Meta Data Coalition

Open Information Model

Version 1.1 (Proposal)

August, 1999

Table of Contents

1 Overview

The Meta Data Coalition (MDC) Open Information Model (OIM) is a vendor-neutral and technology-independent specification of core meta data types found in the operational and data warehousing environment of enterprises. This section presents the motivation for meta data management and the creation of the OIM.

1.1 What is Meta Data?

Meta data is descriptive information about the structure and meaning of data and of the applications and processes that manipulate data. Meta data can be grouped into two categories: technical and business meta data.

Technical meta data supports designers, developers, and administrators during development, maintenance, and management of an information technology environment. It is the technical glue that links the tools, applications, and systems that together constitute a solution. For example, technical meta data can address database structures, installed applications, server systems, and so forth.

Business meta data, on the other hand, makes the services of the enterprise environment more understandable to end-users. For example, it provides explanations of the business objects and processes to ease browsing, navigation, and querying of data.

1.2 Situation Analysis

Corporate globalization and internationalization in a rapidly changing and increasingly competitive business environment requires that companies leverage their information assets in new and more efficient ways. Enterprise data, once viewed as merely operational or tactical in nature, is now being used for strategic decision-making at every enterprise business level.

Managing the strategic information assets and providing timely, accurate, and global access to enterprise data in a secure, manageable, and cost efficient environment is becoming increasingly critical. Competitiveness forces companies to integrate turnkey solutions to achieve the tactical advantages of lower cost and reduced implementation time. The strategic advantages of online access to all knowledge maintained in the distributed computing environment of an enterprise requires blurring the lines between OLTP (Online Transaction Processing), Data Warehousing, and the Web.

Meta data, or information about data, has become the critical enabler for the integrated management of the information assets of an enterprise. The proliferation of data manipulation and management tools throughout an enterprise has resulted in a host of incompatible information technology products, each of which processes meta data differently.

End-users suffer from inaccessible and incompatible meta data locked into individual tools. Meta data has become the number one integration problem in the area of enterprise information and data warehouse management for the following reasons:

- No single tool covers all information processing requirements of a multi-task business or development process. Users are forced to change tools, and of course, they want to reuse the meta data they have already entered.
- Not all components of an integrated tool set may provide the required functionality or performance. Better tools may be available. Mixing and matching of best-of-breed tools is important, or even crucial, for the success of today's corporations. This mixing and matching requires meta data integration of tools from different vendors.
- Organizations may wish or may be required to track meta data for their OLTP or data warehousing environment to make it auditable. This requires the extraction of meta data from individual databases, applications, and tools, and archiving the combined state as a configuration.

- Large enterprises spread over many countries and grouped into individual sub-companies will not be able to agree on a single set of database systems, applications, and tools. Meta data integration is the only solution that provides an overview of the global information inventory in such a situation.
- Today's enterprises are faced with the difficult decision of either undertaking the expensive task of meta data integration by themselves, or missing out on the benefit of sharable and reusable meta data.

1.3 Challenge

Enterprise-wide information management requires global and efficient access to shared meta data by all the heterogeneous products found in today's information technology environment. To use tools efficiently, users need to be able to move meta data between tools or between tools and a repository. In addition, tools are often provided by different vendors and run on different hardware and software platforms.

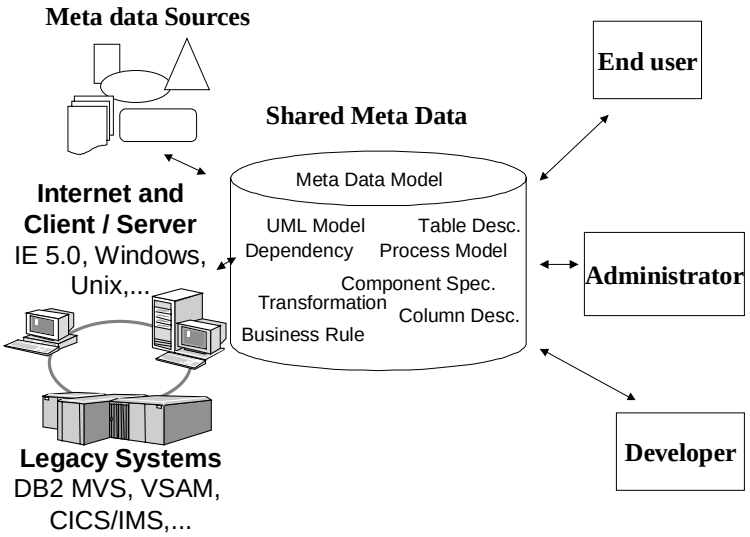


Figure 1: Shared Meta Data Environment

To integrate the different tools and repositories in an enterprise environment, they all must share meta data in the same way. In addition to global access, this requires that meta data is stored, managed, and interpreted in a consistent way by all participants. A successful framework to integrate tools from different vendors through shared meta data must satisfy the following requirements:

- Meta data integration requires a common specification that defines the structure and semantics of meta data in a formal and consistent way to which the different vendors' products can comply. For the industry to agree on this specification, it must be technology-independent and vendor-neutral.
- The common specification must be published in a standard language, so it can be understood and manipulated by humans as well as tools. Automatic translations and mappings are necessary to support multiple implementations as well as new and emerging technologies.
- The common specification has to be abstract and published in a format so that vendors can implement it on many platforms and by many technologies. Vendors are therefore free to innovate in the area of implementation technology and are not forced to use specific component models, APIs, database technologies, or platforms.
- Multi-vendor implementations of the common specification require an interchange mechanism for moving meta data between the resulting heterogeneous products. The interchange enables end-users to integrate conforming products in a plug-and-play fashion.

- Browsing, querying, and reporting on meta data described by the common specification requires that tools and repositories expose a schema. End-users and developers therefore can rely on a consistent queriable view of their meta data even in different implementations.

The MDC Open Information Model is a specification of a core set of meta data types such as database schema, business process, or business object elements. The following figure shows how the MDC OIM fits into the different levels of data modeling and abstraction:

Unified Modeling Language	Class, Attribute, Method, Association, Generalization
Open Information Model	Table, Column, Business Process, Business Object, Dictionary, Term, Synonym
Meta Data	Customer Table, Order Entry Process, Expense Report, Business Object, Cost Center Definitions
Data	Vulcan Coffee, Report 12/99, Cost Center 10747

The UML is the most abstract description of information structures by classes arranged into a generalization / specialization hierarchy. The Open Information Model is a specialization of the abstract concepts of UML into domain specific types that describe meta data. It represents an industry agreement on a detailed semantics of types such as a table definition. The instances of the Open Information Model represent the descriptive information about enterprise data such as actual SQL Schema, OLAP Schema, or business process definitions.

The meta data types – along with their attendant native interchange format and relational query schema – form a comprehensive, easy-to-use, and standards-based solution for the integration of meta data in an enterprise environment, including the extension and customization of the meta data model itself. The use of standard definitions enables linking of heterogeneous implementations. The following standard technologies are used to provide implementations of the MDC OIM:

- The Unified Modeling Language (UML) as the formal specification language for OIM,
- The eXtensible Markup Language (XML) as the interchange format for OIM, and
- The Structured Query Language (SQL) as the query language for OIM.

1.4 The Meta Data Coalition

The Meta Data Coalition (MDC), founded in 1995, is a not-for-profit consortium of vendors and end-users whose goal is to provide a vendor-neutral and technology-independent specification of enterprise meta data.

The Meta Data Coalition brings industry vendors and users together to address a variety of problems and issues regarding access, sharing, and management of meta data. This is a voluntary coalition of interested parties with a common focus and shared goals, not a traditional standards body or regulatory group.

The Meta Data Coalition members agreed upon goals, including:

- Creating a vendor-independent, industry-defined and industry-maintained specification for meta data;
- Enabling users to control and manage the access and interchange of meta data in their unique environments through the use of specification-compliant tools;
- Allowing users to build tool configurations that meet their needs and to incrementally adjust those configurations as necessary to add or subtract tools without impact on the environment;

- Defining a clean, simple interchange implementation framework that will facilitate compliance and speed adoption by minimizing the amount of modification required to existing tools to achieve and maintain compliance;
- Creating a process and procedure not only for establishing and maintaining a meta data standard but for extending and updating it over time as required by evolving industry and user needs; and
- Using or aligning with existing and accepted standard technologies or standards efforts wherever possible.

A non-goal of the MDC is to develop a specification for specific repository implementations, component technologies, or database systems. Furthermore, the scope of the specification is focused on a core set of generic meta data types independent of individual tools or applications. The limits on the scope of the MDC meta data specification are introduced to make it possible to reach a wide consensus and avoid that individual vendor solutions become the only "correct" way.

The Meta Data Coalition maintains both a Web Page site and an e-mail address to allow members or potential members to communicate electronically. The current Web address is:

http://www.MDCinfo.com/

which is available through the World Wide Web. The Council also maintains the e-mail address:

coalition@evtech.com

which includes the e-mail addresses of coalition members, and

mdc-spec@evtech.com

for sending comments regarding technical proposals.

1.5 Development History

The development history of the OIM includes initial designs based on existing standards, collaborations with dozen of vendors, broad reviews by hundreds of vendors, and widely distributed beta releases. The development of the original version of OIM, driven by Microsoft before transferring OIM to the MDC, was marked by the following milestones:

- **October 1996: First OIM Design Preview**
  Microsoft and Texas Instruments (TI) unveiled their Repository design, which included a draft information model, developed by Microsoft, TI, SELECT Software Tools and Rational Software. This initial version of OIM comprised interfaces enabling the development of interoperable tools for component-based development and reuse. The technical review and demonstrations were conducted as part of Microsoft's Open Process, with the participation of over 50 core members of the software development community, including vendors of development tools, design and modeling tools, enterprise applications, and document management and version control systems.
- **January 1997: UML Model Interchange Initiative**
  Twenty-one leading enterprise modeling vendors jointly announced with Microsoft the development of the Unified Modeling Language (UML) subject area of the OIM. This model enables teams of corporate developers to easily share models developed with different modeling tools, enabling high-quality component-based application development and reuse. Vendors involved in the first round of UML model development included Microsoft, Logic Works (now a division of PLATINUM technology), Popkin Software and Systems, Rational Software, SELECT Software Tools and Texas Instruments (TI). Via the Open Process, the information model review was then expanded to include essentially all vendors in the software modeling industry. These vendors were invited to a design preview held on January 31, where the model was presented and the first-round vendors demonstrated full cross-tool interoperability with each other's tools. The final version of the model was shipped two months later in Visual Basic 5.0 and Visual Studio 97.

- **July 1997: Open Information Model Design Preview**
  Over 60 software vendors attended a design preview of a greatly expanded OIM, which provided a common way for development tools to work together across the software development life cycle. This event gave vendors an opportunity to offer input on the design specifications. Over 30 of these vendors had development projects underway that utilize the model. The addition of a new database schema model enabled development tools to automatically target multiple databases without rewriting application code. Based on design input from Business Objects, Cognos, Informatica, Logic Works, PLATINUM technology, Popkin Software and Systems, Powersoft, Prism Solutions and Sterling Software, the new database schema model enabled easy sharing of schema information between multiple vendors' data warehousing and database design tools. At this preview, Microsoft and PLATINUM technology announced a partnership to make the OIM available on non-Microsoft operating systems and database systems, and to draw on PLATINUM's information model expertise in future extensions to OIM.

- **October 1997: Meta Data Coalition Endorsement**
  The MDC was an active reviewer of the OIM and the DBM subject area, proposing extensions to the DBM model to fit the needs of the MDC. The MDC also launched its Metadata Interchange Specification (MDIS) to OIM translator freeware, which reads mapping information from an MDIS file to the DBM model, and vice-versa. This first release emphasized the relational database model, and the transformations and business rules from MDIS that have an explicit representation in the DBM model.

- **December 1997: Data Warehousing Extension Web Review**
  An open design review process was initiated for gathering industry feedback on new data warehousing extensions to the OIM, whose goal was to enable data warehousing products from different vendors to share information. The initial partners in this effort were Apertus Carleton, Business Objects, Cayenne Software, Cognos, Evolutionary Technologies International, Informatica, Logic Works, Microsoft, PLATINUM technology, Popkin Software & Systems, Powersoft, Prism Solutions, and Sterling Software. This open design review period began with the availability of preliminary specifications for data transformation services and online analytical processing (OLAP) extensions to the Open Information Model. More than 550 vendors reviewed the specifications.

- **April 1998: Data Warehousing Workshop**
  The Data Warehousing Workshop brought together more than 200 leading developers and users of data warehousing, software development and data transformation tools to review and shape an open standard for a repository-based data warehousing infrastructure. The event focused on the expanding role of the OIM as a common infrastructure for data warehousing products and software development tools. Evolutionary Technologies International, DWSoft Technology, Logic Works, PLATINUM technology, Sagent Technology and TopTier Software demonstrated early product implementations based on the new extensions. The model shipped in SQL Server 7.0 and Visual Studio 6.0.

- **December 1998 - Technology-Independent OIM Moves to the MDC**
  Microsoft announced its membership in the MDC and the transfer of control of OIM to that consortium. The OIM will be made technology neutral, and in particular, be made independent of Microsoft Repository. MDC will maintain and evolve the OIM as a technology-independent and vendor-neutral meta data standard. Microsoft also announced the availability of the XML Interchange Format, by which meta data can be moved between any two repository products. New submodels of OIM were also announced: a model for Semantic Information, which accommodates meta data about linguistic processing tools that interpret relational databases, and a model for Record-Oriented databases, developed in cooperation with PLATINUM technology, which accommodates meta data from record-oriented legacy systems.

- **July 1999 – MDC Accepts the OIM as a Standard, Work Underway on Extensions**
  The technical work by the MDC concluded with a vote by the membership on July 15th to adopt the OIM 1.0 specification as a standard. In addition, Microsoft and the MDC announced an open design review for proposed extensions to the OIM. The model extensions capture business

modeling information such as business goals, objectives, processes, and rules as well as business terminology and categorizations. This information can be used to enhance the usability and effectiveness of tools and applications such as enterprise business information portals. These extensions were developed in cooperation with CA/PLATINUM technology, KPMG, LEXIS-NEXIS, News Edge, ICL, DWSoft, AppsCo, Deloitte & Touche Consulting, IntelliCorp, Micrografx, VISIO, Longs Drug Stores, Rule Machines, and the members of the MDC Technical Committee.

1.6 Acknowledgments

The development of the OIM involved the collaboration of many vendors. We especially thank Microsoft Corporation for their effort in driving the specification of the initial version of the OIM and for their agreement to pass control of this work to an independent standards organization, namely MDC. Microsoft's primary partners in the development of the OIM were PLATINUM technology and Sterling Software. Other key contributors include AppsCo, Business Objects, Cognos, DWSoft Technology, Evolutionary Technologies International (ETI), Informatica Corporation, Informix Inc., Intellicorp, Visible Systems Corporation, Logic Works (now a division of PLATINUM technology), Popkin Software and Systems, Powersoft (now a division of Sybase), Prism Solutions, Rational Software, Sagent Technology, SELECT Software Tools, and Visio Corporation. The specific contributions of particular vendors are summarized in section 1.5.

2 Introduction

The purpose of the Meta Data Coalition Open Information Model (MDC OIM) is to support tool interoperability across technologies and companies via a shared information model. The OIM is designed to encompass all phases of information systems development, from analysis through deployment. Computing technologies as diverse as CASE, component, application, Intranet, database, and data warehousing are supported.

2.1 Goals and Scope

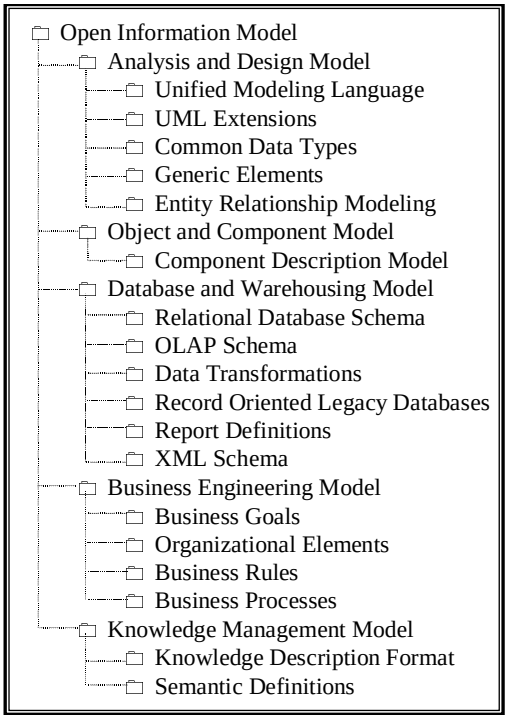
The goals of the MDC OIM are to:

- Be an easy-to-use, expressive, and extensible model of core meta data types.
- Provide mechanisms to specialize and extend the core meta data types rather than to modify or replace core concepts.
- Implement a set of fundamental concepts that are generic and generally applicable, and to reuse these concepts through refinement without repeatedly redefining the fundamental concepts.
- Allow adding new concepts to the core in a consistent way.
- Allow specialization of concepts for particular domains.
- Provide a technology-independent and vendor-neutral specification.
- Support heterogeneous implementation using different component technologies, programming languages, and other technologies.
- Be scalable from individual tools to enterprise-wide meta data repositories.
- Be widely acceptable (general purpose and expressive) and usable (simple and evolutionary). This includes the use of or alignment with existing standards.
- Integrate best engineering practice in the area of meta data management and meta data specification.

The MDC OIM is not a specification of a repository API or implementation. The primary goal of the model is to provide a formal description of meta data types to support sharing of meta data between tools and repositories. This includes interoperability between repositories. However, the MDC OIM focuses on the description of the information, not on data access and management.

2.2 Overview and Package Structure

The meta data types specified by the MDC Open Information Model are structured into domain-specific submodels. The following figure shows the high-level structure of the MDC Open Information Model.



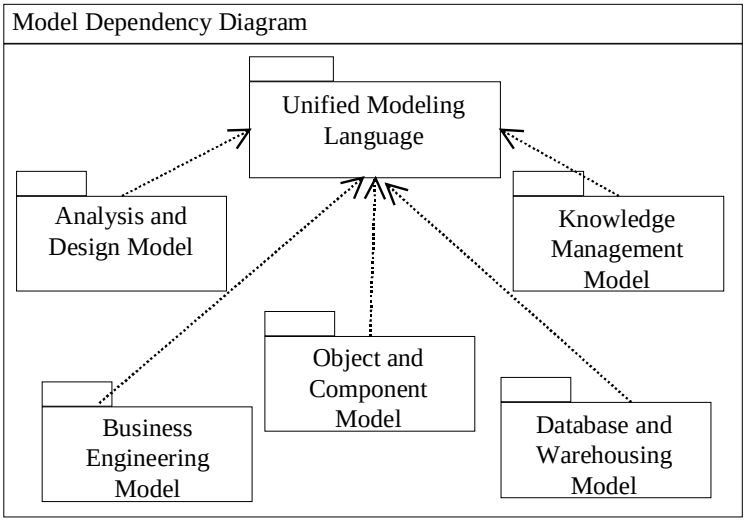
The submodels address the most important areas of information, data warehouse, and knowledge management in the integrated meta data environment found in an enterprise. The submodels provide a general set of meta data types that enable generic access and interchange. Each of the submodels is described in detail in this document.

In addition to its specific use as core of the Analysis and Design Model, UML's breadth and high level of abstraction make it an excellent base model from which other MDC Open Information Model submodels can inherit.[1] UML covers such concepts as type, class, component, package, diagram, method, operation, relationship, attribute, and constraint — concepts that are relevant to virtually all domains described by the MDC Open Information Model. The following figure shows the dependencies between the submodels of the MDC Open Information Model.

[1] The MDC OIM uses the UML in three different roles:

- The UML as modeling language and, as such, as a standard to design and customize the MDC OIM. Focus with this usage of the language is on the UML notation.
- The UML as main part of the Analysis and Design Model subject-area of the MDC OIM. In this role, the UML supplies the meta data types to express object-oriented models. Focus with this usage of the language is on the UML meta model.
- The UML as core model of the MDC OIM from which other submodels inherit concepts. This usage of the language has the goal to minimize the complexity of the MDC OIM by re-using and therefore reducing the number of modeling concepts. Focus with this usage of the language is on the UML meta model.

Confusion may arise because of the three different roles. To avoid this we have separated them as much as possible in this document. Nevertheless, the use of the UML in the MDC OIM, to make it self-describing, as model for the analysis and design domain, and as core model are all important concepts to make the model easier-to-use, standards-based, and expressive.



Each of the dependent models inherits and refines concepts out of the UML. The use of UML as the root of the OIM is discussed in the UML reference section.

2.3 Extensibility Model

The MDC Open Information Model offers a set of extensibility features to accommodate specific tool implementations and to enable vendors to add value with their products. Extensibility is a core feature designed into the model. Individual vendors are able to enhance the core model with tool specific types independent of the MDC or other vendors. Such a flexible and powerful extension mechanism is not only necessary for the feature differentiation of multi-vendor implementations, but is also a prerequisite for the evolution of the core model itself.

The MDC OIM extensibility is based on:

- **Stereotypes** – are used to classify existing modeling elements, thereby introducing new types of modeling elements.
- **Tagged Values** – are name / value pairs which can be used to extend the state of an object without modifying its structural definition.
- **Type Extension** – modifies an existing type by adding one or more inheritance relationships to new types. The modification adds a feature but leaves the existing structural definition of the existing type untouched.
- **Type Reuse** – enables defining a new type through inheritance from an existing type and adding new features. The new type conforms to the existing type, and can replace it, but offers new features to clients, which know of these and are able to use them.

2.4 Scenarios

Meta data exists in every tool and application in an enterprise. The MDC Open Information Model must therefore have a very broad scope. The following lists several general areas in which meta data management and interchange is an important part.

- *Business Process Reengineering* - includes the development of models of an enterprise to document, analyze, and simulate the operational environment and its control and product flow. Models and their complex cross relationships, as well as the links to the outside world, need to be stored, managed, and interchanged between modeling and back-end tools, such as ERP (Enterprise Resource Planning) systems.

- *Application Development* - is one of the main areas in which meta data has been used to integrate tools. Software development is a multi-step process, which often involves several different tools. Integrating these tools can be challenging, especially if they are from different vendors. Meta data models implemented by repositories are the base technology to accomplish this task.
- *Data Modeling and Design* - is a multi-developer task with such problems as global access, multi-version synchronization, and history tracking. A common meta data model enables the integration of network-based tools and the support of team development with a centralized repository. An important part of this architecture is the queryability of the meta data to implement such features as data dictionaries and meta data reporting.
- *Packaged Applications* - require customization to address the needs of individual enterprises. Customization affects not only the application itself, but also the software environment around it. A common meta data model allows third-party vendor applications to interchange information with the packaged application. This ensures a controlled deployment of the application and its modifications, so that customizations can be re-applied to software revisions, and impact analysis can validate changes before they are applied.
- *Data Warehousing* - with its hard-to-solve integration, consistency, lineage, and usability problems, has made meta data management mission critical. Integrated meta data management is a necessity not only to achieve tool integration, but also to aid the end-user with definitions and explanations. The integration of technical and business meta data in a common meta data model makes the tasks of designing, managing, and using a data warehouse easier to do.

2.5 Meta Data Specification with UML

The Unified Modeling Language (UML) is the modeling standard for specifying and representing meta data types for the MDC OIM. Based on the formal representation of the meta data specification in the UML, it is possible to generate automatically all the necessary deliverables to deploy implementations of the specifications in tools or repositories.

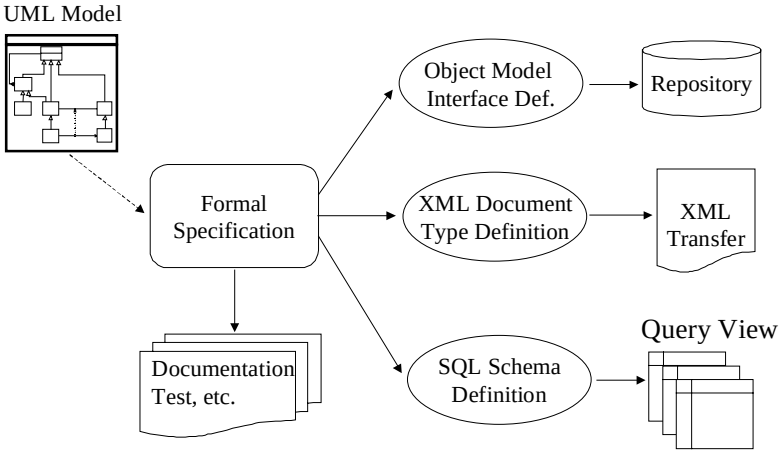


Figure 2: Deliverable Generation from the UML

2.5.1 The Unified Modeling Language (UML) Standard

The UML is a language for modeling information systems and software artifacts. The UML can be used to visualize, specify, construct, and document knowledge about software-intensive systems and their purposes at an abstract level.

The goals of the UML are to unify the most prominent modeling methodologies into a ready-to-use expressive modeling language that is simple and extensible. The modeling language was developed by

Rational Software Corporation and its partners, and was adopted by the Object Management Group (OMG) in November 1997. The UML continues to evolve though this standard body.

Industry organizations and the leading modeling vendors have embraced the UML. Numerous products and services have been announced or introduced into the market since its standardization as UML 1.0. As such, the language enables projects to focus on the modeling task at hand rather than to select or invent a consistent, accepted, and tool-supported representation language.

The UML consists of a notation and a semantic description. The notation defines the visual representation of diagrams and modeling elements. The semantic description, or meta model, is the formal specification of the semantics of the notation. The UML is defined in itself, which means that a subset of the language notation and semantics is used to specify the complete language. It is therefore more than just a language; it provides a conceptual framework for modeling software artifacts beyond the current scope of the UML.

In summary, the UML provides:

- A conceptual framework for modeling software artifacts.
- A modeling language that consists of the UML notation and the UML meta model that defines the semantics of the notation.
- Sufficient notation and semantics to address object-oriented analysis and design.
- Extensibility mechanisms for the addition, variant interpretation, and specialization of concepts.
- A widely accepted standard with industry support and existing products and services.

2.5.2 Modeling Concepts

The following figure shows an example of how UML modeling concepts are used to specify meta data types in the framework.

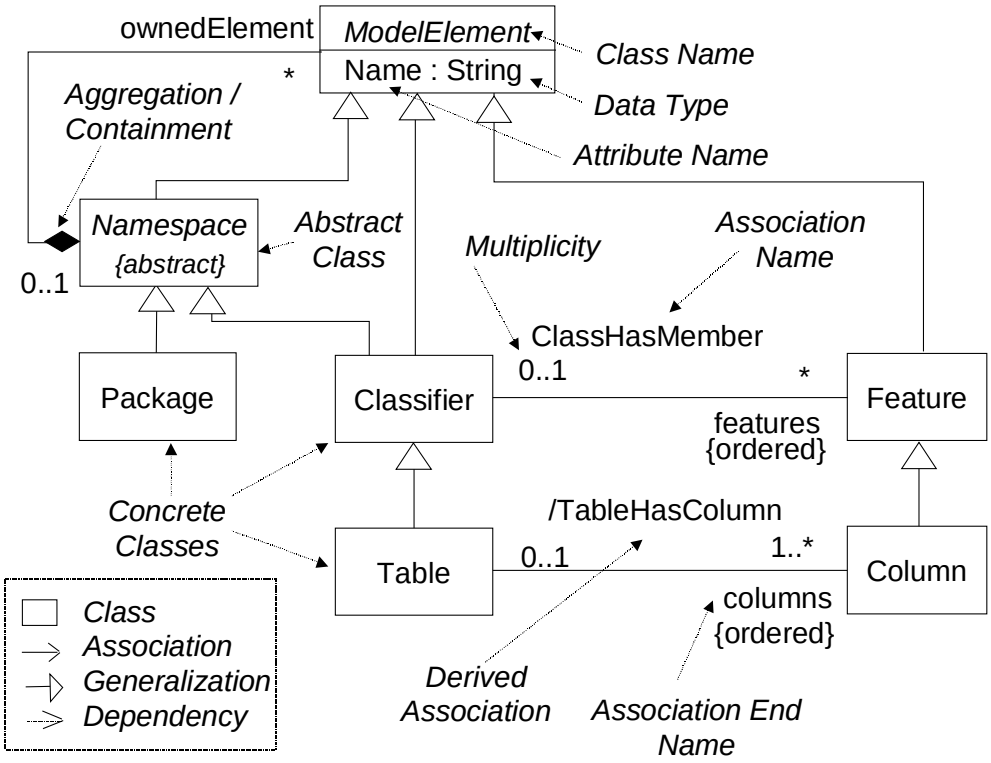


Figure 3: UML Modeling Framework

The example shown is a formal specification of a simplified database schema representation. A package may contain tables, and a table may have columns. All elements can be named.

The following lists the main meta data modeling concepts of the UML that are used to express the MDC Open Information Model.

- **Packages** – are used to group model elements together, such as classes and associations. Packages themselves may be nested into other packages. Each element is directly owned by a single package and packages therefore form a strict hierarchy.
- **Diagrams** – diagrams are graphical representations of a set of model elements that render views of a model from a certain perspective. Diagrams are used to provide a graphical structuring of a model so that it can be better understood by the modeler.
- **Classes** – are specifications of a set of objects that have common structural and behavioral features. Classes are used to model entities with common characteristics and semantics. Classes may be Abstract, and therefore incompletely specified, or Concrete, in which case they are complete and, in an implementation, would be instantiable.
- **Attributes** – are members of classes and describe structural features of entities. Attributes are used to model information associated with entities. They can have a data type and an initial value, usually denoted by: *attribute-name : data type = initial-value*. Attributes may be derived from other attributes in the model.
- **Data Types** – are instances that define data values and are used to model simple values that have no identity.
- **Associations** – are descriptions of relationships between classes, which have similar structural and behavioral features. They are used to relate entities where the relationship has common characteristics and semantics. An association has two Association Ends, which connect the association to two classes. Associations and Association Ends can be named and can have the following properties:

  Navigability – indicates that an association can be navigated towards the class attached to the arrow. Associations that do not have the Navigability flag are either unknown or bi-directional.

  Aggregation / Containment – are properties of association ends modeling containment or composition. The property is attached to the association end of the class that is aggregated or contained. At most one association end can have the property. The Aggregation or Composition property is indicated as a hollow or filled diamond in diagrams.

  Multiplicity – is a constraint on a association end, specifying how many of the elements related by the association end are allowed to participate in the association. Multiplicity provides lower and upper bounds for the participating elements. * is a short form for 0..Infinity and is assumed if no Multiplicity range is provided.

  Ordering - if present, says that, for association ends with multiplicity greater than one, the set of related elements is ordered.

  Sorted – is a property of a association end that specifies that the elements attached by the association end are sorted based on their internal value.

  Derived – indicates an association that is derived from an existing association.
- **Generalization** is defined as the taxonomic relationship between a more general and a more specific element that is fully consistent with the first element, and that provides additional information.
- **Dependencies** - indicate a semantic relationship between two or more model elements. The client of the dependency requires features and therefore the presence of the supplier element of the dependency. The modeling concept is used to show the dependency between model packages.

2.6 Submodels

The remainder of this document describes the submodels of the Open Information Model. These submodels are organized as UML packages addressing four major subject areas, described below.

2.6.1 Analysis and Design Model

Analysis and design tools are integrated in the software design, development, and deployment life cycle at every step either as an input tool, for documentation purposes, or as a tool to analyze or validate results. This requires that they be tightly integrated with all the other applications, either through meta data interchange, or by sharing a common repository.

Object-oriented models, enterprise data models, and other meta data evolve individually or in a configuration with the described system. It is therefore necessary that the relationships between a model and the elements of a system can be expressed and maintained. The Analysis and Design Model therefore must provide not only modeling elements but also mechanisms for referring to elements outside of the scope of the model. This capability, as well as the generic concepts in the model, make it a natural fit to serve as a core model from which other more specialized models inherit more generic concepts such as package, containment, or dependency.

The Analysis and Design Model covers the domain of object-oriented modeling and design of software centric systems. The model provides concepts to describe problems and solutions throughout the complete software life cycle. The core of the model is the UML meta model. The UML consists of a notation and a meta model. The meta model describes the semantics of the notation in a formal way. It consists of a set of meta types, their relationships, and their meaning, and, as such, is ideally suited to become the core of a model for the analysis and design subject area.

2.6.2 Database and Warehousing Model

The Database and Warehousing Model provides meta data concepts for schema management for database design, schema reuse, and data warehousing. The Relational Database Schema package includes concepts found in standard SQL data definitions and similar types of formatted data models. For the most part, the model focuses on logical database concepts. However, it also includes some physical database concepts, because they are needed in nearly all usage scenarios. The Record-Oriented Database Schema package describes data maintained in the files, legacy databases, and so forth, of an enterprise.

Schemas definitions in XML define types for the valid structures in an XML document. The XML Schema package provides meta data types to represent the definitions that constitute an XML schema. The Report Definitions package represents information necessary for data reporting tools and their relationships to the systems they report on.

The Data Warehousing-specific packages extend the database schema model in several important directions in order to support data marts and data warehouse applications. The OLAP schema package data types capture descriptions of multi-dimensional (OLAP) data cubes used in decision support systems. The Data Transformations package captures information about data transformations used in moving data from production databases into a data warehouse or data mart.

2.6.3 Object and Component Model

The use of object-oriented development techniques to facilitate sharing and reuse of code has become strategic for enterprises in order to reduce cost and time to deployment. Reuse and sharing requires tracking meta data throughout the whole life-cycle of a component, from specification through design and subsequent enhancements. The Object and Component submodel intends to cover the various aspects of object-oriented development.

2.6.4 Knowledge Management Model

Knowledge management is the integrated and collaborative process of information asset creation, capture, organization, access, and usage. Information assets include databases, documents, and the experience and knowledge of domain experts.

The Knowledge Management Model provides the necessary meta data types to create catalog structures of enterprise information and to capture business terminology, its semantic relationships, and the mapping to storage structures.

The Knowledge Description Model extensions provide meta data types to define a controlled vocabulary to classify business information. The model allows one to define subject and topic terms and a hierarchy or classification tree of categories. Each category has a defined schema, which is a composition of locally defined properties and schemas inherited from parent categories. Information objects, such as data base tables, queries, reports, and documents, can appear in multiple categories, such as corporate sales, product marketing and finance. The vocabulary of controlled topics and subjects, together with uncontrolled terms, can be used to search the information maintained by the information directory.

The Semantic Definition submodel extensions provide meta data types to describe models for a semantic or linguistic processor. These processors let users interact with their database data without learning a data manipulation language. Before a linguistic processor can interact with a database, however, an analyst must articulate the mappings between the database schema and the semantic constructs familiar to the users. The model provides concepts to define business terms and synonyms and to map them to the names of SQL tables and columns.

2.6.5 Business Engineering Model

The goal of business or enterprise modeling is to develop a blueprint depicting how a company or a part of a company operates or should operate. For the purpose of this specification, a business is defined as a set of cooperative activities that are performed by the interaction of people and machines. Documenting the structure and processes of a business in a formal and accurate way is necessary not only to re-engineer them but also to automate or semi-automate them by computers.

The goal of the Business Engineering submodel is to align the storage and interchange representation of business engineering meta data. A well-defined set of meta data types provides standardization of information representation for the purpose of tool integration by vendors and end-users. No methodology or notation is assumed or enforced by the model. The information captured by the model can be represented visually using modeling techniques such as IDEF or the UML.

The Business Engineering Model contains several related packages. These packages provide meta data types to describe the goals of a business, its organizational structure, the rules that govern the business, and the processes that move information and material. The separation of goals, process, and structure allows a more flexible and understandable description of a business.

2.7 Compatibility

The MDC OIM supersedes the MDIS 1.0 and the Microsoft OIM 1.0 specification. The MDC OIM offers backward compatibility in order to protect the investment software vendors have made in adopting one of the predecessor models.

The changes between the existing models and the MDC OIM can be grouped into three areas:

- Name changes – are simple changes of the identifier of a model element, e.g. *Type* in UML 1.0 has changed to *Classifier* in UML 1.3. The UML representation of the MDC OIM that accompanies this document provide the ability to maintain different name compatibility sets (OIM 1.0, MDIS 1.0) for all model elements including classes, attributes, associations, association ends, etc. For example, the

MDIS 1.0 name compatibility set maps MDIS *DATABASE* onto MDC OIM *Catalog*. An implementation may choose a specific name set to ensure backward compatibility.

- Additions – are modifications that add new elements or attributes to the model. The MDC OIM includes new features such as the UML 1.3 Activity Diagrams package (UML 1.0 did not include Activity Diagrams) for which no semantically equivalent concept existed. The UML concept of derivation of attributes and associations has been used to create backward compatible structures in case an element has been moved. In such a case the old element continues to exist and the new one is derived from the existing one.

- Incompatible changes – are structural or semantic modifications that require a modification or migration of an implementation. The UML 1.3 Model Management package, for example, has changed significantly from the UML 1.0 version. The MDC OIM maintains compatible structures along with the new structures to maintain backward compatibility. These compatibility structures are noted in the model and will be removed in future revisions. They should be used only if backward compatibility is an issue.

3 Analysis and Design: Unified Modeling Language (UML)

3.1 Overview

The UML package describes version 1.3 of the Unified Modeling Language, a standard from the Object Management Group (OMG). It forms the foundation of the MDC Open Information Models and is the package from which all other packages inherit.

Having submodels inherit from the UML yields the usual benefits of reuse via inheritance:

- It reduces the overall size and complexity of OIM by reusing UML concepts in many submodels. The containment hierarchy defined for UML Packages and Model Elements, for example, is generally applicable. For example, database schemas using the generic UML containment mechanism contain table definitions.
- It makes sharing between different types of tools simpler and more efficient. For example, a useful function in a database-oriented development tool is to generate a component definition from a table definition. Usually, this requires translating details about the table definition into details about the component's class. However, in the MDC OIM, since both database table and component class inherits from UML class, many details of a table definition can be interpreted as details of a component class. Thus, a table definition's details can be directly interpreted as a component class definition, without any explicit translation.
- It enables generic tool functionality that can analyze and interpret meta data structures without understanding the complete semantics. Dependencies, for example, are defined in UML and used by all submodels. This allows generic analysis tools to show dependencies independent of the semantics of the dependent objects. Once the user has navigated to a specific item, a specific tool that understands the individual semantics can be invoked.

The architecture of the UML package is based on version 1.3 of the Unified Modeling Language published by the Object Management Group. Please review the UML version 1.3 specifications and model diagrams available at the Web site http://www.omg.org.

3.2 Semantics

The MDC OIM is based on UML 1.3 as standardized by the OMG. The MDC is committed to evolving the MDC OIM as UML evolves and to ensuring backward compatibility to the OIM 1.0 and MDIS 1.0 specifications. This section contains additions to UML 1.3 to ensure backward compatibility of the MDC OIM with UML 1.0, on which OIM 1.0 was based.

The MDC OIM enables the modeling of meta data at multiple levels of abstraction - including conceptual, logical, physical, and deployed. It uses the UML concept of *refinement* to link objects of different levels together, where the refining element is somehow less abstract (more physical) than the refined element. The following diagrams illustrate a common use of this concept:

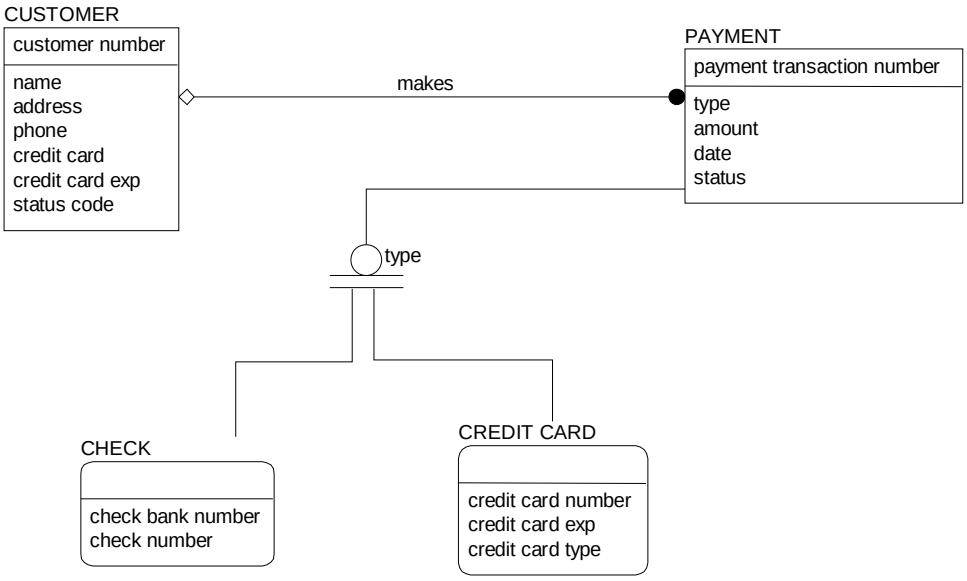


Figure 4: Sample Logical Database Model

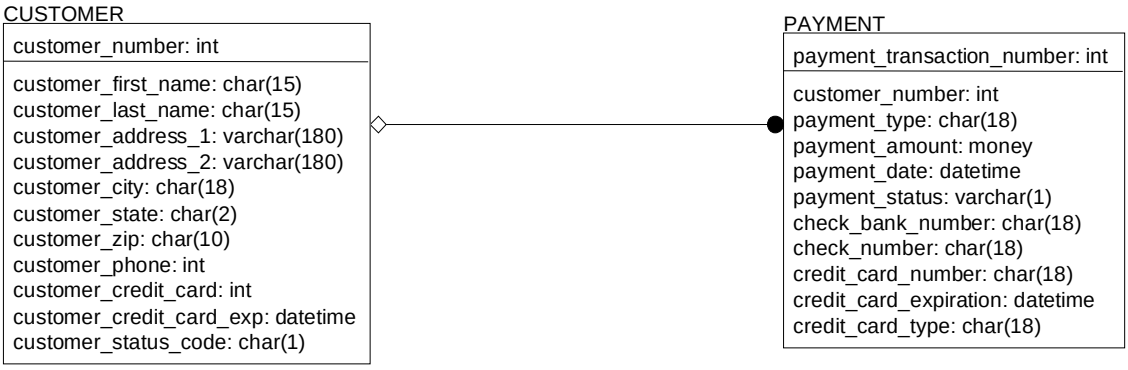


Figure 5: Sample Physical Database Model

The first diagram depicts a logical data model that relates a customer to a payment. The second diagram depicts the same model with implementation (deployment) details added. A refinement object can be used to indicate which specific physical elements are refinements of which logical elements. It can, likewise, be used to tie a single physical element (e.g. payment) as a refinement of multiple logical elements (e.g. payment, check, and creditcard).

In a conceptual model, the type of an object (e.g. table, component) is not strongly defined. For example, a user may want to describe a system that relates the concepts of customers and payments, but not to specify whether it will be deployed in a relational database or as a C++ component. It is suggested that a conceptual element simply modeled as a UML model element with a name and description can be linked to other elements through dependencies and refinements until an equivalent logical element can be identified and created. As model elements, they can still be packaged, diagrammed, named, defined, refined, and related through dependencies. When a more type-specific logical element is created, it should conform to the information model for that type with regard to properties, relationships, and constraints.

3.3 Class Reference

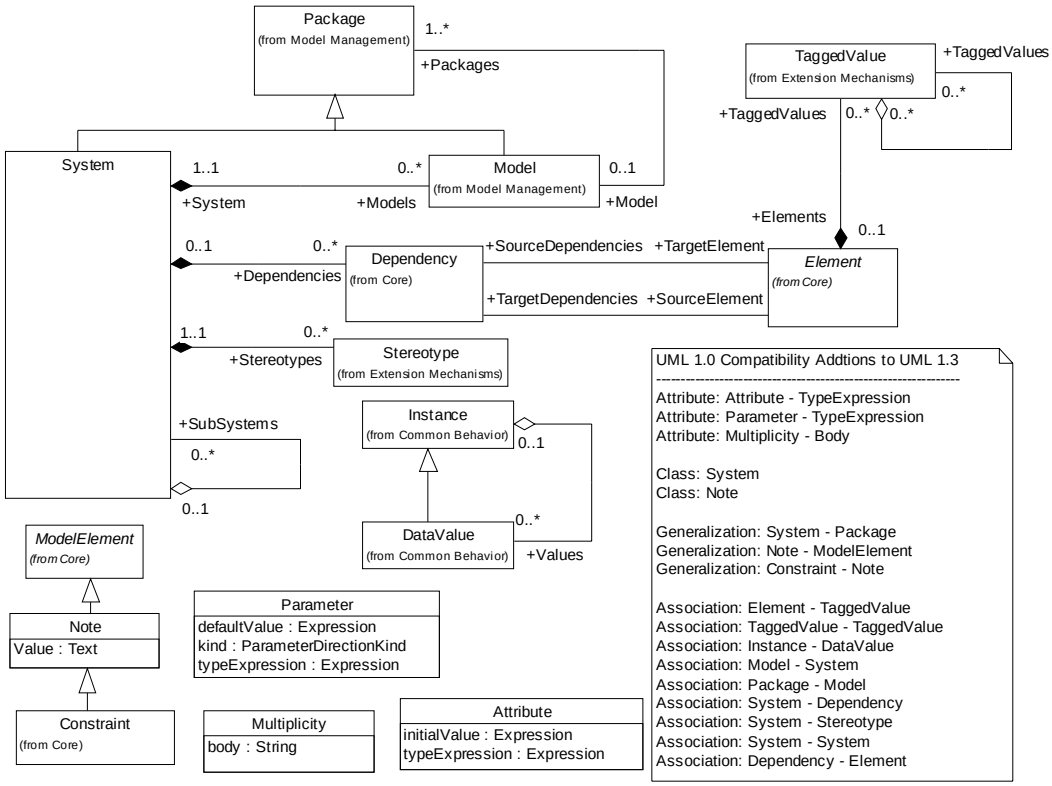


Figure 6: OIM 1.0 Compatibility

3.3.1 Note

A *note* is a comment attached to an element or a collection of elements. A Note has no semantic impact. See UML 1.0 specification for more details.

Specializes

- ModelElement (from UML)

Attribute

- *Value* (Text) – The uninterpreted content of the *note*.

3.3.2 System

A collection of connected units that are organized to accomplish a specific purpose. One or more models can describe a system, possibly from different points of view. See UML 1.0 specification for more details.

Specializes

- Package (from UML)

Association

- *Models* (Model) – collection of Model elements that constitute the System.
- *Dependencies* (Dependency) – dependencies between the System and other model elements.

- *Stereotypes* (Stereotype) – stereotypes that apply to the System and its subelements.
- *SubSystems* (System) – collection of Systems as sub-elements that constitute the System.

4 Analysis and Design: UML Extensions

4.1 Overview

The UML Extensions package has two primary purposes. First, it enhances the UML package by introducing the ability to describe the presentation or display of UML elements. This is extremely relevant information to gather, as anyone who has ever spent time using an object-oriented design tool knows, because much of the time is spent arranging diagrams so that they communicate the model clearly. Second, it provides a place for other general-purpose additions to the UML package.

The UML Extensions package is dependent on the UML package.

4.2 Semantics

The UML extension package is divided in to three subpackages. The Presentation and View Elements package contains all of the classes used to model the presentation of elements on a diagram. The Auxiliary Elements package contains all of the other general-purpose extensions to the UML package.

The Syntax Elements package is used to store and interchange formal definitions of computer languages and expressions. The unambiguous definition of a grammar allows developing parsers for a language automatically. Each Grammar Rule defines a pattern that defines a named structural part of the language. The name forms the set of non-terminal Symbols available in the Syntax. Such a non-terminal Symbol is replaced by the Statement part of the Grammar Rule also called left-hand side.

4.3 Class Reference

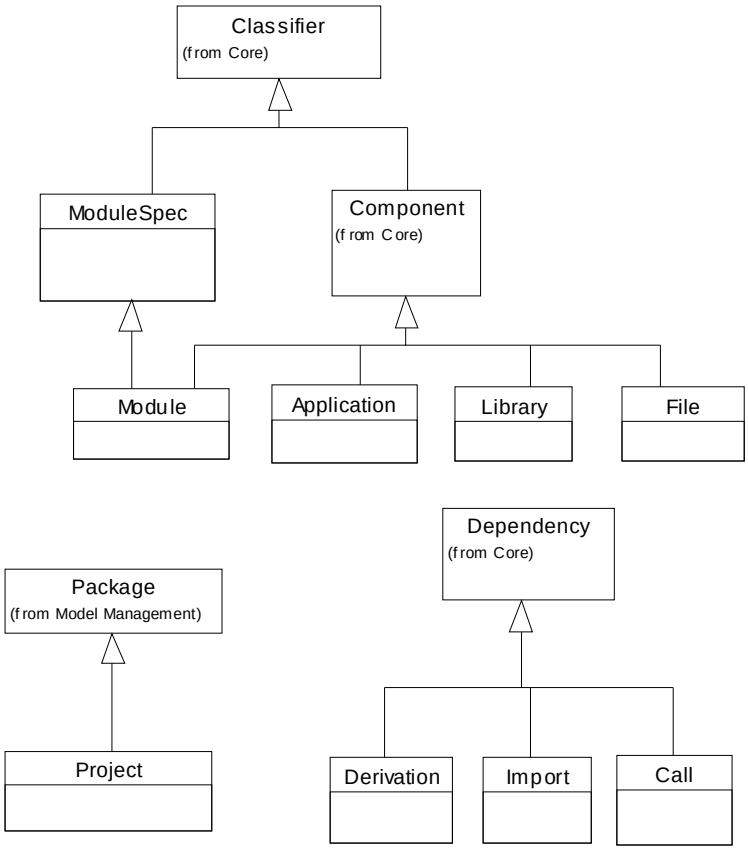


Figure 7: Auxiliary Elements

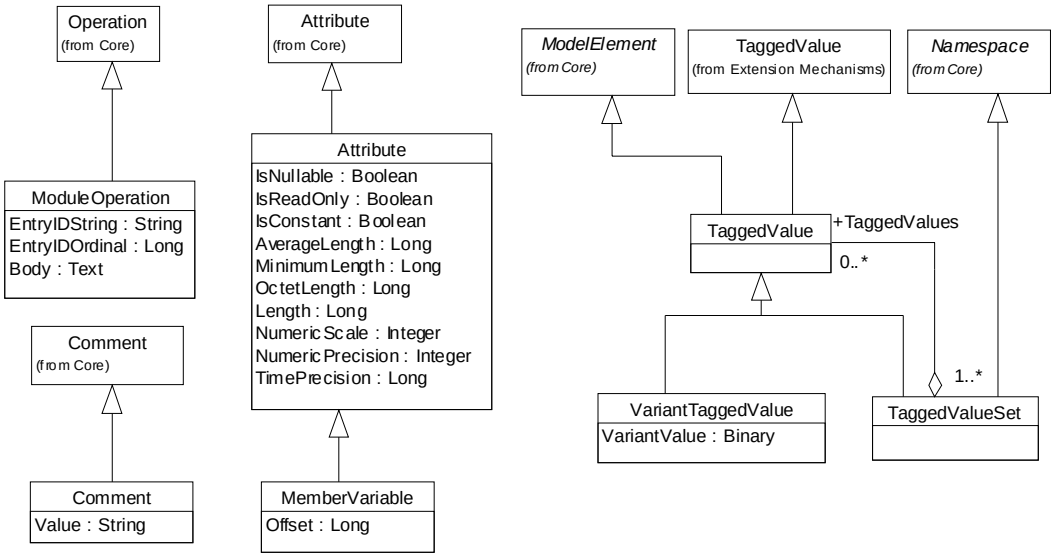
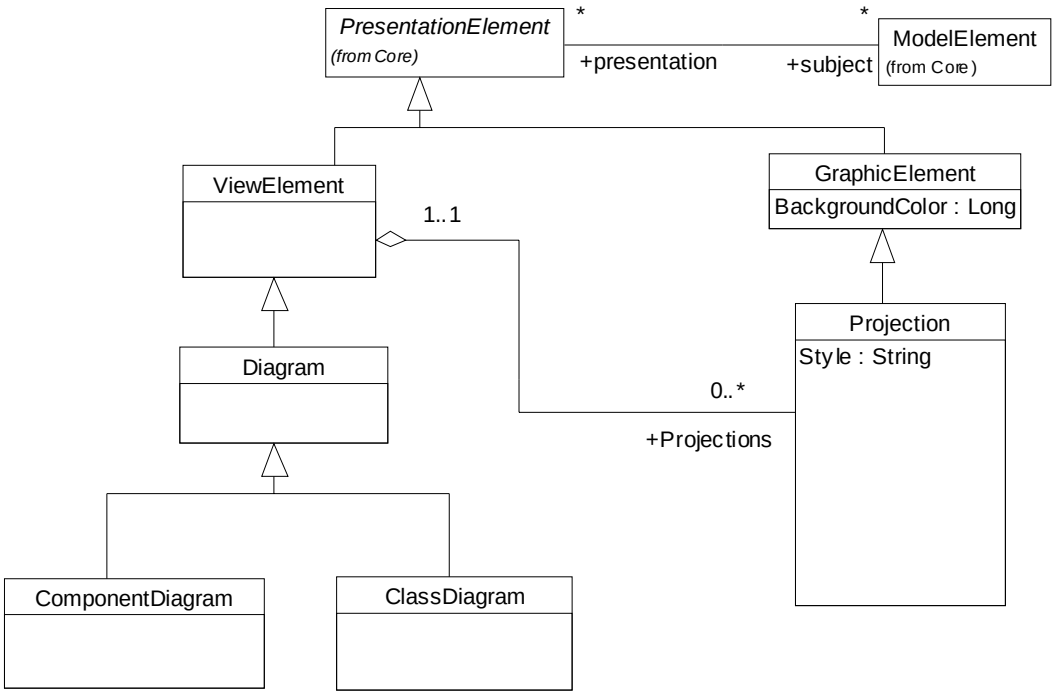


Figure 8: Additional Auxiliary Elements

1



2

3 **Figure 9: View Elements**

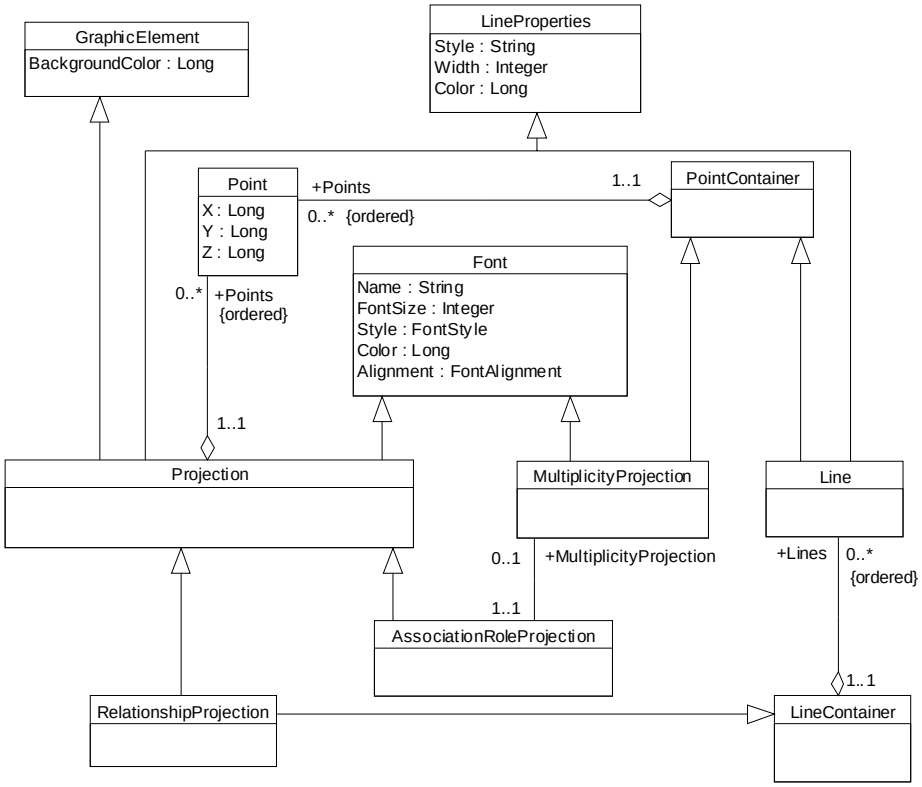


Figure 10: Projections

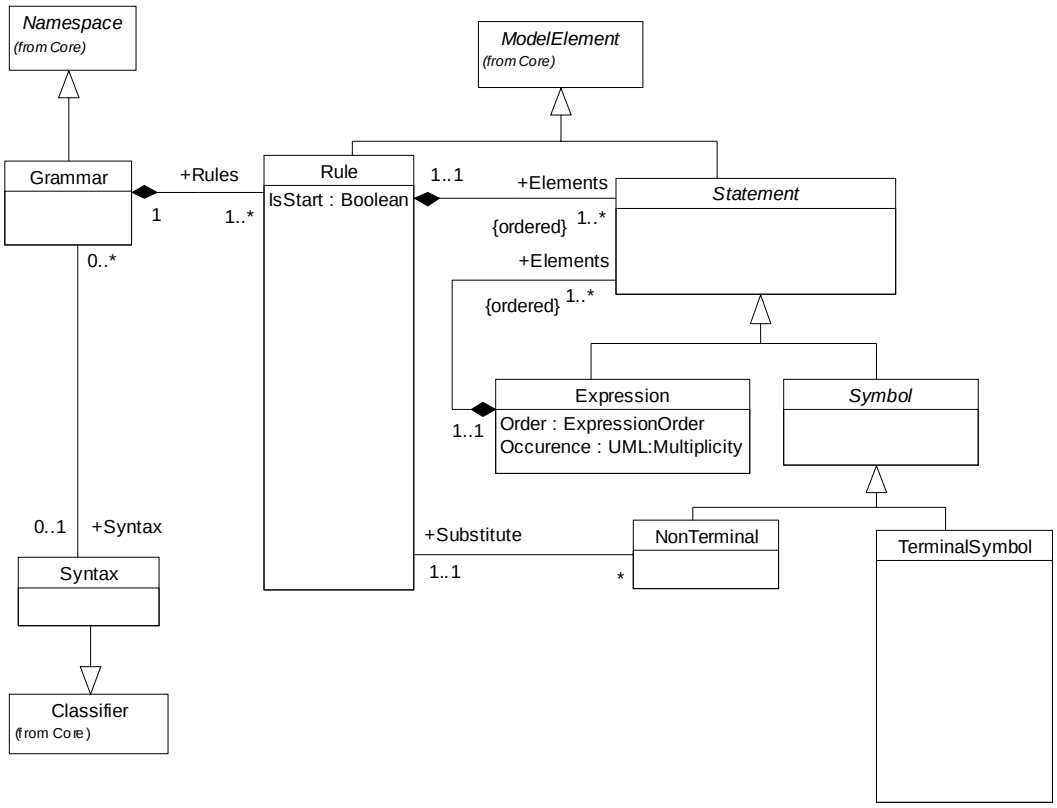


Figure 11: Syntax Elements

4.3.1 Application

This class defines an application (e.g., an .exe). This class realizes the UML "application" stereotype.

Specializes

- Component (from UML)

4.3.2 AssociationEndProjection

This class represents the projection of an association end onto a diagram. This class provides the position (and other view characteristics) of the association end label. Any associated Multiplicity Projection gives the position (and other view characteristics) of the labels showing the association end multiplicity.

Specializes

- Projection (from UML)

Association

- *MultiplicityProjection* (MultiplicityProjection) – The projection of the multiplicity label of the association end.

4.3.3 Attribute

This call extends the features of the UML attribute.

Attribute

- *IsNullable* (Boolean) – Indicates whether the attribute can be null.
- *IsReadOnly* (Boolean) – Indicates whether the attribute is read only.
- *IsConstant* (Boolean) – Indicates that the attribute has a constant value. Such an attribute must have an InitialValue that defines the constant value defined.
- *AverageLength* (Long) – The expected average length of data stored in this attribute.
- *MinimumLength* (Long) – The smallest length of data that can be stored in this attribute.
- *OctetLength* (Long) – Maximum length in octets (bytes) of the attribute, if the type of the attribute is character or binary. A value of zero means the attribute has no maximum length.
- *Length* (Long) – The maximum possible length of a value of the attribute.
- *NumericScale* (Integer) – The number of digits to the right of the decimal point in the column for numeric attributes.
- *NumericPrecision* (Integer) – The maximum number of base 10 digits that can be stored for numeric attributes.
- *TimePrecision* (Long) – Datetime precision (number of digits in the fractional seconds portion) if the attribute is a datetime or interval type.

Association

- DerivedAttributes (Attribute) – The collection of attributes that are derived from this attribute. For example, your age is based on your birthday.

Specializes

- Attribute (from UML)

4.3.4 Call

This class represents the invocation from one element of another.

Specializes

- Dependency (from UML)

4.3.5 ClassDiagram

This class specifies the domain and behavior of a diagram encompassing types, classes, and their relationships.

Specializes

- Diagram

4.3.6 ComponentDiagram

This class specifies the domain and behavior of a diagram encompassing components and their relationships.

Specializes

- Diagram

4.3.7 Derivation

This class represents the derivation of one element to another.

Specializes

- Dependency (from UML)

4.3.8 Diagram

This class specifies the domain and behavior of a graphical projection of a collection of model elements. Diagrams are most often rendered as a connected graph of arcs (relationships) and vertices (other model elements).

Specializes

- ViewElement

4.3.9 Dictionary

Instances of this class maintain the terminal symbols of a syntax, i.e. of a grammar. The dictionary entries are ordered by name and must be unique for a grammar.

Specializes

- ModelElement (from UML)

Associations

- *Symbols* – collection of sorted Terminal symbol objects that constitute the Dictionary.

4.3.10 Expression

An expression consists of a collection of sub-expressions or *symbols*. The property Order of the Expression object indicates if the collection should be treated as a Sequence or a set of Alternatives. An AND expression corresponds to the EBNF sequence (A B C) and the OR expression to an Alternative (A | B | C).

Specializes

- Statement

Attributes

- *Order* (ExpressionOrder) – is an enumeration type property, which controls the type of expression: AND : Alternative, OR : Sequence.

Associations

- *Elements* (Statement) – set of objects of RuleElement type, i.e. Expressions or Symbols.

4.3.11 ExpressionOrder

An enumeration that determines if an expression is a sequence or a set of alternative (choices).

Values

- *AND* – The elements of the Expression represent a sequence.
- *OR* – The elements of the Expression represent alternatives.
- *NONE* – Same as AND (default value)

4.3.12 File

This class represents an operating system file. Instances of the class should not represent specific files (this is handled with the Surrogate class in the Generic Element Package), but may indicate the use of a file type as part of a implementation model.

Specializes

- Component (from UML)

4.3.13 Font

This class represents the use of a graphic font for text rendering.

Attributes

- *Name* (String) – The name of the font (e.g., MS Sans Serif, or Courier).
- *FontSize* (Integer) – The point size of the font (e.g., 10 for pica, 12 for elite).
- *Style* (FontStyle) – Indicates the font style defined by the FontStyle enumeration (e.g., regular, bold, italic, bold italic).
- *Color* (Long) – The color of the text. The value should be in standard RGB three-byte format.
- *Alignment* (FontAlignment) – The alignment or justification of the text within its bounding rectangle.

4.3.14 FontAlignment

An enumeration whose values indicate the alignment or justification of text within its bounding rectangle.

Values

- FONTALIGNMENT_LEFT = 0
- FONTALIGNMENT_CENTER = 1
- FONTALIGNMENT_RIGHT = 2

4.3.15 FontStyle

An enumeration whose values indicate the font style of text.

Values

- FONTSTYLE_REGULAR = 0
- FONTSTYLE_ITALIC = 1
- FONTSTYLE_BOLD = 2
- FONTSTYLE_BOLDITALIC = 3

4.3.16 Grammar

Instances of this class represent a set of rules that can conform to a specific syntax. For example, an XML document or a C++ file could constitute a grammar.

Specializes

- ModelElement (from UML)

Associations

- *Rules* (Rule) – Set of Rules that constitute the Grammar.
- *Syntax* (Syntax) – Syntax to which the Grammar conforms.

4.3.17 GraphicElement

This class represents additional graphics details for a projection.

Specializes

- PresentationElement (from UML)

Attribute

- *BackgroundColor* (Long) – The color of the area which surrounds and/or is between graphical elements of an object. The value should be in standard RGB three-byte format.

4.3.18 Import

This class represents one package being imported by another. An import dependency causes the public contents of the target package to be referenceable in the source package.

Specializes

- Dependency (from UML)

4.3.19 Library

This class represents defining a library (e.g., a DLL). It realizes the UML “library” stereotype. A library is associated with the elements (including modules) it contains by means of the inherited UML “implements” relationship.

Specializes

- Component (from UML)

4.3.20 Line

This class represents a single line on a diagram. This may be made up of several line segments. The first point in the points collection is an absolute point on the diagram and every point thereafter is a point relative to the previous point.

Associations

- *LineContainer* (LineContainer) – The view object described using this line.

Specializes

- PointContainer
- LineProperties

4.3.21 LineContainer

This class represents a set of lines.

Associations

- *Lines* (Line) – The lines contained in this container.

4.3.22 LineProperties

This class represents various properties of a line.

Attributes

- *Style* (String) – The style of line (e.g., solid, dashed, or dotted).
- *Width* (Integer) – The width or weight of the line in points.
- *Color* (Long) – The color of the line. The value should be in standard RGB three-byte format.

4.3.23 MemberVariable

This class represents a member variable. This includes members of structures, unions, member variables/fields on a type, and entry variables of modules.

Attributes

- *Offset* (Long) – The offset of the member variable in the structure.

Specializes

- Attribute (from UML Extensions)

4.3.24 Module

This class represents a module (i.e., a group of operations and entry variables).

Specializes

- Component (from UML Extensions)
- ModuleSpec (from UML Extensions)

4.3.25 ModuleOperation

This class represents an operation that is contained in a module.

Attributes

- *EntryIDString* (String) – Identifies a named entry point in the DLL.
- *EntryIDOrdinal* (Long) – Identifies an entry point in the DLL via an ordinal.
- *Body* (Text) – The text of the body of the module operation. This may include its mandatory encompassing signature.

Specializes

- Operation (from UML Extensions)

4.3.26 ModuleSpec

This class represents the specification of a module (i.e., a group of operations and entry variables). The module is related to its members by the relationship inherited from Classifier (from UML).

Specializes

- Classifier (from UML)

4.3.27 MultiplicityProjection

This class represents the projection of the multiplicity of an association end.

Specializes

- PointContainer

- Font

Associations

- *AssociationEndProjection* (AssociationEndProjection) – The projection of the association end for which this is the multiplicity.

4.3.28 NonTerminalSymbol

Name will contain the name of the symbol.

Specializes

- *Symbol*

Associations

- *Substitute* (Rule) – Rule that substitutes for the non-terminal symbol.

4.3.29 Point

This class specifies the domain and behavior of a three-dimensional Cartesian point in *twips*. A twip is a unit of measurement, implemented as 1/20 of a point, or 1/1440 of an inch. There are 567 twips to a centimeter. Twips are screen-independent measurements.

Attributes

- *X* (Long) – The position along the x-axis.
- *Y* (Long) – The position along the y-axis.
- *Z* (Long) – The position along the z-axis.

4.3.30 PointContainer

This class represents any view object that can be described as a set of points.

Associations

- *Points* (Point) – The points that make up this element.

4.3.31 Project

This class represents a development project, such as a VBP or DSP file.

Specializes

- Package (from UML)

4.3.32 Projection

A projection of a model element onto a view element. This class can accommodate most of what the common types of projection require: a collection of points, a font, a line style, and some basic graphic element details. If a projection requires more advanced details (e.g., a projection composed of multiple graphic components), then another class and/or interface will be required.

For example, the projection for a Class depicted in some tool using the UML notation may simply be two points defining the (left, top) and (width, height), and font information describing the display of the class name label.

In other cases, additional interfaces may provide additional view information. For example:

- A projection of an association end may require positional and font information to be recorded about both the assoication end name label and the multiplicity label.
- A projection of a Class in some tool may require information about whether properties and/or methods are to be shown.

Specializes

- Font
- GraphicElement
- LineProperties

Attributes

- *Style* (String) – A string indicating any presentation information beyond location necessary to render an element on a view element.

Associations

- *Points* (Point) – The collection of points specifying the placement of the referenced model element on the referenced view element. If a model element is composed of several graphical elements on a view element, then it may have collections of points appearing on other interfaces. The specific type must specify what this collection of points is to be used for. For node-like elements, this collection should consist of two points (Left,Top,0) and (Width,Height,0). For line-like elements, the first point in this collection should describe the absolute position of the first point of the line; each subsequent point should describe its position relative to the previous point in the collection.

4.3.33 RelationshipProjection

A projection of any UML relationship model element onto a view element such as a diagram. This projection has a collection of all lines of the relationship that, in turn, contain a collection of points.

The collection of points on the inherited projection type is reserved for the (left, top) and (width, height) points of the relationship's name.

Specializes

- LineContainer
- Projection

4.3.34 Rule

Specializes

- Classifier (from UML)

Associations

- *Elements* (Statement) – Represents the root of the right side of the rule, i.e. an Expression or a Symbol.

4.3.35 Statement

Specializes

- ModelElement

Attributes

- *Occurrence* (Multiplicity) – Occurrence of the statement (Expression or Symbol) in a Rule or Expression (1, 0..N, N, N..M).

4.3.36 Symbol

Abstract class representing elements decomposed within a rule. Is specialized to TerminalSymbol and NonTerminalSymbols.

Specializes

- Statement

4.3.37 Syntax

Represents the syntax or rules to which an grammar conforms. An example is the C++ language.

Specializes

- Classifier (from UML)

Associations

- *SyntaxSymbols* (SyntaxSymbol) - Collection of terminal symbols used described by the Syntax.

4.3.38 TaggedValue

This class represents a tagged value that can be a member of a tagged value set (TaggedValueSet).

Specializes

- TaggedValue (from UML)

4.3.39 TaggedValueSet

This class represents a set of tagged values.

Specializes

- TaggedValue

Associations

- *TaggedValues* (TaggedValue, derived from UML:TaggedValue.taggedValues) – The tagged values in this set.

4.3.40 TerminalSymbol

Specializes

- Symbol

4.3.41 VariantTaggedValue

This class represents a tagged value that may contain a COM variant.

Specializes

- TaggedValue (from UML Extensions)

Attributes

- *VariantValue* (Binary) – An arbitrary variant value to be associated with the name for the associated *Element*.

4.3.42 ViewElement

ViewElement is an abstract class that represents a top-level container for projections, e.g. a diagram.

Specializes

- ModelElement (from UML)

Associations

- *Projections* (Projection) – The collection of all projections referencing the model elements that appear on this view element.

4.4 OIM 1.0 Compatibility

- The reflexive relationship on Attribute (the relationship whose assoication end names are BasedAttributes and DerivedAttributes) has been removed because its functionality was duplicated by Derivation.

5 Analysis and Design: Generic Elements

5.1 Overview

The Generic Elements package provides a set of general-purpose classes that are relevant across diverse information models. In some cases the classes described are designed to fill a temporary gap in other models until a more compete model is introduced.

Specifically, the Generic model:

- Adds the ability to specify component version information.
- Adds the ability to point to external objects, such as files.
- Introduces the concept of handlers for OIM objects.

The Generic Elements package is dependent on the UML package.

5.2 Model Reference

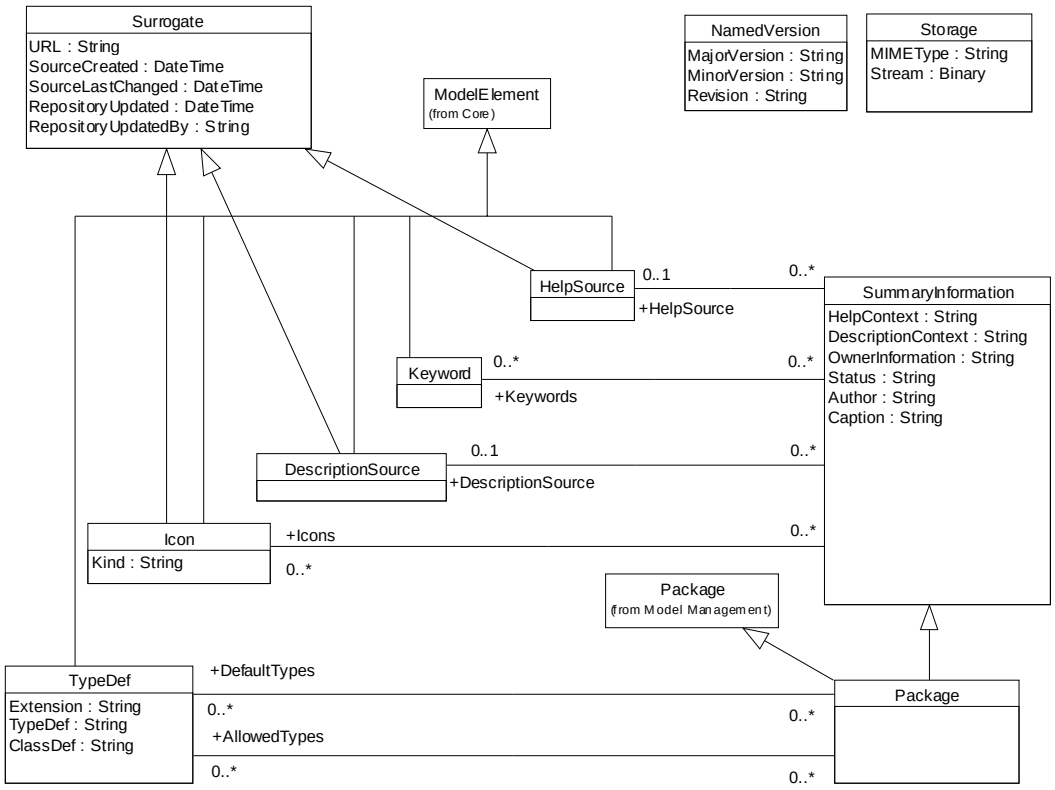


Figure 12: Generic Elements

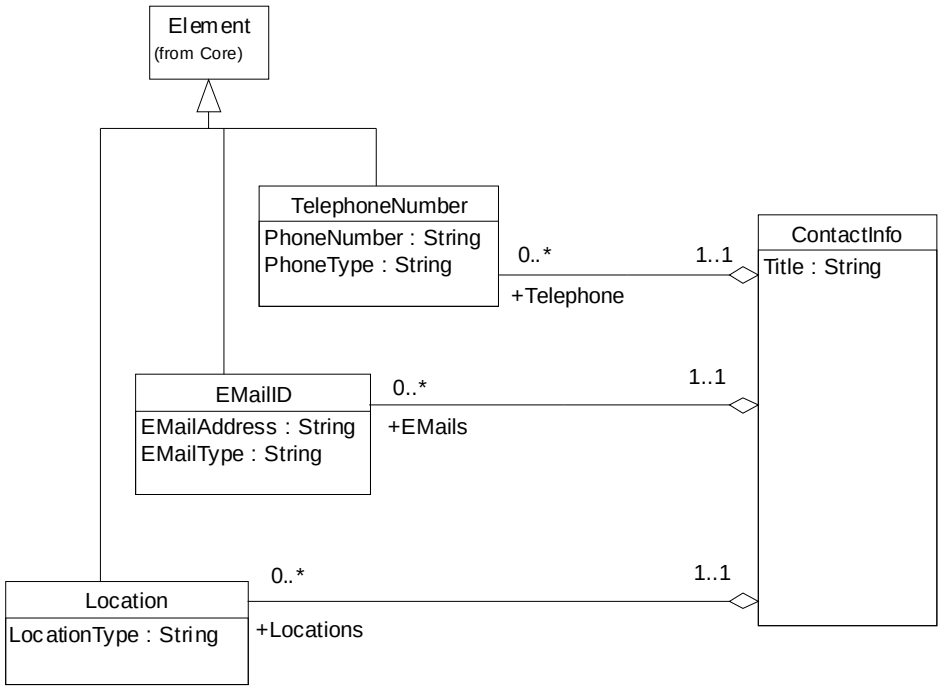


Figure 13: Contact Information

5.2.1 ContactInfo

This class is designed as a simple way to describe a person who can be contacted.

Specializes

- ModelElement (from UML)

Attributes

- *Title* (String) – The contact may have a title, such as “Director”.

Associations

- *EMails* (EMailID) – The contact may have one of more e-mail addresses.
- *Telephone* (TelephoneNumber) – The contact may have one of more telephone numbers.
- *Location* (Location) – The contact may have one of more locations, such as addresses.

5.2.2 DescriptionSource

A d*escription source* is a type of surrogate that points to a resource that provides additional descriptive information. For example, it could point to a specification document for a classifier.

Specializes

- Surrogate (from Generic Elements)
- ModelElement (from UML)

5.2.3 EMailID

An instance of this class defines a contact's e-mail address.

Specializes

- Element (from UML)

Attributes

- *EMailAddress* (String) – The actual e-mail address would be stored here.
- *EMailType* (String) – Usage information about the e-mail address, such as whether it's a home account, business account, or backup business account.

5.2.4 HelpSource

A *help source* is a type of surrogate that points to a resource that provides help. For example, it could point to a help file on a network resource.

Specializes

- Surrogate (from Generic Elements)
- ModelElement (from UML)

5.2.5 Icon

An *icon* is a type of surrogate that points to an icon resource.

Specializes

- Surrogate (from Generic Elements)
- ModelElement (from UML)

Attributes

- *Kind* (String) – Indicates the kind of the icon. The value is user-defined. It allows different icons that represent the object in different conditions to be distinguished. For example: color vs. monochrome or opened vs. closed (for an icon of a folder).

5.2.6 Keyword

This class represents *keywords* that are used for classifying an object. Such keywords can be used as a search criterion when searching catalogs for objects.

Specializes

- ModelElement (from UML)

5.2.7 Location

This class is used to represent any type of physical location, such as a house address.

Specializes

- Element (from UML)

Attributes

- *LocationType* (String) – Indicates the type of address, such as work or home.

5.2.8 NamedVersion

This class represents user-defined version information (e.g., version 2.1.001).

Attributes

- *MajorVersion* (String) – The major version.
- *MinorVersion* (String) – The minor version.
- *Revision* (String) – The Revision (e.g., the build number).

5.2.9 Package

This class is a further refinement of the UML package. It adds the ability to define allowed and default types for items in the package.

Specializes

- SummaryInformation
- Package (from UML)

Associations

- *AllowedTypes* (TypeDef) – The types that are allowed to be contained within the package. If no such types are specified, then the package is allowed to contain any type.
- *DefaultTypes* (TypeDef) – The types that are normally in that package.

5.2.10 SummaryInformation

This class allows additional summary information beyond that provided by SummaryInformation (from UML Extensions).

Attributes

- *HelpContext* (String) – A key into the associated HelpSource.
- *DescriptionContext* (String) – A key into the associated DescriptionSource.
- *OwnerInformation* (String) – The contact information for the object, such as the person or organization which manages this object.
- *Status* (String) – Indicates the status of the object, such as its degree of completeness, its robustness, and so on. For example, a document may have status “draft”, or a component may have status “published”. This property is not intended as a formal classification of objects for use by configuration management tools, but more for browser tools as display information.
- *Author* (String) – The person or organization who was the major creator of the object.
- *Caption* (String) – The human name for an object. This property could store the value for this object that should be displayed on a form or report.

Associations

- *HelpSource* (HelpSource) – The source of help on this object.
- *Keywords* (Keyword) – The keywords used to describe or categorize this object.
- *DescriptionSource* (DescriptionSource) – The description source for the object.
- *Icons* (Icon) – The icons that represent the object.

5.2.11 Storage

This class represents the physical storage of an element. The class may be used when the actual implementation of a component or element accompanies the element description.

Attributes

- *MIMEType* (String) – Describes the format of the storage stream.
- *Stream* (Binary) – Contains the actual element storage.

5.2.12 Surrogate

A *surrogate* represents an object that is not stored in the repository.

This class simply allows a URL of the source object to be recorded, along with relevant timestamp details. It is envisaged that future extensions to this interface will be defined to add an Object property, allowing the source object to be set and/or retrieved, and to provide support for synchronization between the surrogate and its source.

Attributes

- *URL* (String) – The URL of the source (surrogated/replicated) object. For example, for a file with path C:\examples\myfile.txt the URL would be File://C:\examples\myfile.txt.
- *SourceCreated* (DateTime) – The date and time that the source object was created.
- *SourceLastChanged* (DateTime) – The date and time that the source object was last changed. This records any last changed timestamp on the source object at the time the replica/surrogate is created (or refreshed). Comparing this with the current value of the last changed timestamp on the source object indicates whether the replicated details are still up-to-date.
- *RepositoryUpdated* (DateTime) – The date and time that the repository was last updated with information from this source.
- *RepositoryUpdatedBy* (String) – Identifies the user ID that initiated the last repository update from this source.

5.2.13 TelephoneNumber

This class represents a phone number. It allows for the expression of both the actual phone number and its type.

Specializes

- Element (from UML)

Attributes

- *PhoneNumber* (String) – The actual phone number, such as (123) 456-7890.
- *PhoneType* (String) – Explains when this phone number should be used, such as home or business phone, cell phone, pager, or fax.

5.3 OIM 1.0 Compatibility

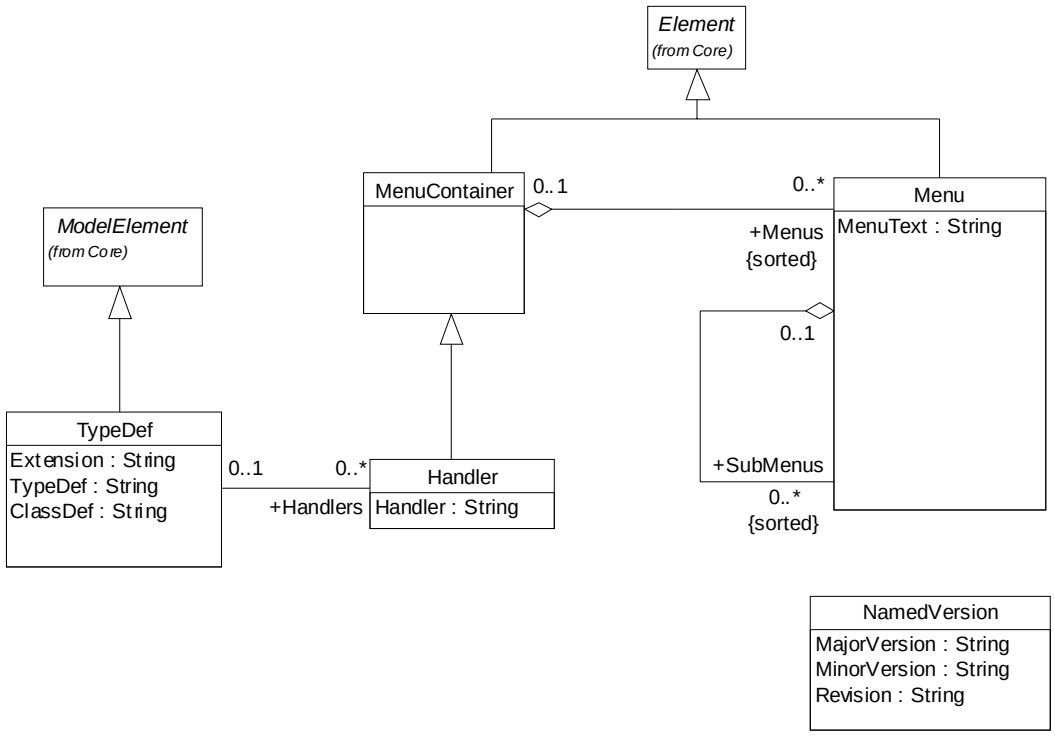


Figure 14: OIM 1.0 Compatibility Classes

5.3.1 Handler

This class defines a *handler* for some type of object within some context. The context is defined through the name on the relationship between a type and its handlers. For example, a particular handler may be responsible for various user interface events on objects of some type within the context of some Explorer. The handler would identify a class that would be instantiated by the Explorer, which would provide the needed services.

Specializes

- MenuContainer

Attributes

- *Handler* (String) – The identifier of the component that serves as the handler. An instance of the handler can be created to handle events within the context.

5.3.2 Menu

This class represents a user interface menu or menu item. This includes both menus appearing on menu bars and free-floating context menus.

Specializes

- Element (from UML)

Attributes

- *MenuText* (String) – The name appearing on the menu item (with ampersands on underscored characters).

Associations

- *SubMenus* (Menu) -The menu items on this menu. There should only be menu items if the supermenu is a menu itself.

5.3.3 MenuContainer

An instance of this class contains menus.

Specializes

- Element (from UML)

Associations

- *Menus* (Menu) – The collection of contained menus.

5.3.4 TypeDef

This class represents an object in the repository that can be associated with a handler.

Specializes

- ModelElement (from UML)

Attributes

- *Extension* (String) A short string of characters which help identify objects of this type (e.g., the three-letter DOS file extension).
- *TypeDef* (String) -The ID of the type this definition this applies to.
- *ClassDef* (String) – The ID of the class this definition this applies to.

Associations

- *Handlers* (Handler) – Associates a type definition with the handlers of that type within various contexts. For example, a type may be associated with a handler that will manage the context menu within the context of some Explorer.

6 Analysis and Design: Common Data Types

6.1 Overview

The Common Data Types package provides data type definitions for the Open Information Model. The goal of the model is to standardize and unify data types. The package elements are used as a base set of types that is extended to represent the data type concepts of other information models in the Open Information Model.

The package is defined as a set of extensions to the Unified Modeling Language Information Model (UML) that provides a set of classes for describing data types. The scope includes the common data types needed for component specifications, component implementation languages, and databases. The intent is that this model is extended and specialized by particular information models of these and other domains.

6.2 Semantics

The Common Data Types package provides definition of data types for the Open Information Model. It forms the basis for extensions to include additional data types in specific domains. It is expected that the types described in the model will be defined as reusable instances, which will minimize the number of instances representing identical data types. Type instances should be uniquely identified by name or identifier within a data type set.

6.3 Class Reference

This section describes the data types of the Common Data Types package.

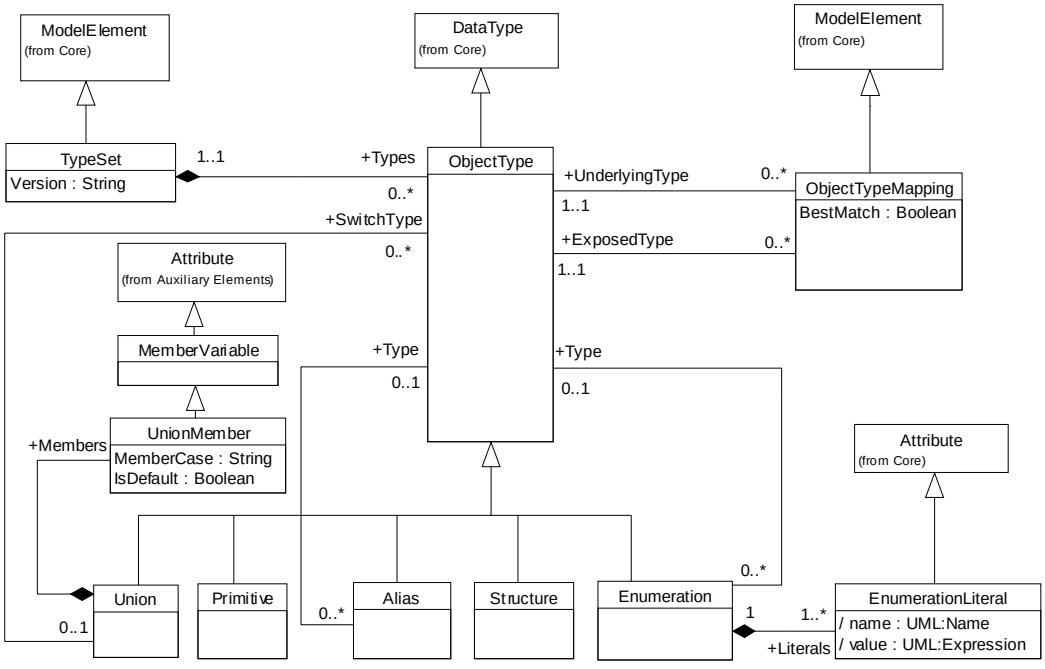


Figure 15: Data Types

6.3.1 Alias

Alias describes an alias for another type (e.g., a C++ typedef). This includes any user-defined datatype that merely provides an alternate name for a type.

Specializes

- ObjectType

Associations

- *Type* (ObjectType) – The type represented.

6.3.2 Enumeration

Enumeration describes an enumeration datatype; i.e., a set of named constants (e.g., a C++ enum). Each enumeration constant is a (constant) attribute, with a defined initial value.

Specializes

- ObjectType

Associations

- *Type* (ObjectType) – The type of the constants within the enumeration.
- *Literals* (EnumerationLiteral) – The set of literals for the enumeration.

6.3.3 EnumerationLiteral

Describes the values that an instance of the attribute of the related enumeration type may contain.

Specializes

- Attribute (from UML)

Attributes

- *Name* (Name, derived from UML:ModelElement.Name) – A display name for the literal value.
- *Value* (Expression, derived from UML:Attribute.InitialValue) – The value (e.g. stored) for the literal.

6.3.4 ObjectType

ObjectType is an abstract type that is supported by all data types in the Common Data Types package. It extends the UML Classifier by allowing relationships to other types that reference the object type in their definition (e.g., aliases or pointers).

Specializes

- DataType (from UML)

6.3.5 ObjectTypeMapping

The natural mapping of an object type in a namespace to a set of object types in another namespace.

Specializes

- ModelElement (from UML)

Attributes

- *BestMatch* (Boolean) – Indicates that the mapping between a pair of object types is the best match. There is a constraint that for each underlying object type, only one instance of the mapping will have BestMatch = TRUE.

Associations

- *UnderlyingType* (ObjectType) – The underlying object type of the mapping pair, i.e. the object type from which the exposed type is mapped.
- *ExposedType* (ObjectType) – The exposed type of the mapping pair, i.e. the object type to which the underlying type is mapped.

6.3.6 Primitive

Instances of this class represent primitive data types in a system (e.g. a C++ char or int). Primitives are generally implemented directly by a system rather than being abstractly defined.

Specializes

- ObjectType

6.3.7 Structure

Structure defines a structured data type (e.g., a C++ struct).

Specializes

- ObjectType

6.3.8 TypeSet

The set of object and data types for a specific system or application, for example, a relational database system or programming language.

Specializes

- Namespace (from UML)

Attributes

- *Version* (String) – The version identifier of the specified object type set. For example, this might differentiate between the data types supported in ODBC 2.0 and 3.0.

Associations

- *Types* (ObjectType) – The collection of data types that make up the set.

6.3.9 Union

Union describes a union data type (e.g., a C++ union).

Specializes

- ObjectType

Associations

- *SwitchType* (ObjectType) – The type of the switch for the union.
- *Members* (UnionMember, derived from UML:Classifier.feature) – The members of the union.

6.3.10 UnionMember

UnionMember describes the members of a union.

Specializes

- MemberVariable (from UML Extensions)

Attributes

- *MemberCase* (String) – Defines the value of the union switch that selects this member.
- *IsDefault* (Boolean) – Indicates whether or not this is the default member of the union. The default is FALSE.

6.4 OIM 1.0 Compatibility

This section describes classes of the Common Data Types package required for OIM version 1.0 compatibility.

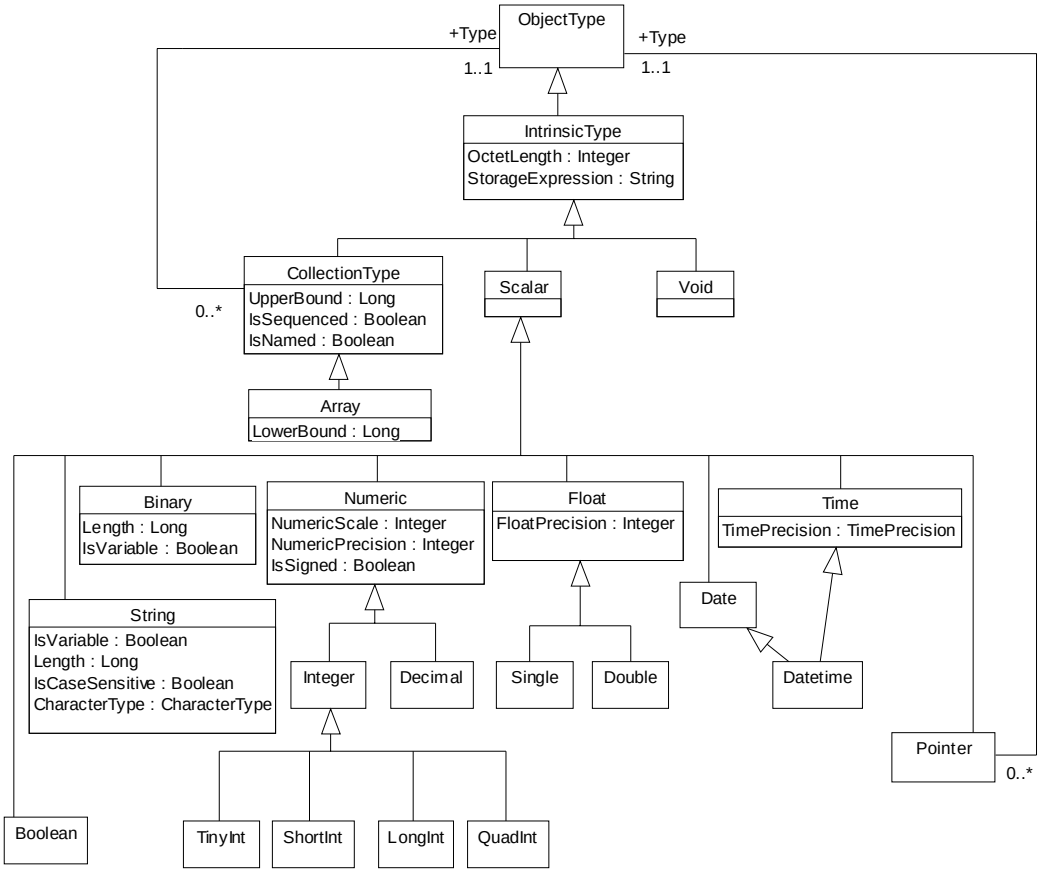


Figure 16: Common Data Types (OIM 1.0 compatibility)

6.4.1 Array

Array is supported by all datatypes whose values are arrays of objects (i.e., sequenced, indexed collections).

Specializes

- CollectionType

Attributes

- *LowerBound* (Long) – The lower bound of the array.

6.4.2 Binary

Binary describes a binary large object datatype. This includes such types as memo and unbounded text.

Specializes

- Scalar

Attributes

- *Length* (Long) – The maximum length of the blob (in bytes).
- *IsVariable* (Boolean) – Indicates if the blob value may be of a variable length. The default is TRUE.

6.4.3 Boolean

Defines a *Boolean* data type. A Boolean is any type that defines only two possible values - TRUE and FALSE.

Specializes

- Scalar

6.4.4 CharacterType

An enumeration whose values indicate the type of the character set used in a string.

Values

- CHARACTER_TYPE_SINGLE_BYTE = 0
- CHARACTER_TYPE_ DOUBLE_BYTE = 1
- CHARACTER_TYPE_MULTI_BYTE = 2
- CHARACTER_TYPE_UNICODE = 1

6.4.5 CollectionType

CollectionType is supported by all datatypes whose values are collections of objects.

Specializes

- IntrinsicType

Attributes

- *UpperBound* (Long) – The upper bound of the collection.
- *IsSequenced* (Boolean) – Indicates whether collections of this type are sequenced. The default is TRUE.

- *IsNamed* (Boolean)- Indicates whether a name can be applied to the membership of elements in collections of this type. The default is FALSE.

Associations

- *Type* (ObjectType) – The type of object contained.

6.4.6 Date

Defines a date data type. This does not include time.

Specializes

- Scalar

6.4.7 Datetime

Datetime describes a combined date time datatype, encapsulating Date and Time.

Specializes

- Time
- Date

6.4.8 Decimal

Decimal describes an exact decimal data type. This differs from Float, as float is an approximate value and Decimal is exact.

Specializes

- Numeric

6.4.9 Double

Double describes a signed, approximate, numeric value with a binary precision 53. (zero or absolute value 10[-308] to 10[308]).

Specializes

- Float

6.4.10 Float

Float describes any floating point data type of an arbitrary precision. The actual precision of an instance that supports the Float interface should be indicated by the FloatPrecision attribute.

Specializes

- Scalar

Attributes

- *FloatPrecision* (Integer) – The maximum number of base 10 digits that can be stored.

6.4.11 Integer

Integer describes a non-specific integer data type. Instances of this type should set the NumericScale property inherited from Numeric type to zero.

Specializes

- Numeric

6.4.12 IntrinsicType

An intrinsic type is one that is built into the information model

Specializes

- ObjectType

Attributes

- *OctetLength* (Integer) – OctetLength is an attribute used to specify the number of 8 bit bytes that are used in the physical storage of this data type. This value differs from length in many of the data types in that this value should take into effect any overhead involved in the storage of this data type.
- *StorageExpression* (String) – Storage Expression is a user-defined attribute describing the physical storage characteristics of the data type in question. The format of this attribute is undefined.

6.4.13 LongInt

LongInt describes a double word (4 byte) integer data type. Instances of this type should set the NumericPrecision attribute inherited from Numeric to less than or equal to 10 and NumericScale to 0. Signed and unsigned 4 byte integers are distinguished by using the IsSigned attribute inherited from Numeric.

Specializes

- Integer

6.4.14 Numeric

Numeric describes a numeric data type. The Numeric Scale and Numeric Precision values represent the scale and precision of the values of this data type rather than the scale and precision allowed by the storage mechanism. For example, a piece of information that can have the values 1-9 should have the precision for its data type set to 1 and the scale to 0, without regard to how the data is actually stored.

Specializes

- Scalar

Attributes

- *NumericScale* (Integer) – The maximum number of digits to the right of the decimal point.
- *NumericPrecision* (Integer) – The maximum number of base 10 digits that can be stored.
- *IsSigned* (Boolean) – Indicates whether or not the value of this type may be signed. The default is FALSE.

6.4.15 Pointer

Pointer describes a pointer data type. A pointer is any indirect reference to an object established by a physical address.

Specializes

- Scalar

Associations

- *Type* (ObjectType) – The type of object referenced.

6.4.16 QuadInt

QuadInt describes a quad word (8 byte) integer data type. Instances of this type should set the NumericPrecision attribute inherited from Numeric to a number less than or equal to 19 (if signed - 20 if unsigned) and the scale to 0. Signed and unsigned 8 byte integers are distinguished by using the IsSigned attribute inherited from Numeric.

Specializes

- Integer

6.4.17 TinyInt

TinyInt describes a half word (1 byte) integer data type. Instances of this type should set the NumericPrecision attribute inherited from Numeric to a number less than or equal to 3 and NumericScale to 0. Signed and unsigned 1-byte integers are distinguished by using the IsSigned attribute inherited from Numeric.

Specializes

- Integer

6.4.18 Scalar

Scalars are atomic data types used in a system. Strings and numbers are examples of scalars. This class simply acts as a classification of such types.

Specializes

- IntrinsicType

6.4.19 ShortInt

ShortInt describes a double word (2 byte) integer data type. Instances of this type should set the NumericPrecision attribute inherited from Numeric to less than or equal to 5 and NumericScale to 0. Signed and unsigned 2-byte integers are distinguished by using the IsSigned attribute inherited from Numeric.

Specializes

- Integer

6.4.20 Single

Single describes a signed, approximate, numeric value with a binary precision 24. (zero or absolute value 10[-38] to 10[38]).

Specializes

- Float.

6.4.21 String

String describes a string data type.

Specializes

- Scalar

Attributes

- *IsVariable* (Boolean) – Indicates whether or not the string value is of a variable length. The default is TRUE.
- *Length* (Long) – The maximum or defined length of the string data type (in characters).
- *IsCaseSensitive* (Boolean) – Indicates whether or not strings of this type are case sensitive. The default is FALSE.
- *CharacterType* (CharacterType) – Character type specifies the width of the character set used in the string. It is an enumeration containing values for single-byte, double-byte, and multi-byte character sets.

6.4.22 Time

Time describes a time data type.

Specializes

- Scalar

Attributes

- *TimePrecision* (Datetime) – Precision (maximum number of digits in the fractional seconds portion) of the data type.

6.4.23 TimePrecision

An enumeration whose values indicate the number of digits in the fractional seconds portion of a time quantity.

Values

- TIMEPRECISION_YEARS = 0
- TIMEPRECISION_MONTHS = 1
- TIMEPRECISION_DAYS = 2
- TIMEPRECISION_HOURS = 3
- TIMEPRECISION_MINUTES = 4
- TIMEPRECISION_SECONDS = 5
- TIMEPRECISION_TENTHS = 6
- TIMEPRECISION_HUNDREDTHS = 7
- TIMEPRECISION_THOUSANDTHS = 8
- TIMEPRECISION_TENTHOUSANDTHS = 9
- TIMEPRECISION_HUNDREDTHOUSANDTHS = 10
- TIMEPRECISION_MILLIONTHS = 11

6.4.24 Void

Void describes a void type.

Specializes

- IntrinsicType

7 Analysis and Design: Entity Relationship Modeling

7.1 Overview

The Entity Relationship Modeling package provides meta data types for ER-based modeling tools to store information about relational systems and provide a logical modeling level for physical database design tools. It is based on IDEF1X, a diagramming method originally developed by the U.S. Air Force and widely used in various governmental agencies, in the aerospace and financial industry, and supported by most database design tools. IDEF1X is a method for designing relational databases with a syntax designed to support the semantic constructs necessary in developing a conceptual schema. A conceptual schema is a single integrated definition of the enterprise data that is unbiased toward any single application and independent of its access and physical storage.

This package extends the UML package.

7.2 Semantics

Entity Relationship Diagrams consist of a few basic concepts. An *entity* specifies a type for real or abstract things that have common attributes or characteristics. Entities can be mapped in other models to deployable or physical concepts such as tables or components. Like UML, a powerful feature of IDEF1X is its support for modeling logical data types through the use of a classification structure or generalization/specialization construct. *Attributes* represent properties of instances of entities. *Keys* are collections of attributes that represent uniqueness constraints over the values of entity attributes.

A *relationship* indicates an association between entities. Relationships may have a specific set of semantics, for example cardinality or *relationship rules* which govern the deletions or changes to related entities.

7.3 Class Reference

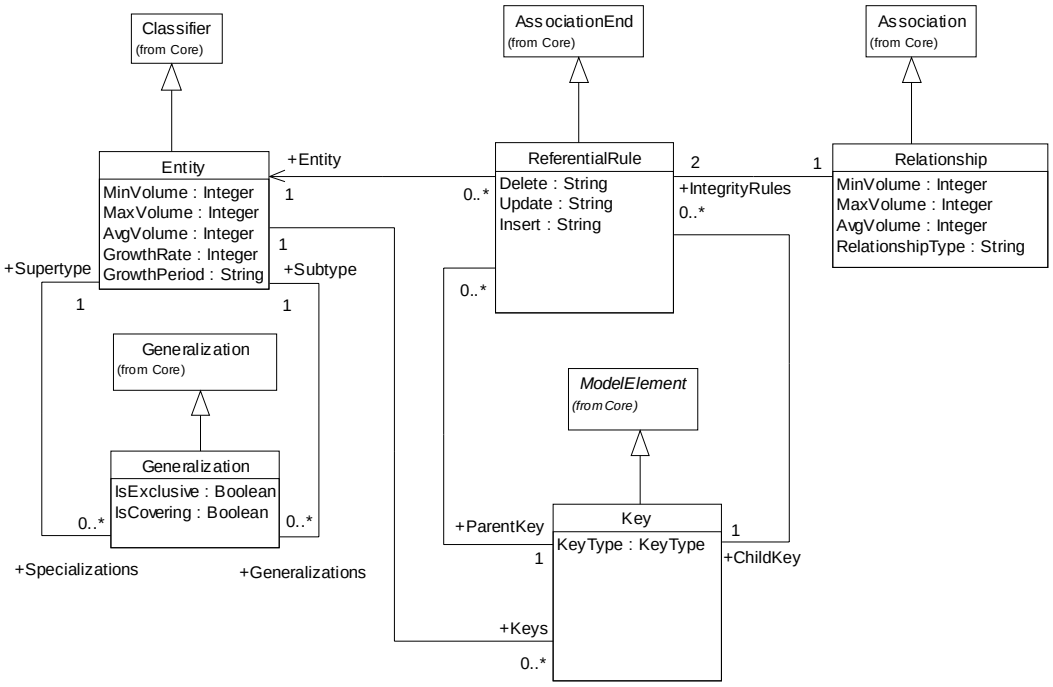


Figure 17: Entities and Relationships

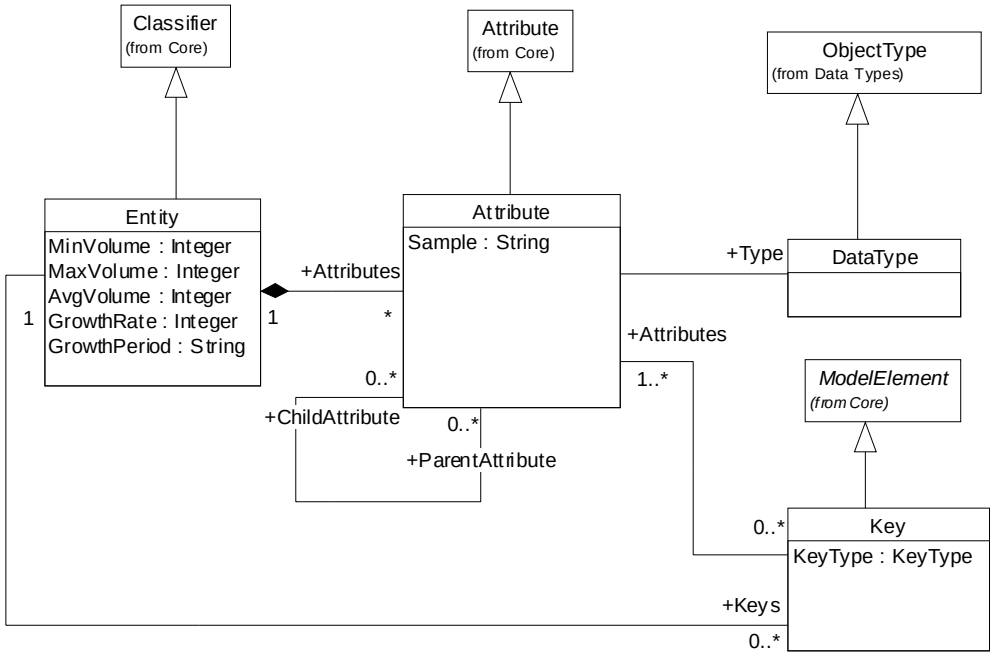


Figure 18: Attributes

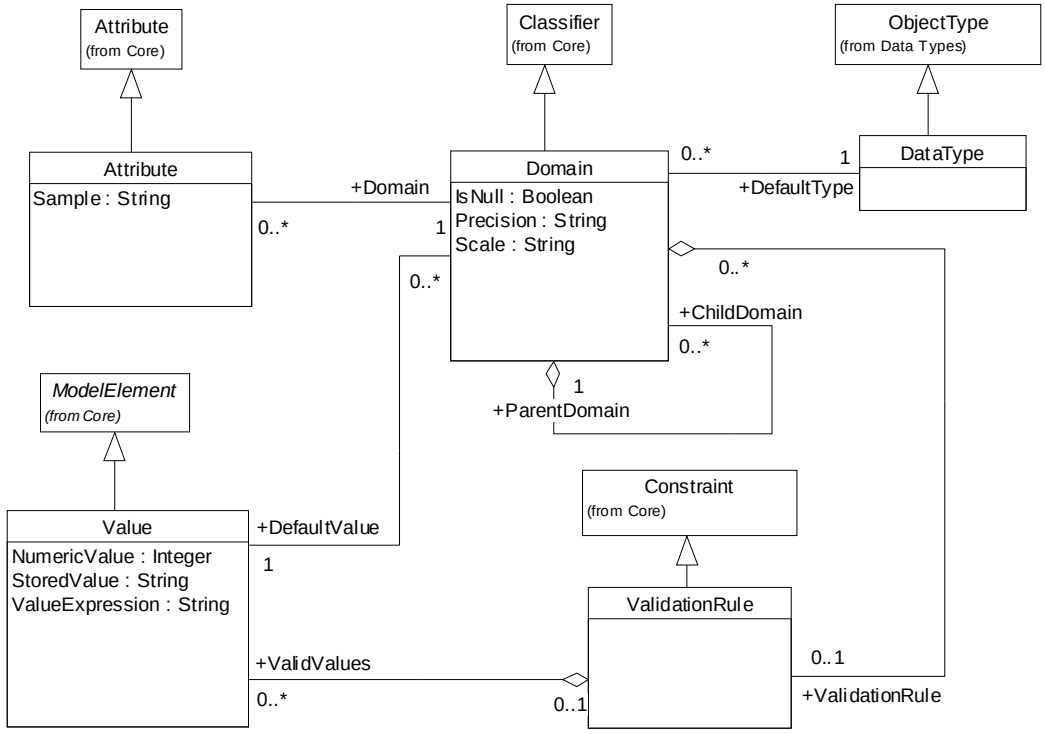


Figure 19: Domains

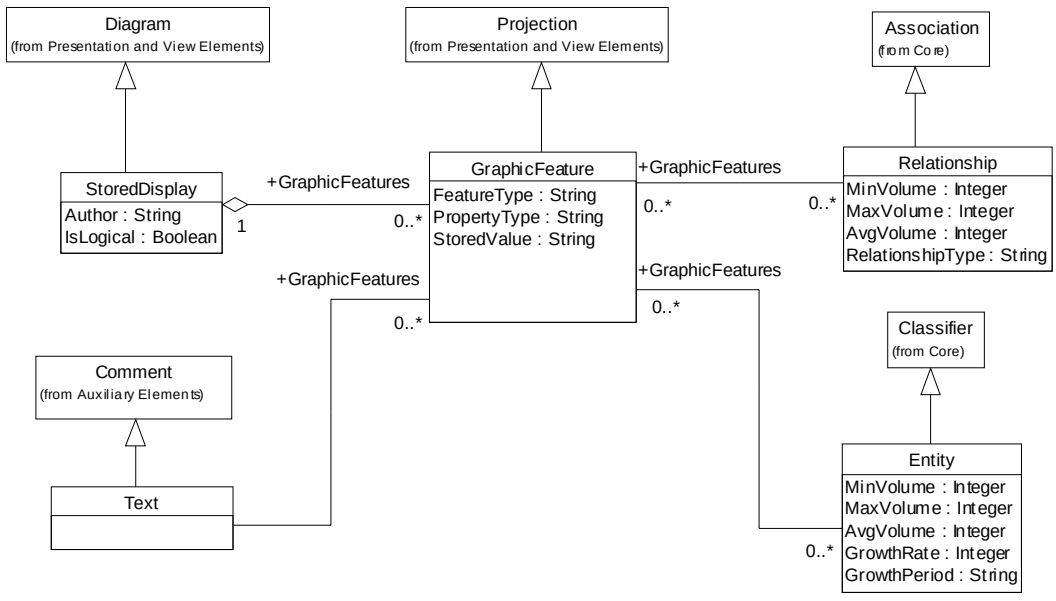


Figure 20: Diagrams

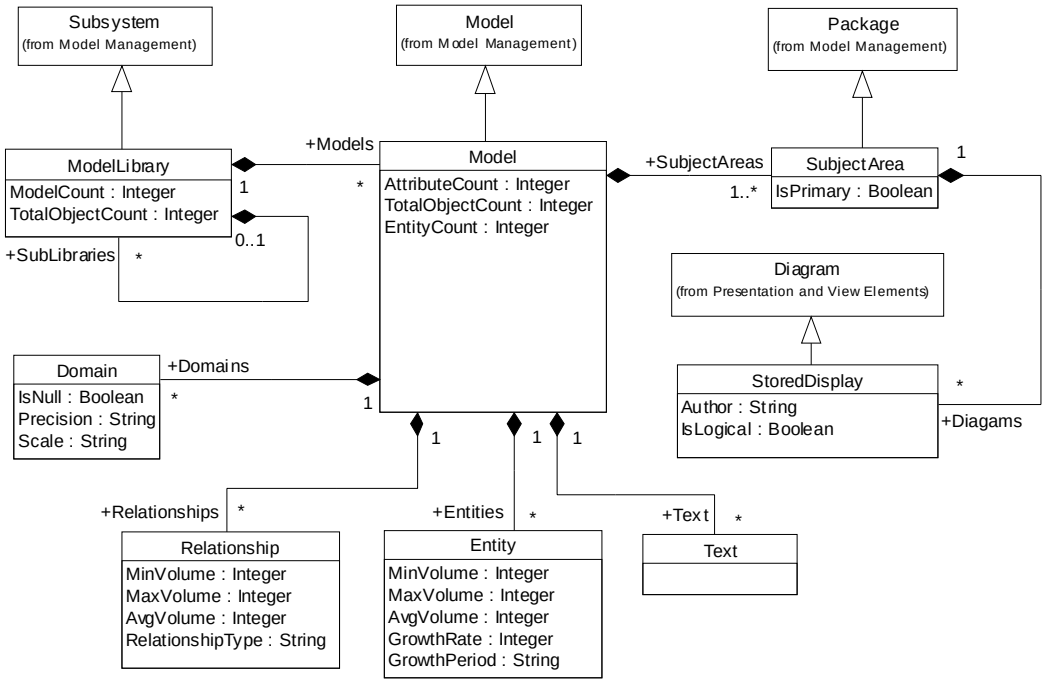


Figure 21: Model Packaging

7.3.1 Attribute

Each instance of this class describes a characteristic or property associated with a set of real or abstract things (people, places, events, etc.). The attribute "age" defined on an entity called "person" is an example of an attribute. An attribute belongs to exactly one entity or subtype.

Specializes

- Attribute (from UML)

Attributes

- *Sample* (String) – A sample value that may be contained in an instance of this attribute.

Associations.

- *ChildAttribute* (Attribute) – Links an attribute to its child attributes, i.e. the attributes that are contained with it.
- *ParentAttribute* (Attribute) – Links an attribute to its parent attribute, i.e. the attributes it inherits from.
- *Domain* (Domain, derived from UML:StructuralFeature.Type) – Links an attribute to the Domain that defines valid values for instances of the attribute.

7.3.2 DataType

Each instance of this class describes the data type associated with a particular domain or attribute.

Specializes

- ObjectType (from Common Data Types)

7.3.3 Domain

Each instance of this class describes a domain, which represents a named and defined set of attribute properties, including constraints on values the attribute can take. Each attribute is associated with exactly one domain. The domain specifies the type, default value, possible values, etc. for attributes belonging to the domain.

Specializes

- Classifier (from UML)

Attributes

- *IsNull* (Boolean) – Whether or not the attribute associated with the domain can have a null value.
- *Precision* (String) – The precision associated with the attributes belonging to the domain.
- *Scale* (String) – The scale associated with the attributes belonging to the domain

Associations

- *ChildDomain* (Domain) – Link to the domains that depend upon this domain. The values of a child domain override the values in the parent domain.
- *ParentDomain* (Domain) – Link to the domain this domain inherits from.
- *DefaultValue* (Value) – Describes the default value taken by attributes belonging to the domain.
- *DefaultType* (DomainDataType) – Describes the default data type associated with attributes belonging to the domain.
- *ValidationRule* (ValidationRule) – Describes the validation rule to be applied to attributes belonging to the domain.

7.3.4 Entity

Each instance of this class describes a set of real or abstract things (people, places, events, etc.), which have common attributes or characteristics. It can represent either a dependent or an independent entity. An example of an entity is the class of employees. Every instance of this class has common attributes like work location, title, etc. The classes, salaried employees and non-salaried employees are dependent entities which derive from the class of employees. The class of salaried employees has all the attributes of the class employees plus additional attributes like salary, etc.

Specializes

- Classifier (from UML)

Attributes

- *MinVolume* (Integer) – The minimum number of instances of the entity.
- *MaxVolume* (Integer) – The maximum number of instances of the entity.
- *AvgVolume* (Integer) – The average number of instances of the entity.
- *GrowthRate* (Integer) – The rate at which the number of instances is projected to grow.
- *GrowthPeriod* (String) – The period of time for which the number of instances is projected to grow.

Associations

- *Keys* (Key) – Whenever entities are connected by a relationship, the relationship contributes a key (or set of key attributes) to the child entity. Links an entity to the keys defined on it.

- *GraphicFeatures* (GraphicFeature, derived from UML:ModelElement.presentation) – Links an the entity to the graphic features (projections) associated with it.
- *EntityAttributes* (Attribute, derived from UML:Classifier.feature) – Describes the attributes belonging to the entity.

7.3.5 GraphicFeature

Each instance of this class describes a graphic feature, which is a representation of the graphic features of a particular object. These represent characteristics of the graphic representation of the associated object. The line color, and line type, etc. used for drawing a relationship is an are examples of a graphic features.

Specializes

- *Projection* (Presentation and View Elements)
- *Font* (Presentation and View Elements)
- *GraphicElement* (Presentation and View Elements)
- *LineContainer* (Presentation and View Elements)
- *MultiplicityProjection* (Presentation and View Elements)

Attributes

- *FeatureType* (String) – The type of feature represented; i.e. whether it's a relationship, entity, etc.
- *PropertyType* (String) – The type of property represented. For example, line color, line width, etc.
- *StoredValue* (String) – The actual value of the property represented.

7.3.6 Key

Each instance of this class describes a key, which is a set of one or more attributes that identifies an instance of the entity associated with it. In addition to a simgle primary (or unique) key, entities can have alternate keys that also uniquely identify the entity, but are not used for describing relationships with other entities.

Specializes

- ModelElement (from UML)

Attributes

- *KeyType (*KeyType) – Identifies the type of key.

Associations

- *Attributes* (Attributes) – Describes the set of attributes that comprises the key.

Constraints

- Only a single key per entity should be designated as the primary key.

7.3.7 KeyType

Identifies the type of key associated with the entity.

Values

KEYTYPE_PRIMARY = 1	The key uniquely identifies an instance of the entity.
KEYTYPE_ALERNATE = 2	If an entity has more than one unique key, all

	unique keys not selected as the primary key are described as alternate keys.
KEYTYPE_NON_UNIQUE = 3	Does not uniquely identify an instance of an entity, but are often used to access instances of entities. Non-unique keys may be mapped to indexes in a relational database.
KEYTYPE_FOREIGN = 4	Identifies the primary key attributes of a parent entity contributed to a child entity across a relationship.

7.3.8 Model

Each instance of this class describes a model, which is a logical collection of entities and the relationships between them. Models are top-level constructs, that is, elements cannot be associated across models.

Specializes

- Model (from UML)

Attributes

- *AttributeCount* (Integer) – The number of attributes of the objects described in the model.
- *TotalObjectCount* (Integer) – The total number of objects represented in the model.
- *EntityCount* (Integer) – The number of entities described in the model.

Associations.

- *Domains* (Domain) – The domains defined for the model.
- *StoredDisplays* (StoredDisplay) – The stored displays associated with the model.
- *SubjectAreas* (SubjectArea) – The subject areas associated with the model.
- *Entities* (Entity) – The entities contained in the model.
- *Relationships* (Relationship) – The relationships contained in the model.
- *Text* (Text) – The textual annotations contained in the model.

7.3.9 ModelLibrary

Each instance of this class describes a model library, which is a collection of models.

Specializes

- Subsystem (from UML)

Attributes

- *ModelCount* (Integer) – The number of models contained in the library.
- *TotalObjectCount* (Integer) – The number of objects contained in all the models contained in the library.

Associations.

- *Models* (Model, derived from UML:Namespace.ownedElement) – The collection of models in this library.
- *SubLibraries* (ModelLibrary, derived from UML:Namespace.ownedElement) – The collection of libraries nested in this library.

7.3.10 Relationship

Each instance of this class describes a relationship, which represents connections, links or associations between entities.

Specializes

- Association (from UML)

Attributes

- *MinVolume* (Integer) – The minimum number of instances of the relationship
- *MaxVolume* (Integer) – The maximum number of instances of the relationship
- *AvgVolume* (Integer) – The average number of instances of the relationship
- *RelationshipType* (RelationshipType) – Describes the specifics of the relationship between the Entities.

Associations.

- *IntegrityRules* (ReferentialRule, derived from UML:Association.connection) – Describes the integrity rules associated with the relationship.
- *GraphicFeatures* (GraphicFeature, derived from UML:ModelElement.presentation) – Describes the graphic features (projections) associated with the relationship.

7.3.11 RelationshipRole

Each instance of this class describes the role and entity plays in a relationship, which can be used to enforce a referential integrity constraint.

Specializes

- AssociationEnd (from UML)

Attributes

- *Delete* (String) – Describes the action associated with deletion of an instance of the associated entity.
- *Update* (String) – Describes the action associated with associated with update of an instance of the associated entity.
- *Insert* (String) – Describes the action associated with associated with instantiation of an instance of the associated entity.

Associations.

- *ParentKey* (Key) – The key that forms the parent of the referential rule.
- *ChildKey* (Key) – The key that depends on the parent key.
- *Entity* (Entity, derived from UML:AssociationEnd.type) – The entity which is participating in the relationship.

7.3.12 RelationshipType

This enumeration describes the possible types of Relationships between Entities.

Values

RELTYPE_IDENTIFYING = 1	A relationship whereby an instance of the child entity is identified through its association with a

	parent entity. The primary key attributes of the parent entity become primary key attributes of the child.
RELTYPE_NONIDENTIFYING = 2	A relationship whereby an instance of the child entity is not identified through its association with a parent entity. The primary key attributes of the parent entity become non-key attributes of the child.
RELTYPE_MANYTOMANY = 3	A relationship where multiple instances of the child entity are related to multiple instances of the parent entity.
RELTYPE_COMPLETESUBTYPE = 4	A subtype relationship (also known as a categorization relationship) is a relationship between a subtype entity and its generic parent. If every instance of the generic parent is associated with one subtype, then the subtype is complete.
RELTYPE_INCOMPLETESUBTYPE = 5	A subtype relationship where instances of the generic parent are not associated with at least one subtype.
RELTYPE_DERIVED = 6	A relationship between entities that is derived from another relationship in the model (used for view relationship).

7.3.13 StoredDisplay

Each instance of this class describes a stored display, which is a graphical presentation of a subject area or model that highlights a particular aspect of the total data structure. A stored display can include objects in a other stored display, but the objects may be positioned differently.

Specializes

- Diagram (from UML Extensions)

Attributes

- *Author* (String) – The author of the display.
- *RelationshipLineType* (String) – The type of line used to represent relationships.
- *IsLogical* (Boolean) – Whether or not the display is represents a logical or physical model.

Associations

- *GraphicFeatures* (GraphicFeature, derived from UML:Namespace.ownedElement) – Describes the graphic features contained in the stored display.

7.3.14 SubjectArea

Each instance of this class describes a *subject area*, a named, manageable and meaningful subset of a model that may include all the entities, relationships, subtypes and diagrams, or any subset of the objects in the complete model.

Specializes

- Package (from UML)

Attributes

- *IsPrimary* (Boolean) – Whether the subject area is designated as the primary one. Tools may designate one area as the main or default subset of the model.

Associations

- *StoredDisplays* (StoredDisplay) – The stored displays associated with the model.

7.3.15 SubType

Instances of this class (sometimes called *categorization relationships*) describe the generalization of an entity into a subtype and supertype. For example, a salaried employee is a specific type of employee. Subtypes are useful for expressing attributes or relationships only relevant to that subtype of the entity.

Specializes

- Generalization (Core)

Attributes

- *IsExclusive* (Boolean) – In an exclusive subtype relationship, each instance in the supertype can relate to one and only one subtype. For example, you might model a business rule that says an employee can be either a full-time or part-time employee but not both. To create the model, you would include an EMPLOYEE supertype entity with FULL-TIME and PART-TIME subtype entities and a discriminator attribute called “employee-status.”
- *IsCovering* (Boolean) - Specifies whether or not the set of subtype entities in a subtype relationship is fully defined. When false, indicates that the modeler feels there may be other subtype entities that have not yet been discovered.

Associations

- *Discriminator* (Attribute) – The value of an attribute in an instance of the generic parent determines to which of the possible subtypes that instance belongs.

7.3.16 Text

Each instance of this class describes a text object, which may be used to store text annotational entries on a stored display.

Specializes

- GraphicFeatureComment (from UML Extensions)

Attributes

- *TextString* (String) – The text contained in the field.

Associations

- *GraphicFeatures* (GraphicFeature, derived from UML:ModelElement.presentation) – Describes the graphic features (projections) associated with the relationship.

7.3.17 ValidationRule

Each instance of this class describes a validation rule, which can be constraint expressions or a list of valid values for attributes belonging to a domain. The validation rule specifies the rule that will be applied in order to verify the validity of the assigned values.

Specializes

- Constraint (from UML)

Associations

- *ValidValues* (ValidValue) – The list of valid values or value ranges the rule checks against.

7.3.18 Value

Each instance of this class describes a value that an attribute can take.

Specializes

- ModelElement (Core)

Attributes

- *NumericValue* (Integer) – The numeric value associated with the instance of the class.
- *StoredValue* (String) – A string version of the value associated with the instance of the class.
- *ValueExpression* (String) – A string expression of the value associated with the instance of the class.

8 Object and Components: Component Descriptions

8.1 Overview

Component-based development is the task of building families of product from kits of interoperable components. Component sharing and reuse has become strategic for enterprises in order to reduce cost and time to deployment. Reuse and sharing requires tracking meta data throughout the whole life-cycle of a component from specification through design and subsequent enhancements.

The Component Descriptions package defines component as “a software package that offers services through interfaces.” This is meant to capture the perspectives of a component as the unit of packaging and delivery, provider of services, and encapsulation boundary.

The Component Descriptions package covers the different component development life-cycle deliverables. It covers component specification, component implementation, and the result of construction - the component executable (or simply “component” for short).

The model is divided into three distinct layers: specification, implementation, and executable. The specification layer contains types whose purpose is to define the behavior specification of a component. The implementation layer contains types that define the implementation of a component. The executable layer contains meta data types that define the run-time characteristics or executable behavior of a component.

The current version of the model defines the specification and executable layers. Future versions will also include the implementation layer. It does not cover the supporting information that gives rise to those deliverables: the requirements of different analysis, design and implementation tools, version and configuration management tools, build tools, or component packaging and deployment concepts.

The Component Descriptions package intends to cover the various aspects of a component implementation, but will not cover the specifics of any particular programming language. For example, a component implementation may be realized using an object-oriented programming (OOP) language such as Java, Smalltalk or C++, or a 3GL, such as COBOL.

The Component Description package includes concepts derived from the following sources:

- Inter-operation Standards. OMG CORBA, Microsoft® OLE, Java/Beans.
- OOA/D Methods. In particular Catalysis, itself based on OMT and Fusion.
- The Unified Modeling Language (UML).

8.2 Semantics

This section explains the key aspects of the Component Description model. The model is generic in the sense that it captures the common aspects of a number of different component models, including COM, CORBA, and Java.

The term “component” is ubiquitous, so this section defines both its meaning in the Component Description Model and how it relates to the UML definition. Benefits of components, such as reusability and replaceability, and requirements, such as “plug-and-play,” have caused some of this lack of clarity by promoting a particular aspect of a component and its consequent requirements and demoting more general defining aspects.

The following are the most common perspectives:

- *Packaging perspective* - component as the unit of packaging, distribution, or delivery.

- *Consumer perspective* - component as the provider of services.
- *Integrity perspective* - component as a data integrity or encapsulation boundary.

All of these perspectives support the notion of *component reuse,* which is, perhaps, the least constraining requirement. The UML defines a component as:

> "a reusable part that provides the physical packaging of model elements."

This definition represents the packaging perspective, is quite general, and accommodates a number of stereotypes: application, document, file, library, web page, and table.

The Component Description Model has a tighter meaning for component, which is common to a number of component models. The model further qualifies the UML definition by adopting the consumer perspective, and defines a component as:

> "a software package which offers services through interfaces."

Components under this definition may also support the integrity perspective by allowing a component to be designated as independently creatable. This independence enables the important requirement of *component replaceability* to be achieved. The integrity perspective is a necessary condition for component replacement in that it defines a component as a software encapsulation boundary, that set of software which collectively maintains the integrity of the data it manages. An encapsulated set of data is referred to as an instance of a component, or a component object. Components that are not independently creatable may be termed "sub" components and are created through specific operations on a related component. Therefore, they cannot be replaced independently of that related component. Sub-components are still components in that they offer services through interfaces, but they do not designate an encapsulation boundary.

The packaging perspective is called out in the Component Description Model as a server and is a separate specialization of the UML component concept. A server may package many components and a component may comprise many servers.

An example of the difference between these perspectives is the application Microsoft® Excel. The packaged item is "excel.exe". This corresponds to a server and contains a number of components such as Application, Chart, and Sheet. Each of these is a component and is an encapsulation boundary. They are independently creatable components and could be individually replaced. For example, an alternative implementation of the Sheet component, which could inter-operate correctly with the application component, could be implemented without having any implementation knowledge of the application component. Within each component there are a number of sub-components that, once instantiated, behave like any other component object but are not independently replaceable. Examples of sub-components within a Sheet are Range and Cell.

The Component Description Model is divided into three distinct layers: specification, implementation, and executable. The specification layer contains classes whose purpose is to define the behavior specification of a component. The implementation layer contains interfaces that define the implementation of a component. The executable layer contains interfaces that define the run-time characteristics or executable behavior of a component. The current version of the Component Description Model specializes the specification and executable layers. Future versions will also extend the implementation layer.

Specification Layer

To understand the Component Description Model, it is instructive to examine how the different component models of CORBA and COM are modeled in UML in a generic way.

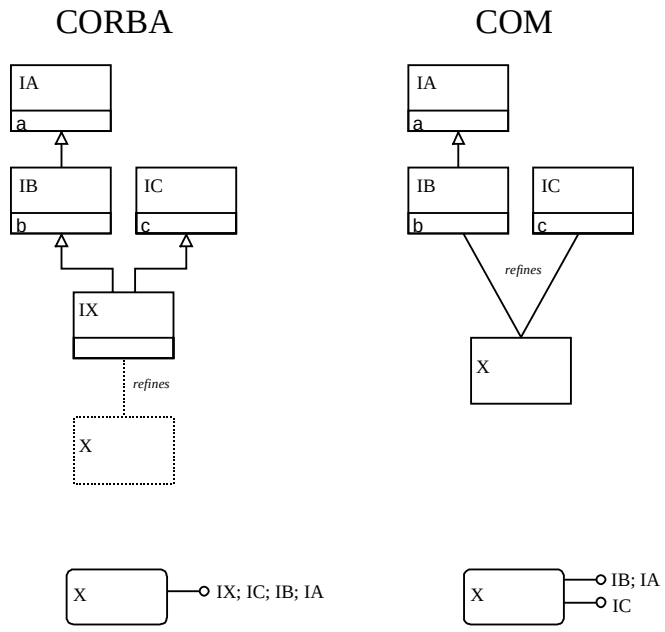


Figure 22: CORBA and COM Component Models

CORBA

CORBA specifies component behavior through interface definition language (IDL), which supports the concept of multiple interface inheritance. The total behavior of a component can therefore be defined in terms of a single interface (IX), which multiply inherits from the range of interfaces that collectively define the behavior of the component (IA, IB, IC). In this scheme there is no need for an explicit notion of component specification; a CORBA component specification is simply an interface. This is covered in the UML conceptual model by the Interface concept, which is a specialization of the type Classifier.

COM

COM supports single interface inheritance and the separate concept of a COM Class, which combines specification and executable information. A COM class therefore defines the total behavior of a component. The COM Class (X) supports the total set of behavior (IA, IB, and IC) but there is no explicit interface (IX) representing this combination. This component model is supported in the UML conceptual model by the Abstraction dependency between types. A component specification implementing a set of interfaces is seen simply as a Classifier, which abstracts other Classifiers. This refinement provides the multiple “inheritance” of behavior specification.

The Component Description Model defines explicit meta data types for the notions of component specification, interface, and the refinement relationship between them. These are called ComponentSpec, Interface, and InterfaceSupport respectively.

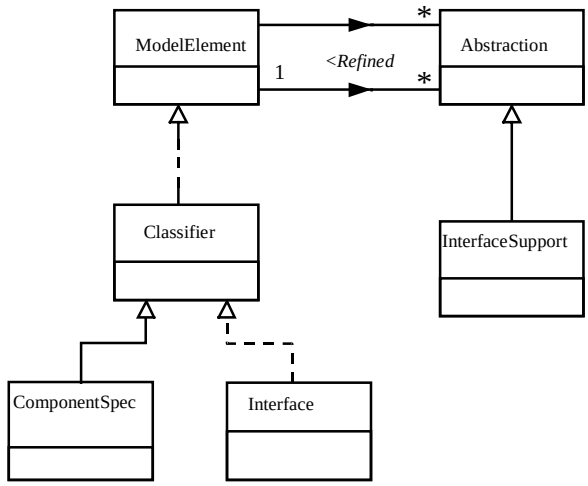


Figure 23: Component Specifications and Interfaces

The instance diagram below shows a component specification instance (CompSpecA of class ComponentSpec) supporting an interface instance (InterfaceX of class Interface) via the interface support instance (Ref_AX of class InterfaceSupport). The *refining* and *refined* relationships are shown linking the appropriate interfaces on each object. Note that in the figure below we use an implementation related Class / Interface representation to represent the MDC OIM types and type inheritance.

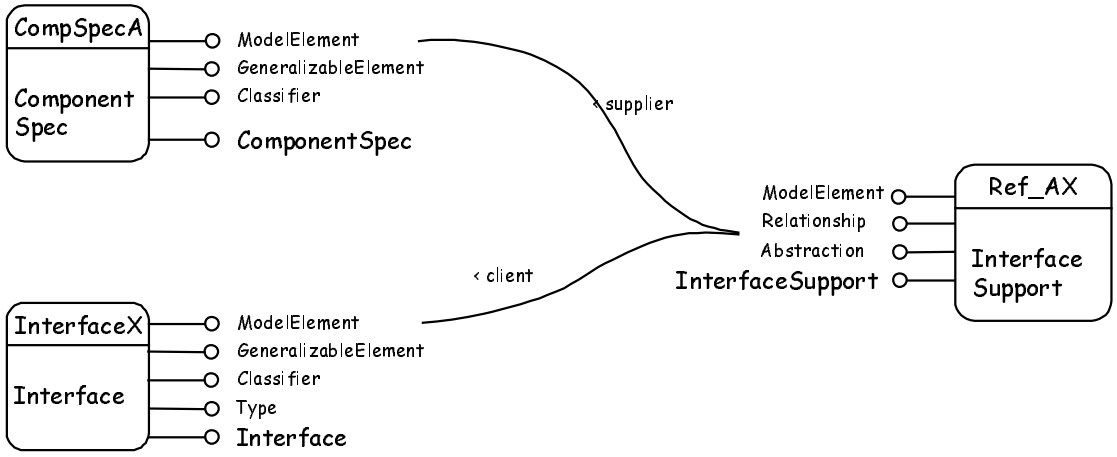


Figure 24: Instance Diagram showing component specifications and interfaces

InterfaceSupport has a property IsAlwaysSupported, which allows a component specification to distinguish between those interfaces it will always support and those that it may support only under certain conditions. Also, ComponentSpec has a property IsInterfaceSetOpen, which allows a component specification to indicate whether instances may support additional interfaces beyond those defined in the specification. In this case, the specification may also be related to those interfaces that it will *never* support under any condition. This differentiation of properties allows for flexible components. Though the behavior of these components may vary at run time, they still capture as much information as is optimal in their specification.

A component specification may also be associated with the interfaces that it requires from some other party. The exact nature of this dependency is not modeled.

ComponentSpec has an important Boolean property IsIndependentlyCreatable. This allows the distinction to be made between specifications of components that may be created directly by an external client and are therefore potentially independently replaceable, and those “sub” components that have identical

specification requirements but may only be created through specific operations on a related component and therefore can neither be created independently of that related component nor independently replaced.

Fundamentally, a component specification is a set of interfaces. Each interface represents a certain aspect of the behavior of a given component. However, for complex behavior-rich components the number of interfaces involved may become large, and the need arises to categorize and define constraints on behavior at a higher level than a single interface. The Component Description Model provides two meta data types for categorizing behavior: ComponentCategory and ComponentType.

ComponentCategory allows a category to be associated with the component specifications that implement it. A category may simply be some definition of a capability offered by such components. Additionally, a category may impose constraints over the set of interfaces that such components may support. ComponentType allows a category to define a set of mandatory interfaces, optional interfaces or disallowed interfaces, and to designate the events raised by compliant components. If a component specification implements such a category, then it must comply with the constraints defined by it. The component specification is still associated with the full set of interfaces it supports.

An important requirement when modeling with interfaces is the ability to specify the constraint that support for one interface (by a component) implies support for another interface. This is the same as the constraint placed on an implementing component by a child and parent interface in an inheritance hierarchy (if the component supports the child interface it must support the parent), but it does not have the additional requirement that one interface inherits from the other. This implication relationship is defined with the dependency InterfaceImplication.

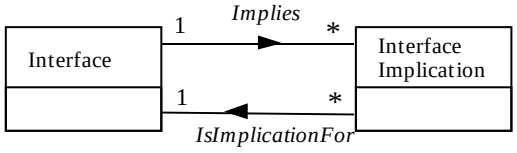


Figure 25: Interface Implication

The Component Description Model defines a class Type. Type is provided as an extension of the UML Classifier and represents a type used in the specification of an interface. These types are called specification types. Interface in the Component Description Model is an extension of Type and represents an interface, a type that defines (part of) the behavior of a component.

An Interface consists of a set of members and a specification type model. A specification type model is the set of specification types that support the definition of Interface behavior in terms of constraints and operation pre- and post-conditions (see below). They represent the vocabulary of an interface, the language in which its members and constraints are described. A specification type model is modeled as a TypeLibrary, which generalizes the UML Package. A Package may own or reference any UML ModelElement. However, if a TypeLibrary is acting as the specification type model of an interface (that is, its relationship to Interface exists), then it is constrained to own or reference only elements that specialize the Type class of the Component Description Model and constraints (from UML).

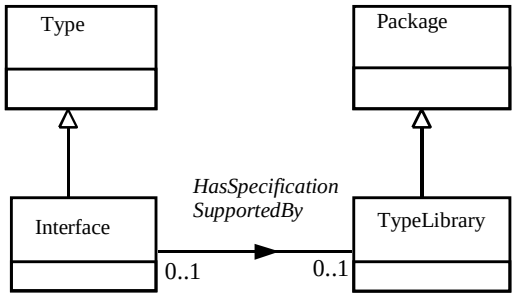


Figure 26: Specification Type Models

The figure below gives an example of the specification type model of a book library interface. The interface has a number of operations (CreateMember, CheckOutCopy, and so on). It has a supporting type library containing types of the Component Description Model such as Member, Reservation, Copy, Title, and so on. These types, and their attributes and association ends, are used to specify the effect of each operation as explained before.

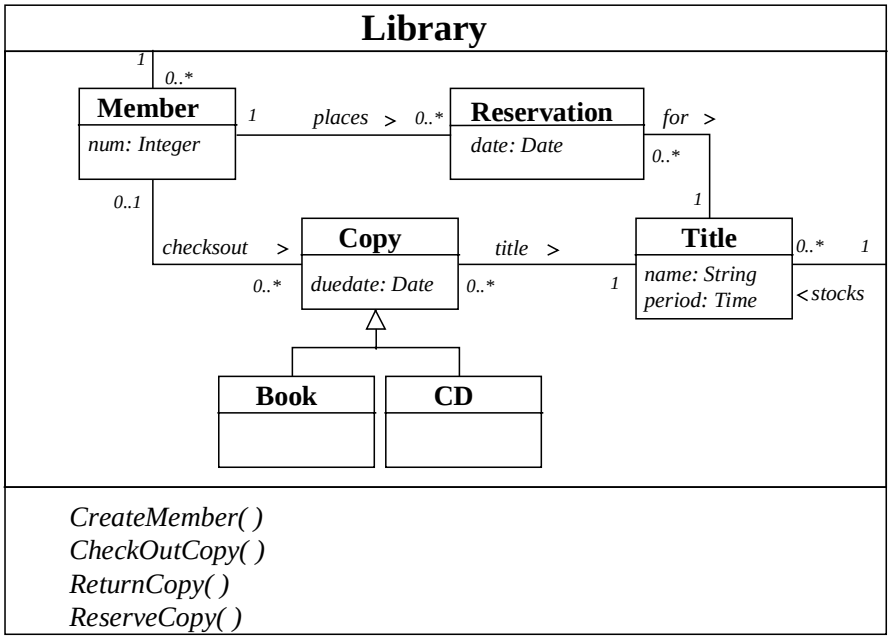


Figure 27: Library Interface Example

A type, and hence an interface, consists of a set of members. This structure is provided at the UML level by the relationship between a classifier and its structural features. UML features may be operations or attributes and these are specialized at the level of the Component Description Model with Operation and Attribute respectively.

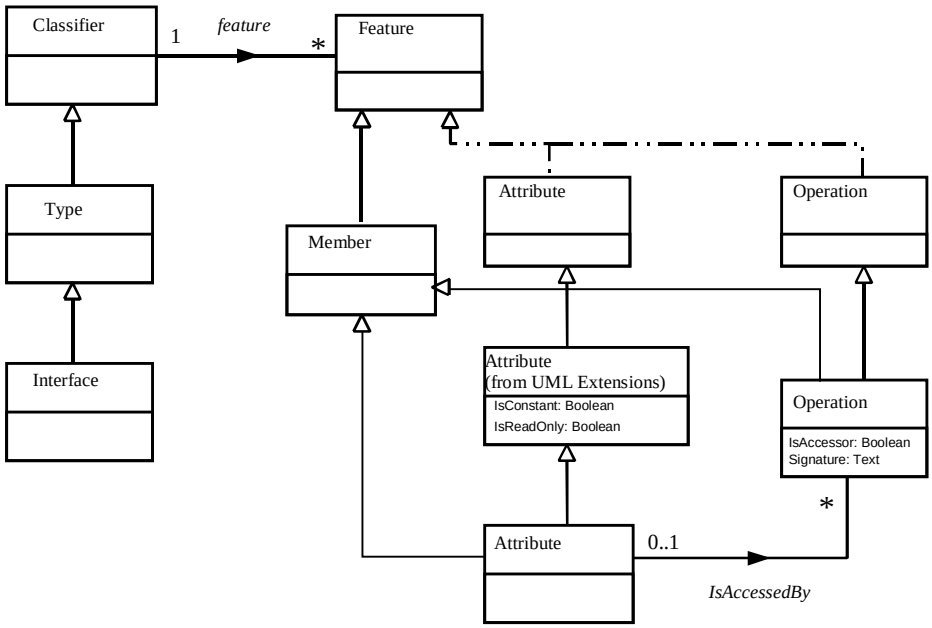


Figure 28: Attributes and Operations

Attributes may be “specification-only” in that they are defined on specification types to support the definition of constraints and operation pre-and post-conditions (for example, Title:period in the library example above). An attribute that is defined on an interface (as opposed to only a specification type) is either a constant or simply an abstraction of a get/put operation. Such attributes are represented by types that inherit from the Component Description Model type Attribute, allowing them to be associated with their accessor operations. An operation has an IsAccessor property that indicates whether or not it is an attribute accessor. If it is, then it is classified as either a put, get, or put by reference accessor via its AccessorKind property. This also means that attributes can be parameterized via the parameters of the operations they represent. For example, the attribute Pay may be associated with the accessor Get_Pay(Grade) which returns the pay for a given grade.

Operation has a Signature property, which provides an alternative way of recording an operation signature to the full representation of parameters and types modeled at the UML level. This property may be populated in place of, or in addition to, the full details of the parameters.

As ComponentSpec also inherits from Classifier, component specifications may also contain attributes and operations. Such attributes include member variables (e.g., a field). Hence, the notion of attribute in UML is used to cover constants; member variables; and, by inheriting from the Component Description Model type Attribute, abstractions of a get/put operation.

The Component Description Model type Operation further extends the UML Operation with exceptions and the notion of a related set of pre- and post-condition pairs. Each pre-condition/post-condition pair details one aspect of the effect of that operation. A pre-condition defines a condition that must hold, prior to execution, for its corresponding post-condition to hold. The pre- and post-conditions are defined in terms of attributes and association ends on the specification types of the interface.

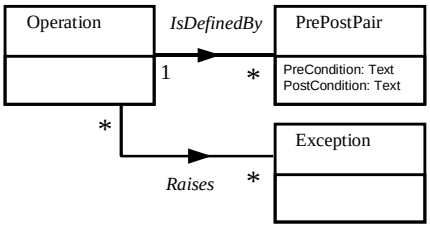


Figure 29: Pre/Post Condition Pairs and Exceptions

Continuing with the library interface example above, here is an example of a pre-condition/post-condition pair for the CheckOutCopy operation:

CheckOutCopy (in t: Title, in m: Member, out c: Copy)

pre *The member belongs to the library and a copy of the title is available*

(m.library≠NIL) ^ (∃ c • c∈ t.copy ^ c.checkedout=NIL)

post *The copy is checked out to the member for a given period*

∃ c • c∈ t.copy ^ c.checkedout=NIL

(m.checksout += c) ^ (c.duedate = TODAY + t.period)

Operations may raise exceptions. This is provided for by the generic type Exception and its relationship to Operation. It is a placeholder for technology-specific extensions that will provide the mechanism for defining exceptions within a particular component model.

For any given interface, there may be a specification type model as described above. As a specification convenience, an operation may be factored onto a type within the specification type model of its interface. This may occur when the operation concerns a particular (specification) instance of the type. By factoring the operation onto that type, its specification can be simplified: A parameter identifying that instance can be omitted, and any pre- and post-conditions and constraints can be simplified by avoiding quantification of

that instance within those expressions. Note that this is simply a technique for simplifying the specification of operations. It does not imply that the specification type is an interface and that the factored operation is an operation of that interface. It remains an operation of the original (outer) interface. However, it may anticipate a potential design decision to implement the operation in that way.

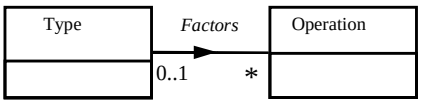


Figure 30: Operation Factoring

The Component Description Model defines a specific meta data type called EventSourceSpec for objects that are the source of events. This allows component specifications, interfaces, and other objects to be event sources by specializing this type. In COM, component specifications (Com class) are event sources. In Corba, interfaces are event sources.

The Component Description Model covers one particular event scheme, where an event is represented as an operation on an interface, and where the parameters of the operation provide data about the event. Many events can be defined on a single interface. The events can be raised in different ways:

- In *push* models, the interface defining the events is implemented on objects other than the one that raised the event. The raiser then invokes an operation on those other objects as a means of signaling the event. Such consumers will have registered interest with the raiser in component model specific ways.
- In *pull* models, the event raiser implements the interface defining the events. A consumer invokes an operation on that interface to poll for the event.

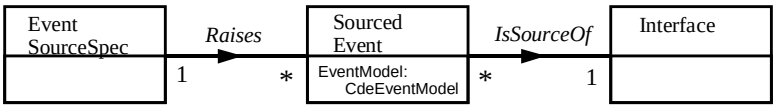


Figure 31: Event Modeling

To model both push and pull models, a meta data type called SourcedEvent is defined, which represents the relationship between an event source and the interfaces defining the events it raises. The property EventModel indicates whether a push or a pull scheme is being used in each case. The default event model is “Push.” An object can source an event using either, or both, schemes. Alternative schemes for the same interface will require separate sourced event objects, one for each scheme.

Executable Layer

The Component Description Model type Component is introduced as an extension to the UML Component and has a tighter meaning than in UML. MemberExe augments the UML Feature and the Component Description Model type Member with some executable level attributes. ExecutionPerformance provides information on the performance characteristics of the member. ResourcesNeeded describes the run-time resources consumed by the member. ExecutionDetails is an uninterpreted string allowing the designation of technology-specific information that may be needed in order to invoke the member. Different technology extensions of the Component Description Model will have different conventions for the content of this attribute.

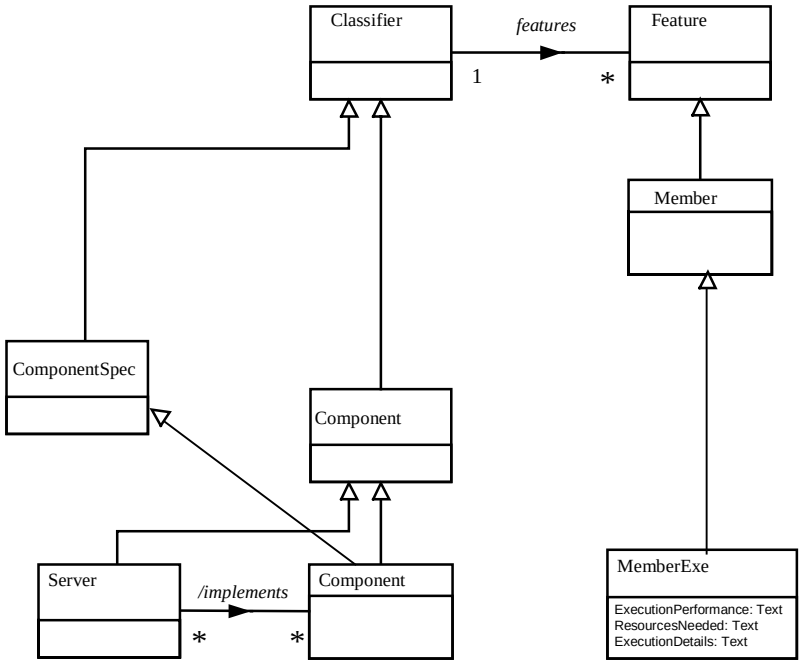


Figure 32: Executable Layer

The meta data type Server inherits from the UML Component type and represents a physical packaging of functionality consistent with the UML definition of component. A Server may package one or more Components through the association implements. This is a many-to-many relationship that allows many components to be packaged into a single server. It also allows a single component to comprise many servers. Server also has a many-to-many <IsDescribedBy> association with Typelibrary, allowing a server to be associated with one or more type libraries that describe the components it implements.

The form this physical packaging often takes is either an executable load module or a dynamic link library (DLL). These concepts are defined in the UML Extension model as Application and Library and are realizations of the UML <<application>> and <<library>> stereotypes of component, respectively.

8.3 Class Reference

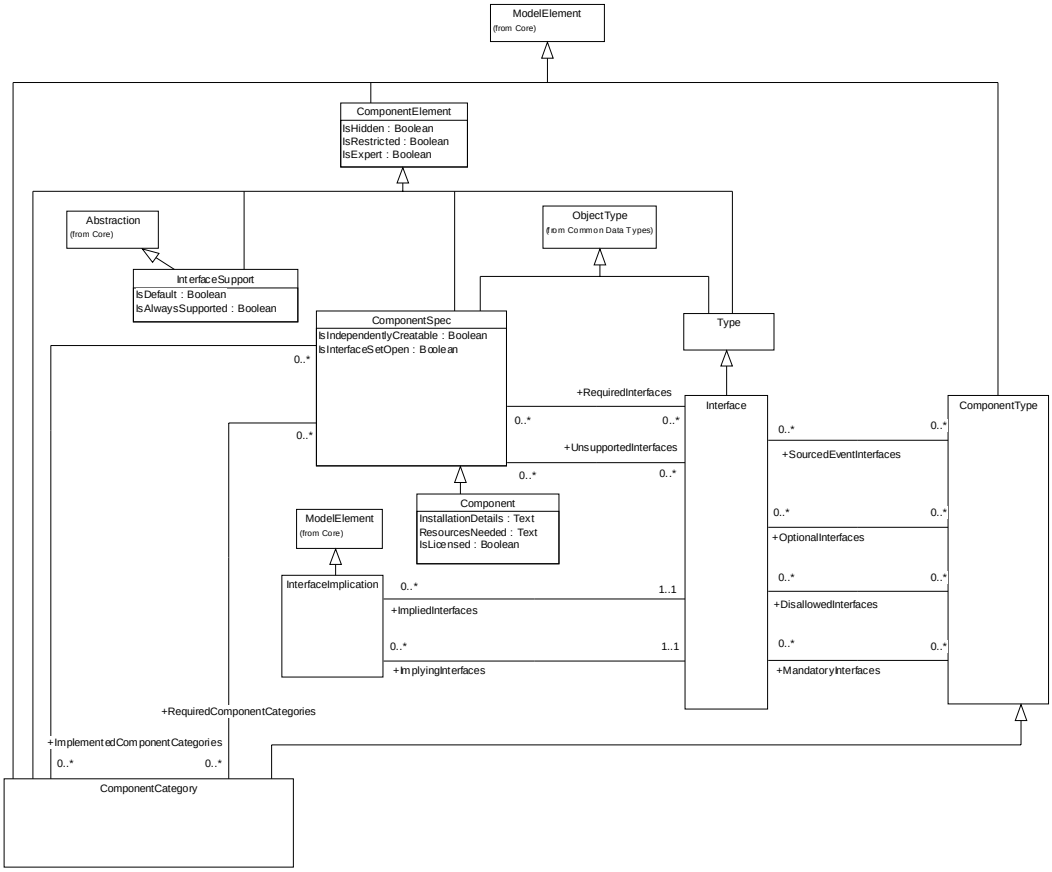


Figure 33: Component Specification

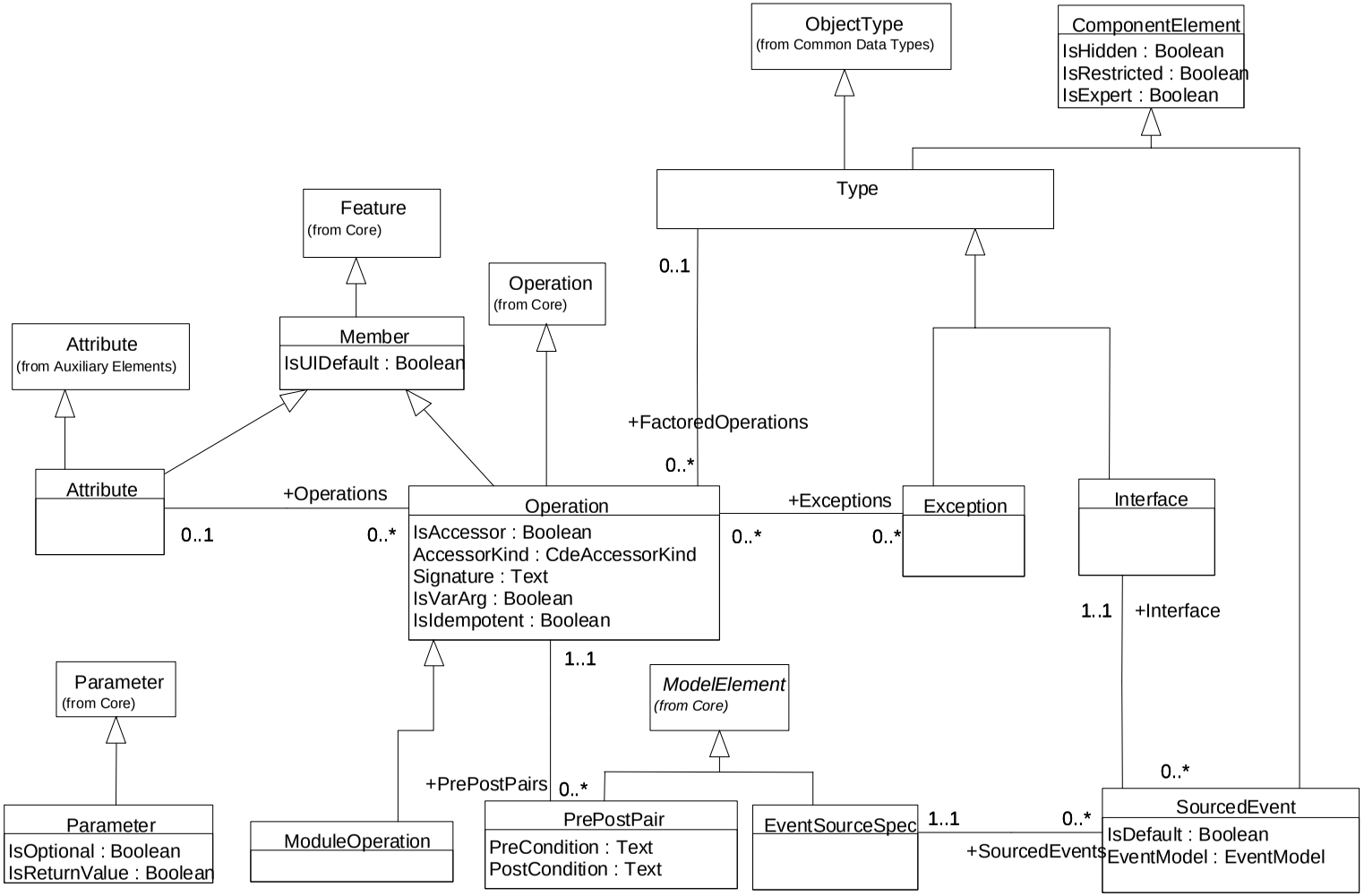


Figure 34: Features and Events

This page is intentionally blank.

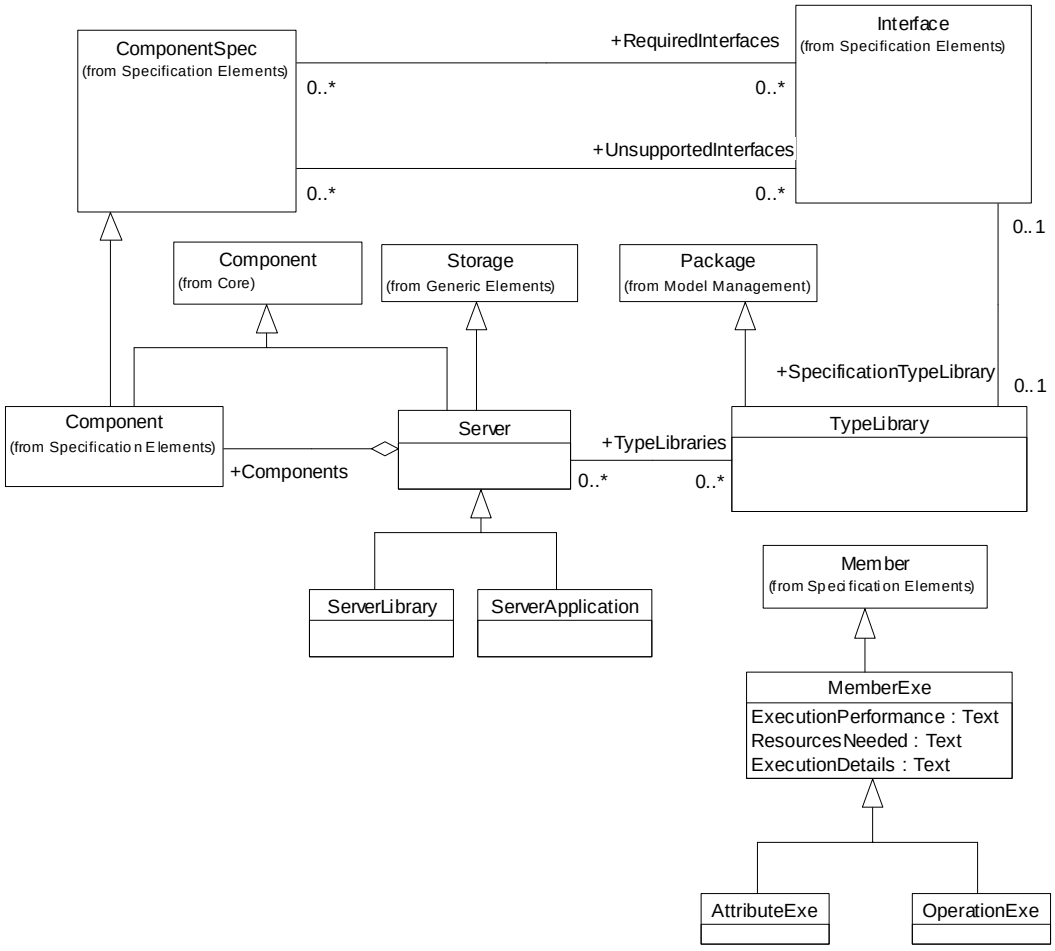


Figure 35: Execution Elements

8.3.1 AccessorKind

An enumeration whose values indicate whether an accessor operation is a get, a put-by-value, or a put-by-reference.

Values

- ACCESSOR_KIND_GET = 2
- ACCESSOR_KIND_PUT = 4
- ACCESSOR_KIND_PUTBYREF = 8

8.3.2 Attribute

Each instance of this class describes an attribute, allowing an attribute to be an abstraction of operations that access it.

Specializes

- Attribute (from UML Extensions)

- Member
- ComponentElement

Associations

- *Operations* (Operation) – The set of accessor operations (that Get, Put or PutRef the value).

8.3.3 AttributeExe

Each instance of this class describes an attribute on a binary or run-time component. This class allows execution information to be recorded for the attribute, if required.

Specializes

- MemberExe
- Attribute (from UML Extensions)
- ComponentElement

8.3.4 Component

Each instance of this class describes a binary or run-time component.

Specializes

- Component (from UML Extensions)
- ComponentSpec

Attributes

- *InstallationDetails* (String) – A natural language description of how to install the component.
- *ResourcesNeeded* (String) – A natural language description of the run-time resources consumed by the component.
- *IsLicenced* (Boolean) – Indicates that the component is licensed. Only clients that are authorized to use them can create licensed components. The default is FALSE.

8.3.5 ComponentCategory

Each instance of this class describes a component category, which identifies a set of functionality that a component either implements or requires. A component may implement or require many categories, and a category may be implemented or required by many components. Categories are not structured and do not combine together to form hierarchy.

Specializes

- ComponentType
- ComponentElement

8.3.6 ComponentElement

Each instance of this class describes a component description element, providing general information relevant to various sorts of objects.

Specializes

- ModelElement (from UML)

Attributes

- *IsHidden* (Boolean) – Indicates that the element exists, but should not be displayed in a user-orientated browser. The default is FALSE.
- *IsRestricted* (Boolean) – Indicates whether macro/scripting programmers should be prevented from using the element. The default is FALSE.
- *IsExpert* (Boolean) – Indicates that the element is intended for expert users only. The default is FALSE.

8.3.7 ComponentSpec

Each instance of this class describes the specification aspects of a component.

Specializes

- ObjectType (from Common Data Types)
- ComponentElement

Attributes

- *IsIndependentlyCreatable* (Boolean) – Indicates that the component can be created directly, and independently of any other component. If the component is not independently creatable, then it must be created by another component. The default is TRUE.
- *IsInterfaceSetOpen* (Boolean) – Indicates whether the interface set is open. If so, then instances may support additional interfaces beyond those associated with the specification. The default is FALSE.

Associations

- *RequiredInterfaces* (Interface) – The set of interfaces that the component specification requires.
- *UnsupportedInterfaces* (Interface) – The set of interfaces that are not supported by any instance of the component specification.
- *RequiredComponentCategories* (ComponentCategory) – The set of categories required by this component specification.
- *ImplementedComponentCategories* (ComponentCategory) – The set of categories implemented by this component specification.

8.3.8 ComponentType

Each instance of this class defines a component type that identifies a set of functionality implemented by a component. It is similar to component category, except ComponentType provides detailed information about the set of interfaces that components conforming to this type will definitely support, may support, or must not support, as well as information about the events raised by such components.

Specializes

- ComponentElement

Associations

- *MandatoryInterfaces* (Interface) – The set of interfaces that must be supported by components compliant with this type.
- *OptionalInterfaces* (Interface) – The set of interfaces that may optionally be supported by components compliant with this type.

- *DisallowedInterfaces* (Interface) – The set of interfaces that must not be supported by components compliant with this type.
- *SourcedEventInterfaces* (Interface) – The set of interfaces defining the events that must be supported by components compliant with this type.

8.3.9 EventModel

An enumeration whose values indicate whether a SourcedEvent is raised using a Push or a Pull model. The models are:

- Push Model: The source of the event will push event data to the consumer by invoking operations defined on an event interface supported by the consumer. Consumers will register interest in the event component model in specific ways.
- Pull Model: The consumer interested in the event will pull event data from the source, by invoking "polling" operations supported by the source.

An object can source any given event interface via either, or both, models. If both models raise an event interface, there would be two SourcedEvent objects associating the source with the event interface. The default model is Push.

Values

- EVENT_MODEL_PUSH = 1
- EVENT_MODEL_PUSH = 2

8.3.10 EventSourceSpec

Each instance of this class describes the specification of objects that are the source of events.

Specializes

- ModelElement (from UML)

Associations

- *SourcedEvents* – The set of events raised by the objects implementing this specification.

8.3.11 Exception

Each instance of this class describes an exception.

Specializes

- Type

8.3.12 Interface

Each instance of this class describes the specification information for an interface. This serves as both an implementation specification to a component builder, and a test specification to a component tester or reuser.

Specializes

- Type

Associations

- *ImpliedInterfaces* (Interface) – The set of implied interface implications.
- *ImplyingInterfaces* (Interface) – The set of implying interfaces.

- *SpecificationTypeLibrary* (TypeLibrary) – The type library containing elements that support the specification of this interface.

8.3.13 InterfaceImplication

Each instance of this class describes the association object between an interface and another interface that it implies. If I1 implies I2, no class should support I1 without also supporting I2.

Specializes

- ModelElement (from UML)

8.3.14 InterfaceSupport

Each instance of this class describes the association object between a component and an interface that it supports.

Specializes

- Abstraction (from UML)
- SummaryInformation (from Generic Elements)
- ComponentElement

Attributes

- *IsDefault* (Boolean) – Indicates that the interface is the default for the component. A component may have only one default interface. The default value is FALSE.
- *IsAlwaysSupported* (Boolean) – Indicates whether the interface is always supported by all instances of the component. Interfaces for which IsAlwaysSupported is FALSE may be supported only under certain conditions. The default is TRUE.

8.3.15 Member

Each instance of this class describes the generalization of the specification of members in an interface. The term “member” in Component Description Model does not cover member variables.

Specializes

- Feature (from UML)

Attributes

- *IsUIDefault* (Boolean) – Indicates that this is the default member for display in the user interface. The default value is FALSE.

8.3.16 MemberExe

Each instance of this class describes the run-time characteristics of a member.

Specializes

- Member

Attributes

- *ExecutionPerformance* (String) – A natural language description of the execution performance of the member. This provides information for using the component, and for selecting between candidate components for reuse.

- *ResourcesNeeded* (String) – A natural language description of the run-time resources consumed by the member. This provides information for using the component and for selecting between candidate components for reuse.
- *ExecutionDetails* (String) – A natural language description that provides any additional details necessary to invoke the member.

8.3.17 ModuleOperation

Each instance of this class describes an operation on a module.

Specializes

- ModuleOperation (from UML Extensions)
- Operation

8.3.18 ModuleOperationExe

Each instance of this class describes an operation on a binary or run-time module. This class allows execution information to be recorded for the operation, if required.

Specializes

- Operation
- ModuleOperation (from UML Extensions)
- MemberExe

8.3.19 Operation

Each instance of this class describes a Component Description Model operation.

Specializes

- Operation (from UML)
- Member

Attributes

- *IsAccessor* (Boolean) – Indicates if the operation is an accessor or an attribute. An attribute accessor either gets the value, puts the value, or puts the value by reference. The AccessorKind member is only relevant if this property is TRUE.
- *AccessorKind* (AccessorKind) – Indicates the kind of the accessor: Get, Put or PutRef. Only relevant if *ISAccessor* is TRUE.
- *Signature* (String) – The signature of the operation. This may be provided in place of, or in addition to, the full parameter details.
- *IsVarArg* (Boolean) – Indicates that the operation takes a variable number of arguments. If TRUE, then the last parameter of the operation must be an array, containing all of the remaining parameters. The default is FALSE.
- *IsIdempotent* (Boolean) – Indicates whether the operation is idempotent. An idempotent operation is one that does not modify state information and returns the same result each time it is performed. Performing the routine more than once has the same effect as performing it once. The default is FALSE.

Associations

- *Exceptions* (Exception) –The set of exceptions raised by the operation.

- *PrePostPairs* (PrePostPair) – The set of pre- and post-condition pairs that define the semantics of the operation. For operations that have no pre- and post-condition pairs, the semantics is only defined in the documentation. Pre- and post-condition pairs provide a basis for component testing because they formalize the effect of an operation in terms of apparent state changes to the object.

8.3.20 OperationExe

Each instance of this class describes an operation on a binary or run-time component. This class allows execution information to be recorded for the operation if required.

Specializes

- Operation
- MemberExe

8.3.21 Parameter

Each instance of this class describes a parameter of an operation.

Specializes

- Parameter (from UML)

Attributes

- *IsOptional* (Boolean) – Indicates that the parameter is optional. The default is FALSE.
- *IsReturnValue* (Boolean) – Designates the parameter as containing the return value for some clients. The default is FALSE.

8.3.22 PrePostPair

Each instance of this class describes a pre-condition/post-condition pair that forms part of the specification of an operation. An operation may define a number of such pairs. Each pair may detail one aspect of the effect of the operation. Pre- and post-conditions are conditions described in terms of queries on types. For a given interface, the pre- and post-conditions of its operations will reference the types that form the specification type model of the interface.

Specializes

- None

Attributes

- *PreCondition* (String) – Operation pre-condition. This is a condition that must hold true prior to the execution of the operation in order for its corresponding post-condition to be guaranteed. If a pre-condition is false this does not mean that the operation cannot, or will not execute, but simply that the corresponding post-condition is not guaranteed.
- *PostCondition* (String) – Operation post-condition. This is a condition that will hold true after the execution of the operation, provided its corresponding pre-condition held prior to the execution. The post-condition defines the guarantees of the operation. Any effect on an object that is not defined in a post-condition of an operation is not a guarantee of the operation and cannot be assumed by a client of that operation. Any dependencies on “side-effects” of operations are likely to cause failure in the client if the component providing that operation is replaced with another meeting the same specification. All effects must be documented as part of the operation via the pre- and post-conditions.

8.3.23 Server

Each instance of this class describes the server of a component. The server is the application or library that implements the component. It is associated with the components for which it is the server by the inherited "implements" relationship of UML. The physical implementation of the server may also be represented.

Specializes

- Component (from UML)
- Storage (from Generic Elements)

Associations

- *TypeLibraries* (TypeLibrary) – The set of type libraries that describe the components implemented by this server.
- *Components* (Component, derived from UML:Component.resident) – The set of components implemented by the server.

8.3.24 ServerApplication

Each instance of this class describes a server that is an application (e.g., an EXE).

Specializes

- Surrogate (from Generic Elements)
- NamedVersion (from Generic Elements)
- SummaryInformation (from Generic Elements)
- Application (from UML Extensions)
- Server

8.3.25 ServerLibrary

Each instance of this class describes a server that is a library (e.g., a DLL).

Specializes

- Surrogate (from Generic Elements)
- NamedVersion (from Generic Elements)
- SummaryInformation (from Generic Elements)
- Library (from UML Extensions)
- Server

8.3.26 SourcedEvent

Each instance of this class describes the association object between a component and the sourced event interfaces.

Specializes

- ComponentElement

Attributes

- *IsDefault* (Boolean) – Indicates that the interface is the default interface raised by the component. Upon registering for events, if no specific interface is provided then it is assumed to be default.

- *EventModel* (EventModel) – The event model (Push or Pull) by which the event is raised.

Associations

- *Interface* (Interface) – The interface describing the events raised.

8.3.27 Type

Each instance of this class defines a specification type. Specification types represent the vocabulary of an interface – the language in which its members and constraints are described. An interface contains its specification types through the specification package that may be associated with the interface.

Specializes

- Classifer (from UML)
- ObjectType (Common Data Types)
- ComponentElement

Associations

- *FactoredOperations* (Operation) – The set of operations that have been factored onto this type.

8.3.28 TypeLibrary

Each instance of this class describes a type library that contains a set of Types.

Specializes

- Package (from UML)
- ModelElement

This page is intentionally blank.

9 Database and Warehousing: Relational Database Schema

9.1 Overview

The Relational Database Schema package describes information about data maintained in the relational databases of an organization. To enable enterprise-wide data management, such metadata must be readily available in a commonly agreed-upon format for tools and applications. The goal of this package is to serve as the core model for such an infrastructure.

Additional goals of the package are to:

- Introduce an industry-standard access mechanism and infrastructure for metadata about relational data sources.
- Introduce a core model for describing metadata about data sources that enables tools to store and exchange such descriptive information.
- Enable tool vendors to extend the model to address requirements of individual tools in the context of a common core model.

The Relational Database Schema package covers the basic elements of a SQL data provider, such as tables, columns, and relationships. It does not address physical or implementation details.

The concepts in the package are modeled after the ANSI SQL-92 standard, with selected extensions supported by popular relational database vendors.

9.2 Semantics

The Database Schema Package contains three packages:

- Schema Elements – The primary package, containing classes for tables, views, queries, columns, indexes, constraints, joins, data sources, catalogs, schemas, triggers, and keys.
- Catalog and Connections – A package containing classes of interest to the client side of client/server applications, such as classes about establishing connections to database servers.
- Data Types – A package containing database-specific extensions to the data type model.

Catalogs are the top-level container for all database definitions. Following the ANSI SQL-92 standard, there is the further constraint that a catalog should only contain *schemas*, an ownership package for database components. A schema should, in turn, only contain other database components (such as tables and views). The following entity relationship diagram illustrates a simple database schema that is referenced in the accompanying text:

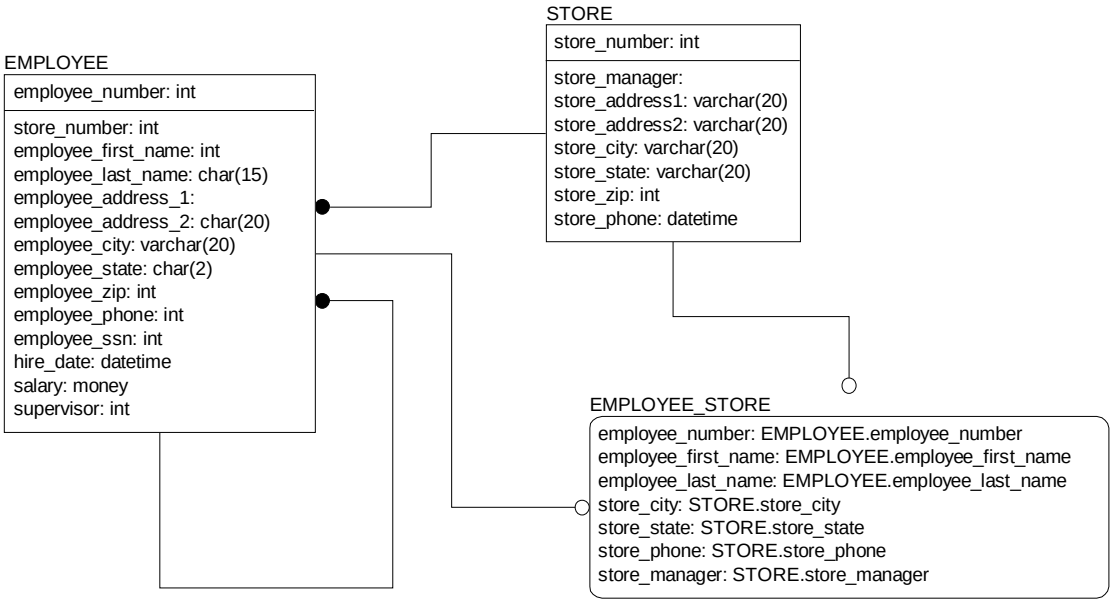


Figure 36: Sample database schema

Tables, *views*, and *queries* all exhibit table-like qualities (their definitions include a set of *columns* and their instances contain a set of *rows*. The generalization of table, view, and query is known as a *column set*. In the diagram above, the employee and store tables contain a set of columns, each with a specified *data type*. The employee_store view is defined as a query of the underlying tables.

Constraints are schema elements used to enforce the integrity of data in a database. Constraints define rules regarding the values allowed in columns and are the standard mechanism for enforcing integrity. The ANSI SQL-92 standard identifies three major types of constraints - *table constraints, Domain constraints*, and *assertion constraints.* Table constraints are further broken down into *referential constraints, unique constraints*, and *check constraints.*

A *key* describes an ordered collection of columns on a single table or view. A key may be one of the following: a *foreign key*, a *unique key* (or candidate key), or an *alternate key* (need not be unique). A key has a relationship to an associated column set (that is, a table or view) and another relationship to an ordered collection of columns (to represent a composite key). In the sample schema above, the employee_number has been designated as a unique key for the employee table.

A key may be associated with zero or more *join roles*. Each join role links the key to another key of the same or different column set. A join role identifies a key that can be used for a meaningful join with another key.

Columns are tied together between the keys on two related join roles. The column order of the two column collections (on the two keys) must be compatible, so that each column corresponds to the column in the same ordinal position in both collections.

A referential integrity constraint is represented by *referential roles*, which are specializations of join role. Each referential role identifies one of the keys that participate in a referential constraint (a unique key on one side and a foreign key on the other). Referential roles appear on each side of a referential association, rather than directly connecting to a key on one side, because update and delete rules, which are properties of a referential role, can appear on both sides of such associations. For example, some database systems allow you to define both a cascade delete from parent to children and a pendant delete from the last child to its parent. In addition, some database systems allow for a many-to-many referential association. In such cases, neither side of the association is a unique key.

In the sample schema above, there is a referential constraint specified between the employee and store tables. The store_number column on the employee table comprises a foreign key that is tied to the store_number column, the unique key on the store table.

The Database Schema package includes information about *connections* and *data sources*. The model captures enough information about a database connection to create a session with a data source (in the OLE DB sense) or server. Typically, when making a connection to a data source, the connection binds to a particular default database at that data source. Information about the data types supported by a particular DBMS product can also be modeled.

9.3 MDIS Compatibility

The Meta Data Interchange Specification (MDIS) provides concepts to support a bi-directional file based interchange of meta data while maintaining the consistency of the transferred information. The file-based transfer specification implies a hierarchical information structure.

The following illustrates the objects and relationships that define the metamodel for MDIS Version 1.0.

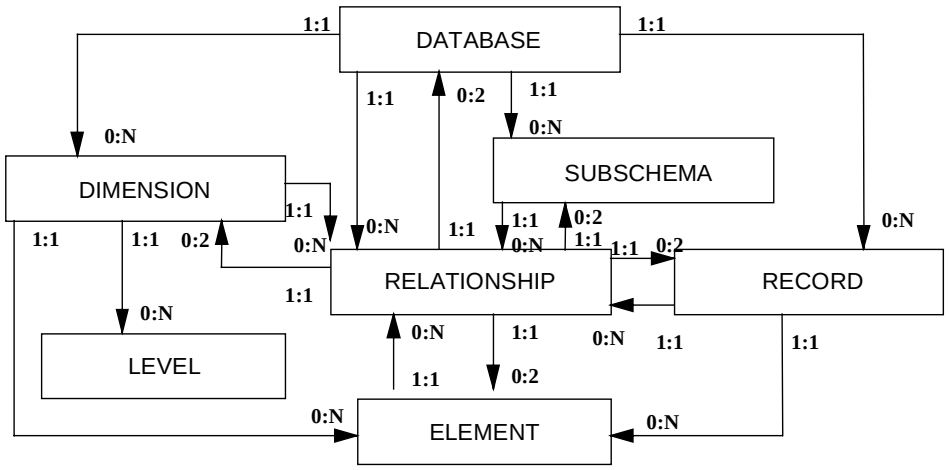


Figure 37: Meta data Interchange Specification Metamodel

This figure labels each object with a <number>:<number/variable> notation, which indicates the possible one-to-many relationships. For example, for every database, there may be as few as no records or as many as "n" records. Likewise every record can contain as few as no element types (though this is more theoretical than likely), or as many as "n" element types.

The model defines how and MDIS file is constructed by embedding each object definition within its parent prior to embedding the parent definition. So for example a database definition can directly contain Dimension, View, Record, or Subschema objects, but cannot contain Element or Level objects.

MDIS objects each have a set of well-defined properties like Identifier, ElementName, or ElementLength, which carry the description of the meta data object. Each of the MDIS objects in its file based representation is encapsulated by a BEGIN / END statement, which may contains either properties of sub-objects. Properties are the leaf-nodes of the hierarchy and simply name / value pairs.

The following example shows the structure of a simple MDIS file:

```
BEGIN HEADER
        MDISVersion “1.0”
        ...
END HEADER

BEGIN DATABASE
        Identifier “053”
        DatabaseName “CUSTOMER-ORDER-RECORD”

        BEGIN RECORD
                Identifier “054”
                RecordName “CUSTOMER-RECORD”
                RecordType “RECORD”

                BEGIN ELEMENT
                        Identifier “055”
                        ElementName “SOCIAL-SECURITY-NUMB”
                        ElementDataType “CHAR”
                        ElementLength “11”
                        ElementNulls “T”
                END ELEMENT
                ...
        END RECORD
        ...
END DATABASE
```

Figure 38: MDIS Example

The MDIS model has been integrated into the MDIC OIM by defining a mapping of the MDIS objects, relationships, and properties onto MDC OIM classes. The compatible name mappings can be found in the UML representation of the MDC OIM. The following provides an overview of the mapping:

DATABASE

A Database object in MDIS can be used to represent: a group of files, a relational database, a network database, a hierarchical database, a multi-dimensional database, or an object database.

Database is mapped onto Catalog (from Database Schema) in the MDC OIM.

SUBSCHEMA

The Subschema object in MDIS is used to provide a logical grouping of record objects that describe a meaningful subset of a database. Instances of the Relationship object (of type "CONTAINS") are used to represent the record types that belong in a particular subschema.

Subschema is mapped onto the Schema (from Database Schema) in the MDC OIM.

RECORD

The purpose of the Record object in MDIS is to provide a physical grouping of element objects that describe a unit of data.

Record is mapped on the Table (from Database Schema) or Record (from Record Oriented Schema) in MDC OIM. Note, that there is no distinction in the MDIS 1.0 specification between tables and views and therefore a record element in a relational database is always mapped onto Table.

ELEMENT

The purpose of the element object in MDIS is to provide a physical description of the smallest piece of data that can be described. The element represents a data value that is logically or physically represented in the database.

Element objects cannot contain any other objects in the object model. They are considered the lowest definable unit of data. Element is mapped to Column (from Database Schema) or Field (from Record Oriented Schema) in the MDC OIM.

DIMENSION

A Dimension in MDIS is made up of a hierarchy of members, where members are data elements that are referenced by a set of coordinates that uniquely define their position in a hypercube.

Dimension is mapped onto Dimension (from OLAP Schema) and DimensionHierarchy (from OLAP Schema) in the MDC OIM.

LEVEL

A Level in MDIS is a part of a Dimension hierarchy that can be referenced by name and numbered from the top.

Level is mapped onto DimensionLevel (from OLAP Schema) in the MDC OIM.

RELATIONSHIP

The Relationship object in MDIS defines a relationship between object types. In many ways, the Relationship object is the most semantically rich and flexible object in the MDIS meta-model. There are seven types of relationships: EQUIVALENT, DERIVED, INHERITS-FROM, CONTAINS, INCLUDES, LINK-TO, and USER-DEFINED.

The DERIVED Relationship is mapped onto Transformation (from Data Transformations) of the MDC OIM while the other types are mapped onto the corresponding UML concepts.

9.4 Class Reference

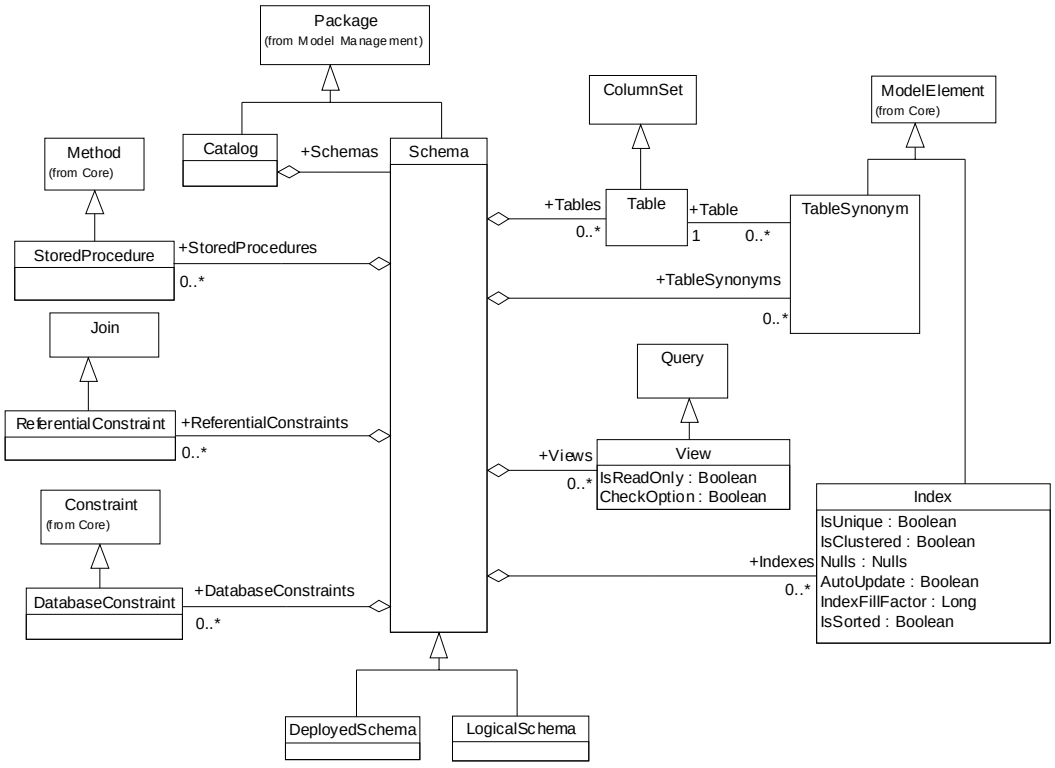


Figure 39: Schema Elements

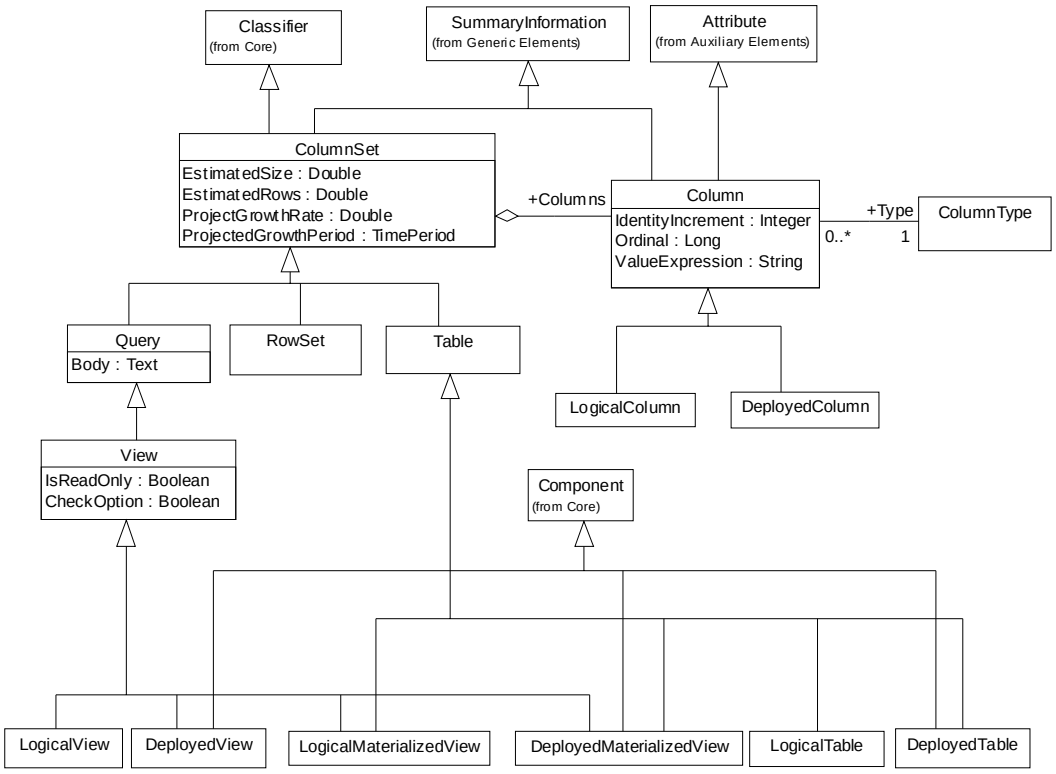


Figure 40: Tables, Columns, and Views

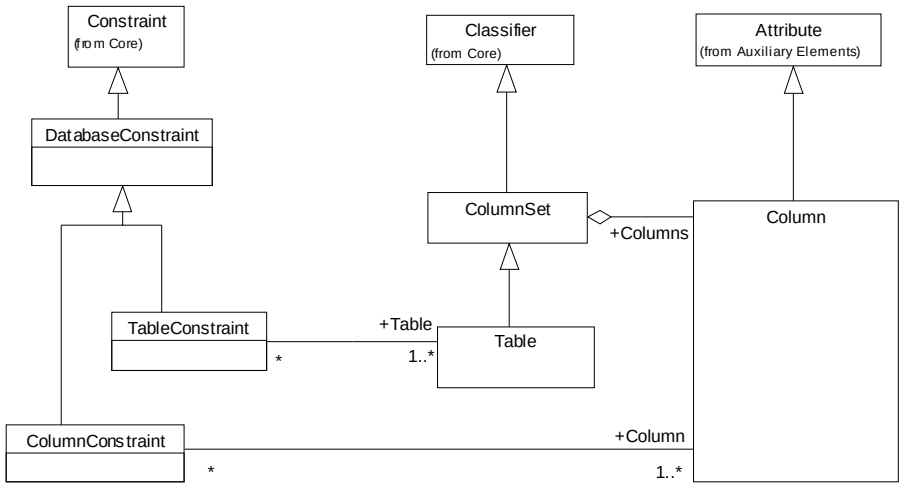


Figure 41: Constraints

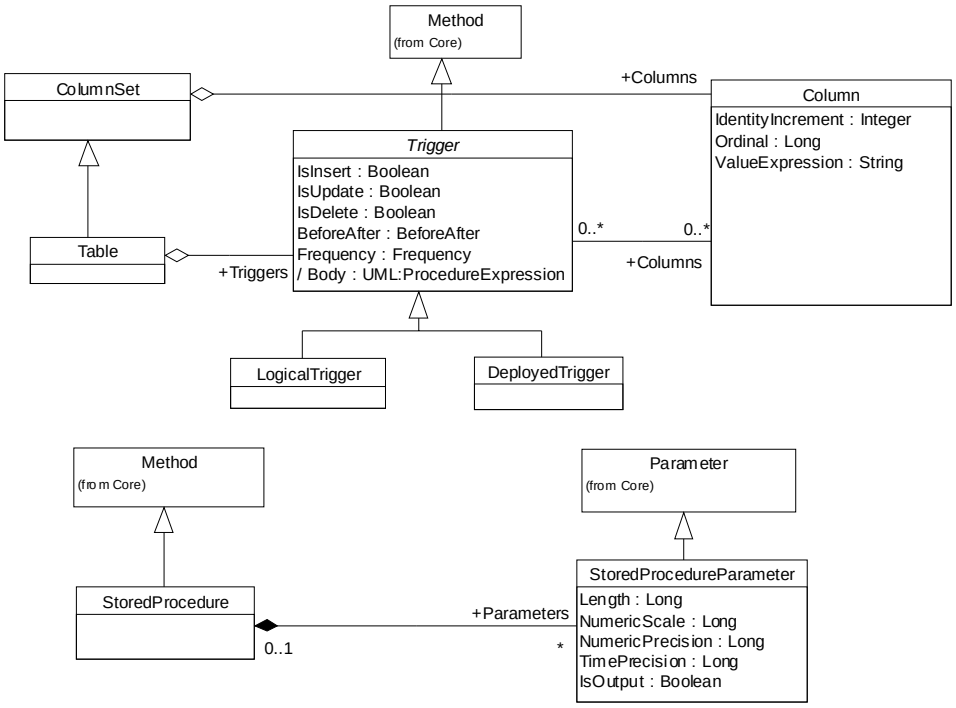


Figure 42: Triggers and Stored Procedures

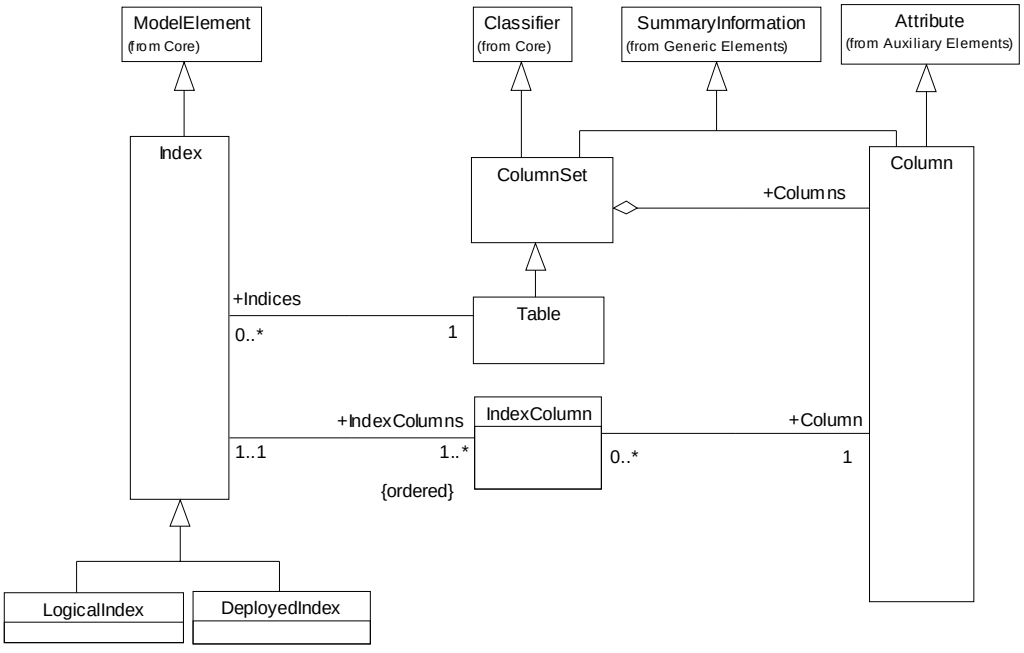


Figure 43: Indexes

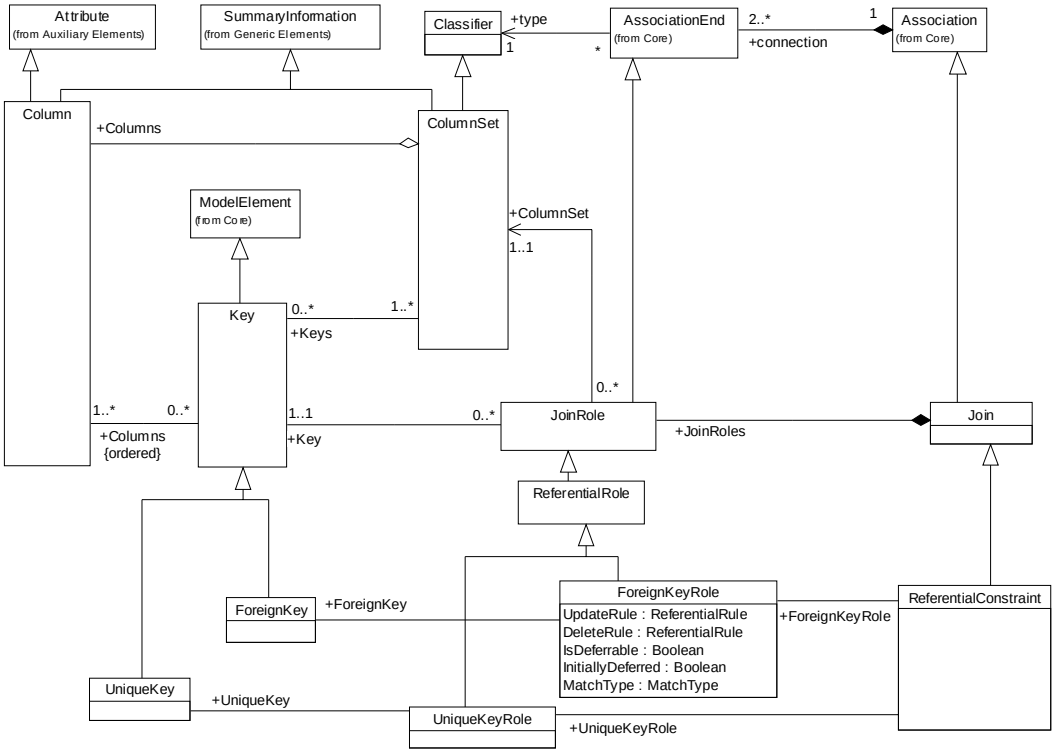


Figure 44: Referential Integrity

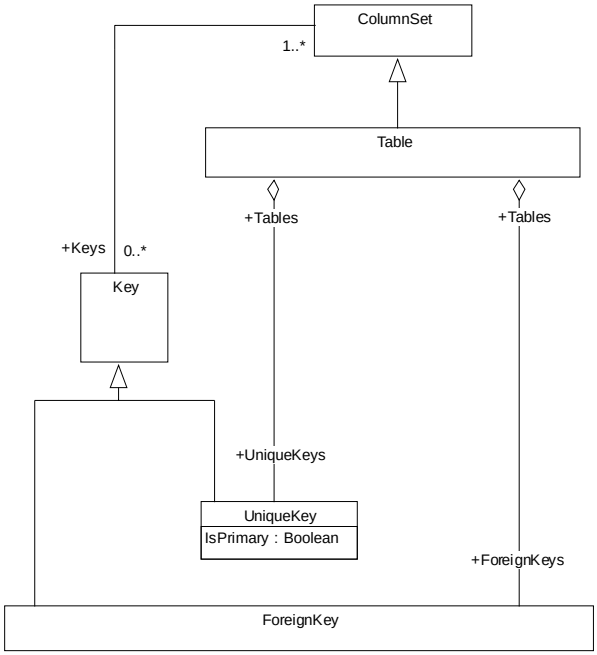


Figure 45: Keys

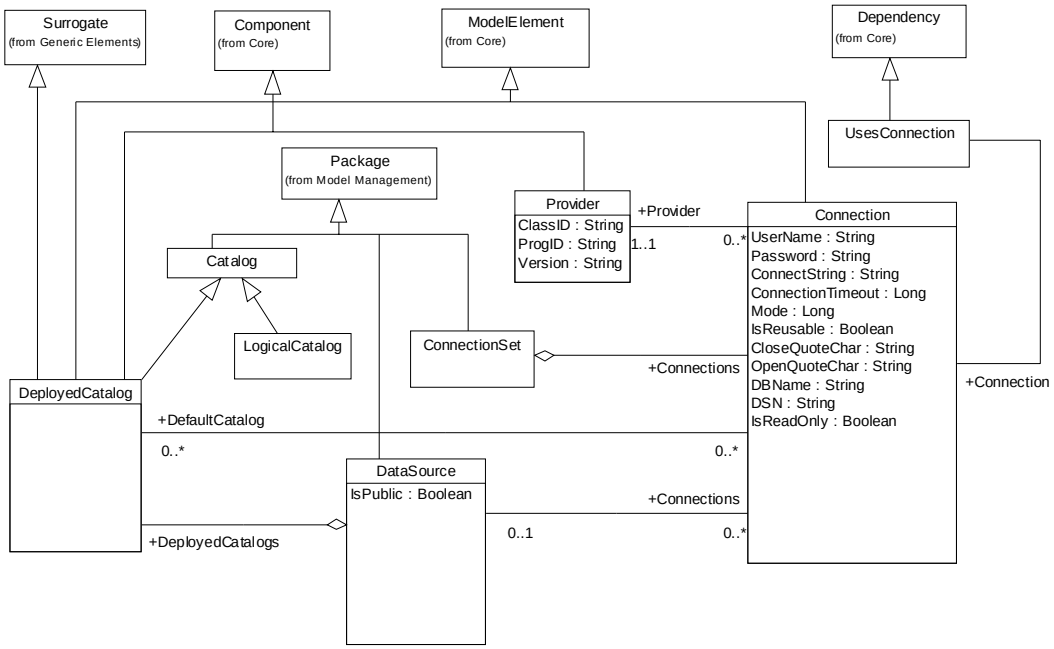


Figure 46 - Catalogs and Connections

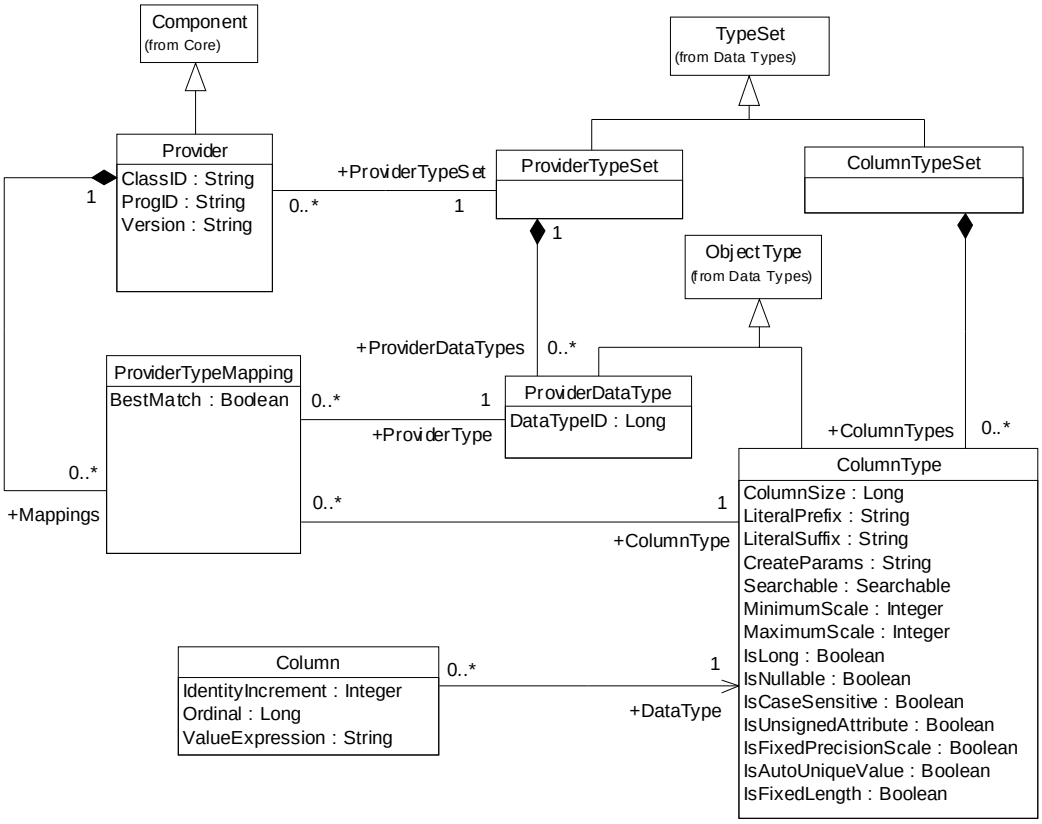


Figure 47: Data Type Mappings

9.4.1 BeforeAfter

An enumeration whose values indicate when a trigger fires.

Values

- BEFOREAFTER_BEFORE = 1 – The trigger is fired before the event.
- BEFOREAFTER_AFTER = 2 – The trigger is fired after the event.

9.4.2 Catalog

Each instance of this class describes a catalog – a named collection of schemas in a SQL environment. This class can represent: a group of files, a relational database, a network database, a hierarchical database, an object database, or any other type of data store. Because a catalog is modeled as a specialization of Package, which inherits from Element, a catalog can be contained in other types of packages that are not specified in the model.

The model distinguishes logical (or deployable) database definitions from definitions that are physically deployed. According to UML, physical tables and views are stereotypes of components (where component is a subtype of class). Having the deployed table and view classes specialize the Component (from UML) class allows them to be deployed.

Specializes

- Package (from UML)
- SummaryInformation (from Generic Elements)

Associations

- *Schemas* (Schema, derived from UML:Namespace.ownedElement) – The set of schemas contained in the catalog.

9.4.3 Column

Each instance of this class describes a column – a multiset of values that may vary over time.

All values of the same column are of the same data type or domain and are values in the same table. A value from a column is the smallest unit of data that can be selected from a table and the smallest unit of data that can be updated.

Column provides a physical description of the smallest piece of data that can be described. A column cannot contain any other object and is considered the lowest definable unit of data.

This class can represent columns in a relational database or properties in an object database.

Specializes

- Attribute (from UML Extensions)
- SummaryInformation (from Generic Elements)

Attributes

- *IdentityIncrement* (Integer) – Indicates the amount that an identity column should be incremented for each new row. This value indicates the amount that the value for this column in the previous instance should be automatically incremented in order to produce the value for this column in the current instance. Identity columns are automatically incrementing columns that are often used to provide unique key identification. If this property's value is greater than zero, then this is an identity column.
- *Ordinal* (Long) – The index of the column within the sequence of columns (1-origin indexing).

- *ValueExpression* (String) – Explains how a derived column is calculated, such as "(Extended Price * Quantity) – Discount".

Associations

- *DataType* (ColumnType, derived from UML:StructuralFeature.type) – The data type of the column.

9.4.4 ColumnConstraint

Each instance of this class describes a *constraint* on the values for a column. Constraints are invariants typically expressed using Boolean logic. A *ColumnConstraint* is a specialization of a *Constraint* (from UML).

Specializes

- Constraint

Associations

- *Column* (Column, derived from UML:Constraint.constrainedElement) – The target of the constraint.

9.4.5 Connection

Each instance of this class describes a *connection* – a client reference to a particular database resource. In the case of Microsoft® SQL Server, a particular database (catalog) within the server can also be specified.

Specializes

- ModelElement (from UML)

Attributes

- *UserName* (String) – The user name or ID used to establish the connection to the data source.
- *Password* (String) – The password used to establish the connection to the datasource. This field is an unencrypted string. This may be empty when using integrated security or if the database provider requires an encrypted password.
- *ConnectString* (String) – A string containing provider-specific extended connection information, such as an ODBC provider string. Although other connection properties (UserName, Password, DSN) can be stored in this string, it is typically used for information that cannot be expressed in other properties.
- *ConnectionTimeout* (Long) – The amount of time (in seconds) for connection initialization.
- *Mode* (Long) – A bitmask specifying access permissions requested by the connection.
- *IsReusable* (Boolean) – Indicates whether the connection may be shared among multiple clients or it is closed immediately after use.
- *CloseQuoteChar* (String) – Defines the right (closing) quoting character used by the data source.
- *OpenQuoteChar* (String) – Defines the left (opening) quoting character used by the data source.
- *DBName* (String) – The name of the database (catalog) used by the connection.
- *DSN* (String) – The name of the datasource to used by the connection.
- *IsReadOnly* (Boolean) – Indicates if the data source is read only.

Associations

- *DefaultCatalog* (Catalog) – An instance of DeployedCatalog – the catalog to be used as the default if no catalog is specified when the connection is established.
- *Provider* (Provider) – The provider that is used by this connection.

9.4.6 ConnectionSet

Each instance of this class describes a connection set – a collection of database connections grouped together for packaging.

Specializes

- Package (from UML)

Associations

- *Connections* (Connection) – A set of instances of the Connection class – these are the connections present in the ConnectionSet.

9.4.7 ColumnSet

Each instance of this class describes any general set of columns – typically a table, view, or query.

Specializes

- Classifier (from UML)
- SummaryInformation (from Generic Elements).

Attributes

- *EstimatedSize* (Double) – The estimated size for this object.
- *EstimatedRows* (Double) – The estimated number of rows for this object.
- *ProjectGrowthRate* (Double) – The projected rate of growth for the column set. Used in conjunction with the projected growth period to determine the rate of growth.
- *ProjectGrowthPeriod* (TimePeriod) – The period of time over which the growth rate holds.

Associations

- *Columns* (Column) – The set of columns in the column set.
- *Keys* (Key) – The set of keys that apply to the column set.

9.4.8 ColumnType

An underlying or base data type object associated with a database column.

Specializes

- ObjectType (from Common Data Types)

Attributes

- *ColumnSize* (Long) – The length of a non-numeric column or parameter that refers to either the maximum or the defined length for this type. For character data, this is the maximum or defined length in characters. For datetime data types, this is the length of the string representation (assuming the maximum allowed precision of the fractional seconds component). If the data type is numeric, this is the upper bound on the maximum precision of the data type.

- *LiteralPrefix* (String) – The character or characters used to prefix a literal of this type in a text command.
- *LiteralSuffix* (String) – The character or characters used to suffix a literal of this type in a text command.
- *CreateParams* (String) – Creation parameters are specified when creating a column of this data type. For example, the SQL data type DECIMAL needs a precision and a scale. In this case, the creation parameters might be the string "precision,scale". In a text command used to create a DECIMAL column with a precision of 10 and a scale of 2, the value of the TYPE_NAME column might be DECIMAL() and the complete type specification would be DECIMAL(10,2).

  The creation parameters appear as a comma-separated list of values, in the order in which they are to be supplied, with no surrounding parentheses. If a creation parameter is length, maximum length, precision, or scale, "length", "max length", "precision", and "scale" should be used, respectively. If the creation parameters are some other value, the text used to describe the creation parameter is provider-specific.

  If the data type requires creation parameters, "()" generally appears in the type name. This indicates the position at which to insert the creation parameters. If the type name does not include "()", the creation parameters are enclosed in parentheses and appended to the end of the data type name.
- *Searchable* (Searchable) – Indicates the searchability of a data type; otherwise, this column is NULL.
- *MinimumScale* (Integer) – If the type corresponds to a numeric or decimal, this is the minimum number of digits allowed to the right of the decimal point.
- *MaximumScale* (Integer) – If the type corresponds to a numeric or decimal, this is the maximum number of digits allowed to the right of the decimal point.
- *IsLong* (Boolean) – Indicates that the data type is a binary or text that contains very long data. The definition of very long data may be provider-specific.
- *IsNullable* (Boolean) – Indicates whether the columns of this data type can be defined as nullable.
- *IsCaseSensitive* (Boolean) – Indicates that the data type is a character type and is case sensitive.
- *IsUnsignedAttribute* (Boolean) – Indicates whether the column data type is unsigned.
- *IsFixedPrecisionScale* (Boolean) – Indicates that, for numeric types, the data type has a fixed precision and scale.
- *IsAutoUniqueValue* (Boolean) – Indicates that values of this type can be autoincrementing.
- *IsFixedLength* (Boolean) – Indicates whether columns of this type created by the DDL will be of fixed length.

9.4.9 ColumnTypeSet

Each instance of this class describes a collection of column types corresponding to a specific version of a DBMS product. ColumnTypes within a typeset should be shared by all columns of a certain type for a specific database product.

Specializes

- TypeSet (from Common Data Types)

Associations

- *ColumnTypes* (ColumnType, from Common Data Types:TypeSet) – Specifies the collection of column data types within this TypeSet.

9.4.10 DatabaseConstraint

Each instance of this class describes a general constraint or assertion that can involve an arbitrary collection of columns from an arbitrary collection of base tables. In most database systems, these are created via CREATE ASSERTION.

Specializes

- Constraint (from UML)

9.4.11 DataSource

Each instance of this class describes data source – a provider of database services to which a client can connect.

Specializes

- Package (from UML)

Attributes

- *IsPublic* (Boolean) – Indicates whether the data source is generally available for reuse.

Associations

- *DeployedCatalogs* (Catalog) – The set of Deployed catalogs present in the data source.
- *DBMS* (ColumnTypeNameSpace) – The instance of a column type namespace product used by the datasource.

9.4.12 DeployedCatalog

Each instance of this class describes an implemented catalog – that is, an actual physical database server. The state of a deployed catalog in the repository may be periodically synchronized with its state in the database where it is deployed. This synchronization activity is captured by the Surrogate class (from Generic Elements).

Specializes

- Catalog
- Component (from UML)
- ModelElement (from UML)
- Surrogate (from Generic Elements)

Attributes

- *SourceType* (String) – The type of provider that operates on this catalog, such as SQL Server 6.5 or Oracle 7.3.

9.4.13 DeployedColumn

Each instance of this class describes a column of a deployed table.

Specializes

- Column
- Component (from UML)

Attributes

- *ConfidenceFactor* (Integer) – Confidence in accuracy of column's data. Integer between 0%-100%

9.4.14 DeployedIndex

Each instance of this class describes an index of a deployed table.

Specializes

- Index

9.4.15 DeployedMaterializedView

Each instance of this class describes a physical grouping of columns from different tables forming a new table.

Specializes

- Table
- View
- Component (from UML)

9.4.16 DeployedSchema

Each instance of this class describes a schema of an implemented database.

Specializes

- Schema
- Component (from UML)

9.4.17 DeployedTable

Each instance of this class describes a table as realized in an implemented schema.

Specializes

- Table
- Component (from UML)

9.4.18 DeployedTrigger

Each instance of this class describes a trigger in a deployed schema.

Specializes

- Trigger

9.4.19 DeployedView

Each instance of this class describes a physical grouping of columns from different tables.

Specializes

- View
- Component (from UML)

9.4.20 ForeignKey

Each instance of this class describes an ordered collection of columns that can be used to refer to another key in another table or view.

Specializes

- Key

9.4.21 ForeignKeyRole

Each instance of this class describes a referential role on the foreign key side of the referential integrity constraint.

Specializes

- ReferentialRole

Associations

- *ForeignKey* (ForeignKey, derived from JoinRole.Key) – The foreign key that participates in this referential constraint.

9.4.22 Frequency

An enumeration whose values indicate how often a trigger fires.

Values

- FREQUENCY_PERROW = 1 – Trigger fires once for each row.
- FREQUENCY_PERSTATEMENT = 2 – Trigger fires once for each statement.

9.4.23 Index

Each instance of this class describes the physical characteristics of an *index*. Each table and materialized view may have zero or more indexes, each of which are associated with a sequence of *index columns*. Each instance can be a B+ tree, linear-hashing hash file, extensible-hashing hash file, or content index.

Specializes

- ModelElement (from UML)

Attributes

- *IsUnique* (Boolean) – TRUE only if the index is a unique index.
- *IsClustered* (Boolean) – TRUE only if the index is a clustered index. That is, TRUE means that the leaf nodes of the index contain full rows, not bookmarks. This is a way to represent a table clustered by key value. On the other hand, FALSE means that the leaf nodes of the index contain bookmarks of the base table rows whose key value matches the key value of the index entry.
- *Nulls* (Nulls) – indicates whether null values are allowed. This property should be set to one of the following values: NULLS_DISALLOWNULL, NULLS_IGNORENULL, NULLS_IGNOREANYNULL.
- *AutoUpdate* (Boolean) – Indicates whether the index is maintained automatically when changes are made to the corresponding base table. The value is either:
  TRUE: The index is automatically maintained.
  FALSE: The index must be maintained by the consumer through explicit calls. Ensuring consistency of the index as a result of updates to the associated base table is the responsibility of the consumer.
- *IndexFillFactor* (Long) – For a B+-tree index, this represents the storage utilization factor of page nodes during the creation of the index. The value is an integer from 1 to 100 representing the percentage of use of an index node. For a linear hash index, this property represents the storage utilization of the entire hash structure (the ratio of used area to total allocated area) before a file structure expansion occurs.

- *IsSorted* (Boolean) – Indicates whether the index totally orders the values of the columns on which it is defined. Typically, a sorted index is implemented by a B-tree and an unsorted index is implemented by hashing. When using sorted indexes, each column may be ordered ascending or descending, which can be specified on the associated *IndexColumn* class.

Associations

- *IndexColumns* (IndexColumn) – The set of IndexColumns, where each IndexColumn indicates the inclusion of a particular column in a particular index.

9.4.24 IndexColumn

Each instance of this class indicates that a particular column contributes to a particular index.

Attributes

- *IsAscending* (Boolean) – Indicates whether the index sorts records in ascending order on the values of the related column. If this property is false, then records will be sorted in descending order. This property is meaningful only if the IsSorted property on this interface is true.

Associations

- *Column* (Column) – The contributing column.

9.4.25 Join

Each instance of this class describes a *join* – that is, a table-to-table (or view-to-view, or table-to-view) link using one key from each table (or view). Join inherits from *Association* (from UML), which has a relationship to AssociationEnd, a generalization of JoinRole.

Specializes

- Association (from UML)

Associations

- *JoinRoles* (JoinRole, derived from UML:Association.Connection) – The pair of join roles on opposite ends of the join.

9.4.26 JoinRole

Each instance of this class describes one “side” of a join. AssociationEnd has a relationship to Classifier that is used to tie the join role to a *ColumnSet*.

Specializes

- AssociationEnd (from UML)

Associations

- *Key* – The *Key* that is used for comparison on this “side” of the join. (Each join connects table rows by comparing the value of a key of one table to the value of a key in another table.)
- *ColumnSet* (derived from UML:AssociationEnd.Type) – The table that participates in the join via this join role.

9.4.27 Key

Each instance of this class describes an ordered collection of columns on a single table or view. The same key may be used in various referential roles or join roles. If a collection of columns happens to be useful as both a unique key and a foreign key there must be two keys, because a referential role must be from a foreign key to a unique key.

Specializes

- ModelElement (from UML).

Associations

- *Columns* (Column) – The ordered set of columns contributing to the key.

9.4.28 LogicalCatalog

Each instance of this class is a canonical description of a catalog that is not deployed in any particular database.

Specializes

- Catalog

9.4.29 LogicalColumn

Each instance of this class is a canonical description of a column that is not deployed in any particular database.

Specializes

- Column

9.4.30 LogicalIndex

Each instance of this class is a canonical description of an index that is not deployed in any particular database.

Specializes

- Index

9.4.31 LogicalMaterializedView

Each instance of this class describes a canonical description of a materialized view that is not deployed in any particular database. (A materialized view is a logical grouping of columns from different tables that forms a new table.)

Specializes

- Table
- View

9.4.32 LogicalSchema

Each instance of this class is a canonical description of a schema that is not deployed in any particular database.

Specializes

- Schema

9.4.33 LogicalTable

Each instance of this class is a canonical description of a table that is not deployed in any particular database.

Specializes

- Table

9.4.34 LogicalTrigger

Each instance of this class is a canonical description of a trigger that is not deployed in any particular database.

Specializes

- Trigger

9.4.35 LogicalView

Each instance of this class is a canonical description of a view that is not deployed in any particular database.

Specializes

- View

9.4.36 MatchType

An enumeration whose values indicate the kind of match type for a referential role.

Values

- MATCHTYPE_FULL_MATCH = 1 – Every column must match for the record to be included in the reference.
- MATCHTYPE_PARTIAL_MATCH = 2 – Some columns may be null but those that are not null must match for the record to be included in the reference.

9.4.37 Nulls

An enumeration whose values indicate the way that nulls are to be handled.

Values

- NULLS_DISALLOWNULL – The index does not allow entries where the key columns are NULL. If the user attempts to insert an index entry with a NULL key, then the provider returns an error.
- NULLS_IGNORENULL – The index does not insert entries containing NULL keys. If the user attempts to insert an index entry with a NULL key, then the provider ignores that entry and no error code is returned.
- NULLS_IGNOREANYNULL – The index does not insert entries where some column key has a NULL value. For an index having a multi-column search key, if the user inserts an index entry with NULL value in some column of the search key, then the provider ignores that entry and no error code is returned.

9.4.38 Provider

A *provider* is a run-time component that provides database information and exposes underlying data types to a client application. Examples of providers are SQLOLEDB (OLE DB for SQL Server), or the ODBC driver for SQL Server. The friendly name of the provider, for example "Microsoft OLE DB Provider for ODBC Driver" should be captured in the description attribute.

Specializes

- Component (from UML)

Attributes

- *ClassID* (String) – The ClassID of the provider used by a connection for provider initialization, if applicable.
- *ProgID* (String) – The ProgID of the provider used by a connection for provider initialization, if applicable.
- *Version* (String) – The version of the provider. The version is of the form ##.##.####, where the first two digits are the major version, the next two digits are the minor version, and the last four digits are the release version. A description of the provider can be appended.

Associations

- *Mappings* (ProviderTypeMapping) – A set of instances of ProviderTypeMapping. These indicate how the provider exposes underlying datatypes to programs manipulating the data.
- *TypeSet* (ProviderTypeSet) – The typeset used by this provider. For example, the ODBC driver for Microsoft® SQL Server would specify the ODBC 3.0 TypeSet.

9.4.39 ProviderDataType

Each instance of this class describes a provider data type – an intrinsic type a provider uses to expose one or more underlying column data types.

Specializes

- ObjectType (Common Data Types)

Attributes

- *DatatypeID* (Long) – An arbitrary identifier for the ProviderDataType.

9.4.40 ProviderTypeSet

Each instance of this class describes a provider typeset – a set of data types exposed by a provider, for example OLE DB 1.0 or ODBC 3.0.

Specializes

- TypeSet (from Common Data Types)

Associations

- *ProviderDataTypes* (ProviderDataType, from Common Data Types:TypeSet) – The set of provider data types for this TypeSet.

9.4.41 ProviderTypeMapping

Each instance of this class indicates that a particular *Provider* uses a particular *ProviderDataType* to expose any column whose underlying data type is *ColumnType*.

Each instance of this class as an ordered triplet (*A*, *B*, *C*), as follows:

Whenever provider *A* encounters a database column whose ColumnType is *B*, it exposes the column's value as a variable with ProviderDatatype *C*.

Attributes

- *BestMatch* (Boolean) – Indicates that the mapping between a pair of object types by a provider is the "best" match. There is a constraint that for each underlying object type, only one instance of the mapping will have BestMatch = TRUE.

9.4.42 Query

Each instance of this class describes a query -- A query is a predefined specification for retrieving a set of information. A query can retrieve data from many different sources, including tables, views, and OLE DB providers.

Specializes

- ColumnSet

Attributes

- *Body* (Text) – The SQL text of the query.

9.4.43 ReferentialConstraint

Each instance of this class describes a referential integrity constraint.

Specializes

- Join

Associations

- *ForeignKeyRole* (ForeignKeyRole, derived from Join.JoinRoles) – The side corresponding to the foreign key.
- *UniqueKeyRole* (UniqueKeyRole, derived from Join.JoinRoles) – The side corresponding to the primary key.

9.4.44 ReferentialRole

Each instance of this class describes one “side” of a referential integrity constraint. Typically, the rules for the referential constraint are stored on the foreign key role, however some database systems may allow for different constraints on each role.

Specializes

- JoinRole

Attributes

- *UpdateRule* (ReferentialRule) – Describes the behavior when a row is updated in the table participating in the constraint.
- *DeleteRule* (ReferentialRule) – Describes the behavior when a row is deleted from the table participating in the constraint.
- *IsDeferable* (Boolean) – TRUE if the referential integrity check can be deferred.
- *InitiallyDeferred* (Boolean) – TRUE if the referential integrity check is initially deferred.
- *MatchType* (MatchType) – Indicates whether or not every referencing column value must match every referenced column value to include the record.

9.4.45 ReferentialRule

An enumeration whose values indicate the type of referential rule.

Values

- REFERENTIALRULE_CASCADE = 1 – Cascade the update or delete to the referenced row.
- REFERENTIALRULE_SET_NULL = 2 – Set the column in the referenced row to null.
- REFERENTIALRULE_SET_DEFAULT = 3 – Set the column in the referenced row to its default value.
- REFERENTIALRULE_NO_ACTION = 4 – Do nothing with the referenced row.

9.4.46 RowSet

Each instance of this class describes a generic column set that is neither a query, a view, nor a table. Such a column set can temporarily exist as the result of a query.

Specializes

- ColumnSet

9.4.47 Schema

Each instance of this class describes a schema – a persistent descriptor packaging database component descriptors. This structure can be used to represent a logical or physical relationship between database objects. For example: All objects owned by a user of a relational database, a logical grouping of objects in an object database, the directory structure of files. An instance of this class can contain objects representing: tables, indexes, and constraints in a relational database, and objects and relationships in an object database.

Specializes

- Package (from UML)
- Module (from UML Extensions)
- SummaryInformation (from Generic Elements)

Associations

- *Indexes* (Index, derived from UML:Namespace.ownedElement) – The indexes contained in the schema.
- *Tables* (Table, derived from UML:Namespace.ownedElement) – The tables contained in the schema.
- *Views* (View, derived from UML:Namespace.ownedElement) – The views contained in the schema.
- *StoredProcedures* (StoredProcedure, derived from UML:Classifier.feature) – The stored procedures of the schema.
- *ReferentialConstraints* (ReferentialConstraint, derived from UML:Namespace.ownedElement) – The set of referential (primary key/foreign key) constraints owned by the schema.
- *DatabaseConstraints* (DatabaseConstraint, derived from UML:Namespace.ownedElement) – The set of general constraints (sometimes called assertions) owned by the schema.

9.4.48 Searchable

An enumeration whose values indicate the searchability of a data type.

Values

- UNSEARCHABLE = 1 – The data type cannot be used in a WHERE clause.

- LIKE_ONLY = 2 – The data type can be used in a WHERE clause only with the LIKE predicate.
- ALL_EXCEPT_LIKE = 3 – The data type can be used in a WHERE clause with all comparison operators except LIKE.
- SEARCHABLE = 4 – The data type can be used in a WHERE clause with any comparison operator.

9.4.49 StoredProcedure

Each instance of this class describes a *stored procedure*, a named set of SQL statements that can be executed by database users. Stored procedures are modeled as extensions of *Method* (from UML). UML requires that a *Classifier* own every *Method*. However, database systems don't offer a natural grouping of stored procedures into classifiers. The solution offered by the Schema Elements package is that the Schema class specialize the Module class (defined in the UML Extensions package), of which Classifier is a specialization ancestor.

Specializes

- Method (from UML)

Associations

- *ProcedureParameters* (StoredProcedureParameter, derived from UML:BehavioralFeature.parameter) – A set of instances of the StoredProcedureParameter class.

9.4.50 StoredProcedureParameter

Each instance of this class describes a parameter of a stored procedure.

Attributes

- *Length* (Long) – The maximum possible length of a value of the attribute.
- *NumericScale* (Integer) – The number of digits to the right of the decimal point in the column for numeric parameters.
- *NumericPrecision* (Integer) – The maximum number of base 10 digits that can be stored for numeric parameters.
- *TimePrecision* (Long) – Datetime precision (number of digits in the fractional seconds portion) if the attribute is a datetime or interval type.
- *IsOutput* (Boolean) – Indicates that the parameter is a return parameter. Output parameters return information to the calling procedure.
- *Default* (String, derived from UML:Parameter.defaultValue) – The default value for the parameter. If a default is defined, the procedure can be executed without specifying a value for that parameter.

Specializes

- Parameter (from UML)

9.4.51 Table

Each instance of this class describes a *table*, a multi-set of rows. A row is a nonempty sequence of values. Every row of the same table has the same cardinality and contains a value of every column of that table. The *n*th value in every row of a table is a value for the *n*th column of that table. A row is the smallest unit of data that can be inserted into or deleted from a table. A table is a grouping of columns that describe a basic unit of data. Instances of this class can represent: record layouts of a file, relational database tables, objects in an object database, records in a network database, or segments in a hierarchical database. Tables

can contain objects representing: columns in a relational database, properties in an object database, or fields in a record type.

Specializes

- ColumnSet

Associations

- *Triggers* (Trigger, derived from UML:Classifier.feature) – The triggers defined on the table.
- *Indices* (Index) – The set of indexes for the table.
- *UniqueKeys* (UniqueKey, derived from ColumnSet.Keys) – The set of unique keys for the table.
- *ForeignKeys* (ForeignKey, derived from ColumnSet.Keys) – The set of foreign keys for the table.

9.4.52 TableConstraint

Each instance of this class describes a database constraint that applies to a table. Constraints are invariants typically expressed using Boolean logic. A *TableConstraint* is a specialization of *Constraint* (from UML).

Specializes

- Constraint

Associations

- *Table* (Table, derived from UML:Constraint.constrainedElement) – The target of the constraint.

9.4.53 TableSynonym

Each instance of this class describes an alternate name or alias for a table. While it shares its schema namespace with tables and views, unlike a view, it does not contain a SQL statement or column specification.

Specializes

- ModelElement (from UML)

Associations

- *Table* (Table) – The table for which this is a synonym.

9.4.54 Trigger

Each instance of this class describes a trigger – essentially a rule that automatically fires on a certain event such as an insert, update, or delete operation. This class captures the ANSI SQL-92 concept of a trigger.

Specializes

- Method (from UML)

Attributes

- *IsInsert* (Boolean) – TRUE if the trigger fires on an insert operation.
- *IsUpdate* (Boolean) – TRUE if the trigger fires on an update operation.
- *IsDelete* (Boolean) – TRUE if the trigger fires on a delete operation.
- *Frequency* (Frequency) – Indicates how often the trigger fires. For example, does the trigger fire once per row, or once per statement?
- *BeforeAfter* (BeforeAfter) – Indicates when the trigger fires, either before or after the operation.

- *Statements* (String, derived from UML:Method.body) – The SQL statements which specify the trigger conditions and actions.

Associations

- *Columns* (Column) – The set of columns controlling whether the trigger fires. If this association is null, then the trigger fires if the event occurs on *any* column in the table.

9.4.55 UniqueKey

Each instance of this class describes a set of columns whose values must be unique for each row of a table.

Specializes

- Key

Attributes

- *IsPrimary* (Boolean) – TRUE only if the key is a primary key.

9.4.56 UniqueKeyRole

Each instance of this class describes a referential role on the unique key side of the referential integrity constraint.

Specializes

- ReferentialRole

Associations

- *UniqueKey* (UniqueKey, derived from JoinRole.Key) – The unique key that participates in the referential constraint.

9.4.57 UsesConnection

Each instance of this class indicates that a particular object uses a particular Connection. A connection is the supplier of the *Dependency*, and the client is the object that uses, or depends on, the connection.

Specializes

- Dependency (from UML)

Associations

- *Connection* (Connection, from UML:Dependency.supplier) – The connection used by the object.

9.4.58 View

Each instance of this class describes a view – a grouping of columns not necessarily from the same table.

Specializes

- Query

Attributes

- *IsReadOnly* (Boolean) – TRUE only if the view cannot be used to insert, delete, or update data.
- *CheckOption* (Boolean) – Indicates whether inserts and updates performed through the view must result in rows that the view query can select.

9.5 OIM 1.0 Compatibility

The original OIM specialized the Common Data Types package to provide relational database specific types. The limitations of this approach have led to the recommendation to use the generic ColumnType class.

This section describes classes included in the OIM for compatibility with earlier versions of OIM.

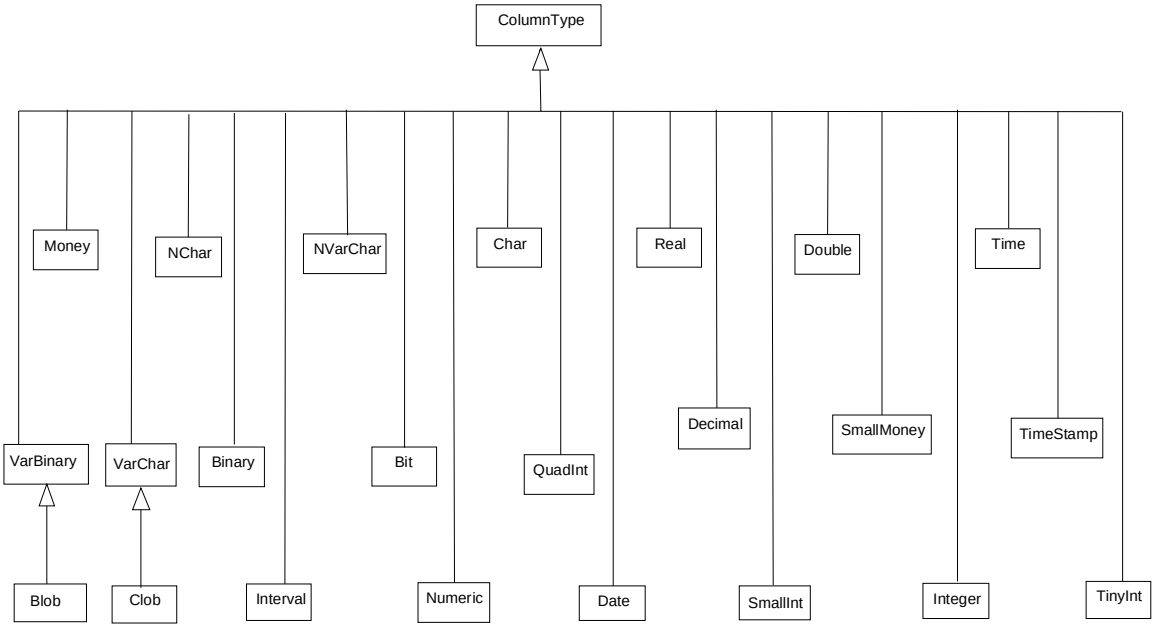


Figure 48: OIM 1.0 Data Types

9.5.1 Binary

Specializes

- Binary
- ColumnType

9.5.2 Bit

Bit describes a single-bit binary-data data type. Instances that support this interface should set the IsVariable property inherited from Binary to FALSE and the Length property inherited from Binary to 1.

Specializes

- Binary
- ColumnType

9.5.3 Blob

Blob describes BLOB data types. Blob data types usually have physical implementations that differ from other variable-length binary data types. These physical implementation differences are specific to the DBMS in question.

Instances that support this interface should set the IsVariable property inherited from the Binary type to TRUE.

Instances that support this interface should set the Length property inherited from the Binary type to the maximum length allowed for a BLOB datatype in the DBMS.

Specializes

- VarBinary

9.5.4 Char

Char describes a fixed length character string data type using the database's default character set to determine the type of character data.

The length of the instance is determined by the Length property inherited from the String data type.

The IsCaseSensitive property inherited from String determines whether the column’s sorting order should consider the case of the value.

The CharacterType property (also inherited from the String data type) indicates whether the value of the instance is composed of single byte characters, double byte characters, or multi-byte characters.

Instances that support this interface should set the IsVariable property inherited from the String type to FALSE.

Specializes

- String
- ColumnType

9.5.5 Clob

Clob describes a character large object data type. Clob data types usually have physical implementations that differ from other variable-length character data. These physical implementation differences are specific to the DBMS.

Instances that support this interface have the maximum length determined by the value of the Length property inherited from the String type. Instances that support this interface should set the IsVariable property inherited from the String type to TRUE.

The IsCaseSensitive property inherited from String determines whether the column’s sorting order should consider the case of the value.

The CharacterType property also inherited from the String data type indicates whether the value of the instance is composed of single byte characters, double byte characters, or multi-byte characters.

Specializes

- VarChar

9.5.6 Date

Date describes database year, month, and day fields conforming to the rules of the Gregorian calendar.

Specializes

- Date
- ColumnType

9.5.7 Double

Specializes

- Double
- ColumnType

9.5.8 Integer

This type represents a double word (4 byte) integer data type.

Instances that implement this interface should set the NumericPrecision attribute of the inherited Numeric type to less than or equal to 10 and NumericScale to 0.

Specializes

- LongInt
- ColumnType

9.5.9 Interval

Each instance of this class describes a datatype representing the difference between two dates or times.

Specializes

- Scalar
- ColumnType

9.5.10 NChar

The type defining a fixed-length character string data type using the database's national character set.

The length of the instance is determined by the Length property inherited from the String data type.

The IsCaseSensitive property inherited from String determines whether the columns sorting order should consider the case of the value.

The CharacterType property (also inherited from the String data type) indicates whether the value of the instance is composed of single byte characters, double byte characters, or multi-byte characters.

Instances that support this interface should set the IsVariable property inherited from the String type to FALSE.

Specializes

- String
- ColumnType

9.5.11 NVarChar

The type defining a variable length character string data type using the database's national character set to determine character data type.

The maximum length determined by the value of the Length property inherited from the String type. Instances of this type should set the IsVariable property inherited from the String type to TRUE.

The IsCaseSensitive property inherited from String determines whether the columns sorting order should consider the case of the value.

The CharacterType property also inherited from the String data type indicates whether the value of the instance is composed of single byte characters, double byte characters, or multi-byte characters.

Specializes

- String
- ColumnType

9.5.12 Numeric

Specializes

- Numeric
- ColumnType

9.5.13 Money

The type defining a money data type. Money usually differs from SmallMoney in the maximum amounts the instance of this type is able to store.

The IsSigned property inherited from Numeric should be set to TRUE.

The NumericScale and NumericPrecision properties inherited from the Numeric type should be set in an implementation specific way. Likewise, the OctetLength property inherited from the IntrinsicType type should be set in an implementation-specific way.

Specializes

- Decimal
- ColumnType

9.5.14 QuadInt

Specializes

- QuadInt
- ColumnType

9.5.15 Real

Specializes

- Single
- ColumnType

9.5.16 SmallInt

The type that represents a double word (2-byte) integer column data type. Instances that implement this interface should set the NumericPrecision attribute of the inherited Numeric type to less than or equal to 5 and NumericScale to 0.

Specializes

- ShortInt
- ColumnType

9.5.17 SmallMoney

The type defining a small money data type.

The IsSigned property inherited from Numeric should be set to TRUE.

The NumericScale and NumericPrecision properties inherited from the Numeric type should be set in an implementation-specific way. Likewise, the OctetLength property inherited from the IntrinsicType type should be set in an implementation-specific way.

Specializes

- Decimal
- ColumnType

9.5.18 Time

Specializes

- Time
- ColumnType

9.5.19 TinyInt

Specializes

- TinyInt
- ColumnType

9.5.20 TimeStamp

The type that describes a timestamp data type.

Specializes

- Datetime
- ColumnType

9.5.21 VarBinary

The type that implements a variable length binary data with a maximum length determined by the value of the Length property inherited from the Binary type.

Instances that support this interface should set the IsVariable property inherited from the Binary interface to TRUE.

Specializes

- Binary
- ColumnType

9.5.22 VarChar

The type defining a variable length character string data type with the maximum length determined by the value of the Length property inherited from the String type. Instances that support this interface should set the IsVariable property inherited from the String type to TRUE.

The IsCaseSensitive property inherited from String determines whether the columns sorting order should consider the case of the value.

The CharacterType property also inherited from the String data type indicates whether the value of the instance is composed of single byte characters, double byte characters, or multi-byte characters.

Specializes

- String
- ColumnType

This page is intentionally blank.

10 Database and Warehousing: Data Transformations

10.1 Overview

When moving data from production databases into a data warehouse or data mart, data typically needs to be transformed into a more suitable form for data analysis. The Data Transformation package describes data transformations, what they do, and what data they access.

This package covers basic transformations for relational-to-relational translations. As additional packages are introduced (legacy languages, VSAM, IDMS, and so forth), other types of sources and destinations may be specified as well.

The Data Transformation package is intended to enable sharing of meta data about transformation activities by making such meta data readily available in a commonly agreed-upon format for tools and applications. More specifically, the goals are to:

- Give data extraction and transformation logic (ETL) tools a common place to store their transformation information. This gives the customers a single place to view all of their warehouse transformations, regardless of the tool.
- Allow exchange of transformation information. Many transformations are sufficiently generic to be used for a variety of databases and applications. By storing such transformations and their constituent pieces in a repository, existing transformations can be reused when building a new data warehouse or data mart. A customer may initially build a data mart by using a low-end transformation tool, but later decide to invest in more powerful tools.
- Provide a meta data storage mechanism for custom-coded applications. This will allow custom applications to have their transformations documented in a consistent manner.
- Leverage the existing Database Schema package information in transformations. Tools can use the schema model information, such as table and column descriptions, to build their transformation models.
- Provide support for package executions, or data lineage. This allows users to trace data in the data warehouse to the transformations that were used to populate it.

The classes in this chapter are related closely to the classes in Database Schema package, which presents both the motivation for relational database schema and the usage scenarios for which they are intended.

Some source and destination classes in the transformation packages model inherit from classes in the Database Schema package. Therefore, an instance of the Data Transformation package can be viewed and accessed by tools that use the Database Schema package but do not use the Data Transformation package itself.

10.2 Semantics

This section provides a discussion of the main features of the Data Transformations package beyond that specified in the reference section. The following figure introduces the core elements of the package described in detail below:

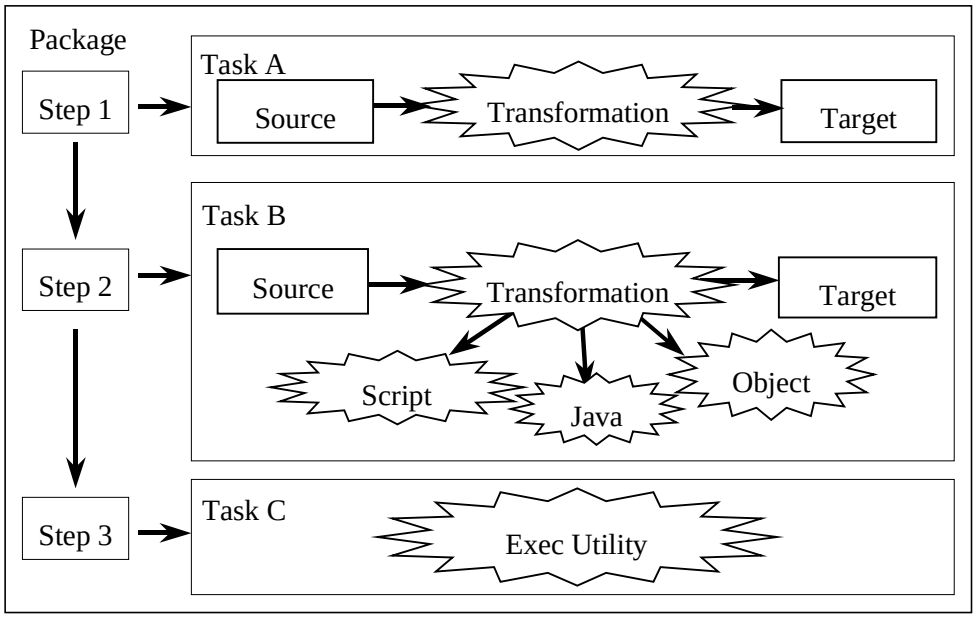


Figure 49: A Sample Transformation Package

A *transformation* maps from a set of *source* objects into a set of *target* objects, both represented by a t*ransformable object set*. The elements of a transformable object set are typically columns or tables. Transformable object sets can be both sources and targets for different transformations. The object set abstraction makes it easier to identify pairs of transformations where the output of one transformation is the input of another. This is often the case with transformations that produce and consume temporary objects.

Transformations allow a wide range of granularity to be defined based upon the information in the specific tool. If the tool is not very granular, only tracking inputs and outputs for a program, the *transformable object set* can be related to a large set of tables or columns. More granular tools can track the transformations in great detail, with each transformation only relating to one or two columns.

Within a transformation, transformation tools may store scripts, a textual description, or actual program code. Extended textual descriptions and summary information can also be stored. The model can also handle temporary transformation fields (those that are not persisted in a database).

Transformations can be packaged into groups. These groups can represent the transformations performed in a single program or in a logically related set of transformations (for example, all of the transformations related to moving the customer master file into the warehouse). There are three levels of grouping that can be done with the transformation model. The first uses a *transformation task*, which describes a set of transformations that must be executed together – a logical unit of work. In this context, a logical unit of work relates to a program or execution unit. There will usually be many physical units of work for each logical unit of work. Transformations can be ordered and executed in a particular sequence for the task.

The second level of grouping is a *transformation step*. A transformation step executes a single transformation task. Steps are used to coordinate the flow of control between tasks. The third level of grouping is a *transformation package*, consisting of a set of transformation steps, transformation tasks, and other related objects. This grouping can be used, for example, as the unit of versioning of transformations and as the unit of access control.

Simple value conversions are often performed during the transformation process. For example, a transformation may wish to map literal values of 1, 2, and 3 in a source database to ‘good’, ‘fair’, and ‘poor’ in a data warehouse. Or the transformation might use a lookup table to convert values such as zip codes to state names.

Conversions are specified using *code/decode sets*. Literal conversions are specified using *code/decode values*. The name/value pairs are specified like constants in an enumeration. Ranged expressions and other

multiple or spanning values can also be defined using tool-specific expressions. A code/decode set can also be stored in a table. For code/decode sets using a lookup table, the model specifies the related *columns* that represent the column storing the code and decode values respectively.

Package executions express the concept of a data lineage. Information about each execution of a package can be stored along with information about the execution of each task within the package. The *ExecutionID* property can be stored in a target database in the object that was populated by the package execution. This enables the lineage of the data to be traced back to the package execution that created it. This allows data in the warehouse to be tracked, so that a user can determine from which data it was derived, and which transformation was used to create it.

10.3 Class Reference

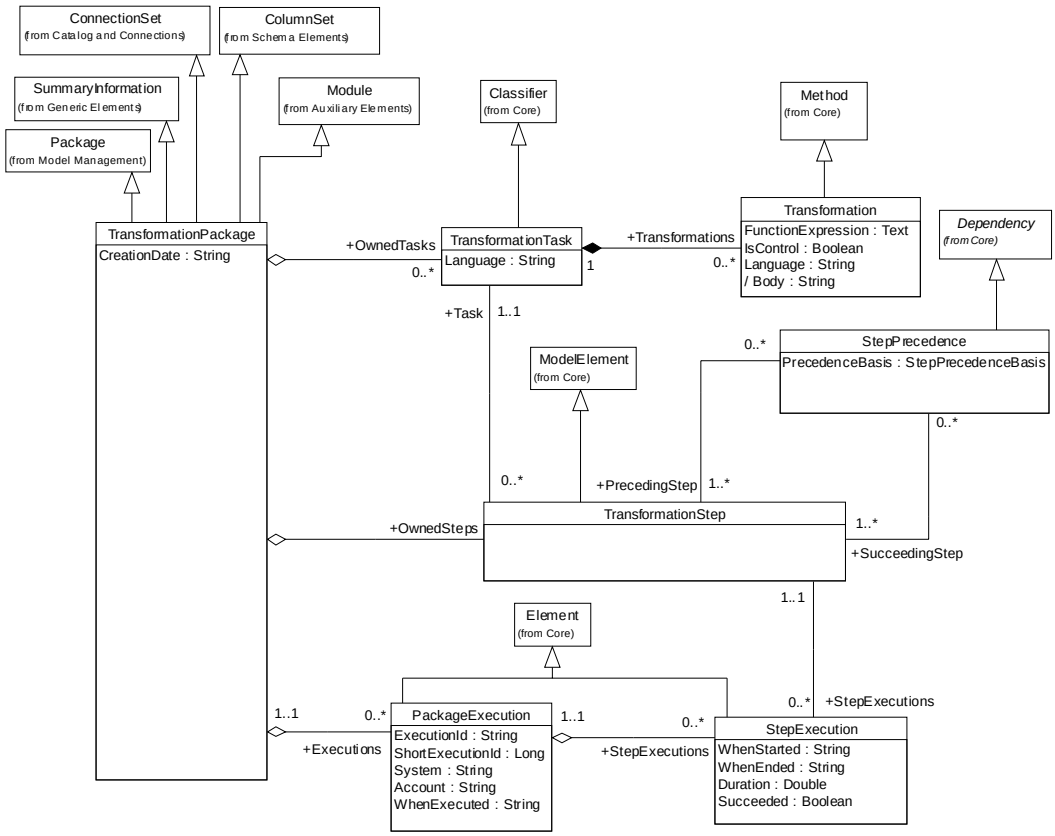


Figure 50: Transformation Packaging

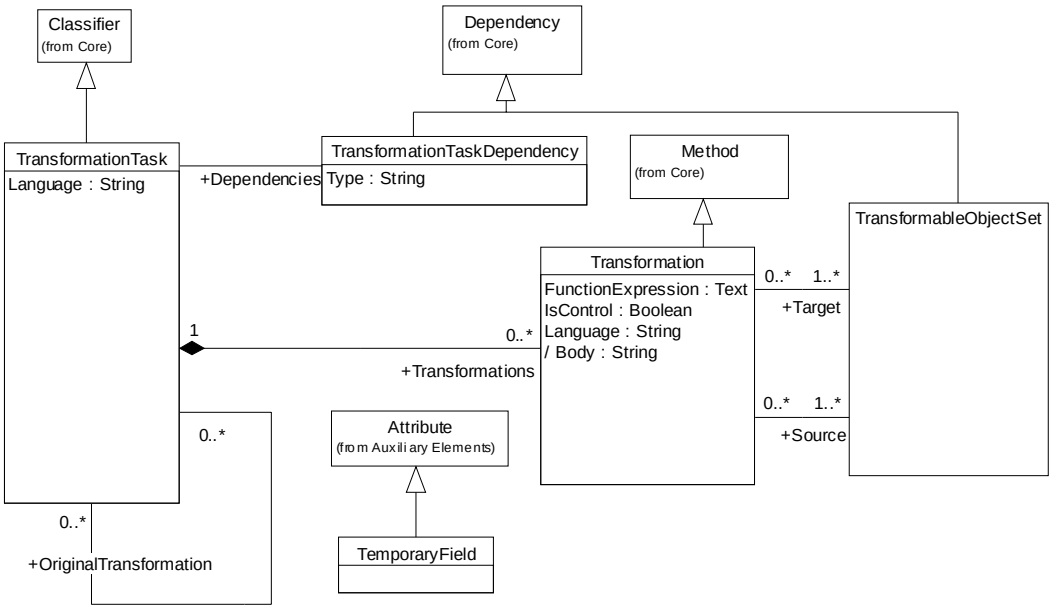


Figure 51: Transformation Tasks

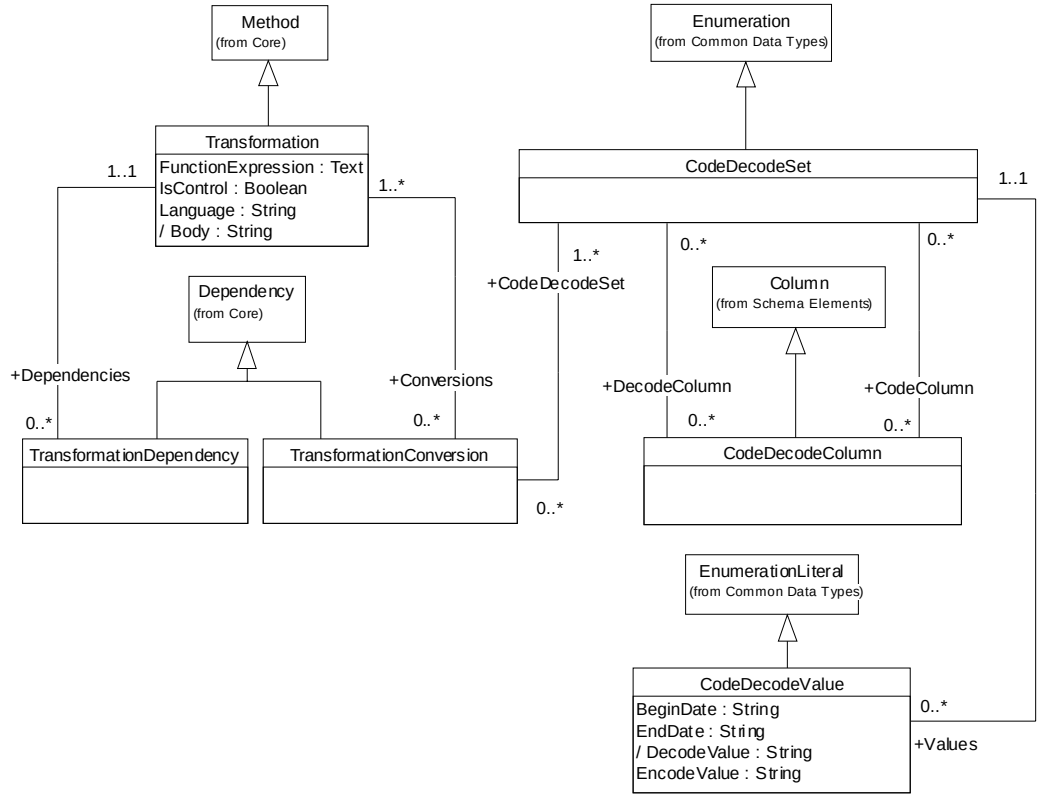


Figure 52: Constraints and Conversions

10.3.1 CodeDecodeColumn

This class defines a specialization of the column class from the Database Schema package that can be used within a *CodeDecodeSet*.

Specializes

- Column (from Schema Elements)

10.3.2 CodeDecodeSet

Each instance of this class defines a set of code/decode pairs for a TransformationConversion. There are two ways to describe code/decode values. The first is to enumerate them explicitly, through a collection of CodeDecodeValues. If the code/decode values are also stored in a table or query, the CodeColumn and DecodeColumn associations can be used.

Specializes

- Enumeration (from Common Data Types)

Associations

- *Values* (CodeDecodeValue, derived from DataTypes:Enumeration.literal) – Contains the set of encode/decode pairs.
- *CodeColumn* (CodeDecodeColumn) – Describes the column where the code value is stored. This association is used in conjunction with the *DecodeColumn* association relationship to show that the code/decode information is stored in a table or query.
- *DecodeColumn* (CodeDecodeColumn) – Describes the column where the decode value is stored. This relationship is used in conjunction with the *CodeColumn* association to show that the code decode information is stored in a table or query.

Constraints

- The target columns for both *CodeColumn* and *DecodeColumn* associations should be on the same *ColumnSet*.

10.3.3 CodeDecodeValue

Each instance of this class describes a *code/decode pair* used for mapping transformation values. The values are specified just like the constants in an *Enumeration*. Ranged expressions and other multiple or spanning values can also be defined using tool-specific expressions.

Specializes

- EnumerationLiteral (from UML)

Attributes

- *BeginDate* (Date) – The effective date for which the pair is valid.
- *EndDate* (Date) – The last effective date for which the pair is valid.
- *DecodeValue* (String, derived from DataTypes:EnumerationLiteral.Name) – The decoded value of the literal, i.e. the value that will be translated from.
- *EncodeValue* (String) – The encoded value of the literal, i.e. the value that will be translated from.

10.3.4 PackageExecution

Each instance of this class describes an execution of the associated *TransformationPackage*. The *lineage* of data can be tracked by having instances of the target data contain the *PackageExecutionID*. Customers can

determine how the data was calculated, where it came from, and when it was loaded into the data warehouse.

Specializes

- Element (from UML)

Attributes

- *Account* (String) – The user account under which the package was executed.
- *ExecutionID* (String) – A GUID that uniquely identifies the package execution, sometimes called a *lineage*.
- *ShortExecutionID* (Long) – Represents the *ExecutionID* in a compressed form.
- *System* (String) – The name of the machine on which the package was executed.
- *WhenExecuted* (String) – The date and time when the package was executed.

Associations

- *StepExecutions* (StepExecution) – Describes the steps executed during this package execution.

10.3.5 StepExecution

Instances of this class represent a *step execution* during the associated *package execution*.

Specializes

- Element (from UML)

Attributes

- *WhenStarted* (String) – Time/date when the step began execution.
- *WhenEnded* (String) – Time/date when the step finished execution.
- *Duration* (Double) – Amount of time (in seconds) that it took the step to complete execution.
- *Suceeded* (Boolean) – A Boolean indicating whether the step completed execution. If a value does not exist, the step never started execution.

10.3.6 StepPrecedence

An instance of this class represents an *order-of-execution constraint* between two transformation steps. It indicates that the successor step may not be executed until all preceding steps have been completed. The initial steps of a package are defined as steps that have no precedence. In a single-threaded system, only one step will match this criterion.

Specializes

- Dependency (from UML)

Attributes

- *PrecedenceBasis* (StepPrecedenceBasis) – Indicates whether to use *Step Status* or *Result* in the Precedence.

Associations

- *PrecedingStep* (TransformationStep, derived from UML:Dependency.supplier) – Set of *TransformationStep* instances that must be executed before the *succeeding steps* may be executed.

- *SucceedingStep* (TransformationStep, derived from UML:Dependency.client) – Set of *TransformationStep* instances that will be executed after the *preceding steps* have completed execution.

Constraints

- Circular precedence is not supported.

10.3.7 StepPrecedenceBasis

An enumeration whose values indicate whether to use *Step Status* or *Result* to determine the step precedence. May be one of the following values:

Values

- STEPPRECEDENCEBASIS_EXECSTATUS = 0 – Use the *Step Status*.
- STEPPRECEDENCEBASIS_ EXECRESULT = 2 – Use the *Result*.

10.3.8 TemporaryField

This class describes an attribute that can be used as a temporary field in a transformation. A Classifier needs to be created to hold the temporary fields (Attributes are owned by a Classifier as StructuralFeatures). Global fields can be owned by the transformation package.

Specializes

- Attribute (from UML Extensions)

10.3.9 TransformableObjectSet

This class defines a set of objects (e.g., columns) used as the source or target of a *Transformation*.

Specializes

- Dependency (from UML)

Associations

- *TransformObjects* (TransformableObject) – The set of objects referenced within the set.

10.3.10 Transformation

Each instance of this class describes a *transformation* from a set of *source* objects to a set of *target* objects. For simple transformations, the FunctionExpression property can be used to provide a short description of the transformation code/script.

To define contraints for a transformation, define a TransformationConstraint from the constraint to the transformation.

To define a simple code/decode translation, define a TransformationConversion from the CodeDecodeSet to the transformation.

Specializes

- Method (from UML)

Attributes

- *Body* (String, derived from UML:Method.body) – Any code or script for the Transformation.

- *FunctionExpression* (String) – A short description for any code/script performed by the *Transformation*.
- *IsControl* (Boolean) – Used to show that this Transformation is the primary transformation for the associated *Task*.
- *Language* (String, derived from UML:Method.body_language) – The language in which the Transformation is expressed. Typically, the name of a programming language.

Associations

- *Source* (TransformableObjectSet, derived from UML:ModelElement.clientDependency) – The set of attributes that will function as the source of the Transformation.
- *Target* (TransformableObjectSet, derived from UML:ModelElement.clientDependency) – The set of attributes that will function as the destination of the Transformation.
- *Dependencies* (TransformationDependency, derived from UML:ModelElement.clientDependency) – The set of objects on which the transformation depends.
- *Conversions* (TransformationConversion, derived from UML:ModelElement.clientDependency) – The set of CodeDecodeSet objects or other objects used by the transformation to convert values.

10.3.11 TransformationConversion

A transformation conversion is a dependency used to attach a code/decode set to a transformation.

Specializes

- Dependency (from UML)

Associations

- *CodeDecodeSet* (CodeDecodeSet) – The CodeDecodeSet that will be used by the TransformationConversion.

10.3.12 TransformationDependency

Transformation dependencies associate transformations with objects required by the transformation, such as a function or query.

Specializes

- Dependency (from UML)

10.3.13 TransformationPackage

A transformation package is the unit of storage and execution for transformations. Each instance of this class describes a grouping of Transformations, TransformationSteps, TransformationTasks, and Connections.

Specializes

- Package (from UML)
- ConnectionSet (from Database Schema)
- SummaryInformation (from Generic Elements)
- Module (from UML Extensions)
- ColumnSet (from Database Schema)

Attributes

- *CreationDate* (Date) – When the TransformationPackage was created/saved.

Associations

- *Executions* (TransformationExecution) – Represents the set of executions for the package.
- *OwnedTasks* (TransformationTask, derived from UML:Namespace.ownedElement) – The set of tasks that are owned by the TransformationPackage.
- *OwnedSteps* (TransformationStep, derived from UML:Namespace.ownedElement) – The set of steps that are owned by the TransformationPackage. Note that StepPrecedence, not the order of steps within the package, solely determines the order of execution of the steps.

10.3.14 TransformationStep

A *transformation step* describes a logical a unit of execution within the package. Each transformation step executes a single transformation task. Order of execution is defined using the *StepPrecedence* class.

Specializes

- ModelElement (from UML)

Associations

- *Task* (TransformationTask) – The TransformationTask that is executed.
- *StepExecutions* (StepExecution) – A set of actual executions of the Transformation Step. Note that this relationship did not exist in OIM 1.0.

10.3.15 TransformationTask

Each *transformation task* describes a logical unit of work within the package. It can also be used to describe an ordered set of transformations that must be executed together.

Specializes

- Classifier (from UML)

Attributes

- *Language* (String) – The language in which the transformation is expressed. Typically, the name of a programming language.

Associations

- *Transformations* (Transformation, derived from UML:Classifier.Feature) – A set of transformations that are to be executed by the task.
- *InverseTransformation* (TransformationTask) – Relates the original transformation task to a transformation task that will return transformed data back to the original state. For example, if a transformation task divides each data value by ten, then the inverse transformation task would multiply each data value by ten. This relationship is used for bi-directional transformations.
- *Dependencies* (TransformationTaskDependency, derived from UML:modelElement.clientDependency) – The set of dependencies that indicate the elements required for this task.

10.3.16 TransformationTaskDependency

A transformation task may use an existing query, table, view, or connection via the tranformation task dependency class. For example, a custom-coded application could define common queries, and each query could have a dependency on the many task instances that use it.

Specializes

- Dependency (from UML)

Attributes

- *Type* (String) – The type of object that is the target of the dependency, such as "SourceTable," "TargetTable," "SourceQuery," "TargetQuery," "InsertQuery," and so forth.

10.4 OIM 1.0 Compatibility

In OIM 1.0, objects that participated in transformations had to have classes that inherited from TransformableObject. This meant that every package that might participate in a transformation had a dependency on the Data Transformation package. In this model, transformable object sets are modeled using UML dependencies, so the class is only required for backward compatibility.

10.4.1 TransformableObject

Instances of this class are objects that can be the source or target of a transformation. Instances are collected in a TransformableObjectSet.

Specializes

- DeployedTable (from Schema Elements)

11 Database and Warehousing: OLAP Schema

11.1 Overview

Online analytical processing (OLAP) is the area of decision support that focuses on the analysis of multidimensional data in a data warehousing setting. The OLAP Schema package describes multidimensional databases.

A multidimensional database allows users to view information through a set of data *cubes*. These cubes allow users to examine a set of data values, called measures, associated with a variety of dimensions. For example, common dimensions are attributes such as time, region, product type, and customer type. Such databases are typically used in a data warehousing setting, where a user explores summaries of the data across each dimension. For example, a cube might summarize the total sales by region for each product type, or it might summarize the total sales per quarter for each region. The user is essentially exploring a data cube, where each dimension attribute defines one dimension of the cube.

The goals of the OLAP Schema package are to:

- Provide a common place for multidimensional tools to store their schema information. This gives the customer a single place to view all of his/her multidimensional data, regardless of the tool.
- Allow limited exchange of multidimensional information. Because implementations of multidimensional tools vary widely, complete exchange among tools may not be possible.
- Leverage existing database information in the multidimensional schemas. Tools can use relational database model information to integrate their multidimensional models.

This model extends the classes defined in the Database Schema package.

11.2 Semantics

This section describes semantics of the package not fully described in the reference section.

A *cube* is the basic component in multidimensional data analysis. The following diagram illustrates the features of a typical cube:

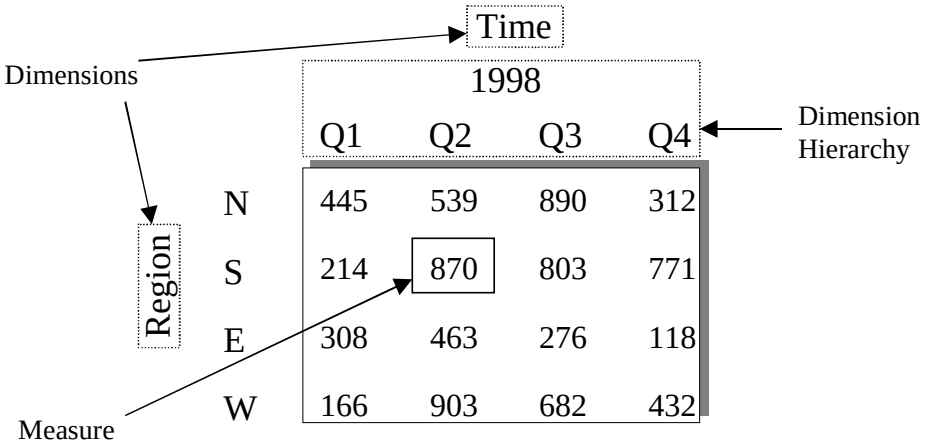


Figure 53: A Typical OLAP Cube

It is usually composed of a *measure* (or fact) table and one or more *dimension* tables. A measure table contains a value of the measure for each combination of values for the dimensions. A dimension table defines the values of a dimension. This creates either the traditional star schema or the snowflake schema commonly used in OLAP processing. For relational OLAP (ROLAP), the data is stored in a relational table and the tool uses its knowledge of the structure to select the appropriate information for the user. Pure OLAP tools retrieve the data from the relational sources and create a true multidimensional store that can then be accessed by the user. Some tools use a combination of both approaches, storing some data locally in the relational system and some in multidimensional storage.

A *dimension hierarchy* defines how a dimension decomposes into subdimensions. For example, for the product dimension each product has a product number, each product is in a product group, and then product groups are grouped into product classes. The product number can also be used to determine the division that produces the product. This would be represented as two hierarchies, one with two levels (product group, product class) and one with one level (division).

Many OLAP tools allow the data from a cube to be divided into multiple subsets, or *partitions*, for performance or storage reasons. A partition contains all of the measures and dimensions used by the partition. A horizontal partition contains all of the measures and dimensions of its cube. A vertical partition contains a subset of the measures and dimensions of its cube. The derived measures and dimensions are related back to the partition.

Aggregations are precalculated roll-ups of data stored in a cube, and they are usually maintained for performance reasons. For example, if sales data is stored by state but is often retrieved by region, an aggregation of sales by region may be created. An aggregation contains all of the measures and dimensions defined for the aggregation. In general, an aggregation contains all the measures contained in its cube, but its store dimensions reference a different level in the dimension hierarchy.

11.3 Class Reference

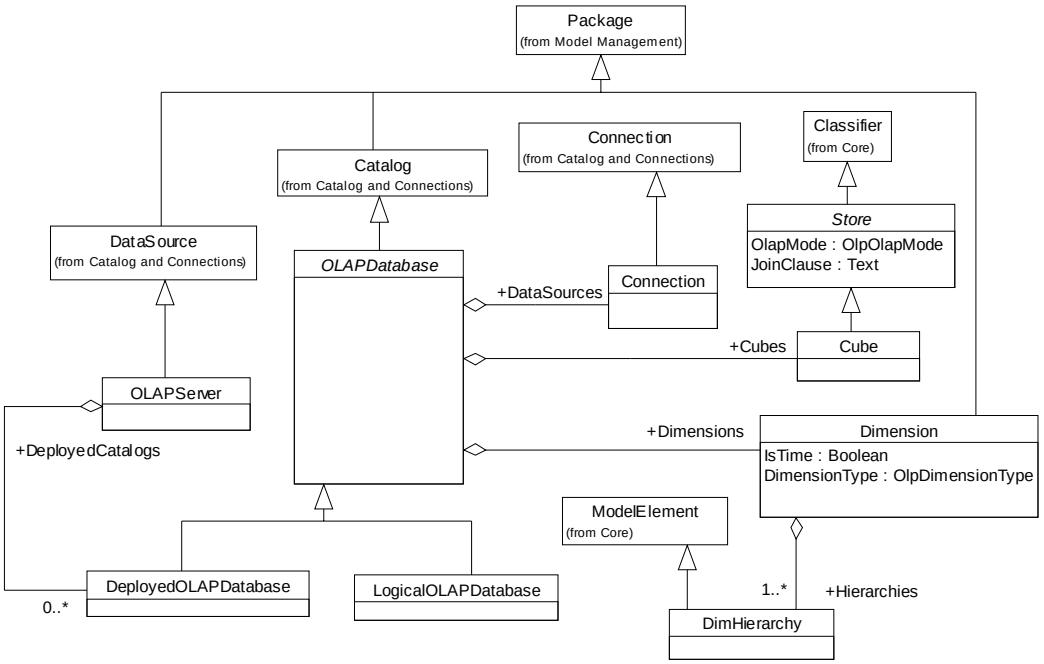


Figure 54: OLAP Servers and Databases

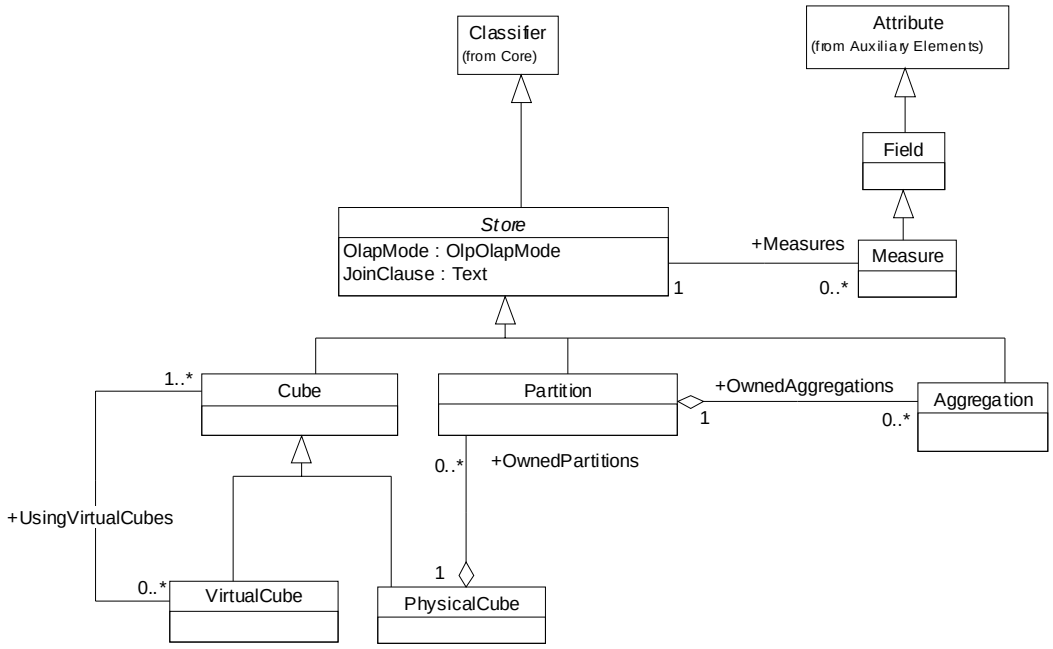


Figure 55: Stores, Cubes, and Partitions

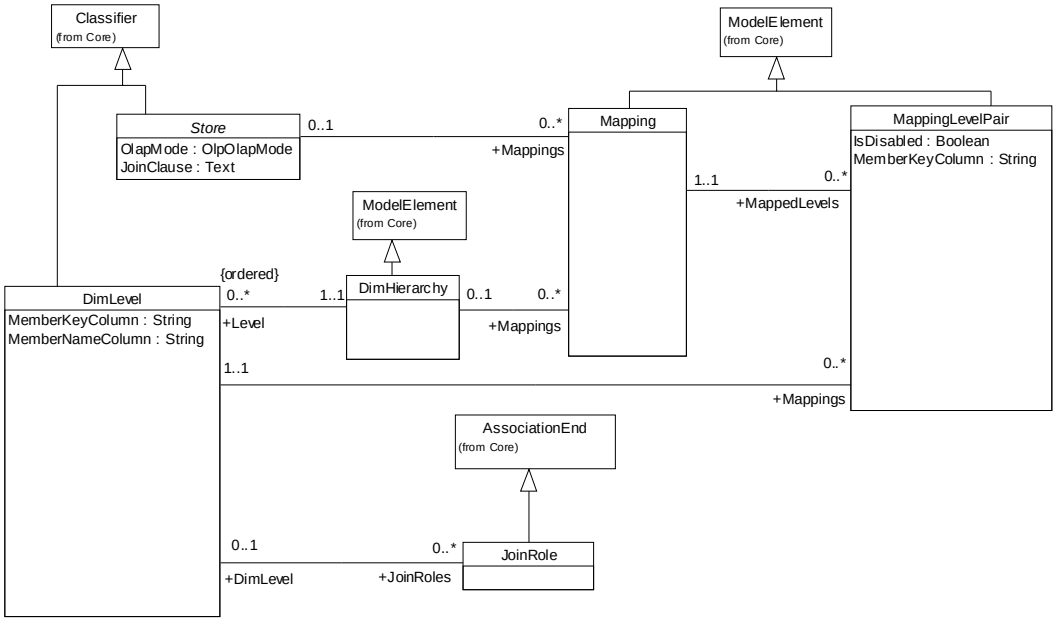


Figure 56: Hierarchy and Levels

11.3.1 Aggregation

Aggregations are precalculated roll-ups of data stored in a cube that are maintained for performance reasons. An aggregation contains all of the measures and dimensions used by the aggregation. In general,

an aggregation contains all the measures contained in its cube, but its store dimensions reference a different level in the dimension hierarchy. The derived measures and dimensions are then related back to the cube.

Specializes

- Store

11.3.2 Catalog

Specializes

- Catalog (from Database Schema)

11.3.3 Connection

Specializes

- Connection (from Database Schema)

11.3.4 Cube

A *cube* is the basic component in multidimensional data analysis. It is usually composed of a measure (or fact) table and one or more dimension tables. A measure table contains a value of the measure for each combination of values for the dimensions. A dimension table defines the values of a dimension.

Specializes

- Store

11.3.5 DeployedOLAPDatabase

An *OLAP database* is an extension of a relational database catalog, and is a container for multidimensional storage components, namely cubes and dimensions. Deployed OLAP databases are stored on an *OLAP server*.

Specializes

- Catalog (Database Schema)

Associations

- *DataSources* (Connection, derived from UML:Namespace.OwnedElement) – The set of data sources used by the catalog.

11.3.6 Dimension

The values in an OLAP cube are tracked or summarized by *dimensions*. For example, sales data can be analyzed using product, store, geography, date, and salesperson dimensions. Dimensions are defined independently of the OLAP stores that use them. A dimension's levels are associated to the stores the same way relational tables are associated to each other. In this way the model can accurately model relational OLAP (ROLAP) as well as multidimensional OLAP (MOLAP). In ROLAP the association would be one with two referential roles describing the keys and columns to join the tables, as described in the database model. With MOLAP it would be the same, except that there might not be any keys or columns to describe the join criteria.

Specializes

- Package (from UML)

Attributes

- *IsTime* (Boolean) – Indicates whether or not this dimension is a time dimension. Many OLAP tools can perform default special processing for time dimensions.
- *DimensionType* (DimensionType) – The type of the dimension. Note that time dimensions also need to set the IsTime flag.

Associations

- *Hierarchies* (DimHierarchy, derived from UML:Namespace.ownedElement) – The set of hierarchies for the dimension. Some OLAP providers support only a single hierarchy per dimension.

11.3.7 DimensionHierarchy

A *dimension hierarchy* represents an entire dimension set. The hierarchy contains one or more levels that represent the roll-up or breakdown of detail. For example, a geography dimension hierarchy could have states, regions, and countries as levels. For each level there can be one or more dimension attributes to describe the members of that level. For example, the attributes for state may be name and two-character abbreviation.

Specializes

- ModelElement (from UML)

Associations

- *Levels* (DimensionLevel) – The set of levels for the hierarchy.
- *Mappings* (Mapping) – Each hierarchy can have *mappings*, which indicate of the use of a *DimensionHierarchy* by a *Store*.

11.3.8 DimensionLevel

A *dimension level* represents a particular level in a dimension hierarchy. A dimension level can have dimension attributes, which represent data about the level that interests the user. For example, the user may want to know the name of the division, its location, and a contact person.

Specializes

- Classifier (from UML)

Attributes

- *MemberKeyColumn* (String) – The column that serves as a key of the dimension level.
- *MemberNameColumn* (String) – The column whose values serve as names of the instances of the dimension level.

Associations

- *Mappings* (MappingLevelPair) – Each level can have mappings, which indicate the use of a specific level of a *DimensionHierarchy* by a *Store*.
- *MemberKey* (Field) – The relationship connecting the level to the column that the level uses as a key. (Note that Field is a class retained for compatibility with OIM 1.0.)
- *MemberName* (Field) – The relationship connecting the level to the column that the level uses as a name. (Note that Field is a class retained for compatibility with OIM 1.0.)
- *JoinRoles* (JoinRole) – The set of joins between keys within an OLAP database.

11.3.9 DimensionType

An enumeration whose values indicate the type of a dimension.

Values

DIMENSION_REGULAR = 0	The type of the dimension cannot be determined.
DIMENSION_TIME = 1	This is a time dimension.
DIMENSION_OTHER = 3	This dimension does not fit into a standard type.
DIMENSION_QUANTITATIVE = 5	This is a quantitative dimension.

11.3.10 Field

This represents a field or column in the cube that is not a dimension or a measure. Usually it is a foreign key used to relate this cube to a dimension table, or a dimension field such as SalesTime.

Specializes

- Attribute (from UML Extensions)

11.3.11 JoinRole

Represents a key that can be joined or related to another key within an OLAP database. Objects implementing this interface will appear on each side of an association representing a join between two keys.

Specializes

- AssociationEnd (from UML)

11.3.12 LogicalOLAPDatabase

A *logical database* is simply an extension of a relational database catalog, and it is a container for multidimensional storage components, namely cubes and dimensions. Local databases can be deployed, but they may not have an associated *OLAPServer*.

Specializes

- OLAPDatabase

11.3.13 Mapping

A *mapping* indicates that a particular OLAP store maps to a particular OLAP dimension hierarchy.

Specializes

- ModelElement (from UML)

Associations

- *MappedLevels* (MappingLevelPair) – Each mapping has a set of mapped levels; each mapped level indicates the participation of a particular *DimensionLevel* within a mapping.

11.3.14 MappingLevelPair

A (mapping,level) pair indicates that a particular dimension level participates in a particular OLAP mapping.

Specializes

- ModelElement (from UML)

Attributes

- *IsDisabled* (Boolean) – Whether the mapping is valid within the store.
- *MemberKeyColumn* (String) – The column containing the key of the dimension hierarchy, as it is used by the OLAP Store.

Associations

- *MappedLevel* (DimensionLevel) – Indications of the use of a level by a store.
- *MemberKey* (Field) – The relationship connecting a dimension level to the column that level uses as a key. (Note that Field is a class retained for OIM 1.0 compatibility.)

11.3.15 Measure

A *measure* (or fact) is a set of data used in multidimensional analysis. It represents a single piece of information (e.g., SalesAmount) that will be analyzed across dimensions.

Specializes

- Field (From UML Extensions)

11.3.16 OLAPDatabase

An *OLAP database* is an extension of a relational database catalog, and it is a container for multidimensional storage components (that is, cubes and dimensions).

Specializes

- Catalog (from Database Schema)

Associations

- *Cubes* (Cube, derived from UML:Namespace.ownedElement) – The set of cubes for the database.
- *Dimensions* (Dimension, derived from UML:Namespace.ownedElement) – The set of dimensions defined with the database. Dimensions may be shared by multiple cubes.

11.3.17 OLAPMode

An enumeration whose values indicate the mode (hybrid, relational, or multidimensional) of operation.

Values

HYBRID_OLAP = 1	The data is stored in a combination of relational and multidimensional stores.
RELATIONAL_OLAP = 2	The data is stored in a relational data source.
MULTI_DIMENSIONAL_OLAP = 3	The data is stored in a multidimensional data source.

11.3.18 OLAPServer

An OLAP server is physical (deployed) provider of multidimensional data.

Specializes

- DataSource (from Database Schema)

Associations

- *DeployedDatabases* (DeployedOLAPDatabase, derived from UML:Namespace.elements) – The set of databases located on the server.

11.3.19 Partition

A *partition* is a subset of a cube used for performance or storage reasons. A partition contains all of the measures and dimensions used by the partition. A horizontal partition contains all of the measures and dimensions of its cube. A vertical partition contains a subset of the measures and dimensions of its cube. The derived measures and dimensions can then be related back to the partition.

Specializes

- Store

Associations

- *OwnedAggregations* (Aggregation) – The set of *Aggregations* owned by the partition.

11.3.20 PhysicalCube

A *physical cube* is a *Cube* that is persisted. Contrast this with a *VirtualCube*, which is a cube that is derived from one or more cubes but not persisted.

Specializes

- Cube

Associations

- *OwnedPartitions* (Partition) – The set of partitions owned by the cube.

11.3.21 Store

A *store* is an abstract class that is the generalization for the different multidimensional storage objects. Its specializations can represent cubes, virtual cubes, cube partitions, or aggregations.

Specializes

- ColumnSet (from Database Schema)

Attributes

- *JoinClause* (String) – The SQL syntax necessary to join all of the FROM tables together.
- *OlapMode* (OlapMode) – The storage mode of multidimensional data.

Associations

- *Mappings* (Mapping) – The set of instances of the Mapping class.
- *Measures* (Measure, derived from UML:Classifier.feature) – The set of measures in a cube, virtual cube, or aggregation.

11.3.22 VirtualCube

In a multidimensional schema, a *virtual cu*be is analogous to a relational view. Like a cube, the virtual cube has measures and dimensions, but those measures and dimensions are based on other measures and dimensions instead of relational columns. The measures and dimensions of a virtual cube point back to the underlying cube measures and dimensions through a derivation, with the virtual cube as the source and the cube as the target. A virtual cube typically is based on other cubes.

Specializes

- Cube

Associations

- *UsedCubes* (Cube) – The cubes on which the virtual cube is based.

11.4 OIM 1.0 compatibility

This section describes classes retained for compatibility with earlier versions of OIM.

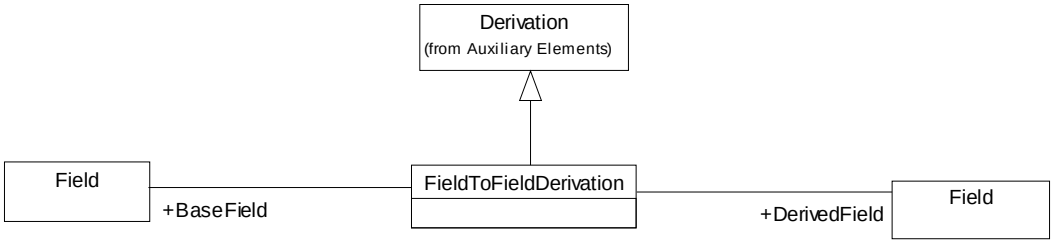


Figure 57: Field-to-Field Derivation

11.4.1 FieldToFieldDerivation

It is often the case that a measure or dimension member does not come directly from a source column, but is a simple or complex derivation of one or more source columns. For simple cases where the derivation can be expressed in SQL-like syntax, the expression can be stored as a property of the source column. For more complex cases, a *field-to-field derivation* indicates that one *Measure* is based on another *Measure*.

Specializes

- Derivation (from UML Extensions)

Associations

- *BaseField* (Field) – The underlying field.
- *DerivedField* (Field) – The field that is derived.

This page is intentionally blank.

12 Database and Warehousing: Record-Oriented Database Schema

12.1 Overview

The Record-Oriented Database Schema package describes record-oriented information, that is, information *about* data maintained in the files, legacy databases, and so forth, of an enterprise.

The goals of the Record-Oriented Schema package are to:

- Introduce a core model for describing meta data about record-oriented data sources that enables tools to store and exchange such descriptive information.
- Enable tool vendors to extend the model to address requirements of individual tools in the context of a common core model.
- Allow data warehousing tools to define the structure of some common data sources. VSAM, flat files, and record-oriented databases are commonly used as sources for data warehouses. This model and some of its derived models will provide the warehouse tools with a common metadata definition for these record definitions in these sources. Note that information about the file systems or databases themselves will be defined in their own information models (e.g., IMS database information model, VSAM information model, and so forth).

The model covers the basic elements of a record-oriented file structure or database, such as records, fields, and relationships. It also includes some deployment information for locating physical files based upon the structure, but does not address all physical or implementation details.

Some typical usage scenarios of the Record-Oriented Database Schema package are:

- Exchange of Schema Information
  Tools and applications are able to manipulate schema information stored in the repository by using the common model definitions. A repository implementing record model can be a global store for record-oriented metadata. This includes such data providers as COBOL/VSAM, Excel spreadsheets, or ASCII files, which have only limited capabilities to describe their schema or store additional design-tool-related information. Using only the repository interfaces, tools will be able to browse such information at a common location, without activating the individual data providers one-by-one.
- Reuse of Schema Information
  Storage of metadata about data sources in a repository enables the reuse of basic descriptions. An enterprise is therefore able to standardize on core definitions, such as data types, making its environment easier to maintain. A user who wants to find the definition of the customer record has one well-defined location to search and a well-defined interface to use while searching for this information.
- Catalog for Enterprise and Warehouse Information
  The Record-Oriented Database Schema model provides a one-stop store for information about enterprise data. The repository acts as a catalog that offers a common view of individual data sources and the attendant relationships. A description of a data source may not only consist of records and fields but also may have relationships that describe where it resides or how it can be accessed. Furthermore, the repository allows the user to track the history of how the metadata has evolved.

  Record-oriented structures are also common sources for data warehouses, and this model provides a common place for these definitions. The Transformations package provides for the definition of warehouse transformations; the record model can be used as the source (or target) of transformations.

- Additional Scenarios
  By storing many record schemas in the same repository, objects can be related to each other. For example, a record definition may be shared by many designs. Another scenario is to relate record information to information models covering other subject areas, such as component descriptions. A repository may be used to store a relationship between a record object and another object that references it, such as the relationship between a component and the file/database it references or populates.

12.2 Semantics

This section provides a discussion of the main features of the Record-Oriented Schema package beyond what is specified in the reference section.

The Record-Oriented Schema package does not cover detailed logical-to-physical mapping or information about any of the file systems or databases that may use record structures. This includes VSAM, IMS, IDMS, and so forth. These will later be documented in subsequent packages.

Types in the record model are different from relational schema in that record types do not have a natural owner like a database catalog. It is up to the application to determine how (if at all) to logically group record types together. The typical choice would be to use a single package to contain the record types (the package could be owned by a model, another package, or the root object). For items like COBOL copylibs, the record types could be owned by the copylib, and the copylib owned by some other object.

Care should be taken with the other related items used by the record model, such as file and node. Because they are created as part of a record type definition, they should also have their ownership defined when they are created.

Many languages offer record-oriented features that go beyond basic record processing. The record model explicitly accommodates many of the more common features – such as redefines and variable dimensions. For the others, there are properties and relationships in the model to capture these items.

There are essentially two kinds of features – those that apply to a single item, and those that link items together. The first case includes features like number of occurrences, byte alignment, and justification. Some are defined as properties of RecordItem, while the rest will use the FeatureExpression property of RecordItem. For the second case, a number of dependencies have been defined to capture features that relate one item to another. For those not already defined, LanguageFunction is used.

To indicate that two fields on separate records have values from the same domain (e.g., to indicate user-defined data type), use *Alias* to encapsulate the domain. For example, to indicate that CustId:VarChar(10) in the Customer record has the same domain CustomerId:VarChar(10) in the CustomerContact record, create a *Alias* CustomerId for VarChar(10), and then indicate that CustId in the Customer record is of type CustomerId and CustomerId in the CustomerContact record is also of type CustomerId.

The record model makes use of the existing data type model to capture the data types of the fields. Refer to the Common Data Types package for more information on how to define data types for a given language or database.

12.3 Class Reference

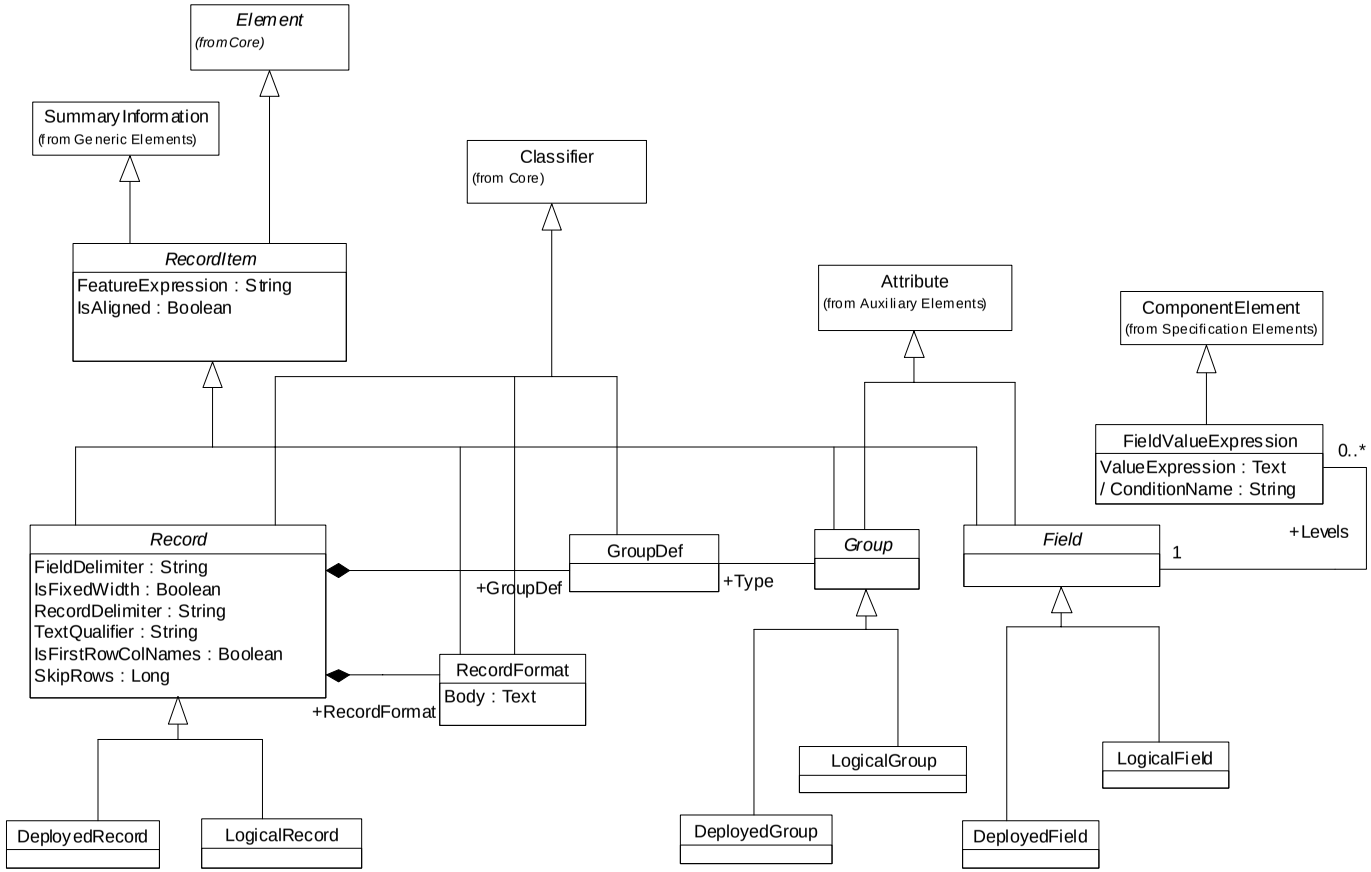


Figure 58: Records, Groups, Fields, and Formats

This page is intentionally blank.

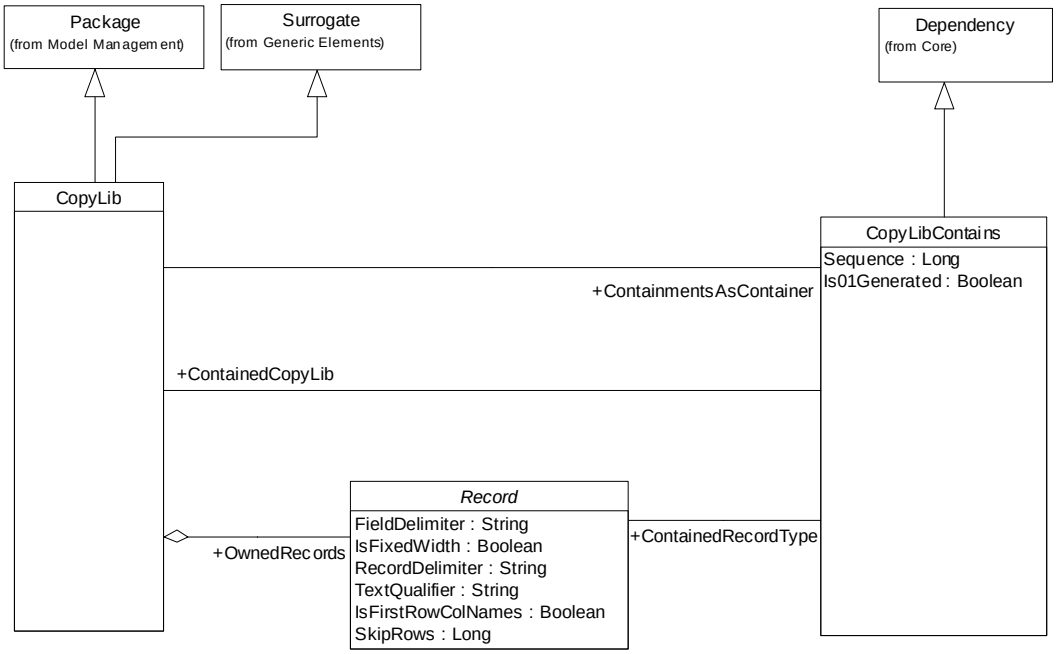


Figure 59: CopyLibs

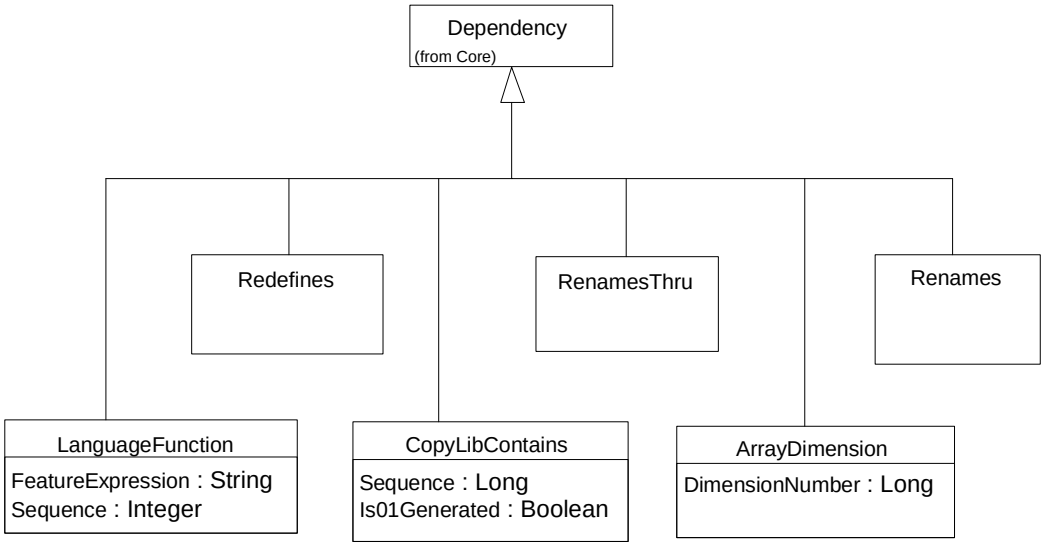


Figure 60: Constraints and Dependencies

12.3.1 ArrayDimension

Each instance of this class describes a variable dimension for a record item. This corresponds to the OCCURS DEPENDING clause in COBOL and variables used as dimensions in PL/I.

Specializes

- Dependency (from UML)

Constraints

- Only use items that support RecordItem in the client and supplier collections.

12.3.2 CopyLib

Each instance of this class describes a copy library member. Many legacy languages support the use of Copylibs, which are reusable definitions stored external to the program, much like C and C++ header files. The copylib can contain multiple record definitions, and even references to other copylibs.

There are often one or more physical files somewhere in the enterprise that include data corresponding to the record definition. The actual file that stores the copylib is defined using the File object.

To ensure that all items are owned, the copylib will also include the records, file, and FieldValueExpressions.

Specializes

- Package (from UML)
- Surrogate (from Generic Elements)

Associations

- *OwnedRecords* (Record, derived from UML:Namespace.ownedElement) – Used to provide ownership for the records in the copylib. The copylib will also relate to the record via the CopyLibContains dependency.

12.3.3 CopyLibContains

Each instance of this class describes the records and copylibs that a copylib contains.

Specializes

- Dependency (from UML)

Attributes

- *Sequence* (Long) – Used to define the ordering of the records and copylibs in the copylib.
- *Is01Generated* (Boolean) – Used to define whether or not the record item (i.e., the 01 level item) is generated in this copybook, or if generation should start at the first subordinate item.

Associations

- *ContainingCopyLib* (CopyLib, derived from UML:Dependency.supplier) – Used to define the parent copyLib that contains the subordinate record or CopyLib.
- *ContainedCopylib* (CopyLib, derived from UML:Dependency.client) – Used to define the copylib that is contained.
- *ContainedRecord* (Record, derived from UML:Dependency.client) – Used to define the record that is contained.

Constraints

- There can only be one ContainingCopylib.
- There can only be either one ContainedCopylib or one ContainedRecord.

12.3.4 DeployedField

A deployed field represents a field in a particular file or DBMS system.

Specializes

- Field

Constraints

- Can only be contained by a DeployedRecord or GroupDef from a DeployedGroup.

12.3.5 DeployedGroup

A deployed Group represents a group of fields in a particular file or DBMS system.

Specializes

- Group

Constraints

- Can only be contained by a DeployedRecord or GroupDef from a DeployedGroup.

12.3.6 DeployedRecord

A deployed record represents a group of fields in a particular file or DBMS system. Contained fields can be atomic fields (Field) or other groups of fields (Group).

This deployed record is then related to the appropriate file that contains the information defined by this record.

Specializes

- Record

Constraints

- Only a DeployedRecord can be related to a File (from Auxiliary Elements) via the ImplementationLocation collection.

12.3.7 Field

A field is an abstract data type that represents an atomic piece of information.

Specializes

- Attribute (from UML)
- RecordItem

Attributes

- *InitialValue* (String, derived from UML:Attribute.initialValue) – Used to define the default value for the field, as in the COBOL VALUE clause.
- *TypeExpression* (String, derived from UML:Attribute.typeExpression) – Used to express complex data types, as in PICTURE $ZZZ,999.9ZZ.

Associations

- *Levels* (FieldValueExpression) – Used to describe the conditional value expressions defined for this element.

12.3.8 FieldValueExpression

Record-oriented languages such as COBOL support the concept of assigning names to particular values of a field. Because these expressions are not explicitly owned by the field that uses them, they must be owned by another package. In the case of a COBOL 88Level, the CopyLib can provide the ownership.

Specializes

- ModelElement (Component Description Model)

Attributes

- *ValueExpression* (Text) – Used to define the value of the expression.
- *ConditionName* (String, from UML:ModelElement.name) – Used to define the condition name.

12.3.9 FormatOf

Used to define which record a RecordFormat is based upon.

Specializes

- Dependency (from UML)

Constraints

- The client collection must contain a RecordFormat.
- The item in the supplier collection must be a single Record.

12.3.10 Group

Each instance of this class describes a Group in a record. This is an abstract interface – all groups will be either logical or deployed groups.

Specializes

- Attribute (from UML)
- RecordItem

Associations

- *Type* (GroupDef, derived from UML:StructuralFeature.type) – Used to define the format for the group. The groupdef contains the subordinate types (fields and groups) for this group.

Constraints

- The type can only relate to a GroupDef.

12.3.11 GroupDef

A GroupDef is used to define the format of a Group. The GroupDef relates to the group via the StructuralFeatureHasType relationship. The Group is contained by the record or recordformat, and the groupdef is used to define the subordinates of the group. The groupdef relates to its subordinate fields and groups via the feature collection from Classifier.

Specializes

- Classifier (from UML)

Constraints

- Can only be used as a type definition for a group.
- Can only contain fields and groups via the feature collection.

12.3.12 LanguageFunction

Each instance of this class describes an additional language specific feature that does not have a specific interface.

Specializes

- Dependency (from UML)

Attributes

- *FeatureExpression* (String) – Used to define what the function represents. Suggested format is to use the native language expression.
- *Sequence* (Integer) – Used to sequence multiple functions of the same type.

Constraints

- Language functions can only use items that support RecordItem in the client and supplier collections.

12.3.13 LogicalField

A logical field represents a field definition that is not associated with a physical file.

Specializes

- Field

Constraints

- A Logical Field can only be contained by a LogicalRecord or GroupDef from a LogicalGroup.

12.3.14 LogicalGroup

A logical Group represents a Group definition that is not associated with a physical file.

Specializes

- Group

Constraints

- A LogicalGroup can only be contained by a LogicalRecord or GroupDef from a LogicalGroup.

12.3.15 LogicalRecord

A *logical record* represents a record definition that is not associated with a physical file. Contained fields can be atomic fields (Field) or other groups of fields (Group).

Specializes

- Record

Constraints

- A LogicalRecord cannot be related to a File (from Auxiliary Elements) via the ImplementationLocation collection.

12.3.16 Record

This is an abstract interface that represents a group of fields. Contained fields can be atomic fields (Field) or other groups of fields (Group).

This root record of a nested set can represent a number of different things – a record definition for a COBOL program, a delimited ASCII file, or a VSAM record definition. These all have very different characteristics, and thus require a very flexible model. There is some common information that can be captured about each of these, and that has been stored explicitly in the model. There are also constructs that allow additional information to be captured that may be specific to a particular language or file type.

Also, note that many languages allow for duplicate names in a record definition, e.g., COBOL FILLER. The names given to the items as they are placed in the feature collection will have to recognize this. A simple method for ensuring uniqueness would be to use NAME.SEQ as the relationship name when adding items to the collection.

Specializes

- RecordItem
- Classifier (from UML)

Attributes

- *FieldDelimiter* (String) – For files that are delimited (as opposed to fixed width), the character(s) that are used to delimit the fields.
- *IsFixedWidth* (Boolean) – Indicates whether the fields in the file are fixed width (as opposed to delimited).
- *RecordDelimiter* (String) – The character(s) used to indicate the end of a record.
- *TextQualifier* (String) – The character used to delimit text strings. For example, a quotation mark (").
- *IsFirstRowColNames* (Boolean) – Indicates whether the first row of the file contains column names for the fields.
- *SkipRows* (Long) – The number of rows at the top of the file that do not contain data.

Associations

- *Format* (RecordFormat, derived from UML:Namespace.ownedElement) – Used to provide ownership for the RecordFormat.
- *GroupDef* (GroupDef, derived from UML:Namespace.ownedElement) – Used to provide ownership for the GroupDef.

Constraints

- A record cannot be contained by another record.

Sample Data

This is an example COBOL record layout that could be expressed in the record-oriented schema model.

```
01  EMP-RECORD.
        05  EMPLOYEE-INFO OCCURS 100 TIMES
            ASCENDING KEY IS HOURLY-RATE EMPLOYEE-NO
            INDEXED BY A,  B.
          10  EMPLOYEE-NAME                      PIC X(20).
          10  EMPLOYEE-NO                        PIC 9(6).
          10  NUMBER-YEARS-EMPLOYED              PIC S9(5) COMP.
          10  HOURLY-RATE                        PIC 9999V99.
          10  WEEKLY-TALLY OCCURS 52 TIMES
              ASCENDING KEY IS NUMBER-OF-WEEK INDEXED BY C.
            15  NUMBER-OF-WEEK                   PIC 99.
            15  VACATION-DAYS                    PIC  9.
            15  UNEXPLAINED-ABSENCE              PIC  9.
            15  DAYS-LATE                        PIC  9.
```

12.3.17 RecordFormat

A record format is used to describe a usage format (view) of a record.

In many cases, the use of redefines and record types allow a single file to have many different types of logical records in one physical file. A record format is used to describe a single logical record for a record definition - a format should be unambiguous. Note that it may contain redefines in order to access sub-fields at the same time it is accessing the larger field.

The conditions under which a given format are valid (e.g., when FIELD1 = 1) also must be described.

Record formats are used to deal with the variability in many record-oriented structures. A common example would be a file that has a record type in the first byte. If it is "D" it is a detail record, and if it is "S" it is a summary record. The record definition may use redefinition to describe these different layouts in a single record. This is especially common in legacy languages such as COBOL and PL/I. There may be cases that are more complex.

The format will have its own groups and fields (which will correspond to the appropriate subset of the original record), and they will relate back to the original record groups and fields via the Derivation dependency. The format items will be in the client collection for the dependency and the record items will be in the supplier collection. The format will also relate to the record it is based upon using the FormatOf dependency, with the format in the client collection and the record in the supplier collection.

Specializes

- RecordItem
- Classifier (from UML)

Attributes

- *Body* (Text) – The condition under which the format is valid is defined using the Body property of the Query class. This value is an uninterpreted string with the format dependent on the tool storing the information.

Constraints

- A recordformat cannot be contained by another record or recordformat.

Sample Data

The purpose of the record format is to capture the conditions under which a given definition is valid. Consider the following record pseudo-definition:

```
01 Record-Layout.
        05 Record-Type                          Character(1)
        05 Detail-Info                          Character(250)
        05 Summary-Info Redefines Detail-info   Character(250).
```

If a tool is to read this file to convert the information for a data warehouse, it needs to know the differing formats and conditions under which they are valid. In this example, there would be two formats – one for the detail record and one for the summary record.

12.3.18 RecordItem

This is an abstract interface to represent the common information for record-oriented schemas.

Specializes

- SummaryInformation (from Generic Elements)

Attributes

- *FeatureExpression* (String) – Used to describe any language features that apply to this item. This would only be used for features that don't have specific attributes to describe them. The preferred method would be to separate the individual feature expressions with a semicolon (;). For example, "AUTOMATIC; Dimension (4,*,3)". It is suggested that the native language syntax be used when defining data for this item.
- *Multiplicity* (String, derived from UML:StructuralFeature.multiplicity) – Used to define the occurs for repeating fields (e.g., COBOL OCCURS clause).
- *IsAligned* (Boolean) – Flag indicating if the fields of the record are aligned to byte/word boundaries.

12.3.19 Redefines

This class is used to represent items that share the same memory location. They map to the COBOL REDFINES clause or the PL/I Def clause.

Specializes

- Dependency (from UML)

Constraints

- Only use items that support RecordItem in the client and supplier collections.

12.3.20 Renames

This class is used to represent the COBOL RENAMES clause.

Specializes

- Dependency (from UML)

Constraints

- Only use items that support RecordItem in the client and supplier collections.

12.3.21 RenamesThru

This class is used to represent the COBOL RENAMES THRU clause. This relates to the item named in the THRU portion of the clause.

Specializes

- Dependency (from UML)

Constraints

- Only use items that support RecordItem in the client and supplier collections.

13 Database and Warehousing: XML Schema

13.1 Overview

Schemas definitions in XML define types for the valid structures in an XML document. The XML Schema package provides meta data types to represent the definitions that constitute an XML schema.

Currently there are two schema definition languages for XML in use. Data Type Definition (DTD) language is a simple format that is closely related to EBNF's. DTD's lack expressiveness and therefore introduce limitations if complex schemas have to be described. Limitations are, for example, the lack of inheritance; missing data types formats, and no relationship types. XML Data (and XML Data Reduced) are efforts led by Microsoft and other companies to introduce a more expressive schema language for XML.

XML Schema is the current standardization effort by the W3C. XML Schema will provide a comprehensive schema language for XML. Parts of XML Data have been carried forward into XML Schema.

The XML Schema package is based on the subset of XML Data implemented in Internet Explorer 5 and includes all the concepts provided by DTDs. The model has been validated against a draft of the W3C XML Schema proposal. The model will be enhanced to include all the concepts of XML Schema once they become available.

Related standards:

- Data Type Definition (DTD)
- XML Data
- XML Schema

13.2 Semantics

The definition of the structure of an XML document starts with a Schema object. The Schema is a container for all the definitions of individual structures that might occur in the documents. A Schema may contain structure definitions and attribute definitions.

Element types are content models that provide the rules how XML elements can be nested and how such sub-structures can be combined with text. A content model may require that all sub-elements of the described element have to conform to the schema. Such a model is called a closed content model, while an open model does not restrict the possible sub-elements. A special case of element is the empty XML element that simply consists of a single tag and therefore has no content. Other content types may be text only, sub-elements only and a mixture of both.

Attribute types specify name/value pairs for start tags of XML Elements. Part of the definition of an attribute is its data type, default value and if it is mandatory or optional.

The content model consists of references to element and attribute types that are used to define the valid structures of the element. A content model imprints a sequence and occurrence onto the definitions it contains. In order to change these definitions for a specific subset of references Groups may be introduced, which encloses the subset of references.

In order to avoid name collisions between schema elements in a document, XML *namespaces* are used. Namespaces are sets of names identified by a URI reference. The names may be used once the namespace has been declared by an XML structure such as a schema, element, or data type.

13.3 Model Reference

The following shows the UML diagram for the XML Schema package.

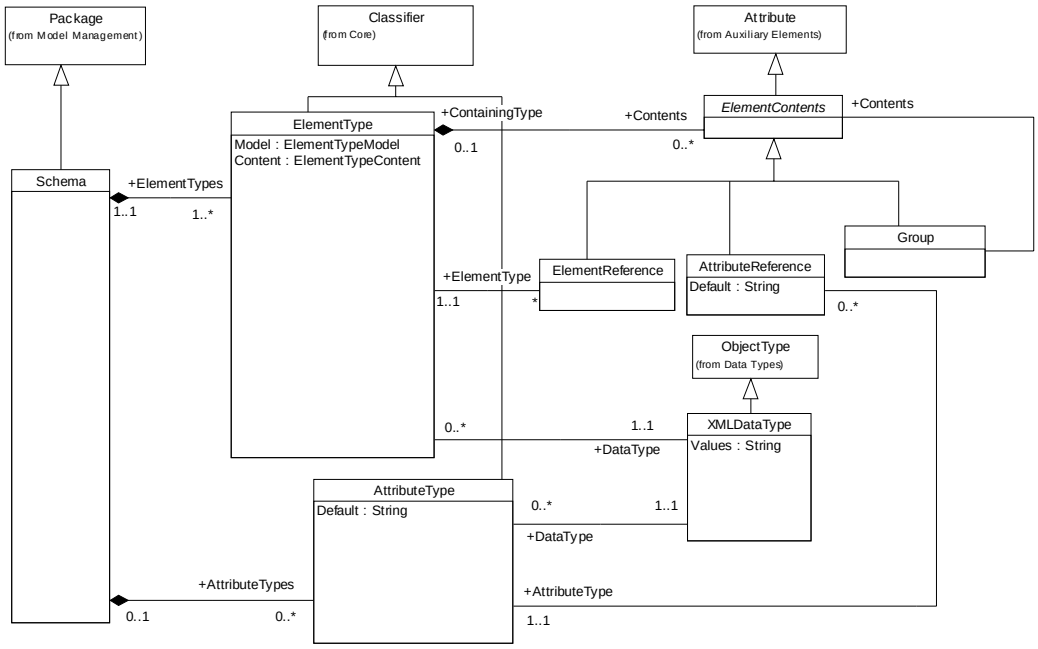


Figure 61: XML Schema

The following sections describe the different meta data types of the XML Schema package in alphabetical order.

13.3.1 AttributeReference

An AttributeReference is a reference to an attribute definition (AttributeType) contained in the definition of an ElementType, thereby including it into the definition. The reference may supply a default value for the specific use of the AttributeType in the containing ElementType definition.

Specializes

- ElementContent

Attributes

- *Default* (String) – default value for this specific instance of the attribute definition.
- *Occurrence* (Multiplicity) – indicates if the attribute is required or optional in an XML element. Note that this definition can not override a more restrictive specification supplied by the AttributeType. The possible values are:
  - 0..1 the attribute is optional
  - 1..1 the attribute is required

Associations

- *Type* – a collection of at most one AttributeType object that represents the referenced attribute definition.

13.3.2 AttributeType

An AttributeType is a definition of a XML attribute that can be used in the definition of ElementTypes. In XML an attribute is represented as a name/value pair contained in a tag, e.g. <element_tag attribute_name=value >. The AttributeType represents a grammar rule of how to parse the name / value statement. The definition also declares if the attribute has a default value and if it is required or optional.

Specializes

- Classifier (from UML)

Attributes

- *Name* (String) – defines the name of the AttributeType.
- *ShortDescrition* (String) – Description of the purpose of the AttributeType is present.
- *Default* (String) – default value for the attribute if not present in an XML element.
- *Occurrence* (Multiplicity) – indicates if the attribute is required or optional in an XML element. The possible values are:
  - 0..1 the attribute is optional
  - 1..1 the attribute is required

Associations

- *DataType* – a collection with at most one XMLDataType object, which defines the data type of the attribute.

13.3.3 ElementContent

Specializes

- Attribute (from UML)

13.3.4 ElementReference

An ElementReference references an ElementType as part of a content model definition. The object specifies if the sub-element may be required, optional, or may occur multiple times.

Specializes

- ElementContent

Attributes

- *Occurrence* (Multiplicity) – indicates if the attribute is required or optional and how often it may occur in an XML element. The possible values are:
  - 0..1 the attribute is optional and may occur only once (OPTIONAL)
  - 1..1 the attribute is required and must occur only once (REQUIRED)
  - 0..* the attribute may occur unlimited times (ZEROORMORE)
  - 1..* the attribute may occur from 1 to an unlimited number of times (ONEORMORE)

Associations

- *ElementType* (ElementType from UML:StructuralFeature.type) – collection that contains an ElementType object that represents the referenced ElementType.

13.3.5 ElementType

An ElementType provides a content model for a well-defined XML structure and as such is the equivalent of a class for an object definition. The content model consists of references to element and attribute types. Element references describe the XML sub-elements that may occur in this ElementType and attribute references define the name/value pairs that are allowed for an element.

Specializes

- Classifier (from UML)

Attributes

- *Name* (String) – defines the name of the ElementType.
- *ShortDescrition* (String) – Description of the purpose of the ElementType is present.
- *Model* (ElementTypeModel) – defines if the content model can be extended with additional elements (open) or if the addition of structured is not allowed (closed).
- *Content* (ElementTypeContent) – defines the structure of the content.

Associations

- *Elements* – collection of ElementReference or AttributeReference objects that constitute the content model for the ElementType.

13.3.6 ElementTypeContent

ElementTypeContent is an enumeration that provides the valid definitions for the content part of an ElementType.

Attributes

- ELEMENTTYPECONTENT_MIXED = 0 – elements and characters (text) together may be contained in an element described by this type.
- ELEMENTTYPECONTENT_EMPTY = 1 – specifies that the element can have no content
- ELEMENTTYPECONTENT_TEXT_ONLY = 2 – the element can have only text as content and no sub-elements.
- ELEMENTTYPECONTENT_ELEMENTS_ONLY = 4 – the element can contain only sub-elements and no text.

13.3.7 ElementTypeModel

ElementTypeModel is an enumeration that provides the content model types of an ElementType.

Values

- ELEMENTTYPEMODEL_OPEN = 0 - the content model can be extended with additional elements
- ELEMENTTYPEMODEL_CLOSED = 1 - the addition of structures not defined in the schema to an element is not allowed

13.3.8 Group

A Group represents a set or sequence of elements in a content model, i.e. can be used to introduce alternative orderings among elements.

Specializes

- ElementContent

Attributes

- *Occurrence* (Multiplicity) – indicates if the group is required or optional and how often it may occur in an XML element. The possible values are:
  - 0..1 the attribute is optional and may occur only once (OPTIONAL)
  - 1..1 the attribute is required and must occur only once (REQUIRED)
  - 0..* the attribute may occur unlimited times (ZEROORMORE)
  - 1..* the attribute may occur from 1 to an unlimited number of times (ONEORMORE)
- *Order* (ExpressionOrder) – indicates if the group is a set (OR) or a sequence (AND)

Associations

- *Elements* – collection of ElementReference or AttributeReference objects that constitute the content model for the ElementType.

13.3.9 Schema

A Schema object is a container for the ElementType and AttributeType definitions that makeup the definition of a XML document. A Schema might be named.

Specializes

- Package (from UML)

Associations

- *Elements* – collection of ElementType and AttributeType definition objects.

13.3.10 XMLDataType

XMLDataType defines the format of Attributes or Elements.

Specializes

- DataType (from UML)

Attributes

- *Name* (String) – defines the name of the data type.
- *Values* (String) – provides a set of values in case that the data type is an enumeration.

1

14 Database and Warehousing: Report Definitions

14.1 Overview

The Report Definitions package provides meta data types to represent information necessary for reporting tools and their relationships to the systems they report on. The goals of the Report Definitions package are to:

- Introduce a core model for describing meta data about reports that enables tools to store and exchange this type of meta data in a consistent format.
- Enable tool vendors to extend the model to address requirements of individual tools in the context of a common core model.
- Allow for business intelligence and reporting tools to define the structure of their reports and how they relate to existing systems, such as warehouse databases.

Storage of meta data about reports in a common format enables the reuse of basic descriptions. A business is therefore able to standardize on core definitions, such as common report fields, making its environment easier to maintain. A user who wants to find the definition of “total sales” has one well-defined location to search and a well-defined description while searching for this information.

The Report Definitions package also provides a one-stop store for information about enterprise data. Implemented by a repository it acts as a catalog that offers a common view of individual reports and the relationships in-between. A description of a report may not only consist of reports and fields but also may have relationships that describe where it resides or how it can be accessed. Reports are also commonly used to access data warehouses, and this model provides a common place for these definitions. Users can scan the model to find existing reports for a given topic, or see the source for a given report field.

The initial target is to support the definition of reports for business intelligence and reporting tools accessing a relational / OLAP warehouse. Extensions for reporting for other domains may be added in the future.

14.2 Semantics

Below is a sample report. It has two groups and a number of fields. It illustrates most of the concepts in the report model.

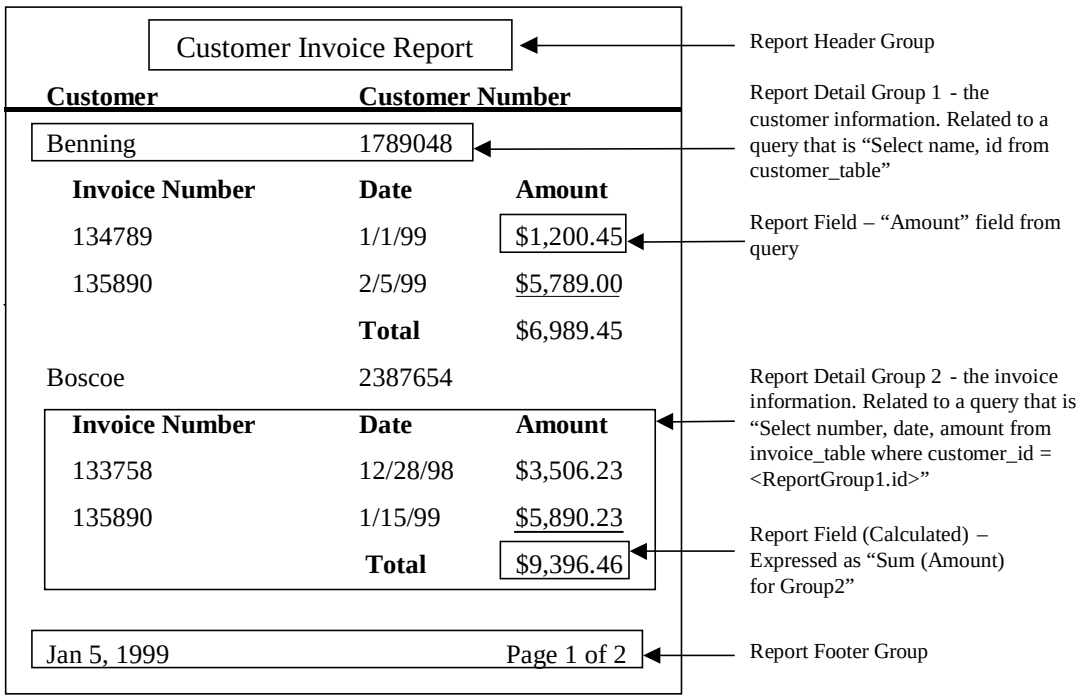


Figure 62 - Sample Report

A *report* is a set of related formatting definitions. It is structured into *report groups* that may contain *report fields*. Report groups may either be headers, footers, detail bands or a custom type interpreted by the reporting tool. Fields provide the actual definitions for pieces of information. Fields may either be expressed as query result columns, derived from other fields, or a functional expression.

The UML concept of *dependency* is used to describe how one field is based upon another column or field. In the case of report fields, the report field is dependent on the underlying columns or other report fields upon which it is based.

The report model makes use of the OIM data type definitions in order to capture the data types of the fields. A reporting tool could either use the data types in a data provider namespace (e.g. ODBC) or create a custom set of types. Refer to the Common Data Types package for more information on how to define data types for a given language or database.

Reports are contained in a *report package*, which represents a simple grouping of related reports. Reports may be grouped into tool specific, user specific packages, or classified based on the information they report on.

The package currently does not cover detailed semantics for layouts or graphical positions. Each tool will likely have very different means for storing this information.

14.3 Class Reference

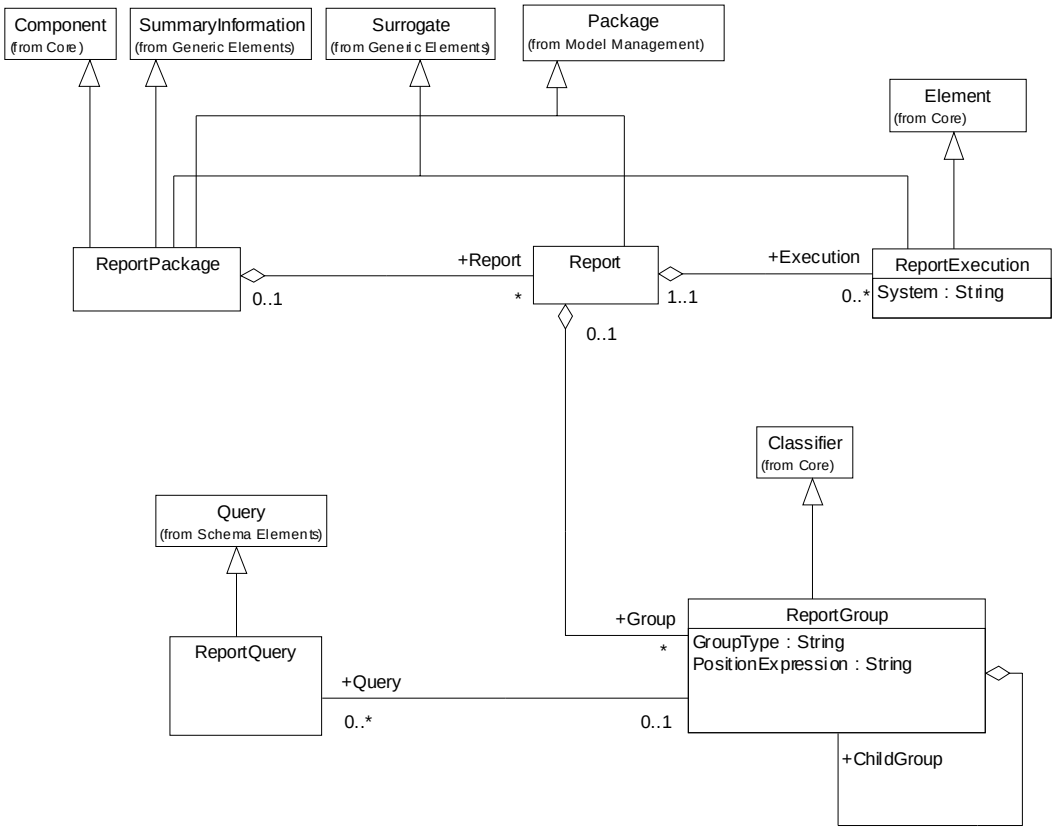


Figure 63- Report Grouping Elements

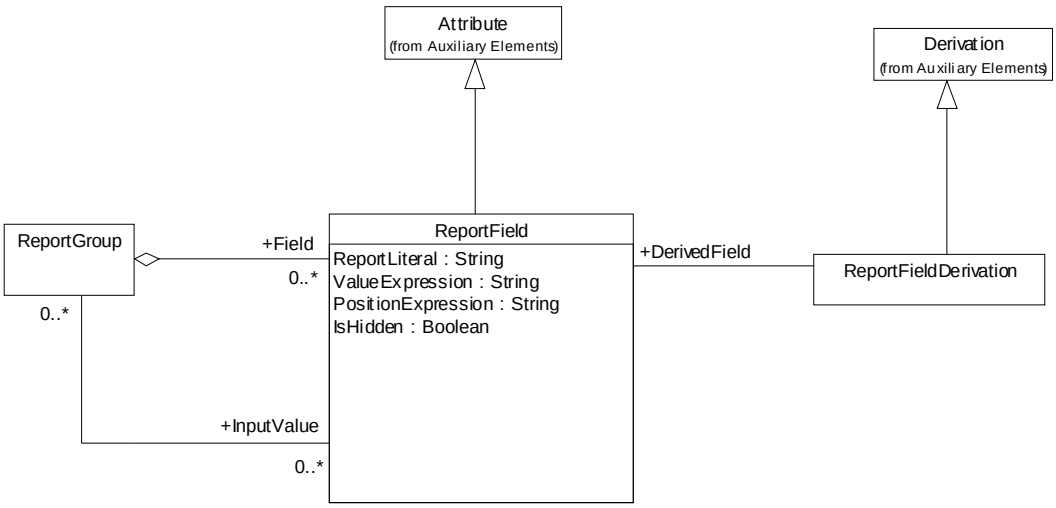


Figure 64 - Report Field Elements

14.3.1 Report

Each instance of this class describes a report used to represent a single set of information formatted to be understandable by a user. It can represent a printed, HTML, or dynamic on-line report.

The source for reports can be files, relational databases, OLAP stores, etc.

Specializes

- Package (from Model Management)
- Surrogate (from Generic Elements)

Associations

- *Execution* (ReportExecution) – Links a report to its executions.
- *Group* (ReportGroup, from Namespace.ownedElement) – Describes the Report groups in a report.

Constraints

- Every report must have at least one ReportGroup to contain any fields on the report.

14.3.2 ReportDerivation

Each instance of this class describes a derivation for the report field. This is what columns, report fields, etc., the report fields value is derived from (if any). The source for reports can be files, relational databases, OLAP stores, etc.

Specializes

- Derivation (from Auxiliary Elements)

Associations

- *DerivedField* (ReportField, from Dependency.Client) – Defines the report field that is derived from another object.

14.3.3 ReportElement

Each instance of this class describes a single piece of information appearing on a report. Note that a report field may repeat on a report, as in the case of a tabular report, or it may be represented graphically as in a chart.

Specializes

- Attribute (from Auxiliary Elements)

Attributes

- *ElementType* (ReportElementType) – Indicates the type of the element. One of the following:
  ELEMENT_TYPE_FIELD = 1
  ELEMENT_TYPE_TEXT = 2
  ELEMENT_TYPE_GRAPHIC = 3
  ELEMENT_TYPE_OTHER = 4
- *Literal* (String) – Literal or column heading used to describe the field on the report (if any). If the ElementType is text, will contain the text displayed.
- *ValueExpression* (String) – Explains how a derived field is calculated, such as "(Extended Price * Quantity) - Discount"
- *PositionExpression* (String) – Expression describing where this element exists within the group on the report.

- *StyleExpression* (String) – Expression describing the formatting of the element. For example, the font or color.
- *IsHidden* (Boolean) – Indicates that the field is not visible on the report.

14.3.4 ReportElementType

Enumeration indicating the type of the report element.

Values

- ELEMENT_TYPE_FIELD = 1
- ELEMENT_TYPE_TEXT = 2
- ELEMENT_TYPE_GRAPHIC = 3
- ELEMENT_TYPE_OTHER = 4

14.3.5 ReportExecution

Each instance of this class describes an execution of a report. Can be used to track who is executing reports, or to store the location for saved reports.

Specializes

- Element (from Core)
- Surrogate (from Generic Elements)

Attributes

- *System* (String) – Location where the report was executed, usually a machine name.

14.3.6 ReportGroup

Each instance of this class describes a grouping of fields on a report or a report section.

A common use would be the results of an SQL statement displayed in one group. Additional groups could be embedded within, as in a group with invoice information and a subgroup with line item detail. Other examples would be a detail section or a footer section.

Specializes

- Classifier (from Core)

Attributes

- *GroupType* (ReportGroupType) – Describes the function of the group. May be one of the following:
  REPORT_GROUP_HEADER = 1
  REPORT_GROUP_FOOTER = 2
  REPORT_GROUP_DETAIL = 3
  REPORT_GROUP_OTHER = 4
- *PositionExpression* (String) – An uninterpreted string describing the position of the group on the report.

Associations

- *ChildGroup* (ReportGroup, derived from Namespace.OwnedElement) – Used to links groups that are embedded in one another on the report.
- *Query* (ReportQuery) – Used to relate the group to the query used to derive the data in the group.

- *InputValue* (ReportField) – Used to show that a field (not directly owned by the current group) provides a value to the group for use in its query or calculation. As an example, an outer group displays customer information, and the customer ID is passed into an inner group to display invoice information.
- *Element* (ReportField, from Classifier.Feature) – Defines the fields in this section of the report.

14.3.7 ReportPackage

Each instance of this class describes a grouping of reports. Many products or companies will group reports by subject area or topic. This can also be used to represent the physical packaging (i.e., the report file).

Specializes

- Package (from Model Management)
- Surrogate (from Generic Elements)
- SummaryInformation (from Generic Elements)
- Component (from Core)

Associations

- *Report* (Report, from Namespace.OwnedElement) – Describes the ***Reports*** contained in a report package.

14.3.8 ReportQuery

Each instance of this class describes a query that is used by a report. The columns of the query can be used as the source for field derivations.

Specializes

- Query (from Schema Elements)

15 Business Engineering: Business Goals

15.1 Overview

Business goals describe the reason a business operates in a special market and why it operates in a specific way. The Business Goals Model enables the capture of unstructured information related to a business. It describes the goals of a business as well as the measures for their achievement.

The Business Goals sub-model is linked to all of the other sub-models of the Business Engineering Model and explains the purpose of structures and processes. The main meta data type of this model is the Goal, which is linked into a semantic network with other goals and supporting elements.

15.2 Semantics

The Business Goals package provides a set of elements that describe the goals of a business.

Vision and Mission are elements that document why a certain business and its processes and tasks exist. The Vision type is used to describe the purpose of a business or business process, while Mission documents the necessary achievements to fulfill the Vision.

A Goal is a desired state of a business that an individual or organization wishes to achieve. A goal can be expressed as a measurable set of steps (objectives) or by general visions and mission directives. The model captures goals of a business in a hierarchy of Goal instances with more general ones at the top and more specific ones (sub-goals) at the bottom.

The following figure shows goals for an Order Fulfillment Process.

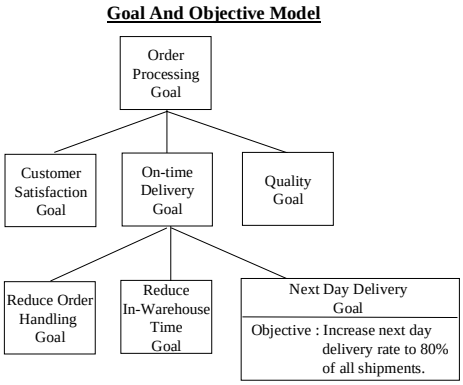


Figure 65: Goal and Objective Model

Goals are associated with other elements of the model using the concept of dependency, which indicates that the achievement of the goal depends on the outcome of a process or the performance of an business unit.

Goals can be structured in different ways. They may be decomposed into sub-goals to model the fact that abstract goals may be decomposed into or refined by more specific goals. Note that such decomposition can be a graph. In addition to the decomposition structure, the model allows the capturing of information about how different goals are related to each other. For example, a goal may support, may prevent the achievement of, or may be in conflict with, another goal.

Objective and Measure types are used to drive a process and track the achievement or non-achievement of targets. This is typically done against the information stored in a data warehouse. An Objective is a

measurable step to achieve a Goal. An Objective uses Measures to quantify the achievement or non-achievement of the quantified result (Goal).

A Measure in form of an Expression depends on one or more quantifiable business objects, which can be any of the Open Information Model meta data types that inherit from Classifier. Business objects therefore can be tables, OLAP cubes, components, executables, and so forth.

15.3 Class Reference

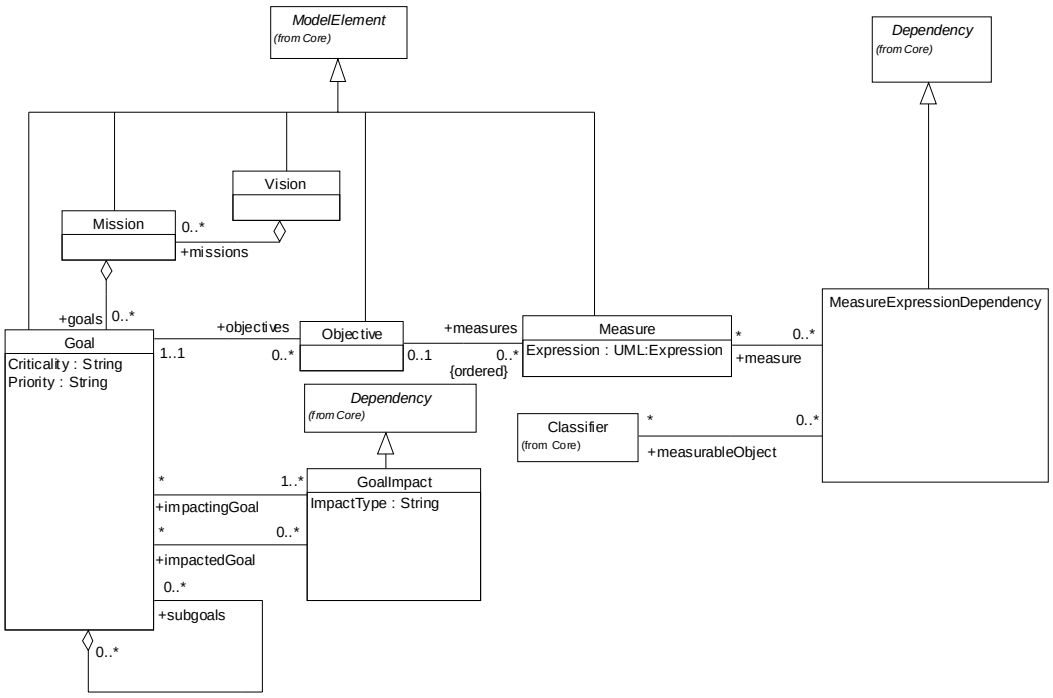


Figure 66: Goal and Measures

15.3.1 Goal

Goal captures the major goals a business has to achieve in order to fulfill its Mission. Goals, beside the textual description, can be further classified by providing priority and criticality information.

Specializes

- ModelElement (from UML)

Attributes

- *Name* (String) – Name of the Goal.
- *Comments* (String) – Additional unstructured information about the Goal.
- *ShortDescription* (String) – Description of the Goal.
- *Criticality* (String) – Describes the perceived criticality of the Goal (low, medium, high).
- *Priority* (String) – The priority of the Goal (low, medium, high).

Associations

- *subgoals* – Goals of which the Goal is composed.
- *objectives* – Collection of Objectives that need to be fulfilled to achieve the Goal.

15.3.2 GoalImpact

The GoalImpact class captures the different ways in which goals may interact.

Specializes

- Dependency (from UML)

Attributes

- *ImpactType* (String) – One of the following:
  - Supports – A Goal supports other goals in the decomposition or refinement structure.
  - Impedes – A Goal impedes other goals if it has a negative influence on them.
  - Conflicts – A Goal conflicts with other Goals, i.e., it prevents achievement of other goals.

Associations

- *ImpactingGoal* (Goal, derived from Dependency.supplier) – Source of the interaction between two goals.
- *ImpactedGoal* (Goal, derived from Dependency.client) – Destination of the interaction between goals, i.e., the Goal that is impacted by the source Goal.

15.3.3 Measure

Measure describes a quantifiable measure of an objective that can be based on the value of a classifier, for example, on the data maintained in a data warehouse.

Specializes

- ModelElement (from UML)

Attributes

- *Name* (String) – Name of the MeasureExpression.
- *Comments* (String) – Additional unstructured information about the MeasureExpression.
- *ShortDescription* (String) – Description of the MeasureExpression.
- *Expression* (UML::Expression) –String that represents the Expression of the measure.

15.3.4 MeasureExpressionDependency

MeasureExpressionDependency describes the relationship that links a MeasureExpression to a set of measurable classifiers in the Open Information Model.

Specializes

- Dependency (from UML)

Attributes

- *Name* (String) – Identifier of the MeasureElementDependency.
- *Comments* (String) – Additional unstructured information about the MeasureElementDependency.
- *ShortDescription* (String) – Description of the MeasureElementDependency.

Associations

- *Measure* – The quantifiable measure of the related business object.
- *MeasurableObject* – The instance of a class, which inherits from Classifier and provides the result to be measured.

15.3.5 Mission

Mission describes at an abstract level the means by which the Vision of an organization can be fulfilled. A mission is usually expressed as a set of Goals.

Specializes

- ModelElement (from UML)

Attributes

- *Name* (String) – Name of the Mission.
- *Comments* (String) – Additional unstructured information about the Mission.
- *ShortDescription* (String) – Description of the Mission.

Associations

- *Goals* – Set of Goals required to achieve the Mission.

15.3.6 Objective

An Objective describes a measurable step to achieve a Goal.

Specializes

- ModelElement (from UML)

Attributes

- *Name* (String) – Identifier of the Objective.
- *Comments* (String) – Additional unstructured information about the Objective.
- *ShortDescription* (String) – Description of the Objective.

Associations

- *Measures* – The quantifiable measures (MeasuerExpression) of the fulfillment of the Objective.

15.3.7 Vision

Vision captures the ultimate purpose of an organization and the associated business processes. It is a very high-level statement that needs to be further detailed in related mission statements that explain how it can be fulfilled.

Specializes

- ModelElement (from UML)

Attributes

- *Name* (String) – Name of the Vision.
- *Comments* (String) – Additional unstructured information about the Vision.
- *ShortDescription* (String) – Description of the Vision.

Associations

- *Missions* – Set of mission statements that indicate how the goal will be realized.

16 Business Engineering: Organizational Elements

16.1 Overview

The Organizational Elements package defines the resources and structures that are involved in the processes and activities of a business. Its main goal is to capture common information about organizational features relevant to business process and task modeling. The Organizational Elements package does not store or maintain complete organizational information, or serve as the model for extensive organizational research and analysis. Such functions are often performed by systems such as Enterprise Resource Planning (ERP) systems.

16.2 Semantics

The Organizational Model captures an organization's structure, resources, and jobs and describes the relationships to its market and industry. The most generic meta data type in the package is a Resource, which can play a specific role in a business process or in a relationship to the business process itself. A Resource is an abstract type and as such is never instantiated. It serves to capture the common characteristics of its sub-types PhysicalResource, InformationResource, and BusinessUnit.

Physical Resources are resources that exist in the real world, such as a conference room or an automobile. Information Resources carry information about business objects, such as a loan file or customer database. Note that both PhysicalResource and InformationResource types may play roles in a business and may be related to other resources, especially of BusinessUnit type. For example, a printer may belong to a specific department.

A resource may play a role regarding other resources in different contexts, such as an engineer who performs the administrator's role for a printer. The ResourceRole meta data type represents this kind or relationship.

A Business Unit captures the topological aspects of a business, its division into organizational functions, its geographical distribution, or the context in which it operates. BusinessUnit is an abstract type that is specialized into Industry and Business Unit. Business Units may be arranged into a hierarchy with more general entities at the top and more specialized ones towards the bottom.

A Business Unit may have a set of Policies that govern its activities. Policies may be decomposed into a hierarchy of sub-policies reflecting the fact that a Policy may consist of several other, more specific, Policies.

A business operates in an environment such as the market, the competitive landscape, the legal environment, and so forth. Industry is the generic meta data type that allows modeling of the business environment or market, such as its trading partners. Industries are Business Units and can therefore be arranged in a hierarchy with industries lower in the hierarchy being increasingly more detailed.

A Business Unit can model every organizational structure of a business such as a subsidiary, department, division, group, or team. Business Units can be arranged into hierarchies to reflect the reporting and management hierarchy found in a company. An business unit may contain other units or may contain the actual resources that perform Organizational Roles. The following figure shows an example organization with units and roles.

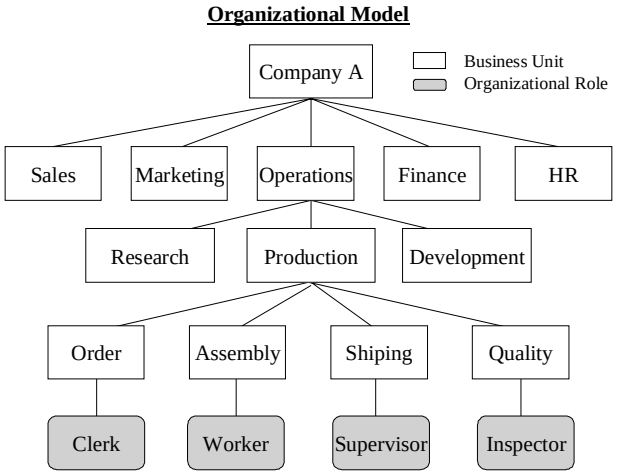


Figure 67: Organizational Model

A Resource can play one or more roles in an organization. For example, a software developer may also play the hardware administrator role. OrganizationalRole captures the specific jobs in a business and as such may have a Person assigned. The Person performs the OrganizationalRole.

BusinessUnits, OrganizationalRoles, and Persons can be related to Skills. Skill is a meta data type that describes required or offered abilities to perform tasks. A BusinessUnit may offer a set of skills, (e.g., a consulting company focused on a vertical market), whereas an Organizational Role may require a set of specific skills from its owner, (e.g., a bank teller must be able to count). The Person that owns a role or is part of a unit ultimately has to have the skills required or offered. The model allows the capturing and matching of skills at different levels.

Authority is a meta data type that captures the different authorities Business Units, Organizational Roles, and Persons may have assigned or exercise in an organization. Authority can be used to describe rights such as signing authority as well as more abstract things such as power and influence.

Key Persons or individuals are usually included in a business model to clarify roles and add meaning. The model is not intended to serve as a generic structure for maintaining the organizational information of a business in an operational environment.

The Organizational Model offers a core set of types to structure a business and to capture actors and resources. Business may have more detailed and specialized organizational types to model their structures. Such types can be captured using the extensibility mechanism (stereotypes) of the UML or by introducing new types that inherit from the core Organizational Model.

16.3 Class Reference

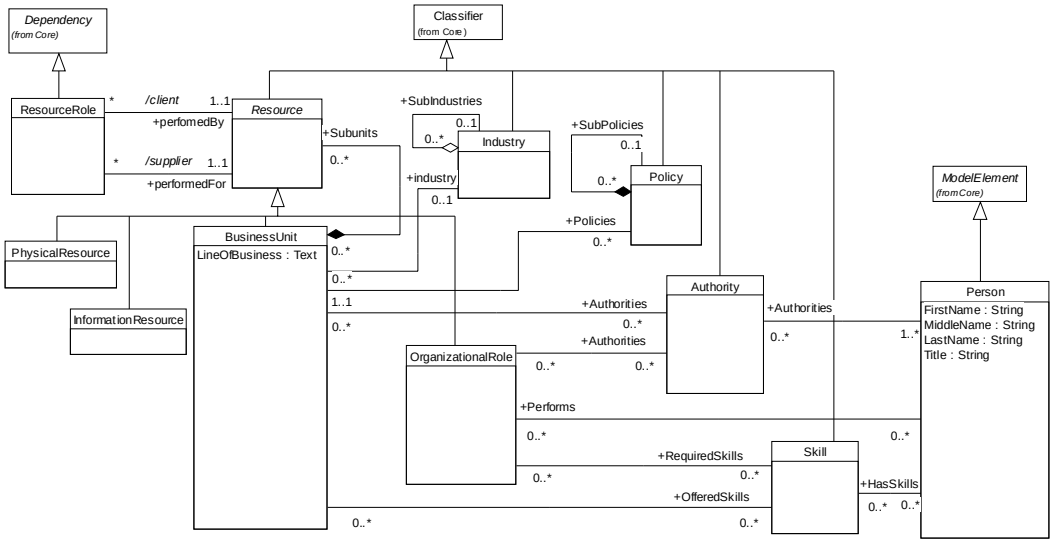


Figure 68: Organizational Definitions

16.3.1 Authority

Authority describes the type of authorization required to perform a Task, access rights for resources, responsibilities for specific tasks, or any combination of these elements. It also captures the more abstract definitions such as the power and influence a unit, role, or person may have.

Specializes

- Classifier (from UML)

Attributes

- *Name* (String) – Name of the Authority, such as Read or Write.
- *Comments* (String) – Additional unstructured information about the Authority.
- *ShortDescription* (String) – Description of the Authority.

16.3.2 BusinessUnit

BusinessUnit characterizes an industry, an organization, or department that has a goal of performing business activities. BusinessUnits can be arranged into a hierarchy to reflect the structure of industries or organizations. Because they are Resources, BusinessUnits may play specific ResourceRoles in relationship to other Resources.

Specializes

- Resource

Attributes

- *LineOfBusiness* (String) – Characterization of the major line of business for the BusinessUnit.

Associations

- *Subunits* – Set of BusinessUnits, Industries, or OrganizationalRole objects a BusinessUnit contains.

- *Policies* – Set of Policies defined for this BusinessUnit.
- *Authorities* – Set of Authorities this BusinessUnit has or requires.
- *OfferedSkills* – Set of Skills this BusinessUnit offers.
- *Industry* – Describes the industry in which the BusinessUnit participates.

16.3.3 Industry

An Industry describes a market segment. Each industry usually has a set of business processes that are typical for that industry. For example, the health care industry registers patients, confirms medical benefits, records medical histories, and tracks health care specialists. Within an industry, a process specific to the industry may be performed differently by individual businesses even though the basic process is the same. Therefore, an industry may have a set of business process templates that can be customized to specific companies.

Specializes

- Classifier (from UML)

16.3.4 InformationResource

Information resources carry information about business objects, such as a loan file or customer database.

Specializes

- Resource

16.3.5 OrganizationalRole

An OrganizationalRole represents one or more human resources exhibiting a specific set of skills within an organization. Typically any resource assigned to a particular OrganizationalRole can undertake a task or work item that requires a resource with the same set of skills.

OrganizationalRole forms a leaf node of an organizational hierarchy. It represents jobs that are performed by a Person linked to perform tasks. A role may be generic, such as a position or title, or more specific, such as a job description. OrganizationalRole captures the static knowledge about the tasks that a resource can perform. This knowledge is described by the set of Skills a role requires.

For example, the roles of an electronic technician can include:

- Troubleshooting to locate problems.
- Repairing faulty equipment.
- Reading and understanding wiring diagrams.

Specializes

- BusinessUnit

Associations

- *Policies* – The Policy set defined for this OrganizationalRole.
- *Authorities* – The Authority set this OrganizationalRole has or requires.
- *RequiredSkills* – The set of Skills an OrganizationalRole requires.

16.3.6 Person

Person describes a human actor that participates in a Business Process in one or more roles. Example individuals with specific skills and authorities are commonly included in a business process model for

clarification and in order to support simulation. Typically, the actual instances of Person will be contained in ERP systems or network directories.

Specializes

- ModelElement (from UML)

Attributes

- *Name* (String) – Identifier of the Person.
- *Comments* (String) – Additional unstructured information about the Person.
- *ShortDescription* (String) – Description of the Person.
- *FirstName* (String) – First part of the name of a Person.
- *MiddleName* (String) – Middle part or initial of the name of a Person.
- *LastName* (String) – Last part of the name of a Person.
- *Title* (String) – Title, such as PhD or Dr., of a Person.

Associations

- *HasAuthorities* – The Authority set a Person has.
- *HasSkills* – The Skill set a Person has.
- *Performs* (OrganizationalRole) – The set of roles a Person performs in an organizational structure.

16.3.7 PhysicalResource

PhysicalResource is a representation of an entity from the physical world, such as a conference room or an automobile.

Specializes

- Resource

16.3.8 Policy

Policy describes the rules that govern the actions of a Business Unit. It is also used to express policies from the outside world that affect industries or individual organizations. Policies can be recursively decomposed into a set of supporting sub-policies. Note that Policies may overlap with BusinessRules, which can also express policies within an organization.

Specializes

- Classifier (from UML)

Attributes

- *Name* (String) – Identifier of the Policy.
- *Comments* (String) – Additional unstructured information about the Policy.
- *ShortDescription* (String) – Description of the Policy.

Associations

- *composedOf* – Collection of sub-Policies of which the Policy is composed.

16.3.9 Resource

Resource describes an entity that participates in the tasks that constitute a Business Process or which may be related to the Business Process itself. Resource is an abstract type and therefore cannot be instantiated. It captures common features for its sub-types.

A resource may play a specific ResourceRole in relationship to other Resources, such as an administrator role for a physical resource such as a printer.

Specializes

- Classifier (from UML)

Attributes

- *Name* (String) – Name of the Resource.
- *Comments* (String) – Additional unstructured information about the Resource.
- *ShortDescription* (String) – Description of the function of the Resource.

16.3.10 ResourceRole

- ResourceRole describes the role played by a Resource in a context. For example, an Engineer organizational role may play the Administrator role for a Printer resource.

Specializes

- Dependency (from UML)

Associations

- *PerformedBy* (Resource, derived from UML:Dependency.supplier) – The Resource that performs the ResourceRole.
- *PerformedFor* (Resource, derived from UML:Dependency.client) – The Resource for which a specific ResourceRole is performed.

16.3.11 Skill

Skill describes the specific skills each role an organization requires or offers, or the specific skills a person has.

Specializes

- Classifier (from UML)

Attributes

- *Name* (String) – Identifier of the Skill.
- *Comments* (String) – Additional unstructured information about the Skill.
- *ShortDescription* (String) – Description of the Skill.

17 Business Engineering: Business Processes

17.1 Overview

The Business Process package provides meta data types to capture the semantic content of process models and associated structures in an object-oriented environment. Business processes are described by activities performed by resources and by transitions between activities.

Business Processes can be either long-lived, such as the budget supervision for a financial year, or relatively short-lived, such as the approval of an expense report.

Examples of business processes are:

- Software Development
- Project Management and Scheduling
- Order Entry or Fulfillment
- Resource and Product Planning

The Business Processes package extends UML 1.3 and therefore supports the more generic concepts of dynamic modeling in a seamlessly integrated way. This makes the model highly adaptable to individual methodologies and allows the use of UML concepts to develop more specialized models such as business interaction diagrams, and the use of case models and system decomposition diagrams.

The Business Process package includes concepts derived from the following sources:

- Deloitte & Touche's Notation (IndustryPrint)
- Ernst & Young's Notation
- UML (Unified Modeling Language) and the BPM extensions
- Flow Chart
- Gane-Sarson DFD (Data Flow Diagram)
- IDEF0
- Petri Net
- SAP EPC (Event Process Chain)
- SAD Actigram
- ISAC Activity Graph
- ICN (Information Control Nets)

17.2 Semantics

A Business Process representation for a specific application domain can be structured into a model of the tasks, a model of the available business processes, and the behavioral representation of the business process as graph of states and transitions.

Tasks and Business Processes are modeled as services provided by some higher-level structure such as an Industry or Business Unit. In terms of the UML, both types of services are described as operations exposed by a Classifier. Industry and Business Units are specializations of Classifier.

Tasks are services exposed by Industries or BusinessUnits. Tasks can be either Manual Tasks or Automated Tasks depending on whether they are to be executed manually or performed by computer. Tasks can be decomposed into sub-tasks along different dimensions such as time, goal, or resources.

Tasks are clearly separated from the state/transition graph that represents the behavior of the Business Process. This separation allows the reuse of tasks by one or more Business Processes.

A Task is usually an activity that is performed with a specific Goal in mind. The operation, “deleting a file” could be the task of “cleaning one’s hard-drive” or the task of “protecting confidential information”. Depending on the granularity of the task model and the level of reuse, tasks and goals can be separated and the structures of the Business Goal Model used to represent and associate goals.

Business Processes may correspond to services provided by Industries or BusinessUnits. A Business Process (like an Operation) represents the signature or entry point and can have multiple implementations in form of Business Process Methods. A BusinessProcessGraph, i.e. the graph of states and transitions, defines the behavior of this implementation.

A BusinessProcessGraph is represented by a set of Task States and the Transition of control as well as the Data Flow between these states. The individual states refer to the Tasks to be performed and the Resources that perform the Tasks. A process is initiated with a well-defined state, i.e. the flow begins with a single Initiator state. From this pseudostate the flow follows Transitions to other states or pseudo states. A Transition may be guarded by Boolean expressions allowing it to fire or not. Depending on the result (True or False), a path through a Transition is taken or not taken. States can be the source of many guarded transitions, which allow control of the execution path through a process.

The following figure shows a business process model expressed using the meta data types discussed in this section:

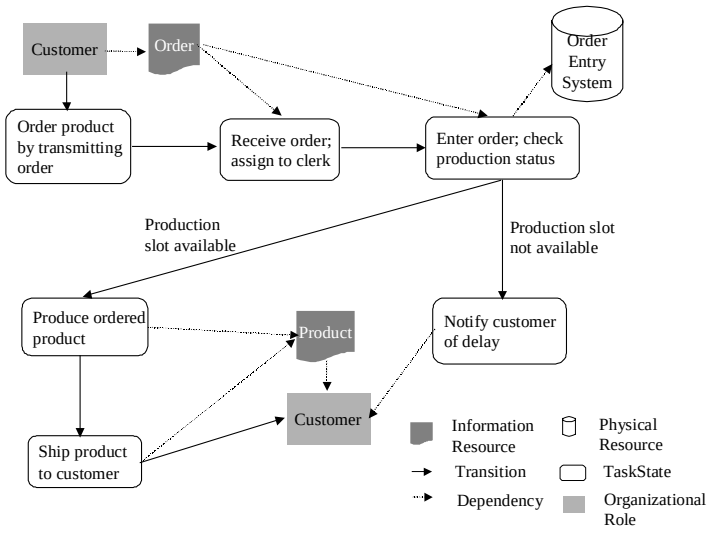


Figure 69: Business Process Model

Process flow can be influenced by two special meta data types, Fork and Join, which allow one to create and combine parallel execution of tasks. In a Business Process Graph, States can be either production activities or coordinating activities. Production activities (Data Flow) modify the environment and manipulate information or materials whereas Coordinating activities (Task States) control the flow of the process and do not change the environment.

A Business Process Graph terminates after all its TaskStates have been completed, i.e. all parallel process flows have reached a Terminator.

A Process Partition is a clustering of Task States that constitutes a business process graph. This figure shows the partitioning of an example business process graph.

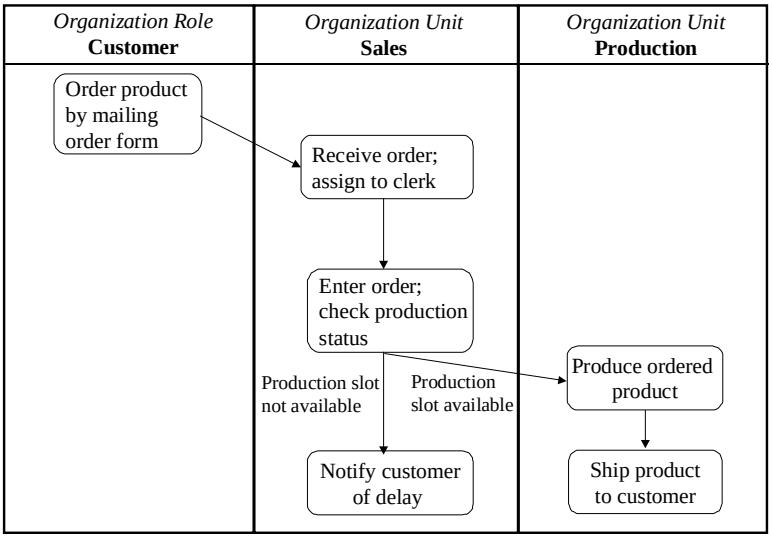


Figure 70: Process Partition

Each partition represents the responsibilities of a Business Unit in performing the tasks. Process Partitions separate the tasks performed by separate Business Units into concurrent flows loosely synchronized by transitions. The Business Process Model allows Task States to belong to multiple Partitions.

17.3 Class Reference

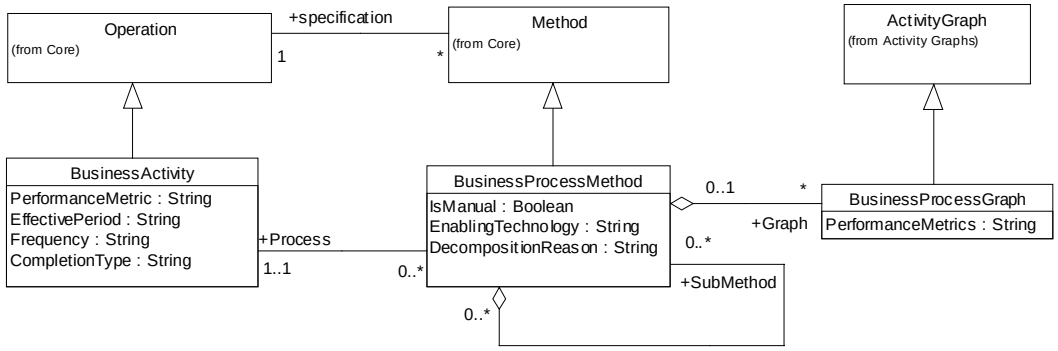


Figure 71: Process Definitions

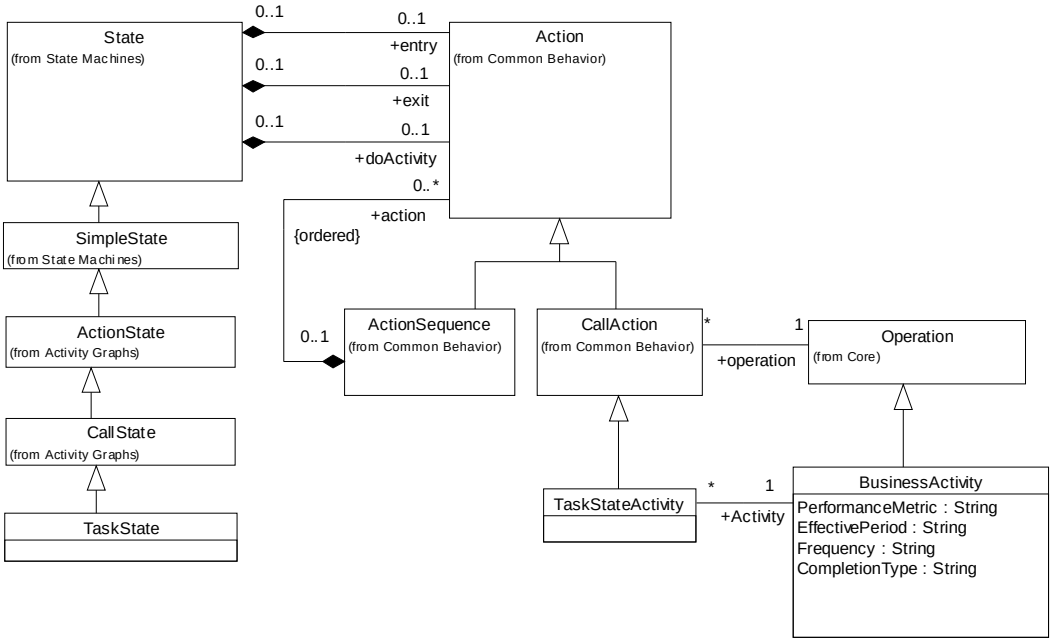


Figure 72: Task Definitions

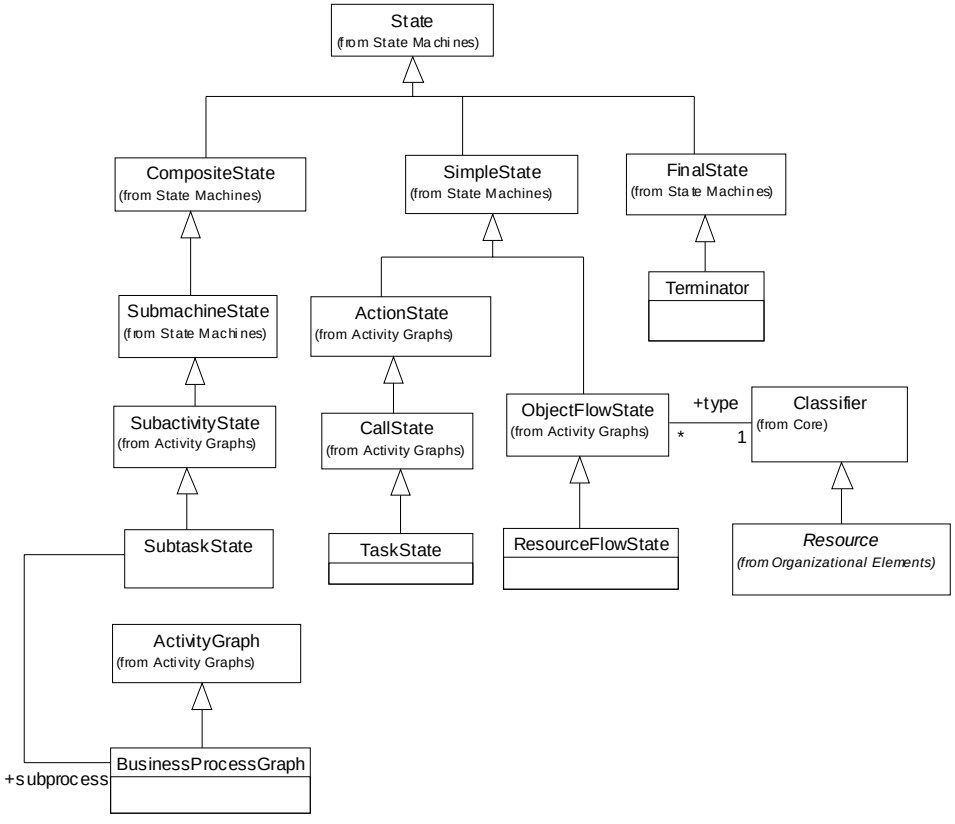


Figure 73: State Definitions

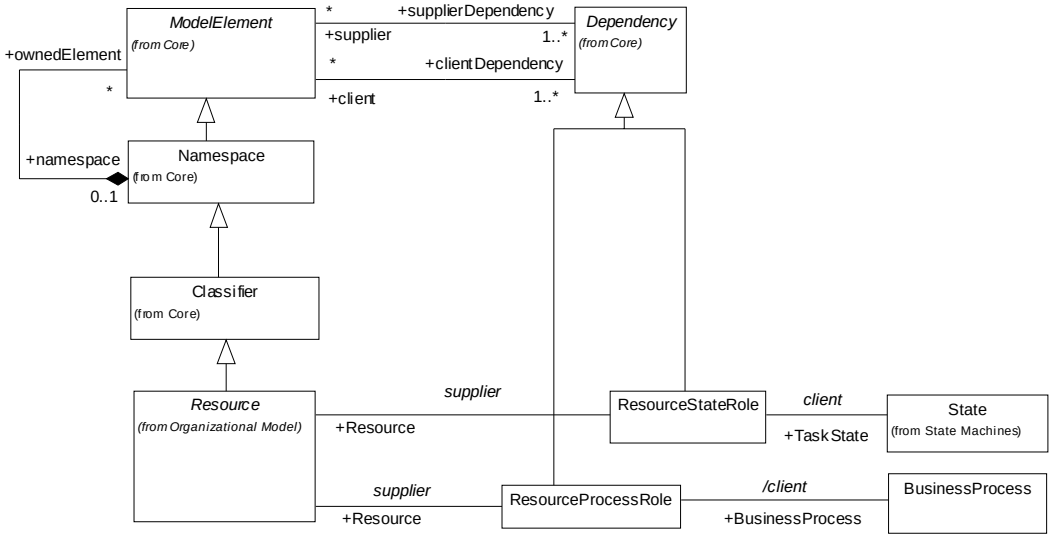


Figure 74: Resource Role Definitions

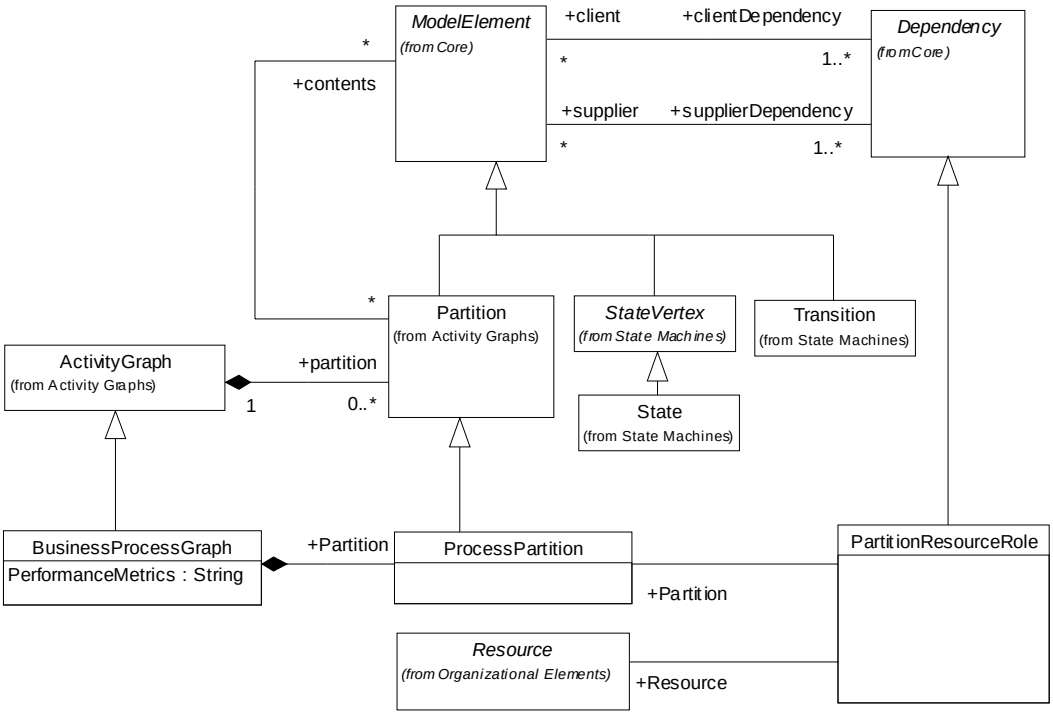


Figure 75: Process Partitions

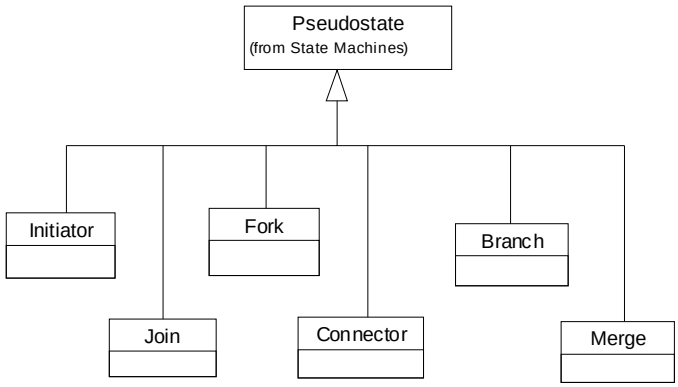


Figure 76: Pseudostates

17.3.1 BusinessActivity

A business activity is a service offered by a business object (e.g. Industry or BusinessUnit). If a BusinessActivity is represented by a process graph, the activity may references one or more implementations by BusinessProcessMethods, which in turn link to the BusinessProcessGraph that specifies the behavior as states and transitions.

A BusinessActivity may also be the specification of a logical work item within a BusinessProcessGraph. It forms the operation for a specific TaskState and related TaskStateActivity.

Specializes

- Operation (from UML)

Attributes

- *Name* (String) – Name or identifier of the activity.
- *Comments* (String) – Additional unstructured information about the activity.
- *ShortDescription* (String) – Description of the activity or what problem it addressees.
- *Frequency* (String) – Provides information about the frequency of execution of the activity. The might be an absolute – “every morning” – or relative – “with every shipment”.
- *PerformanceMetrics* (String) – Information relating to cost, cycle time, and other performance requirements. These may be related to a specified Goal.
- *CompletionTime* (String) – Information about the estimated, permissible, or typical time for the completion of this Task.
- *EffectivePeriod* (String) – Indicates when this activity becomes effective and for how long.

17.3.2 BusinessProcessGraph

A BusinessProcessGraph is defined by a network of TaskStates. TaskStates are linked by Transitions that may have Guards to control the flow of execution. Task States reference BusinessActivities through a TaskStateActivity, and a BusinessActivity can be referenced by multiple TaskStates. Special nodes in the process graph act as Initiator and Terminator of the process flow, or Fork and Join of parallel execution threads. TaskStates also relate resources in a specific role in the execution of the process.

Specializes

- ActivityGraph (from UML)

Attributes

- *Name* (String) – Name of the BusinessProcessGraph.
- *Comments* (String) – Additional unstructured information about the BusinessProcessGraph.
- *ShortDescription* (String) – Description of the BusinessProcessGraph.

Associations

- *Partitions* (ProcessPartition, derived from ActivityGraph.Partition) – The set of partitions contained within this BusinessProcessGraph.

17.3.3 BusinessProcessMethod

A BusinessProcessMethod can be executed automatically or manually. Automated methods can be performed by information systems and may have their implementation specified in a BusinessProcessGraph.

Attributes

- *DeompositionReason* (String) – Provides a rational for the decomposition of the method into sub-methods.
- *EnablingTechnology* (String) – Identifies the underling system that enables processing, e.g., a specific ERP system.
- *IsManual* (Boolean) – Whether the method is performed by a computer system. A process definition may include such types to provide a complete description, but the execution of the method is outside the scope of an automatic system, i.e., it is assumed that the specified resources perform the method.

Associations

- *SubMethod* (BusinessProcessMethod) – Set of child methods that decompose the method.

17.3.4 Branch

Indicates a decision point for flow-of-control within a business process graph. Unlike a fork, only a single path will be taken.

Specializes

- Pseudostate

17.3.5 ResourceFlowState

A ResourceFlowState describes the flow of information or physical resources associated with the flow of control through Activities and Transitions. A transition from a TaskState to a ResourceFlowState indicates that the Resource is the product of the task. A transition from a ResourceFlowState to a TaskState indicates that the resource is used as the input to the task.

Specializes

- ObjectFlowState (from UML)

Attributes

- *Name* (String) – Name of the ResourceFlowState.
- *Comments* (String) – Additional unstructured information about the ResourceFlowState.
- *ShortDescription* (String) – Description of the ResourceFlowState.

Associations

- *outgoing* – Specifies the Transition departing from the ResourceFlowState to an Activity.
- *incoming* – Specifies the Transitions entering the ResourceFlowState emanating from an Activity.
- *Container* (CompositeState, derived from StateVertex.Container) – The SubProcess that contains this ResourceFlowState, if any.

17.3.6 Fork

Fork describes the split of control flow into several parallel execution threads. A Fork is the target of a single Transition and is the source of two or more outgoing Transitions.

Specializes

- Pseudostate (from UML)

17.3.7 Initiator

Initiator is a pseudo state in a process that describes the single start point for the execution. An Initiator has no incoming Transitions.

Specializes

- Pseudostate (from UML)

17.3.8 Join

Join describes the junction of several parallel threads of execution in a process flow. A Join is the target of two or more Transitions and the source of a single outgoing Transition.

Specializes

- Pseudostate (from UML)

17.3.9 Merge

Describes the combination of two paths of execution into a single path. Unlike a join, it is not synchronous, it can join non parallel thread paths. It is the opposite of the Branch pseudostate.

Specializes

- Pseudostate (from UML)

17.3.10 PageConnector

A PageConnector describes a simple node between Transitions. It allows tools to have a process diagram span multiple pages or sections. An instance of the PageConnector pseudostate could tell the tool to continue the model on the next page starting with the transition after the connector (i.e., a page break). Tools that are not concerned with pagination can simply ignore the connector.

Specializes

- Pseudostate (from UML)

17.3.11 PartitionResourceRole

PartitionResourceRole associates one or more Resources to a Partition in different roles. For example, an OrganizationalRole may be the owner of a set of activities that are owned by a Partition.

Specializes

- Dependency (from UML)

Attributes

- *Name* (String) – Name of the PartitionResourceRole.
- *Comments* (String) – Additional unstructured information about the PartitionResourceRole.
- *ShortDescription* (String) – Description of the PartitionResourceRole.

Associations

- *Partition* – Partition with which the specific Resource is associated.
- *Resource* – Resource that is associated with a Partition.

17.3.12 ProcessPartition

A ProcessPartition groups TaskStates, normally with respect to their responsibility. ProcessPartitions are used for several different purposes, for example, to group a set of actions functionally or to show in which part of an organization an action is performed.

Specializes

- Partition (from UML)

Attributes

- *Name* (String) – Name of the ProcessPartition.
- *Comments* (String) – Additional unstructured information about the ProcessPartition.
- *ShortDescription* (String) – Description of the ProcessPartition.

Associations

- *contents* – The set of process elements (State types) associated with this ProcessPartition.

17.3.13 ResourceProcessRole

ResourceProcessRole describes the role a Resource has regarding a Business Process Graph. This might include roles such as owner, steward, contact person, or administrator (if the process is automated).

Specializes

- Dependency (from UML)

Attributes

- *Name* (String) – Identifier of the ResourceProcessRole.
- *Comments* (String) – Additional unstructured information about the ResourceProcessRole.
- *ShortDescription* (String) – Description of the ResourceProcessRole.

Associations

- *BusinessProcess* – The BusinessProcess that the Resource is related to in a specific Role.
- *Resource* – The Resource that is associated with a BusinessProcess in a specific Role.

17.3.14 ResourceStateRole

ResourceStateRole associates workflow participants (Resources) to a collection of business process TaskStates. The role defines the context in which the Resource participates in a particular activity such as responsibility or authority.

Specializes

- Dependency (from UML)

Attributes

- *Name* (String) – Identifier of the ResourceStateRole.
- *Comments* (String) – Additional unstructured information about the ResourceStateRole.
- *ShortDescription* (String) – Description of the ResourceStateRole.

Associations

- *TaskState* – The TaskState (State) that the Resource is related to in a specific Role.
- *Resource* – The Resource that participate in a TaskState (State) in a specific Role.

17.3.15 SubTaskState

SubTaskState represents the execution of sub-process within a parent process graph. It is a structuring mechanism enabling hierarchically structured complex flows and facilitates the reuse of predefined processes. A SubTaskState may reference either the composition of other states and transitions or a defined BusinessProcessGraph.

Specializes

- SubactivityState (from UML)

Attributes

- *Name* (String) – Name of the SubTaskState.
- *Comments* (String) – Additional unstructured information about the SubTaskState.
- *ShortDescription* (String) – Description text about the SubTaskState or what problem it addresses.
- *PerformanceMetrics* (String) – Information relating to cost, cycle time, and other performance requirements.
- *EnablingTechnology* (String) – Identifies the underlying system that enables processing, e.g., a specific transaction of an ERP system.
- *EffectivityPeriod* (String) – Indicates when this process SubTaskState becomes effective and for how long.

Associations

- *subprocess* (BusinessProcessGraph, derived from UML:SubmachineState.submachine)– The business process graph to be substituted in place of this state.
- *subvertex* – The set of subprocess elements (TaskState, SubTaskState, DataFlow, etc.) owned by the SubTaskState if it is a composition.

17.3.16 TaskState

A TaskState models a state of a business process. A State becomes active when it is entered because of incoming transitions. The state executes the related TaskActions and passes control to its outgoing Transitions it the actions have been completed.

Specializes

- ActionState CallState (from UML)

Attributes

- *Name* (String) – Name or identifier of the Task.
- *ShortDescription* (String) – Description of the Task or what problem it addressees.

Associations

- *incoming* – Set of incoming Transitions for the TaskState.
- *outgoing* – Set of outgoing Transitions for the TaskState.

17.3.17 TaskStateActivity

TaskStateActivity represents the execution of a BusinessActivity in a specific TaskState. A BusinessActivity can be re-used by multiple TaskActions, i.e. may be related to multiple different Task States. The invocation of the Task may be either synchronous or asynchronous, indicating whether the TaskState waits for the execution to be finished or not.

Specializes

- CallAction (from UML)

Attributes

- *Name* (String) – Name or identifier of the activity.
- *IsAsynchronous* (Boolean) – Indicates if the related BusinessActivity is activated synchronously (True) or asynchronously (False).

Associations

- *Activity* (BusinessActivity, derived from CallAction.Operation)– The business activity that is invoked by the TaskStateActivity.

17.3.18 Terminator

Terminator is a pseudo state in a process that identifies one of the possible end points. A business process terminates when all parallel flows have reached a Terminator.

Specializes

- FinalState (from UML)

Attributes

- *Name* (String) – Name of the Terminator.
- *Comments* (String) – Additional unstructured information about the Terminator.
- *ShortDescription* (String) – Description of the Terminator.

Associations

- *incoming* – Specifies the Transition entering the Terminator.
- *container* – The SubProcess that contains this Terminator, if any.

Constraints

- A Terminator has no outgoing Transitions.

17.3.19 Transition

Transition is a directed relationship between a source TaskState and a target TaskState.

Specializes

- ModelElement (from UML)

Associations

- *guard* – The Guard or logical expression that determines if the transition is taken.

18 Business Engineering: Business Rules

18.1 Overview

A *business rule* is a statement that controls or defines some aspect of a business. It either asserts the structure of a business or governs its business processes. The Business Rules package provides meta data types to capture, classify, and store business rules. Scenarios supported by the package are:

- Capturing tools used by analysts to describe and document the rules of a business.
- Interchanging business rule definitions between capturing tools, business process modeling environments, and back-ends such as workflow engines

The package is an integrated part of the Business Engineering Sub-model of the Open Information Model. This makes the model highly adaptable to individual methodologies and allows the use of UML concepts to develop more specialized models.

The Business Rules package includes concepts derived from the following sources:

- UML 1.3
- GUIDE Business Rule Project

18.2 Semantics

A b*usiness rule* describes how to transition from one state to another or how to prohibit such a transition. As such it is a declarative statement rather than a procedural description. Business rules are highly structured atomic statements that are usually extracted from informal documentation found in a business.

The b*usiness rambling* has been introduced in the model to support the process of isolating business rules. It is the starting point for deriving business rules and as such may contain more than a single rule, may be inconsistent, contradict other ramblings, and even be untrue. The model allows specifying the source of a business rambling by relating it to a *resource*, a meta-data type defined in the Organizational Model.

A formalized representation of a Business Rule can be categorized into the following types:

- *Term Rule* – Introduces the definition of a term into a business term dictionary. It is used to define the vocabulary of a business.
- *Fact Rule* – Documents the connections between items. Examples are relationships between entities, e.g., the “belongs to” relationship between an attribute and an entity.
- *Action Rule* – Action rules are concerned with the invocation of actions. They state the conditions under which actions must be taken; this includes pre-conditions, post-conditions, and triggering conditions.
- *Inference Rule* – Describes the inference or derivation of a business rule from other rules or by mathematical calculations. Inference rules are sometimes called d*erived rules* or *Derivation Rules* because they capture knowledge that is dynamically derived instead of explicitly stored.

Business Rules may be grouped into Business Rule Sets that reflect a simple sequencing, a specific business area, implementation considerations, or organizational as well as project structures. Sets organize business rules into manageable groups.

A BusinessRule may interact with other rules by supporting, conflicting, or subsuming them. This fact is modeled by the Impacts Rule relationship, which can be used to define a network of semantic relationships in-between rules.

BusinessRules are extracted from the informal knowledge that governs a business. They usually have one or more sources, a steward, and one or more supporters. A Resource in an organization may play different

roles regarding a Business Rule. These roles are captured by the relationship type, ResourceRuleRole. This relationship also allows a resource to register interest in a rule and how it evolves, and information that can be used by a system to send out notifications.

The abstract meta-data type BusinessRule has RuleTypes that capture the different types of business rules listed above. A DefinitionRule is an expression that introduces a business term into the vocabulary of a business model in a specific context. The context in which a term may exist controls its definition. For example, the term “Table” may have a vastly different meaning in the context “Data Warehouse” than in the context “Furniture Warehouse”. The Knowledge Representation Model presents a meta-data type Term that fits the context-to-term framework used by DefinitionRule and could be referenced by standard UML Dependency instances.

A FactRule establishes a form of relationship between two or more business terms. Types of relationships include Aggregation, Association, Generalization, and so forth. A Fact Rule, for example, may state the fact that a rental car (term) has a license plate (term).

ActionRules are statements that are concerned with the invocation of actions. Action Rules capture the condition under which an activity has to occur, i.e., the events, pre-conditions, and post-conditions that must hold before and after the rule has been applied.

A constraint is a special type of Action Rule. A constraint is a condition that must evaluate to True. Constraints may be static or transitional. Static constraints are structural and time independent; they must hold at any point in time. Transition constraints assert the dynamic integrity of a system; they are behavioral in nature and restrict the transitions from one state of the system to another.

InferenceRules are statements that express knowledge in terms of information items that are already present in the model of a business. InferenceRules capture structural domain knowledge that does not need to be stored explicitly because it can be dynamically derived from existing or other derived information. For example, if a person’s birth date is known, then the person’s age can be calculated (mathematical derivation). Another example is that a student with 32 to 64 credits is known as a “sophomore” (logical derivation).

18.3 Class Reference

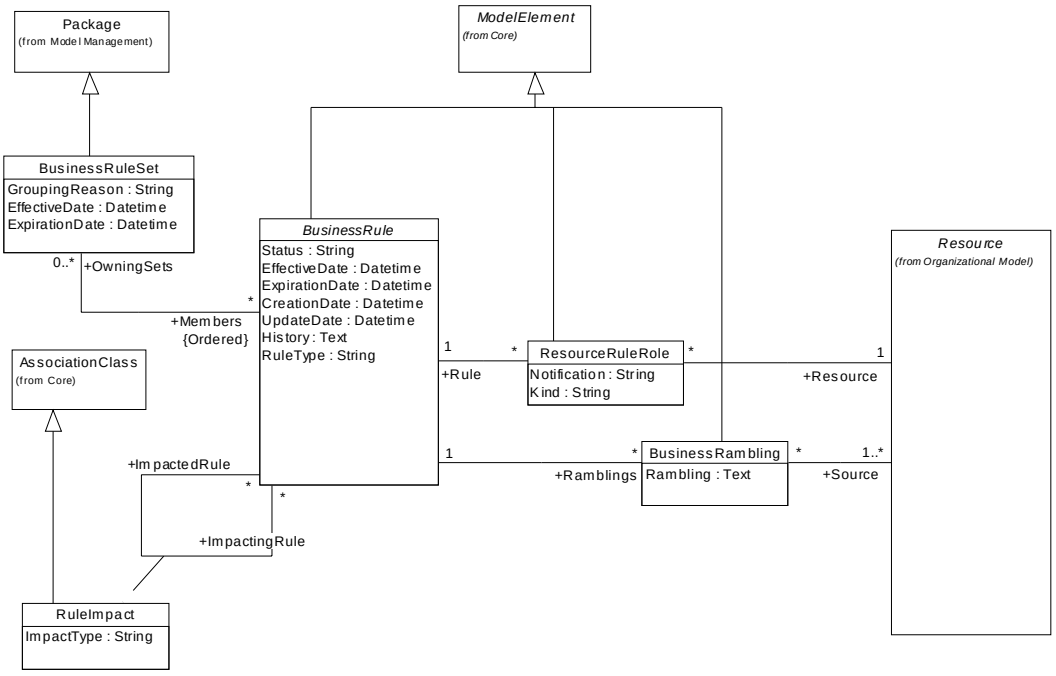


Figure 77: Core Definitions

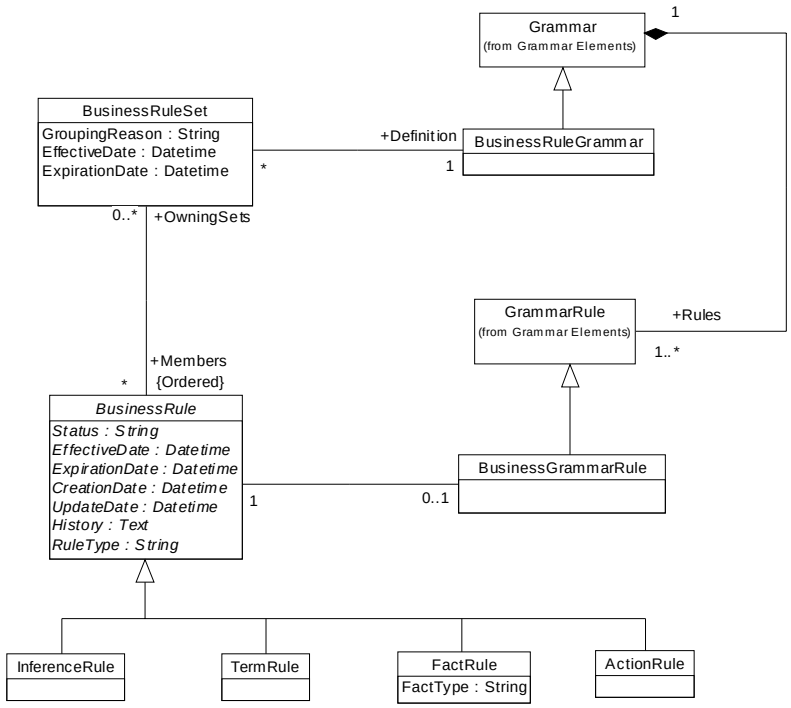


Figure 78: Rule Type Definitions

The following sections describe the different meta-data types of the Business Rule Model in alphabetical order.

18.3.1 ActionRule

ActionRule describes actions and their conditional invocation as well as the special case of constraints.

Specializes

- BusinessRule

18.3.2 BusinessRambling

BusinessRambling is an unstructured piece of information about a business. It is the starting point for deriving business rules, and may contain more than a single rule, be inconsistent, contradict other ramblings, or even be untrue.

Specializes

- ModelElement (from UML)

Attributes

- *Rambling* – A textual representation of the business rambling as stated by a source.

Associations

- *Source* – The Resources that state the business rambling. It is important to record these sources of the rambling so that details can be clarified.

18.3.3 BusinessRule

BusinessRule is a statement of a rule under which a business operates. Its classification is further defined by RuleType.

Specializes

- Constraint (from UML)

Attributes

- *Name* – An English phrase that describes the purposed of the business rule. The name should be worded in noun form (Order Number Validation) rather than verb form (Validate Order Number).
- *ShortDescription* – A declarative expression of the business policy that the rule enforces. For example, “All prescriptions for schedule 2 drugs shall be verified with the prescribing doctor.”
- *Status* – The status of the rule. Valid values for the rule status are the following:
  - *Proposed* – A potential rule that has been discovered by any of the normal means, such as code scan, extraction from business ramblings, interviews, and so forth.
  - *Validated* – Indication that the potential rule has been reviewed by a business analyst and determined preliminarily to be valid.
  - *Approved* – Indication that the business owner or steward has approved the rule. The implication of an “approved” status is that a business owner or steward has been assigned.
  - *Archived* – Business rules can change. When they do, the old rule should be kept around, but put into an “archived” status. The archived status should be connected to the new version of the business rule via the “version of” link.
- *EffectiveDate* – The date on which the business rule becomes effective. The primary purpose of this field is to indicate when business rules will become effective at a future date (perhaps due to

pending legislation). A blank value indicates that the business rule was effective prior to being put into the repository.

- *ExpirationDate* – Business rules may expire, i.e., become no longer effective. This attribute documents the date on which the business rule is no longer valid.
- *CreationDate* – The date on which the rule was entered into the business model.
- *UpdatedDate* – The date on which the rule was last updated.
- *History* – Documentation of the evolution of the rule to its present state.
- *RuleType* - The intended use of Rule. Valid values for the RuleType are:
  - *DefinitionRule* – Introduces the definition of a term into a business term dictionary. It is used to define the vocabulary of a business.
  - *FactRule* – Documents the connections between items. Examples are relationships between entities, such as Aggregation, Association, Generalization and Feature.
  - *ActionRule* – Action Rules are concerned with the invocation of actions. They state the conditions under which actions must be taken, this includes pre-conditions, post-conditions, and triggering conditions.
  - *InferenceRule* – Describes the inference or derivation of a business rule from other rules or by mathematical calculations. InferenceRules capture knowledge that is dynamically derived instead of explicitly stored.

Associations

- *Ramblings* – Set of BusinessRamblings from which the rule was extracted.
- *ImpactedRule* – A set of rules affected by this rule.
- *ImpactingRule* – A set of rules that affect this rule.

18.3.4 BusinessRuleSet

BusinessRuleSet is a grouping of related BusinessRules into meaningful sets. The grouping might reflect a simple sequencing, a specific business area, implementation considerations, organizational structures, or project structures.

Specializes

- *ModelElement* (from UML)

Attributes

- *GroupingReason (String)* – A descriptor that denotes the reason why the rules were grouped together. Possible values might include:
  - Order – The rules must be tested or executed in a certain sequence.
  - Business Area – The rules are all related to a particular line or area of the business.
  - System Implementation – The rules are all implemented within a certain system.
  - Project Implementation – The rules are all implemented within a certain project.
  - Organization Implementation – The rules are all implemented within a certain project.
- *EffectiveDate* – The date on which the rule grouping became effective.
- *ExpirationDate* – The date on which the rule grouping is no longer valid. This might occur because the grouping was due to a project, and the project has been completed.

Associations

- *Rules* (BusinessRule) – Set of ordered business rules grouped into the rule set.

18.3.5 FactRule

A FactRule establishes a form of relationships between two or more terms. Types of relationships include Aggregation, Association, Generalization, etc., and they are indicated by a property of the FactRule.

Specializes

- BusinessRule

Attributes

- *FactType* – The following lists the most generic relationship types:
  - *Aggregation* – (part-of) Expresses the fact that one Term is a component of the other one and that they form a whole.
  - *Association* – (associated-with) Expresses a generic type of relationship between Terms.
  - *Generalization* – (is-a) Expresses a specialization of a Term by another Term.
  - *Feature* – (member-of) Expresses that an attribute or operation belongs to an entity.

18.3.6 InferenceRule

An InferenceRule is a rule that describes how information is derived from existing structures and terms of a business. It allows capturing domain knowledge that is computed rather than persisted. For example, if a person's birth date is known, then the person's age can be calculated. Another example is that a student with 32 to 64 credits is known as a "sophomore".

Specializes

- BusinessRule

18.3.7 ResourceRuleRole

ResourceRuleRole describes the role a Resource plays for a specific business rule.

Specializes

- ModelElement (from UML)

Attributes

- *Kind* – Type of the role:
  - *RuleSource* – An InformationResource maybe the source of the business rule or rambling. This is normally a document (such as a manual, program code, or official policy) or an OrganizationalRole, such as CTO. OrganizationalRoles can be related to Persons, i.e., BusinessRules may be related to individuals through their specific organizational role.
  - *RuleSteward* – The responsibility that a BusinessUnit (or a related Person) has for a business rule. For a given rule, either an OrganizationalRole or an BusinessUnit can fulfill this particular role.
  - *RuleRequestor* – The BusinessUnit that requested that the rule be added to the repository. This may not be known or may be redundant to the rule source, in which case no requestor should be defined.

- *RuleAdministrator* – OrganizationalRole (and related Person) that entered the rule into the repository. This is the OrganizationalRole to which questions can be directed concerning the way in which the rule was entered.

- *Notification* – Type of notification a Resource wants to receive if the state of the related BusinessRule changes:
  - *Approval* – The rule cannot be changed or deleted without the approval of the resource.
  - *Notify* – The resource needs to be notified before the rule is modified.
  - *Interested* – The resource can be notified after the rule is modified
  - *Validate* – A resource that needs to be notified before a rule is activated. This second resource has to agree that the rule is correct.

Associations

- *Rule* – The BusinessRule for which a Resource plays a specific role.
- *Resource* – Resource that plays a specific role for a BusinessRule.

18.3.8 RuleImpact

The RuleImpact class describes semantic relationships between BusinessRules. For example, a rule may support or conflict with another rule.

Specializes

- ModelElement (from UML)

Attributes

- *ImpactType* (String) – The reason for the relationship between the rules. Valid reasons are:
  - *Redundant* – A rule is covered by one or more other business rules.
  - *Supports* – A rule is decomposed into several supporting rules.
  - *Conflicts* – A rule has a negative impact on another rule.
  - *Subsumed* – A rule has been replaced by another business rule.
  - *Variant* – A business rule has been customized and is a variant of an existing one.

Associations

- *ImpactingRule* – The BusinessRule that impacts another rule.
- *ImpactedRule* – The BusinessRule that is impacted by another rule.

18.3.9 TermRule

A TermRule defines a term, a symbol, word or phrase that has a specific meaning for a business. A term is defined by a term rule in a specific context, which makes the meaning unique. Process, for example, has vastly different meanings in operating system environments and car manufacturing businesses.

Specializes

- BusinessRule

19 Knowledge Management: Knowledge Descriptions

19.1 Overview

Knowledge Management (KM) is the systematic approach of capturing, organizing, and using the information resources of an enterprise to add business value and achieve strategic market advantages. A KM environment usually consists of a combination of different systems, such as Enterprise Resource Planning (ERP) systems, Data Warehouses (DW), Document Management (DM) systems, Groupware applications, and Intranets.

Sharing and collaboration of knowledge amassed in information systems across organizational and geographical boundaries of an enterprise requires an efficient mechanism to find and access relevant data. Knowledge portals, which in their simplest form can be viewed as giant resource directories, offer the entry points into information resources for users, groups, and communities with common interests.

At the core of a Knowledge Portal lies the cataloging and categorization of information using a consistent taxonomy that reflects a business or user specific view. A taxonomy is a description of domain specific concepts and their relationships, covering such areas as financial services, health care, commodities, sales and marketing and including such concepts as Bond, Benefit, Country, and Product.

A consistently applied taxonomy can be used to improve upon the usual keyword and full text based techniques. It allows a knowledge worker to retrieve information using business standard terminology and avoids problems of poor selectivity and quality of results caused by missing, inconsistent, or conflicting terminology.

As an example, a simple textual search for the term “table” may yield a result set that covers furniture items as well as relational database definitions, e.g. the order entry table. It would be up to the user to sort through the set of items and determine their relevance. However, if “table” items had been classified either as furniture or data definitions, then the retrieval results would have been of much higher quality for the end user.

Additional information such as synonyms (Customer ~ Client), abbreviations (Department ~ DEP), or preferred terms provided by the taxon would allow the system to offer an even higher degree of precision and user-friendliness. Information about the semantic relationships between different search terms enables the system to automatically adjust the query to include or exclude certain concepts.

A more significant benefit of a taxon is that it often reflects the dimensions a business uses to track unstructured as well as structured information. For example, a taxon for the support department might define product, problem, and resolution. A support person may search for the resolution of a specific problem for an individual product and then pivot the view to search for related problems in other products that are also solved by the fix.

The introduction of business terminology and taxonomies in an enterprise requires the alignment of categorizations and controlled vocabularies between its information systems. The first step in this process is to enable the interchange of such definitions through a standard format. The goals of the Knowledge Description Model are to provide the basic mechanisms to define or interchange:

- Representations of a basic meta data schema for knowledge, i.e. schemas for the representation of meta data about unstructured or semi-structured data.
- Structures to classify content into sets of related concepts that describe the meaning of real world entities.
- Descriptions of taxons or controlled vocabulary. The representation of the controlled vocabulary consists of sets of terms arranged into a hierarchy of glossaries.

Note that the Knowledge Description Model does not define the schema or the vocabulary in a specific or vertical knowledge domain. The model instead provides the basic mechanisms to describe such schema and vocabulary in order to maintain them by or interchange them between computer systems.

The Knowledge Description Model package includes concepts derived from the following sources:

- Resource Description Framework (RDF)
- Knowledge Interchange Format (KIF)
- UML (Unified Modeling Language)

19.2 Semantics

The Knowledge Descriptions package provides meta data types to describe and categorize information managed by computer systems. It deals with topics interesting to humans modeled as *concepts*. Users can then choose familiar *terms* to refer to these concepts. Most likely, the terms a user chooses to identify an individual concept will be a domain-specific subset of a larger set of terms that can refer to the same concept.

The definition of a controlled vocabulary for a set of concepts reduces ambiguity and complexity for the user. The package provides three main structural features to express a controlled vocabulary and the related semantics:

- Thesaurus – is a collection of Concepts that provide the context for the intended meaning of a particular term.
- Glossary – is a collection of Term definitions and various related forms of the term.
- Index – is a collection of Words or Phrases that are related to internal or external definitions.

A *thesaurus* is a collection of *concepts*. Concepts are identified by Terms, which in turn are manifested by a word or phrase. Note that Terms used in normal language may create ambiguity by describing the same Concepts (e.g. they might have different meanings in different contexts). What differentiates the meaning is the semantic relationship of a Term (i.e. a word or phrase) to a specific Concept. As such Concepts are placeholders for semantic information and relationships, which will be modeled in further detail at a later time.

A *glossary* is a collection of terms that are related through implicit or explicit relationships. The Knowledge Representation Model provides Glossary and Term meta data types and a set of relationships that model Concept synonyms, hierarchically correlated terms, and related “See Also” terms. These relationships allow modeling the most common relationship types found in taxonomies, i.e. between Terms. In reality, such relationships are much more complex and extensive, but this would make a glossary too hard to construct or maintain. The model therefore separates the semantic modeling features of the Thesaurus from the more narrowly scoped Glossary.

A *term* has a definition and may reference one or more words or phrases denoted by i*ndex entries*.

Terms may be preferred (the term best representing its Concept), and as such represent the vocabulary of a user or domain, or non-preferred. Non-preferred Terms are synonyms and point at the preferred Term that should be used instead. The synonym’s relationship between a preferred Term and several non-preferred Terms represents the fact that several Terms describe the same concept, although they might do it with different shades of meaning. Synonyms need to be identified in order to make a vocabulary useful.

Terms may be arranged into a hierarchy of more generic and more specific entries. This Broader/Narrower type of relationship allows substituting “USA” with “Country” or “State” with “Region”. This relationship allows the development of Glossaries without using the, as yet undeveloped, more complex semantic modeling features of the Thesaurus.

Sets of Terms may be related to each other through occurrences. The result of the analysis of the strength of the inter-term co-occurrences in a specific domain is captured by the Related Term relationship. For any particular Term, the relationship captures how strongly the Term is related to a set of other Terms.

An Index organizes Terms into collections. It is a mechanism that provides the entry point into a Glossary or Taxonomy. An index consists of a set of index entries represented by the meta data type IndexEntry. An IndexEntry represents a word, acronym, abbreviation, phrase, or PartOfSpeech that serves as a hook for a reference to a definition or a relationship to a Term. Indexes might be nested to allow structuring of indexes or the grouping of entries into sets meaningful to the user.

19.3 Class Reference

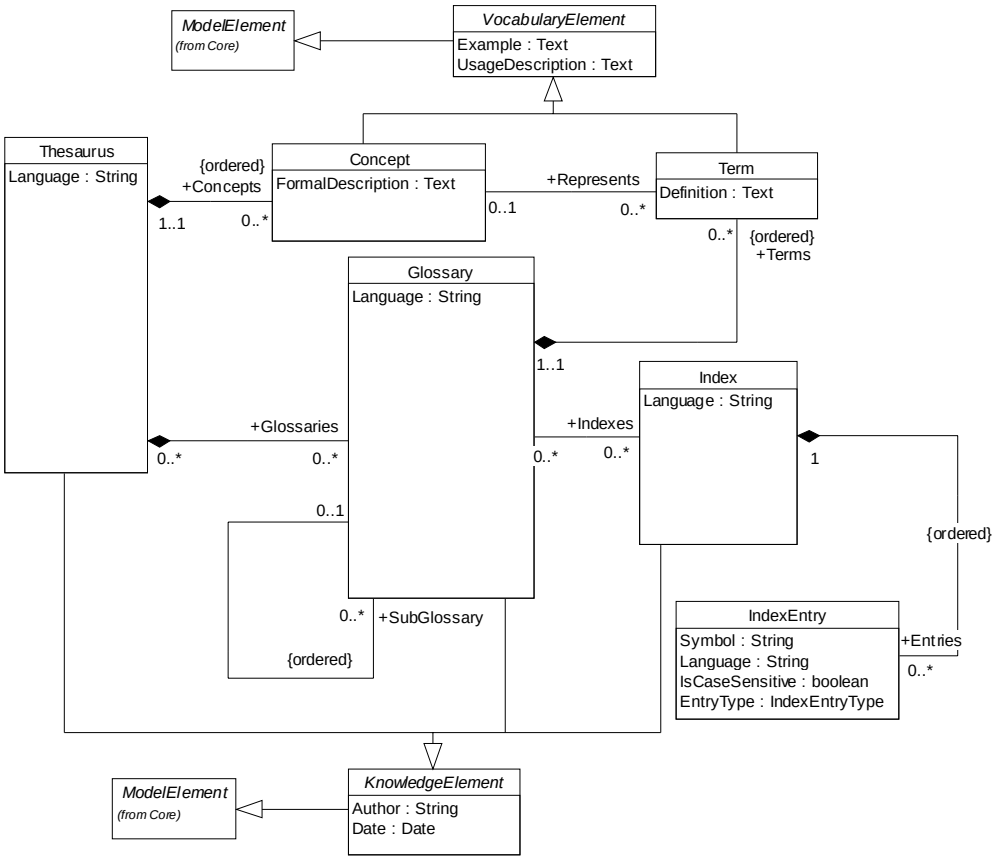


Figure 79: Core Elements

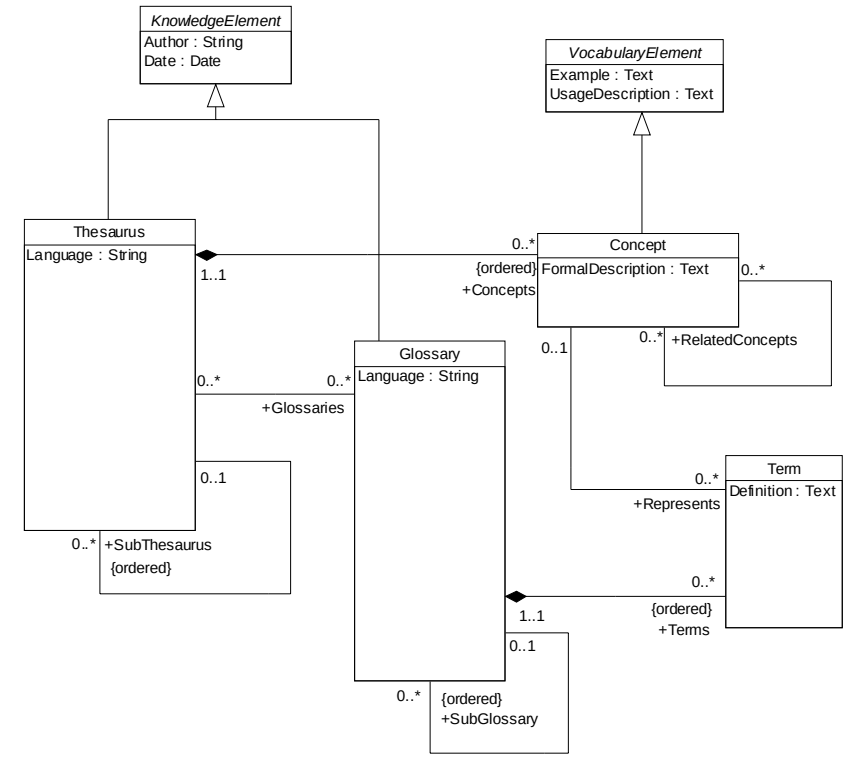


Figure 80: Thesaurus Elements

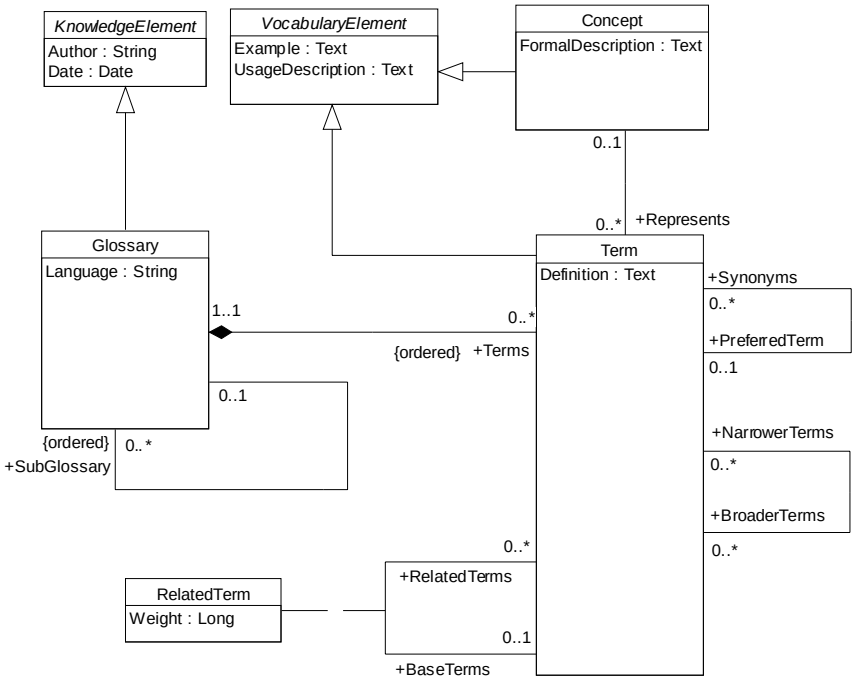


Figure 81: Glossary Elements

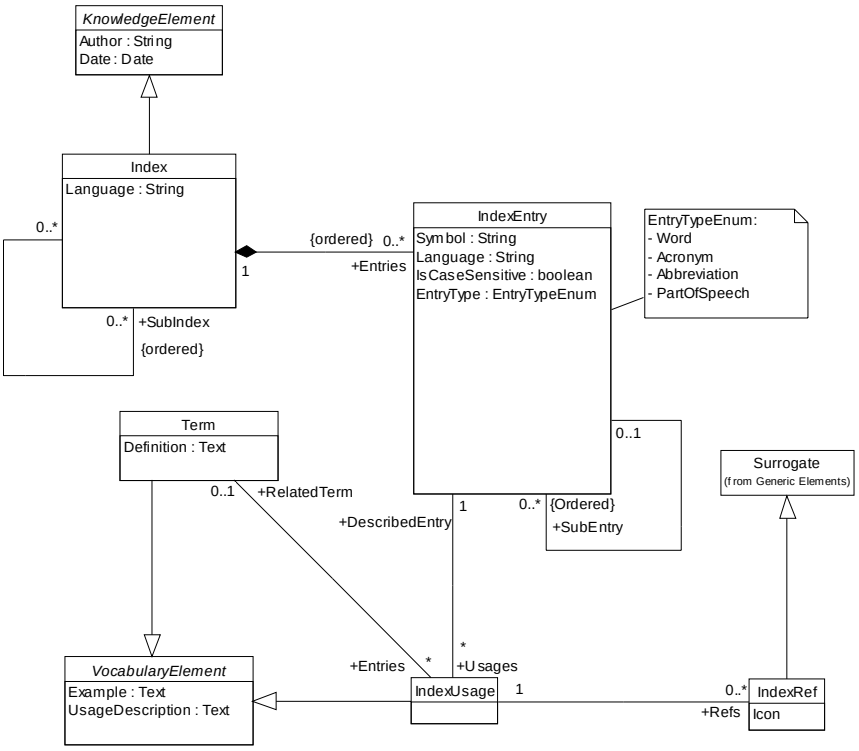


Figure 82: Index Elements

The following describes the different meta data types of the Knowledge Description Model in alphabetical order.

19.3.1 Concept

A Concept represents a semantic type or relationship in a taxon. Semantic types are nodes and relationships are links in a network that represent knowledge at the conceptual level. The purpose of a semantic network is to categorize and relate information in a Thesaurus, i.e. it talks about the topics a user is interested in.

Concepts are represented by Terms. Users choose the Terms that are familiar to them in their environment and that represent a subset of the larger set of Terms that could be used to represent the specific concept. The combination of Context and Term unambiguously defines the topic a user has selected.

Concept is a very broad type, allowing for the semantic categorization of a wide range of terminology in multiple domains. Using the extension mechanisms of the OIM, stereotyping or sub-classing, developers may specialize Concept into more domain specific types.

Concepts are placeholders for the future introduction of relationship semantics.

Example Concept: “Patty” having the FormalDescription of “a Flat food product” could represent the following Terms “Beef Patty”, “Chicken Patty”, “Mint Patty”.

Specializes

- VocabularyElement

Attributes

- *FormalDescription* (String) – Formal description of the Concept.

Associations

- *Represents* – A Concept both represents and is further defined by its associated Terms.

19.3.2 Glossary

A Glossary is a collection of Terms and their various usage forms. A Glossary may contain sub-glossaries, i.e. it may be nested. Nesting of Glossaries may be used for assembly of a large Glossary from several smaller glossaries or for grouping of related Terms. Such a grouping may label a set of Terms for better representation at the user level.

The Glossary is a container for Terms. Terms can also be organized into broader-narrower Term structures or by synonymous and related term relationships that capture non-hierarchical related meanings (for example "See Also" references). This allows the navigation of a Glossary by starting with a Term and following relationships to the set of linked Terms.

Example Glossary: "Food products" with contained Terms such as "Hamburger" and "Hamburger Patty".

Specializes

- KnowledgeElement

Attributes

- *Language* (String) – Language of the Glossary. Note that the Language definition, if provided, applies hierarchically to all contained Index and IndexEntry objects that have no local Language definition. It is recommended to avoid multi-lingual Glossaries and rather use a Glossary or Taxonomy as context for a specific language.

Associations

- *SubGlossaries* – Set of Glossary objects that are contained and as such form the Glossary.
- *Terms* – Set of Term objects that define the entries of the Glossary.

19.3.3 Index

An Index is a collection of IndexEntry items that represents words or multi-words and references to their definitions. Indexes may be nested to group the entries in a meaningful way. An Index represents the set of entry points into a taxon. Once such a point is chosen, the user can navigate through Terms and Concepts (if defined) or access the referenced internal and external information.

For example, the index "Cooking" could have sub-indexes of "Food Products", "Cooking Implements", and "Cooking Techniques".

Specializes

- KnowledgeElement

Attributes

- *Language* (String) – Language of the Index. Note that the Language definition, if provided, applies hierarchically to all contained IndexEntry objects that have no local Language definition.

Associations

- *SubIndex* – Set of Index objects that are contained and as such form the Index. Example usage: an MSDN Index of "Visual Programming Language References" could have a set of SubIndex's such as VC++, VJ++. To support this semantic, a particular Index can be contained by several Index's.
- *Entries* – Set of IndexEntry objects that isolate document references for this index. To reduce problems resulting from updating documents, an IndexEntry may be contained by only one Index.

19.3.4 IndexEntry

An IndexEntry identifies the text from which an entry in the index is made. Each IndexEntry is a word or multi-word representation that may have sub entries further narrowing the entry. An IndexEntry can be of type Word, Abbreviation, Acronym, Phrasing, or PartOfSpeech.

Example IndexEntry: “Patty”, with no associated IndexUsage’s but with SubEntry’s “Melt”, “Hamburger” each having IndexUsage’s representing their locations within documents.

Usage observation. Indexes could be kept “semantically pure” but blended by the presentation interface with IndexEntry’s linked by their Term’s (via Synonyms, RelatedTerms and NarrowerTerms). Conversely, IndexEntry could have SubEntry’s who’s related Term’s were from widely varied Concepts such as “Patty” sub “Melt” and sub “Hearst”.

Specializes

- ModelElement

Attributes

- *Symbol* (Text) – Value of the IndexEntry.
- *Language* (String) – Language of the IndexEntry. Note that if no language is defined, a definition may be inherited from the Index, Glossary, or Taxonomy objects the entry is contained in.
- *IsCaseSensitive* (Boolean) – Indicates if the value of the IndexEntry – symbol attribute – is case sensitive or not (default).
- *EntryType* (IndexEntryType) – Type of the IndexEntry:

Associations

- *SubEntry* – Set of IndexEntry objects that are contained and as such form the Index.

19.3.5 IndexEntryType

Values

- Word – is a string of characters that represent the Term at the linguistic level. = 1
- Acronym – a set of characters or symbols that represent a Term in addition to its representation as word or multi-word phrase.
- Abbreviation – is the representation of a Term by omitting one of more characters from the word or multi-word representation.
- Phrasing – a multi word representation of a Term.
- PartOfSpeech – any part of speech that represents a Term.

19.3.6 IndexUsage

An IndexUsage provides semantic encapsulation of a particular example of an IndexEntry, the particular document references and any associated Term.

Example: The Term “Hamburger Patty” could have two associated IndexUsage’s, one referencing the IndexEntry “Hamburger, sub Patty” and another with an IndexEntry of “Patty, sub Hamburger”, and each IndexUsage could have several IndexRef’s. The set of IndexRef’s would more than likely be the same for each IndexUsage, but that may not be true for all cases.

Specializes

- VocabularyElement – Since an IndexEntry/IndexUsage doesn't need an associated Term. Note this actually provides the user an opportunity to create a more focused example related to this IndexUsage.

Associations

- *Refs* – Set of IndexRef objects representing specific document locations where this IndexEntry is located. The back cardinality on this association is one to simplify index updates when the associated document changes.
- *DescribedEntry – The* IndexEntry being described by this IndexUsage.
- *RelatedTerm* –The Term that is being isolated by this IndexEntry/IndexUsage pair. Semantically speaking the example used by IndexUsage can only apply to one Term. Although there may be other related Terms (Synonyms, NarrowerTerms, RelatedTerms), to be effective an Index should represent these as additional IndexEntry's or rely upon the referenced Term's related Term collections.

19.3.7 IndexRef

An IndexRef identifies a specific document location where the IndexEntry is mentioned.

Specializes

- Surrogate (From Generic Elements)

Attributes

- *Icon* –The Icon from the referenced document's "reader" application.

Associations

- *IndexUsage* – The IndexUsage this IndexRef is elaborating. The cardinality is one to ease the problem of updating Index's when the Document changes.

19.3.8 KnowledgeElement

KnowledgeElement is an abstract type that serves as common super type for the Knowledge Representation Model elements. It defines administration information such as Author and Date.

Specializes

- ModelElement

Attributes

- *Author* (String) - Name of the person or tool that created the KnowledgeElement.
- *Date* (Date) – Date and time when the KnowledgeElement was created or last updated.

19.3.9 RelatedTerm

RelatedTerm is an association class relating Terms outside the current Concept. A common use of the related term relationship is to establish "See Also" links to other Terms based on the strength of the inter-term occurrence. The related instance in the RelatedTerm association class provides the strength or Weight factor.

Example: The Term "Hamburger Patty" within the Concept of "flattened food products", could have a RelatedTerm for "Patty Hearst" within the Concept of "Famous People".

Specializes

- ModelElement

Attributes

- *Weight* (Long) – Weight factor or strength of the relationship between the Terms.

19.3.10 Term

A Term captures words, phrases, etc. and their definition as a formal entry in a Glossary. Terms are very context dependent, e.g. Table in the furniture business has a completely different meaning than table in database technology. The word "Table" therefore may be used by multiple Terms (Furniture or Database Table) to represent different Concepts ("Thing to put things on" or "Relation").

A Term references one or more IndexUsage's and may have a definition in a specific context. The context is provided by a related Concept that describes the underlying semantics and makes the Term unique.

Terms may be grouped into preferred (best representing its Concept) or non-preferred Terms. A non-preferred Term is one that is invalid to use from a perspective of its related Glossary and therefore should lead to a preferred or valid Term. PreferredTerm's are the ones that make up a valid vocabulary and all non-preferred Terms from the set of synonyms.

Terms also can be grouped into broader/narrower term hierarchies capturing the fact that a Term may be of more generic meaning then another Term.

Terms may be related to other terms based on occurrences through the Related Term relationship.

Specializes

- VocabularyElement

Attributes

- *Definition* (String) – Textual representation of the definition of a Term.

Associations

- *Synonyms* (Term) – Set of synonymous Terms for the preferred Term. For a Term to be a true Synonym, it must be contained by the same Concept as the preferred Term.
- *NarrowerTerms* (Term) – Set of Terms with a narrower meaning then the Term. These are Terms whose related Concepts denote a hierarchical relationship such as County within State within Country.
- *RelatedTerms* (Term) – Set of Terms that are related to the Term through similar Concepts. Example usage would be for "See Also" references.
- *Entries* (IndexUsage) – Set of IndexUsages for the given term.

19.3.11 Thesaurus

A Thesaurus is a collection of Concepts that form an ontology. Concepts are the entities and relationships of a semantic network. Thesauruses may be nested to construct larger entities from existing ones or to group Concepts into sets with user meaningful labels.

Concepts are related to the Terms that identify them. Terms are words or phrases used by humans to refer to the Concepts.

Specializes

- KnowledgeElement

Attributes

- *Language* (String) – Language of the Thesaurus. Note that the Language definition, if provided, applies hierarchically to all contained Glossary and Index objects that have no local Language definition.

Associations

- *SubThesaurus* – Set of Thesaurus objects from which the Thesaurus is constructed.
- *Glossary* – Glossary object that contains the Terms that are related to the Concepts of the Thesaurus.
- *Concepts* – Set of Concept contained in the Thesaurus.

19.3.12 VocabularyElement

VocabularyElement is an abstract class that servers as general meta data type that captures common properties for Concepts, Terms, and IndexEntry's.

Specializes

- Element (from UML)

Attributes

- *Example* (String) – Textual representation of a sample of the VocabularyElement, i.e. Concept, Term, or Word.
- *UsageDescription* (String) – Textual description of the usage scenarios for the VocabularyElement, i.e. Concept, Term, or Word.

20 Knowledge Management: Semantic Definitions

20.1 Overview

The Semantic Definitions package accommodates conceptual models of user information. The models are conceptual in that they are independent of any storage structure or programming structure (DBMS schema, object model, etc.). Instead, they conform to canonical and linguistic expressions of categories of data and the interactions among those categories.

With a semantic or linguistic processor, users can interact with data in databases without learning data manipulation languages. Before a linguistic processor can interact with a database, however, an analyst must articulate the mappings between the database schema and the semantic constructs familiar to the users. The Semantic Definition Elements information model accommodates such schema-to-semantic mappings.

The model derives from the UML model and the Database Schema package.

20.2 Semantics

The UML *Package* class is a general-purpose mechanism for establishing containment hierarchies. The Semantic Elements package uses instances of the UML Package class to organize the information in a semantic model. Each instance of *Model* can own several other packages, named "Entities," "Relationships," and "Dictionary."

Each of these packages in turn can owns other UML ModelElements:

- The "Entities" package owns instances of the Entity class.
- The "Relationships" package owns instances of the Relationship class.
- The "Dictionary" package owns instances of the DictionaryEntry class.

The "Relationships" package can also own a package named "PhrasingGroups." The "PhrasingGroups" package owns instances of the PhrasingGroup class.

Here is a summary of the package hierarchy:

SimModel
- *Entities* package
  - Entity1
  - Entity2
  - …
- *Relationships* package
  - Relationship1
  - Relationship2
  - …
  - *PhrasingGroups* package
    - PhrasingGroup1
    - PhrasingGroup1
    - …
- *Dictionary* package
  - DictionaryEntry1
    - DictEntryIrregularity
  - DictionaryEntry2
    - DictEntryIrregularity

The semantic information model honors the way humans (and communities as a whole) speak about the information that is important to them. There are a few fundamental principles that apply:

- There are entities.

  An entity is a named category, such as “Author” or “Book.” Entities have instances, such as “Mark Twain”, or Huckleberry Finn.” Most (but not all) entities correspond to a database table, a database field, or a set of fields.

- There are relationships.

  A relationship is a type of association that exists between entities. For example, one relationship indicates that authors write books. Relationships also have instances, such as “Mark Twain wrote Huckleberry Finn.” There can be several relationships between the same pair of entities. For example another relationship can indicate that authors review books.

  A relationship can include more than two entities. For example “Authors write Books for Publishers.” (The third entity is “Publisher.”)

  Some relationships include only one entity. For example, “Authors are important.”

  A relationship can include the same entity in several different roles. For example, “Authors admire Authors.” An instance of this relationship might be “S.J. Perleman admires Mark Twain.”

- Relationships have phrasings.

  A phrasing is a syntactic template that formalizes one way that people talk about a particular relationship. For example, one phrasing for the relationship between authors and the books they write is “Authors write Books.”

  A relationship can have several phrasings (because people can talk about a relationship in several different ways.) For example, all the following phrasings apply to the same relationship:

  Authors write Books.
  Books are by Authors.
  Books have Authors.
  Authors are the Creators of Books.

Through their syntactic and semantic structure, phrasings allow linguistic processors to interpret relationships.

Each phrasing conforms to a particular syntactic and semantic structure. That is, each phrasing is a particular type of phrasing: Verb Phrasing, Adjective Phrasing, Trait Phrasing, Subset Phrasing, Command Phrasing, Prepositional Phrasing, or Name Phrasing. For example “Authors write Books” is a verb phrasing; it is of the form <SubjectEntity> <Verb> <ObjectEntity>. The subject entity is “Author”; the object entity is “Book.” A linguistic processor can harvest two important facts from this verb phrasing. First, the verb is “write” (rather than “review,” or “dislike,” or “burn.”) Second, the relationship is that “Authors write Books” (rather than “Books write Authors”). Because the syntactic structure of the phrasing *type* (verb phrasing) distinguishes between the subject entity and the object entity, a linguistic processor can understand which entity acts upon the other.

Every relationship must have at least one phrasing. Without at least one phrasing for a relationship, a linguistic processor could not interpret the relationship.

Note that the various phrasings of a relationship need not be phrasings of the same type. For example, "Authors write Books" is a verb phrasing, whereas "Books are by Authors" is a prepositional phrasing.

20.3 Class Reference

This section describes the classes of the Semantic Definitions package in detail.

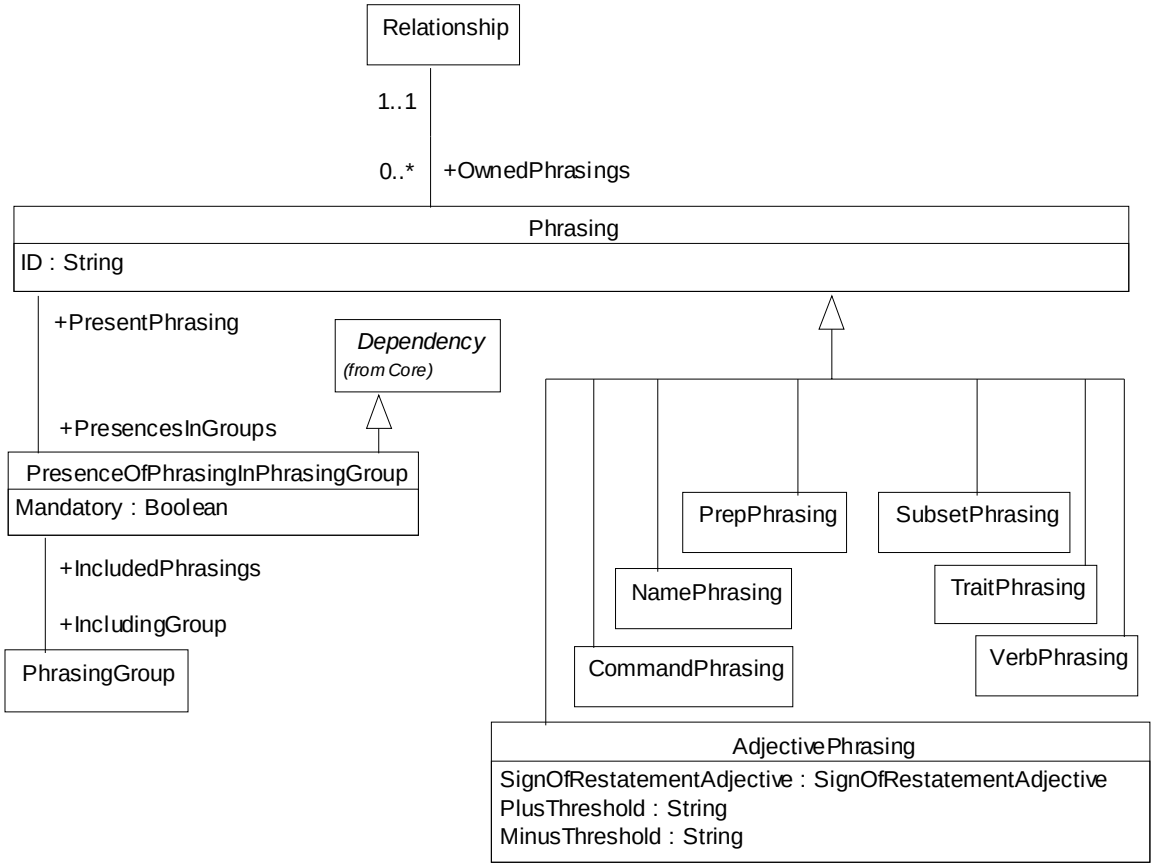


Figure 83: Phrasing Groups and Types of Phrasing

This page is intentionally blank.

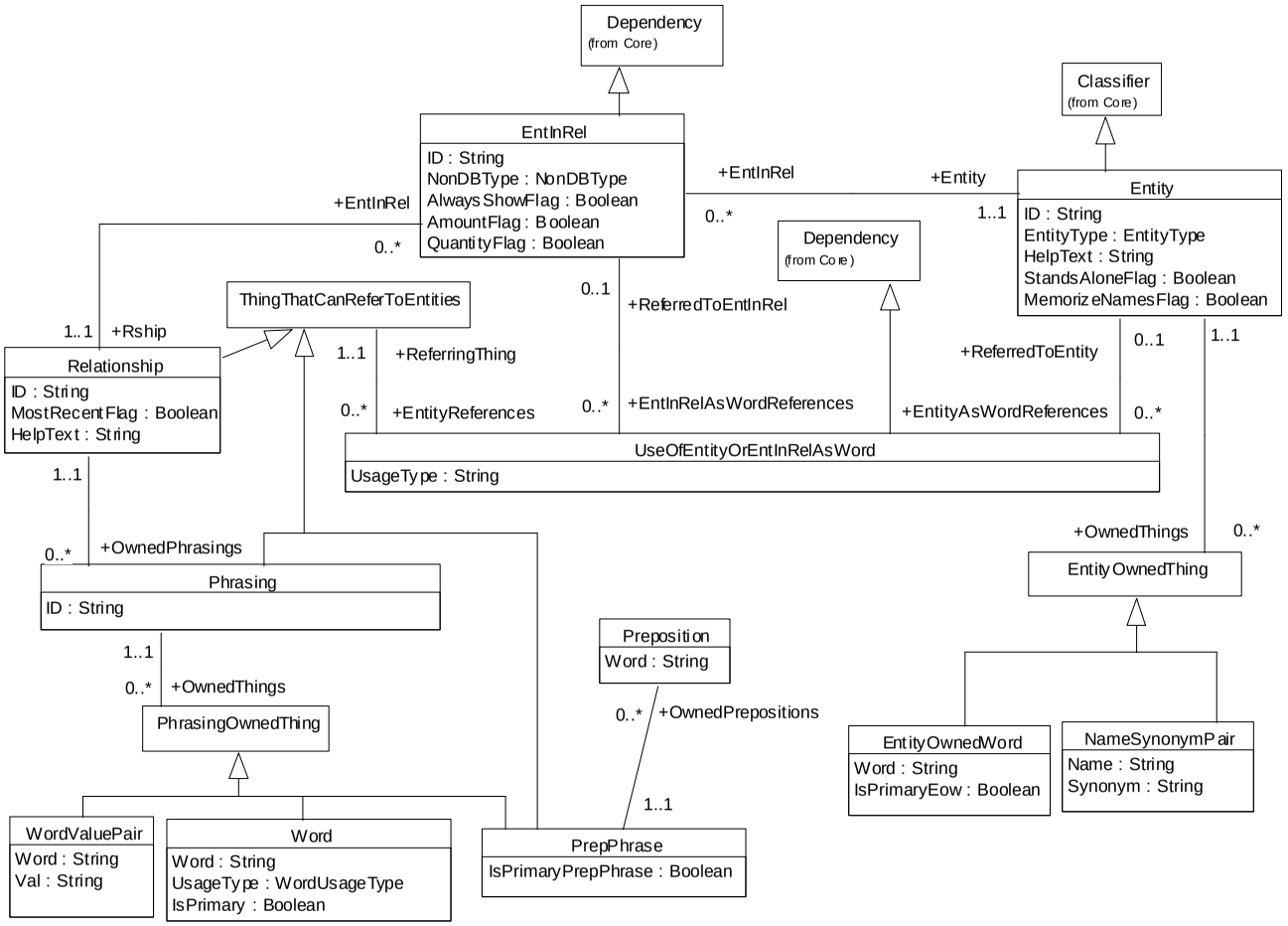


Figure 84: Relationships and Phrasings

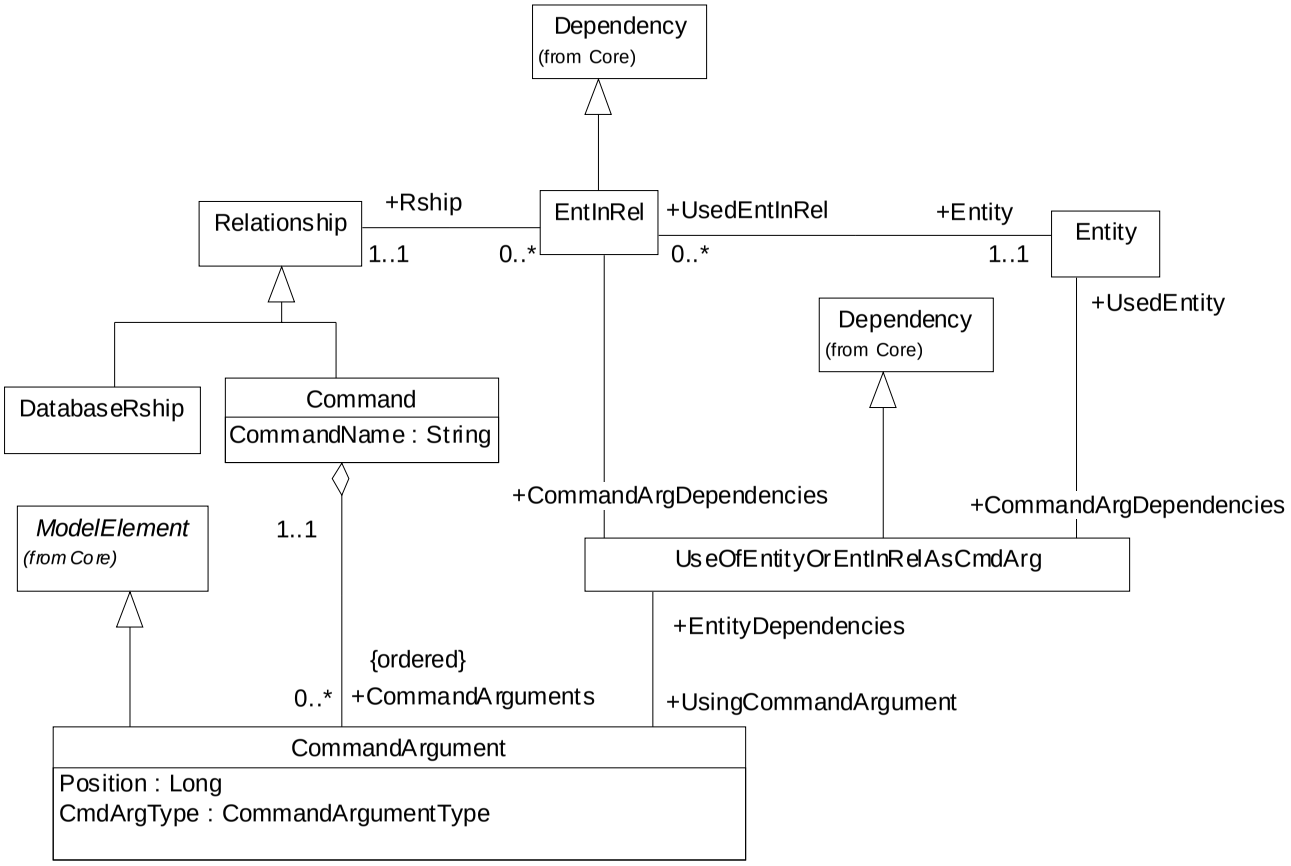


Figure 85: Command Arguments

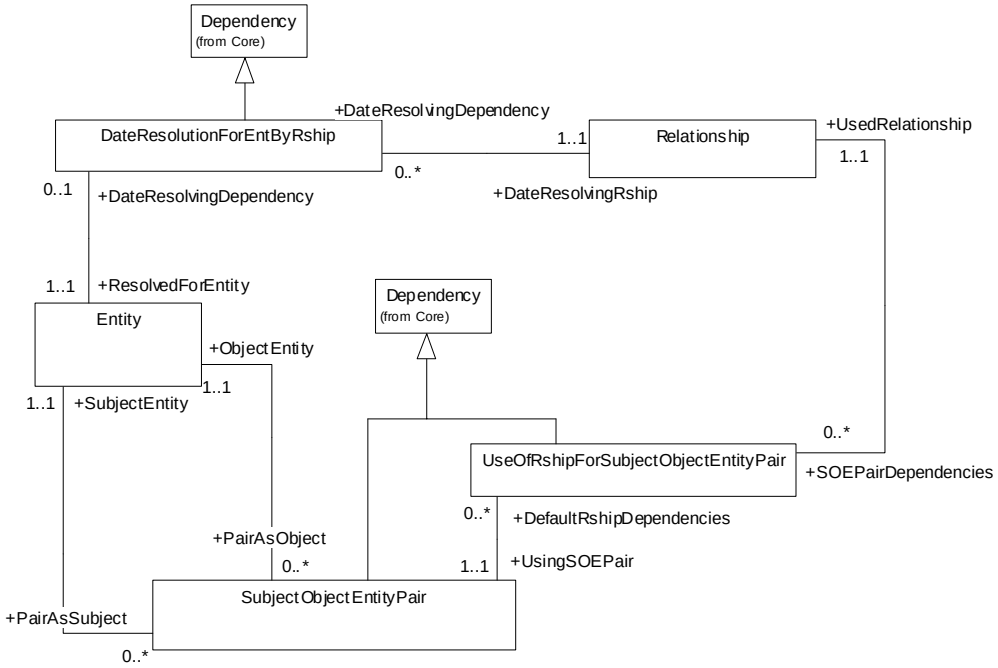


Figure 86: Default Relationships and Date Resolution

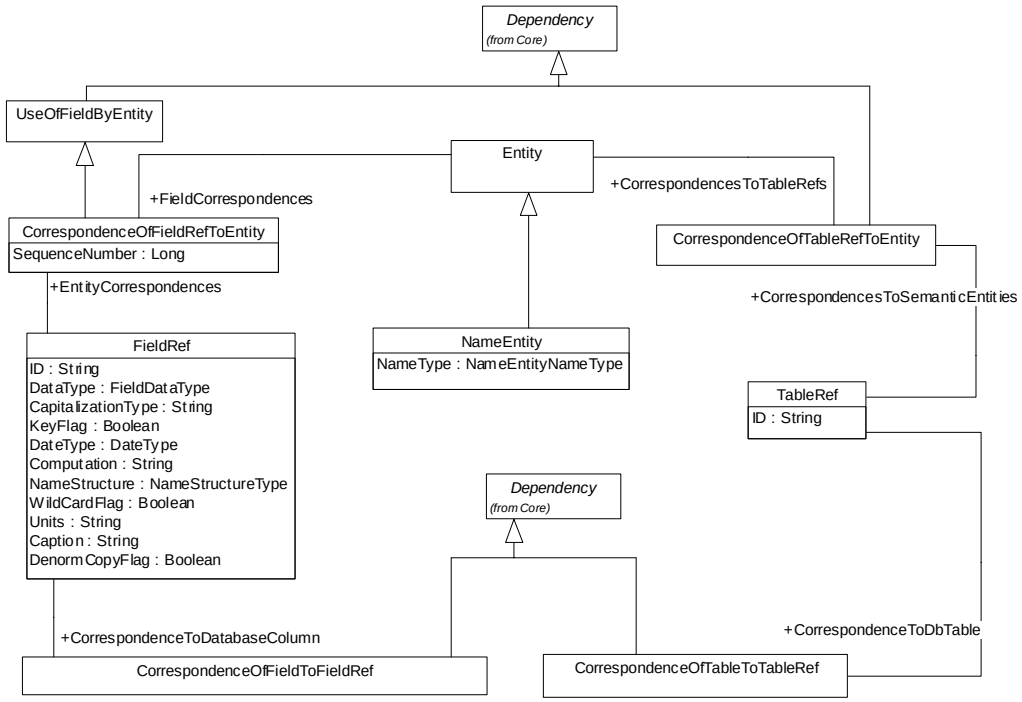


Figure 87: Entity-To-Database Links

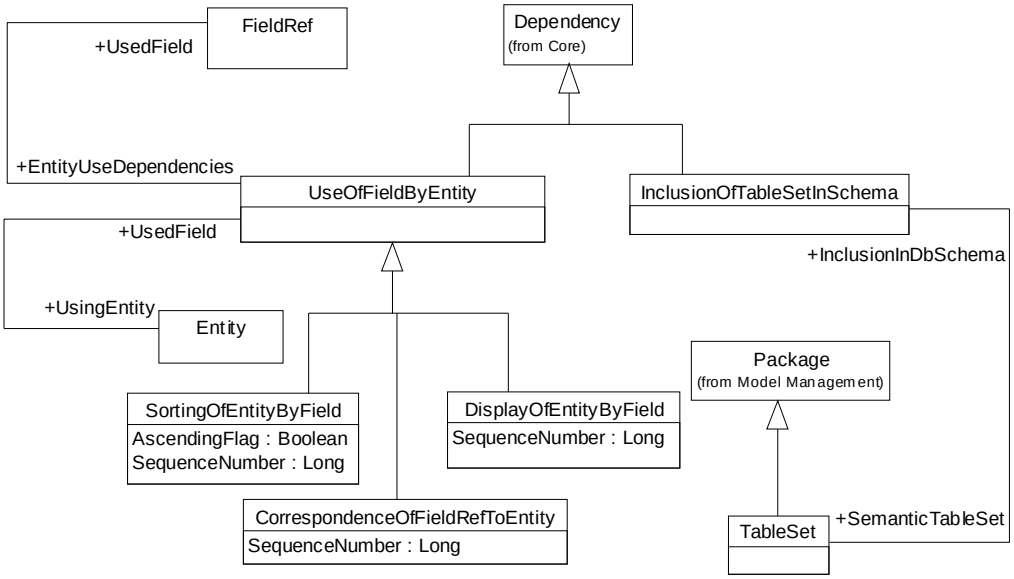


Figure 88: More Database Links

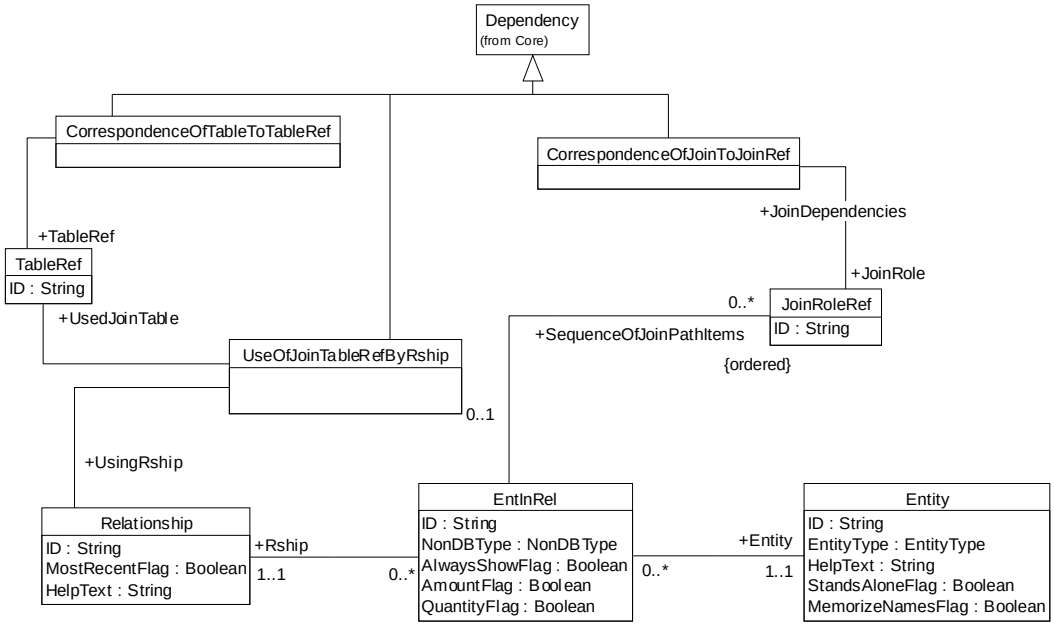


Figure 89: Semantics and Database Joins

20.3.1 AdjectivePhrasing

Each instance of this class describes an AdjectivePhrasing. There are three kinds of adjective phrasing:

- Single-entity adjective phrasings
- Two-entity adjective phrasings

- Measurement phrasings.

Each of these is described in a subsequent section.

Specializes

- Phrasing

Attributes

- *SignOfRestatementAdjective* (SignOfRestatementAdjective) – Controls tool restatement of adjective phrasings containing measurement words (PlusWords or MinusWords). That is, when a linguistic processor paraphrases a user-entered sentence involving a measurement phrasing, does the processor rephrase using the primary PlusWord or the primary MinusWord?
- *PlusThreshold* (String) – The minimum value that a linguistic tool considers a high value for a measurement adjective (e.g., what is the minimum age at which a person is considered old?).
- *MinusThreshold* (String) – The maximum value that a linguistic tool considers a low value for a measurement adjective (e.g., what is the maximum age at which a person is considered young?).

20.3.1.1 Single-entity adjective phrasings

Some adjective phrasings are of the form:

> *X* are *Y*.

For example, *Customers* are *Important*.

In such a phrasing, there is an instance of UseOfEntOrEntInRelAsWord (with UsageType = subject) referring to the EntInRel characterizing the Customer entity's participation in the relationship. "Important" is stored as an instance of Word in the phrasing's *OwnedThings* collection; the string is "Important" and the UsageType is WordUsageType_Adjective.

Notice that in this relationship, there is only one EntInRel; only one entity participates in the relationship.

20.3.1.2 Two-entity adjective phrasings

Some adjective phrasings are of the form:

> The values of *Y* yield adjectives describing the instances of *X*.

For example, *BranchTypes* describe *Branches*.

In such a phrasing, *BranchType* is an entity, and there is an EntInRel describing the participation of it in the relationship. This is different from a single-entity adjective phrasing, because here there are two EntInRels. One EntInRel corresponds to *X*'s participation as the described thing, and the other corresponds to *Z*'s participation as entity containing the describing adjectives.

Note that the values of *Y* might not be adjectives – they could be codes that yield adjectives when applied to some lookup table. For example, the possible values of *Y* could be {1,2,3} corresponding to the adjectives {Good, Fair, Poor} respectively. For information about expressing (code,adjective) pairs, see the section about the *WordValuePair* class.

20.3.1.3 Measurement phrasings

Some adjective phrasings are of the form:

> *Y*s indicate how *Z X* are.
>
> …where *Y* is an entity corresponding to a numeric field, *Z* is an adjective, and *X* is an entity to which the adjective can apply.

For example, "Ages indicate how old customers are."

There can be several adjectives associated with *Y*. For example, two such adjectives are "old" and "elderly."

The measurement adjectives can correspond to opposite ends of the spectrum of the numeric attribute. (e.g., "old" and "young".) Thus, these measurement adjectives can be characterized as PlusWords and MinusWords. PlusWords are the adjectives associated with high values of the measurement. MinusWords are associated with low values. There is no semantic requirement that the plus words be favorable and the minus words be unfavorable. For example, if the phrasing is "Scores indicate how good golfers are," the PlusWords could be "bad" and "poor." Likewise, the MinusWords could be "good," "skillful," and "talented."

To indicate an Adjective, the model includes an instance of *Word* in the phrasing's OwnedThings collection. The value of the WordUsageType is WordUsageType_PlusWord or WordUsageType_MinusWord.

Sample Data

Consider the adjective phrasing "Ages indicate how old customers are." This phrasing belongs to a relationship (called, say, "AgesOfCustomers") with two EntInRels:

- One EntInRel describes the participation of the Customer entity as the characterized thing.
- One EntInRel describes the participation of the Age entity as the characterizing thing.

What's more, the relationship has an adjective phrasing:

- PhrasingName: AgesIndicateHowOldCustomersAre
  SignOfRestatementAdjective: SignOfRestatementAdjective_Plus
  PlusThreshold: 70
  MinusThreshold: 10

The phrasing can have several Words in its OwnedThings collection:

- PlusWord: "Old" (IsPrimary = TRUE)
- PlusWord: "Aged" (IsPrimary = FALSE)
- PlusWord: "Elderly" (IsPrimary = FALSE)
- PlusWord: "Young" (IsPrimary = TRUE)
- PlusWord: "Youthful" (IsPrimary = FALSE)

The phrasing also has two instances of UseOfEntityOrEntInRelAsWord:

- UsageType: Subject
  ReferredToEntInRel: Customer(AsCharacterizedThing)
- UsageType: ObjectOfMeasurement
  ReferredToEntInRel: Age(AsCharacterizingThing)

20.3.2 Command

Each instance of this class describes a command – an imperative statement. Typical commands are "Run the Quarterly Financial Report" or "Buy *n* of the best-selling book."

Specializes

- Relationship

Attributes

- *CommandName* (String) – The name of the command.

Associations

- *CommandArguments* (CommandArgument) – The set of arguments of the command. A command can have zero or more command arguments.

Note also that Command is a specialization of Relationship.

Constraint: Within a command's *OwnedPhrasings* collection, every phrasing must be a CommandPhrasing (rather than a VerbPhrasing, TraitPhrasing, etc.).

Sample Data

A command is kind of relationship. Thus, it can have all the properties and members of relationships, such a phrasings and EntInRels. For example, the instance of Command corresponding to "Send 5000 light bulbs to the Chicago office" would have two instances of EntInRel:

- One instance of EntInRel describes the participation of the *InventoryItem* entity in the relationship as the to-be-sent thing. (In the database, "light bulb" is an instance of the InventoryItem entity.)
- One instance of EntInRel describes the participation of the *Office* entity in the relationship as the send-to destination. (In the database, "Chicago" is an instance of the Office entity.)

The command would also have a phrasing, a command phrasing.

20.3.3 CommandArgument

Each instance of this class describes an argument of a particular command. In the command "Buy *n* copies of the latest book written by *author*" there are two instances of CommandArgument. One instance corresponds to *n* – the number of copies to be bought. The other corresponds to *author* – the author whose most recent book is to be bought.

Specializes

- ModelElement (from UML)

Attributes

- *Position* (Long) – The ordinal position of the command argument.
- *CmdArgType* (CommandArgumentType) – One of {Entity, Amount, Quantity}.

Associations

- *EntityDependencies* (UseOfEntityOrEntInRelAsCmdArg) – A set containing one instance of UseOfEntityOrEntInRelAsCmdArg.
- *Command* (Command) – A set containing one instance of Command – The command to which this CommandArgument contributes.

20.3.4 CommandArgumentType

An enumeration whose values indicate the type of a command argument.

Values

- COMMANDARGUMENTTYPE_ENTITY = 1
- COMMANDARGUMENTTYPE_AMOUNT = 2
- COMMANDARGUMENTTYPE_QUANTITY = 3

20.3.5 CommandPhrasing

Each instance of this class describes a phrasing that consists of:

- An imperative verb.
- Zero, one, or two nouns.
- Zero or more PrepPhrases.

Specializes

- Phrasing

Examples

- Zero nouns: "Reboot."
- One noun: "Send notifications."
- Two nouns: "Send customers products."
- Two nouns and one PrepPhrase: "Send customers products via shippers."

Sample Data

Consider the command phrasing "Send customers products via shippers." This phrasing belongs to a Command with three EntInRels:

- One EntInRel describes the participation of the Product entity as the to-be-sent thing.
- One EntInRel describes the participation of the Customer entity as the receiving thing.
- One EntInRel describes the participation of the Shipper entity as the object of the preposition "via."

Within its OwnedThings collection, the phrasing can have at least one instance of the Word class and one instance of the PrepPhrase class In this example, the OwnedThings collection contains these instances of the Word class:

- Type = Verb, Word = "Send," IsPrimary = TRUE
- Type = Verb, Word = "Transmit," IsPrimary = FALSE
- Type = Verb, Word = "Ship," IsPrimary = FALSE

The PrepPhrase corresponds to the "via shippers" part of the sentence. The PrepPhrase can have within its OwnedPrepositions role at least one Preposition. In this example, the OwnedThings collection contains these instances of the Preposition class:

- Word = "via"
- Word = "through"

The phrasing has two instances of UseOfEntityOrEntInRelAsWord:

- UsageType: FirstObject
  ReferredToEntInRel: Customer(AsReceivingThing)
- UsageType: SecondObject
  ReferredToEntInRel: Product(AsToBeSentThing)

In addition, the PrepPhrase (corresponding to "via shippers") has one UseOfEntityOrEntInRelAsWord:

- UsageType: FirstObject
  ReferredToEntInRel: Shipper(AsObjectOfPrepositionVia)

20.3.6 CorrespondenceOfFieldRefToEntity

Each instance of this class indicates that a particular FieldRef corresponds to a particular entity.

Specializes

- UseOfFieldByEntity

Attributes

- *SequenceNumber* (Long) – Establishes the ordering of FieldRefs within an entity.

20.3.7 CorrespondenceOfFieldToFieldRef

Each instance of this class indicates that a particular FieldRef corresponds to a particular Column. Note that this dependency crosses a package boundary, from the Semantic Elements package to the Schema Elements package.

Specializes

- Dependency (from UML)

20.3.8 CorrespondenceOfJoinToJoinRef

Each instance of this class indicates that a particular JoinRoleRef corresponds to a particular Join. Note that this dependency crosses a package boundary, from the Semantic Elements package to the Schema Elements package.

Specializes

- Dependency (from UML)

20.3.9 CorrespondenceOfTableRefToEntity

Each instance of this class indicates that a particular TableRef corresponds to a particular entity.

Specializes

- Dependency (from UML)

20.3.10 CorrespondenceOfTableToTableRef

Each instance of this class indicates that a particular TableRef corresponds to a particular LogicalTable. Note that this dependency crosses a package boundary, from the Semantic Elements package to the Schema Elements package.

20.3.11 DatabaseRship

Each instance of this class describes a relationship that uses a join table (see *UseOFJoinTableRefByRship*) to link the related entities.

Specializes

- Relationship

Associations

- *JoinTableRefDependencies* (UseOFJoinTableRefByRship) – A set of instances of the UseOFJoinTableRefByRship class.

20.3.12 DateResolutionForEntByRship

Each instance of this class indicates that a particular entity uses a particular relationship situate its instances in time. That is, each instance indicates that a linguistic tool can use a particular relationship to resolve unexpressed (assumed) dates in user expressions.

Specializes

- Dependency (from UML)

Example

Suppose that there are two relationships involving the entity Author:

- Authors write books on dates.
- Authors are born on dates.

If a linguistic processor encounters the question "Who are the 1998 authors?" it must determine whether the question means the authors born in 1998 or the authors who have published books in 1998. If the semantic model includes an instance of DateResolutionForEntByRship where the resolved-for entity is *Author*, the processor knows to use the date from the resolving relationship.

20.3.13 DateType

An enumeration whose values indicate the type of a date.

Values

- DATETYPE_DATE = 1
- DATETYPE_TIME = 2
- DATETYPE_DATETIME = 3
- DATETYPE_INTYEAR = 4
- DATETYPE_INTMONTH = 5
- DATETYPE_MONTHNAME = 6
- DATETYPE_MMM = 7
- DATETYPE_DAY = 8
- DATETYPE_NONE = 0

20.3.14 DictEntryIrregularity

Each instance of this class describes an irregular form of a dictionary entry. Some verbs have unusual past-tense forms – forms that do not conform to standard grammatical rules. For example, the past tense of *sell* is *sold* (rather than *selled*). Similarly, some nouns have unusual plural forms. For example, the plural of *alumnus* is *alumni* (rather than *alumnuses*).

Specializes

- Feature (from UML)

Attributes

- *Type* (IrregularType) – One of {IrregularType_PastTense, IrregularType_Plural}.
- *Form* (String) – The plural form of the noun or the past-tense form of the verb.

20.3.15 DictionaryEntry

Each instance of this class describes a word.

Specializes

- Classifier (from UML)

Attributes

- *ID* (String) – An arbitrary identifier.
- *PartOfSpeech* (PartOfSpeech) – One of the following:
  - PartOfSpeech_Pnoun (proper noun)
  - PartOfSpeech_Noun
  - PartOfSpeech_Verb
  - PartOfSpeech_Preposition
  - PartOfSpeech_Adjective
  - PartOfSpeech_Adverb
  - PartOfSpeech_Pronoun

Associations

- *Irregularities* (DictEntryIrregularity) – A set of instances of DictEntryIrregularity.

20.3.16 DisplayOfEntityByField

Each instance of this class indicates that an entity depends on a field; specifically that when a tool displays instances of the entity, values of this field are included in the display.

Specializes

- UseOfFieldByEntity

Attributes

- *SequenceNumber* (Long) – Sequences the display fields of an entity.

20.3.17 EntInRel

Each instance of this class describes a particular entity's participation in a particular relationship.

Specializes

- Dependency (from UML)

Attributes

- *ID* (String) – An arbitrary identifier of the EntInRel.
- *NonDBType* (enumerated data type) – A coarse characterization of the data type of the entity. The enumeration is NonDBType, with domain of values {NonDBType_Numeric, NonDBType_Text, NonDBType_Date}.
- *AlwaysShowFlag* (Boolean) – TRUE indicates that each time the relationship is used to answer a question, the entity is considered to be part of the query result.
- *AmountFlag* (Boolean) – TRUE only if this EntInRel can have an amount applied to it.
- *QuantityFlag* (Boolean) – TRUE only if this EntInRel can have a quantity applied to it.

Associations

- SequenceOfJoinPathItems – This is the JoinPath from the Relationship's JoinTable to the Database object that corresponds to the entity (of this EntInRel).

Examples

The AmountFlag and QuantityFlag indicate whether an entity can have a quantity or amount associated with it when a user-entered sentence uses this relationship. For example, consider an instance of the *CommandPhrasing* class: "Send customers products." The command associated with this phrasing has two EntInRels:

- An EntInRel (corresponding to the participation of the Customer entity in the relationship). This EntInRel has QuantityFlag = FALSE and AmountFlag = FALSE.
- An EntInRel (corresponding to the participation of the Product entity in the relationship). This EntInRel as QuantityFlag = TRUE and AmountFlag = FALSE.

The following user-entered sentence conforms to this relationship. "Send Hank's Hotel 1000 light bulbs." Light bulb is an instance of the Product entity and Hank's Hotel is an instance of the Customer entity. Because the second EntInRel has the QuantityFlag = TRUE, the number "1000" is appropriate.

20.3.18 Entity

Each instance of this class describes a type of thing. Note that an entity can correspond to any of the following:

- A single database column
- An ordered set of database columns
- A single database table

What's more, an entity might be freestanding – corresponding to no database construct whatsoever.

Specializes

- Classifier (from UML)

Attributes

- *ID* (String) – An arbitrary identifier.
- *EntityType* (EntityType) – Person, Geographic, DateOrTime, or None.
- *HelpText* (Text) – Explanatory text for the Semantic entity.
- *StandsAloneFlag* (Boolean) – TRUE indicates that a linguistic process can display a field entity independently, without the context of the attendant table entity. For many field entities, it makes no sense to display them alone. For example, "Show the ages" should not simply return a column of numbers, since "age" is inherently dependent on what it is the age of. By default, field entities are shown in the context of the table entity they are most directly related to (if any). For example, "Show the ages" would be interpreted as "Show the people and their ages." Certain field entities, however, can be displayed alone. This flag marks such entities, instructing the linguistic processor to not automatically include the major entity (and associated relationship).
- *MemorizeNamesFlag* (Boolean) – TRUE only if the linguistic processor should load this entity's values from the database into memory whenever the model is loaded.

Associations

- OwnedThings

20.3.19 EntityOwnedThing

Each instance of this class describes either an EntityOwnedWord or NameSynonymPair.

Specializes

- ModelElement (from UML)

20.3.20 EntityOwnedWord

Each instance of this class describes a word that can be used to refer to an entity.

Specializes

- EntityOwnedThing

Attributes

- *Word* (String) – The text of the word that can refer to the entity.
- *IsPrimaryEow* (Boolean) – TRUE only if this word is the preferred word for the entity. (Linguistic processors typically use the preferred word when paraphrasing user-entered sentences.) Each entity will have at most one preferred word.

20.3.21 EntityType

An enumeration whose values indicate the type of an entity.

Values

- ENTITYTYPE_PERSON = 1
- ENTITYTYPE_GEOGRAPHIC = 2
- ENTITYTYPE_DATEORTIME = 3
- ENTITYTYPE_NONE = 0

20.3.22 FieldDataType

An enumeration whose values indicate the type of a field.

Values

- FIELDDATATYPE_INTEGER = 1
- FIELDDATATYPE_FLOAT = 2
- FIELDDATATYPE_DATE = 3
- FIELDDATATYPE_STRING = 4
- FIELDDATATYPE_BIT = 5
- FIELDDATATYPE_TEXT = 6
- FIELDDATATYPE_BINARY = 7
- FIELDDATATYPE_OTHER = 0

20.3.23 FieldRef

Each instance of this class is a simplified, abbreviated description of a database column; the description is limited to those things of interest to a semantic or linguistic processor. That is, a FieldRef is a not a complete description of a database column. For a complete description of any column corresponding to the FieldRef, see *CorrespondenceOfFieldToFieldRef*.

Specializes

- ModelElement (from UML)

Attributes

- *ID* (String) – An arbitrary identifier.

- *DataType* (FieldDataType) – Integer, Float, Date, String, Bit, Text, Binary, or Other.
- *CapitalizationType* (String) – Upper, Lower, or FirstLetter
- *KeyFlag* (Boolean) – TRUE only if the field contributes to the table's key.
- *Computation* (String) – For computed fields, Computation contains the SQL computation.
- *NameStructure* (NameStructure) – FirstName, LastName, FirstAndMiddleAndLast, LastAndFirstAndMiddle, or Middle.
- *WildCardFlag* (Boolean) – TRUE only if searches against this field should be automatically wildcarded. For example, "table.field = 'ABC'" becomes "table.field like '*ABC*'."
- *Units* (String) – The unit of measure for a (generally numeric) field. Allows questions referring to units of measure (e.g., "How many feet tall is Abraham?") including known conversions (e.g., "How many inches tall is Abraham?")
- *Caption* (String) – The caption to put on the field in the displayed result set. DenormCopyFlag is set to TRUE only if this FieldRef refers to a denormalized copy of a field.

20.3.24 InclusionOfTableSetInSchema

Each instance indicates that a TableSet exists within a particular Schema object.

Specializes

- Dependency (from UML)

20.3.25 InheritanceOfEntityFromEntity

Each instance indicates that one entity inherits from another.

Specializes

- Dependency (from UML)

20.3.26 IrregularType

An enumeration whose values indicate the type (part of speech) of an instance of the DictEntryIrregularity class.

Values

- IRREGULARTYPE_PASTTENSE = 1
- IRREGULARTYPE_PLURAL = 2

20.3.27 JoinRoleRef

Each instance indicates that a particular EntInRel uses a particular database join as part of its SequenceOfJoinPathItems.

Specializes

- ModelElement (from UML)

Attributes

- *ID* (String) – An arbitrary identifier.

Associations

- *JoinDependencies* (CorrespondenceOfJoinToJoinRef, derived from UML:ModelElement.clientDependency) – a set of instances of the CorrespondenceOfJoinToJoinRef class.

20.3.28 Model

Each instance of this class describes an individual semantic model.

Specializes

- Package (from UML)

20.3.29 NameEntity

Each instance of this class describes a naming entity –an entity that contains names of things.

Specializes

- Entity

Attributes

- *NameType* (NameEntityNameType) – One of the following:
  - EntityType_ProperNoun
  - EntityType_CommonNoun
  - EntityType_ClassifierNoun
  - EntityType_ModelNoun
  - EntityType_UniqueID
  - EntityType_None

20.3.30 NameEntityNameType

An enumeration whose values indicate the type of names contained in an instance of the NameEntity class.

Values

- ENTITYTYPE_PROPERNOUN = 1
- ENTITYTYPE_COMMONNOUN = 2
- ENTITYTYPE_CLASSIFIERNOUN = 3
- ENTITYTYPE_MODELNOUN = 4
- ENTITYTYPE_UNIQUEID = 5
- ENTITYTYPE_NONE = 0

20.3.31 NamePhrasing

Each instance of this class describes a NamePhrasing – a phrasing that describes how things are named, such as “Titles are the Names of Books.”

A NamePhrasing consists of the following:

- A subject (e.g., *Books*) – Stored as an instance of UseOfEntityOrEntInRelAsWord with UsageType = “Subject.”

- An object (e.g., *Titles*) – Stored as an instance of UseOfEntityOrEntInRelAsWord with UsageType = "FirstObject."

Specializes

- Phrasing

20.3.32 NameStructureType

An enumeration whose values indicate the structure of names.

Values

- NAMESTRUCTURETYPE_FIRSTNAME = 1
- NAMESTRUCTURETYPE_LASTNAME = 2
- NAMESTRUCTURETYPE_FIRSTANDMIDDLEANDLAST = 3
- NAMESTRUCTURETYPE_LASTANDFIRSTANDMIDDLE = 4
- NAMESTRUCTURETYPE_MIDDLE = 5

20.3.33 NameSynonymPair

Each instance of this class describes a pair of equivalent entity values. For example, the author surname "Twain" is paired with the author surname "Clemens."

Note that name-synonym pairs indicate synonymy between entity VALUES (Twain = Clemens) rather than between entity NAMES (e.g., Author = Writer).

Specializes

- EntityOwnedThing

Attributes

- *InstanceName* (String, derived from UML:ModelElement.name) – The Name part of a Name-Synonym pair.
- *Synonym* (String) – The Synonym part of a Name-Synonym pair.

20.3.34 NonDBType

An enumeration whose values provide a generic, coarse data type for a field.

Values

- NONDBTYPE_NUMERIC = 1
- NONDBTYPE_TEXT = 2
- NONDBTYPE_DATE = 3

20.3.35 PartOfSpeech

An enumeration whose values indicate the part of speech of a word.

Values

- PARTOFSPEECH_PNOUN = 1 (proper noun)
- PARTOFSPEECH_NOUN = 2
- PARTOFSPEECH_VERB = 3

- PARTOFSPEECH_PREPOSITION = 4
- PARTOFSPEECH_ADJECTIVE = 5
- PARTOFSPEECH_ADVERB = 6
- PARTOFSPEECH_PRONOUN = 7

20.3.36 Phrasing

Each instance of this class describes an AdjectivePhrasing, CommandPhrasing, NamePhrasing, PrepPhrasing, SubsetPhrasing, TraitPhrasing, or VerbPhrasing. Each type of phrasing is described in another section of this chapter.

Specializes

- ThingThatCanReferToEntities

Attributes

- *ID* (String) – An arbitrary identifier.

Associations

- *OwnedThings* (PhrasingOwnedThing)

20.3.37 PhrasingGroup

Each instance of this class describes a group of phrasings that can be used together as a unit.

Sometimes, multiple phrasings are required to work together to describe a single relationship. For example, consider a database that contains people and their hair color. One phrasing which describes this relationship is the trait phrasing "people have hair color." However, this will not be sufficient to answer questions such as "What is the color of John's hair?" For this, we need the phrasings "people have hair" and "hair has color." (In this case, "hair" is an entity that is not represented by a database object). These two phrasings collectively describe the relationship between people and hair color. These two phrasings need to be grouped so that a linguistic processor knows to treat them as a logical unit.

Specializes

- ModelElement (from UML)

20.3.38 PhrasingOwnedThing

Each instance of this class describes a WordValuePair, a Word, or a PrepPhrase. Each of these classes is described elsewhere within this chapter.

Specializes

- ModelElement (from UML)

20.3.39 Preposition

Each instance of this class describes a preposition that is owned by a PrepPhrase. Note that if two different PrepPhrases use the same preposition, there will be two different instances of PrepPhrase.

Specializes

- ModelElement (from UML)

Attributes

- *Word* (String) – the text of the word that is the preposition.

20.3.40 PrepPhrase

Each instance of this class describes a prepositional phrase that is attached to a phrasing.

(Note: Do not confuse PrepPhrase with PrepPhrasing, described in the next section. Also note that some prepositional phrases are better stored in another way – not as instances of the PrepPhrase class. For more information, see *ThingThatCanReferToEntities: Relationship.*)

Grammatically, a prepositional phrase is a preposition (*in, out, above, below, of, from, to, through, within,* etc.) followed by a noun.

Within the Semantic Elements package, each PrepPhrase has these parts:

- The OwnedPrepositions role. A single PrepPhrase can have multiple prepositions because in some situations, certain prepositions are interchangeable. (For example, "…in years" and "…during years" are equivalent.) Note that two prepositions are not interchangeable in all situations. (For example, "…in cities" and "…during cities" are not equivalent.)
- A UseOfEntityOrEntInRelAsWord. (PrepPhrase is a specialization of ThingThatCanReferToEntities.) The UseOfEntityOrEntInRelAsWord indicates which EntInRel serves as the object of the preposition.

Specializes

- Phrasing

Attributes

- *IsPrimaryPrepPhrase* (Boolean) – TRUE only if the prepositional phrase is the primary prepositional phrase of its owning phrasing.

20.3.41 PrepPhrasing

(Note: Do not confuse PrepPhrasing with PrepPhrase, described in the previous section.)

Each instance of this class describes a Prepositional Phrasing – a phrasing that relates a subject to a prepositional phrase, such as "People are on medications."

A PrepPhrasing consists of the following:

- A subject (e.g., *People*) – Stored as an instance of UseOfEntityOrEntInRelAsWord with UsageType = "Subject."
- One or more prepositions (e.g., *on*) – Each stored as an instance of *Word* with UsageType – WordUsageType = WordUsageType_Preposition. Exactly one of the prepositions will have the IsPrimary flag set to TRUE.
- An object of the preposition (e.g., *medications*) – Stored as an instance of UseOfentityOrEntInRelAsWord with UsageType = "FirstObject."
- Zero or more prepositional phrases (e.g., "People are on medications *for conditions*") – Each stored as an instance of PrepPhrase in the PrepPhrasing's OwnedThings collection.

Specializes

- ThingThatCanReferToEntities
- PhrasingOwnedThing

Associations

- *OwnedPrepositions* (Preposition)

20.3.42 PresenceOfPhrasingInPhrasingGroup

Each instance of this class indicates that a particular phrasing group includes a particular phrasing.

Specializes

- Dependency (from UML)

Attributes

- *Mandatory* (Boolean) – TRUE only if the phrasing is a required part of the phrasing group. It might be senseless to ask about some phrasings in a group without also including other required phrasings. For example, you may wish to ask the question "List the colors of the parts" to always be interpreted as "List the colors of the parts that suppliers supply," so the user clearly understands that colors of parts are known only in the context of suppliers supplying them.

  If the phrasing group includes the verb phrasing "suppliers supply parts" and the adjective phrasing "parts have colors," mark the verb phrasing as mandatory and the adjective phrasing as not mandatory.

20.3.43 Relationship

Each instance of this class describes a semantic relationship.

Specializes

- ThingThatCanReferToEntities

Attributes

- *ID* (String) – An arbitrary identifier.
- *MostRecentFlag* (Boolean) – TRUE only if the linguistic processor should show only the most recent data when this relationship is used. For example, if the user asks the question "Show the blood pressure of patient 123," the linguistic processor would interpret the question as "Show the most recent blood pressure of patient 123."
- *HelpText* (String) – Explanatory text about the Relationship.

Associations

- *OwnedPhrasings* (Phrasing)

Examples

One phrasing describing a relationship is "Authors write Books." This phrasing is a verb phrasing. That means it is of the form <SUBJECT> <VERB> <OBJECT>. (Some verb phrasings are of the form <SUBJECT> <VERB> <OBJECT><OBJECT>, like "Professors give students grades.")

The verb phrasing has in its OwnedThings collection an instance of Word, where:

- Word.Word = "write"
- Word.UsageType = VERB

The verb phrasing also has two instances of UseOfEntityOrEntInRelAsWord (which is a Dependency). The first instance connects the verb phrasing to the particular instance of EntInRel describing the AUTHORS's participation in the relationship. It has this property value:

- UseOfEntityOrEntInRelAsWord.UsageType = SUBJECT

The second instance of UseOfEntityOrEntInRelAsWord connects the verb phrasing to the particular instance of EntInRel describing the BOOK's participation in the relationship. It has this property value:

- UseOfEntityOrEntInRelAsWord.UsageType = OBJECT

20.3.44 SignOfRestatementAdjective

An enumeration whose values indicate whether an adjective of measurement uses the primary PlusWord or the primary MinusWord in ordinary discourse.

Values

- SIGNOFRESTATEMENTADJECTIVE_PLUS = 0
- SIGNOFRESTATEMENTADJECTIVE_MINUS = 1

20.3.45 SortingOfEntityByField

Each instance of this class indicates the contribution of a field to an entity's sort order (whenever instances of that entity are displayed).

Specializes

- UseOfFieldByEntity

Attributes

- *AscendingFlag* (Boolean) – TRUE only if the field's values are arranged in ascending order in the entity's display.
- *SequenceNumber* (Long) – Indicates the significance of this field among all of this entity's sorting fields: 1 = most significant; *n* = least significant..

20.3.46 SubjectObjectEntityPair

Each instance of this class describes a pair of entities for which some default relationship is declared.

In a semantic model two entities (e.g., Author and Book) could have several relationships between them (e.g., Authors-Write-Books and Authors-Review-Books and Authors-Own-Books). The designer of the semantic model can declare one of these relationships to be the default relationship for connecting instances of the two entities to each other. In this example, the designer would probably declare Author-Write-Books to be the default relationship. This means that if the query is "Show me the books and their authors," the linguistic processor will assume that user means "... the books and the authors who wrote them," rather than "...the books and the authors who reviewed them."

Specializes

- Dependency (from UML)

Associations

- *DefaultRshipDependencies* (UseOfRshipForSubjectObjectEntityPair) – An instance of the class UseOfRshipForSubjectObjectEntityPair.

Note

Note that (subject entity, object entity) is an *ordered* pair. This makes sense, and an example will show why. Consider the two phrases:

- Authors and their books – The designer of the semantic model might want this phrase to be interpreted as "the authors and the books they own."
- Books and their authors – The designer of the semantic model might want this phrase to be interpreted as "the books and the authors who wrote them."

In other words, the default relationship for (Author, Book) is not the same as the default relationship for (Book, Author).

20.3.47 SubsetPhrasing

Each instance of this class describes a subset phrasing.

Specializes

- Phrasing

Note

There are two kinds of subset phrasing:

- Some *X* are *Y*.

  For example, Some *Authors* are *Poets*.

  In such a phrasing, there is an instance of UseOfEntOrEntInRelAsWord (with UsageType = subject) referring to the EntInRel characterizing the Author's participation in the relationship. Poet is stored as an instance of Word in the phrasing's *OwnedThings* collection; the string is Poet (singular; not "Poets") and the UsageType is "SubsetWord."

  Notice that in this relationship, there is only one EntInRel – only one entity participates in the relationship.

- Each *X* is characterized by the value of *Z*.

  For example, each *author* is characterized by the value of *genre*.

  In such a phrasing, *genre* is an entity, and there is an EntInRel describing the participation of the *genre* entity in the relationship. This is different from the other kind of subset phrasing, because in this kind, there are two EntInRels. One EntInRel corresponds to *X*'s participation as the superset, and the other corresponds to *Z*'s participation as the classifying axis.

20.3.48 TableRef

Each instance of this class is a simplified, abbreviated description of a database table; the description is limited to those things of interest to a semantic or linguistic processor. That is, a TableRef is a not a complete description of a database table. For a complete description of any table corresponding to the TableRef, see *CorrespondenceOfTableToTableRef*.

Specializes

- ModelElement (from UML)

Attributes

- *ID* (String) – An arbitrary identifier.

20.3.49 TableSet

Each instance of this class is a set of database tables all existing within the same database catalog. This is simply a grouping mechanism for convenience to the designers of Semantic models. A TableSet does not have to include all the tables defined in a catalog. Within a semantic model, there can be several TableSets referring to tables from the same catalog. (Within any semantic model, however, each table can appear in at most one TableSet.)

Specializes

- Package (from UML)

20.3.50 ThingThatCanReferToEntities

Each instance of this class describes a Phrasing, PrepPhrase, or Relationship.

Specializes

- ModelElement (from UML)

20.3.50.1 ThingThatCanReferToEntities: Phrasing

A Phrasing is a thing that can refer to entities because a phrasing can use an entity – or more typically, an EntInRel – as a noun. For example, one possible form for a verb phrasing is <SUBJECT> <VERB> <OBJECT> (e.g., "Customers buy Products.").

The <VERB> (e.g., "buy") is stored as an instance of *Word* in the VerbPhrasing's OwnedThings collection.

The <SUBJECT> (Customers) is stored as an instance of UseOfEntOrEntInRelAsWord (with UsageType = "Subject"), in the VerbPhrasing's EntityReferences collection. The VerbPhrasing has an EntityReferences collection because it specializes Phrasing, which specializes ThingThatCanReferToEntities.

Similarly, the <OBJECT> (e.g., Products) is stored as an instance of UseOfEntOrEntInRelAsWord (with UsageType = "FirstObject").

20.3.50.2 ThingThatCanReferToEntities: PrepPhrase

A PrepPhrase is a thing that can refer to entities because a prepositional phrase can use an entity – or more typically, an EntInRel – as a noun. A prepositional phrase is of the form <PREPOSITION> <NOUN>. For example a typical prepositional phrase is "through shippers," as in "Branches send Customers Products *through Shippers*."

The <PREPOSITION> (in this example, "through") is stored as an instance of *Preposition* in the PrepPhrase's OwnedPrepositions collection.

Similarly, the <NOUN> (in this example, "Shippers") is stored as an instance of UseOfEntOrEntInRelAsWord (with UsageType = "FirstObject")

20.3.50.3 ThingThatCanReferToEntities: Relationship

In some cases, what is grammatically a prepositional phrase is stored differently (that is, it is not stored as an instance of PrepPhrase) to take advantage of the semantic capabilities of linguistic processors. For example, consider "Authors write books during years." The phrase "…*during years*" is not stored as a prepositional phrase. Rather, there is an instance of *UseOfEntOrEntInRelAsWord* whose ReferringThing is the relationship itself and whose ReferredToEntity is some entity whose instance describes a year. This particular instance of *UseOfEntOrEntInRelAsWord* has UsageType = "RELATIONSHIP_WHEN." By storing the relationship this way, a linguistic process could answer questions like "what author wrote books before 1974?"

In this context, there are five noteworthy values of UseOfEntityOrEntInRelAsWord.Usage_Type:

- RELATIONSHIP_WHEN
  This UsageType applies when an entity's participation in a relationship indicates *when* the relationship occurred (e.g., "Authors write books *during years*.").
- RELATIONSHIP_WHERE
  This UsageType applies when an entity's participation in a relationship indicates *where* the relationship occurred (e.g., "Authors write books *in cities*.").
- RELATIONSHIP_START
  This UsageType applies when an entity's participation in a relationship indicates when the relationship *started* (e.g., "Employees work on projects *from dates* until dates.").

- RELATIONSHIP_END
  This UsageType applies when an entity's participation in a relationship indicates *when* the relationship *ended* (e.g., "Employees work on projects from dates *until dates*.").
- RELATIONSHIP_DURATION
  This UsageType applies when an entity's participation in a relationship indicates *for how long* the relationship occurred (e.g., "Employees commute to work for *time_periods*.").

20.3.51 TraitPhrasing

Each instance of this class describes a TraitPhrasing – a phrasing between entities in which the instances of one entity describe a characteristic of instances of the other entity.

For example, a trait phrasing is "Employees have PhoneNumbers." The values of the entity PhoneNumber describe a characteristic of the instances of the entity Employee.

Each instance of TraitPhrasing must indicate the characterized entity (e.g., "Employee") and the characterizing entity (e.g., "PhoneNumber"). To indicate the characterized entity, the model includes an instance of UseOfEntityOrEntInRelAsWord whose usage type is "Subject" and whose referred-to thing is either:

- The entity describing "Employee."
- The EntInRel describing the participation of the "Employee" entity in the relationship to which the TraitPhrasing belongs. (Remember, each phrasing belongs to a particular relationship.)

To indicate the characterizing entity, the model includes an instance of UseOfEntityOrEntInRelAsWord whose usage type is "FirstObject" and whose referred-to thing is either:

- The entity describing "Employee."
- The EntInRel describing the participation of the "Employee" entity in the relationship to which the TraitPhrasing belongs.

Specializes

- Phrasing

20.3.52 UseOfEntityOrEntInRelAsCmdArg

Each instance of this class indicates that a particular CommandArgument uses a particular entity or EntInRel.

20.3.53 UseOfEntityOrEntInRelAsWord

Each instance indicates that a semantic entity is used in some phrasing, relationship, or prepositional phrase.

Specializes

- Dependency (from UML)

Attributes

- *UsageType* (String) – Indicates how the entity participates (e.g., as Subject or Object,) in the semantic construct that uses it.
- Entity (Entity, derived from Dependency.Supplier) – The Entity referred to.
- EntInRel (EntInRel, derived from Dependency.Supplier) – The EntInRel referred to.

20.3.54 UseOfFieldByEntity

Each instance of this class indicates that a semantic entity uses a field.

Specializes

- Dependency (from UML)

20.3.55 UseOfJoinTableRefByRship

Each instance indicates that a Semantic Relationship uses a database table as the starting table for the various join paths leading to each of the database objects corresponding to the various EntInRels.

Specializes

- Dependency (from UML)

20.3.56 UseOfRshipForSubjectObjectEntityPair

Each instance indicates that a Semantic Relationship is the default relationship for a particular (subject entity, object entity) pair. For more information, see "SubjectObjectEntityPair."

Specializes

- Dependency (from UML)

20.3.57 VerbPhrasing

Each instance of this class describes a VerbPhrasing – a phrasing indicating a verb-based relationship between entities. For example, in the phrasing "Salespeople sell customers books," the verb is "sell."

Specializes

- Phrasing

Notes

Each instance of VerbPhrasing must indicate the subject entity (e.g., "Salesperson"), the verb ("sell"), the direct object ("book"), and potentially the indirect object ("customer").

To indicate the subject entity, the model includes an instance of UseOfEntityOrEntInRelAsWord whose usage type is "Subject" and whose referred-to thing is either:

- The Entity describing "Salesperson."
- The EntInRel describing the participation of the "Salesperson" entity in the relationship to which the VerbPhrasing belongs. (Remember, each phrasing belongs to a particular relationship.)

To indicate the Verb, the model includes an instance of *Word* in the phrasing's OwnedThings collection. The value of the WordUsageType is WordUsageType_Verb.

To indicate the object or objects, the model includes one or two instances of UseOfEntityOrEntInRelAsWord. There are two cases:

- Case 1: If there is a direct object only, the model includes an instance of UseOfEntityOrEntInRelAsWord whose UsageType is "FirstObject" and whose referred-to thing is the EntInRel describing the direct object's participation in the relationship.
- Case 2: If there is a direct object and an indirect object, the model includes:
  - For the indirect object: An instance of UseOfEntityOrEntInRelAsWord whose UsageType is "FirstObject" and whose referred-to thing is the EntInRel describing the indirect object's participation in the relationship.

- For the direct object: An instance of UseOfEntityOrEntInRelAsWord whose UsageType is "SecondObject" and whose referred-to thing is the EntInRel describing the direct object's participation in the relationship.

20.3.58 Word

Each instance of this class describes a word that is used in a phrasing.

Specializes

- PhrasingOwnedThing

Attributes

- *Word* (String) – The text of the word.
- *UsageType* (WordUsageType)
- *IsPrimary* (Boolean) – TRUE if the word is the preferred word.

IsPrimary is used because some phrasings can have several interchangeable words – all of the same UsageType. For example, there can be a single verb phrasing that allows several different verbs, such as:

- "Authors write books."
- "Authors pen books."
- "Authors author books."

In such a situation, the verb phrasing has three instances of *Word* in its OwnedThings collection. The following table describes:

Word	**UsageType**	**IsPrimary**
Write	WordUsageType_Verb	TRUE
Pen	WordUsageType_Verb	FALSE
Author	WordUsageType_Verb	FALSE

A linguistic processor might use the preferred word when paraphrasing user-entered sentences for clarification.

20.3.59 WordUsageType

An enumeration whose values indicate what syntactic role an entity or EntInRel plays in a particular phrasing.

Values

- WORDUSAGETYPE_ADJECTIVE = 1
- WORDUSAGETYPE_MINUSWORD = 2
- WORDUSAGETYPE_PLUSWORD = 3
- WORDUSAGETYPE_SUBSETWORD = 4
- WORDUSAGETYPE_VERB = 5
- WORDUSAGETYPE_PREPOSITION = 6

20.3.60 WordValuePair

Each instance of this class describes an (Adjective, StoredEncoding) pair.

In two-entity adjective phrasings, the object entity sometimes contains codes instead of adjectives. Such a phrasing can have any number of (Word, Value) pairs to equate the adjectives to their stored encoding. For example, an adjective phrasing might have three word value pairs: (Agree, 1); (Disagree, -1); (NoOpinion, 0).

Specializes

- PhrasingOwnedThing

Attributes

- *Word* (String) – The Word of a Word-Value pair.
- *Val* (String) – The Value of a Word-Value pair.

Glossary

aggregate

A total created from smaller units. For example, the population of a county is an aggregate of the populations of the cities, rural areas, etc. that comprise the county.

application

A program or set of programs designed to assist in the performance of a task, for example, an order entry system.

architecture

An organized framework consisting of principles, rules, conventions, and standards that serve to guide development and construction activities such that all components of the intended structure will work together to satisfy the ultimate objective of the structure.

BPR

Business process reengineering. A radical improvement approach that critically examines, rethinks, and redesigns mission product and service processes within a political environment. It achieves dramatic mission performance gains from multiple customer and stakeholder perspectives. BPR is a key part of a process management approach that continually evaluates, adjusts or removes processes to achieve optimal performance.

business

An organization or group of people that have formed to perform a specific mission and to achieve specific goals and objectives.

business object

An object that is modeled after a business concept, such as a person, place, event, or process. Business objects represent real world things such as employees, products, invoices, or payments.

cleansing

A process that checks data quality for adherence to standards, internal consistency, referential integrity, domain validity, and replaces incorrect data with correct data. For example, an invalid zip code can be replaced by a zip code derived from the state/city/address information. Cleansing methods can include combinations of: look-up against valid data (e.g. a list of mailing addresses), look-up against domain values (e.g. a list of valid State, Province, or Territory codes), domain range checks (e.g. Employees less than 15 or greater than 90 years old), consistency checks among table data, pattern analysis of exceptions, correlation, and frequency distributions.

condition

Sufficient prerequisite for an object to enter, transfer, or exit a state.

constant

A language element that specifies an unchanging value. Constants are classified as string constants or numeric constants. The opposite of a constant is a variable.

constraint

A statement that must hold or not hold in a system. Constraints most often are logical properties of objects or transitions between objects that must apply for a system to function correctly.

data warehouse

A special database assembled from data extracted from operational databases and other data sources. Often used for analysis and decision support.

decomposition

The process of breaking down an activity into more detailed component activities.

expression

An operand or a collection of operators and operands that yields a single return value.

extraction

The process that selects data from various data sources in preparation of a copy to a target database. Extraction includes the selection of the data to be copied and the access to the physical storage that manages the data.

fact

A relationship between objects that hold in the actual world.

field

An atomic piece of information in a file or database. A field has a format such as character, number, or date and its presence can be optional or mandatory.

file

A complete, named collection of information. The basic unit of storage that enables a computer to distinguish one set of information from another.

IDEF

Integrated Definition Language

IDEF modeling techniques

A combination of graphic and narrative symbols and rules designed to capture the processes and structure of information in an organization. IDEF0 is an activity, or behavior, modeling technique; IDEF1X is a rule, or data, modeling technique.

information

Data in context related to a specific purpose.

information pump

A tool that extracts data from source systems, such as mainframe or client/server systems, performs cleansing and transformations, and loads the resulting information into another database.

knowledge

The sum of what has been perceived, discovered, or learned through experience or study.

meta data

Descriptive information about data.

model

A representation of a complex, real-world phenomenon that can answer questions about the real-world phenomenon within some acceptable and predictable tolerance.

object

A distinct entity.

object-oriented

Of, pertaining to, or being a system or language that supports the use of objects.

relational database

A database that can be perceived as a set of tables and manipulated in accordance with the relational model of data.

repository

A model-driven facility for the storage of meta data.

scrubbing

See cleansing.

skill

The ability to perform a task or function to an agreed-upon criterion.

UML

The Unified Modeling Language.

variable

A data element that is a container for a value that can be changed.

Class Index

Table of Figures