated with a Dimension.

| | |
|---|---|
| *class:* | DimensionedObject |
| *defined by:* | DimensionsReferenceDimensionedObjects ::dimensionedObject |
| *multiplicity:* | zero or more; ordered |
| *inverse:* | DimensionedObject::dimension |

***component***

References "component" Dimensions comprising this Dimension.

| | |
|---|---|
| *class:* | Dimension |
| *defined by:* | CompositesReferenceComponents::component |
| *multiplicity:* | zero or more |
| *inverse:* | Dimension::composite |

***composite***

References "composite" Dimensions comprised (in part) from this Dimension.

| | |
|---|---|
| *class:* | Dimension |
| *defined by:* | CompositesReferenceComponents::composite |
| *multiplicity:* | zero or more |
| *inverse:* | Dimension::component |

***memberSet***

References the collection of MemberSets owned by a Dimension.

| | |
|---|---|
| *class:* | MemberSet |
| *defined by:* | DimensionOwnsMemberSets::memberSet |
| *multiplicity:* | zero or more |
| *inverse:* | MemberSet::dimension |

***schema***

References the Schema owning a Dimension.

| | |
|---|---|
| *class:* | Schema |
| *defined by:* | MDSchemaOwnsDimensions::schema |
| *multiplicity:* | exactly one |
| *inverse:* | Schema::dimension |

### *Constraints*

A Dimension may not reference itself as a component, nor as a composite. [C-1]

The transitive closure of components of an instance of Dimension must not include the Dimension instance.

The transitive closure of composites of an instance of Dimension must not include the Dimension instance.

## *11.3.2 DimensionedObject*

DimensionedObject represents an attribute of Dimension.

### *Superclasses*

Attribute

### References

***dimension***

References the collection of Dimensions associated with this DimensionedObject.

| | |
|---|---|
| *class:* | Dimension |
| *defined by:* | DimensionsReferenceDimensionedObjects::dimension |
| *multiplicity:* | zero or more |
| *inverse:* | Dimension::dimensionedObject |

***schema***

References the Schema owning a DimensionedObject.

| | |
|---|---|
| *class:* | Schema |
| *defined by:* | MDSchemaOwnsDimensionedObjects::schema |
| *multiplicity:* | exactly one |
| *inverse:* | Schema::dimensionedObject |

## 11.3.3 Member

Member represents a member of a Dimension.

### Superclasses

Object

## 11.3.4 MemberSet

MemberSet represents the collection of Members associated with an instance of Dimension.

### Superclasses

Extent

### Contained Elements

- Member
- MemberValue

### References

***dimension***

References the Dimension owning a MemberSet.

| | |
|---|---|
| *class:* | Dimension |
| *defined by:* | DimensionOwnsMemberSets::dimension |
| *multiplicity:* | exactly one |
| *inverse:* | Dimension::memberSet |

## 11.3.5 MemberValue

MemberValue represents an instance value of a Member.

### Superclasses

DataValue

## 11.3.6 Schema

Schema contains all elements comprising a Multidimensional database.

### Superclasses

Package

### Contained Elements

- Dimension
- DimensionedObject

### References

#### *dimensionedObject*

References the collection of DimensionedObjects owned by a Schema.

| | |
|---|---|
| *class:* | DimensionedObject |
| *defined by:* | MDSchemaOwnsDimensionedObjects::dimensionedObject |
| *multiplicity:* | zero or more |
| *inverse:* | DimensionedObject::Schema |

#### *dimension*

References the collection of Dimensions owned by a Schema.

| | |
|---|---|
| *class:* | Dimension |
| *defined by:* | MDSchemaOwnsDimensions::dimension |
| *multiplicity:* | zero or more |
| *inverse:* | Dimension::Schema |

## 11.4 Multidimensional Associations

### 11.4.1 CompositesReferenceComponents

A Dimension may reference other instances of Dimension in order to derive more complex dimensional structures.

### Ends

#### *composite*

"Composite" Dimensions referencing "Component" Dimensions.

| | |
|---|---|
| *class:* | Dimension |
| *multiplicity:* | zero or more |

***component***

"Component" Dimensions referenced by "Composite" Dimensions.

| | |
|---|---|
| *class:* | Dimension |
| *multiplicity:* | zero or more |

### 11.4.2 DimensionOwnsMemberSets

A Dimension may own any number of MemberSets.

*Ends*

***dimension***

Dimension owning MemberSets.

| | |
|---|---|
| *class:* | Dimension |
| *multiplicity:* | exactly one |
| *aggregation:* | composite |

***memberSet***

MemberSets owned by a Dimension.

| | |
|---|---|
| *class:* | MemberSet |
| *multiplicity:* | zero or more |

### 11.4.3 DimensionsReferenceDimensionedObjects

A Dimension may reference several instances of DimensionedObject. A DimensionedObject may be referenced by several Dimensions.

*Ends*

***dimension***

Dimensions referencing DimensionedObjects.

| | |
|---|---|
| *class:* | Dimension |
| *multiplicity:* | zero or more |

***dimensionedObject***

DimensionedObjects referenced by Dimensions.

| | |
|---|---|
| *class:* | DimensionedObject |
| *multiplicity:* | zero or more; ordered |

### 11.4.4 MDSchemaOwnsDimensionedObjects

A Multidimensional Schema may own any number of DimensionedObjects.

*Ends*

***schema***

Schema owning DimensionedObjects.

| | |
|---|---|
| *class:* | schema |
| *multiplicity:* | exactly one |
| *aggregation:* | composite |

***dimensionedObject***

DimensionedObjects owned by a Schema.

| | |
|---|---|
| *class:* | DimensionedObject |
| *multiplicity:* | zero or more |

### 11.4.5 MDSchemaOwnsDimensions

A Multidimensional Schema may own any number of Dimensions.

*Ends*

***schema***

Schema owning Dimensions.

| | |
|---|---|
| *class:* | schema |
| *multiplicity:* | exactly one |
| *aggregation:* | composite |

***dimension***

Dimensions owned by a Schema.

| | |
|---|---|
| *class:* | Dimension |
| *multiplicity:* | zero or more |

## 11.5 OCL Representation of Multidimensional Constraints

[C-1] A Dimension may not reference itself as a component, nor as a composite.

**context** Dimension

**inv:** self.component->excludes( self )

**inv:** self.composite->excludes( self )

# XML 12

## 12.1 Overview

XML is rapidly becoming a very important type of data resource, especially in the Internet environment. On the one hand, HTML is evolving to be XML-compliant; in the near future, all HTML documents can be expected to become valid XML documents. On the other hand, XML is quickly becoming the standard format for interchange of data and/or metadata (e.g., XMI). Therefore, XML documents (or streams) representing data and/or metadata can be expected to appear everywhere.

The XML package contains classes and associations that represent common metadata describing XML data resources. It is based on XML 1.0 [XML]. XML Schema is an ongoing activity in the W3C. As future standards are adopted by the W3C on XML Schema, this package will be revised and extended accordingly.

### 12.1.1 Semantics

This section provides a description of the main features of the XML package.

An XML *schema* contains a set of definitions and declarations, in the form of XML *element type* definitions. An XML element type may contain a set of XML *attributes* and/or a *content* model. An attribute can have one of the following defaults: *required*, *implied*, *default*, or *fixed*. The content model can be one of the following types: *empty*, *any*, *mixed*, or *element*. Except for the *empty* content model, a content model consists of constituent parts, particularly *element type references*. The allowed occurrence of the constituents can be one of the following types: *one*, *zero or one*, *zero or more*, or *one or more*.

An *any* content model consists of any element types. A *mixed content* model consists of character data and specified element type references. An *element content* model consists of specified element type references and/or element content models. An element content model can be one of the following types: *choice* or *sequence*.

## 12.2 Organization of the XML Package

The XML package depends on the following packages:

- omg.org::CWM::ObjectModel::Core
- omg.org::CWM::ObjectModel::Instance
- omg.org::CWM::Foundation::DataTypes

The metamodel diagram for the XML package is split into two parts. The first diagram shows the XML classes and associations, while the second shows the inheritance hierarchy.

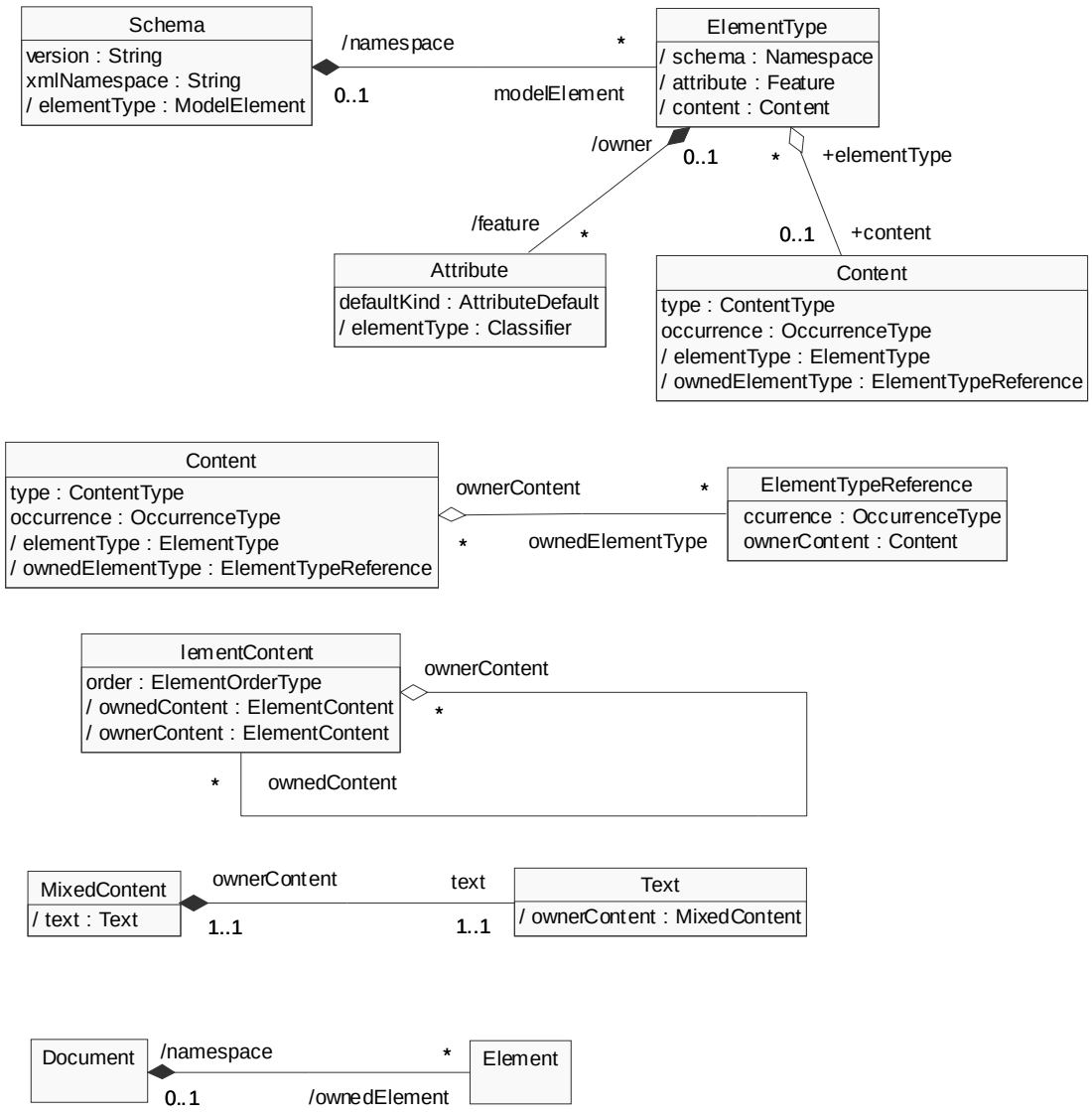


*Figure 12-1* XML Package: Relationships

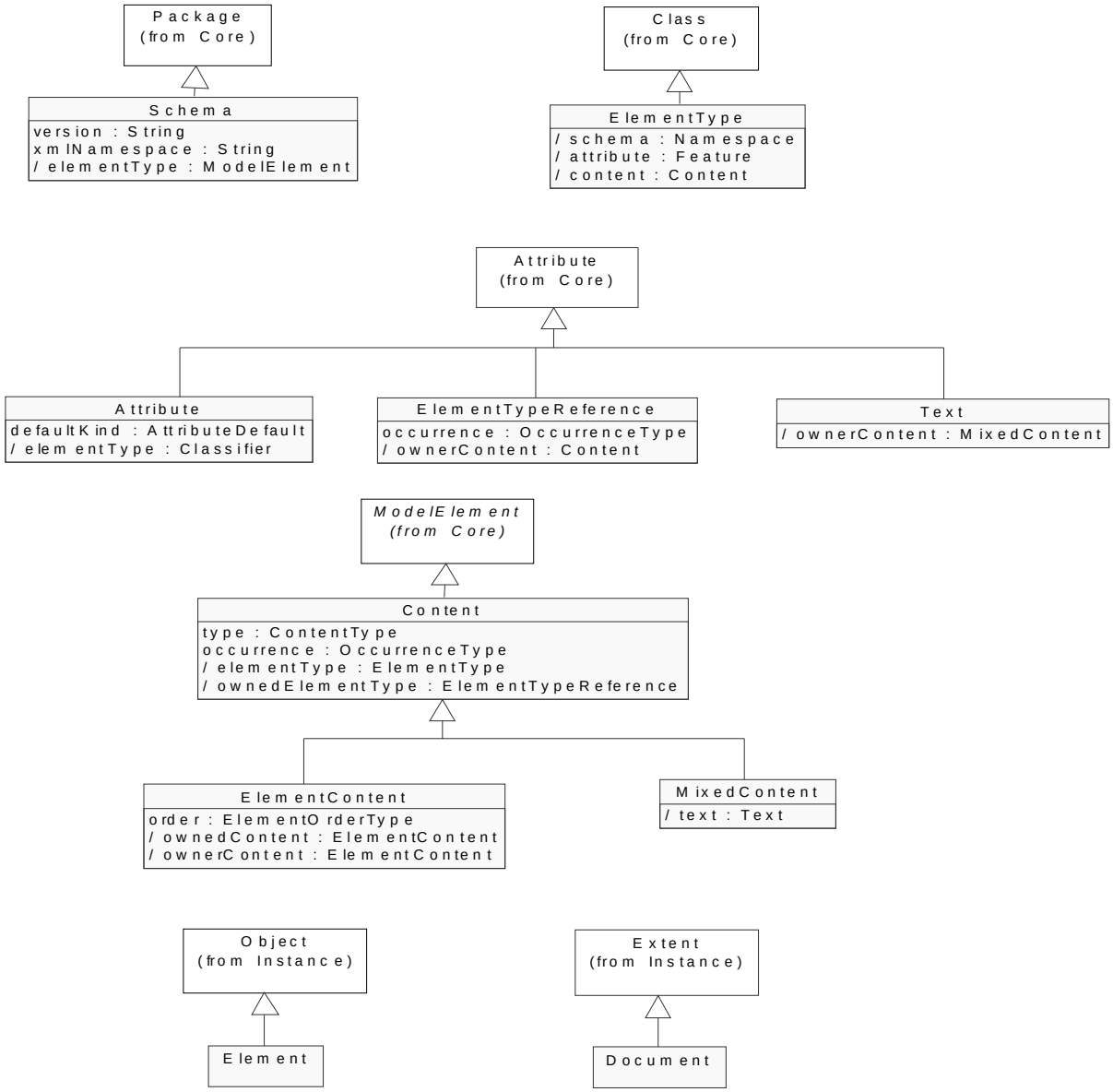


*Figure 12-2* XML Package: Hierarchy

## 12.3 XML Classes

The XML package contains the following classes, in alphabetical order:

- Attribute
- Content
- Document
- Element
- ElementContent
- ElementType
- ElementTypeReference
- MixedContent
- Schema
- Text

### 12.3.1 Attribute

This represents an XML attribute declaration. In XML, attributes are used to associate name-value pairs with elements. Each attribute declaration specifies the name, data type, and default value (if any) of each attribute associated with a given element type.

#### *Superclasses*

org.omg::CWM::ObjectModel::Core::Attribute

### Attributes

#### *defaultKind*

Identifies the kind of attribute default.

| | |
|---|---|
| *type:* | AttributeDefault (xml_required \| xml_implied \| xml_default \| xml_fixed) |
| *multiplicity:* | exactly one |

### References

#### *elementType*

Identifies the ElementType that owns the Attribute.

| | |
|---|---|
| *class:* | Classifier |
| *defined by:* | Classifier-Feature::owner |
| *multiplicity:* | zero or one |
| *inverse:* | ElementType::attribute |

## 12.3.2 Content

This represents the content model of an ElementType. In XML, each document contains one or more elements, the boundaries of which are normally delimited by start-tags and end-tags. The body between the start-tag and end-tag is called the element's content. An element type declaration constrains the element's content.

## Superclasses

ModelElement

## Attributes

### *type*

Identifies the type of the content model.

| | |
|---|---|
| *type:* | ContentType (xml_empty \| xml_any \| xml_mixed \| xml_element) |
| *multiplicity:* | exactly one |

### *occurrence*

Identifies the allowed occurrence of the content constituents.

| | |
|---|---|
| *type:* | OccurrenceType (xml_one \| xml_zeroOrOne \| xml_zeroOrMore, \| xml_oneOrMore) |
| *multiplicity:* | exactly one |

## References

### *elementType*

Identifies the ElementType which owns the Content.

| | |
|---|---|
| *class:* | ElementType |
| *defined by:* | ElementTypeContent::elementType |
| *multiplicity:* | zero or more |
| *inverse:* | ElementType::content |

### *ownedElementType*

Identifies the ElementTypeReferences owned by the Content.

| | |
|---|---|
| *class:* | ElementTypeReference |
| *defined by:* | ContentElementTypeReference::ownedElementType |
| *multiplicity:* | zero or more |
| *inverse:* | ElementTypeReference::ownerContent |

### 12.3.3 Document

This represents an XML document, which is a collection of XML Elements.

#### Superclasses

Extent

#### Contained Elements

Element

### 12.3.4 Element

This represents an instance of an ElementType.

#### Superclasses

Object

### 12.3.5 ElementContent

This represents an element content which contains only ElementTypeReferences. In XML, an element type has element content when elements of that type must contain only child elements (no character data), optionally separated by white space. In this case, the constraint includes a content model that governs the allowed types of the child elements and the order in which they are allowed to appear.

#### Superclasses

Content

### Attributes

***order***

Identifies the order type of the element content.

| | |
|---|---|
| *type:* | ElementOrderType (xml_choice \| xml_sequence) |
| *multiplicity:* | exactly one |

### References

***ownedContent***

Identifies the content owned by the ElementContent.

| | |
|---|---|
| *class:* | ElementContent |
| *defined by:* | OwnedElementContent::ownedContent |
| *multiplicity:* | zero or more |
| *inverse:* | ElementContent::ownerContent |

***ownerContent***

Identifies the content that owns the ElementContent.

| | |
|---|---|
| *class:* | ElementContent |
| *defined by:* | OwnedElementContent::ownerContent |
| *multiplicity:* | zero or more |
| *inverse:* | ElementContent::ownerElement |

### Constraints

An ElementContent may not be its own owner content or owned content, transitive closure.

## 12.3.6 ElementType

This represents an XML element type definition. In XML, each document contains one or more elements. The element structure may, for validation purposes, be constrained using element type and attribute declarations. An element type declaration constrains the element's content.

### Superclasses

Class

### Contained Elements

Attribute

### References

***schema***

Identifies the Schema that owns the ElementType.

| | |
|---|---|
| *class:* | Namespace |
| *defined by:* | Namespace-ModelElement::namespace |
| *multiplicity:* | zero or one |
| *inverse:* | Schema::elementType |

***attribute***

Identifies the Attributes owned by the ElementType.

| | |
|---|---|
| *class:* | Feature |
| *defined by:* | Classifier-Feature::feature |
| *multiplicity:* | zero or more |
| *inverse:* | Attribute::elementType |

***content***

Identifies the content of the ElementType.

| | |
|---|---|
| *class:* | Content |
| *defined by:* | ElementTypeContent::content |
| *multiplicity:* | zero or one |
| *inverse:* | Content::elementType |

## 12.3.7 ElementTypeReference

This represents an XML element type reference. In XML, an element content or a mixed content of an element type may contain references to element type definitions.

### Superclasses

org.omg::CWM::ObjectModel::Core::Attribute

### Attributes

***occurrence***

Identifies the allowed occurrence of the ElementTypeReference.

| | |
|---|---|
| *type:* | OccurrenceType (xml_one \| xml_zeroOrOne \| xml_zeroOrMore \| xml_oneOrMore) |
| *multiplicity:* | exactly one |

### References

***ownerContent***

Identifies the Content that owns the ElementTypeReference.

| | |
|---|---|
| *class:* | Content |
| *defined by:* | ContentElementTypeReference::owner |
| *multiplicity:* | zero or more |
| *inverse:* | Content::ownedElementType |

## 12.3.8 MixedContent

This represents a mixed content of character data and ElementTypeReferences. In XML, an element type has mixed content when elements of that type may contain character data, optionally interspersed with child elements. In this case, the types of the child elements may be constrained, but not their order or their number of occurrences.

### Superclasses

Content

### Contained Elements

Text

### *References*

***text***

Identifies the Text owned by the MixedContent.

| | |
|---|---|
| *class:* | Text |
| *defined by:* | MixedContentText::text |
| *multiplicity:* | exactly one |
| *inverse:* | Text::ownerContent |

## *12.3.9 Schema*

This represents an XML schema which contains a set of definitions and declarations. In XML, this is known as document type definition, or DTD, which provides a grammar for a class of documents.

### *Superclasses*

Package

### *Contained Elements*

ElementType

### Attributes

***version***

Identifies the version of the XML.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

***xmlNamespace***

Identifies the XML namespace of the Schema.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### References

***elementType***

Identifies the ElementTypes owned by the Schema.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | Namespace-ModelElement::ownedElement |
| *multiplicity:* | zero or more |
| *inverse:* | Element::schema |

## 12.3.10 Text

This represents character data. In XML, a mixed content of an element type may contain text.

### Superclasses

org.omg::CWM::ObjectModel::Core::Attribute

### References

***ownerContent***

Identifies the Content that owns the Text.

| | |
|---|---|
| *class:* | MixedContent |
| *defined by:* | MixedContentText::ownerContent |
| *multiplicity:* | exactly one |
| *inverse:* | MixedContent::text |

## 12.4 XML Associations

The XML package contains the following associations, in alphabetical order:

- ContentElementTypeReference
- ElementTypeContent
- MixedContentText
- OwnedElementContent

### 12.4.1 ContentElementTypeReference *protected*

This association relates a Content with its constituent ElementTypeReferences.

### Ends

***ownerContent***

Identifies the owner Content.

| | |
|---|---|
| *class:* | Content |
| *multiplicity:* | zero or more |
| *aggregation:* | shared |

***ownedElementType***

Identifies the owned ElementTypeReferences.

| | |
|---|---|
| *class:* | ElementTypeReference |
| *multiplicity:* | zero or more |

### 12.4.2 ElementTypeContent *protected*

This association relates an ElementType with its Content.

*Ends*

***elementType***

Identifies the ElementType.

| | |
|---|---|
| *class:* | ElementType |
| *multiplicity:* | zero or more |
| *aggregation:* | shared |

***content***

Identifies the Content of the ElementType.

| | |
|---|---|
| *class:* | Content |
| *multiplicity:* | zero or one |

### *12.4.3 MixedContentText* *protected*

This association relates a MixedContent with its Text.

*Ends*

***ownerContent***

Identifies the owner MixedContent.

| | |
|---|---|
| *class:* | MixedContent |
| *multiplicity:* | exactly one |
| *aggregation:* | composite |

***text***

Identifies the Text of the MixedContent.

| | |
|---|---|
| *class:* | Text |
| *multiplicity:* | exactly one |

### *12.4.4 OwnedElementContent* *protected*

This association relates an ElementContent with its constituent ElementContents.

*Ends*

***ownerContent***

Identifies the owner ElementContent.

| | |
|---|---|
| *class:* | ElementContent |
| *multiplicity:* | zero or more |
| *aggregation:* | shared |

***ownedContent***

Identifies the owned ElementContents.

| | |
|---|---|
| *class:* | ElementContent |
| *multiplicity:* | zero or more |

## *12.5 OCL Representation of XML Constraints*

None

# Transformation 13

## 13.1 Overview

A key aspect of data warehousing is to extract, transform, and load data from operational resources to a data warehouse or data mart for analysis. Extraction, transformation, and loading can all be characterized as transformations. In fact, whenever data needs to be converted from one form to another in data warehousing, whether for storage, retrieval, or presentation purposes, transformations are involved. Transformation, therefore, is central to data warehousing.

The Transformation package contains classes and associations that represent common transformation metadata used in data warehousing. It covers basic transformations among all types of data sources and targets: object-oriented, relational, record, multidimensional, XML, OLAP, and data mining.

The Transformation package is designed to enable interchange of common metadata about transformation tools and activities. Specifically it is designed to:

- Relate a transformation with its data sources and targets. These data sources and targets can be of any type (e.g., object-oriented, relational) or granularity (e.g., class, attribute, table, column). They can be persistent (e.g., stored in a relational database) or transient.
- Accommodate both "black box" and "white box" transformations. In the case of "black box" transformations, data sources and targets are related to a transformation and to each other at a coarse-grain level. We know the data sources and targets are related through the transformation, but we don't know how a specific piece of a data source is related to a specific piece of a data target. In the case of "white box" transformations, however, data sources and targets are related to a transformation and to each other at a fine-grain level. We know exactly how a specific piece of a data source is related to a specific piece of a data target through a specific part of the transformation.

- Allow grouping of transformations into logical units. At the functional level, a logical unit defines a single unit of work, within which all transformations must be executed and completed together. At the execution level, logical units can be used to define the execution grouping and sequencing (either explicitly through precedence constraints or implicitly through data dependencies). A key consideration here is that both parallel and sequential executions (or a combination of both) can be accommodated.

The Transformation package assumes the existence of the following packages that represent types of potential data sources or targets: ObjectModel (object-oriented), Relational, Record, Multidimensional, XML, OLAP, and Data Mining. The Transformation package is an integral part of the following packages: OLAP, Data Mining, Warehouse Process, and Warehouse Operation. In particular, the Transformation and Warehouse Process packages together provide metamodel constructs that facilitate scheduling and execution in data warehousing, and the Transformation and Warehouse Operation packages together provide metamodel constructs that enable data lineage in data warehousing.

### *13.1.1 Semantics*

This section provides a description of the main features of the Transformation package, as illustrated in Figure 13-1 on page 287:

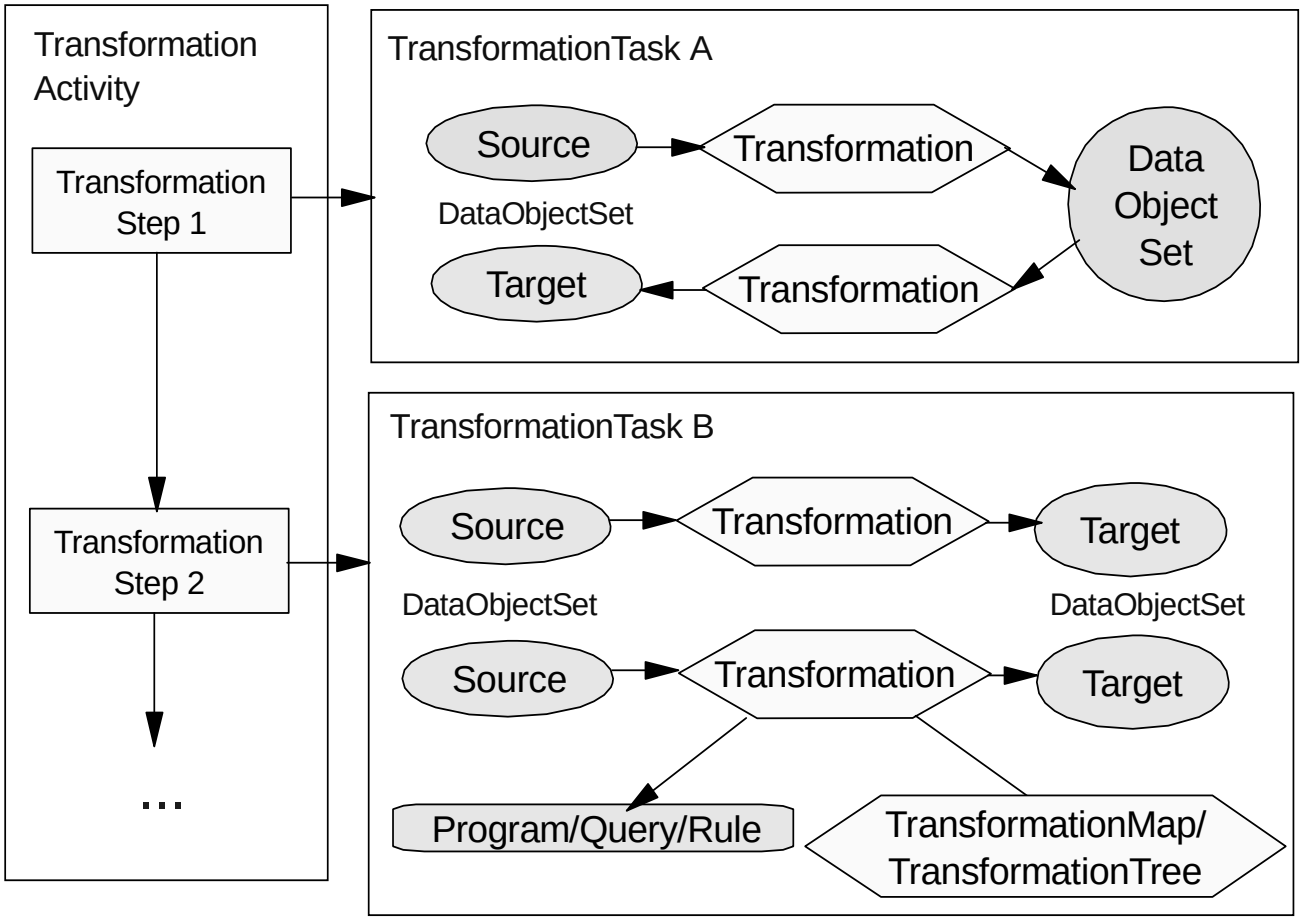


*Figure 13-1* A sample Transformation package.

A *transformation* transforms a set of *source* objects into a set of *target* objects. The elements of a *data object set* can be any ObjectModel *model elements*, but typically are tables, columns, or model elements that represent transient, in memory, objects. Data object sets can be both sources and targets for different transformations. In particular, a given data object set can be the target of one transformation and the source of one or more transformations within the same logical unit. This is often the case with transformation that produce and consume temporary objects.

Transformations allow a wide range of types (and granularity) to be defined for their data sources and targets. For example, the source type of a transformation can be an XML schema while the target type is a column, if the transformation deals with storing an XML document in a column of a relational database. More typically, the source types of a transformation are classes and attributes while the target types are tables and columns, or vice versa, if the transformation deals with converting object data into relational data, or vice versa.

Existing programs, queries, or rules (in fact, any ObjectModel *operations*) can be used to perform a transformation by associating them with the transformation using the *transformation use* dependency.

Transformations can be grouped into logical units. At the functional level, they are grouped into *transformation tasks*, each of which defines a set of transformations that must be executed and completed together - a logical unit of work. At the execution level, *transformation steps* are used to coordinate the flow of control between transformation tasks, with each transformation step executing a single transformation task. The transformation steps are further grouped into *transformation activities*. Within each transformation activity, the execution sequence of its transformation steps are defined either explicitly by using the *step precedence* dependency or *precedence constraint*, or implicitly through data dependency.

There are certain "white-box" transformations which are commonly used and which can relate data sources and targets to a transformation and to each other at a detailed level. These transformations are convenient to use and they provide data lineage at a fine-grain level. One such transformation is the *transformation map* which consists of a set of *classifier maps* that in turn consists of a set of *feature maps or classifier-feature maps*. The other is the *transformation tree*, which represents a transformation as an unary or binary expression tree. For an example usage of the transformation map, please see Figure 13-4 on page 294.

## 13.2 Organization of the Transformation Package

The Transformation package depends on the following packages:

- omg.org::CWM::ObjectModel::Behavioral
- omg.org::CWM::ObjectModel::Core
- omg.org::CWM::Foundation::Expressions
- omg.org::CWM::Foundation::SoftwareDeployment

The CWM uses packages to control complexity and create groupings of logically interrelated classes and associations. The Transformation package is one such package. Within the Transformation package itself, however, the definition of subpackages is purposefully left out to reduce the length and complexity of the fully qualified names of Transformation classes and associations. There are, however, several groupings of classes and associations that form related sets of functionality within the Transformation package. Although separate subpackages have not been created for these functional areas, their identification improves the understandability of the Transformation package.

The Transformation package contains metamodel elements that support the following functions:

- Transformation and data lineage. These classes and associations deal with transformations and their sources, targets, constraints, and operations.
- Transformation grouping and execution. These classes and associations deal with grouping of transformations to form logical units and to define execution sequences.

- Specialized transformations. These classes and associations define specialized, "white box", transformations that are commonly used in data warehousing.

The specific Transformation classes and associations supporting each functional area are delineated in Table 13-1.

**Table 13-1 Functional areas within the Transformation package.**

| Functional Area | Classes | Associations |
|---|---|---|
| Transformation and data lineage | Transformation | TransformationSource<br>TransformationTarget |
| | DataObjectSet | DataObjectSetElement |
| | TransformationUse | |
| Transformation grouping and execution | TransformationTask | TransformationTaskElement<br>InverseTransformationTask |
| | TransformationStep | TransformationStepTask |
| | TransformationActivity | |
| | PrecedenceConstraint | |
| | StepPrecedence | |
| Specialized transformations | TransformationMap | |
| | ClassifierMap | ClassifierMapSource<br>ClassifierMapTarget |
| | FeatureMap | FeatureMapSource<br>FeatureMapTarget |
| | ClassifierFeatureMap | CFMapClassifier<br>CFMapFeature |
| | TransformationTree | |

The metamodel diagram for the Transformation package is split into four parts. The first two diagrams show the Transformation classes and associations, while the last two show the inheritance hierarchy.

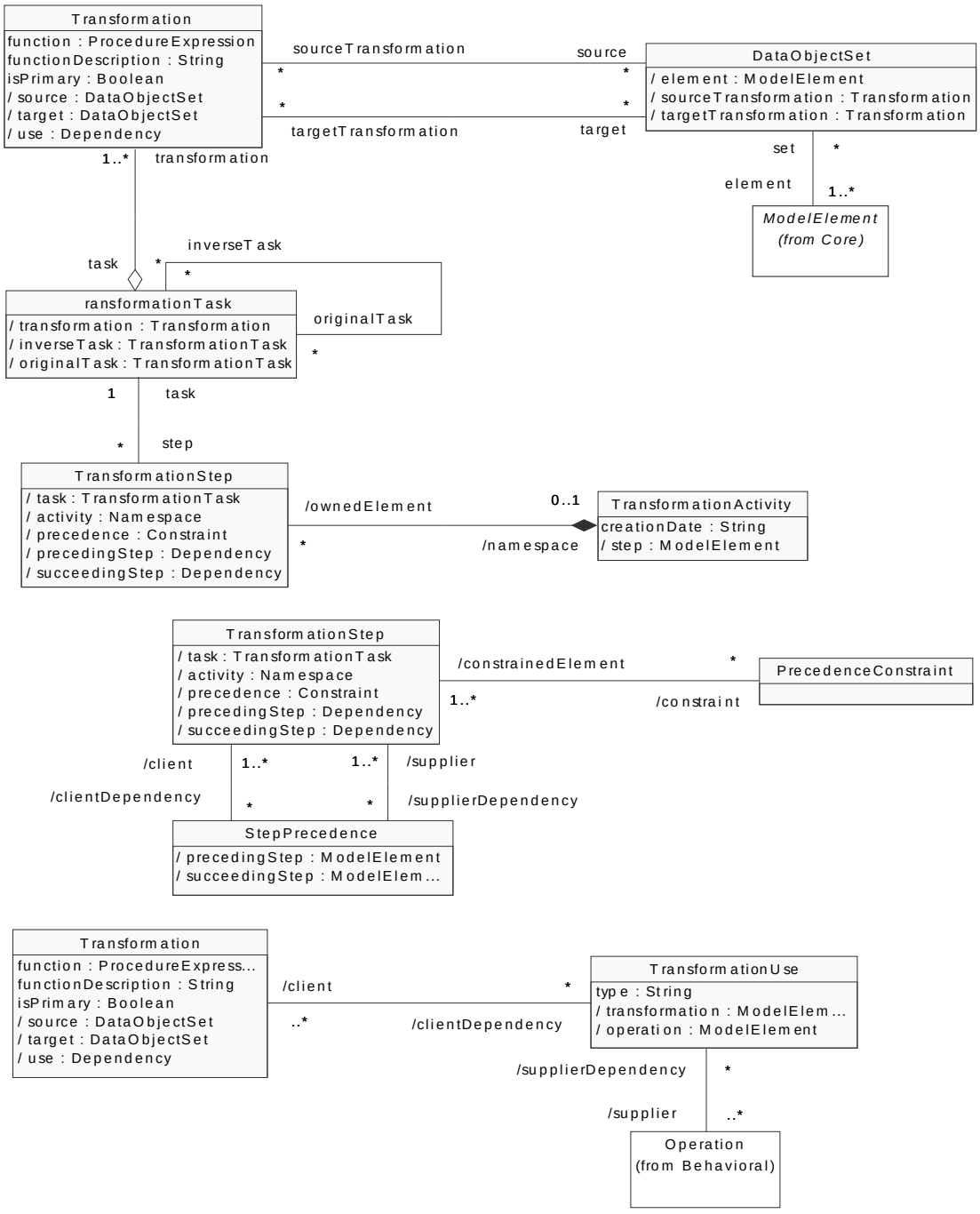


*Figure 13-2* Transformation Package: Relationships - 1

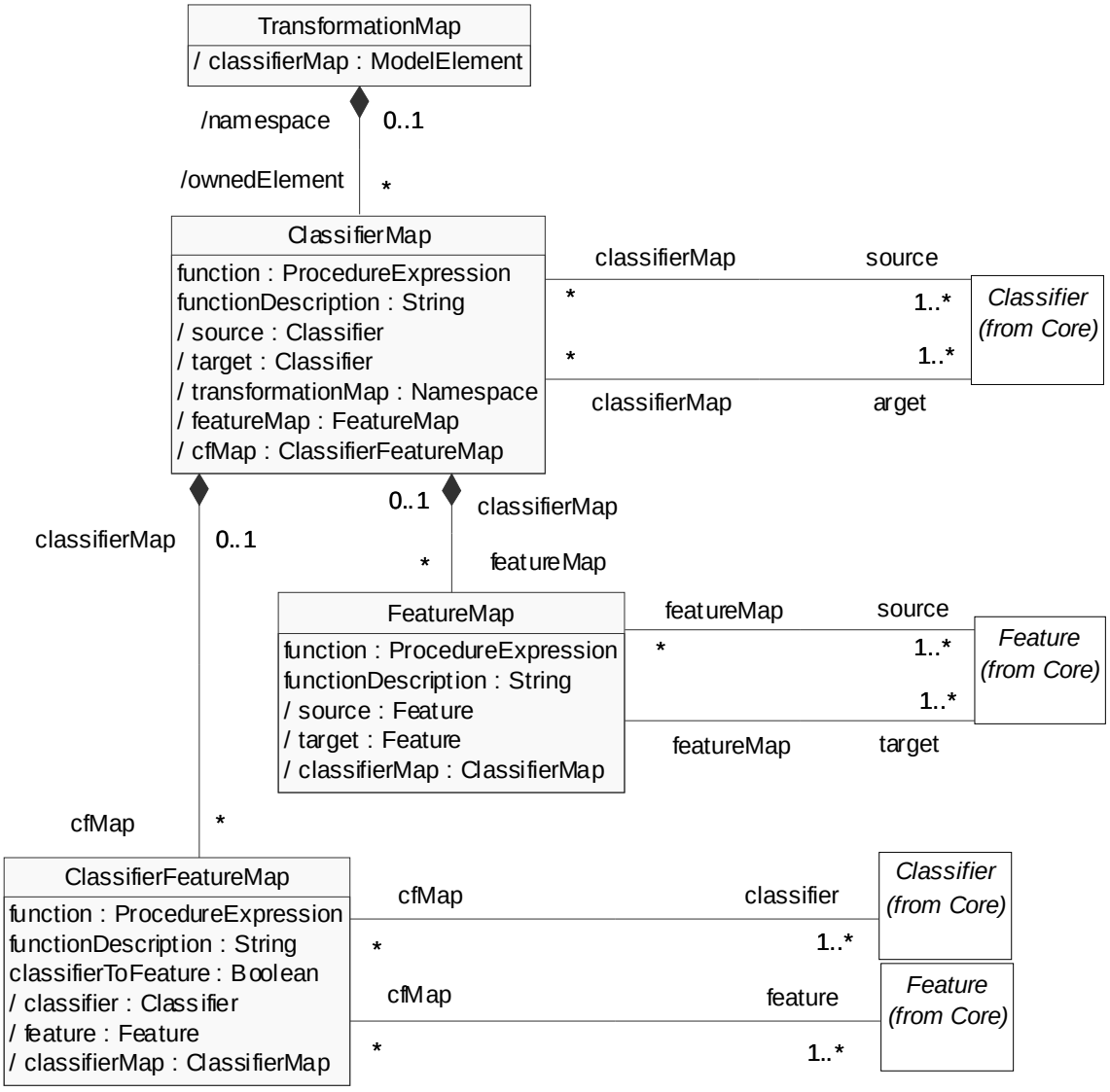


*Figure 13-3* Transformation Package: Relationships - 2

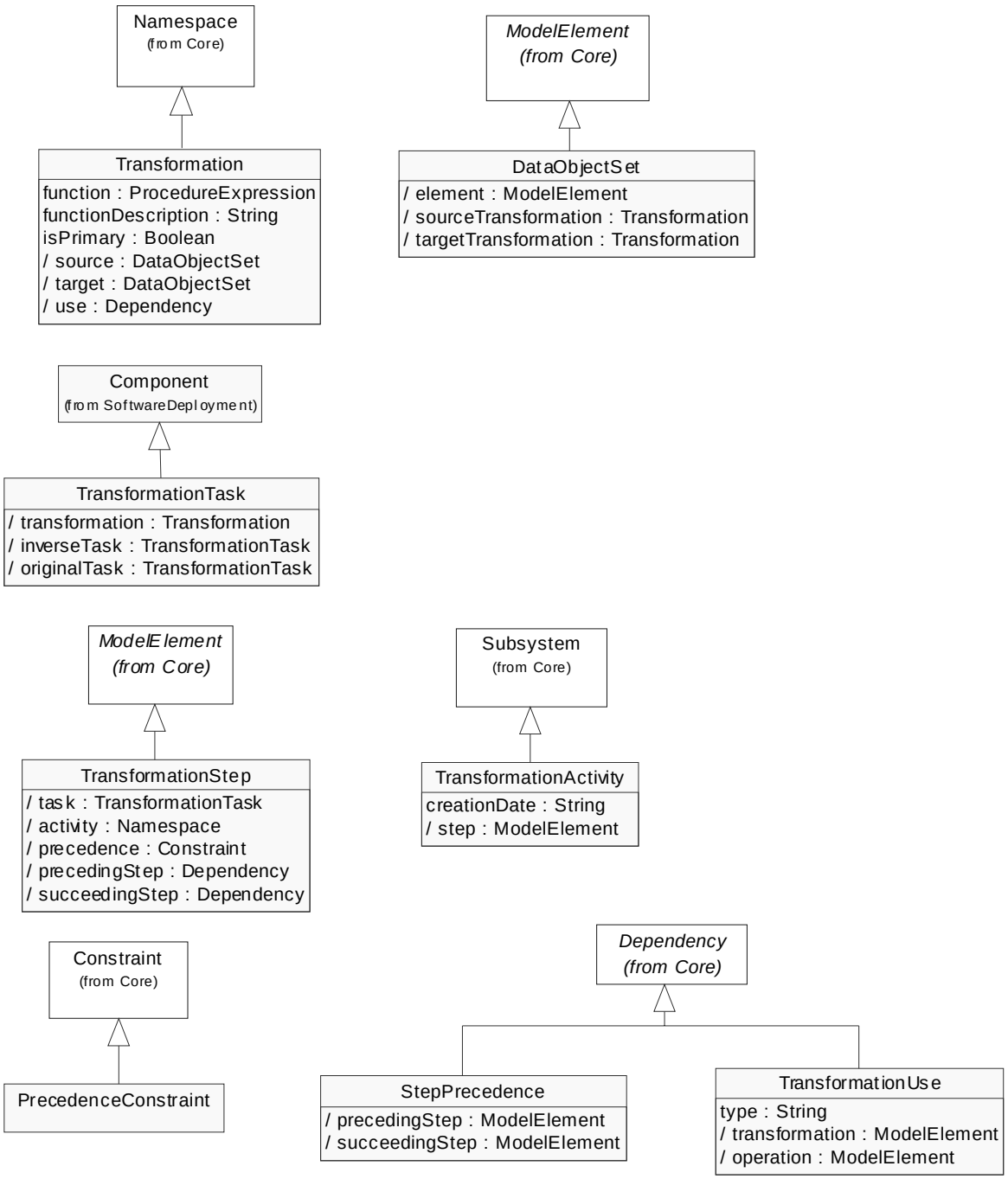


*Figure 13-4* Transformation Package: Hierarchy - 1

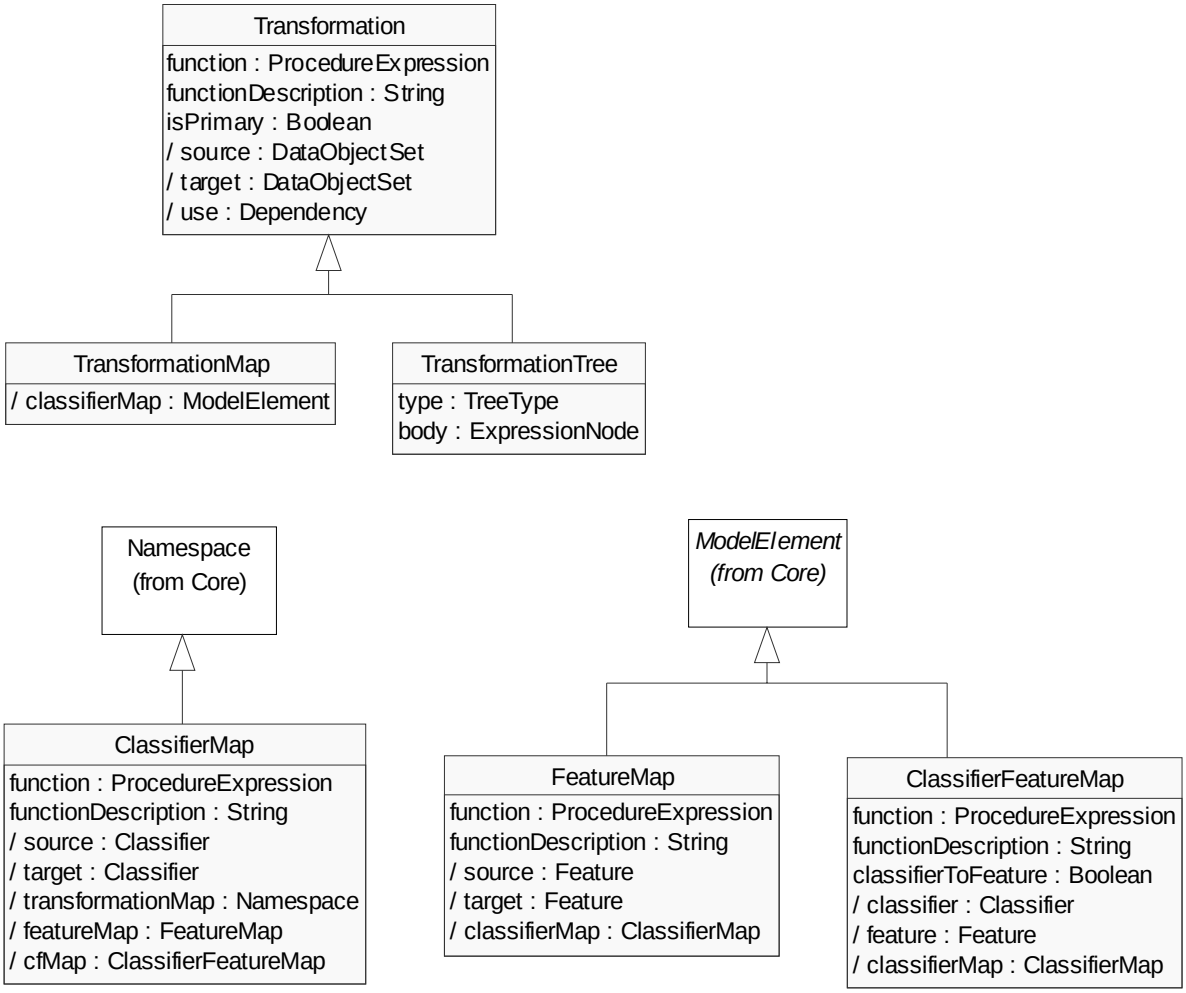


*Figure 13-5* Transformation Package: Hierarchy - 2

## 13.3 Transformation Classes

The Transformation package contains the following classes, in alphabetical order:

- ClassifierFeatureMap
- ClassifierMap
- DataObjectSet
- FeatureMap
- PrecedenceConstraint
- StepPrecedence
- Transformation
- TransformationActivity
- TransformationMap
- TransformationStep
- TransformationTask
- TransformationTree
- TransformationUse

### 13.3.1 ClassifierFeatureMap

This represents a mapping of Classifiers to Features.

#### Superclasses

ModelElement

#### Attributes

***function***

Any code or script for the FeatureMap.

| | |
|---|---|
| *type:* | ProcedureExpression |
| *multiplicity:* | exactly one |

***functionDescription***

A short description for any code or script performed by the FeatureMap.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

***classifierToFeature***

Identifies if the mapping is from Classifiers (source) to Features (target). The default is true.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

### *References*

***classifierMap***

Identifies the ClassifierMap owning the ClassifierFeatureMap.

| | |
|---|---|
| *class:* | ClassifierMap |
| *defined by:* | ClassifierMapToCFMap::classifierMap |
| *multiplicity:* | zero or one |
| *inverse:* | ClassifierMap::cfMap |

***classifier***

Identifies the source/target Classifier of the ClassifierFeatureMap

| | |
|---|---|
| *class:* | Classifier |
| *defined by:* | CFMapClassifier::classifier |
| *multiplicity:* | one or more |

***feature***

Identifies the source/target Features of the ClassifierFeatureMap

| | |
|---|---|
| *class:* | Feature |
| *defined by:* | CFMapFeature::feature |
| *multiplicity:* | one or more |

## *13.3.2 ClassifierMap*

This represents a mapping of source Classifiers to target Classifiers. A ClassifierMap may consists of a group of ClassifierFeatureMaps and/or FeatureMaps.

### *Superclasses*

Namespace

## Contained Elements

ClassifierFeatureMap, FeatureMap

## Attributes

### *function*

Any code or script for the ClassifierMap.

| | |
|---|---|
| *type:* | ProcedureExpression |
| *multiplicity:* | exactly one |

### *functionDescription*

A short description for any code or script performed by the ClassifierMap.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

## References

### *transformationMap*

Identifies the TransformationMap that owns the ClassifierMap.

| | |
|---|---|
| *class:* | Namespace |
| *defined by:* | Namespace-ModelElement::namespace |
| *multiplicity:* | zero or one |
| *inverse:* | TransformationMap::classifierMap |

### *source*

Identifies the source Classifiers of the ClassifierMap

| | |
|---|---|
| *class:* | Classifier |
| *defined by:* | ClassifierMapSource::source |
| *multiplicity:* | one or more |

***target***

Identifies the target Classifiers of the ClassifierMap

| | |
|---|---|
| *class:* | Classifier |
| *defined by:* | ClassifierMapTarget::target |
| *multiplicity:* | one or more |

***featureMap***

Identifies the FeatureMaps owned by the ClassifierMap.

| | |
|---|---|
| *class:* | FeatureMap |
| *defined by:* | ClassifierMapToFeatureMap::featureMap |
| *multiplicity:* | zero or more |
| *inverse:* | FeatureMap::classifierMap |

***cfMap***

Identifies the ClassifierFeatureMaps owned by the ClassifierMap.

| | |
|---|---|
| *class:* | ClassifierFeatureMap |
| *defined by:* | ClassifierMapToCFMap::cfMap |
| *multiplicity:* | zero or more |
| *inverse:* | ClassifierFeatureMap::classifierMap |

## 13.3.3 DataObjectSet

This represents a set of data objects that can be the source or target of a Transformation.

### Superclasses

ModelElement

### References

***element***

Identifies the elements in the DataObjectSet

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | DataObjectSetElement::element |
| *multiplicity:* | one or more |

***sourceTransformation***

Identifies the Transformation of the source

| | |
|---|---|
| *class:* | Transformation |
| *defined by:* | TransformationSource::sourceTransformation |
| *multiplicity:* | zero or more |
| *inverse:* | Transformation::source |

***targetTransformation***

Identifies the Transformation of the target

| | |
|---|---|
| *class:* | Transformation |
| *defined by:* | TransformationTarget::targetTransformation |
| *multiplicity:* | zero or more |
| *inverse:* | Transformation::target |

### *13.3.4 FeatureMap*

This represents a mapping of source Features to target Features.

#### *Superclasses*

ModelElement

#### *Attributes*

***function***

Any code or script for the FeatureMap.

| | |
|---|---|
| *type:* | ProcedureExpression |
| *multiplicity:* | exactly one |

***functionDescription***

A short description for any code or script performed by the FeatureMap.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### *References*

***classifierMap***

Identifies the ClassifierMap owning the FeatureMap.

| | |
|---|---|
| *class:* | ClassifierMap |
| *defined by:* | ClassifierMapToFeatureMap::classifierMap |
| *multiplicity:* | zero or one |
| *inverse:* | ClassifierMap::featureMap |

***source***

Identifies the source Features of the FeatureMap

| | |
|---|---|
| *class:* | Feature |
| *defined by:* | FeatureMapSource::source |
| *multiplicity:* | one or more |

***target***

Identifies the target Features of the FeatureMap

| | |
|---|---|
| *class:* | Feature |
| *defined by:* | FeatureMapTarget::target |
| *multiplicity:* | one or more |

## *13.3.5 PrecedenceConstraint*

This is used to define order-of-execution constraint among TransformationSteps. It may be used independent of or in conjunction with StepPrecedence.

### *Superclasses*

Constraint

### 13.3.6 StepPrecedence

This is used to define explicit order-of-execution relationships among TransformationSteps. It may be used independent of or in conjunction with PrecedenceConstraint

#### Superclasses

Dependency

#### References

***precedingStep***

Identifies the preceding TransformationStep that the StepPrecedence dependency is for.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | Dependency-ModelElement::supplier |
| *multiplicity:* | one or more |
| *inverse:* | TransformationStep::succeedingStep |

***succeedingStep***

Identifies the succeeding TransformationStep that the StepPrecedence dependency is for.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | Dependency-ModelElement::client |
| *multiplicity:* | one or more |
| *inverse:* | TransformationStep::precedingStep |

#### Constraints

The preceding step and succeeding step must not be the same. [C-1]

### 13.3.7 Transformation

This represents a transformation from a set of sources to a set of targets.

If a model already exists for the object that performs the Transformation, then the model can be related to the Transformation via a TransformationUse dependency.

## Superclasses

Namespace

## Attributes

### *function*

Any code or script for the Transformation.

| | |
|---|---|
| *type:* | ProcedureExpression |
| *multiplicity:* | exactly one |

### *functionDescription*

A short description for any code or script performed by the Transformation.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### *isPrimary*

This Transformation is the primary transformation for the associated TransformationTask.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

## References

### *source*

Identifies the sources of the Transformation.

| | |
|---|---|
| *class:* | DataObjectSet |
| *defined by:* | TransformationSource::source |
| *multiplicity:* | zero or more |
| *inverse:* | DataObjectSet::sourceTransformation |

### *target*

Identifies the targets of the Transformation.

| | |
|---|---|
| *class:* | DataObjectSet |

| | |
|---|---|
| *defined by:* | TransformationTarget::target |
| *multiplicity:* | zero or more |
| *inverse:* | DataObjectSet::targetTransformation |

***use***

Identifies the TransformationUse dependency.

| | |
|---|---|
| *class:* | Dependency |
| *defined by:* | Dependency-ModelElement::clientDependency |
| *multiplicity:* | zero or more |
| *inverse:* | TransformationUse::transformation |

### 13.3.8 TransformationActivity

This represents a transformation activity. Each TransformationActivity consists of a set of TransformationSteps.

#### Superclasses

Subsystem

#### Contained Elements

TransformationStep

#### Attributes

***creationDate***

When the TransformationActivity was created.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### References

***step***

Identifies the TransformationSteps owned by the TransformationActivity.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | Namespace-ModelElement::ownedElement |
| *multiplicity:* | zero or more |
| *inverse:* | TransformationStep::activity |

## 13.3.9 TransformationMap

This represents a specialized Transformation which consists of a group of ClassifierMaps.

### Superclasses

Transformation

### Contained Elements

ClassifierMap

### References

***classifierMap***

Identifies the ClassifierMaps owned by the TransformationMap.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | Namespace-ModelElement::ownedElement |
| *multiplicity:* | zero or more |
| *inverse:* | ClassifierMap::transformationMap |

## 13.3.10 TransformationStep

This represents the usage of a TransformationTask in a TransformationActivity. A TransformationStep relates to one TransformationTask.

TransformationSteps are used to coordinate the flow of control between their TransformationTasks. Ordering of the TransformationSteps are defined using the PrecedenceConstrainedBy dependency.

## Superclasses

ModelElement

## References

***task***

Identifies the TransformationTask that the TransformationStep performs.

| | |
|---|---|
| *class:* | TransformationTask |
| *defined by:* | TransformationStepTask::task |
| *multiplicity:* | exactly one |
| *inverse:* | TransformationTask::step |

***activity***

Identifies the TransformationActivity that owns the TransformationStep.

| | |
|---|---|
| *class:* | Namespace |
| *defined by:* | Namespace-ModelElement::namespace |
| *multiplicity:* | zero or one |
| *inverse:* | TransformationActivity::step |

***precedence***

Identifies the PrecedenceConstraint.

| | |
|---|---|
| *class:* | Constraint |
| *defined by:* | Constraint-ModelElement::constraint |
| *multiplicity:* | zero or more |

***precedingStep***

Identifies the preceding StepPrecedence dependency.

| | |
|---|---|
| *class:* | Dependency |
| *defined by:* | Dependency-ModelElement::clientDependency |
| *multiplicity:* | one or more |
| *inverse:* | StepPrecedence::succeedingStep |

***succeedingStep***

Identifies the succeeding StepPrecedence dependency.

| | |
|---|---|
| *class:* | Dependency |
| *defined by:* | Dependency-ModelElement::supplierDependency |
| *multiplicity:* | one or more |
| *inverse:* | StepPrecedence::precedingStep |

## 13.3.11 TransformationTask

This represents a set of Transformations that must be executed together as a single task (logical unit).

A TransformationTask may have an inverse task. A transformation task that maps a source set "A" into a target set "B" can be reversed by the inverse transformation task that maps "B" into "A".

### Superclasses

Component

### References

***transformation***

Identifies the Transformations that belong to the TransformationTask.

| | |
|---|---|
| *class:* | Transformation |
| *defined by:* | TransformationTaskElement::transformation |
| *multiplicity:* | one or more |

***inverseTask***

Identifies the inverse TransformationTask.

| | |
|---|---|
| *class:* | TransformationTask |
| *defined by:* | InverseTransformationTask::inverseTask |
| *multiplicity:* | zero or more |
| *inverse:* | TransformationTask::originalTask |

***originalTask***

Identifies the original TransformationTask.

| | |
|---|---|
| *class:* | TransformationTask |
| *defined by:* | InverseTransformationTask::originalTask |
| *multiplicity:* | zero or more |
| *inverse:* | TransformationTask::inverseTask |

### *Constraints*

A TransformationTask may not be its own inverse task [C-2] or original task [C-3].

## *13.3.12 TransformationTree*

This represents a specialized Transformation which can be modeled as an expression tree.

### *Superclasses*

Transformation

### *Attributes*

***type***

Identifies the type of TransformationTree, which can be unary or binary.

| | |
|---|---|
| *type:* | TreeType (tfm_unary \| tfm_binary) |
| *multiplicity:* | exactly one |

***body***

Identifies the expression tree that embodies the TransformationTree.

| | |
|---|---|
| *type:* | ExpressionNode |
| *multiplicity:* | exactly one |

## 13.3.13 TransformationUse

This is a specialized dependency used to associate a Transformation to the model of an existing object (e.g., program, query, or rule) that performs the transformation.

### Superclasses

Usage

### Attributes

***type***

Identifies the type of object that can perform the transformation.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### References

***transformation***

Identifies the Transformation that the TransformationUse dependency is for.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | Dependency-ModelElement::client |
| *multiplicity:* | one or more |
| *inverse:* | Transformation::use |

***operation***

Identifies the Operation that the TransformationUse dependency is on.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | Dependency-ModelElement::supplier |
| *multiplicity:* | one or more |

## 13.4 Transformation Associations

The Transformation package contains the following associations, in alphabetical order:

- CFMapClassifier
- CFMapFeature
- ClassifierMapSource
- ClassifierMapTarget
- ClassifierMapToCFMap
- ClassifierMapToFeatureMap
- DataObjectSetElement
- FeatureMapSource
- FeatureMapTarget
- InverseTransformationTask
- TransformationSource
- TransformationStepTask
- TransformationTarget
- TransformationTaskElement

### 13.4.1 CFMapClassifier

This association relates a ClassifierFeatureMap to its source/target Classifiers.

#### Ends

***cfMap***

Identifies the ClassifierFeatureMap

| | |
|---|---|
| *class:* | ClassifierFeatureMap |
| *multiplicity:* | zero or more |

***classifier***

Identifies the source/target Classifiers of the ClassifierFeatureMap

| | |
|---|---|
| *class:* | Classifier |
| *multiplicity:* | one or more |

### 13.4.2 CFMapFeature

This association relates a ClassifierFeatureMap to its source/target Features.

*Ends*

***cfMap***

Identifies the ClassifierFeatureMap

| | |
|---|---|
| *class:* | ClassifierFeatureMap |
| *multiplicity:* | zero or more |

***feature***

Identifies the source/target Features of the ClassifierFeatureMap

| | |
|---|---|
| *class:* | Feature |
| *multiplicity:* | one or more |

### *13.4.3 ClassifierMapSource*

This association relates a ClassifierMap to its source Classifiers.

*Ends*

***classifierMap***

Identifies the ClassifierMap

| | |
|---|---|
| *class:* | ClassifierMap |
| *multiplicity:* | zero or more |

***source***

Identifies the source Classifiers of the ClassifierMap

| | |
|---|---|
| *class:* | Classifier |
| *multiplicity:* | one or more |

### *13.4.4 ClassifierMapTarget*

This association relates a ClassifierMap to its target Classifiers.

### Ends

***classifierMap***

Identifies the ClassifierMap

| | |
|---|---|
| *class:* | ClassifierMap |
| *multiplicity:* | zero or more |

***target***

Identifies the target Classifiers of the ClassifierMap

| | |
|---|---|
| *class:* | Classifier |
| *multiplicity:* | one or more |

## 13.4.5 ClassifierMapToCFMap *derived protected*

This association relates a ClassifierMap to its ClassifierFeatureMaps.

### Ends

***classifierMap***

Identifies the owning ClassifierMap

| | |
|---|---|
| *class:* | ClassifierMap |
| *multiplicity:* | zero or one |

***cfMap***

Identifies the owned ClassifierFeatureMaps

| | |
|---|---|
| *class:* | ClassifierFeatureMap |
| *multiplicity:* | zero or more |

### Derivation

This association is derived from the Namespace-ModelElement association. All ownedElement ends of the association must be ClassifierFeatureMaps. [C-4]

## 13.4.6 ClassifierMapToFeatureMap *derived protected*

This association relates a ClassifierMap to its FeatureMaps.

### Ends

***classifierMap***

Identifies the owning ClassifierMap

| | |
|---|---|
| *class:* | ClassifierMap |
| *multiplicity:* | zero or one |

***featureMap***

Identifies the owned FeatureMaps

| | |
|---|---|
| *class:* | FeatureMap |
| *multiplicity:* | zero or more |

### Derivation

This association is derived from the Namespace-ModelElement association. All ownedElement ends of the association must be FeatureMaps. [C-5]

## 13.4.7 DataObjectSetElement

This association relates a DataObjectSet to its elements.

### Ends

***set***

Identifies the DataObjectSet

| | |
|---|---|
| *class:* | DataObjectSet |
| *multiplicity:* | zero or more |

***element***

Identifies the elements in the DataObjectSet

| | |
|---|---|
| *class:* | ModelElement |
| *multiplicity:* | one or more |

## 13.4.8 FeatureMapSource

This association relates an FeatureMap to its source Features.

*Ends*

***featureMap***

Identifies the FeatureMap

| | |
|---|---|
| *class:* | FeatureMap |
| *multiplicity:* | zero or more |

***source***

Identifies the source Features of the FeatureMap

| | |
|---|---|
| *class:* | Feature |
| *multiplicity:* | one or more |

### *13.4.9 FeatureMapTarget*

This association relates an FeatureMap to its target Features.

*Ends*

***featureMap***

Identifies the FeatureMap

| | |
|---|---|
| *class:* | FeatureMap |
| *multiplicity:* | zero or more |

***target***

Identifies the target Features of the FeatureMap

| | |
|---|---|
| *class:* | Feature |
| *multiplicity:* | one or more |

### *13.4.10 InverseTransformationTask* *protected*

This association relates a TransformationTask to its inverse. A transformation task that maps a source set "A" into a target set "B" can be reversed by the inverse transformation task that maps "B" into "A"

*Ends*

***originalTask***

Identifies the original TransformationTask

| | |
|---|---|
| *class:* | TransformationTask |
| *multiplicity* | zero or more |

***inverseTask***

Identifies the inverse TransformationTask

| | |
|---|---|
| *class:* | TransformationTask |
| *multiplicity* | zero or more |

### *13.4.11 TransformationSource* *protected*

This association relates a Transformation to its sources.

*Ends*

***sourceTransformation***

Identifies the Transformation

| | |
|---|---|
| *class:* | Transformation |
| *multiplicity:* | zero or more |

***source***

Identifies the sources of the Transformation

| | |
|---|---|
| *class:* | DataObjectSet |
| *multiplicity:* | zero or more |

### *13.4.12 TransformationStepTask*

This association relates a TransformationStep to its TransformationTask.

*Ends*

***step***

Identifies the TransformationStep

| | |
|---|---|
| *class:* | TransformationStep |
| *multiplicity* | zero or more |

***task***

Identifies the TransformationTask

| | |
|---|---|
| *class:* | TransformationTask |
| *multiplicity* | exactly one |

### *13.4.13 TransformationTarget* *protected*

This association relates a Transformation to its targets.

*Ends*

***targetTransformation***

Identifies the Transformation

| | |
|---|---|
| *class:* | Transformation |
| *multiplicity:* | zero or more |

***target***

Identifies the targets of the Transformation

| | |
|---|---|
| *class:* | DataObjectSet |
| *multiplicity:* | zero or more |

### *13.4.14 TransformationTaskElement*

This association relates a TransformationTask to its Transformations.

*Ends*

***task***

Identifies the TransformationTask

| | |
|---|---|
| *class:* | TransformationTask |
| *multiplicity:* | zero or more |
| *aggregation:* | shared |

***transformation***

Identifies the Transformations

| | |
|---|---|
| *class:* | Transformation |
| *multiplicity:* | one or more |

## 13.5 OCL Representation of Transformation Constraints

[C-1] The preceding step and succeeding step of StepPrecedence must not be the same.

**context** StepPrecedence

**inv**: self.precedingStep->forAll( p | self.succeedingStep->forAll( q | p <> q ) )

[C-2] A TransformationTask may not be its own inverse task.

**context** TransformationTask

**inv**: self.inverseTask->forAll( p | p <> self )

[C-3] A TransformationTask may not be its own original task.

**context** TransformationTask

**inv**: self.originalTask->forAll( p | p <> self )

[C-4] The ClassifierMapToCFMap association is derived from the Namespace-ModelElement association. All ownedElement ends of the association must be ClassifierFeatureMaps.

**context** ClassifierMapToCFMap

**inv** Namespace-ModelElement.allInstances.select( ownedElement.oclIsKindOf( ClassifierFeatureMap ) )

[C-5] The ClassifierMapToFeatureMap association is derived from the Namespace-ModelElement association. All ownedElement ends of the association must be FeatureMaps.

**context** ClassifierMapToFeatureMap

**inv** Namespace-ModelElement.allInstances.select( ownedElement.oclIsKindOf( FeatureMap ) )

# OLAP 14

## 14.1 Overview

Online Analytical Processing (OLAP) is a class of analytic application software that exposes business data in a multidimensional format. This multidimensional format usually includes the consolidation of data drawn from multiple and diverse information sources. Unlike more traditionally structured representations (e.g., the tabular format of a relational database), the multidimensional orientation is a more natural expression of the way business enterprises view their strategic data. For example, an analyst might use an OLAP application to examine total sales revenue by product and geographic region over time, or, perhaps, compare sales margins for the same fiscal periods of two consecutive years. The ultimate objective of OLAP is the efficient construction of analytical models that transform raw business data into strategic business insight.

There are many ways to implement OLAP. Most OLAP systems are constructed using OLAP server tools that enable logical OLAP structures to be built on top of a variety of physical database systems, such as relational or native multidimensional databases. The following features are generally found in most OLAP systems:

- Multidimensional representation of business data.
- Upward consolidation of multidimensional data in a hierarchical manner, possibly with the application of specialized processing rules.
- The ability to navigate a hierarchy from a consolidated value to the lower level values forming it.
- Support for time-series analysis; i.e., OLAP users are generally concerned with data and consolidations at specific points in time -- By date, week, quarter, etc.
- Support for modeling and scenario analysis -- A user should be able to apply arbitrary “what-if” analyses to a result set without affecting the stored information.
- Consistent response times, regardless of how queries are formulated -- This is critical for effective analysis and modeling.

OLAP applications integrate well into the data warehousing environment, because a data warehouse provides relatively clean and stable data stores to drive the OLAP application. These data stores are usually maintained in relational tables that can be read directly by OLAP tools or loaded into OLAP servers. These relational tables are often structured in a manner that reveals the inherent dimensionality of the data (such as the ubiquitous Star and Snowflake schemas). Also, the data transformation and mapping services provided by a data warehouse can be used to supply OLAP systems with both metadata and data. Transformation-related metadata can be used to track the lineage of consolidated OLAP data back to its various sources.

## 14.2 Objectives of the OLAP Package

The primary objectives of the CWM OLAP package are:

- Define a metamodel of essential OLAP concepts common to most OLAP systems.
- Provide a facility whereby instances of the OLAP metamodel are mapped to deployment-capable structures (i.e., models of physical data resources, such as the CWM Relational and Multidimensional packages).
- Ensure that navigation through the logical OLAP model hierarchy and its various resource models is always performed in a uniform manner (i.e., by defining a standard usage of the CWM Transformation package as a means of implementing these mappings).
- Leverage services provided by other CWM packages, where appropriate (e.g., use the CWM Foundation package to supply a standard representation of expressions).

## 14.3 Organization of the OLAP Package

### 14.3.1 Dependencies

The OLAP package depends on the following packages:

org.omg::CWM::ObjectModel::Core

org.omg::CWM::Foundation::Expressions

org.omg::CWM::Analysis::Transformation

### 14.3.2 Major Classes and Associations

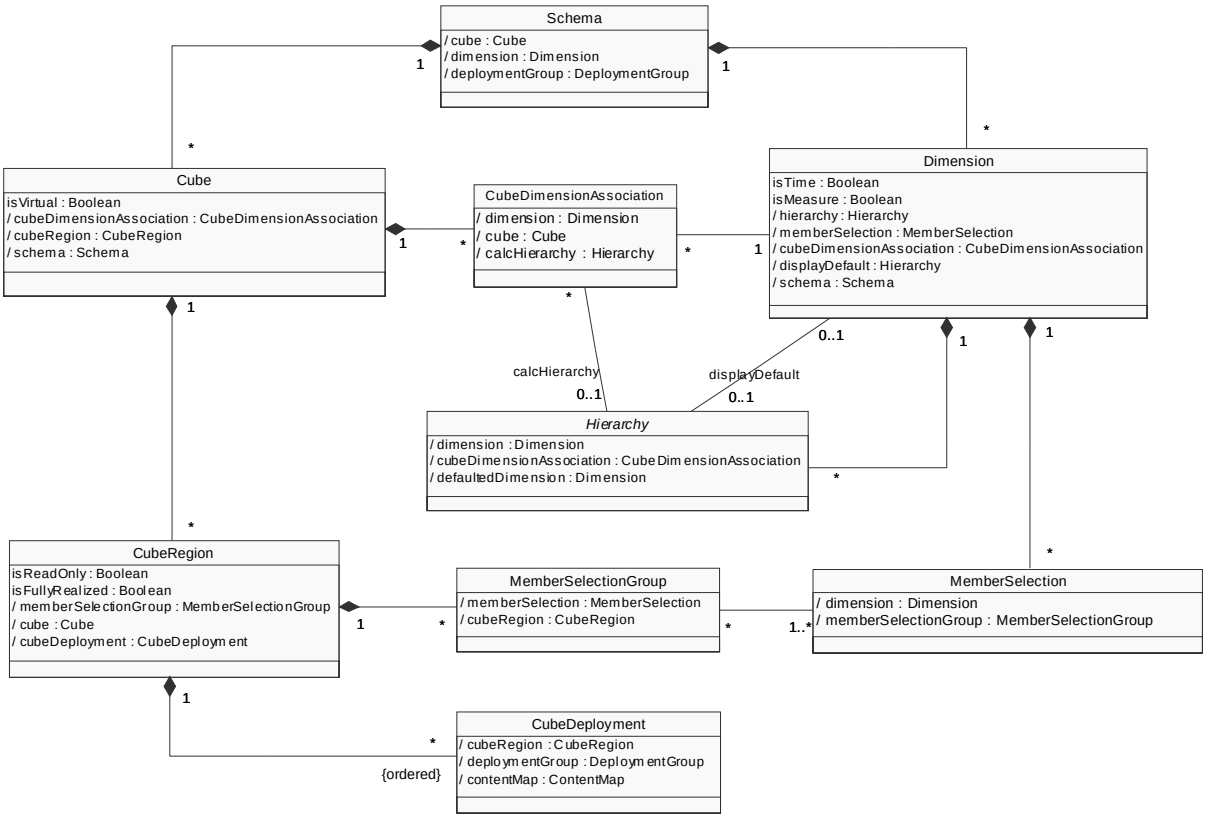


*Figure 14-1* OLAP Metamodel: Major Classes and Associations

The major classes and associations of the OLAP metamodel are shown in Figure 14-1.

Schema is the logical container of all elements comprising an OLAP model. It is the root element of the model hierarchy and marks the entry point for navigating OLAP models.

A Schema contains Dimensions and Cubes. A Dimension is an ordinate within a multidimensional structure and consists of a list of unique values (i.e., members) that share a common semantic meaning within the domain being modeled. Each member designates a unique position along its ordinate.

A Cube is a collection of analytic values (i.e., measures) that share the same dimensionality. This dimensionality is specified by a set of unique Dimensions from the Schema. Each unique combination of members in the Cartesian product of the Cube's Dimensions identifies precisely one data cell within a multidimensional structure.

CubeDimensionAssociation relates a Cube to its defining Dimensions. Features relevant to Cube-Dimension relationships (e.g., calcHierarchy) are exposed by this class.

A Dimension has zero or more Hierarchies. A Hierarchy is an organizational structure that describes a traversal pattern through a Dimension, based on parent/child relationships between members of a Dimension. Hierarchies are used to define both navigational and consolidation/computational paths through the Dimension (i.e., a value associated with a child member is aggregated by one or more parents). For example, a Time Dimension with a base periodicity of days might have a Hierarchy specifying the consolidation of days into weeks, weeks into months, months into quarters, and quarters into years.

A specific Hierarchy may be designated as the default Hierarchy for display purposes (e.g., a user interface that displays the Dimension as a hierarchical tree of members). CubeDimensionAssociation can also identify a particular Hierarchy as the default Hierarchy for consolidation calculations performed on the Cube.

Dimensions and Hierarchies are described further in Section 13.3.3.

MemberSelection models mechanisms capable of partitioning a Dimension's collection of members. For example, consider a Geography Dimension with members representing cities, states, and regions. An OLAP client interested specifically in cities might define an instance of MemberSelection that extracts the city members.

CubeRegion models a sub-unit of a Cube that is of the same dimensionality as the Cube itself. Each "dimension" of a CubeRegion is represented by a MemberSelection of the corresponding Dimension of the Cube. Each MemberSelection may define some subset of its Dimension's members.

CubeRegions are used to implement Cubes. A Cube may be realized by a set of CubeRegions that map portions of the logical Cube to physical data sources. The MemberSelections defining CubeRegions can also be grouped together via MemberSelectionGroups, enabling the definition of CubeRegions with specific semantics. For example, one can specify a CubeRegion containing only the "input level" data cells of a Cube.

A CubeRegion may own any number of CubeDeployments. CubeDeployment is a metaclass that represents an implementation strategy for a multidimensional structure. The ordering of the CubeDeployment classes may optionally be given some implementation-specific meaning (e.g., desired order of selection of several possible deployment strategies, based on optimization considerations).

### *14.3.3 Dimension and Hierarchy*

Figure 14-2 shows Dimension and Hierarchy, along with several other classes that model hierarchical structuring and deployment mappings.

## *Dimension*

The OLAP metamodel defines two special types of Dimension: Time and Measure.

A Time Dimension provides a means of representing time-series data within a multidimensional structure. The members of a Time Dimension usually define some

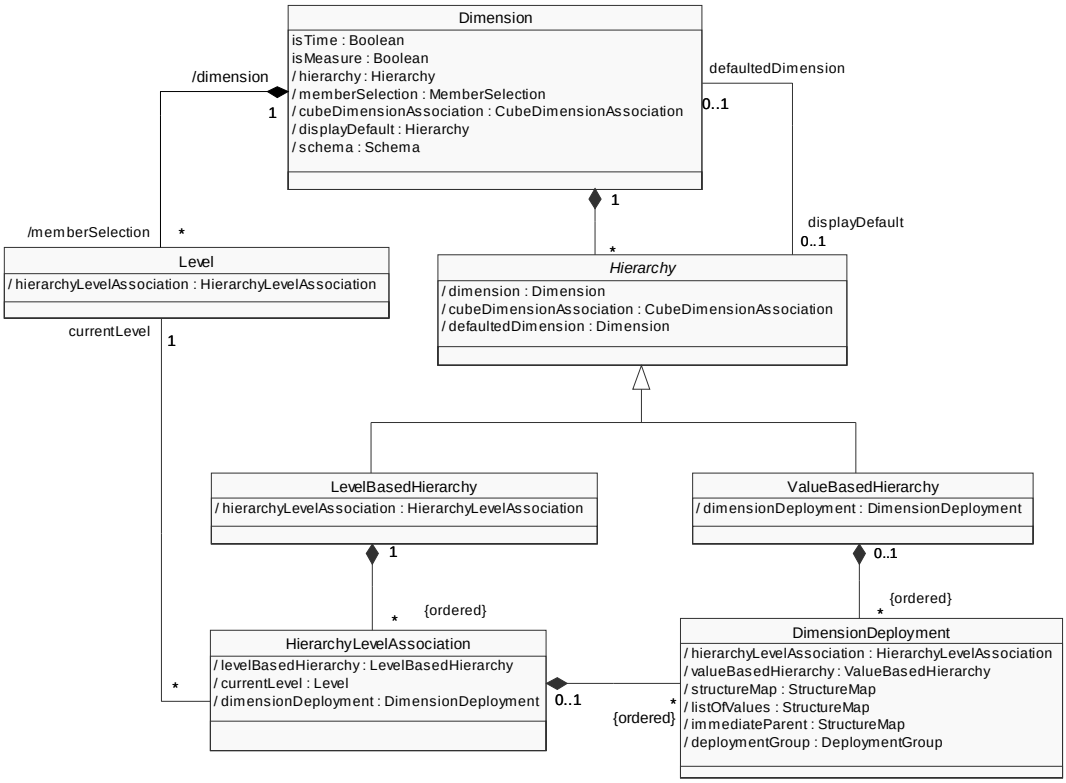


*Figure 14-2* OLAP Metamodel: Dimension and Hierarchy

base periodicity (e.g., days of the week). The implementation of a Time Dimension might provide support for advanced "time-intelligent" functionality, such as the ability to automatically convert between different periodicities and calendars.

The members of a Measure Dimension describe the meaning of the analytic values stored in each data cell of a multidimensional structure. For example, an OLAP application may define Sales, Quantity and Weight as its measures. In this case, each data cell within the Cube stores three values, with each value corresponding to one of the three measures. A measure may have an associated data type. For example, Sales might be of a monetary type, Quantity an integer, and Weight a real number.

## Hierarchy

The OLAP metamodel specifies two subclasses of Hierarchy: LevelBasedHierarchy and ValueBasedHierarchy.

### *LevelBasedHierarchy*

LevelBasedHierarchy describes hierarchical relationships between specific levels of a Dimension. LevelBasedHierarchy is used to model both "pure level" hierarchies (e.g., dimension-level tables) and "mixed" hierarchies (i.e., levels plus linked nodes). Dimensional levels are modeled by the Level class, a subclass of MemberSelection that partitions a Dimension's members into disjoint subsets, each representing a distinct level.

For example, the Geography Dimension cited earlier contains members representing cities, states, and regions, such as "Stamford", "Connecticut", and "NorthEast". It might also contain a single member called "USA" representing all regions of the United States. Therefore, the Geography Dimension could have four Levels named "City", "State", "Region", and "ALL", respectively. Each Level specifies the subset of members belonging to it: All cities belong to the "City" Level, all states belong to the "State" Level, all regions belong to the "Region" Level, and the single "USA" member belongs to the "ALL" Level.

When used in the definition of a consolidation path, the meaning of "level" is quite clear: Members occupying a given Level consolidate into the next higher Level (e.g., City rolls up into State, State into Region, and Region into ALL).

LevelBasedHierarchy contains an ordered collection of HierarchyLevelAssocations that defines the natural hierarchy of the Dimension. The ordering defines the hierarchical structure in top-down fashion (i.e., the "first" HierarchyLevelAssociation in the ordered collection represents the upper-most level of the dimensional hierarchy). A HierarchyLevelAssociation may own any number of DimensionDeployments. DimensionDeployment is a metaclass that represents an implementation strategy for hierarchical Dimensions. The ordering of the DimensionDeployment classes may optionally be given an implementation-specific meaning (e.g., desired order of selection of several possible deployment strategies, based on optimization considerations).

### *ValueBasedHierarchy*

A ValueBasedHierarchy defines a hierarchical ordering of members in which the concept of level has little or no significance. Instead, the topological structure of the hierarchy conveys meaning. ValueBasedHierarchies are often used to model situations where members are classified or ranked according to their distance from a common root member (e.g., an organizational chart of a corporation). In this case, each member of the hierarchy has some specific "metric" or "value" associated it with it.

ValueBasedHierarchy can be used to model pure "linked node" hierarchies (e.g., asymmetric hierarchical graphs or parent-child tables).

As with LevelBasedHierarchy, ValueBasedHierarchy also has an ordered collection of DimensionDeployments, where the ordering semantics are left to implementations to define.

### 14.3.4 Inheritance from the Object Model

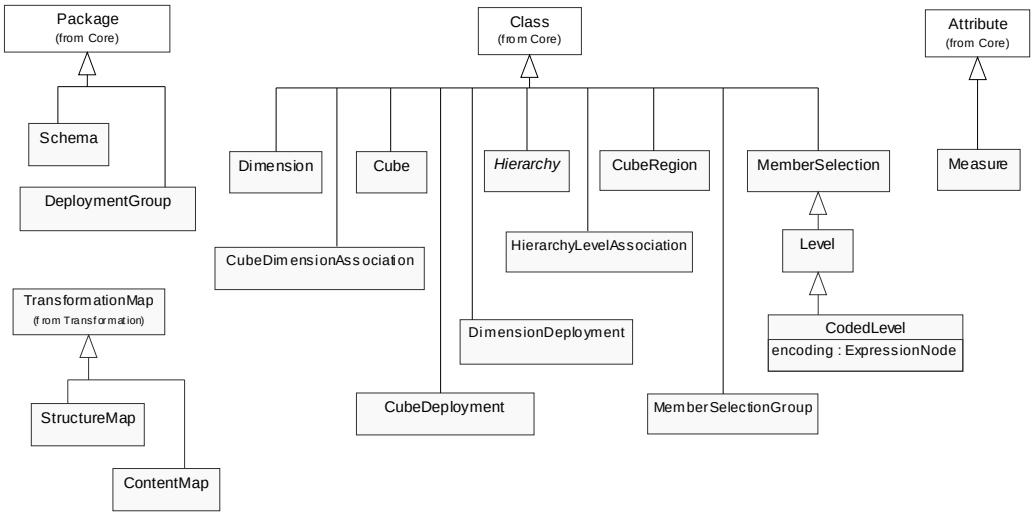


*Figure 14-3* OLAP Metamodel: Inheritance from Object Model

Figure 14-3 illustrates how classes of the OLAP metamodel inherit from the CWM Object Model. Two classes requiring further explanation are:

- *Measure*: A subclass of Attribute that describes the meaning of values stored in the data cells of a multidimensional structure. Different OLAP models often give different interpretations to the term "measure". In a relational Star Schema, individual measures might be represented by non-key columns of a Fact table (e.g., "Sales" and "Quantity" columns). In this case, measure may be an attribute of a Cube or CubeRegion that models the Fact table. On the other hand, measures can also be represented by members of a Measure Dimension. A Fact table supporting this representation has a single Measure column with column values consisting of the members "Sales" and "Quantity", and a single "value" column (i.e., an implicit data dimension) storing the corresponding measure values. A similar notion of Measure Dimension is used in modeling pure hypercube representations of multidimensional servers. Thus, the concept of measure can be represented either as a Dimenson or as an Attribute, depending on the type of OLAP system being modeled.
- *Coded Level*: A subclass of Level that assigns a unique encoding, or label, to each of its members. CodedLevel is not essential to the OLAP metamodel, but is provided as a helper class for certain applications that might benefit from the ability of OLAP systems to structure data hierarchically. For example, CodedLevel could be used to model systems of nomenclature or classification.

### 14.3.5 Deploying OLAP Models

The CWM OLAP metamodel describes logical models of OLAP systems, but does not directly specify how an OLAP system is physically deployed. Modeling the deployment of an OLAP system requires mapping instances of OLAP metaclasses to instances of other CWM metaclasses representing physical resources (e.g., mapping an OLAP Dimension to a Relational Table). This approach offers several advantages:

- The status of the OLAP metamodel as a conceptual model is preserved by this level of indirection. When using OLAP, a client may perceive to be working directly with OLAP objects, but the actual implementation of those objects is hidden from the client. For example, a client may view a member as a value of a Dimension, but whether that member value comes from a row of a relational table, or from a cell in a multidimensional database, is usually not obvious to the client. On the other hand, if a client needs to determine how a logical OLAP structure is physically realized, the metadata describing this mapping is fully available (assuming that the implementation allows the client to drill-down through the metadata).
- The possibility of defining mappings based on expressions means that the amount of metadata required to describe large models (e.g., Dimensions containing large collections of members) can be kept within reasonable bounds. It is generally more efficient to provide expressions that specify where large metadata sets reside, how to connect to them, and how to map their contents, rather than representing them directly as part of the metadata content.

All of the OLAP metaclasses are potential candidates for such deployment mappings. In addition, some OLAP models may also define mappings between several OLAP metaclass instances, forming a natural hierarchy of logical objects (e.g., Dimension Attributes are mapped to Level Attributes which, in turn, are mapped to Table Columns).

The CWM Transformation package is used as the primary means of describing these mappings. A modeler constructing an OLAP model based on CWM will generally define instances of the TransformationMap metaclass to link logical OLAP objects together, and to link those logical objects to other objects representing their physical data sources.

StructureMap is a subclass of TransformationMap that models *structure-oriented* transformation mappings (i.e., member identity and hierarchical structure). This type of transformation mapping needs to be connected to the OLAP metamodel in a very specific way (according to Level and Hierarchy), so the StructureMap subclass is defined to make these associations explicit. Two specific usages of StructureMap are defined: *ListOfValues*, which maps attributes identifying members residing at a specific Level, or at a specific Level within a particular Hierarchy, and *ImmediateParent*, which maps attributes identifying the hierarchical parent(s) of the members.

On the other hand, relatively simple TransformationMaps can be defined within any OLAP model to represent *attribute-oriented* transformations (e.g., mapping Dimension Attributes to Table Columns that store attribute values).

ContentMap is a subclass of TransformationMap that models *content-oriented* transformaton mappings (i.e., cell data or measure values). For example, an instance of ContentMap might be used to map each of a CubeRegion's Measures to Columns of an underlying Fact Table.

Note that, in either case (structural mapping or content mapping), the traversal patterns used by any CWM OLAP implementation are always the same, since both deployment mappings are based on the same usage of CWM TransformationMaps.

In addition to representing structural mappings, instances of TransformationMap and its subclasses are also capable of storing implementation-dependent functions or procedures that yield the instance values associated with mapped model elements. For example, a "list of values" StructureMap might store an SQL statement such as "select memberName from Product where productFamily = 'consumerElectronics' ", as the value of its formula attribute.

Figure 14-4 illustrates the CWM metaclasses and associations that describe deployment mappings between logical OLAP models and physical resource models. Note that it is possible to combine both Cube (content) and Dimension (structure) deployments together within the context of a single OLAP Schema (via the DeploymentGroup metaclass). Thus, an OLAP Schema can have several possible deployments that users may select based on implementation-specific considerations (e.g., physical optimizations).

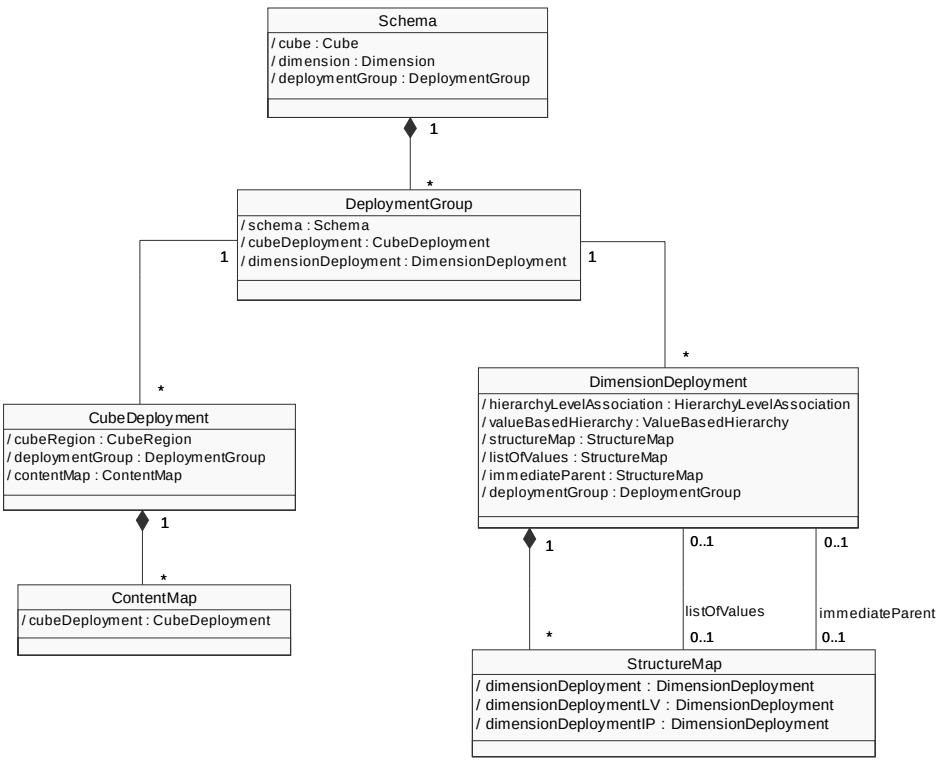


*Figure 14-4* OLAP Metamodel: Deployment Mapping Structures

## 14.4 OLAP Classes

### 14.4.1 CodedLevel

CodedLevel is a subclass of Level that assigns a unique encoding, or label, to each of its Dimension members.

#### Superclasses

Level

#### Attributes

***encoding***

Encoding is an expression that generates a unique encoding, or label, for each member of a CodedLevel.

| | |
|---|---|
| *type:* | ExpressionNode |
| *multiplicity:* | exactly one |

### 14.4.2 ContentMap

ContentMap is a subclass of TransformationMap that maps CubeRegion attributes to their physical data sources.

#### Superclasses

TransformationMap

### References

***cubeDeployment***

References the CubeDeployment owning a ContentMap.

| | |
|---|---|
| *class:* | CubeDeployment |
| *defined by:* | CubeDeploymentOwnsContentMaps::<br>cubeDeployment |
| *multiplicity:* | exactly one |
| *inverse:* | CubeDeployment::contentMap |

## 14.4.3 Cube

A Cube is a collection of analytic values (i.e., measures) that share the same dimensionality. This dimensionality is specified by a set of unique Dimensions from the Schema. Each unique combination of members in the Cartesian product of the Cube's Dimensions identifies precisely one data cell within a multidimensional structure.

Synonyms: Multidimensional Array, Hypercube, Hypervolume.

### Superclasses

Class

### Contained Elements

- CubeDimensionAssociation
- CubeRegion

## Attributes

### *isVirtual*

If true, then this Cube is a Virtual Cube (i.e., it has no physical realization).

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

## References

### *cubeDimensionAssociation*

References the collection of CubeDimensionAssociations owned by a Cube.

| | |
|---|---|
| *class:* | CubeDimensionAssociation |
| *defined by:* | CubeOwnsCubeDimensionAssociations:: cubeDimensionAssociation |
| *multiplicity:* | zero or more |
| *inverse:* | CubeDimensionAssociation::cube |

### *cubeRegion*

References the collection of CubeRegions owned by a Cube.

| | |
|---|---|
| *class:* | CubeRegion |
| *defined by:* | CubeOwnsCubeRegions::cubeRegion |
| *multiplicity:* | zero or more |
| *inverse:* | CubeRegion::cube |

### *schema*

References the Schema owning a Cube.

| | |
|---|---|
| *class:* | Schema |
| *defined by:* | SchemaOwnsCubes::schema |
| *multiplicity:* | exactly one |
| *inverse:* | Schema::cube |

### *Constraints*

Ensure that the Dimensions defining a Cube are unique. [C-1]

A Cube without CubeRegions cannot be mapped to a deployment structure (i.e., physical source of data). [C-2]

## *14.4.4 CubeDeployment*

CubeDeployment represents a particular implementation strategy for the data portions of an OLAP model. It does so by organizing a collection of ContentMaps, which in turn define a mapping to an implementation model.

### *Superclasses*

Class

### *Contained Elements*

- ContentMap

## References

### *cubeRegion*

References the CubeRegion owning a CubeDeployment.

| | |
|---|---|
| *class:* | CubeRegion |
| *defined by:* | CubeRegionOwnsCubeDeployments::cubeRegion |
| *multiplicity:* | exactly one |
| *inverse:* | CubeRegion::cubeDeployment |

### *deploymentGroup*

References the DeploymentGroup associated with this CubeDeployment.

| | |
|---|---|
| *class:* | DeploymentGroup |
| *defined by:* | DeploymentGroupReferencesCubeDeployments::cubeDeployment |
| *multiplicity:* | exactly one |
| *inverse:* | DeploymentGroup::cubeDeployment |

### *contentMap*

References the ContentMaps owned by a CubeDeployment.

| | |
|---|---|
| *class:* | ContentMap |
| *defined by:* | CubeDeploymentOwnsContentMaps::contentMap |
| *multiplicity:* | zero or more |
| *inverse:* | ContentMap::cubeDeployment |

### 14.4.5 CubeDimensionAssociation

CubeDimensionAssociation relates a Cube to the Dimensions that define it. Features relevant to Cube-Dimension relationships (e.g., calcHierarchy) are exposed by this class.

#### Superclasses

Class

### References

***dimension***

References the Dimension associated with a CubeDimensionAssociation.

| | |
|---|---|
| *class:* | Dimension |
| *defined by:* | CubeDimensionAssociationsReferenceDimension::dimension |
| *multiplicity:* | exactly one |
| *inverse:* | Dimension::cubeDimensionAssociation |

***cube***

References the Cube owning a CubeDimensionAssociation.

| | |
|---|---|
| *class:* | Cube |
| *defined by:* | CubeOwnsCubeDimensionAssociations::cube |
| *multiplicity:* | exactly one |
| *inverse:* | Cube::cubeDimensionAssociation |

***calcHierarchy***

References the default calculation Hierarchy of the Dimension associated with a CubeDimensionAssociation.

| | |
|---|---|
| *class:* | Hierarchy |
| *defined by:* | CubeDimensionAssociationsReferenceCalcHierarchy::calcHierarchy |
| *multiplicity:* | zero or one |

### Constraints

If a calcHierarchy is defined, it must be a Hierarchy owned by the Dimension referenced by the CubeDimensionAssociation. [C-3]

## 14.4.6 CubeRegion

CubeRegion models a sub-unit of a Cube that is of the same dimensionality as the Cube itself. Each "dimension" of a CubeRegion is represented by a MemberSelection of the corresponding Dimension of the Cube. Furthermore, these MemberSelections may define subsets of their Dimension members.

Synonyms: Sub-Cube, Partition, Slice, Region, Area.

## Superclasses

Class

## Contained Elements

- CubeDeployment
- MemberSelectionGroup

## Attributes

### *isReadOnly*

If true, then the CubeRegion content is read-only (i.e., may not be written or updated through the CubeRegion -- e.g., a CubeRegion implemented via an SQL view may not permit updates to the underlying base table).

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

### *isFullyRealized*

If true, then this CubeRegion has a direct physical realization and is not bound by any MemberSelections.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

## References

### *memberSelectionGroup*

References the collection of MemberSelectionGroups owned by a Cube.

| | |
|---|---|
| *class:* | MemberSelectionGroup |
| *defined by:* | CubeRegionOwnsMemberSelectionGroups:: memberSelectionGroup |
| *multiplicity:* | zero or more |
| *inverse:* | MemberSelectionGroup::cubeRegion |

### *cube*

References the Cube owning a CubeRegion.

| | |
|---|---|
| *class:* | Cube |
| *derived from:* | CubeOwnsCubeRegions::cube |
| *multiplicity:* | exactly one |
| *inverse:* | Cube::cubeRegion |

### *cubeDeployment*

References the CubeDeployments owned by a CubeRegion.

| | |
|---|---|
| *class:* | CubeDeployment |
| *derived from:* | CubeRegionOwnsCubeDeployments:: cubeDeployment |
| *multiplicity:* | zero or more; ordered |
| *inverse:* | CubeDeployment::CubeRegion |

## Constraints

A "fully realized" CubeRegion has no MemberSelectionGroups (and hence, no MemberSelections). [C-4]

A CubeRegion defined by MemberSelections must have, for each Dimension of its owning Cube, a corresponding MemberSelection within each of its MemberSelectionGroups. [C-5]

A CubeRegion defined by MemberSelections must have, within each MemberSelectionGroup, a MemberSelection corresponding to each Dimension of its owning Cube. [C-6]

### 14.4.7 DeploymentGroup

DeploymentGroup represents a logical grouping of model elements defining a single, complete deployment of an instance of Olap Schema (i.e., CubeDeployments and DimensionDeployments).

The usage of DeploymentGroup is as follows: A user may specify a particular DeploymentGroup as the session-wide, default deployment for all metadata queries performed throughout the session. Alternatively, for queries involving some particular deployed object (e.g., a Cube or a Dimension), the user may be asked to choose from a list of deployments. This either becomes the default deployment for the remainder of the session, or the user may continue to be asked to specify a deployment for each subsequent query against deployed objects.

#### *Superclasses*

Package

## References

### *schema*

References the Schema owning a DeploymentGroup.

| | |
|---|---|
| *class:* | Schema |
| *defined by:* | SchemaOwnsDeploymentGroups::schema |
| *multiplicity:* | exactly one |
| *inverse:* | Schema::deploymentGroup |

### *cubeDeployment*

References the collection of CubeDeployments associated with a DeploymentGroup.

| | |
|---|---|
| *class:* | CubeDeployment |
| *defined by:* | DeploymentGroupReferencesCubeDeployments ::cubeDeployment |
| *multiplicity:* | zero or more |
| *inverse:* | CubeDeployment::deploymentGroup |

### *dimensionDeployment*

References the collection of DimensionDeployments associated with a DeploymentGroup.

| | |
|---|---|
| *class:* | DimensionDeployment |
| *defined by:* | DeploymentGroupReferencesDimensionDeployments ::dimensionDeployment |
| *multiplicity:* | zero or more |
| *inverse:* | DimensionDeployment::deploymentGroup |

### 14.4.8 Dimension

A Dimension is an ordinate within a multidimensional structure, and consists of a unique list of values (i.e., members) that share a common semantic meaning within the domain being modeled. Each member designates a unique position along its ordinate.

Typical Dimensions are: Time, Product, Geography, Scenario (e.g., actual, budget, forecast), Measure (e.g., sales, quantity).

#### *Superclasses*

Class

#### *Contained Elements*

- Hierarchy
- MemberSelection

## Attributes

### *isTime*

If true, then this Dimension is a Time Dimension (i.e., its members collectively represent a time series).

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

### *isMeasure*

If true, then this Dimension is a Measure Dimension (i.e., its members represent Measures).

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

## References

### *hierarchy*

References the collection of Hierarchies owned by a Dimension.

| | |
|---|---|
| *class:* | Hierarchy |
| *defined by:* | DimensionOwnsHierarchies::hierarchy |
| *multiplicity:* | zero or more |
| *inverse:* | Hierarchy::dimension |

### *memberSelection*

References the collection of MemberSelections owned by a Dimension.

| | |
|---|---|
| *class:* | MemberSelection |
| *defined by:* | DimensionOwnsMemberSelections::memberSelection |
| *multiplicity:* | zero or more |
| *inverse:* | MemberSelection::dimension |

***cubeDimensionAssociation***

References the collection of CubeDimensionAssociations referencing this Dimension.

| | |
|---|---|
| *class:* | CubeDimensionAssociation |
| *defined by:* | CubeDimensionAssociationsReferenceDimension:: cubeDimensionAssociation |
| *multiplicity:* | zero or more |
| *inverse:* | CubeDimensionAssociation::dimension |

***displayDefault***

References the default display Hierarchy of a Dimension.

| | |
|---|---|
| *class:* | Hierarchy |
| *defined by:* | DimensionHasDefaultHierarchy::displayDefault |
| *multiplicity:* | zero or one |

***schema***

References the Schema owning a Dimension.

| | |
|---|---|
| *class:* | Schema |
| *defined by:* | SchemaOwnsDimensions::schema |
| *multiplicity:* | exactly one |
| *inverse:* | Schema::dimension |

## *Constraints*

A Dimension may be a Time Dimension, a Measure Dimension, or neither, but never both types at the same time. [C-7]

The default display Hierarchy (if defined) must be one of the Hierarchies owned by the Dimension. [C-8]

### 14.4.9 DimensionDeployment

A DimensionDeployment represents a particular implementation strategy for the dimensional/hierarchical portions of an OLAP model. It does so by organizing a collection of StructureMaps, which in turn define a mapping to an implementation model.

#### Superclasses

Class

#### Contained Elements

StructureMap

## References

### *hierarchyLevelAssociation*

References the HierarchyLevelAssociation owning a DimensionDeployment.

| | |
|---|---|
| *class:* | HierarchyLevelAssociation |
| *defined by:* | HierarchyLevelAssociationOwnsDimension Deployments::hierarchyLevelAssociation |
| *multiplicity:* | zero or one |
| *inverse:* | HierarchyLevelAssociation::dimensionDeployment |

### *valueBasedHierarchy*

References the ValueBasedHierarchy owning a DimensionDeployment.

| | |
|---|---|
| *class:* | ValueBasedHierarchy |
| *defined by:* | ValueBasedHierarchyOwnsDimensionDeployments:: valueBasedHierarchy |
| *multiplicity:* | zero or one |
| *inverse:* | ValueBasedHierarchy::dimensionDeployment |

### *structureMap*

References the collection of StructureMaps owned by a DimensionDeployment.

| | |
|---|---|
| *class:* | StructureMap |
| *defined by:* | DimensionDeploymentOwnsStructureMaps:: structureMap |
| *multiplicity:* | zero or more |
| *inverse:* | StructureMap::dimensionDeployment |

### *listOfValues*

References the "list of values" StructureMap owned by a DimensionDeployment.

| | |
|---|---|
| *class:* | StructureMap |
| *defined by:* | DimensionDeploymentHasListOfValues::listOfValues |
| *multiplicity:* | zero or one |
| *inverse:* | StructureMap::dimensionDeploymentLV |

***immediateParent***

References the "immediate parent" StructureMap owned by a DimensionDeployment.

| | |
|---|---|
| *class:* | StructureMap |
| *defined by:* | DimensionDeploymentHasImmediateParent:: immediateParent |
| *multiplicity:* | zero or one |
| *inverse:* | StructureMap::DimensionDeploymentIP |

***deploymentGroup***

References the DeploymentGroup associated with this DimensionDeployment.

| | |
|---|---|
| *class:* | DeploymentGroup |
| *defined by:* | DeploymentGroupReferencesDimensionDeployments ::deploymentGroup |
| *multiplicity:* | exactly one |
| *inverse:* | DeploymentGroup::dimensionDeployment |

## *Constraints*

An instance of DimensionDeployment must be referenced exclusively by either a HierarchyLevelAssociation or a ValueBasedHierarchy. [C-9]

Within a DimensionDeployment, an "immediate parent" StructureMap must always have an associated and distinct "list of values" StructureMap. [C-10]

A StructureMap referenced as a "list of values" StructureMap must not reside outside of the DimensionDeployment's collection of StructureMaps. [C-11]

A StructureMap referenced as an "immediate parent" StructureMap must not reside outside of the DimensionDeployment's collection of StructureMaps. [C-12]

### 14.4.10 Hierarchy *abstract*

A Hierarchy is an organizational structure that describes a traversal pattern through a Dimension, based on parent/child relationships between members of the Dimension. Hierarchies are used to define both navigational and consolidation/computational paths through the Dimension (i.e., a value associated with a child member is aggregated by one or more parents).

#### *Superclasses*

Class

## References

### *dimension*

References the Dimension owning a Hierarchy.

| | |
|---|---|
| *class:* | Dimension |
| *defined by:* | DimensionOwnsHierarchies::dimension |
| *multiplicity:* | exactly one |
| *inverse:* | Dimension::hierarchy |

### *cubeDimensionAssociation*

References the collection of CubeDimensionAssociations designating this Hierarchy as their default calculation Hierarchy.

| | |
|---|---|
| *class:* | cubeDimensionAssociation |
| *defined by:* | CubeDimensionAssociationsReferenceCalcHierarchy ::cubeDimensionAssociation |
| *multiplicity:* | zero or more |
| *inverse:* | CubeDimensionAssociation::calcHierarchy |

### *defaultedDimension*

References the Dimension for which this Hierarchy is the "display default" Herarchy.

| | |
|---|---|
| *class:* | Dimension |
| *defined by:* | DimensionHasDisplayDefault::defaultedDimension |
| *multiplicity:* | zero or one |
| *inverse:* | Dimension::displayDefault |

### 14.4.11 HierarchyLevelAssociation

HierarchyLevelAssociation is a class that orders Levels within a LevelBasedHierarchy, and provides a means of mapping Level and/or Hierarchy -oriented Dimension attributes to deployment structures (i.e., physical data sources).

#### Superclasses

Class

#### Contained Elements

DimensionDeployment

## References

### *levelBasedHierarchy*

References the LevelBasedHierarchy owning this HierarchyLevelAssociation.

| | |
|---|---|
| *class:* | LevelBasedHierarchy |
| *defined by:* | LevelBasedHierarchyOwnsHierarchyLevel Associations::levelBasedHierarchy |
| *multiplicity:* | exactly one |
| *inverse:* | LevelBasedHierarchy::hierarchyLevelAssociation |

### *currentLevel*

References the "current" Level associated with this HierarchyLevelAssociation.

| | |
|---|---|
| *class:* | Level |
| *defined by:* | HierarchyLevelAssociationsReferenceLevel:: currentLevel |
| *multiplicity:* | exactly one |
| *inverse:* | Level::hierarchyLevelAssociation |

### *dimensionDeployment*

References the collection of DimensionDeployments owned by a HierarchyLevelAssociation.

| | |
|---|---|
| *class:* | DimensionDeployment |
| *defined by:* | HierarchyLevelAssociationOwnsDimension Deployments::dimensionDeployment |
| *multiplicity:* | zero or more; ordered |
| *inverse:* | DimensionDeployment::hierarchyLevelAssociation |

### 14.4.12 Level

Level is a subclass of MemberSelection that assigns each member of a Dimension to a specific level within the Dimension.

#### Superclasses

MemberSelection

#### References

***hierarchyLevelAssociation***

References the HierarchyLevelAssociations denoting this Level as "current level".

| | |
|---|---|
| *class:* | HierarchyLevelAssociation |
| *defined by:* | HierarchyLevelAssociationsReferenceLevel:: hierarchyLevelAssociation |
| *multiplicity:* | zero or more |
| *inverse:* | HierarchyLevelAssociation::currentLevel |

### 14.4.13 LevelBasedHierarchy

A LevelBasedHierarchy is a Hierarchy that describes relationships between specific levels of a Dimension. LevelBasedHierarchy is used to model both "pure level" hierarchies (e.g., dimension-level tables) and "mixed" hierarchies (i.e., levels plus linked nodes).

#### Supertypes

Hierarchy

#### Contained Elements

HierarchyLevelAssociation

### References

#### *hierarchyLevelAssociation*

References the collection of HierarchyLevelAssociations owned by a LevelBasedHierarchy.

| | |
|---|---|
| *class:* | HierarchyLevelAssociation |
| *defined by:* | LevelBasedHierarchyOwnsHierarchyLevel Associations::hierarchyLevelAssociation |
| *multiplicity:* | zero or more; ordered |
| *inverse:* | HierarchyLevelAssociation::levelBasedHierarchy |

### Constraints

The currentLevel of each HierarchyLevelAssociation must refer to a Level owned by the Dimension of the LevelBasedHierarchy containing the HierarchyLevelAssociation. [C-13]

No two HierarchyLevelAssociations may designate the same Level instance as their "current level". [C-14]

## 14.4.14 Measure

Measure is a subclass of Attribute representing Dimension Measures (e.g., Sales, Quantity, Weight). Synonym: Variable.

### Supertypes

Attribute

## 14.4.15 MemberSelection

MemberSelection represents an arbitrary subset of the members of a Dimension.

### Superclasses

Class

### References

#### *dimension*

References the Dimension owning a MemberSelection.

| | |
|---|---|
| *class:* | Dimension |
| *defined by:* | DimensionOwnsMemberSelections::dimension |
| *multiplicity:* | exactly one |
| *inverse:* | Dimension::memberSelection |

#### *memberSelectionGroup*

References the collection of MemberSelectGroups associated with a MemberSelection.

| | |
|---|---|
| *class:* | MemberSelectionGroup |
| *defined by:* | MemberSelectionGroupReferencesMemberSelection::memberSelectionGroup |
| *multiplicity:* | zero or more |
| *inverse:* | MemberSelectionGroup::memberSelection |

## 14.4.16 MemberSelectionGroup

MemberSelectionGroup enables the grouping together of semantically-related MemberSelections.

### Superclasses

Class

### References

***memberSelection***

References the collection of MemberSelections associated with a MemberSelectionGroup.

| | |
|---|---|
| *class:* | MemberSelection |
| *defined by:* | MemberSelectionGroupReferencesMemberSelections ::memberSelection |
| *multiplicity:* | one or more |
| *inverse:* | MemberSelection::memberSelectionGroup |

***cubeRegion***

References the CubeRegion owning a MemberSelectionGroup.

| | |
|---|---|
| *class:* | CubeRegion |
| *defined by:* | CubeRegionOwnsMemberSelectionGroups:: cubeRegion |
| *multiplicity:* | exactly one |
| *inverse:* | CubeRegion::memberSelectionGroup |

## 14.4.17 Schema

Schema contains all elements comprising an OLAP model. A Schema may also contain any number of DeploymentGroups, representing the various physical deployments of the logical Schema.

### Superclasses

Package

### Contained Elements

- Cube
- DeploymentGroup
- Dimension

### References

***cube***

References the collection of Cubes owned by a Schema.

| | |
|---|---|
| *class:* | Cube |
| *defined by:* | SchemaOwnsCubes:cube |
| *multiplicity:* | zero or more |
| *inverse:* | Cube::schema |

***deploymentGroup***

References the collection of DeploymentGroups owned by a Schema.

| | |
|---|---|
| *class:* | DeploymentGroup |
| *defined by:* | SchemaOwnsDeploymentGroups::deploymentGroup |
| *multiplicity:* | zero of more |
| *inverse:* | DeploymentGroup::schema |

***dimension***

References the collection of Dimensions owned by a Schema.

| | |
|---|---|
| *class:* | Dimension |
| *defined by:* | SchemaOwnsDimensions::dimension |
| *multiplicity:* | zero or more |
| *inverse:* | Dimension::schema |

## 14.4.18 StructureMap

StructureMap is a subclass of TransformationMap that maps Dimension attributes to their physical data sources.

### Superclasses

TransformationMap

### References

***dimensionDeployment***

References the DimensionDeployment owning this StructureMap.

| | |
|---|---|
| *class:* | DimensionDeployment |
| *defined by:* | DimensionDeploymentOwnsStructureMaps:: dimensionDeployment |
| *multiplicity:* | exactly one |
| *inverse:* | DimensionDeployment::structureMap |

***dimensionDeploymentLV***

References the DimensionDeployment designating this StructureMap as a "list of values" StructureMap.

| | |
|---|---|
| *class:* | DimensionDeployment |
| *defined by:* | DimensionDeploymentHasListOfValues:: dimensionDeployment |
| *multiplicity:* | zero or one |
| *inverse:* | DimensionDeployment::listOfValues |

***dimensionDeploymentIP***

References the DimensionDeployment designating this StructureMap as an "immediate parent" StructureMap.

| | |
|---|---|
| *class:* | DimensionDeployment |
| *defined by:* | DimensionDeploymentHasImmediateParent:: dimensionDeployment |
| *multiplicity:* | zero or one |
| *inverse:* | DimensionDeployment::immediateParent |

## 14.4.19 ValueBasedHierarchy

ValueBasedHierarchy is a subclass of Hierarchy that ranks Dimension members according to their relative distance from the root. Each member of a ValueBasedHierarchy has a specific "metric" or "value" associated with it.

ValueBasedHierarchy is used to model pure "linked node" hierarchies (e.g., parent-child tables).is a subclass of Hierarchy that ranks Dimension members according to their relative distance from a common root member.

### *Superclasses*

Hierarchy

### *Contained Elements*

DimensionDeployment

### *References*

***dimensionDeployment***

References the collection of DimensionDeployments owned by a ValueBasedHierarchy.

| | |
|---|---|
| *class:* | DimensionDeployment |
| *defined by:* | ValueBasedHierarchyOwnsDimensionDeployments::dimensionDeployment |
| *multiplicity:* | zero or more; ordered |
| *inverse:* | DimensionDeployment::valueBasedHierarchy |

## *14.5 OLAP Associations*

### *14.5.1 CubeDeploymentOwnsContentMaps*

A CubeDeployment owns any number of ContentMaps.

### *Ends*

***cubeDeployment***

The CubeDeployment owning a ContentMap.

| | |
|---|---|
| *class:* | CubeDeployment |
| *multiplicity:* | exactly one |
| *aggregation:* | composite |

***contentMap***

The collection of ContentMaps owned by a CubeDeployment.

*class:* ContentMap

*multiplicity:* zero or more

## 14.5.2 CubeDimensionAssociationsReferenceCalcHierarchy

A CubeDimAssociation may designate a default Hierarchy for calculation purposes.

*Ends*

***calcHierarchy***

The Hierarchy designated by a CubeDimensionAssociation as the default Hierarchy to be used in consolidation calculations performed on the Cube.

*class:* Hierarchy

*multiplicity:* zero or one

***cubeDimensionAssociation***

CubeDimensionAssociations designating the Hierarchy to be used in consolidation calculations performed on the Cube.

*class:* CubeDimensionAssociation

*multiplicity:* zero or more

## 14.5.3 CubeDimensionAssociationsReferenceDimension

Each CubeDimensionAssociation references a single Dimension.

*Ends*

***cubeDimensionAssociation***

CubeDimensionAssociations referencing the Dimension.

*type:* CubeDimensionAssociation

*multiplicity:* zero or more

***dimension***

The Dimension referenced by CubeDimensionAssociations.

| | |
|---|---|
| *type:* | Dimension |
| *multiplicity:* | exactly one |

### 14.5.4 CubeOwnsCubeDimensionAssociations

The dimensionality of a Cube is defined by a collection of unique Dimensions. Each Dimension is represented by an instance of CubeDimensionAssociation.

*Ends*

***cube***

The Cube owning CubeDimensionAssociations.

| | |
|---|---|
| *class:* | Cube |
| *multiplicity:* | exactly one |
| *aggregation:* | composite |

***cubeDimensionAssociation***

CubeDimensionAssociations owned by the Cube.

| | |
|---|---|
| *class:* | CubeDimensionAssociation |
| *multiplicity:* | zero or more |

### 14.5.5 CubeOwnsCubeRegions

A Cube may own any number of CubeRegions.

*Ends*

***cube***

The Cube owning CubeRegions.

*class:* Cube

*multiplicity:* exactly one

*aggregation:* composite

***cubeRegion***

CubeRegions owned by the Cube.

*class:* CubeRegion

*multiplicity:* zero or more

### 14.5.6 CubeRegionOwnsCubeDeployments

A CubeRegion may own any number of CubeDeployments.

*Ends*

***cubeRegion***

The CubeRegion owning a CubeDeployment.

*class:* CubeRegion

*multiplicity:* exactly one

*aggregation:* composite

***cubeDeployment***

The CubeDeployments owned by a CubeRegion.

*class:* CubeDeployment

*multiplicity:* zero or more; ordered

### 14.5.7 CubeRegionOwnsMemberSelectionGroups

A CubeRegion may own any number of MemberSelectionGroups.

*Ends*

***cubeRegion***

The CubeRegion owning MemberSelectionGroups.

| | |
|---|---|
| *class:* | CubeRegion |
| *multiplicity:* | exactly one |
| *aggregation:* | composite |

***memberSelectionGroup***

MemberSelectionGroups owned by the CubeRegion.

| | |
|---|---|
| *class:* | MemberSelectionGroup |
| *multiplicity:* | zero or more |

## *14.5.8 DeploymentGroupReferencesCubeDeployments*

A DeploymentGroup may reference any number of CubeDeployments.

*Ends*

***deploymentGroup***

The DeploymentGroups referencing a CubeDeployment.

| | |
|---|---|
| *class:* | DeploymentGroup |
| *multiplicity:* | exactly one |

***cubeDeployment***

The CubeDeployments referenced by a DeploymentGroup.

| | |
|---|---|
| *class:* | CubeDeployment |
| *multiplicity:* | zero or more |

## *14.5.9 DeploymentGroupReferencesDimensionDeployments*

A DeploymentGroup may reference any number of DimensionDeployments.

*Ends*

***deploymentGroup***

The DeploymentGroups referencing a DimensionDeployment.

| | |
|---|---|
| *class:* | DeploymentGroup |
| *multiplicity:* | exactly one |

***dimensionDeployment***

The DimensionDeployments referenced by a DeploymentGroup.

| | |
|---|---|
| *class:* | DimensionDeployment |
| *multiplicity:* | zero or more |

## *14.5.10 DimensionDeploymentHasImmediateParent*

An instance of DimensionDeployment may reference zero or one StructureMaps as its "immediate parent" StructureMap.

*Ends*

***immediateParent***

The StructureMap referenced by a DimensionDeployment as its "immediate parent".

| | |
|---|---|
| *class:* | StructureMap |
| *multiplicity:* | zero or one |

***dimensionDeploymentIP***

The DimensionDeployment referencing an "immedate parent" StructureMap.

| | |
|---|---|
| *class:* | DimensionDeployment |
| *multiplicity:* | zero or one |

## *14.5.11 DimensionDeploymentHasListOfValues*

An instance of DimensionDeployment may reference zero or one StructureMaps as its "list of values" StructureMap.

*Ends*

***structureMap***

The StructureMap referenced by a DimensionDeployment as its "list of values" StructureMap.

| | |
|---|---|
| *class:* | StructureMap |
| *multiplicity:* | zero or one |

***dimensionDeploymentLV***

The DimensionDeployment referencing a "list of values" StructureMap.

| | |
|---|---|
| *class:* | DimensionDeployment |
| *multiplicity:* | zero or one |

## *14.5.12 DimensionDeploymentOwnsStructureMaps*

A DimensionDeployment may own any number of StructureMaps.

*Ends*

***structureMap***

The StructureMaps owned by a DimensionDeployment.

| | |
|---|---|
| *class:* | StructureMap |
| *multiplicity:* | zero or more |

***dimensionDeployment***

The DimensionDeployment owning a StructureMap.

| | |
|---|---|
| *class:* | DimensionDeployment |
| *multiplicity:* | exactly one |
| *aggregation* | composite |

### 14.5.13 DimensionHasDefaultHierarchy

A Dimension may designate a default Hierarchy for display purposes.

*Ends*

***displayDefault***

The Hierarchy designated by the Dimension as its default Hierarchy for display purposes.

| | |
|---|---|
| *class:* | Hierarchy |
| *multiplicity:* | zero or one |

***defaultedDimension***

The Dimension designating the Hierarchy as its default Hierarchy for display purposes.

| | |
|---|---|
| *class:* | Dimension |
| *multiplicity:* | zero or one |

### 14.5.14 DimensionOwnsHierarchies

A Dimension may own several Hierarchies.

*Ends*

***dimension***

The Dimension owning Hierarchies.

| | |
|---|---|
| *class:* | Dimension |
| *multiplicity:* | exactly one |
| *aggregation:* | composite |

***hierarchy***

Hierarchies owned by the Dimension.

| | |
|---|---|
| *class:* | Hierarchy |
| *multiplicity:* | zero or more |

### 14.5.15 DimensionOwnsMemberSelections

A Dimension may own several MemberSelections.

*Ends*

***dimension***

The Dimension owning MemberSelections.

| | |
|---|---|
| *class:* | Dimension |
| *multiplicity:* | exactly one |
| *aggregation:* | composite |

***memberSelection***

MemberSelections owned by the Dimension.

| | |
|---|---|
| *class:* | MemberSelection |
| *multiplicity:* | zero or more |

### 14.5.16 HierarchyLevelAssociationOwnsDimensionDeployments

A HierarchyLevelAssociation may own any number of DimensionDeployments.

*Ends*

***hierarchyLevelAssociation***

The HierarchyLevelAssociation owning DimensionDeployments.

| | |
|---|---|
| *class:* | HierarchyLevelAssociation |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

***dimensionDeployment***

The DimensionDepolyments owned by a HierarchyLevelAssociation.

| | |
|---|---|
| *class:* | DimensionDeployment |
| *multiplicity:* | zero or more; ordered |

### 14.5.17 HierarchyLevelAssociationsReferenceLevel

Each HierarchyLevelAssociation references precisely one Level as its current level.

*Ends*

***currentLevel***

The Level designated by a HierarchyLevelAssociation as its current level.

| | |
|---|---|
| *class:* | Level |
| *multiplicity:* | exactly one |

***hierarchyLevelAssociation***

The HierarchyLevelAssociations designating this Level as their current level.

| | |
|---|---|
| *class:* | HierarchyLevelAssociation |
| *multiplicity:* | zero or more |

### 14.5.18 LevelBasedHierarchyOwnsHierarchyLevelAssociations

A LevelBasedHierarchy may own any number of HierarchyLevelAssociations.

*Ends*

***levelBasedHierarchy***

The LevelBasedHierarchy owning HierarchyLevelAssociations.

| | |
|---|---|
| *class:* | LevelBasedHierarchy |
| *multiplicity:* | exactly one |
| *aggregation:* | composite |

***hierarchyLevelAssociation***

HierarchyLevelAssociations owned by the LevelBasedHierarchy.

| | |
|---|---|
| *class:* | HierarchyLevelAssociation |
| *multiplicity:* | zero or more; ordered |

### 14.5.19 MemberSelectionGroupReferencesMemberSelections

A MemberSelectionGroup references at least one unique MemberSelection.

*Ends*

***memberSelection***

MemberSelections referenced by MemberSelectionGroups.

| | |
|---|---|
| *class:* | memberSelection |
| *multiplicity:* | one or more |

***memberSelectionGroup***

MemberSelectionGroups referencing MemberSelections.

| | |
|---|---|
| *class:* | memberSelectionGroup |
| *multiplicity:* | zero or more |

### 14.5.20 SchemaOwnsCubes

A Schema may own any number of Cubes.

*Ends*

***cube***

The Cubes owned by a Schema.

| | |
|---|---|
| *class:* | Cube |
| *multiplicity:* | zero or more |

***schema***

The Schema owning a Cube.

| | |
|---|---|
| *class:* | Schema |
| *multiplicity:* | exactly one |
| *aggregation* | composite |

### 14.5.21 SchemaOwnsDeploymentGroups

A Schema may own any number of DeploymentGroups.

*Ends*

***deploymentGroup***

The DeploymentGroups owned by a Schema.

| | |
|---|---|
| *class:* | DeploymentGroup |
| *multiplicity:* | zero or more |

***schema***

The Schema owning a DeploymentGroup.

| | |
|---|---|
| *class:* | Schema |
| *multiplicity:* | exactly one |
| *aggregation* | composite |

### 14.5.22 SchemaOwnsDimensions

A Schema may own any number of Dimensions.

*Ends*

***dimension***

The Dimension owned by a Schema.

| | |
|---|---|
| *class:* | Dimension |
| *multiplicity:* | zero or more |

***schema***

The Schema owning a Dimension

| | |
|---|---|
| *class:* | Schema |
| *multiplicity:* | exactly one |
| *aggregation* | composite |

### 14.5.23 ValueBasedHierarchyOwnsDimensionDeployments

A ValueBasedHierarchy may own any number of DimensionDeployments.

*Ends*

***valueBasedHierarchy***

The ValueBasedHierarchy owning a DimensionDeployment.

| | |
|---|---|
| *class:* | ValueBasedHierarchy |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

***dimensionDeployment***

The DimensionDepolyments owned by a ValueBasedHierarchy.

| | |
|---|---|
| *class:* | DimensionDeployment |
| *multiplicity:* | zero or more; ordered |

## 14.6 OCL Representation of OLAP Constraints

[C-1] Ensure that the Dimensions defining a Cube are unique.

**context** Cube **inv:**
self.cubeDimensionAssociation->forAll( c1, c2 | c1 <> c2 implies
c1.dimension <> c2.dimension )

[C-2] A Cube without CubeRegions cannot be mapped to a deployment structure (i.e., physical source of data).

**context** Cube **inv:**
self.cubeRegion->isEmpty implies self.isVirtual = true

[C-3] If a calcHierarchy is defined, it must be a Hierarchy owned by the Dimension referenced by the CubeDimensionAssociation.

**context** CubeDimensionAssociation **inv:**
self.calcHierarchy->notEmpty implies self.calcHierarchy.dimension = self.dimension

[C-4] A "fully realized" CubeRegion has no MemberSelectionGroups (and hence, no MemberSelections).

**context** CubeRegion **inv:**
self.isFullyRealized implies self.memberSelectionGroup->isEmpty

[C-5] A CubeRegion defined by MemberSelections must have, for each Dimension of its owning Cube, a corresponding MemberSelection within each of its MemberSelectionGroups.

**context** CubeRegion **inv:**
self.memberSelectionGroup->notEmpty implies
self.cube.cubeDimensionAssociation->forAll( d |
self.memberSelectionGroup->forAll( g |
g.memberSelection->exists( m | m.dimension = d.dimension ) ) )

[C-6] A CubeRegion defined by MemberSelections must have, within each MemberSelectionGroup, a MemberSelection corresponding to each Dimension of its owning Cube.

**context** CubeRegion **inv:**
self.memberSelectionGroup->notEmpty implies
self.memberSelectionGroup->forAll( g |
g.memberSelection->forAll( m |
self.cube.cubeDimensionAssociation->exists( d | d.dimension = m.dimension ) ) )

[C-7] A Dimension may be a Time Dimension, a Measure Dimension, or neither, but never both types at the same time.

**context** Dimension **inv:**
not ( self.isTime and self.isMeasure )

[C-8] The default display Hierarchy (if defined) must be one of the Hierarchies owned by the Dimension.

**context** Dimension **inv:**
self.displayDefault->notEmpty implies self.hierarchy->includes( self.displayDefault )

[C-9] An instance of DimensionDeployment must be referenced exclusively by either a HierarchyLevelAssociation or a ValueBasedHierarchy.

**context** DimensionDeployment **inv:**
self.hierarchyLevelAssociation->isEmpty xor self.valueBasedHierarchy->isEmpty

[C-10] Within a DimensionDeployment, an "immediate parent" StructureMap must always have an associated and distinct "list of values" StructureMap.

**context** DimensionDeployment **inv:**
self.immediateParent->notEmpty implies
( self.listOfValues->notEmpty and self.listOfValues <> self.immediateParent )

[C-11] A StructureMap referenced as a "list of values" StructureMap must not reside outside of the DimensionDeployment's collection of StructureMaps.

**context** DimensionDeployment **inv:**
self.listOfValues->notEmpty implies self.structureMap->includes( self.listOfValues )

[C-12] A StructureMap referenced as an "immediate parent" StructureMap must not reside outside of the DimensionDeployment's collection of StructureMaps.

**context** DimensionDeployment **inv:**
self.immediateParent->notEmpty implies
self.structureMap->includes( self.immediateParent )

[C-13] The currentLevel of each HierarchyLevelAssociation must refer to a Level owned by the Dimension of the LevelBasedHierarchy containing the HierarchyLevelAssociation.

**context** LevelBasedHierarchy **inv:**
self.hierarchyLevelAssociation->notEmpty implies
self.hierarchyLevelAssociation->forAll( h |
self.dimension.memberSelection
->select( oclType = Olap::Level )->includes( h.currentLevel ) )

[C-14] No two HierarchyLevelAssociations may designate the same Level instance as their "current level".

**context** LevelBasedHierarchy **inv:**
self.hierarchyLevelAssociation->forAll( h1, h2 | h1 <> h2 implies
h1.currentLevel <> h2.currentLevel )

# Data Mining 15

## 15.1 Overview

*Data mining* is the application of mathematical or statistical processes for the purpose of extracting hidden knowledge from large data sets. This knowledge is subsequently used as actionable business intelligence.

Data mining techniques provide descriptive information that is manifest in inherent patterns or relations between the data. This can be achieved, for example, with algorithms for clustering or association rules detection (link analysis).

They also uncover correlations, often due to causal relationships, between the data and a specific target property. This information is used to make predictions about unknown data or future behavior. Techniques generating these models are known as *supervised learning algorithms*, and include classification and numeric prediction algorithms.

Whereas most analysis tools support the retrospective analysis of data sets by verifying a user's hypotheses, data mining attempts to discover trends and behaviors without the need for guessing about possible relationships.

Data mining tools are particularly effective in the data warehouse environment, because data warehouses offer large quantities of cleansed business data for consumption by data mining tools. Also, the advanced query and analytical capabilities available in most data warehouses (e.g., relational databases, OLAP servers, and information visualization tools) can be used to great advantage by data mining tools in their formulation of models, and in the evaluation of those models by human users.

## 15.2 Organization of the Data Mining Metamodel

### 15.2.1 Dependencies

The Data Mining package depends on the following packages:

- org.omg::CWM::ObjectModel::Core
- org.omg::CWM::ObjectModel::Instance

### 15.2.2 Major Classes and Associations

The CWM Data Mining metamodel represents three conceptual areas: The overall Model description itself, Settings, and Attributes. Each area is represented by the diagrams in Figure 15-1, Figure 15-2, and Figure 15-3, respectively.

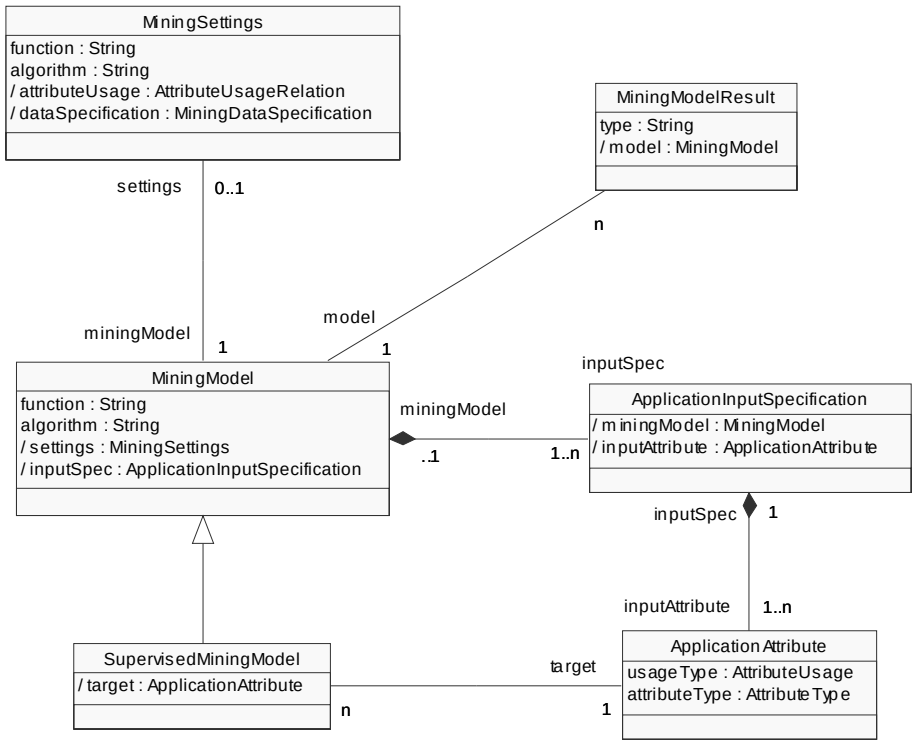


*Figure 15-1* CWM Data Mining Metamodel: Model

The Model conceptual area consists of a generic representation of a data mining model (that is, a mathematical model produced or generated by the execution of a data mining algorithm). This consists of MiningModel, a representation of the mining model itself, MiningSettings, which drive the construction of the model, ApplicationInputSpecification, which specifies the set of input attributes for the model, and MiningModelResult, which represents the result set produced by the testing or application of a generated model.

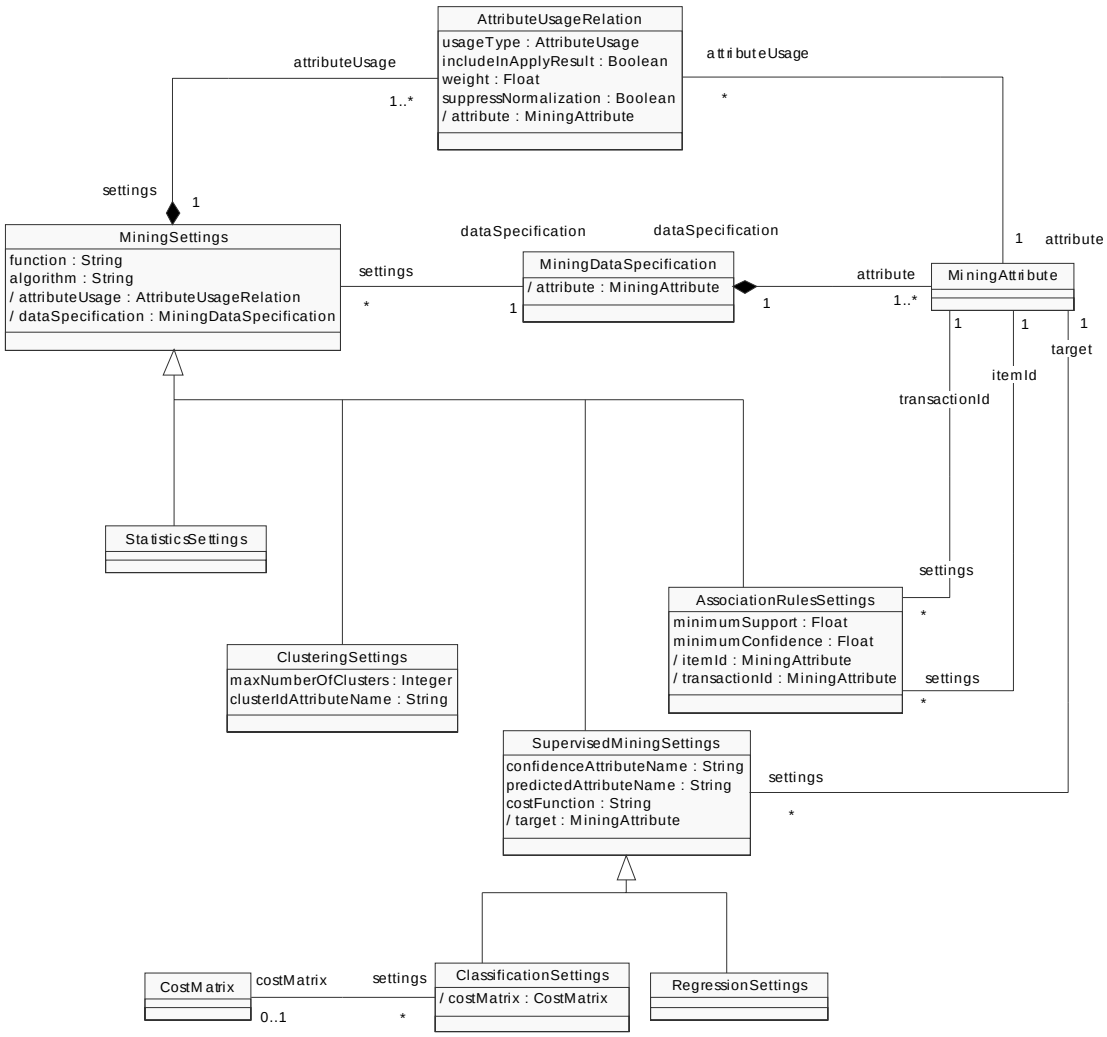


*Figure 15-2* CWM Data Mining Metamodel: Settings

The Settings conceptual area elaborates further on the Mining Settings and their usage relationships to the attributes of the input specification.

Mining Settings has four subclasses representing settings for StatisticsSettings, ClusteringSettings, SupervisedMiningSettings and AssociationRulesSettings. The SupervisedMiningSettings are further subclassed as ClassificationSettings and RegressionSettings, and a CostMatrix is defined for representing cost values associated with misclassifications.

AttributeUsageRelation consists of attributes that further classify the usage of MiningAttributes by MiningSettings (e.g., relative weight). Several associations are

also used to explicitly define requirements placed on attributes by certain subclasses of settings (e.g., target, transactionId and itemId).

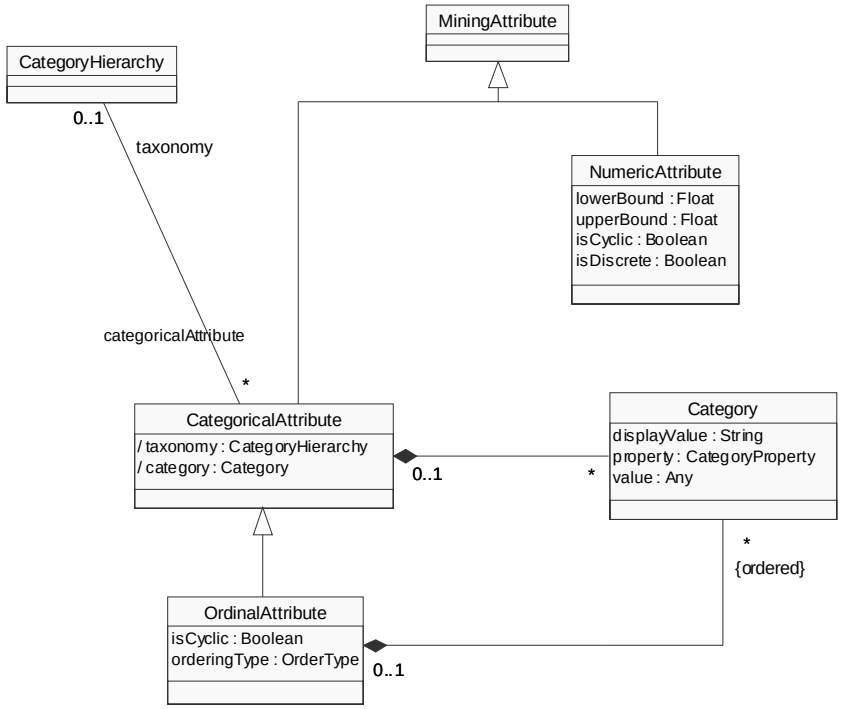


*Figure 15-3* CWM Data Mining Metamodel: Attributes

The Attributes conceptual area defines two subclasses of Mining Attribute: NumericAttribute and CategoricalAttribute. Category represents the category properties and values that either a CategoricalAttribute or OrdinalAttribute might possess, while CategoryHierarchy represents any taxonomy that a CategoricalAttribute might be associated with.

## *15.2.3 Inheritance from the ObjectModel*

The inheritance of the Data Mining metamodel from the CWM ObjectModel is shown in Figure 15-4 below.

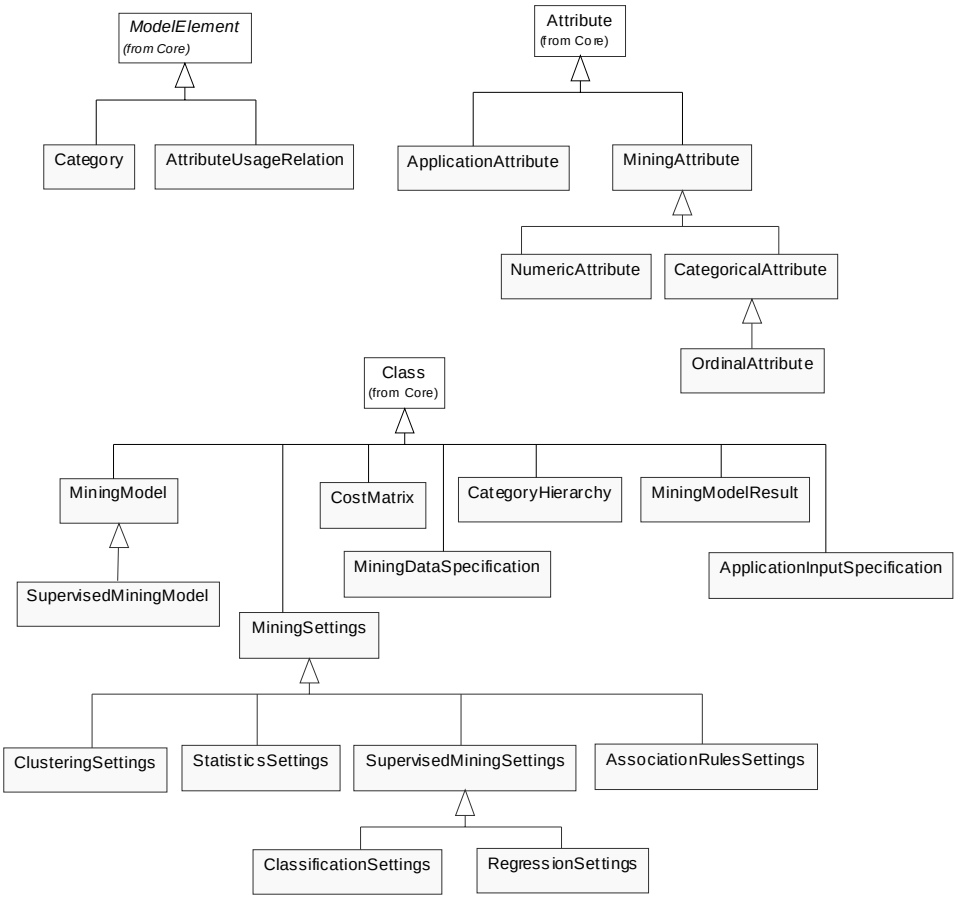


*Figure 15-4* CWM Data Mining Metamodel: Inheritance from UML

## 15.3 Data Mining Classes

### 15.3.1 ApplicationAttribute

Attribute used when the model was generated.

#### Superclasses

Attribute

### Attributes

***usageType***

Indicates whether attribute was actively used when the model was generated.

| | |
|---|---|
| *type:* | AttributeUsage ( active \| inactive \| supplementary ) |
| *multiplicity:* | exactly one |

***attributeType***

Type of ApplicationAttribute.

| | |
|---|---|
| *type:* | AttributeType ( categorical \| numerical ) |
| *multiplicity:* | exactly one |

## 15.3.2 ApplicationInputSpecification

ApplicationInputSpecification is a collection of ApplicationAttributes that drive the MiningModel.

### Superclasses

Class

### Contained Elements

ApplicationAttribute

### References

***inputAttribute***

ApplicationAttributes owned by the ApplicationInputSpecification.

| | |
|---|---|
| *class:* | ApplicationAttribute |
| *defined by:* | InputSpecOwnsAttributes::inputAttribute |
| *multiplicity:* | one or more |

***miningModel***

The MiningModel owning an ApplicationInputSpecification.

| | |
|---|---|
| *class:* | MiningModel |
| *defined by:* | MiningModelOwnsInputSpecification::miningModel |
| *multiplicity:* | one or more |
| *inverse:* | MiningModel::inputSpec |

## *15.3.3 AssociationRulesSettings*

Parameters for computing association rules.

### *Superclasses*

MiningSettings

### Attributes

***minimumSupport***

The minimum support required for association rules.

| | |
|---|---|
| *type:* | Float |
| *multiplicity:* | exactly one |

***minimumConfidence***

The minimum confidence required for association rules.

| | |
|---|---|
| *type:* | Float |
| *multiplicity:* | exactly one |

### References

***itemId***

References MiningAttribute as Item ID.

| | |
|---|---|
| *class:* | MiningAttribute |
| *defined by:* | UsesItemId::itemID |
| *multiplicity:* | exactly one |

***transactionId***

References MiningAttribute as Transaction ID.

| | |
|---|---|
| *class:* | MiningAttribute |
| *defined by:* | UsesTransactionId::transactionID |
| *multiplicity:* | exactly one |

### Constraints

Function must specify “AssociationRules”. [C-1]

## 15.3.4 AttributeUsageRelation

Parameters for mining activities that are specific for an attribute. Mining attribute usage is intended as a specification apart from the mining input specification itself to enable reuse of the mining input specification for different mining settings.

## Superclasses

ModelElement

## Attributes

### *usageType*

Indicates how the attribute is used by the mining function.

*type:* AttributeUsage( active | inactive | supplementary )

*multiplicity:* exactly one

### *includeInApplyResult*

Indicates whether the attribute is included in the output.

*type:* Boolean

*multiplicity:* exactly one

### *weight*

Relative weight of the attribute.

*type:* Float

*multiplicity:* exactly one

### *suppressNormalization*

Indicates whether normalization is to be suppressed.

*type:* Boolean

*multiplicity:* exactly one

## References

### *attribute*

Reference to the MiningAttribute.

*class:* MiningAttribute

*defined by:* PertainsToAttribute::attribute

*multiplicity:* exactly one

### 15.3.5 CategoricalAttribute

An attribute with discrete values upon which performing numeric operations is typically not meaningful.

#### Superclasses

MiningAttribute

#### Contained Elements

Category

#### References

***taxonomy***

References the taxonomy.

| | |
|---|---|
| *class:* | CategoryHierarchy |
| *defined by:* | UsesAsTaxonomy::taxonomy |
| *multiplicity:* | zero or one |

***category***

References the Category.

| | |
|---|---|
| *class:* | Category |
| *defined by:* | ContainsCategory::category |
| *multiplicity:* | zero or more |

#### Constraints

Category values must be unique. [C-2]

### 15.3.6 Category

A potential value of categoricalAttribute. For a given categoricalAttribute, all categories must be of the same Category subclass.

#### Superclasses

ModelElement

### Attributes

***displayValue***

A string used when the category is displayed.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

***property***

Categories with "missing" property represent that no information is available. If there are categories with property "invalid" then other categories are valid by default. If there are categories with property "valid" then other categories are invalid by default. Positive and negative define allowed values of a binary attribute.

| | |
|---|---|
| *type:* | CategoryProperty ( missing \| invalid \| valid \| positive \| negative ) |
| *multiplicity:* | exactly one |

***value***

Value holder for the Category.

| | |
|---|---|
| *type:* | Any |
| *multiplicity:* | exactly one |

### Constraints

An instance of Category must be owned by precisely one instance of CategoricalAttribute or any of its subclasses. [C-3]

## 15.3.7 CategoryHierarchy

Defines a hierarchical ordering (aka taxonomy) between groups of items.

### Superclasses

Class

## 15.3.8 ClassificationSettings

Parameters for computing a classification model.

### *Superclasses*

SupervisedMiningSettings

### *References*

***costMatrix***

References the CostMatrix.

| | |
|---|---|
| *class:* | CostMatrix |
| *defined by:* | UsesCostMatrix.costMatrix |
| *multiplicity:* | zero or one |

### *Constraints*

Function must specify “Classification”. [C-4]

## *15.3.9 ClusteringSettings*

Parameters for computing a clustering model partitioning the input records into segments.

### *Superclasses*

MiningSettings

### Attributes

***maxNumberOfClusters***

Upper limit for the number of computed clusters.

| | |
|---|---|
| *type:* | Integer |
| *multiplicity:* | exactly one |

***clusterIdAttributeName***

Attribute name for output of cluster id values.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### Constraints

Function must specify “Clustering”. [C-5]

maxNumberOfClusters must be positive. [C-6]

## 15.3.10 CostMatrix

Defines cost of misclassifications.

### Superclasses

Class

## 15.3.11 MiningAttribute

An attribute (aka field or variable). It carries the following user specifications:

- valid values
- ordering
- taxonomy
- normalization

### Superclasses

Attribute

### 15.3.12 MiningDataSpecification

The collection of mining attributes specifying how to interpret the input data attributes. A description of the attributes accepted by the model for scoring data.

#### Superclasses

Class

#### Contained Elements

MiningAttribute

#### References

***attribute***

References the MiningAttributes.

| | |
|---|---|
| *class:* | MiningAttribute |
| *defined by:* | HasAttribute::attribute |
| *multiplicity:* | one or more |

#### Constraints

Attributes must have unique names. [C-7]

### 15.3.13 MiningModel

Description of the data produced by a mining function.

#### Superclasses

Class

#### Contained Elements

ApplicationInputSpecification

## Attributes

### *function*

Names the mining function. The following functions are predefined: StatisticalAnalysis, FeatureSelection, AssociationRules, Classification, Clustering, Regression.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### *algorithm*

Names the algorithm used to perform the mining function. The following algorithms are predefined: decisionTree, neuralNetwork, naiveBayes, selfOrganizingMap, centerBasedClustering, distributionBasedClustering, associationRules, polynomialRegression, radialBasisFunction, ruleBasedClassification, principalComponentAnalysis, factorAnalysis, bivariateAnalysis, descriptiveAnalysis, geneticAlgorithm

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

## References

### *settings*

References the mining model's settings.

| | |
|---|---|
| *class:* | MiningSettings |
| *defined by:* | DerivedFromSettings::settings |
| *multiplicity:* | zero or one |

### *inputSpec*

References the MiningAttributes.

| | |
|---|---|
| *class:* | ApplicationInputSpecification |
| *defined by:* | MiningModelOwnsInputSpecification::inputSpec |
| *multiplicity:* | one or more |

## Constraints

Function and algorithm must be equal to function and algorithm of MiningSettings, respectively. [C-8]

### 15.3.14 MiningModelResult

Describes the result set produced by a run of a MiningModel.

#### Superclasses

Class

#### Attributes

***type***

Type of information contained in mining result. The following types are predefined: sensitivityResult, liftResult, classificationEvalResult, regressionEvalResult

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

#### References

***model***

References the MiningModel associated with this instance of MiningModelResult.

| | |
|---|---|
| *class:* | MiningModel |
| *defined by:* | ProducedByModel.model |
| *multiplicity:* | exactly one |

### 15.3.15 MiningSettings

Parameters for mining activities. Mining settings indicate how the model should be or was built.

#### Superclasses

Class

#### Contained Elements

AttributeUsageRelation

## Attributes

### *function*

Names the mining function. The following functions are predefined: StatisticalAnalysis, FeatureSelection, AssociationRules, Classification, Clustering, Regression

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### *algorithm*

Names the algorithm used to perform the mining function. The following algorithms are predefined: decisionTree, neuralNetwork, naiveBayes, selfOrganizingMap, centerBasedClustering, distributionBasedClustering associationRules, polynomialRegression, radialBasisFunction, ruleBasedClassification, principalComponentAnalysis, factorAnalysis, bivariateAnalysis, descriptiveAnalysis, geneticAlgorithm

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

## References

### *attributeUsage*

References the AttributeUsageRelations.

| | |
|---|---|
| *class:* | AttributeUsageRelation |
| *defined by:* | ContainsAttributeUsage::attributeUsage |
| *multiplicity:* | one or more |

### *dataSpecification*

References the MiningDataSpecification.

| | |
|---|---|
| *class:* | MiningDataSpecification |
| *defined by:* | UsesAsInput::dataSpecification |
| *multiplicity:* | exactly one |

### 15.3.16 NumericAttribute

Attribute containing numbers, for which numeric operations are meaningful.

#### Superclasses

MiningAttribute

#### Attributes

***lowerBound***

Least non-outlier value.

| | |
|---|---|
| *type:* | Float |
| *multiplicity:* | exactly one |

***upperBound***

Greatest non-outlier value.

| | |
|---|---|
| *type:* | Float |
| *multiplicity:* | exactly one |

***isCyclic***

Indicates attributes with cyclic value range such as angles or numbers representing the day of the week. If true, lowerBound and upperBound define the base interval.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

***isDiscrete***

Tells the algorithms whether to deal with the numbers as discrete values.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

#### Constraints

lowerBound must be less than or equal to upperBound. [C-9]

### 15.3.17 OrdinalAttribute

Subclass of CategoricalAttribute that represents ordinal attributes.

#### Superclasses

CategoricalAttribute

#### Attributes

***isCyclic***

Indicates ordinal attributes with cyclic value ranges, in which case the first and last attributes in the ordered sequence of attributes define the base interval.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

***orderingType***

Indicates if categories are ordered. If orderingType is inSequence then the aggregation of categories defines the ordering relation.

| | |
|---|---|
| *type:* | OrderType |
| *multiplicity:* | exactly one |

### 15.3.18 RegressionSettings

Parameters for computing a regression model.

#### Superclasses

SupervisedMiningSettings

#### Constraints

Function must specify “Regression”. [C-10]

### 15.3.19 StatisticsSettings

Parameters for computing statistics-based models.

### *Superclasses*

MiningSettings

### *Constraints*

Function must specify “StatisticalAnalysis”. [C-11]

## *15.3.20 SupervisedMiningModel*

Description of data produced by a predictive mining function.

### *Superclasses*

MiningModel

### *References*

***target***

References the "target" ApplicationAttribute instance.

| | |
|---|---|
| *class:* | ApplicationAttribute |
| *defined by:* | SupervisedMiningModelReferencesTargetAttribute ::target |
| *multiplicity:* | exactly one |

## *15.3.21 SupervisedMiningSettings*

Parameters for computing a supervised model, i.e., one that requires a target attribute against which to measure model accuracy.

### *Superclasses*

MiningSettings

## Attributes

### *confidenceAttributeName*

Attribute name for output of confidence values of the prediction.

*type:* String

*multiplicity:* exactly one

### *predictedAttributeName*

Attribute name for output of predicted values.

*type:* String

*multiplicity:* exactly one

### *costFunction*

Function specifying the cost of incorrect predictions. Predefined methods are: entropy, Gini, costMatrix, pnorm, none.

*type:* String

*multiplicity:* exactly one

## References

### *target*

Reference MiningAttribute as Target.

*class:* MiningAttribute

*defined by:* UsesAsTarget::target

*multiplicity:* exactly one

## 15.4 Data Mining Associations

### 15.4.1 ContainsAttributeUsage

Settings may have one or more AttributeUsageRelations.

*Ends*

***settings***

The Settings owning AttributeUsageRelations.

| | |
|---|---|
| *class:* | MiningSettings |
| *multiplicity:* | exactly one |
| *aggregation:* | composite |

***attributeUsage***

AttributeUsageRelations owned by the Settings.

| | |
|---|---|
| *class:* | AttributeUsageRelation |
| *multiplicity:* | one or more |

### 15.4.2 ContainsCategory

A CategoricalAttribute may have zero or more Categories.

*Ends*

***categoricalAttribute***

The CategoricalAttribute owning Categories.

| | |
|---|---|
| *class:* | CategoricalAttribute |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

***category***

Categories owned by the CategoricalAttribute.

| | |
|---|---|
| *class:* | Category |
| *multiplicity:* | zero or more |

### 15.4.3 DerivedFromSettings

A mining model is derived from its settings.

*Ends*

***miningModel***

The mining model derived from its settings.

| | |
|---|---|
| *class:* | MiningModel |
| *multiplicity:* | exactly one |

***settings***

The settings used to derive the mining model.

| | |
|---|---|
| *class:* | MiningSettings |
| *multiplicity:* | zero or one |

### 15.4.4 HasAttribute

MiningDataSpecification owns one or more MiningAttributes.

*Ends*

***dataSpecification***

The MiningDataSpecification owning MiningAttributes.

| | |
|---|---|
| *class:* | MiningDataSpecification |
| *multiplicity:* | exactly one |
| *aggregation:* | composite |

***attribute***

MiningAttributes owned by the MiningDataSpecification.

| | |
|---|---|
| *class:* | MiningAttribute |
| *multiplicity:* | one or more |

### 15.4.5 InputSpecOwnsAttributes

An ApplicationInputSpecification owns one or more ApplicationAttributes.

*Ends*

***inputSpec***

The ApplicationInputSpecification owning ApplicationAttributes.

| | |
|---|---|
| *class:* | ApplicationInputSpecification |
| *multiplicity:* | exactly one |
| *aggregation:* | composite |

***inputAttribute***

ApplicationAttributes owned by the ApplicationInputSpecification.

| | |
|---|---|
| *class:* | ApplicationAttribute |
| *multiplicity:* | one or more |

### 15.4.6 MiningModelOwnsInputSpecification

A MiningModel owns at most one ApplicationInputSpecification.

*Ends*

***miningModel***

The MiningModel owning an ApplicationInputSpecification.

| | |
|---|---|
| *class:* | MiningModel |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

***inputSpec***

An ApplicationInputSpecification owned by a MiningModel.

| | |
|---|---|
| *class:* | ApplicationInputSpecification |
| *multiplicity:* | one or more |

## 15.4.7 OrdersCategory

An OrdinalAttribute has an ordered collection of zero or more Categories.

### Ends

***ordinalAttribute***

The OrdinalAttribute that orders instances of Category.

| | |
|---|---|
| *class:* | OrdinalAttribute |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

***category***

The instances of Category ordered by an instance of OrdinalAttribute

| | |
|---|---|
| *class:* | Category |
| *multiplicity:* | zero or more; ordered |

## 15.4.8 PertainsToAttribute

An AttributeUsageRelation references a single MiningAttribute.

### Ends

***attributeUsage***

AttributeUsageRelations referencing the MiningAttribute.

| | |
|---|---|
| *class:* | AttributeUsageRelation |
| *multiplicity:* | zero or more |

***attribute***

The MiningAttribute referenced by AttributeUsageRelations

*class:* MiningAttribute

*multiplicity:* exactly one

### 15.4.9 ProducedByModel

A mining model has a result set produced by a run of the model.

*Ends*

***miningResult***

The result set produced by the mining model.

*class:* MiningModelResult

*multiplicity:* zero or more

***model***

The mining model that produced the result set.

*class:* MiningModel

*multiplicity:* exactly one

### 15.4.10 SupervisedMiningModelReferencesTargetAttribute

Each instance of SupervisedMiningModel references a single instance of ApplicationAttribute as its “target”.

*Ends*

***target***

The “target” ApplicationAttribute referenced by SupervisedMiningModels.

*class:* ApplicationAttribute

*multiplicity:* exactly one

***supervisedMiningModel***

SupervisedMiningModels referencing the ApplicationAttribute as their “target”.

*class:* SupervisedMiningModel

*multiplicity:* zero or more

### 15.4.11 UsesAsInput

Settings references one MiningDataSpecification.

#### Ends

***settings***

MiningSettings referencing the MiningDataSpecification.

*class:* MiningSettings

*multiplicity:* zero or more

***dataSpecification***

The MiningDataSpecification referenced by the MiningSettings.

*class:* MiningData Specification

*multiplicity:* exactly one

### 15.4.12 UsesAsTarget

The attribute that contains the target values for building supervised models.

#### Ends

***settings***

The SupervisedMiningSettings referencing the MiningAttribute as a Target.

*class:* SupervisedMiningSettings

*multiplicity:* zero or more

***target***

The MiningAttribute referenced by SupervisedMiningSettings as a Target.

| | |
|---|---|
| *class:* | MiningAttribute |
| *multiplicity:* | exactly one |

### *15.4.13 UsesAsTaxonomy*

CategoricalAttributes may reference a taxonomy.

*Ends*

***categoricalAttribute***

The CategoricalAttributes referencing a taxonomy.

| | |
|---|---|
| *class:* | CategoricalAttribute |
| *multiplicity:* | zero or more |

***taxonomy***

The taxonomy referenced by CategoricalAttributes

| | |
|---|---|
| *class:* | CategoryHierarchy |
| *multiplicity:* | zero or one |

### *15.4.14 UsesCostMatrix*

ClassificationSettings may reference a CostMatrix.

*Ends*

***settings***

ClassificationSettings referencing the CostMatrix

| | |
|---|---|
| *class:* | ClassificationSettings |
| *multiplicity:* | zero or more |

***costMatrix***

The CostMatrix referenced by ClassificationSettings.

| | |
|---|---|
| *class:* | CostMatrix |
| *multiplicity:* | zero or one |

## 15.4.15 UsesItemId

The attribute that contains the item identifier needed for building the association rules model.

### Ends

***settings***

The AssociationRulesSettings referencing the MiningAttribute as an Item ID.

| | |
|---|---|
| *class:* | AssociationRulesSettings |
| *multiplicity:* | zero or more |

***itemID***

The MiningAttribute referenced by AssociationRulesSettings as an Item ID.

| | |
|---|---|
| *class:* | MiningAttribute |
| *multiplicity:* | exactly one |

## 15.4.16 UsesTransactionId

The attribute that contains the transaction identifier needed for building the association rules model.

*Ends*

***settings***

The AssociationRulesSettings referencing the MiningAttribute as a Transaction ID.

| | |
|---|---|
| *class:* | AssociationRulesSettings |
| *multiplicity:* | zero or more |

***transactionID***

The MiningAttribute referenced by AssociationRulesSettings as a Transaction ID.

| | |
|---|---|
| *class:* | MiningAttribute |
| *multiplicity:* | exactly one |

## 15.5 OCL Representation of Data Mining Constraints

[C-1] Function must specify “AssociationRules”.

**context** AssociationRulesSettings **inv:**

self.function = “AssociationRules”

[C-2] Category values must be unique.

**context** CategoricalAttribute **inv:**

self.category->forAll( c1, c2 | c1.value = c2.value implies c1 = c2 )

[C-3] An instance of Category must be owned by precisely one instance of CategoricalAttribute or any of its subclasses.

**context** Category **inv:**

self.categoricalAttribute->isEmpty xor self.ordinalAttribute->isEmpty

[C-4] Function must specify “Classification”.

**context** ClassificationSettings **inv:**

self.function = “Classification”

[C-5] Function must specify “Clustering”.

**context** ClusteringSettings **inv:**

self.function = “Clustering”

[C-6] maxNumberOfClusters must be positive.

**context** ClusteringSettings **inv:**

self.maxNumberOfClusters > 0

[C-7] Attributes must have unique names.

**context** MiningDataSpecification **inv:**

self.attribute->forAll( a1, a2 | a1.name = a2.name implies a1 = a2 )

[C-8] Function and algorithm must be equal to function and algorithm of MiningSettings, respectively.

**context** MiningModel **inv:**

self.settings->isEmpty or (self.function = self.settings.function and

self.algorithm = self.settings.algorithm)

[C-9] lowerBound must be less than or equal to upperBound.

**context** NumericAttribute **inv:**

self.lowerBound <= self.upperBound

[C-10] Function must specify “Regression”.

**context** RegressionSettings **inv:**

self.function = “Regression”

[C-11] Function must specify “StatisticalAnalysis”.

**context** StatisticsSettings **inv:**

self.function = “StatisticalAnalysis”

# Information Visualization 16

## 16.1 Overview

The CWM Information Visualization metamodel defines metadata supporting the problem domain of "information publishing" or, more generally, "information visualization".

Within the data warehousing environment, data is collected from numerous, diverse sources and transformed into a unified representation that facilitates the analysis of data for purposes of gaining business insight. Robust and flexible information visualization tools are key to the effective analysis of this information. Information visualization tools must be capable of understanding and preserving the "logical structure" of data warehouse information, while enabling the user to perform any number of "rendering transformations" on information content (e.g., displaying the same query result set in several different formats, such as a printed report, Web page, pie chart, bar graph, etc.).

Since information visualization is a very broad problem domain, with a diverse set of possible solutions and many evolving standards, the CWM Information Visualization metamodel defines very generic, container-like metadata constructs that either contain or reference more complex visualization mechanisms at the M1-level. These metadata structures are intended to support the minimal metadata required to interchange more complex M1 models of visualization mechanisms.

## 16.2 Organization of the Information Visualization Metamodel

### 16.2.1 Dependencies

The Information Visualization package depends on the following packages:

org.omg.cwm.objectmodel.core

org.omg.cwm.foundation.expressions

## 16.2.2 Major Classes and Associations

The major classes and associations of the Information Visualization metamodel are shown in Figure 16-1.

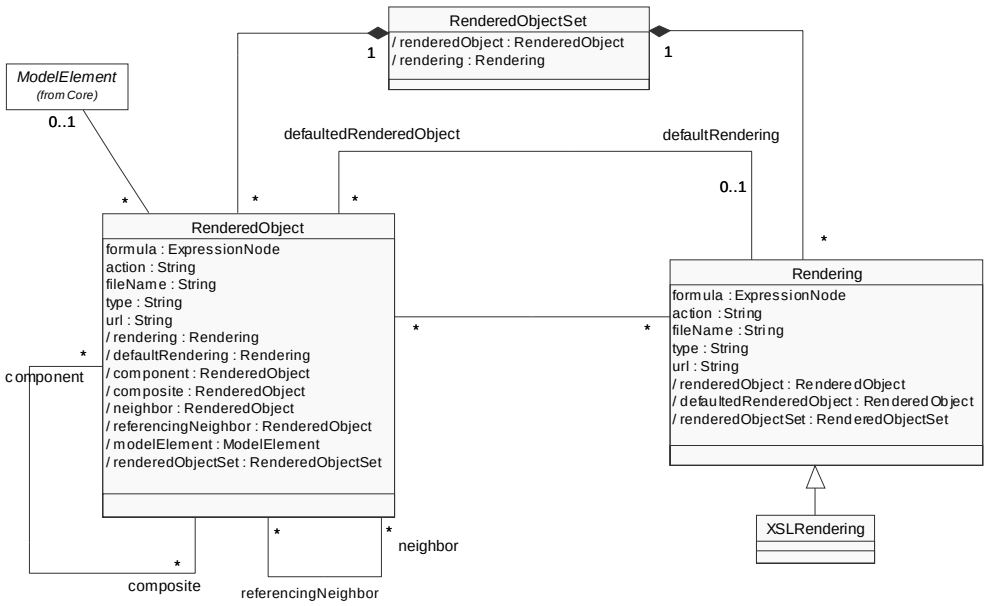


*Figure 16-1* CWM Information Visualization Metamodel

RenderedObject is the logical proxy for an arbitrary ModelElement that is to be rendered via some rendering transformation or process.

A RenderedObject may be composed of an arbitrary number of other RenderedObjects (i.e., components), and may have topological relationships to still other RenderedObjects. The formula attribute allows for the specification of any implementation-dependent expression that completes the definition of a RenderedObject. For example, the formula might specify the position of the RenderedObject within a two-dimensional grid, or in relation to one of its neighbors; e.g., formula = "neighbor(x, y) + (delta-x, delta-y)".

A RenderedObject generally references one or more Renderings that specify how the RenderedObject is actually presented. One of these associated Renderings may optionally be designated as a default Rendering.

A Rendering is semantically equivalent to a transformation, in that it transforms a source RenderedObject to some target "displayed" (or otherwise "presented" object -- e.g., a displayed image or an audio clip) . An instance of Rendering is fully specified via its formula attribute, which, like RenderedObject, contains an implementation-dependent expression.

Thus, a RenderedObject may be viewed as the "logical description" of an object to be rendered, independently of how it is actually presented by any of its associated Renderings, and Renderings may be viewed as transformations that control the presentation of the RenderedObject while preserving its logical structure.

Note that a RenderedObject may be the target of a complex transformation (i.e., utilizing the CWM Transformation package). For example, an N-dimensional OLAP cube might be transformed into an equivalent, two-dimensional, composite RenderedObject, with several dimensions mapped to row and column edges, respectively, and all other dimensions constrained to single member values. Several Renderings may then be defined and associated with the resultant RenderedObject, mapping the two-dimensional logical structure to the surface of a display screen in various different formats (e.g., spreadsheet, pie chart, bar graph, etc.).

Possible types of Renderings include: Screen, paper, voice, Web, HTML documents, XML/XSL, languages based on extensions to XML, SVG, visual objects, responses to keying (e.g., keying interception plus rules), etc.

XSLRendering represents a useful subtype of Rendering that's based on XSL (e.g., this subtype's formula might contain a procedure that uses XSL to create an HTML document).

Finally, RenderedObjectSet represents a simple container of both logical RenderedObjects and available Renderings.

## 16.3 Inheritance from the Object Model

The inheritance of the Information Visualization metamodel from the Object Model is shown in Figure 16-2 below.

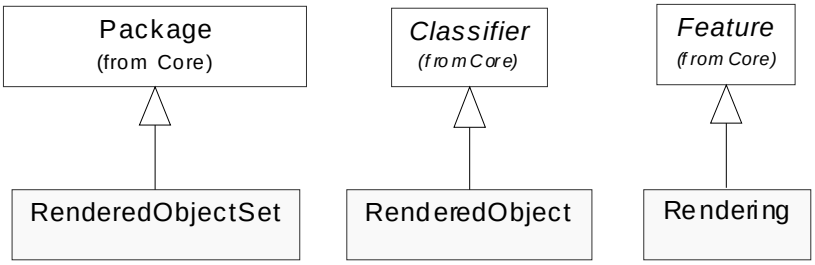


*Figure 16-2* CWM Information Visualization Metamodel: Inheritance

## 16.4 Information Visualization Classes

### 16.4.1 RenderedObject

RenderedObject serves as a logical "proxy" for an arbitrary ModelElement that is to be rendered.

#### Superclasses

Classifier

#### Attributes

***formula***

Allows for the specification of any implementation-dependent expression that completes the definition of a RenderedObject.

| | |
|---|---|
| *type:* | ExpressionNode |
| *multiplicity:* | exactly one |

***action***

Specifies some implementation-dependent action associated with a RenderedObject.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

***fileName***

Specifies the name of a file persisting an instance of RenderedObject.

*type:* String

*multiplicity:* exactly one

***type***

Specifies some implementation-dependent type associated with a RenderedObject

*type:* String

*multiplicity:* exactly one

***url***

Specifies a URL identifying some instance of RenderedObject.

*type:* String

*multiplicity:* exactly one

## References

### *rendering*

References the collection of Renderings associated with a RenderedObject.

| | |
|---|---|
| *class:* | Rendering |
| *defined by:* | RenderedObjectsReferenceRenderings::rendering |
| *multiplicity:* | zero or more |
| *inverse:* | Rendering::renderedObject |

### *defaultRendering*

References the default Rendering within the collection of Renderings associated with a RenderedObject.

| | |
|---|---|
| *class:* | Rendering |
| *defined by:* | RenderedObjectsReferenceDefaultRendering ::defaultRendering |
| *multiplicity:* | zero or one |

### *component*

References the collection of “component” RenderedObjects comprising this “composite” RenderedObject.

| | |
|---|---|
| *class:* | RenderedObject |
| *defined by:* | CompositesReferenceComponents::component |
| *multiplicity:* | zero or more |
| *inverse:* | RenderedObject::composite |

### *composite*

References the collection of "composite" RenderedObjects of which this RenderedObject is a "component".

| | |
|---|---|
| *class:* | RenderedObject |
| *defined by:* | CompositesReferenceComponents::composite |
| *multiplicity:* | zero or more |
| *inverse:* | RenderedObject::component |

***neighbor***

References the collection of RenderedObjects that are "neighbors" to this RenderedObject.

| | |
|---|---|
| *class:* | RenderedObject |
| *defined by:* | NeighborsReferenceNeighbors::neighbor |
| *multiplicity:* | zero or more |

***referencingNeighbor***

References the collection of RenderedObjects that reference this RenderedObject as a "neighbor".

| | |
|---|---|
| *class:* | RenderedObject |
| *defined by:* | NeighborsReferenceNeighbors::referencingNeighbor |
| *multiplicity:* | zero or more |

***modelElement***

References the ModelElement that a RenderedObject represents.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | RenderedObjectsReferenceModelElement ::modelElement |
| *multiplicity:* | zero or one |

***renderedObjectSet***

References the RenderedObjectSet owning a RenderedObject.

| | |
|---|---|
| *class:* | RenderedObjectSet |
| *defined by:* | RenderedObjectSetOwnsRenderedObjects ::renderedObjectSet |
| *multiplicity:* | exactly one |

## *Constraints*

The set of Renderings includes the default Rendering. [C-1]

A RenderedObject may not reference itself as a Neighbor nor as a Component. [C-2]

The transitive closure of Neighbors of an instance of RenderedObject must not include the RenderedObject instance.

The transitive closure of Components of an instance of RenderedObject must not include the RenderedObject instance.

A RenderedObject may not reference one of its Neighbors as a Component (and vice versa). [C-3]

## 16.4.2 RenderedObjectSet

RenderedObjectSet is a container of RenderedObjects and available Renderings.

### Superclasses

Package

### Contained Elements

- RenderedObject
- Rendering

### References

***renderedObject***

References the collection of RenderedObjects owned by a RenderedObjectSet.

| | |
|---|---|
| *class:* | RenderedObject |
| *defined by:* | RenderedObjectSetOwnsRenderedObjects::renderedObject |
| *multiplicity:* | zero or more |
| *inverse:* | RenderedObject::renderedObjectSet |

***rendering***

References the collection of Renderings owned by a RenderedObjectSet.

| | |
|---|---|
| *class:* | Rendering |
| *defined by:* | RenderedObjectSetOwnsRenderings ::rendering |
| *multiplicity:* | zero or more |
| *inverse:* | Rendering::renderedObjectSet |

### 16.4.3 Rendering

Rendering is a specification of how an associated RenderedObject is to be "rendered" in some medium. This usually consists of a projection of an object of arbitrary dimensionality onto a two-dimensional surface, but it may also include non-physical representations as well (such as audio).

#### Superclasses

Feature

#### Attributes

***formula***

Implementation-dependent procedure for generating the Rendering (e.g., a usage of XSL to generate an HTML document). Tracks the transformation lineage of the Rendering.

| | |
|---|---|
| *type:* | ExpressionNode |
| *multiplicity:* | exactly one |

***action***

Specifies some implementation-dependent action associated with a Rendering.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

***fileName***

Specifies the name of a file persisting an instance of Rendering.

*type:* String

*multiplicity:* exactly one

***type***

Specifies some implementation-dependent type associated with a Rendering.

*type:* String

*multiplicity:* exactly one

***url***

Specifies a URL identifying some instance of Rendering.

*type:* String

*multiplicity:* exactly one

## References

### *renderedObject*

References the collection of RenderedObjects that are associated with this Rendering.

| | |
|---|---|
| *class:* | RenderedObject |
| *defined by:* | RenderedObjectsReferenceRenderings:: renderedObject |
| *multiplicity:* | zero or more |
| *inverse:* | RenderedObject::rendering |

### *defaultedRenderedObject*

References the collection of RenderedObjects whose default Renderings are represented by this Rendering.

| | |
|---|---|
| *class:* | RenderedObject |
| *defined by:* | RenderedObjectsReferenceRenderings:: defaultedRenderedObject |
| *multiplicity:* | zero or more |
| *inverse:* | RenderedObject::rendering |

### *renderedObjectSet*

References the RenderedObjectSet owning a Rendering.

| | |
|---|---|
| *class:* | RenderedObjectSet |
| *defined by:* | RenderedObjectSetOwnsRenderings:: renderedObjectSet |
| *multiplicity:* | exactly one |
| *inverse:* | RenderedObjectSet::rendering |

### 16.4.4 XSLRendering

XSLRendering represents a useful subclass of Rendering based on XSL (i.e., the formula of this subclass might contain a procedure that uses XSL to create an HTML document).

#### Superclasses

Rendering

## 16.5 Information Visualization Associations

### 16.5.1 CompositesReferenceComponents

A RenderedObject may reference one or more "component" RenderedObjects, from which it is logically composed.

#### Ends

***components***

"Component" RenderedObjects referenced by "composite" RenderedObjects.

| | |
|---|---|
| *class:* | RenderedObject |
| *multiplicity:* | zero or more |

***composites***

"Composite" RenderedObjects referencing "component" RenderedObjects.

| | |
|---|---|
| *class:* | RenderedObject |
| *multiplicity:* | zero or more |

### 16.5.2 NeighborsReferenceNeighbors

A RenderedObject may reference one or more “neighboring” RenderedObjects.

*Ends*

***neighbor***

RenderedObjects referenced by this RenderedObject as its “neighbor” (or neighboring object).

| | |
|---|---|
| *class:* | RenderedObject |
| *multiplicity:* | zero or more |

***referencingNeighbor***

RenderedObjects referencing this RenderedObject as its “neighbor”.

| | |
|---|---|
| *class:* | RenderedObject |
| *multiplicity:* | zero or more |

## *16.5.3 RenderedObjectSetOwnsRenderedObjects*

A RenderedObjectSet may own any number of RenderedObjects.

*Ends*

***renderedObject***

RenderedObjects owned by a RenderedObjectSet.

| | |
|---|---|
| *class:* | RenderedObject |
| *multiplicity:* | zero or more |

***renderedObjectSet***

RenderedObjectSet owning RenderedObjects.

| | |
|---|---|
| *class:* | RenderedObjectSet |
| *multiplicity:* | exactly one |

## *16.5.4 RenderedObjectSetOwnsRenderings*

A RenderedObjectSet may own any number of Renderings.

*Ends*

***rendering***

Renderings owned by a RenderedObjectSet.

| | |
|---|---|
| *class:* | Rendering |
| *multiplicity:* | zero or more |

***renderedObjectSet***

RenderedObjectSet owning Renderings.

| | |
|---|---|
| *class:* | RenderedObjectSet |
| *multiplicity:* | exactly one |

## *16.5.5 RenderedObjectsReferenceDefaultRendering*

A RenderedObject may reference a default Rendering.

*Ends*

***defaultRendering***

The Rendering referenced by one or more RenderedObjects as the default Rendering.

| | |
|---|---|
| *class:* | Rendering |
| *multiplicity:* | zero or one |

***defaultedRenderedObject***

RenderedObjects referencing this Rendering as the default Rendering.

| | |
|---|---|
| *class:* | RenderedObject |
| *multiplicity:* | zero or more |

## *16.5.6 RenderedObjectsReferenceModelElement*

One or more RenderedObjects may reference an arbitrary ModelElement.

*Ends*

***renderedObject***

RenderedObjects referencing the ModelElement.

| | |
|---|---|
| *class:* | RenderedObject |
| *multiplicity:* | zero or more |

***modelElement***

The ModelElement referenced by the RenderedObjects.

| | |
|---|---|
| *class:* | ModelElement |
| *multiplicity:* | zero or one |

### *16.5.7 RenderedObjectsReferenceRenderings*

A RenderedObject may reference any number of Renderings. A Rendering may be referenced by any number of RenderedObjects.

*Ends*

***rendering***

Renderings referenced by RenderedObjects.

| | |
|---|---|
| *class:* | Rendering |
| *multiplicity:* | zero or more |

***renderedObject***

RenderedObjects referencing Renderings.

| | |
|---|---|
| *class:* | RenderedObject |
| *multiplicity:* | zero or more |

## *16.6 OCL Representation of Information Visualization Constraints*

[C-1] The set of Renderings includes the default Rendering.

**context** RenderedObject **inv:**

self.defaultRendering->notEmpty implies

self.rendering->includes( self.defaultRendering )

[C-2] A RenderedObject may not reference itself as a Neighbor nor as a Component.

**context** RenderedObject

**inv:** self.neighbor->excludes( self )

**inv:** self.component->excludes( self )

[C-3] A RenderedObject may not reference one of its Neighbors as a Component (and vice versa).

**context** RenderedObject **inv:**

(self.neighbor->notEmpty and self.component->notEmpty) implies

self.neighbor->intersection( self.component )->isEmpty

# Business Nomenclature 17

## 17.1 Overview

Business users of data warehouses need to have a good understanding of what information and tools exist in a data warehouse. They need to understand what the information means from a business perspective, how it is derived, from what data resources it is derived, and what analysis and reporting tools exist for manipulating and reporting the information. They may also need to subscribe to analysis and reporting tools, and have them run with results delivered to them on a regular basis.

The BusinessNomenclature package contains classes and associations that can be used to represent business metadata. Easy access to this business metadata enables business users to exploit the value of the information in a data warehouse. It can also aid technical users in certain tasks. An example is the use of common business terms and concepts for discussing information requirements with business users. Another example is accessing business intelligence tools for analyzing the impact of warehouse design changes.

The scope of the BusinessNomenclature package is restricted to the domain of data warehousing and business intelligence.

### 17.1.1 Semantics

This section provides a description of the main features of the BusinessNomenclature package.

The BusinessNomenclature package provides two main constructs to represent business terms and concepts and related semantics:

- *Taxonomy* is a collection of concepts that provide the context for the meaning of a particular term.
- *Glossary* is a collection of terms and various related forms of the term.

A *taxonomy* is a collection of *concepts*. Concepts represent semantic information and relationships. Concepts are identified by *terms*, which in turn are manifested by a word or phrase. More than one term may describe the same concept and a given term may describe more than one concept.

A *glossary* is a collection of terms that are related through explicit or implicit relationships. Terms may be *preferred* (the term best representing its concept) and thus represent the vocabulary of a business domain or user. Terms may be *synonyms* and point at the preferred term. A preferred term and its synonyms represent the fact that several terms describe the same concept although with different shades of meaning. Terms may be arranged into a hierarchy of more generic and more specific elements. This relationship allows substituting a *narrower term*, such as "USA", for a *wider term*, such as "country".

## 17.2 Organization of the Business Nomenclature Package

The BusinessNomenclature package depends on the following packages:

- omg.org::CWM::ObjectModel::Core

The metamodel diagram for the BusinessNomenclature package is split into two parts. The first diagram shows the BusinessNomenclature classes and associations, while the second shows the inheritance hierarchy.

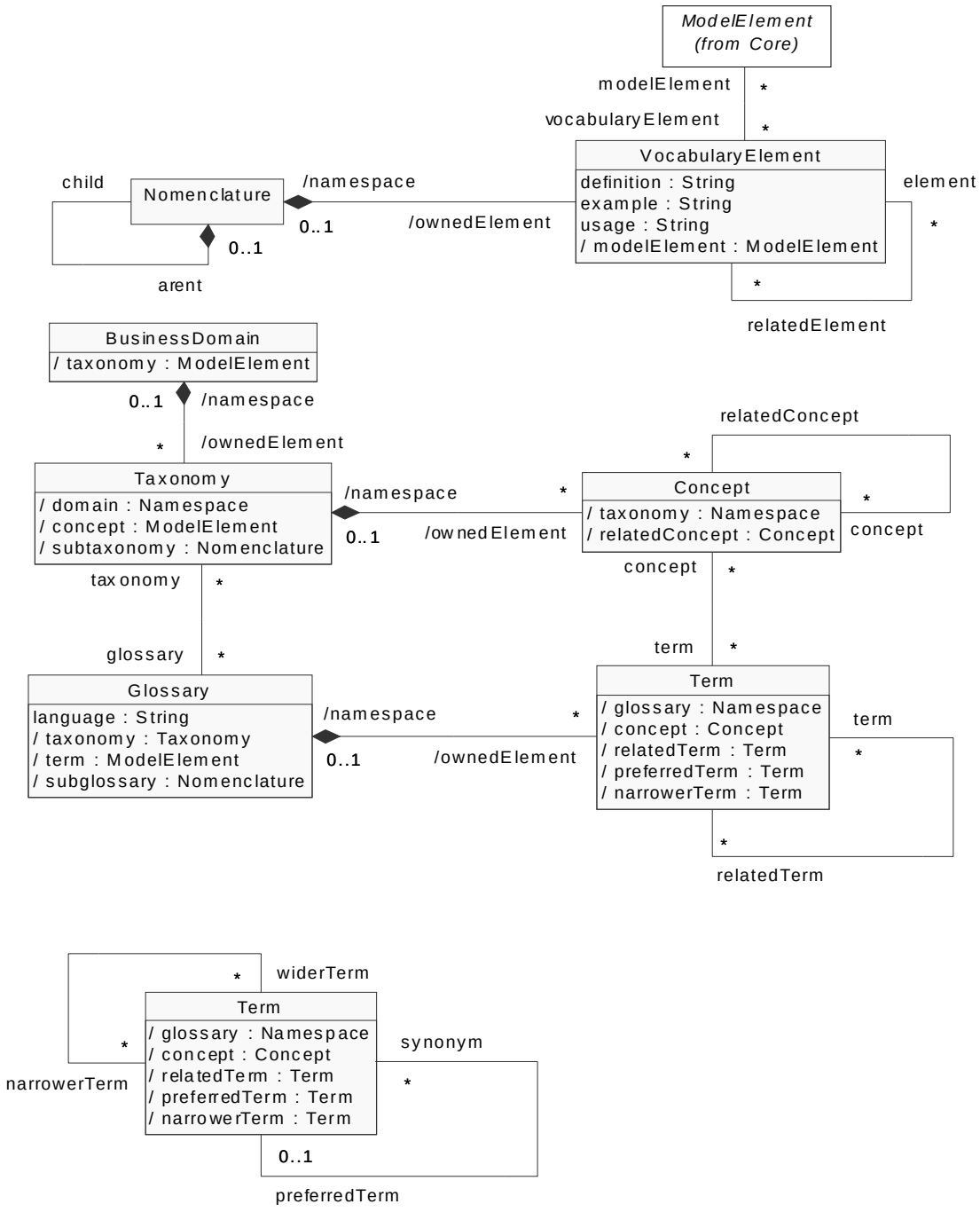


*Figure 17-1* BusinessNomenclature Package: Relationships

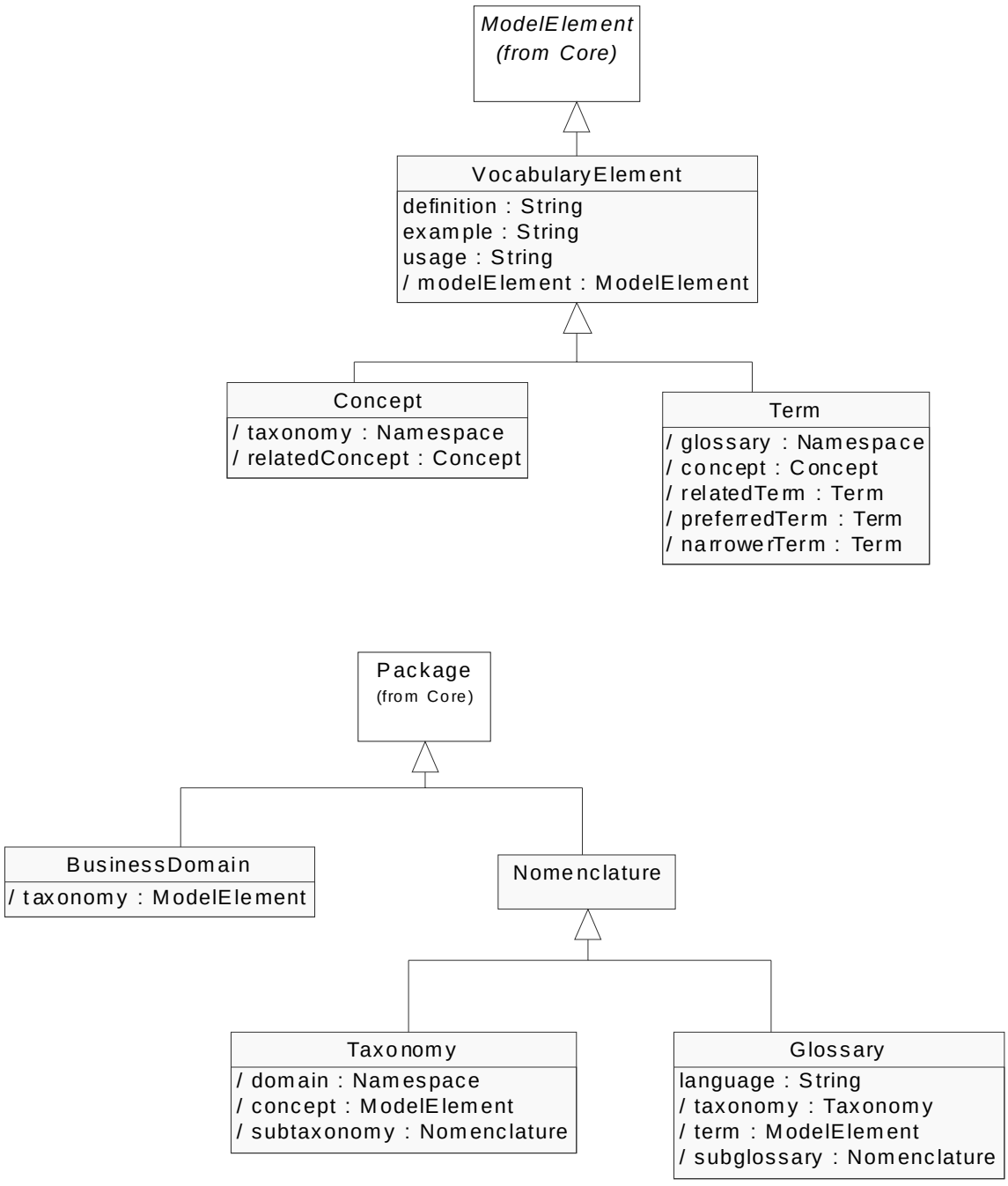


*Figure 17-2* BusinessNomenclature Package: Hierarchy

## 17.3 Business Nomenclature Classes

The BusinessNomenclature package contains the following classes, in alphabetical order:

- BusinessDomain
- Concept
- Glossary
- Nomenclature
- Taxonomy
- Term
- VocabularyElement

### 17.3.1 BusinessDomain

This represents a business domain.

#### Superclasses

Package

#### Contained Elements

Taxonomy

### References

***taxonomy***

Identifies the Taxonomies owned by the BusinessDomain.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | Namespace-ModelElement::ownedElement |
| *multiplicity:* | zero or more |
| *inverse:* | Taxonomy::domain |

## 17.3.2 Concept

This represents a business idea or notion.

Concepts are represented by Terms. Users use Terms that are familiar to them in their business environment to refer to Concepts.

### Superclasses

VocabularyElement

### References

#### *taxonomy*

Identifies the Taxonomy that owns the Concept.

| | |
|---|---|
| *class:* | Namespace |
| *defined by:* | Namespace-ModelElement::namespace |
| *multiplicity:* | zero or one |
| *inverse:* | Taxonomy::concept |

#### *relatedConcept*

Identifies the related Concepts.

| | |
|---|---|
| *class:* | Concept |
| *defined by:* | RelatedConcepts::relatedConcept |
| *multiplicity:* | zero or more |

### Constraints

A Concept may not relate to itself. [C-1]

## 17.3.3 Glossary

This represents a collection of Terms.

### Superclasses

Nomenclature

### Contained Elements

Term

### Attributes

#### *language*

Identifies the language that the Glossary is represented in.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### References

***taxonomy***

Identifies the Taxonomies that the Glossary is derived from.

| | |
|---|---|
| *class:* | Taxonomy |
| *defined by:* | GlossaryToTaxonomy::taxonomy |
| *multiplicity:* | zero or more |

***term***

Identifies the Terms that are owned by the Glossary.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | Namespace-ModelElement::ownedElement |
| *multiplicity:* | zero or more |
| *inverse:* | Term::glossary |

***subglossary***

Identifies the child Glossaries.

| | |
|---|---|
| *class:* | Nomenclature |
| *defined by:* | NomenclatureHierarchy::child |
| *multiplicity:* | zero or more |

### Constraints

The parent [C-2] or child [C-3] of a Glossary must be a Glossary.

## 17.3.4 Nomenclature

This represents a common superclass for Taxonomy and Glossary.

### Superclasses

Package

### Contained Elements

Nomenclature, VocabularyElement

### *Constraints*

A Nomenclature may not be its own parent or child, transitive closure.

## *17.3.5 Taxonomy*

This represents a collection of Concepts that form an ontology.

### *Superclasses*

Nomenclature

### *Contained Elements*

Concept

### References

***domain***

Identifies the BusinessDomain that owns the Taxonomy.

| | |
|---|---|
| *class:* | Namespace |
| *defined by:* | Namespace-ModelElement::namespace |
| *multiplicity:* | zero or one |
| *inverse:* | BusinessDomain::taxonomy |

***concept***

Identifies the Concepts that are owned by the Taxonomy.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | Namespace-ModelElement::ownedElement |
| *multiplicity:* | zero or more |
| *inverse:* | Concept::taxonomy |

***subtaxonomy***

Identifies the child Taxonomies.

| | |
|---|---|
| *class:* | Nomenclature |
| *defined by:* | NomenclatureHierarchy::child |
| *multiplicity:* | zero or more |

### Constraints

The parent [C-4] or child [C-5] of a Taxonomy must be a Taxonomy.

## 17.3.6 Term

This represents words or phrases used by business users to refer to Concepts.

A Term has a definition in a specific context. The context is provided by the referenced Concept that describes the underlying semantics.

### Superclasses

VocabularyElement

## References

### *glossary*

Identifies the Glossary that owns the Term.

| | |
|---|---|
| *class:* | Namespace |
| *defined by:* | Namespace-ModelElement::namespace |
| *multiplicity:* | zero or one |
| *inverse:* | Clossary::term |

### *concept*

Identifies the Concepts from which the Term is derived.

| | |
|---|---|
| *class:* | Concept |
| *defined by:* | TermToConcept::concept |
| *multiplicity:* | zero or more |

### *relatedTerm*

Identifies the related Terms.

| | |
|---|---|
| *class:* | Term |
| *defined by:* | RelatedTerms::relatedTerm |
| *multiplicity:* | zero or more |

### *preferredTerm*

Identifies the preferred Term.

| | |
|---|---|
| *class:* | Term |
| *defined by:* | SynonymToPreferredTerm::preferredTerm |
| *multiplicity:* | zero or one |

### *narrowerTerm*

Identifies the narrower Terms.

| | |
|---|---|
| *class:* | Term |
| *defined by:* | WiderToNarrowerTerm::narrowerTerm |
| *multiplicity:* | zero or more |

### Constraints

A Term may not relate to itself. [C-6]

A Term may not be its own preferred term or synonym, transitive closure.

A Term may not be its own narrower term or wider term, transitive closure.

## 17.3.7 VocabularyElement

This represents a common superclass for Concepts and Terms.

### Superclasses

ModelElement

### Attributes

***definition***

Provides the definition of the VocabularyElement.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

***example***

Provides examples of the VocabularyElement.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

***usage***

Identifies typical usage of the VocabularyElement.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### References

***modelElement***

Identifies the ModelElement (the physical metadata) that represents this VocabularyElement (the business metadata).

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | VocabularyElementToModelElement::modelElement |
| *multiplicity:* | zero or more |

### Constraints

A VocabularyElement may not relate to itself. [C-7]

## 17.4 Business Nomenclature Associations

The BusinessNomenclature package contains the following associations, in alphabetical order:

- GlossaryToTaxonomy
- NomenclatureHierarchy
- RelatedConcepts
- RelatedTerms
- RelatedVocabularyElements
- SynonymToPreferredTerm
- TermToConcept
- VocabularyElementToModelElement
- WiderToNarrowerTerm

### 17.4.1 GlossaryToTaxonomy

This association relates a Glossary to its Taxonomies.

*Ends*

***glossary***

Identifies a Glossary.

| | |
|---|---|
| *class:* | Glossary |
| *multiplicity:* | zero or more |

***taxonomy***

Identifies the Taxonomies from which the Glossary is derived.

| | |
|---|---|
| *class:* | Taxonomy |
| *multiplicity:* | zero or more |

### *17.4.2 NomenclatureHierarchy*

This aggregation relates a parent Nomenclature to its child Nomenclatures.

*Ends*

***parent***

Identifies the parent Nomenclature.

| | |
|---|---|
| *class:* | Nomenclature |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

***child***

Identifies the child Nomenclatures.

| | |
|---|---|
| *class:* | Nomenclature |
| *multiplicity:* | zero or more |

### *17.4.3 RelatedConcepts* *derived*

This association relates a Concept to its related Concepts.

### *Ends*

***concept***

Identifies a Concept.

| | |
|---|---|
| *class:* | Concept |
| *multiplicity:* | zero or more |

***relatedConcept***

Identifies the related Concepts.

| | |
|---|---|
| *class:* | Concept |
| *multiplicity:* | zero or more |

### *Derivation*

This association is derived from the RelatedVocabularyElements association. All ends of the association must be Concepts. [C-8]

## *17.4.4 RelatedTerms* *derived*

This association relates a Term to its related Terms.

### Ends

***term***

Identifies a Term.

| | |
|---|---|
| *class:* | Term |
| *multiplicity:* | zero or more |

***relatedTerm***

Identifies the related Terms.

| | |
|---|---|
| *class:* | Term |
| *multiplicity:* | zero or more |

### Derivation

This association is derived from the RelatedVocabularyElements association. All ends of the association must be Terms.[C-9]

## 17.4.5 RelatedVocabularyElements

This association relates a VocabularyElement to its related VocabularyElements.

*Ends*

***element***

Identifies a VocabularyElement.

| | |
|---|---|
| *class:* | VocabularyElement |
| *multiplicity:* | zero or more |

***relatedElement***

Identifies the related VocabularyElements.

| | |
|---|---|
| *class:* | VocabularyElement |
| *multiplicity:* | zero or more |

### *17.4.6 SynonymToPreferredTerm*

This association relates a synonym to its preferred terms.

*Ends*

***synonym***

Identifies a Term.

| | |
|---|---|
| *class:* | Term |
| *multiplicity:* | zero or more |

***preferredTerm***

Identifies the preferred term for the synonym.

| | |
|---|---|
| *class:* | Term |
| *multiplicity:* | zero or one |

### *17.4.7 TermToConcept*

This association relates a Term to its Concepts.

*Ends*

***term***

Identifies a Term.

| | |
|---|---|
| *class:* | Term |
| *multiplicity:* | zero or more |

***concept***

Identifies the Concepts from which the Term is derived.

| | |
|---|---|
| *class:* | Concept |
| *multiplicity:* | zero or more |

### *17.4.8 VocabularyElementToModelElement*

This association relates a VocabularyElement to the ModelElements for which the VocabularyElement provides business meaning.

*Ends*

***vocabularyElement***

Identifies a VocabularyElement.

| | |
|---|---|
| *class:* | VocabularyElement |
| *multiplicity:* | zero or more |

***modelElement***

Identifies the ModelElements for which the VocabularyElement provides business meaning.

| | |
|---|---|
| *class:* | ModelElement |
| *multiplicity:* | zero or more |

## *17.4.9 WiderToNarrowerTerm*

This association relates a wider term to its narrower terms.

*Ends*

***widerTerm***

Identifies a Term.

| | |
|---|---|
| *class:* | Term |
| *multiplicity:* | zero or more |

***narrowerTerm***

Identifies the narrower terms for the wider term.

| | |
|---|---|
| *class:* | Term |
| *multiplicity:* | zero or more |

## *17.5 OCL Representation of Business Nomenclature Constraints*

[C-1] A Concept may not relate to itself.

**context** Concept

**inv**: self.relatedConcept->forAll ( p | p <> self )

[C-2] The parent of a Glossary must be a Glossary.

**context** Glossary

**inv**: self.parent.oclIsKindOf( Glossary )

[C-3] The child of a Glossary must be a Glossary.

**context** Glossary

**inv**: self.child->forAll( p | p.oclIsKindOf( Glossary ) )

[C-4] The parent of a Taxonomy must be a Taxonomy.

**context** Taxonomy

**inv**: self.parent.oclIsKindOf( Taxonomy )

[C-5] The child of a Taxonomy must be a Taxonomy.

**context** Taxonomy

**inv**: self.child->forAll( p | p.oclIsKindOf( Taxonomy ) )

[C-6] A Term may not relate to itself.

**context** Term

**inv**: self.relatedTerm->forAll ( p | p <> self )

[C-7] A VocabularyElement may not relate to itself.

**context** Vocabulary

**inv**: self.relatedElement->forAll ( p | p <> self )

[C-8] The RelatedConcepts association is derived from the RelatedVocabularyElements association. All ends of the RelatedConcepts association must be Concepts.

**context** RelatedConcepts

**inv**: RelatedVocabularyElements.allInstances.select( element.oclIsKindOf( Concept ) and relatedElement.oclIsKindOf( Concept ) )

[C-9] The RelatedTerms association is derived from the RelatedVocabularyElements association. All ends of the RelatedTerms association must beTerms.

**context** RelatedTerms

**inv**: RelatedVocabularyElements.allInstances.select( element.oclIsKindOf( Term ) and relatedElement.oclIsKindOf( Term ) )

# Warehouse Process 18

## 18.1 Overview

The Warehouse Process package documents the process flows used to execute transformations. These process flows may be documented at the level of a complete TransformationActivity or its individual TransformationSteps. A WarehouseProcess object associates a transformation with a set of events which will be used to trigger the execution of the transformation.

## 18.2 Organization of the Warehouse Process Package

The Warehouse Process package depends on the following packages:

- org.omg::CWM::ObjectModel::Core
- org.omg::CWM::ObjectModel::Behavioral
- org.omg::CWM::Analysis::Transformation

A WarehouseProcess object represents the processing of a transformation. It is instantiated as one of its subtypes WarehouseActivity or WarehouseStep, depending on whether it represents the processing of a TransformationActivity or a Transformation Step.

A WarehouseProcess may be associated with one or more WarehouseEvents, each identifying events that cause the processing to be initiated. It may also be associated with one or more internal events that will be triggered when processing terminates.

A ProcessPackage may be used to group together related WarehouseActivities.

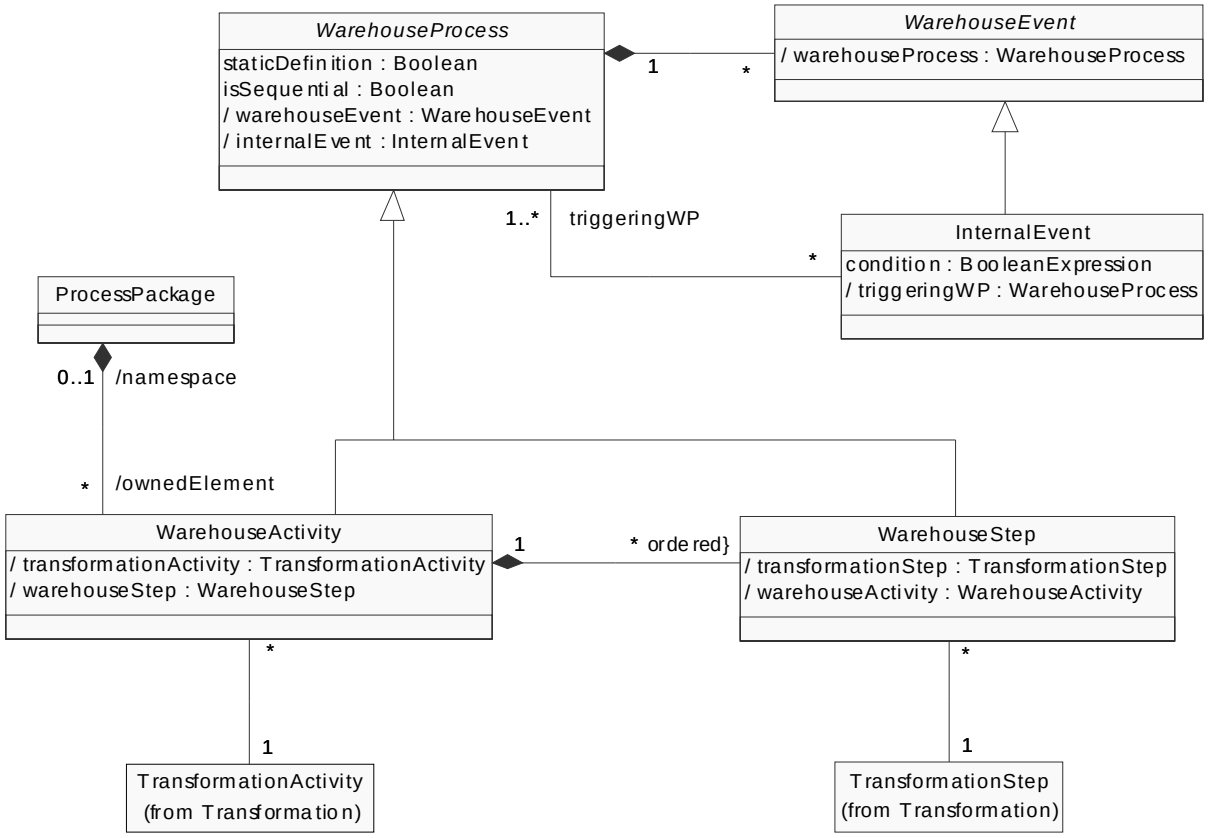


*Figure 18-1* Warehouse Process package overview.

WarehouseEvents are divided into three categories: scheduled, external and internal.

Scheduled events can either be defined as a point in time (each Wednesday at 2 pm) or be defined by intervals (every five minutes). A point in time event can be defined as a custom calendar which contains a set of calendar dates. This allows a series of dates to be reused across several WarehouseProcesses.

External events are triggered by something happening outside the data warehouse, for example by a batch process which is not described as a WarehouseProcess.

Internal events are triggered by the termination of a WarehouseProcess. They can be either retry events or cascade events. Retry events normally trigger a rerun of the same WarehouseProcess, whereas cascade events normally trigger a different WarehouseProcess. An internal event may define a condition that determines whether or not the event is triggered. This condition can use details of the execution of the triggering WarehouseProcess recorded in the relevant ActivityExecution and StepExecution objects.

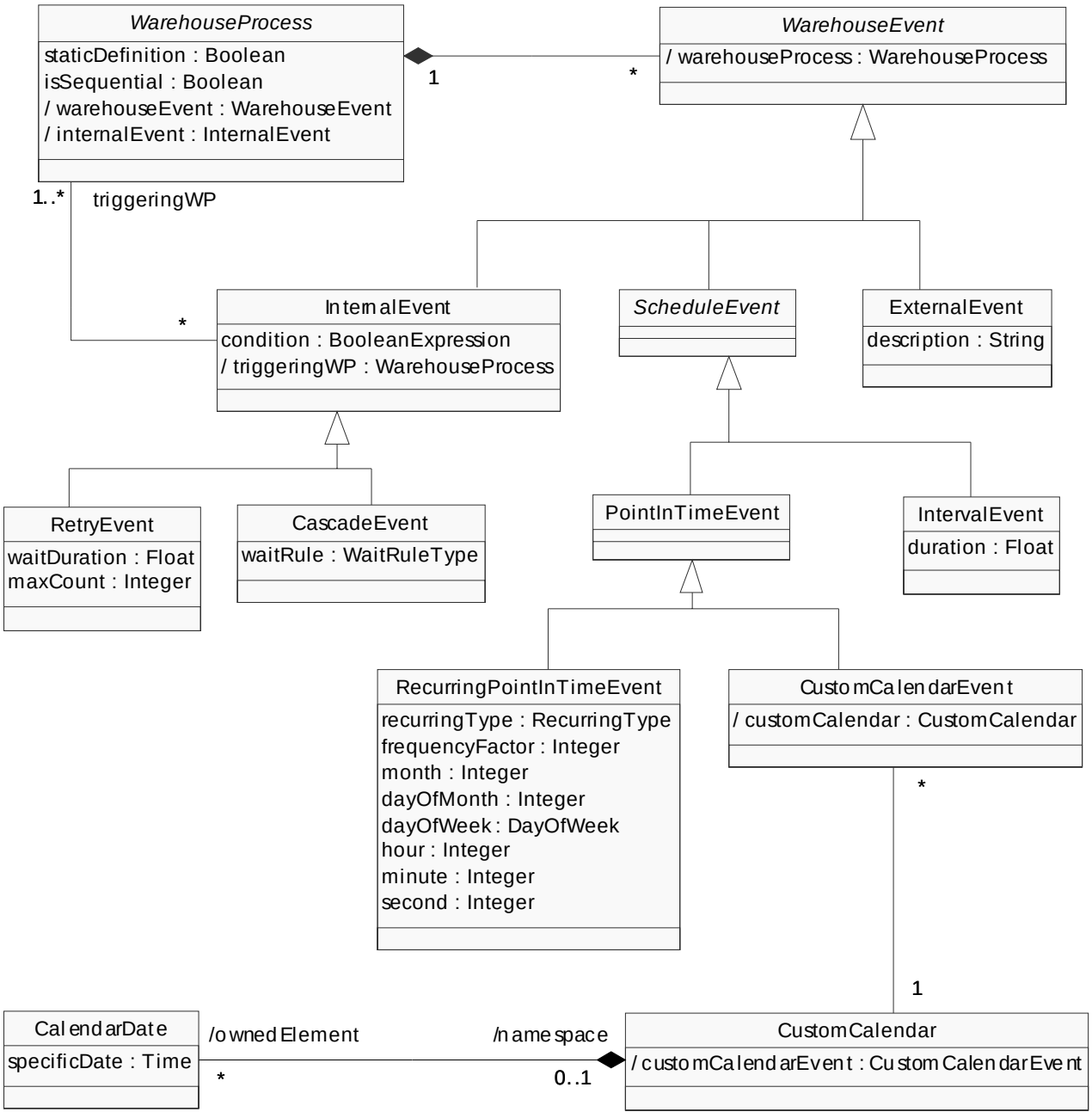


*Figure 18-2* Warehouse Events and Custom Calendars.

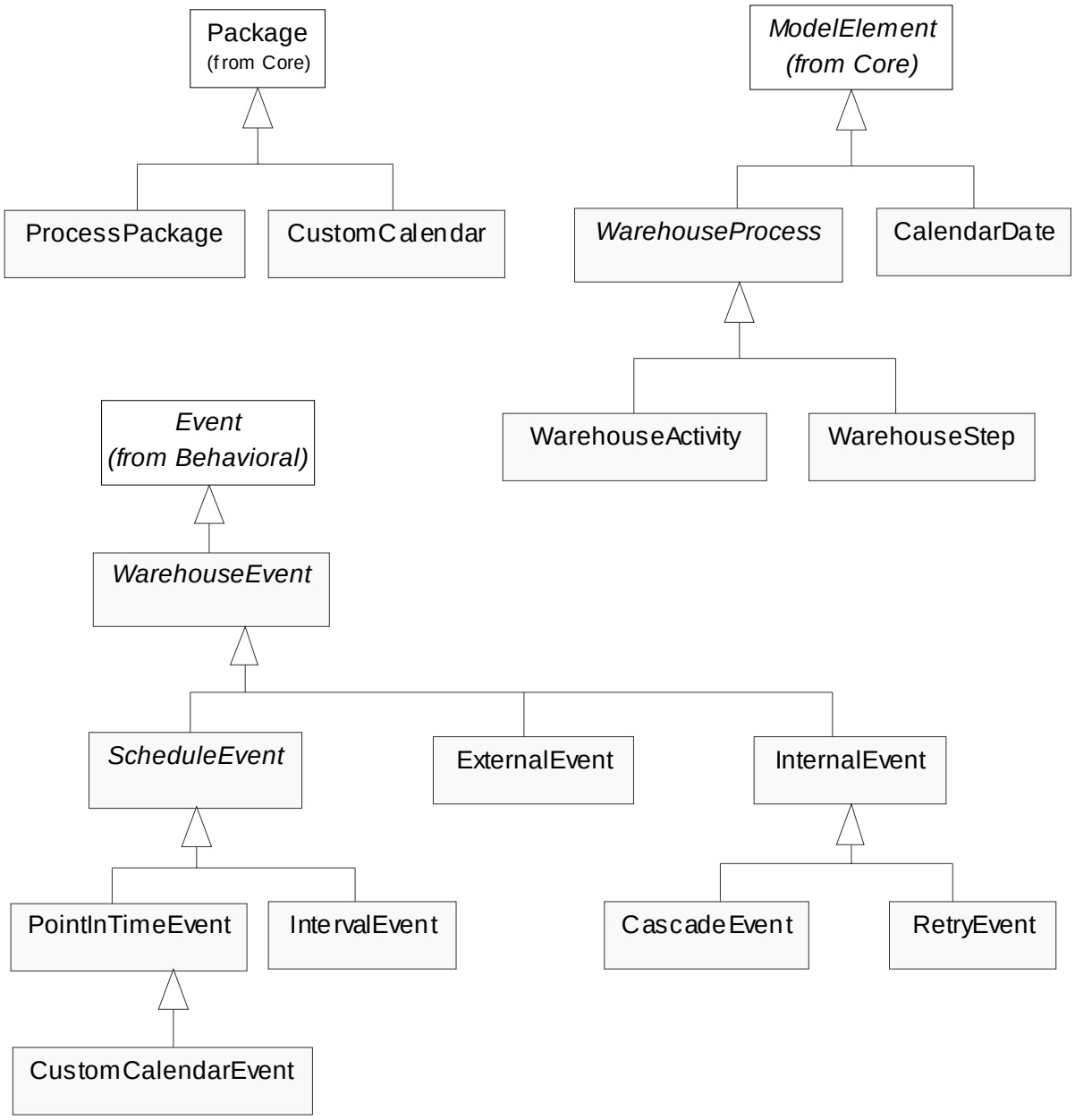


*Figure 18-3* Warehouse Process package inheritance structure.

The instance diagram below shows how the scheduled (every Wednesday at 2 pm) unload process cascades with the load process:

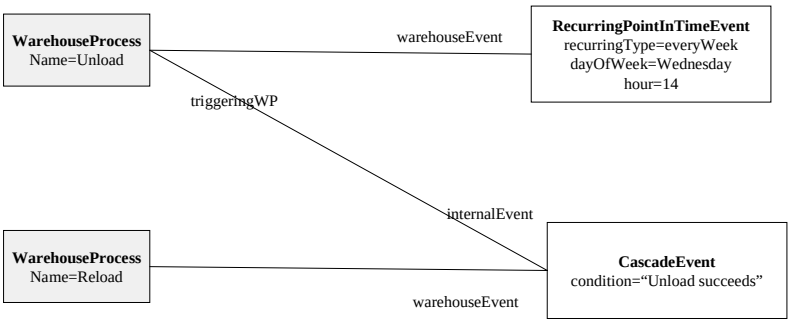


*Figure 18-4* Instance diagram of cascade event.

## 18.3 Warehouse Process Classes

The Warehouse Process package contains the following classes, in alphabetical order:

- CalendarDate
- CascadeEvent
- CustomCalendar
- CustomCalendarEvent
- ExternalEvent
- InternalEvent
- IntervalEvent
- PointInTimeEvent
- ProcessPackage
- RecurringPointInTimeEvent
- RetryEvent
- ScheduleEvent
- WarehouseActivity
- WarehouseEvent
- WarehouseProcess
- WarehouseStep

### 18.3.1 CalendarDate

An entry in a CustomCalendar representing a specific date and time.

#### *Superclasses*

ModelElement

### Attributes

***specificDate***

The value of the date.

| | |
|---|---|
| type: | Time (i.e. a date and time) |
| multiplicity: | exactly one |

## 18.3.2 CascadeEvent

A CascadeEvent indicates that completion of one or more triggering WarehouseProcesses triggers another WarehouseProcess.

### Superclasses

InternalEvent

### Attributes

***waitRule***

Indicates if the event should be triggered as soon as any of the triggering WarehouseProcesses has completed that satisfies the condition (inherited from InternalEvent) or only when all the triggering WarehouseProcesses have completed (provided the condition is satisfied).

| | |
|---|---|
| type: | WaitRuleType ( waitForAll \| waitForAny ) |
| multiplicity: | exactly one |

## 18.3.3 CustomCalendar

A named list of dates and times.

### Superclasses

Package

### Contained Elements

CalendarDate

### References

***customCalendarEvent***

Indicates which events use this custom calendar.

| | |
|---|---|
| class: | CustomCalendarEvent |
| defined by: | EventUsesCustomCalendar::customCalendarEvent |
| multiplicity: | zero or more |
| inverse: | CustomCalendarEvent::customCalendar |

## 18.3.4 CustomCalendarEvent

This event is controlled by a list of dates and times. To make the list easily shareable between multiple WarehouseProcesses the calendar itself is in a separate class.

### Superclasses

PointInTimeEvent

### References

***customCalendar***

Indicates which custom calendar is used for this schedule.

| | |
|---|---|
| class: | CustomCalendar |
| defined by: | EventUsesCustomCalendar::customCalendar |
| multiplicity: | exactly one |
| inverse: | CustomCalendar::customCalendarEvent |

## 18.3.5 ExternalEvent

An ExternalEvent allows the description of the triggering of a WarehouseProcess by a task which is not described in the model. This is merely a place holder. The actual behavior and the connection with the external trigger is left to the implementation of the scheduler.

### Superclasses

WarehouseEvent

### Attributes

***description***

A free text description of where the external triggering signal comes from.

| | |
|---|---|
| type: | String |
| multiplicity: | exactly one |

## 18.3.6 InternalEvent

An event which may be triggered, depending on whether or not a condition is satisfied, by the conclusion of one or more WarehouseProcess runs.

There are two types of InternalEvents, depending whether the event triggers a series of different WarehouseProcesses, or whether the event triggers the same WarehouseProcess in an attempt to retry a failed run.

### Superclasses

WarehouseEvent

### Attributes

***condition***

Indicates what condition the triggering WarehouseProcess run must meet to be considered (success, failure, warnings, etc.).
How the condition is expressed, and how the result of a Transform is generated is left to the implementation of the scheduler and the transformation, respectively.

| | |
|---|---|
| type: | BooleanExpression |
| multiplicity: | exactly one |

### References

***triggeringWP***

Associates an internal event with the triggering WarehouseProcess.

| | |
|---|---|
| class: | WarehouseProcess |
| defined by: | TriggeringProcess::triggeringWP |
| multiplicity: | one or more |
| inverse: | WarehouseProcess::internalEvent |

## 18.3.7 IntervalEvent

An IntervalEvent controls a continuous run of a WarehouseProcess. The Warehouse Process will run, then wait for the duration specified in the event, then run again.

An IntervalEvent is not affected by the result of the WarehouseProcess.

### Superclasses

ScheduleEvent

### Attributes

***duration***

Indicates the length of time (in seconds) to wait after a run of the WarehouseProcess before triggering the next one.

| | |
|---|---|
| type: | Float |
| multiplicity: | exactly one |

## 18.3.8 PointInTimeEvent

A PointInTime event is triggered at a fixed time, independently of any external context.

The triggering time can be either defined functionally (as in the RecurringPointInTimeEvent extension of this class), or by an explicit list of times (CustomCalendarEvent).

### Superclasses

ScheduleEvent

### 18.3.9 ProcessPackage

A group of related WarehouseActivities.

#### Superclasses

Package

#### Contained Elements

WarehouseActivity

### 18.3.10 RecurringPointInTimeEvent

This event triggers a WarehouseProcess on a regular basis such as a specific date or time (for example, the Wednesday of every other week, at 2:30 pm).

#### Superclasses

PointInTimeEvent

#### Attributes

***recurringType***

Indicates how often the event should be triggered (weekly, daily, etc.).

| | |
|---|---|
| type: | RecurringType ( everyYear \| everyMonth \| everyWeek \| everyDay \| everyHour \| everyMinute ) |
| multiplicity: | exactly one |

***frequencyFactor***

Indicates the repetition of the event. For example, for a weekly recurringType, a value of 1 will mean that it is to be triggered every week, a value of 2 will mean that it is to be triggered every other week, etc.

| | |
|---|---|
| type: | Integer |
| multiplicity: | exactly one |

***month***

Indicates which month of the year (from 1 to 12) an annual event is to be triggered.

type: Integer

multiplicity: zero or one

***dayOfMonth***

Indicates which day of the month (from 1 to 31) a monthly or annual event is to be triggered. For a monthly event, if the day of the month is greater than the number of days in the month, it is assumed that the scheduler will run the WarehouseProcess on the last day of the month.

type: Integer

multiplicity: zero or one

***dayOfWeek***

Indicates which day of the week a weekly schedule is running.

type: DayOfWeek ( monday | tuesday | wednesday | thursday | friday | saturday | sunday | workingDay | nonworkingDay )

multiplicity: zero or one

***hour***

Indicates at what hour (from 0 to 23) an annual, monthly, weekly, or daily event is being triggered.

type: Integer

multiplicity: zero or one

***minute***

Indicates at what minute (from 0 to 59) an event is triggered. Applies to all events except the "everyMinute" ones.

type: Integer

multiplicity: zero or one

***second***

Indicates at what second (from 0 to 59) an event must be run. Applies to all events.

| | |
|---|---|
| type: | Integer |
| multiplicity: | exactly one |

### *Constraints*

***month*** must be specified when ***recurringType*** is everyYear. [C-1]

***month*** must be between 1 and 12 (inclusive) when specified. [C-2]

***dayOfMonth*** must be specified when ***recurringType*** is everyYear or everyMonth. [C-3]

***dayOfMonth*** must be between 1 and 31 (inclusive) when specified. [C-4]

***dayOfWeek*** must be specified when ***recurringType*** is everyWeek. [C-5]

***hour*** must be specified when ***recurringType*** is everyYear or everyMonth or everyWeek or everyDay. [C-6]

***hour*** must be between 0 and 23 (inclusive) when specified. [C-7]

***minute*** must be specified when ***recurringType*** is not everyMinute. [C-8]

***minute*** must be between 0 and 59 (inclusive) when specified. [C-9]

***second*** must be between 0 and 59 (inclusive). [C-10]

## *18.3.11 RetryEvent*

Indicates that a WarehouseProcess should be retried upon failure. This type of event is used for example when a WarehouseProcess relies on sources with uncertain availability (connection or uptime).

In general, the triggering WarehouseProcess and the triggered WarehouseProcess are the same, and only one WarehouseProcess is involved. But this is not an imposed limitation. It is left to the schedulers to decide on the implementation behavior for complex cases.

### *Superclasses*

InternalEvent

### Attributes

***waitDuration***

Indicates the length of time (in seconds) to wait before retrying the triggered WarehouseProcess.

| | |
|---|---|
| type: | Float |
| multiplicity: | exactly one |

***maxCount***

Indicates how many times the triggered WarehouseProcess should be retried before being declared failed.

| | |
|---|---|
| type: | Integer |
| multiplicity: | exactly one |

## 18.3.12 ScheduleEvent *abstract*

A ScheduleEvent is an abstract class which covers all the clock based events.

### Superclasses

WarehouseEvent

## 18.3.13 WarehouseActivity

A WarehouseActivity is a subtype of WarehouseProcess that represents the processing of a TransformationActivity. It may identify WarehouseEvents that trigger the processing of the TransformationActivity and InternalEvents that are triggered by the conclusion of this processing. It may contain a set of WarehouseSteps that define in more detail the processing of the individual TransformationSteps of the TransformationActivity.

### Superclasses

WarehouseProcess

### Contained Elements

WarehouseEvent
WarehouseStep

### References

#### *transformationActivity*

Associates a WarehouseActivity with the TransformationActivity it performs.

| | |
|---|---|
| class: | TransformationActivity |
| defined by: | WarehouseActivityRunsTransformationActivity ::transformationActivity |
| multiplicity: | exactly one |

#### *warehouseStep*

Identifies WarehouseSteps that are components of the WarehouseActivity.

| | |
|---|---|
| class: | WarehouseStep |
| defined by: | WarehouseActivityStep::warehouseStep |
| multiplicity: | zero or more; ordered |
| inverse: | WarehouseStep::warehouseActivity |

## 18.3.14 WarehouseEvent *abstract*

A virtual class to refer to any Event.

A WarehouseEvent (or its derivations) represents what triggers the running of a WarehouseProcess. An event can be initiated by a clock, by an external trigger, or by an internal trigger (the conclusion of some WarehouseProcess).

### Superclasses

Event

### References

#### *warehouseProcess*

Identifies the WarehouseProcess that is triggered by the WarehouseEvent.

| | |
|---|---|
| class: | WarehouseProcess |
| defined by: | Event::warehouseProcess |
| multiplicity: | exactly one |
| inverse: | WarehouseProcess::warehouseEvent |

### 18.3.15 WarehouseProcess *abstract*

A WarehouseProcess represents the processing of a transformation. It is instantiated as one of its subtypes WarehouseActivity or WarehouseStep, depending on whether it represents the processing of a TransformationActivity or a Transformation Step.

A WarehouseProcess may be associated with one or more WarehouseEvents, each identifying events that cause the processing to be initiated. It may also be associated with one or more internal events that will be triggered when processing terminates.

#### *Superclasses*

ModelElement

#### *Attributes*

***staticDefinition***

When a WarehouseProcess is a constant mapping (such as a Relational View of legacy data or a continuous data propagation process), this flag indicates that the mapping does not require to be run for the target to be up-to-date and in sync with the source.

| | |
|---|---|
| type: | Boolean |
| multiplicity: | exactly one |

***isSequential***

This flag indicates if more than one instance of this WarehouseProcess may run at a time. If this flag is true, the scheduler should fail any attempt to trigger this WarehouseProcess while an instance is already in progress.

| | |
|---|---|
| type: | Boolean |
| multiplicity: | exactly one |

### *References*

***warehouseEvent***

Associates a WarehouseProcess with a set of events of various types, which will be used to trigger the execution of the WarehouseProcess and its associated transformation.

| | |
|---|---|
| class: | WarehouseEvent |
| defined by: | Event::warehouseEvent |
| multiplicity: | zero or more |
| inverse: | WarehouseEvent::warehouseProcess |

***internalEvent***

Associates a WarehouseProcess with the internal events it may trigger.

| | |
|---|---|
| class: | InternalEvent |
| defined by: | TriggeringProcess::internalEvent |
| multiplicity: | zero or more |
| inverse: | InternalEvent::triggeringWP |

## *18.3.16 WarehouseStep*

A WarehouseStep is a component of a WarehouseActivity. It represents the processing of an individual TransformationStep. It may be used to identify WarehouseEvents that trigger the processing of the TransformationStep and/or InternalEvents that are triggered by the conclusion of the processing of the TransformationStep.

For example, a WarehouseStep may be used to document how a specific TransformationStep should be retried upon failure.

### *Superclasses*

WarehouseProcess

### *Contained Elements*

WarehouseEvent

### *References*

***transformationStep***

Associates a WarehouseStep with the TransformationStep it performs.

| | |
|---|---|
| class: | TransformationStep |
| defined by: | WarehouseStepRunsTransformationStep ::transformationStep |
| multiplicity: | exactly one |

***warehouseActivity***

Identifies the WarehouseActivity which includes this WarehouseStep.

| | |
|---|---|
| class: | WarehouseActivity |
| defined by: | WarehouseActivityStep::warehouseActivity |
| multiplicity: | exactly one |
| inverse: | WarehouseActivity::warehouseStep |

## *18.4 Warehouse Process Associations*

The Warehouse Process package contains the following associations, in alphabetical order:

- Event
- EventUsesCustomCalendar
- TriggeringProcess
- WarehouseActivityRunsTransformationActivity
- WarehouseActivityStep
- WarehouseStepRunsTransformationStep

### 18.4.1 Event *protected*

Associates a WarehouseProcess with a set of events of various types, which will be used to trigger the execution of the WarehouseProcess and its associated transformation.

*Ends*

***warehouseProcess***

Identifies the WarehouseProcess which will be triggered by the event.

| | |
|---|---|
| class: | WarehouseProcess |
| multiplicity: | exactly one |
| aggregation: | composite |

***warehouseEvent***

Identifies a set of events of various types, which will be used to trigger the execution of the WarehouseProcess and its associated transformations.

| | |
|---|---|
| class: | WarehouseEvent |
| multiplicity: | zero or more |

### 18.4.2 EventUsesCustomCalendar *protected*

Indicates which custom calendar is used for this schedule.

*Ends*

***customCalendar***

Indicates which custom calendar is used for this event.

| | |
|---|---|
| class: | CustomCalendar |
| multiplicity: | exactly one |

***customCalendarEvent***

Indicates which event uses this custom calendar.

| | |
|---|---|
| class: | CustomCalendarEvent |
| multiplicity: | zero or more |

### 18.4.3 TriggeringProcess *protected*

Associates an internal event with the WarehouseProcess that triggers it when processing of that WarehouseProcess terminates.

*Ends*

***triggeringWP***

Identifies the triggering WarehouseProcess.

| | |
|---|---|
| class: | WarehouseProcess |
| multiplicity: | one or more |

***internalEvent***

Identifies an internal event triggered by the termination of the WarehouseProcess.

| | |
|---|---|
| class: | InternalEvent |
| multiplicity: | zero or more |

### 18.4.4 WarehouseActivityRunsTransformationActivity

Indicates which TransformationActivity is run by the WarehouseActivity.

*Ends*

***transformationActivity***

Associates a WarehouseActivity with the TransformationActivity it performs.

| | |
|---|---|
| class: | TransformationActivity |
| multiplicity: | exactly one |

***warehouseActivity***

Identifies WarehouseActivities that perform a TransformationActivity.

| | |
|---|---|
| class: | WarehouseActivity |
| multiplicity: | zero or more |

### 18.4.5 WarehouseActivityStep *protected*

Associates a WarehouseActivity with its constituent WarehouseSteps.

*Ends*

***warehouseActivity***

Identifies the WarehouseActivity of which a WarehouseStep is a component.

| | |
|---|---|
| class: | WarehouseActivity |
| multiplicity: | exactly one |
| aggregation: | composite |

***warehouseStep***

Identifies a WarehouseStep that is a component of the WarehouseActivity.

| | |
|---|---|
| class: | WarehouseStep |
| multiplicity: | zero or more; ordered |

### 18.4.6 WarehouseStepRunsTransformationStep

Identifies a TransformationStep that is run by a WarehouseStep.

*Ends*

***transformationStep***

Associates a WarehouseStep with the TransformationStep it performs.

| | |
|---|---|
| class: | TransformationStep |
| multiplicity: | exactly one |

***warehouseStep***

Identifies WarehouseSteps that perform a TransformationStep.

| | |
|---|---|
| class: | WarehouseStep |
| multiplicity: | zero or more |

## 18.5 OCL Representation of Warehouse Process Constraints

[C-1] ***month*** must be specified when ***recurringType*** is everyYear.

**context** RecurringPointInTimeEvent **inv:**

self.recurringType=everyYear **implies** self.month->notEmpty

[C-2] ***month*** must be between 1 and 12 (inclusive) when specified.

**context** RecurringPointInTimeEvent **inv:**

self.month->notEmpty **implies** 1 <= self.month <= 12

[C-3] ***dayOfMonth*** must be specified when ***recurringType*** is everyYear or everyMonth.

**context** RecurringPointInTimeEvent **inv:**

self.recurringType=everyYear or self.recurringType=everyMonth
**implies** self.dayOfMonth->notEmpty

[C-4] ***dayOfMonth*** must be between 1 and 31 (inclusive) when specified.

**context** RecurringPointInTimeEvent **inv**:

self.dayOfMonth->notEmpty **implies** 1 <= self.dayOfMonth <= 31

[C-5] ***dayOfWeek*** must be specified when ***recurringType*** is everyWeek.

**context** RecurringPointInTimeEvent **inv:**

self.recurringType=everyWeek **implies** self.dayOfWeek->notEmpty

[C-6] ***hour*** must be specified when ***recurringType*** is everyYear or everyMonth or everyWeek or everyDay.

**context** RecurringPointInTimeEvent **inv:**

self.recurringType=everyYear or self.recurringType=everyMonth or
self.recurringType=everyWeek or self.recurringType=everyDay
**implies** self.hour->notEmpty

[C-7] ***hour*** must be between 0 and 23 (inclusive) when specified.

**context** RecurringPointInTimeEvent **inv:**

self.hour->notEmpty **implies** 0 <= hour <= 23

[C-8] ***minute*** must be specified when ***recurringType*** is not everyMinute.

**context** RecurringPointInTimeEvent **inv:**

self.recurringType<>everyMinute **implies** self.minute->notEmpty

[C-9] ***minute*** must be between 0 and 59 (inclusive) when specified.

**context** RecurringPointInTimeEvent **inv**:

self.minute->notEmpty **implies** 0 <= self.minute <= 59

[C-10] ***second*** must be between 0 and 59 (inclusive).

**context** RecurringPointInTimeEvent **inv**:

0 <= self.second <= 59

# Warehouse Operation 19

## 19.1 Overview

The Warehouse Operation package contains classes recording the day-to-day operation of the warehouse processes.

The package covers three separate areas:

- Transformation Executions
- Measurements
- Change Requests

### 19.1.1 Transformation Executions

Details of the most recent executions of transformations can be recorded, identifying when they ran and whether they completed successfully. This can be used to determine how complete and up-to-date specific information in the data warehouse is.

An ActivityExecution represents an execution of a whole TransformationActivity, and a StepExecution object represents an execution of an individual TransformationStep. If a TransformationStep involves the use of an Operation , an associated StepExecution may reference a CallAction that records the actual arguments passed to the Operation.

These classes allow the lineage of data in a data warehouse to be preserved, by recording when and how it was derived, and where it came from.

### 19.1.2 Measurements

Measurement objects allow metrics to be held for any ModelElement. For example, they may be used to hold actual, estimated or planned values for the size of a table.

### 19.1.3 Change Requests

ChangeRequests allow details of proposed changes affecting any ModelElement to be recorded. They may also be used to keep a historical record of changes implemented or rejected.

## 19.2 Organization of the Warehouse Operation Package

The Warehouse Operation package depends on the following packages:

- org.omg::CWM::ObjectModel::Core
- org.omg::CWM::ObjectModel::Behavioral
- org.omg::CWM::Analysis::Transformation

Separate model diagrams are shown below for each of the three main areas supported by the package.

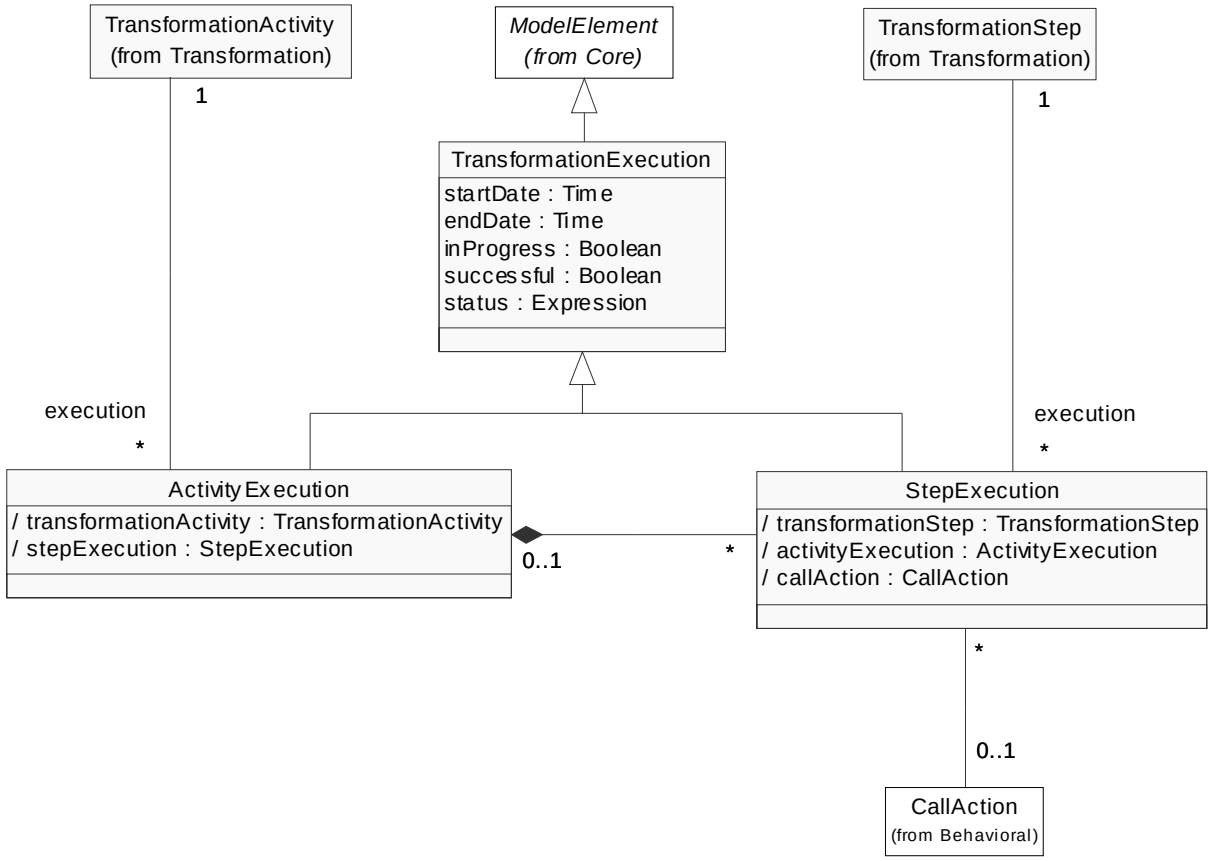


*Figure 19-1* Transformation Executions

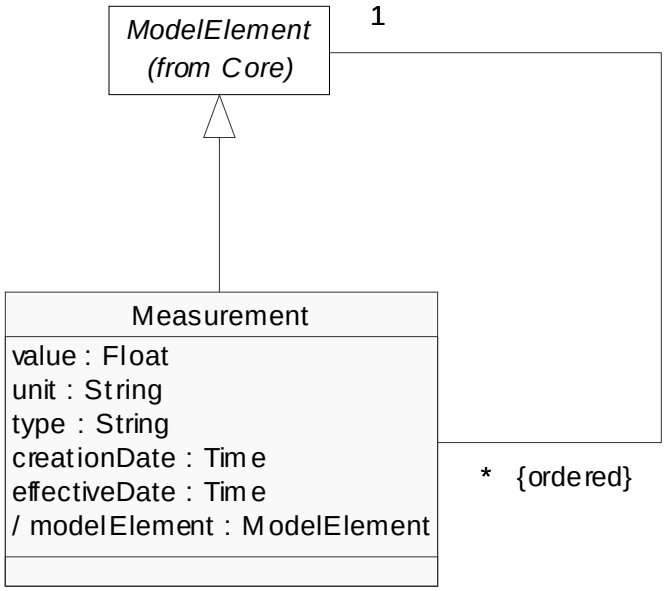


*Figure 19-2* Measurements

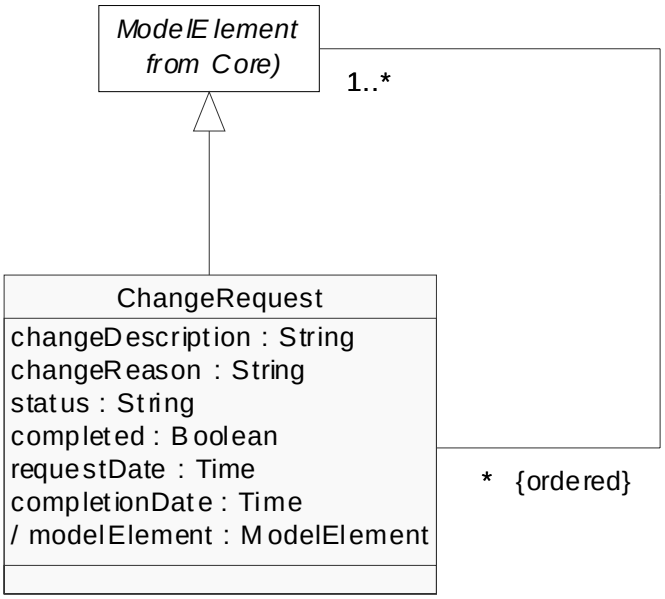


*Figure 19-3* Change Requests

## 19.3 Warehouse Operation Classes

The Warehouse Operation package contains the following classes, in alphabetical order:

- ActivityExecution
- ChangeRequest
- Measurement
- StepExecution
- TransformationExecution

### 19.3.1 ActivityExecution

An ActivityExecution is used to record details of a specific execution of a TransformationActivity.

#### Superclasses

TransformationExecution

#### Contained Elements

StepExecution

#### References

***transformationActivity***

Identifies the TransformationActivity of which this is an execution.

| | |
|---|---|
| *class:* | TransformationActivity |
| *defined by:* | TransformationActivityExecutions ::transformationActivity |
| *multiplicity:* | exactly one |

***stepExecution***

Identifies the StepExecutions that record the results of executing the individual TransformationSteps of the TransformationActivity.

| | |
|---|---|
| *class:* | StepExecution |
| *defined by:* | ActivityStepExecutions::stepExecution |
| *multiplicity:* | zero or more |
| *inverse:* | StepExecution::activityExecution |

### 19.3.2 ChangeRequest

This represents a request for change affecting one or more ModelElements. The change request may represent a proposed change or one that has been implemented or rejected.

#### Superclasses

ModelElement

#### Attributes

***changeDescription***

A description of the change.

*type:* String

*multiplicity:* exactly one

***changeReason***

The reason or justification for the ChangeRequest.

*type:* String

*multiplicity:* exactly one

***status***

The status of the ChangeRequest. This would normally contain a string such as proposed, agreed, implemented or rejected.

*type:* String

*multiplicity:* exactly one

***completed***

Indicates that no further action is required for this change request, i.e. it has either been implemented or been rejected.

*type:* Boolean

*multiplicity:* exactly one

***requestDate***

When the change request was raised.

| | |
|---|---|
| *type:* | Time |
| *multiplicity:* | exactly one |

***completionDate***

The date when all action on the change request was completed (i.e. when implementation was completed or it was rejected).

| | |
|---|---|
| *type:* | Time |
| *multiplicity:* | zero or one |

### *References*

***modelElement***

Identifies the ModelElement(s) to which the ChangeRequest applies.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | ModelElementChangeRequest::modelElement |
| *multiplicity:* | one or more |

### *Constraints*

A ChangeRequest instance must not apply to itself. [C-1]

A completionDate may only be provided for a completed ChangeRequest. [C-2]

## *19.3.3 Measurement*

A Measurement object indicates the value of some attribute of an object. It can be the number of rows in a table, the number of pages in an index, the number of different values in a column, etc.

The flexibility of this class allows for product specific extensions, without changing the model.

### *Superclasses*

ModelElement

## *Attributes*

### ***value***

The value of this Measurement.

| | |
|---|---|
| *type:* | Float |
| *multiplicity:* | exactly one |

### ***unit***

The unit of measurement.

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### ***type***

Identifies how the value was computed.
The following values have specific meanings:

| | |
|---|---|
| measure | (measured value) |
| estimate | (estimated value) |
| plan | (planned value) |
| minimum | (minimum value) |
| maximum | (maximum value) |
| average | (average value) |

| | |
|---|---|
| *type:* | String |
| *multiplicity:* | exactly one |

### ***creationDate***

When the value has been computed (see also effectiveDate).

| | |
|---|---|
| *type:* | Time |
| *multiplicity:* | exactly one |

### ***effectiveDate***

When the value is effective. For measured values, effective and creation dates should be the same. For estimated actual values, the creation date may be later than the effective date. For plan values, the effective date is normally later than the creation date.

| | |
|---|---|
| *type:* | Time |
| *multiplicity:* | exactly one |

### References

***modelElement***

Identifies the ModelElement to which the Measurement applies.

| | |
|---|---|
| *class:* | ModelElement |
| *defined by:* | ModelElementMeasurement::modelElement |
| *multiplicity:* | exactly one |

### Constraints

A Measurement instance must not apply to itself. [C-3]

## 19.3.4 StepExecution

A StepExecution is used to record details of a specific execution of a TransformationStep.

### Superclasses

TransformationExecution

### References

***transformationStep***

Identifies the TransformationStep of which this is an execution.

| | |
|---|---|
| *class:* | TransformationStep |
| *defined by:* | TransformationStepExecutions::transformationStep |
| *multiplicity:* | exactly one |

***activityExecution***

Identifies an ActivityExecution of which this StepExecution is a part.

| | |
|---|---|
| *class:* | ActivityExecution |
| *defined by:* | ActivityStepExecutions::activityExecution |
| *multiplicity:* | zero or one |
| *inverse:* | ActivityExecution::stepExecution |

***callAction***

Where a TransformationStep involves the use of an Operation, a CallAction may be used to record details of the actual parameters used in the StepExecution.

| | |
|---|---|
| *class:* | CallAction |
| *defined by:* | StepExecutionCallAction::callAction |
| *multiplicity:* | zero or one |

## 19.3.5 TransformationExecution

A TransformationExecution is used to record details of a specific execution.

### Superclasses

ModelElement

### Attributes

***startDate***

The date and time when the execution started.

| | |
|---|---|
| *type:* | Time |
| *multiplicity:* | exactly one |

***endDate***

The date and time when the execution ended.

| | |
|---|---|
| *type:* | Time |
| *multiplicity:* | zero or one |

***inProgress***

A boolean indicating whether or not the execution is in progress.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | exactly one |

***successful***

A boolean indicating whether or not the execution completed successfully.

| | |
|---|---|
| *type:* | Boolean |
| *multiplicity:* | zero or one |

***status***

An expression that may be used to provide status details of the execution. For example it could provide comments for a successful execution, or details of errors for an unsuccessful execution.

| | |
|---|---|
| *type:* | Expression |
| *multiplicity:* | zero or one |

### *Constraints*

If the TransformationExecution is not inProgress, the successful, status and endDate attributes must be present, and endDate must not be earlier than startDate. [C-4]

## *19.4 Warehouse Operation Associations*

The Warehouse Operation package contains the following associations, in alphabetical order:

- ActivityStepExecutions
- ModelElementChangeRequest
- ModelElementMeasurement
- StepExecutionCallAction
- TransformationActivityExecutions
- TransformationStepExecutions

### 19.4.1 ActivityStepExecutions *protected*

Identifies all the StepExecutions associated with an ActivityExecution.

*Ends*

***activityExecution***

Identifies the ActivityExecution of which the StepExecution is a part.

| | |
|---|---|
| *class:* | ActivityExecution |
| *multiplicity:* | zero or one |
| *aggregation:* | composite |

***stepExecution***

Identifies the StepExecutions recording the results of executing the individual TransformationSteps.

| | |
|---|---|
| *class:* | StepExecution |
| *multiplicity:* | zero or more |

### 19.4.2 ModelElementChangeRequest

Associates ChangeRequests with the ModelElement(s) which they affect.

*Ends*

***modelElement***

Identifies a ModelElement affected by a ChangeRequest.

| | |
|---|---|
| *class:* | ModelElement |
| *multiplicity:* | one or more |

***changeRequest***

Identifies a ChangeRequest for a ModelElement.

| | |
|---|---|
| *class:* | ChangeRequest |
| *multiplicity:* | zero or more; ordered |

### 19.4.3 ModelElementMeasurement

Associates a Measurement object to any ModelElement.

#### Ends

***modelElement***

Identifies the ModelElement to which a Measurement relates.

| | |
|---|---|
| *class:* | ModelElement |
| *multiplicity:* | exactly one |

***measurement***

Identifies a Measurement for a ModelElement.

| | |
|---|---|
| *class:* | Measurement |
| *multiplicity:* | zero or more; ordered |

### 19.4.4 StepExecutionCallAction

Where a TransformationStep involves the use of an Operation, this association between StepExecution and CallAction allows the actual parameters used in a specific execution of the TransformationStep to be recorded.

#### Ends

***stepExecution***

Identifies the StepExecution to which the CallAction applies.

| | |
|---|---|
| *class:* | StepExecution |
| *multiplicity:* | zero or more |

***callAction***

Identifies the CallAction for a StepExecution.

| | |
|---|---|
| *class:* | CallAction |
| *multiplicity:* | zero or one |

### 19.4.5 TransformationActivityExecutions

Identifies the ActivityExecutions that record details of each execution of a TransformationActivity.

*Ends*

***transformationActivity***

Identifies the TransformationActivity.

| | |
|---|---|
| *class:* | TransformationActivity |
| *multiplicity:* | exactly one |

***execution***

Identifies an ActivityExecution recording details of a specific execution of a TransformationActivity.

| | |
|---|---|
| *class:* | ActivityExecution |
| *multiplicity:* | zero or more |

### 19.4.6 TransformationStepExecutions

Identifies the StepExecutions that record details of each execution of a TransformationStep.

*Ends*

***transformationStep***

Identifies the TransformationStep.

| | |
|---|---|
| *class:* | TransformationStep |
| *multiplicity:* | exactly one |

***execution***

Identifies a StepExecution recording details of a specific execution of a TransformationStep.

| | |
|---|---|
| *class:* | StepExecution |
| *multiplicity:* | zero or more |

## 19.5 OCL Representation of Warehouse Operation Constraints

| | |
|---|---|
| [C-1] | A ChangeRequest instance must not apply to itself. |
| **context** | ChangeRequest |
| **inv:** | self.modelElement -> forAll (element \| element <> self) |

| | |
|---|---|
| [C-2] | A completionDate may only be provided for a completed ChangeRequest. |
| **context** | ChangeRequest |
| **inv:** | self.completionDate->notEmpty **implies** self.completed |

| | |
|---|---|
| [C-3] | A Measurement instance must not apply to itself. |
| **context** | Measurement |
| **inv:** | self.modelElement <> self |

| | |
|---|---|
| [C-4] | If the TransformationExecution is not inProgress, the successful, status and endDate attributes must be present, and endDate must not be earlier than startDate. |
| **context** | TransformationExecution |
| **inv:** | self.inProgress=false **implies** (self.successful->notEmpty and self.status->notEmpty and self.endDate->notEmpty and self.endDate >= self.startDate) |

# Compatibility with Other Standards 20

## 20.1 Introduction

This section identifies, at a very high level, points of both commonality and divergence between CWM and the following, existing metadata standards:

- The MetaData Coalition's MetaData Interchange Specification (MDIS), Version 1.1.
- The Meta Data Coalition's Open Information Model, Version 1.0.
- The OLAP Council's Multidimensional API (MDAPI), Version 2.0.

Only major commonalities or differences are emphasized. This section can serve as the starting point for any alignment effort one may want to undertake between CWM and any one of the other standards. However, it is not intended to be detailed enough to specify all possible requirements for alignment.

## 20.2 Background: Components of the OMG Metamodeling Architecture

The CWM specification addresses the metadata interchange requirement of the OMG repository architecture specific to the data warehousing domain. The CWM specification leverages the following standards:

- MOF, the Meta Object Facility, is an OMG metadata interface standard that can be used to define and manipulate a set of interoperable metamodels and their instances (models). The MOF also defines a simple meta-metamodel (based on the OMG UML - Unified Modeling Language) with sufficient semantics to describe metamodels in various domains starting with the domain of object analysis and design. CWM uses MOF as its meta-metamodel.
- UML, the Unified Modeling Language, is an OMG standard modeling language for specification, construction, visualization and documentation of the artifacts of a software system. CWM uses UML as its graphical notation, and defines a base metamodel (i.e., the CWM Object Model) that is consistent with the core UML metamodel.

- XMI, or XML Metadata Interchange, is an OMG standard mechanism for the stream-based interchange of MOF-compliant metamodels. XMI is essentially a mapping of the W3C's eXtensible Markup Language (XML) to the MOF. By being implicitly MOF-compliant, any CWM model instance can be interchanged by enabled tools using the facilities of XMI.

In summary, CWM is a domain-specific extension of the OMG's Metamodeling Architecture, and as such, implicitly supports the MOF, UML and XMI standards. Although CWM has certain "compatibilities" with various other standards (as outlined in subsequent sections), these compatibilities should be regarded as touch points for mapping or integration; they do not represent dependencies of any kind. CWM is not dependent upon any standards outside of those of the OMG Metamodeling Architecture.

## 20.3 CWM and MDC Meta Data Interchange Specification

### 20.3.1 Overview

The Meta Data Coalition's MetaData Interchange Specification (MDIS) is a non-proprietary and extensible mechanism for the interchange of meta data between MDIS-aware tools.

MDIS Version 1.1 consists of a metamodel, which defines the syntax and semantics of the metadata to be exchanged, as well as the specification of a framework for supporting an actual MDIS implementation. The MDIS Metamodel is a hierarchically-structured, semantic database model that's defined by a tag language. The metamodel consists of a number of generic, semantic constructs, such as Element, Record, View, Dimension, Level, and Subschema, plus a Relationship entity that can be used in the specification of associations between arbitrary source and target constructs. The MDIS metamodel may be extended through the use of named properties that are understood to be tool-specific and not defined within MDIS. Interchange is accomplished via an ASCII file representation of an instance of this metamodel. Although support for an API is mentioned in the specification, no API definition is provided.

The MDIS Access Framework specifies several fairly general mechanisms that support the interchange of metamodel instances. The Tool and Configuration Profiles define semaphores that ensure consistent, bidirectional metadata exchange between tools. The MDIS Profile defines a number of system parameters (environment variables) that would be necessary in the definition of an MDIS deployment. Finally, Import and Export functions are exposed by the framework as the primary file interchange mechanisms for use by tools.

### 20.3.2 Comparison with CWM

Each of the following bullet items identifies a relevant comparison point between MDIS and CWM, and describes the degree to which the two standards either converge or diverge.

- Scope. In general, the overall scope of the MDIS specification is considerably narrower than that of the CWM. Whereas the CWM defines a metamodel of a complete data warehouse (including various types of databases and data sources, specification of warehouse processes and deployment structures, and transformations between data sources and targets), MDIS is restricted to the specification and interchange of database schema concepts only. While MDIS is sufficiently general to specify just about any conceivable database schema, there is no explicit support for any process-oriented semantics. For example, an MDIS metamodel could define a mapping (association) between a relational source and OLAP target, but can not specify the transformation logic at the meta-level (this would have to be done within tool-specific content areas of the interchange structure).
- Separation of Metamodels and Instances. MDIS is rather monolithic in that there is no crisp separation between the MDIS metamodel and its instances. Both are interchanged in a single ASCII file, with instances realized by values associated with metamodel tags. There is no provision for a separate definition of the metamodel itself, apart from an instance. It is not possible for two or more instances to refer to a single metamodel definition. Instead, the metamodel definition must be copied into each instance. In comparison, the CWM metamodel, by virtue of XMI, has a normative expression that's completely independent of any of its instances. This normative expression is in the form of an XML Document Type Definition (DTD), and instances, which are streamed via XML Documents, can simply contain references to their defining DTDs.
- Visual Modeling Support. The MDIS metamodel has a "text-oriented" definition, with no obvious support for graphically-oriented expressions. The CWM metamodel, on the other hand, is an extension of the UML metamodel. This means that any graphical tool (CASE tool, Web browser, etc.) that understands the UML metamodel can also be easily enabled to render the CWM metamodel and, therefore, CWM model instances.
- Tag Language. The tag language used to define the MDIS metamodel is specific to MDIS only. While non-proprietary in the sense of tool-specific implementations, it does not enjoy the same level of broad, industry acceptance that XML does today.
- API Support. Since CWM is MOF-compliant, the CWM metamodel has inherent API support in terms of CORBA IDL. Furthermore, this API support can be mapped to almost any programming language for which an IDL (or straight MOF) mapping exists. MDIS, on the other hand, does not appear to support an API. This is a disadvantage because there is no way to acquire "fine-grained", programmatic access to the MDIS metamodel.
- Relative Cost of Entry. Implementing MDIS requires the writing of interpreters of the ASCII-based, MDIS metamodel to function according to the MDIS specification. On the other hand, an XMI rendering of CWM can be consumed and validated by any (relatively inexpensive or free) XML parser. The consuming XML application can then easily make use of other XML standard facilities (such as DOM) for browsing or manipulating the metamodel and its instance data.

In conclusion, CWM is more comprehensive in scope than MDIS 1.1. CWM is more powerful, more flexible, and easier to adopt and use than MDIS, mainly because it

leverages facilities already defined by the OMG Metamodeling Architecture (i.e., MOF, UML and XMI), and because there is widespread industry support for these standards and their attendant implementation technologies (such as XML parsers). Although CWM is oriented to the data warehousing environment, the degree of package separation in the CWM metamodel means that submodels can easily be co-opted for other purposes. Any thing that can be accomplished using MDIS can be accomplished using CWM.

However, in all fairness, it should be noted that MDIS is a relatively older standard that was crafted prior to the widespread acceptance of technologies such as UML and XML, and that it could not have possibly leveraged such technologies at the time it was drafted. MDIS represents a noble early attempt at defining a metadata interchange standard and is a baseline against which subsequent standards must be compared. At the time of this writing, the MDC has decided that MDIS will be superceded by OIM, which is discussed next.

## 20.4 CWM and MDC Open Information Model

### 20.4.1 Overview

The Meta Data Coalition's Open Information Model (OIM) is a non-proprietary and technology-neutral, and extensible specification of the core metadata types that are representative of enterprise-wide information architectures and environments. This enterprise-wide view includes analysis and design, objects and components, database and warehousing, and knowledge management, so in this sense, the scope of the OIM is much broader than that of the CWM, which is focused primarily on the data warehousing domain.

MDC-OIM was originally developed primarily by Microsoft Corporation and Platinum Technology. OIM was subsequently transferred to the MDC, under whose auspices it continues to evolve as a public-domain specification.

MDC-OIM uses UML as its formal specification language. OIM defines common representations of various types of data sources and targets (record, relational, OLAP) and transformations between sources and targets. The OIM metamodel derives from the UML metamodel, and the OIM specification claims that OIM has a repository orientation, but unlike CWM, is not compliant with the MOF. OIM does not use XMI as an interchange mechanism. Rather, it uses a specific OIM to XML encoding to generate interchange files.

The following subsections describe commonalities and differences between CWM and OIM. In the interests of specificity, these comparisons are limited to the salient features of the Database Schema, Data Transformation, OLAP Schema and Record-Oriented Database Schema models. These comparisons can serve as the starting point for an alignment exercise between CWM and OIM in these model areas, but it should be noted that not all possible points of convergence and divergence are covered here.

### 20.4.2 Comparison with CWM: Database Schema

The MDC-OIM Database Schema is a metamodel describing relational data sources. Just as with CWM, the purpose of the relational metamodel is to provide a means by which tools may exchange commonly-understood descriptions of relational schemas, with the possible inclusion of tool-specific extensions. It is modeled largely after the ANSI SQL-92 standard. Here are the major comparison points between the CWM Relational Package and the OIM Database Schema:

- Reference standards. OIM is based on the SQL-92 standard, while CWM is based on the SQL-99 standard and is compatible with JDBC.
- Base metaclasses. Both OIM and CWM have fairly similar base metaclass structures, centered on the notion of column set and the subsequent derivation of table, view and query from the column set.
- Keys and indexes. The concepts of keys (unique keys, foreign keys) and indexes are defined in the CWM as CWM Foundation metaclasses, so they have general applicability to other data models within the CWM, not just the CWM Relational Package. OIM confines keys and indexes to its relational schema. Hence, only OIM data source models that derive from, or are based on, the Database Schema, can provide these concepts.
- Catalog and schema. Both the CWM and OIM relational models support the basic structure of catalogs containing schemas and schemas, in turn, containing all other relational objects.
- Deployment structures. The OIM generally provides Logical and Deployment subclasses of all of its major semantic classes throughout the OIM Database Schema. For example, LogicalTable and DeployedTable both derive from the (semantic) Table metaclass. However, these Logical and Deployed subclasses are generally not defined much further, except DeployedCatalog is represented as being owned by a DataSource which in turn has associations with metaclasses representing Connections and Providers. Note that most of the OIM models derive from the Database Schema model; hence, the ultimate deployment of any part of the OIM must be via mappings to the Database Schema (relational) metamodel. The overall deployment structures of the CWM metamodel, by comparison, are much more general than this. CWM defines a Software Deployment metamodel which defines concepts of providers, data managers, and connections. Any logical data model (whether Relational, Multidimensional, Record) models its own deployment by mapping to an appropriate metaclass of the CWM Software Deployment package. For example, the Catalog metaclass of the CWM Relational metamodel is implicitly owned by the DataManager metaclass of the Software Deployment metamodel, and this metamodel in turn relates the physical DataManager to its associated DataProviders, ProviderConnections, Machine, Site, and most importantly, deployment-specific TypeMappings (which in turn derive from the CWM Foundation package).

### 20.4.3 Comparison with CWM: Data Transformations

The MDC-OIM Data Transformations metamodel, like its CWM counterpart, defines metadata that describes the processes which map and transform the contents of various source and target data stores. This might include, for example, the transformation of operational data to a normalized, relational representation or analysis-oriented store. Both also provide facilities whereby data lineage may be tracked across a series of transformations.

There are, however, some fundamental differences between the two metamodels. In particular, the OIM Data Transformation model is specific to the OIM Database Schema model. In its current form, it can describe relational-to-relational transformations only, and has certain dependencies on the Database Schema package (e.g., the CodeDecodeSet derives from Database Schema Columns).

The CWM Transformation package, on the other hand, is more generalized and is not tied to any one particular data store or schema. This is because the CWM Transformation package describes transformational mappings in terms of the Object Model core metaclasses of Classifier and Feature. Hence, transformation mappings may be defined on any CWM metaclasses that derive from these metaclasses.

For example, under CWM, Relational Tables and Multidimensional Dimensions derive from Object Model Class, respectively, and CWM Relational Columns and Multidimensional DimensionedObjects derive from Object Model Attribute, respectively. So the same Transformation metamodel can be used to describe both relational-to-relational mappings, as well as relational-to-multidimensional mappings.

The CWM and MDC-OIM metamodels are most similar, however, in their overall representation of the transformation process. Both metamodels support the specification of transformations in terms of TransformationSteps, TransformationTasks, and dependencies or constraints between steps. Both support the generic specification of Transformation logic based on expressions; however, CWM Transformations can be specified using either an opaque expression (a textual string) or a tree-based expression structure (which comes from the CWM Foundation package's Expression model). Using structured expressions further facilitates the tracking of transformation lineage.

The historical records of transformations are modeled in similar ways in CWM and MDC-OIM. OIM's StepExecution and ActivityExecution correspond to similar objects in the CWM Warehouse Operation package.

### 20.4.4 Comparison with CWM: OLAP Schema

MDC-OIM provides an OLAP Schema metamodel for describing the use of multidimensional database technology within the enterprise in support of advanced business analytics and decision support capabilities. OLAP technology has broad applicability, both within the data warehousing environment, specifically, and across the enterprise, in general. Hence, both CWM and OIM have a requirement for representing OLAP and multidimensional metadata.

The CWM and MDC-OIM OLAP metamodels have many similarities, but many fundamental differences, as well. Perhaps the most fundamental difference is in the overall orientation of the two metamodels.

The CWM OLAP metamodel is a pure, semantic model of general OLAP concepts, and does not define any particular logical or physical deployment constructs of its own. This is done for two reasons:

- OLAP and multidimensional concepts (what the user sees) tend to be rather abstract in nature and very broad in applicability; for example, notions such as “dimension” and “dimensioned variable” are concepts that span the enterprise and really aren’t specific to any particular technology that provides computational support for such concepts.
- OLAP concepts may be implemented in many different ways, depending on the objectives of the enterprise and the technologies available. For example, OLAP applications are often implemented using either relational database technology (ROLAP), multidimensional database servers (MOLAP), or some hybrid mixture of the both relational and multidimensional technologies.

So the CWM OLAP metamodel defines generic OLAP concepts only and leverages the CWM Transformation metamodel to map OLAP metaclasses to metaclasses of other packages that could be used to describe logical models of implementations (e.g., the CWM Relational and Multidimensional metamodels). Those logical models, in turn, rely on the Software Deployment metamodel to describe their actual, physical deployments.

The MDC-OIM OLAP model, on the other hand, is largely derived from the OIM Database Schema model (in the same manner that the Data Transformation model is). For example, Cubes and Partitions are ultimately derived from ColumnSet. This may have the effect of restricting the usage of the OIM OLAP model to the representation of relational-OLAP constructs only.

The OIM OLAP model also includes a number of logical and physical deployment metaclasses, such as OLAPServer, DataSource, and Connection metaclasses, plus DeployedOLAPDatabase and LogicalOLAPDatabase subclasses, in keeping with the OIM’s overall dichotomization of the concepts of logical versus deployed subclasses. As stated earlier in the discussion on the relational Database Schema, there is no need for the CWM OLAP metamodel to include these kinds of metaclasses, since logical descriptions are implicitly defined by transformation mappings of OLAP semantics to more logical constructs (e.g., relational), and the physical deployment metaclasses are provided within a single, Software Deployment metamodel.

Areas where the CWM OLAP and OIM OLAP metamodels are mostly (though not completely) similar include the following:

- Cubes and Dimensions. Both metamodels support the concept of Cubes and Dimensions being separate from one another and both contained within an OLAP Database (called Schema in CWM). Both support the special designation of a Time Dimension, although the CWM OLAP metamodel further defines a Measures Dimension. Both metamodels also support the concepts of *virtual* versus *physical* Cubes, as well as the concept of a Cubes being composed from *sub-cubes* (called

Cube Regions by CWM and Partitions by OIM). However, OIM includes the notion of an Aggregation metaclass, which represents pre-calculated aggregations in relational stores, generally what one might find in a typical, relational Star-schema deployment of OLAP. CWM provides no such concept, because this is regarded as being an implementation detail that would be addressed at the model instance level.

- Levels and Hierarchies. Both OLAP metamodels support the concept of Hierarchy as being a separate entity from its owning Dimension. Both metamodels support the concept of multiple Hierarchies per Dimension. Both metamodels also support the concepts of Dimension Levels and the association of Dimension Levels with Dimension Hierarchies, and both also define mapping constructs that enable Hierarchies and Levels to be mapped to logical deployment structures. However, within the OIM OLAP metamodel, these deployment mappings are explicitly geared toward a relational database (and optionally Star-Schema) deployment, whereas the CWM OLAP contains mapping constructs that derive from more general CWM Transformation mapping metaclasses and, hence, can be used to specify deployment mappings to any conceivable logical structure that might be supported elsewhere within the CWM metamodel.

### 20.4.5 Comparison with CWM: Record-Oriented Database Schema

The MDC-OIM Record-Oriented Database Schema is a metamodel describing record-oriented data sources. Just as with CWM, the purpose of the record-oriented metamodel is to provide a means by which tools may exchange commonly-understood descriptions of record-oriented data resources, with the possible inclusion of tool-specific extensions. Here are the major comparison points between the CWM Record package and the OIM Record-Oriented Database Schema:

- Scope. OIM limits the scope of its record-oriented model to database schemas. CWM, in contrast, permits the description of a broader range of record data resources including both traditional record-oriented resources such as databases, files, and programmatic data structures and non-traditional, hierarchical data resources such as documents, reports, and forms.
- Specificity. OIM includes metaclasses supporting a number of language-specific constructs such as COBOL renaming and data structure overlay capabilities and source management constructs such as Copylibs. Many of these constructs are not reusable by other programming languages that support similar notions. CWM, on the other hand, models such capabilities in a general fashion and relegates language-specific constructs to the appropriate language extension packages.

## 20.5 CWM and OLAP Council/MDAPI

### 20.5.1 Overview

The OLAP Council's Multidimensional API (MDAPI) is a non-proprietary specification for an object-oriented API that exposes a full range of OLAP functions that a given vendor's implementation of an OLAP product might want to support. This includes: Server connection and login, Metadata querying functions, multidimensional

data querying functions, generic filtering and sorting capabilities, and error handling and progress monitoring functions. Vendors implementing the MDAPI may also add their own extensions whereever necessary, through pass-through capabilities inherent in the MDAPI.

The MDAPI provides a query-oriented interface to an OLAP metadata/data provider (such as an OLAP server) that can be used to expose both metadata and data cell contents of the provider, and supports the incremental modification of queries, as well as the navigation of result sets and extraction of values from result sets.

### *20.5.2 Comparison with CWM*

There are a number of fundamental differences between the MDAPI and the CWM that make direct comparisons some what difficult.

First of all, the MDAPI is an implementation model, rather than a metamodel. The MDAPI primarily defines interfaces that can be used to query metadata from an OLAP metadata provider, which usually (but not necessarily) means a commercially-available OLAP server. For example, an OLAP server can utilize both the CWM OLAP metamodel and the MDAPI in following manner:

The server initially consumes a CWM model instance and sets up its internal, multidimensional metadata structures accordingly. After the server has been loaded with data input values and calculations, etc., are performed, clients of the server could then issue multidimensional queries against the server through the MDAPI. This has the benefit of providing a unified metadata instance and data querying mechanism. For example, a user can define several metadata queries to subset Dimension Members and then issue a data query that uses the metadata query result sets as the basis for forming and exposing a data result (essentially a cube region or cube view). In this scenario, CWM is used to define the core OLAP metadata to a CWM-enabled provider, and the provider exposes the MDAPI as its primary client interface for exposing both metadata instances and multidimensional data values.

Note that, since a CWM model instance is MOF-compliant, instances of CWM metaclasses have inherent support for CORBA (or programming language mapped) interfaces that provide access and navigation of the model itself. However, this is not necessarily sufficient for integrated multidimensional metadata and data querying, which requires support for generating and navigating result sets, among other things (since the CWM OLAP metamodel is a semantic model and not an implementation model, it defines neither behavioral semantics, nor interfaces). Hence, the MDAPI and CWM can play rather complementary roles in the deployment of a multidimensional data server.

The key to integrating the CWM and the MDAPI in the manner described above is through the alignment of the CWM OLAP metamodel and MDAPI data model, a conceptual model that defines the semantic underpinnings of the metadata objects and interfaces. Alignment, in this case, would generally consist of mapping the major classes of the MDAPI data model to the CWM OLAP metaclasses. The following paragraphs do not attempt such a detailed mapping/derivation, but rather just point out some of the major areas of correspondence between the two models:

- Cube. MDAPI, being primarily a query model, does not define the notion of Cube as a persistent, multidimensional database, but rather defines a Cube View. Cube View corresponds closely to the CWM OLAP concept of Cube Region, if the Cube Region's formula is interpreted as the multidimensional query processed by the Cube View.
- Dimension. Both the MDAPI data model and CWM OLAP metamodel support similar concepts of Dimension and Dimension types.
- MemberSelection. Both model support the concept of a member query on a Dimension. This is called MemberSelection by CWM, and Membership by MDAPI. In both models, this member query is expression based.
- Hierarchy and Level. Both models support the concepts of Hierarchy and Level and associations between them. A Dimension can have an arbitrary number of Hierarchies in either model. In the MDAPI data model, Dimension, Hierarchy, and Level are all subclasses of Membership, and are all, therefore, expression (query) based by default. In the CWM OLAP metamodel, only Level derives from MemberSelection, but the correspondence in this regard is close enough.
- Properties. The MDAPI data model supports user-defined property types and values as a means of extending the core data model. A client of the metadata and data query objects (MemberSelection and CubeView) can specify both searches and sorts based on property types and value or ranges of values. The closest equivalent the CWM OLAP metamodel has in this regard is the general association to UML Attributes that's inherited by any subclasses of the core UML Class. So, at least at the instance level, there is a close correspondence between both models in this regard, as well.

# Conformance Points 21

## 21.1 Introduction

This section describes the required and optional points of compliance with the CWM specification.

## 21.2 Required Compliance

### 21.2.1 CWM Metamodel Compliance

A CWM-compliant warehouse platform is required to implement the following packages:

- ObjectModel
- Foundation
- Transformation
- Warehouse Process
- Warehouse Operation

A warehouse platform provides generic capabilities for integrating different types of warehouse tools and for managing warehouse processes and warehouse operations.

### 21.2.2 CWM XML Compliance

The CWM XML is a normative part of CWM. This definition must be used when interchanging the CWM metamodel, in accordance with the XMI specification.

### 21.2.3 CWM IDL Compliance

The CWM IDL is a normative part of CWM. This definition, or equivalent OMG-compliant language bindings, must be used for programmatic access to warehouse metadata conforming to the CWM metamodel, in accordance with the MOF specification.

### 21.2.4 CWM DTD Compliance

The CWM DTD is a normative part of CWM. This definition must be used when interchanging warehouse metadata conforming to the CWM metamodel, in accordance with the XMI specification.

## 21.3 Optional Compliance Points

A CWM-compliant warehouse platform or warehouse tool that supports relational data resources is required to implement the following package and its dependencies:

- Relational

A CWM-compliant warehouse platform or warehouse tool that supports record data resources is required to implement the following package and its dependencies:

- Record

A CWM-compliant warehouse platform or warehouse tool that supports multidimensional data resources is required to implement the following package and its dependencies:

- Multidimensional

A CWM-compliant warehouse platform or warehouse tool that supports XML data resources is required to implement the following package and its dependencies:

- XML

A CWM-compliant warehouse tool that provides data transformation functionality is required to implement the following package and its dependencies:

- Transformation

A CWM-compliant warehouse platform or warehouse tool that provides OLAP functionality is required to implement the following package and its dependencies:

- OLAP

A CWM-compliant warehouse platform or warehouse tool that provides data mining functionality is required to implement the following package and its dependencies:

- Data Mining

A CWM-compliant warehouse platform or warehouse tool that provides information visualization functionality is required to implement the following package and its dependencies:

- Information Visualization

A CWM-compliant warehouse platform or warehouse tool that provides or handles business metadata is required to implement the following package and its dependencies:

- Business Nomenclature

# CWM Data Types 22

## 22.1 Overview

The CWM Foundation, in its DataTypes package, provides metamodel types supporting definition of data types required by data sources, data targets, and tools that implement transformations between them. Although these metamodel types are sufficient to permit the definition of most data types, they do not themselves actually create definitions of data types. This is because the metamodel types are M2 level types whereas data type definitions are M1 level definitions.

This approach to the creation of data types was chosen because the specific data type needs of individual transformation tools and source and target data systems are sufficiently different that their interchange cannot be specified fully in advance. Unfortunately, data type incompatibility is often true even for systems that claim to support the same data language (consider, for example, the many variants of “SQL”). Even though some tools and systems may enjoy compatibility for commonly used data types (such as *integer* and *string*), systems which are compatible across the full range of their data types are indeed rare.

Data type incompatibilities between systems result from a number of factors including specific characteristics of hardware implementation platforms, software vendors’ desire to differentiate their products in the marketplace, and other, largely historical, causes. These factors combine to make definition of a common set of data types supporting the diverse, and frequently incompatible, needs of existing and future CWM-compliant tools impossible in any practical way. Consequently, modelers of software systems in CWM may find it necessary to create both data type definitions compatible with their tools and to create TypeMapping instances to indicate mappings between their tools’ data types and the native data types of systems with which they interchange data.

Nevertheless, the CWM recognizes the importance of shared data types -- especially those based on industry standards such as CORBA IDL, SQL and Java -- as a means of promoting data interchange between disparate systems. Consequently, this chapter

provides a set of data type definitions for several widely used industry specifications. These data type definitions serve two purposes within the CWM:

- Provide a pre-defined basis for data interchange among diverse tools and systems that support a selection of standard data types.
- Provide examples of the appropriate use of the CWM Foundation's metamodel types for creating tool-specific data type definitions.

To further promote understanding of the appropriate use of other CWM Foundation metamodel types, this chapter also contains examples showing how tool-specific expressions can be mapped into the CWM Foundation's expression metamodel types.

In general, the CWM packages only support data type attributes that are considered necessary for interchange of information between systems; attributes that are thought to be system specific are left to tool modelers. When such attributes must be represented, modelers may create model-specific types that derive from supplied CWM types and house the necessary attributes therein.

The information and definitions in this chapter, while considered important to accomplishing the overall goals of CWM, are supplementary in nature and are not considered a normative part of the CWM specification.

## 22.2 Organization of the CWM Data Types

The CWM DataTypes contains definitions of data types for the CORBA IDL language [CORBA], the SQL-99 language [SQL], and the Java programming language [Java]. Because they are M1 level entities, data type definitions for these languages are expressed in a tabular form that indicate the instances of M2 level CWM metaclasses that can be created in an appropriate CWM metadata store to define the M1 level data types. The data type definitions might then be used to create M1 level models appropriate for specific tools and software systems.

The example M1 instances define only primitive data types; structured data types are not generally defined in these examples. (However, the CORBA IDL metamodel types required to define M1 structured types are provided as an example of how this might be done, if needed.) Data types that require no additional information to complete their definition, such as SQL's INTEGER type, are completely defined. However, data types that are in some sense "parameterized," such as SQL's CHARACTER(n) and FIXED(p, s) data types, are incompletely defined because it is not practical to anticipate all possible parameter values! Tools that need to declare such parameterized data types should do so as they encounter them. The data type instances in this chapter define a few parameterized data types, where appropriate, as examples.

As an example of appropriate usage of the CWM Foundation's TypeMapping metamodel to indicate preferred and non-preferred mappings between the data types of different tools and software systems, the last section of the chapter contains some example mappings between Java and CORBA IDL and between Java and SQL-99.

## 22.3 CORBA IDL Data Types

The CORBA IDL Data Types package depends on the following packages:

- org.omg::CWM::ObjectModel::Core
- org.omg::CWM::Foundation::DataTypes

### 22.3.1 Overview

A CORBA IDL metamodel extension to the CWM Foundation is required to support the CORBA IDL data types in the CWM model. It is provided here as an example of extending the DataTypes metamodel and is not a normative part of the CWM specification.

The chief motivation for the creation of this metamodel is the need to provide a ***typeCode*** attribute for CORBA IDL data types. These extensions also serve as an illustration of the use of CWM Foundation metamodel types as superclasses of the metamodel types for a specific language environment.

### 22.3.2 Organization of the CORBA IDL Data Types

Because the M1 data type instances are of primary import and because of the length of the metamodel subsection, the M1 instances are described before the metamodel types. When reviewing the M1 instances, refer to the appropriate metamodel type definitions and the following figure for more information about metamodel types.

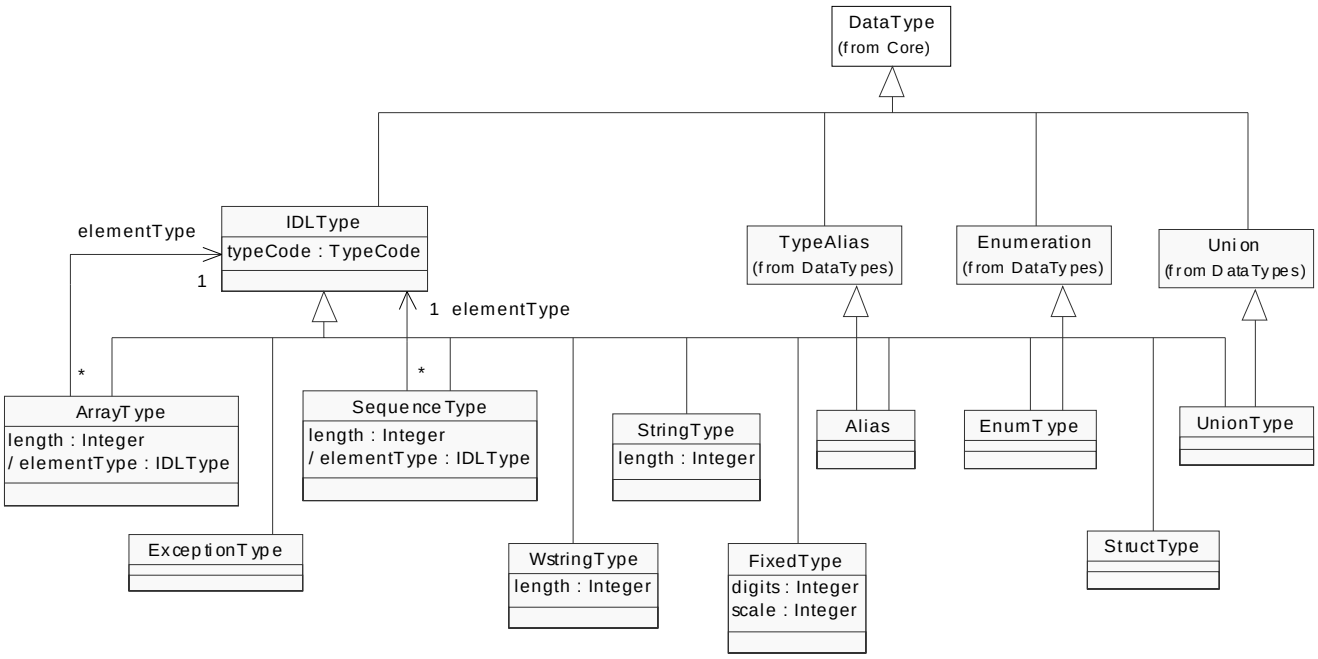


*Figure 22-3-1* CORBA IDL data type metamodel types.

### 22.3.3 CORBA IDL Data Type Instances

Data type instances for CORBA IDL non-structured data types are presented in the following table. The M1 data types instances correspond to those described in the CORBA IDL language specification.

**Table 22-3-1 CORBA IDL primitive data type instances.**

| CORBA IDL Data Type | Instance of | Attributes |
|---|---|---|
| any | IDLType | typeCode = tk_any |
| octet | IDLType | typeCode = tk_octet |
| boolean | IDLType | typeCode = tk_boolean |
| char | IDLType | typeCode = tk_char |
| wchar | IDLType | typeCode = tk_wchar |
| short | IDLType | typeCode = tk_short |
| long | IDLType | typeCode = tk_long |
| long long | IDLType | typeCode = tk_longlong |
| unsigned short | IDLType | typeCode = tk_ushort |
| unsigned long | IDLType | typeCode = tk_ulong |
| unsigned long long | IDLType | typeCode = tk_ulonglong |
| float | IDLType | typeCode = tk_float |
| double | IDLType | typeCode = tk_double |
| long double | IDLType | typeCode = tk_longdouble |

### 22.3.4 CORBA IDL Data Types Classes

CORBA IDL metamodel classes are provided to support the definition of CORBA data types that cannot be represented simply as instances of the IDLType class. This group of types includes all CORBA structured and array-like data types as well as those that also derive from the types defined in the CWM Foundation's Data Types conceptual area.

#### 22.3.4.1 Alias

The Alias type represents CORBA IDL type aliases. Aliases must be represented by their own type so that they can have a typeCode attribute as required by the CORBA IDL definition.

##### *Superclasses*

IDLType

TypeAlias

### 22.3.4.2 ArrayType

The ArrayType class represents CORBA IDL array data types.

#### Superclasses

IDLType

#### Attributes

***length***

The number of elements in the array. Multiply dimensioned arrays are treated as arrays of array in CORBA IDL.

| | |
|---|---|
| *type:* | Integer |
| *multiplicity:* | exactly one |

#### References

***elementType***

The type of elements of an array.

| | |
|---|---|
| *class:* | IDLType |
| *defined by:* | ArrayElementType::elementType |
| *multiplicity:* | exactly one |

### 22.3.4.3 EnumType

The EnumType class represents the CORBA IDL enumerated data type, enum.

#### Superclasses

IDLType

Enumeration

### 22.3.4.4 ExceptionType

The ExceptionType class represents the CORBA IDL exception data type.

#### Superclasses

IDLType

### 22.3.4.5 FixedType

The FixedType class represent CORBA IDL fixed data types.

#### Superclasses

IDLType

#### Attributes

***digits***

Number of digits of precision.

| | |
|---|---|
| *type:* | Integer |
| *multiplicity:* | exactly one |

***scale***

Number of implied decimal places. Scale may be either positive (implied left decimal places) or negative (implied right decimal places).

| | |
|---|---|
| *type:* | Integer |
| *multiplicity:* | zero or more |

### 22.3.4.6 IDLType

The IDLType class is a common superclass for all CORBA IDL data type classes that require a typeCode.

#### Superclasses

DataType

#### Attributes

***typeCode***

The type code value identifying a CORBA IDL data type.

| | |
|---|---|
| *type:* | TypeCode |
| *multiplicity:* | exactly one |

### 22.3.4.7 SequenceType

The SequenceType class represents CORBA IDL sequence data types. Sequences are single dimensioned arrays of a user-specified type.

#### Superclasses

IDLType

#### Attributes

***length***

The number of elements in the sequence expressed in type units.

| | |
|---|---|
| *type:* | Integer |
| *multiplicity:* | exactly one |

#### References

***elementType***

The type of elements of a sequence.

| | |
|---|---|
| *class:* | IDLType |
| *defined by:* | SequenceElementType::elementType |
| *multiplicity:* | exactly one |

### 22.3.4.8 StringType

The StringType class represents CORBA IDL string data types.

#### Superclasses

IDLType

### Attributes

***length***

The number of characters in the string. If length is zero, the string in considered unbounded.

| | |
|---|---|
| *type:* | Integer |
| *multiplicity:* | exactly one |

## 22.3.4.9 StructType

The StructType class represents CORBA IDL user -defined data types created with the *typedef* keyword.

### Superclasses

IDLType

## 22.3.4.10 UnionType

The UnionType class represents CORBA IDL union data types.

### Superclasses

IDLType

Union

## 22.3.4.11 WstringType

The WstringType class represents CORBA IDL wstring data types. A CORBA wstring is an ordered sequence of wchars, each of which represents a ‘wide’ character from any character set.

### Superclasses

IDLType

*Attributes*

***length***

The number of wchars in the string. If length is zero, the string in considered unbounded.

| | |
|---|---|
| *type:* | Integer |
| *multiplicity:* | exactly one |

### *22.3.5 CORBAL IDL Data Types Associations*

#### *22.3.5.1 ArrayElementType* *Protected*

Associates an ArrayType with the type of its elements.

*Ends*

***arrayType***

Arrays having elements of this type.

| | |
|---|---|
| *class:* | ArrayType |
| *multiplicity:* | zero or more |

***elementType***

Identifies the type of an array's elements.

| | |
|---|---|
| *class:* | IDLType |
| *multiplicity:* | exactly one |

#### *22.3.5.2 SequenceElementType* *Protected*

Identifies the type of elements in a sequence.

*Ends*

***elementType***

Identifies the type of elements in a sequence.

| | |
|---|---|
| *class:* | IDLType |
| *multiplicity:* | exactly one |

***sequence***

Sequences of this type.

| | |
|---|---|
| *class:* | SequenceType |
| *multiplicity:* | zero or more |

## 22.4 Java Data Types

Creation of primitive data type instances for the Java language is straightforward because they are all simple, unparameterized types. These primitive data types are used for simple declarations and for building more complex data types implemented as Java classes. Even such common data types as String are implemented as classes in Java. The CWM ObjectModel provides sufficient support for the description of Java classes that CWM classes (notably, Class and Attribute) should be used directly to define any needed Java classes. Consequently, CWM need not provide metamodel classes supporting the definition of Java classes or primitive data types -- the available CWM classes are sufficient.

The Java language specification provides additional semantics about the meaning of, and restrictions on, primitive data types. For example, the ***int*** data type is restricted to integer values in the range $-2^7$ to $2^7 - 1$. However, because these restrictions are constant for all variables of type ***int***, they do not need to be encoded into the metamodel. Consequently, the DataType class is sufficient as the container of all Java primitive data types as is shown the following table.

**Table 22-4-1 Java primitive data types**

| Data type | Instance of | Attributes |
|---|---|---|
| boolean | DataType | None |
| char | DataType | None |
| byte | DataType | None |
| short | DataType | None |
| int | DataType | None |
| long | DataType | None |
| double | DataType | None |
| float | DataType | None |

## 22.5 SQL-99 Data Types

The data types defined by the SQL-99 specification are created within CWM as instances of the Relational package's SQLSimpleType metaclass. These data type instances are a superset of those defined by the SQL-92 specification and follow the SQL-99 specification's Data_Type_Descriptor information. Practical implementations of SQL-based systems will have variations on the types presented here; consult relevant product information for details.

The SQL-99 data type instances provide a number of examples of the use of "parameterized" types. Because the CWM Relational package separates the notions of data type and column, the data type instances do not contain all seemingly relevant data type parameters. Rather, the Column instances associated with a particular Table instance contain the values of some parameters. For example, for a Column instance of declared data type DECIMAL(5, 2), the precision ("5") and scale ("2") would be recorded in the attributes Column::precision and Column::scale, respectively, whereas the DECIMAL data type instance would have its SQLSimpleType::precisionRadix attribute set to the value 10, meaning that the precision and scale values are store as base-10 numeric values. Similarly, a Column instance declared as CHARACTER(80) would have the Column::length attribute set to 80 while the CHARACTER data type's SQLSimpleType::characterOctetLength attribute would be set to value 8 indicating that the data type contains 8-bit character codes.

**Table 22-5-1 SQL-99 data type instances. Data types marked with an asterisk (*) are not part of the SQL-92 specification.**

| SQL-99 Data Type | Instance of | Attributes |
|---|---|---|
| BIT | SQLSimpleType | characterMaximumLength = IDV<br>characterOctetLength = null (defined in Column)<br>numericPrecision = null<br>numericPrecisionRadix = null<br>numericScale = null<br>dateTimePrecision = null |
| BIT VARYING | SQLSimpleType | characterMaximumLength = IDV<br>characterOctetLength = null (defined in Column)<br>numericPrecision = null<br>numericPrecisionRadix = null<br>numericScale = null<br>dateTimePrecision = null |
| BINARY LARGE OBJECT* | SQLSimpleType | characterMaximumLength = IDV<br>characterOctetLength = null (defined in Column)<br>numericPrecision = null<br>numericPrecisionRadix = null<br>numericScale = null<br>dateTimePrecision = null |
| CHARACTER | SQLSimpleType | characterMaximumLength = IDV<br>characterOctetLength = null (defined in Column)<br>numericPrecision = null<br>numericPrecisionRadix = null<br>numericScale = null<br>dateTimePrecision = null |

**Table 22-5-1 SQL-99 data type instances. Data types marked with an asterisk (*) are not part of the SQL-92 specification.**

| SQL-99 Data Type | Instance of | Attributes |
|---|---|---|
| CHARACTER VARYING | SQLSimpleType | characterMaximumLength = IDV<br>characterOctetLength = null (defined in Column)<br>numericPrecision = null<br>numericPrecisionRadix = null<br>numericScale = null<br>dateTimePrecision = null |
| CHARACTER LARGE OBJECT* | SQLSimpleType | characterMaximumLength = IDV<br>characterOctetLength = null (defined in Column)<br>numericPrecision = null<br>numericPrecisionRadix = null<br>numericScale = null<br>dateTimePrecision = null |
| NATIONAL CHARACTER | SQLSimpleType | characterMaximumLength = IDV<br>characterOctetLength = null (defined in Column)<br>numericPrecision = null<br>numericPrecisionRadix = null<br>numericScale = null<br>dateTimePrecision = null |
| NATIONAL CHARACTER VARYING | SQLSimpleType | characterMaximumLength = IDV<br>characterOctetLength = null (defined in Column)<br>numericPrecision = null<br>numericPrecisionRadix = null<br>numericScale = null<br>dateTimePrecision = null |
| NATIONAL CHARACTER LARGE OBJECT* | SQLSimpleType | characterMaximumLength = IDV<br>characterOctetLength = null (defined in Column)<br>numericPrecision = null<br>numericPrecisionRadix = null<br>numericScale = null<br>dateTimePrecision = null |
| NUMERIC | SQLSimpleType | characterMaximumLength = null<br>characterOctetLength = null<br>numericPrecision = null (defined in Column)<br>numericPrecisionRadix = 10<br>numericScale = null (defined in Column)<br>dateTimePrecision = null |

**Table 22-5-1 SQL-99 data type instances. Data types marked with an asterisk (*) are not part of the SQL-92 specification.**

| SQL-99 Data Type | Instance of | Attributes |
|---|---|---|
| DECIMAL | SQLSimpleType | characterMaximumLength = null<br>characterOctetLength = null<br>numericPrecision = null (defined in Column)<br>numericPrecisionRadix = 10<br>numericScale = null (defined in Column)<br>dateTimePrecision = null |
| INTEGER | SQLSimpleType | characterMaximumLength = null<br>characterOctetLength = null<br>numericPrecision = IDV<br>numericPrecisionRadix = 2 or 10 (IDV)<br>numericScale = 0<br>dateTimePrecision = null |
| SMALLINT | SQLSimpleType | characterMaximumLength = null<br>characterOctetLength = null<br>numericPrecision = IDV<br>numericPrecisionRadix = 2 or 10 (IDV)<br>numericScale = 0<br>dateTimePrecision = null |
| FLOAT | SQLSimpleType | characterMaximumLength = null<br>characterOctetLength = null<br>numericPrecision = IDV<br>numericPrecisionRadix = 2<br>numericScale = null<br>dateTimePrecision = null |
| REAL | SQLSimpleType | characterMaximumLength = null<br>characterOctetLength = null<br>numericPrecision = IDV<br>numericPrecisionRadix = 2<br>numericScale = null<br>dateTimePrecision = null |
| DOUBLE PRECISION | SQLSimpleType | characterMaximumLength = null<br>characterOctetLength = null<br>numericPrecision = IDV<br>numericPrecisionRadix = 2<br>numericScale = null<br>dateTimePrecision = null |
| BOOLEAN* | SQLSimpleType | characterMaximumLength = null<br>characterOctetLength = null<br>numericPrecision = null<br>numericPrecisionRadix = null<br>numericScale = null<br>dateTimePrecision = null |

**Table 22-5-1 SQL-99 data type instances. Data types marked with an asterisk (*) are not part of the SQL-92 specification.**

| SQL-99 Data Type | Instance of | Attributes |
|---|---|---|
| DATE | SQLSimpleType | characterMaximumLength = null<br>characterOctetLength = null<br>numericPrecision = IDV<br>numericPrecisionRadix = IDV<br>numericScale = null<br>dateTimePrecision = IDV |
| TIME | SQLSimpleType | characterMaximumLength = null<br>characterOctetLength = null<br>numericPrecision = IDV<br>numericPrecisionRadix = IDV<br>numericScale = null<br>dateTimePrecision = IDV |
| TIME WITH TIMEZONE | SQLSimpleType | characterMaximumLength = null<br>characterOctetLength = null<br>numericPrecision = IDV<br>numericPrecisionRadix = IDV<br>numericScale = null<br>dateTimePrecision = IDV |
| TIMESTAMP | SQLSimpleType | characterMaximumLength = null<br>characterOctetLength = null<br>numericPrecision = IDV<br>numericPrecisionRadix = IDV<br>numericScale = null<br>dateTimePrecision = IDV |
| TIMESTAMP WITH TIMEZONE | SQLSimpleType | characterMaximumLength = null<br>characterOctetLength = null<br>numericPrecision = IDV<br>numericPrecisionRadix = IDV<br>numericScale = null<br>dateTimePrecision = IDV |
| INTERVAL | SQLSimpleType | characterMaximumLength = null<br>characterOctetLength = null<br>numericPrecision = IDV<br>numericPrecisionRadix = IDV<br>numericScale = null<br>dateTimePrecision = IDV |

## 22.6 Type Mapping Examples

To promote understanding of the appropriate use of the CWM Foundation's TypeMapping package for recording mappings between data types defined by different software systems, this section presents example instances illustrating how the CORBA IDL and Java primitive data types can be mapped to each other and how the Java and SQL-99 primitive data types can be mapped to each other. These mappings are obtained from relevant published standards documents: [IDL-Java], [Java-IDL] and

[JDBC]. Although the CWM Relational package supports the SQL-99 standard, the type mappings between Java and SQL are derived from the JDBC specification which uses X/Open CLI SQL as its SQL language standard rather than SQL-99. Consequently, the Java/SQL mappings are not exactly equivalent to those that would be needed to map to SQL-99 but should serve to illustrate the mapping techniques required. SQL typeNumbers from the java.sql.Types file can be used to uniquely identify SQL types.

The following tables present sample type mapping instances for CORBA IDL/Java and Java/SQL-99 mappings. Because TypeMapping instances are unidirectional, two instances -- one for each direction -- are required to indicate that a pair of data types can be mutually interchanged. To keep the size of the tables manageable, only type mapping instances with ***isBestMatch*** = True are shown; other, non-preferred mappings can be added as necessary to support particular implementation needs. Also, values for the ***isLossy*** attribute of TypeMapping instances are omitted because their precise values are may be implementation dependent.

**Table 22-6-1 TypeMapping instances mapping CORBA IDL data types to Java data types.**

| SourceType (IDL) | TargetType (Java) |
|---|---|
| boolean | boolean |
| char | char |
| wchar | char |
| octet | byte |
| string | java.lang.String |
| wstring | java.lang.String |
| short | short |
| unsigned short | unsigned short |
| long | int |
| unsigned long | int |
| long long | long |
| unsigned long long | long |
| float | float |
| double | double |
| fixed | java.math.BigDecimal |

**Table 22-6-2 TypeMapping instances mapping Java data types to CORBA IDL data types.**

| SourceType (Java) | TargetType (IDL) |
|---|---|
| void | void |
| boolean | boolean |
| char | wchar |
| byte | octet |
| short | short |
| int | long |
| long | long long |
| float | float |
| double | double |

**Table 22-6-3 TypeMapping instances mapping X/Open CLI SQL data types to Java data types.**

| SourceType (X/Open CLI SQL) | TargetType (Java) |
|---|---|
| CHAR | String |
| VARCHAR | String |
| LONGVARCHAR | String |
| NUMERIC | java.math.BigDecimal |
| DECIMAL | java.math.BigDecimal |
| BIT | boolean |
| TINYINT | byte |
| SMALLINT | short |
| INTEGER | int |
| BIGINT | long |
| REAL | float |
| FLOAT | double |
| DOUBLE | double |
| BINARY | byte[] |
| VARBINARY | byte[] |
| LONGVARBINARY | byte[] |
| DATE | java.sql.Date |
| TIME | java.sql.Time |
| TIMESTAMP | java.sql.Timestamp |

**Table 22-6-4 TypeMapping instances mapping Java data types to X/Open CLI SQL data types.**

| SourceType (Java) | TargetType (X/Open CLI SQL) |
|---|---|
| String | VARCHAR (or LONGVARCHAR) |
| java.math.BigDecimal | NUMERIC |
| Boolean | BIT |
| Integer | INTEGER |
| Long | BIGINT |
| Float | REAL |
| Double | DOUBLE |
| byte[] | VARBINARY (or LONGVARBINARY) |
| java.sql.Date | DATE |
| java.sql.Time | TIME |
| java.sql.Timestamp | TIMESTAMP |

# *References*

## *Normative*


**[MOF]** MOF, an adopted standard of the OMG. http://www.omg.org

**[UML]** UML, an adopted standard of the OMG. http://www.omg.org

**[XMI]** XMI, an adopted standard of the OMG. http://www.omg.org

**[XML]** XML 1.0, an adopted standard of the W3C. http://www.w3c.org


## *Non-Normative*


**[CORBA]** CORBA/IIOP 2.3.1 Specification, 99-10-07

**[CORBA IDL]** CORBA IDL, an adopted standard of the OMG. http://www.omg.org/cgi-bin/doc?formal/99-10-17

**[IDL-Java]** IDL to Java Mapping, an adopted standard of the OMG. http://www.omg.org/cgi-bin/doc?formal/99-07-57

**[Java]** http://java.sun.com/docs/books/jls/html/index.html

**[Java-IDL]** Java to IDL Mapping, an adopted standard of the OMG. http://www.omg.org/cgi-bin/doc?formal/99-07-63

**[JDBC]** JDBC 2.0 API. http://java.sun.com/products/jdbc/

**[OIM]** MDC Open Information Model, Version 1.0, 1999

**[SQL]** ISO/IEC 9075-2:1999, *Information technology - Database languages - SQL - Part 2: Foundation (SQL/Foundation)*, 1999

**[WFM]** Workflow Management Facility (OMG, bom/98-06-07)

**[WfMC]** Workflow Management Coalition Standards. http://www.aiim.org/wfmc/

# *Glossary*

This glossary defines the terms that are used to describe CWM. The glossary includes concepts from the Meta Object Facility (MOF), the Unified Modeling Language (UML), and XML Metadata Interchange (XMI) for completeness. The rationale for including key MOF, UML and XMI terms is to be consistent in the definition and usage of fundamental object modeling as well as meta modeling constructs. This glossary builds on the UML 1.3, MOF 1.3 and XMI 1.1 glossaries.

Glossary entries are listed alphabetically. The new glossary entries have been marked (CWM) and mainly consist of data warehousing related terminology.

## *Scope*

This glossary includes terms from the following sources:

- Meta Object Facility 1.3 specification [MOF]
- UML 1.3 specification [UML]
- XMI 1.1 specification [XMI]
- Object Management Architecture object model [OMA]
- CORBA 2.0 [CORBA]
- W3C XML 1.0 specification [XML]

## *Notation Conventions*

The entries in the glossary usually begin with a lowercase letter. An initial uppercase letter is used when a word is usually capitalized in standard practice. Acronyms are all capitalized, unless they traditionally appear in all lowercase.

When brackets enclose one or more words in a multi-word term, it indicates that those words are optional when referring to the term. For example, *aggregate [class]* may be referred to as simply *aggregate*.

The following conventions are used in this glossary:

- *Contrast: <term>*. Refers to a term that has an opposed or substantively different meaning.
- *See: <term>*. Refers to a related term that has a similar, but not synonymous meaning.
- *Synonym: <term>*. Indicates that the term has the same meaning as another term, which is referenced.
- *Acronym: <term>*. This indicates that the term is an acronym. The reader is usually referred to the spelled-out term for the definition, unless the spelled-out term is rarely used.

The glossary is extensively cross-referenced to assist in the location of terms that may be found in multiple places.

## *Terms*

**abstract class** — A class that cannot be instantiated.

**abstraction** — A group of essential characteristics of an entity that distinguish it from other entities. An abstraction defines a boundary relative to the perspective of the viewer.

**abstract language** — A system of expression for expressing information that is independent of any particular human readable notation. Contrast: concrete language or notation. (MOF)

**actual parameter** — Synonym: argument.

**aggregate [class]** — A class that represents the "whole" in an aggregation (whole-part) relationship. See: aggregation. (UML)

**aggregation** — A special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part. See: composition.

**analysis** — A phase of the software development process whose primary purpose is to formulate a model of the problem domain. Analysis focuses on what to do, design focuses on how to do it.

**analysis time** — Refers to something that occurs during an analysis phase of the software development process.

**annotation** — Synonym: note. (MOF)

**any** — A CORBA primitive data type. A strongly typed “universal union” type that can contain any value whose type is a CORBA data type. This data type is typically used in CORBA IDL when it is not possible to choose an appropriate type at the time the

| | |
|---|---|
| | interface is defined. Use of CORBA anys entails dynamic type checking, and extra overheads in value transmission. See strong typing, *dynamic typing*, TypeCode. (CORBA) |
| **architecture** | The organizational structure of a *system*. An architecture can be recursively decomposed into parts that interact through *interface*s, relationships that connect parts, and *constraint*s on the way that parts can be assembled. |
| **argument** | A specific value corresponding to a parameter. Synonym: *actual parameter*. |
| **array** | 1. A CORBA constructed data type.<br>2. A collection (1) whose type fixes the number of elements. The ordering and uniqueness properties of an array are indeterminate. (MOF) |
| **artifact** | A piece of information that is used or produced by a software development process. An artifact can be a model, a description or a piece of software. |
| **association** | 1. A semantic relationship two or more types describes a set of connections between their respective instances. (UML)<br>2. An association (1) between classes. (MOF) |
| **Association** | A model element that defines an association (2) in a MOF metamodel. (MOF) |
| **association end** | See: association role. |
| **AssociationEnd** | A model element that defines an association end in a MOF metamodel. (MOF) |
| **association class** | A modeling element that has both association and class properties. An association class can be seen as an association that also has class, or as a class that also has association properties. (UML) |
| **association role** | The role that a *type* or class plays in an association. Synonym: association end. |
| **attribute** | 1. An attribute of an object is an identifiable association between the object and some other entity or entities. (OMA)<br>2. An attribute is a named property of a type. (UML)<br>3. An attribute is a named property of a class. (MOF) |
| **Attribute** | A model element that defines an *attribute* in a MOF metamodel. (MOF) |
| **bag** | An unordered collection in which duplicate members are allowed. (MOF) |
| **base type** | The base type of a collection (1) is the *type* (1) of its elements. |
| **behavior** | The observable effects of an operation, including its results (MOF). Synonym: behavior (OMA) |
| **binary association** | An association between two classes. The degenerate case of an n-ary association where “n” is two. |
| **boolean** | 1. A UML enumeration type whose values are true and false. (UML)<br>2. A CORBA primitive data type whose values are true and false. (CORBA) |

**builtin type** A type in a type system which is available as a predefined type in all instantiations of the type system; e.g. “short” and “string” are builtin types in CORBA IDL. Contrast: primitive type.

**boolean expression** An expression that evaluates to a boolean value.

**business metadata** Business metadata is used to help end users understand and utilize the data in the warehouse, in business terms. It describes the business context and meaning of the warehouse data. (CWM)

**CDATA section** A part of an XML Document in which any markup (e.g. tags) is not interpreted, but is passed to the application as is. (W3C)

**cardinality** The number of elements in a collection. Contrast: multiplicity.

**class** 1. A *type* (3) that characterizes objects that share the same attributes, operations, methods, relationships, and semantics. (UML)
2. An implementation that can be instantiated to create multiple objects with the same behavior. Types classify objects according to a common interface; classes classify objects according to a common implementation. (OMA)

**Class** A model element that defines a class (1) in a MOF metamodel. (MOF)

**classifier** 1. A category of UML model elements that roughly correspond to types in programming languages. The category includes association classes, classes (1), data types (2), *interface*s, subsystems and *use case*s. (UML)
2. The category of MOF model elements analogous to classifier (1):

**classifier level** In MOF metamodels and UML models, this label indicates that the labelled feature is common to all instances of its classifier. For example, a classifier level attribute of a class is common to all instances of the class. Synonym: static. Contrast: instance level. (UML, MOF)

**class diagram** A UML diagram that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships. (UML)

**class proxy** A MOF metaobject that carries the classifier level attributes and operations for an instance of a MOF class. (MOF)

**client** A type, class, or component that requests a service from another type, class or component. (UML)

**closure** The transitive closure of some object under some relationship or relationships.

**collection** 1. A group of values or objects. The values in a collection are often refered to as members or elements of the collection.
2. A collection (1) in which the members are instances of the same base type. The type of a collection is defined by the base type and a multiplicity. See: array, sequence, bag, *set*, list and unique list. (MOF)

**compile time** Indicates something that occurs during the compilation of a software module.

**component** An executable software module with an identity and a well-defined interface.

**composite [class]** A class that is related to one or more classes by a composition relationship. See: composition.

**composite aggregation** Synonym: composition.

**composition** A form of aggregation with strong ownership and coincident lifetime as part of the whole. Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it (i.e. they share lifetimes). Such parts can also be explicitly removed before the death of the composite. Composition may be recursive. Synonym: composite aggregation. (UML)

**concrete class** A class that can be directly instantiated. Contrast: abstract class.

**concrete language** Synonym: notation.

**constraint** A semantic condition or restriction. Certain constraints are predefined, others may be user defined. Constraints may be expressed in natural language or a formal language. (UML, MOF)

**Constraint** A model element that defines a *constraint* on another element in a MOF metamodel. (MOF)

**container** 1. An entity that exists to contain other entities. See containment.
2. An entity's container is the entity that contains it.

**containment** A form of aggregation that is similar to composition. The fundamental properties of containment are:

- an entity can have at most one container at any given time, and
- an entity cannot directly or indirectly contain itself.

**containment hierarchy** A containment hierarchy is a tree-shaped graph of entities, consisting of a root entity and all other entities that are directly or indirectly contained by it.

**containment matrix** A set of constraints on a containment relationship (expressible as a matrix of boolean values) that determine what other kinds of entities a given kind of entity can contain. For example, the MOF Model definition includes such a matrix to specify which concrete subclasses of ModelElement can be contained by each concrete subclass of Namespace. (MOF)

**CORBA** Acronym: The Common Object Request Broker Architecture.

**CORBA IDL** Synonym: IDL.

**CWM** Acronym: Common Warehouse Metamodel. The proposed OMG specification for representing and managing warehouse metadata. (CWM)

**data** 1. A representation of information.
2. Items representing facts, text, graphics, images, sound, and video. Data is the raw material of a system supplied by data producers and is used by information consumers to create information. (CWM)

**data analysis tools** Software that provides a logical view of data in a data warehouse. (CWM)

**data element** The most elementary unit of data that can be identified and described in a system. (CWM)

**data management** Controlling, protecting, and facilitating access to data in order to provide information consumers with timely access to the data they need. (CWM)

**data transformation** Creating information from data. This includes decoding operational data and merging of data from multiple operational data sources. (CWM)

**data type** A type whose values have no identity. The data types in a type system are typically into the primitive built-in types, and constructed types such as enumerations and so on.

**DataType** A model element that defines a data type on another element in a MOF metamodel. (MOF)

**data warehouse** An implementation of an informational database used to store sharable data sourced from an operational database. (CWM)

**dependency** 1. A relationship between two entities in which a change to an aspect of one entity affects the other (dependent) entity in some way.
2. A dependency (1) between two modeling elements such that a change to an element changes the meaning of the dependent element. (UML, MOF)

**derived attribute** An pseudo-attribute whose value is not stored explicitly as part of an object, but is calculated from other state when required. Derived attributes can also be updated. (MOF)

**derived association** A pseudo-association whose component links are not stored explicity, but are calculated from other state when queried. Derived associations can also be updated. (MOF)

**derived element** 1. A model element whose value can be computed from another element, but that is shown for clarity or that is included for design purposes even though it adds no semantic information. (UML)
2. An element in a metamodel that is derived from other metamodel elements, and yet is visible in the interfaces produced by an object mapping. See derived attribute, derived association. (MOF)

**design** The phase of the software development process whose primary purpose is to decide how the system will be implemented. During the design phase, strategic and tactical decisions are made to meet the required functional and quality requirements of a system.

**design time** Refers to something that occurs during a design phase of the software development process. Contrast: analysis time.

**development process** A set of partially ordered steps performed for a given purpose during software development, such as constructing models or implementing models.

**diagram** A graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements).

**document element** See root element. (XML)

**Document Type Definition** See DTD (XML)

**domain** An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.

**dynamic typing** A category of type safety that can only be enforced by dynamic type checking. Type systems with dynamic typing are more expressive than those with static typing only. at the cost of run time overheads and potential type errors. Contrast: static typing.

**dynamic type checking** A type checking activity that occurs at run time. Contrast: static type checking.

**DTD** A set of rules governing the element types that are allowed within an XML document and rules specifying the allowed content and attributes of each element type. The DTD also declares all the external entities referenced within the document and the notations that can be used. (XML)

**EBNF** Acronym: Extended Backus-Naur Form. A widely used notation for expressing grammars.

**element** 1. An atomic constituent of a model. Synonym: model element. (MOF, UML)
2. A logical unit of information in a XML document. An XML element consists of a *start tag*, an element content and a matching end tag. (XML)

**element attributes** The name-value pairs that can appear within the *start tag* of an element (2). (XML)

**element content** The elements or text that is contained between the *start tag* and end tag of an element. (XML)

**element type** A particular type of element, such as a paragraph in a document or a class in an XMI encoded metamodel. The element type is indicated by the name that occurs in its start-tag and end-tag. (XML)

**empty string** A string with zero characters.

**end tag** A tag that marks the end of an element, such as </Model>. See *start tag*. (XML)

**entity** 1. A “thing”.
2. An item of interest in a system being modelled.

**enumeration** 1. A type that is defined as a finite list of named values. For example, Color = {Red, Green, Blue}. (UML)
2. A kind of constructed data type in the CORBA type system. (CORBA)

**export** 1. To transmit a description of an object to an external entity. (OMA)
2. In the context of packages, to make an element visible outside of its enclosing namespace. See: *visibility*, import (2). (UML)

**expression** A formula in some language that can be evaluated in some context to give a value. For example, the expression (7 + 5 * 3) evaluates to 22.

**extent** The set of objects that belong to a MOF package instance, class proxy or association instance. (MOF)

**feature** A (meta-)model element that defines part of another (meta-)model element. For example an UML class has attributes and operations as features. (UML, MOF)

**formal language** A language with a specified syntax and meaning.

**formal parameter** Synonym: parameter.

**framework** A micro-architecture that provides an extensible template for applications within a specific domain. (UML)

**frozen** Synonym: immutable. (MOF)

**grammar** A formal specification of the syntax of a language.

**generalizable element** A model element that may participate in a generalization relationship. See: generalization. (UML)

**generalization** A taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element and contains additional information. An instance of the more specific element may be used where the more general element is allowed. See: specialization.

**generic interface** Interfaces that are shared by all MOF metaobjects. See Reflective. Contrast: specific interfaces. (MOF)

**HTML** Acronym: Hyper Text Markup Language. A language for associating visual markup and hyperlinks with textual information that is one of the cornerstones of the World Wide Web. HTML is a particular application of SGML. (W3C)

**identifier** A value that denotes an instance with identity. See: name, object reference.

**identity** "Thingness". A instance has identity if it can be distinguished from other instances irrespective of its component values. For example, objects have identity but numbers do not.

**IDL** 1. Acronym: Interface Definition Language. The OMG language for specifying CORBA object interfaces. (OMA)
2. An interface specification in CORBA IDL (1) - colloquial.

**IDL mapping** 1. A mapping of the design expressed in a model onto CORBA IDL.
2. An IDL mapping (1) defined in the MOF standard that maps a MOF metamodel into CORBA IDL for metaobjects that represent metadata for the metamodel.

**immutable** The property of an entity or value that it will never change. For example, the number 42 is immutable. Synonym: frozen. Contrast: read only. (MOF)

**implementation** 1. An artifact that is the realization of an abstraction in more concrete terms. For example, a class is an implementation of a type, a method is an implementation of an operation. (UML)
2. A realization of a design object in engineering technology; e.g. IDL or program source code.
3. The process of producing an implementation (1)(2).

**implementation inheritance** The use of inheritance to produce one implementation artifact from another implementation artifact. Implementation inheritance presupposes interface inheritance.

**import** 1. To create an object based on a description of an object transmitted from an external entity. See import (1). (OMA)
2. In the context of package, a dependency that shows the packages whose classes may be referenced within a given package (including packages recursively embedded within it). Contrast: export (2). (UML)
3. A relationship between packages in a MOF metamodel that makes the contents of the imported package visible within the importing package. (MOF)

**Import** A model element that in a MOF metamodel that specifies that one package imports another package. (MOF)

**information** 1. The conjunction of data and structure. For example, facts.
2. Data that has been processed in such a way that it can increase the knowledge of the person who receives it. (CWM)

**information consumer** A person or software service that uses data to create information. (CWM)

**information set** A domain-specific extension of OLAP that defines logical structures for raw data collection from mainly human sources (e.g., questionnaire, report form). (CWM)

**inheritance** The mechanism by which more specific elements incorporate structure and behavior of more general elements related by behavior. See generalization. (UML, MOF)

**instance** 1. An instance of a *type* (1) is some value that satisfies the type predicate. (ODP)
2. An object created by instantiating a class. (OMA)
3. An entity to which a set of operations can be applied and which has a state that stores the effects of the operation. (UML)

**instance level** In MOF metamodels and UML models, this label indicates that the labelled feature is common to all instances of its classifier. For example, a classifier level attribute of a class is common to all instances of the class. Contrast: classifier level. (UML, MOF)

**instantiate** The act or process of making an instance of something. See: reify.

**interface** A *type* (1) that describes the externally visible behavior common to a set of objects. An interface includes the signatures of any operations common to all of the objects.

**interface inheritance** The inheritance of the interface of a more specific element. This does not imply inheritance of behavior.

**introspection** A style of programming in which a program is able to examine parts of its own definition. Contrast: reflection (1).

**invariant** A *constraint* on an entity or group of entities that must hold at all times.

**link** A semantic connection between a tuple of objects. An instance of an association. See: association.

**link role** An instance of an association role. See: link, role.

**list** A collection in which the order of the contents is significant, and duplicates are allowed. An ordered collection. See: Set, Array, Unique list.

**knowledge** The conjunction of information with some aspect of understanding.

| | |
|---|---|
| **language** | A means of expression. See *abstract language*, concrete language, natural language. |
| **lumpy cube** | A jagged multidimensional array. A cube whose dimensionality changes dynamically. |
| **markup** | Information that is intermingled with the text of an XML document to indicate its logical and physical structure. (XML) |
| **member** | Synonym: feature. |
| **meta-** | A prefix that denotes a Describes relationship. For example, “metadata” describes “data”. (MOF) |
| **metadata** | 1. Data that describes other data. A constituent of a model. (MOF)<br>2. An inclusive term for metadata (1), meta-metadata and meta-meta-metadata. (XMI)<br>3. Metadata is data about data. Examples of metadata include data element descriptions, data type descriptions, attribute/property descriptions, range/domain descriptions, and process/method descriptions. (CWM) |
| **meta-level** | The level of “meta-”ness of a concept in a metadata framework. |
| **meta-metadata** | Data that describes metadata. A constituent of a metamodel. (MOF) |
| **meta-meta-metadata** | Data that describes meta-metadata. A constituent of a meta-metamodel. (MOF) |
| **meta-metamodel** | A model that defines an *abstract language* for expressing metamodels. The relationship between a meta-metamodel and a metamodel is analogous to the relationship between a metamodel and a model. See: MOF Model, the. (MOF) |
| **metamodel** | A model that defines an *abstract language* for expressing other models. An instance of a meta-metamodel. See: MOF metamodel. (MOF) |
| **metamodel elaboration** | The process of generating a repository type from a published metamodel. Can includes the generation of interfaces and repository implementations for the metamodel being elaborated. (MOF) |
| **metaobject** | 1. An object that represents metadata (2). (MOF)<br>2. Often, a MOF metaobject. (MOF) |
| **metaobject protocol** | A reflection (1) technology in which a program can alter the behavior of the instances of a class by send a message to its metaclass. This style of reflection is not part of the MOF specification. |
| **Meta Object Facility, the** | See: MOF, the. |
| **method** | The implementation of an operation. The algorithm or procedure that effects the results of an operation. (UML) |
| **model** | 1. A semantically closed abstraction of a system. See: *system*. (UML)<br>2. A semantically closed collection of metadata described by a single metamodel. (MOF) |
| **model aspect** | A dimension of modeling that emphasizes particular qualities of the metamodel. For example, the structural model aspect emphasizes the structural qualities of the metamodel. (MOF) |

**model element** Synonym: element. (MOF, UML)

**ModelElement** The abstract superclass of all model elements in a MOF metamodel. (MOF)

**modeling time** Refers to something that occurs during a modeling phase of the software development process. It includes analysis time and design time. Usage note: When discussing object systems it is often important to distinguish between modeling-time and run-time concerns.

**module** A software unit of storage and manipulation. Modules include source code modules, binary code modules, and executable code modules. See: component.

**MODL** Acronym: Meta Object Definition Language. A textual language developed by DSTC that can be used to define MOF metamodels. (MOF)

**MOF, the** 1. Acronym: Meta Object Facility. The OMG adopted standard for representing and managing metadata. (MOF)
2. A metadata service that implements the MOF, the (1) specification. (MOF)

**MOF-based model** Synonym: MOF model.

**MOF-based metamodel** Synonym: MOF metamodel.

**MOF meta-metamodel** Synonym: MOF Model, the.

**MOF metamodel** A metamodel whose meta-metamodel is the MOF Model. (MOF)

**MOF model** A model (2) whose metamodel is a MOF metamodel. (MOF)

**MOF Model, the** The MOF Model is the standard meta-metamodel that is used to describe all MOF metamodels. It is defined in the MOF specification. (MOF)

**multiple inheritance** A kind of inheritance in which a type may have more than one supertype.

**multiplicity** 1. A specification of the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given for roles within associations, parts within composites, repetitions, and other purposes. Essentially a multiplicity is a (possibly infinite) subset of the non-negative integers. (UML)
2. A specification of the allowable cardinalities of the values of an attribute, parameter or association end, along with its uniqueness and orderedness. In the MOF, the allowable cardinalities of a multiplicity must form a contiguous subrange of the non-negative integers. (MOF)

**multi-valued** A ModelElement with multiplicity said to be multi-valued when the ‘upper’ bound of its multiplicity is greater than one. The term does not the number of values held by an attribute, parameter, etc., at any point in time, but rather to the number of values that it can have at one time. Contrast: *single-valued*. (MOF)

**n-ary association** An association involving three or more classes. Each link of the association is an n-tuple of values from the respective classes.

**name** 1. A human readable identifier. See: identifier.
2. The name (1) of a model element. (MOF, UML)

**namespace** — 1. A mapping from names (1) to entities denoted by those names. 2. An element of a metamodel whose primary purpose is to act as a namespace (1) for element names. (MOF)

**Namespace** — The abstract class in the MOF model that is the supertype of those classes that act as namespaces (2). The Namespace class also provides element containment in the MOF Model. (MOF)

**natural language** — A language that has no specification. A language that has evolved for human to human communication; e.g. English, Sanskrit, Amercan Sign Language.

**nested package** — A package that is defined as contained by another package in a MOF metamodel. An instances of a nested package can only exist in the context of an instance of its enclosing package. (MOF)

**node** — 1. A component in a network. A network consists of nodes connected by edges. 2. A run-time physical object that represents a computational resource, generally having at least a memory and often processing capability as well. Run-time objects and components may reside on nodes. (UML)

**notation** — A system of human readable (textual or graphical) symbols and constructs for expressing information.

**note** — A comment attached to an element or a collection of elements. A note has no semantics. (UML)

**object** — An entity with a well-defined boundary and identity that encapsulates state and behavior. State is represented by attributes and relationships, behavior is represented by operations and methods. An object is an instance of a class. (MOF, UML)

**object reference** — An identifier for an object, typcally a CORBA object. (OMA)

**OCL** — Acronym: Object Constraint Language. A pure expression language that is a non-normative part of the UML specification (ad/99-06-08) that is designed for expressing constraints. (UML)

**OLAP** — On-Line Analytical Processing. OLAP uses a multidimensional view of aggregate data to provide quick access to strategic information for further analysis. OLAP and data warehouses are complementary. A data warehouse stores and manages data. OLAP transforms this data into strategic information. (CWM)

**operation** — A service that can be requested from an object to effect behavior. An operation has a signature, which may restrict the *actual parameter*s that are possible. (MOF, UML)

**operation database** — The operational database contains detailed data used to run the day-to-day operations of a business. It is the source of data for the data warehouse. (CWM)

**ordered collection** — A collection that is ordered. See ordering. (MOF)

**ordering** — A property of collections. A collection is ordered if the sequence in which the elements appear needs to be preserved. (MOF)

**package** — A mechanism for organizing the elements of a model or metamodel into groups. Packages may be nested within other packages. (MOF, UML)

**Package** — The class in the MOF Model that describes a package in a metamodel. (MOF)

**package cluster** — A package that groups together a number of packages so that a set of instances of those packages can form a single extent. A package composition mechanism. (MOF 1.x)

**package consolidation** — Synonym: package cluster. (MOF 1.x)

**package importing** — See: import (3). A package composition mechanism. (MOF)

**package inheritance** — A generalization relationship between packages. Analogous to interface interface inheritance for classes. A package composition mechanism. (MOF)

**package nesting** — Defining one package inside another. A package composition mechanism. See: nested package. (MOF)

**parameter** — 1. A place holder for a value that can be changed, passed or returned by a computation. A parameter typically consists of a parameter name, a type and attributes that specify the information passing semantics for actual parameters. Synonym: *formal parameter*. Contrast: *actual parameter*, argument.
2. A parameter (1) of an operation or exception. (CORBA, MOF)
3. A parameter (1) of an operation, message or event. (UML)

**postcondition** — An *constraint* that must be true at the completion of a computation.

**precondition** — An *constraint* that must be true at the start of a computation.

**primitive type** — A type from which other types may be constructed, but that is not constructed from other types. See type system.

**product** — The artifacts of development, such as models, code, documentation, work plans. (UML)

**profile** — A simplified subset of a language or a metamodel.

**projection** — 1. A primitive operation in relational algebra which produces a relation by "slicing" one or more columns from another relation.
2. The set of MOF class instances that is visible via the reference operations of a class instance. For a class X, a n-ary association $A(X,Y_1, ... Y_{n-1})$ and an instance $x \in X$ then the expression

$$\text{PROJECT } [Y_1, ... Y_{n-1}] \text{ (SELECT A WHERE X = x)}$$

defines the set of links. In the binary case, the set is a set of instances. (MOF)
3. A mapping from a set to a subset. (UML)

**property** — 1. A characteristic of an entity.
2. A property (1) that is represented as a mapping from an entity and a property name to a value for the property. See tagged value. (UML)

**pseudo-code** — An informal description of an algorithm in a language whose meaning is not fully defined.

**published (meta-)model** — A (meta-)model which has been frozen, and made available for use. For example, a published metamodel can be used to instantiate repositories and can be safely reused in other metamodels.

**quokka** — A small scrub-wallaby found on Rottnest Island, Western Australia.

**read only** — Describes an object or attribute for which no explicit update operations are provided. (MOF)

**recursive** — See recursive.

**reference** — 1. An identifier.
2. A use of a model element. (UML, MOF)
3. A feature of a class that allows a client to navigate from one instance to another via association links. See projection (2). (MOF)

**Reference** — A model element that defines a *reference* in a MOF metamodel. (MOF)

**reflection** — 1. A style of programming in which a program is able to alter its own execution model. A reflective program can create new classes and modify existing ones in its own execution. Examples of reflection technology are metaobject protocols and callable compilers.
2. In the MOF, reflection characterizes what happens when a client examines and updates metadata without compile time knowledge of its metamodel. (MOF)

**reflective** — Describes something that uses or supports reflection.

**reflective interfaces** — Synonym: generic interface. (MOF)

**Reflective** — The name of the CORBA IDL module containing the MOF's reflective interfaces. (MOF)

**reify** — To produce an object representation of some information.

**relation** — A collection of relationships (1) with the same roles. A relation is typically pictured as a two dimensional table with the rows representing relationship tuples, and the columns representing the roles and their values.

**relationship** — 1. A semantic connection between 2 or more entities where each entity fills a distinct role. A relationship is typically expressed as a tuple.
2. Colloquially, a relation.
3. A relationship (1) between elements of a model. Examples include associations and generalizations (MOF, UML).

**repository** — 1. A logical container for metadata. (MOF)
2. A distributed service that implements a repository (1). (MOF)

**requirement** — A desired feature, property (1), or behavior of a system.

**responsibility** — A contract or obligation of a type or class. (UML)

**reuse** — The act or process of taking a concept or artifact defined in one context and using it again in another context.

**role** 1. A position in a relationship or column in a relation.
2. The named specific behavior of an entity participating in a particular context. A role may be static (e.g., an association role) or dynamic (e.g., a collaboration role). (UML)

**root element** The single outermost element in an XML Document. Synonym: document element. (XML)

**run time** The period of time during which a computer program executes.

**scope** 1. A region of a specification in which a given identifier or entity may be used.
2. An attribute of some features in the UML metamodel and MOF Model that determines if the feature is instance level or classifier level. (MOF, UML)

**sequence** 1. A CORBA constructed data type. (CORBA)
2. A collection whose data type does not specify ordering or uniqueness semantics. Differs from an array in that the number of elements is not fixed. (MOF)

**set** An unordered collection in which a given entity may appear at most once.

**SGML** Acronym: Standard Generalized Markup Language. An International Standard (ISO 8879:1986) that describes a generalized markup scheme for representing the logical structure of documents in a system-independent and platform independent manner.

**signature** The name and parameters of an operation. Parameters may include an optional returned parameter. (MOF)

**single inheritance** A form of generalization in which a type may have only one supertype.

**single-valued** A ModelElement with a multiplicity is called single-valued when its upper bound is equal to one. The term single-valued does not pertain to the number of values held by the corresponding feature of an instance at any point in time. For example, a single-valued attribute, with a multiplicity lower bound of zero may have no value. Contrast: multi-valued.

**specialization** The reverse of a generalization relationship.

**specific interfaces** An interface for metadata described by a given metamodel that is tailored to the abstract syntax of that metamodel. Contrast: generic interface.

**specification** A precise description that can or should be used to produce things.

**Standard Generalized Markup Language** See: SGML

**start tag** A tag that marks the beginning of an element, such as <Model>. Also see end-tag. (XMI)

**state** The state of an object is the group of values that constitute its properties at a given point in time.

**static** In C++ or Java, a static attribute or a static member function is shared by all instances of a class. Synonym: classifier level.

**static type checking** Contrast: dynamic type checking.

**static typing** Contrast: *dynamic typing*.

**strong typing** — A characteristic of a computational system that type failures are guaranteed not to occur.

**stereotype** — A new type of modeling element that extends the semantics of the metamodel. Stereotypes must be based on certain existing types or classes in the metamodel. Stereotypes may extend the semantics, but not the structure of pre-existing types and classes. Certain stereotypes are predefined in the UML, others may be user defined. Stereotypes are one of three extendibility mechanisms in UML.

**string** — A sequence of text characters. The details of string representation depends on implementation, and may include character sets that support international characters and graphics.

**subclass** — In a generalization relationship the specialization of another class, the superclass. See: generalization.

**subtype** — In a generalization relationship the specialization of another type, the supertype. See: generalization.

**subsystem** — A part of a system that it is meaningful to describe in isolation.

**superclass** — In a generalization relationship the generalization of another class, the subclass. See: generalization.

**supertype** — In a generalization relationship the generalization of another type, the subtype. See: generalization.

**supplier** — A type, class or component that provides services that can be invoked by others.

**system** — A collection of connected units that are organized to accomplish a specific purpose. A system can be described by one or more models, possibly from different viewpoints. (UML)

**tagged value** — A representation of a property as a name-value pair. In a tagged value, the name is referred as the tag. Certain tags are predefined; others may be user defined. (UML, MOF)

**technical metadata** — Technical metadata, such as transformation mappings, is used to build and maintain the data warehouse processes. It describes the data used by various tools to store, manipulate, or move warehouse data. (CWM)

**technology mapping** — A mapping that transforms a design expressed as a model or metamodel into implementation artifacts; e.g. CORBA IDL or program source code.

**top-level package** — A package that is not nested in another package. (MOF)

**transitive closure** — 1. The transitive closure of the value $v_0$ in V under the mapping $m : V \rightarrow V$ is defined by the following equation:

$$TC(v_0, m) \cong \{ v \in V : (v = v_0) \vee (\exists v_i \in TC(v_0, m) : m(v_i) = v) \}$$

In other words, the set of all V's that are "reachable" from $v_0$ via the mapping. (Math)
2. The transitive closure of an initial object under an association is the set of objects reachable from the initial object via extant links in the association. (MOF, XMI)

**type** 1. A predicate characterizing a collection of entities. (RM-ODP)
2. A predicate defined over values that can be used to restrict a possible parameter or characterize a possible result. Synonym: *type* (1). (OMA)
3. A stereotype of class that is used to specify a domain of instances (objects) together with the operations applicable to the objects. A *type* (3) may not contain methods. (UML)

**type checking** A process that checks for programs or executions that could lead to type failure.

**TypeCode** A CORBA primitive data type. The TypeCode type is used in CORBA to pass runtime descriptions of CORBA types. A CORBA any value contains a TypeCode to describe the embedded value's type. See any. (CORBA)

**type error** An event that is triggered when type checking detects a situation which could lead to type failure.

**type expression** An expression that evaluates to a reference to one or more types. (UML)

**type failure** A type failure occurs when a computation erroneously uses a value thinking it has one type when it has a different (incompatible) type. The consequences of a type failure are often completely unpredictable.

**type loophole** A construct or artifice that allows a program to breach type safety.

**type safety** A desirable property of a program or computation that type failures are guaranteed not occur.

**type system** A language for expressing *type*s (1). A type system is typically defined from a small set of primitive type and type constructors. See metamodel.

**typing** Synonym: type checking.

**unique list** An ordered collection in which no entity may not appear more than once as a collection member; i.e. a list in which duplicate elements are not allowed. (MOF)

**uniqueness** A property of collection types that determines whether a given element may appear more than once in the collection. (MOF)

**unordered collection** A collection in which the order in which the collection members appear has no significance. See ordering. (MOF)

**UML, the** Acronym: The Unified Modeling Language. (UML)

**UUID** Acronym: Universally Unique IDentifier. An identifier that guaranteed to be unique across all computer systems and across time, provided certain assumptions hold.

**valid XML document** An XML Document that conforms to its DTD. (XML)

**value** 1. An element of a type domain. (UML)
2. An entity that can be a possible actual parameter in a request. (OMA)

**view** A projection (3) of a model, which is seen from a given perspective or vantage point and omits entities that are not relevant to this perspective. (UML)

**visibility** An *attribute* of a model element whose value (public, protected, private, or implementation) determines the extent to which the model element may be seen, and hence used, outside of the namespace in which it is defined.

**W3C, the** Acronym: the World Wide Web Consortium. The standards body that takes the lead in developing standards related to the Web; e.g. HTML, HTTP and XML. (XML)

**well-formed XML document** An XML document that consists of a single element containing properly nested subelements. All entity references within the document must refer to entities that have been declared in the DTD, or be one of a small set of default entities. (XML)

**XLink** An XML construct for representing links to external documents. See Xpointer. (XML)

**XMI** Acronym: XML-based Metadata Interchange. The adopted OMG standard for a metadata interchange format that is based on the W3C's XML specification. (XMI)

**XML** Acronym: Extensible Markup Language. A profile of SGML. XML is the W3C standard for representing structured information; e.g. web metadata. (XML)

**XML Declaration** A processing instruction at the start of an XML document, which asserts that the document is an XML Document. (XML)

**XML Document** An XML document consists of an optional XML Declaration, followed by an optional DTD, followed by a document element. (XML)

**XPointer** An XML construct for linking to an element, range of elements, or text region within the same XML document. (XML-Link 6)