Date: August 2011

OMG Unified Modeling Language™ (OMG UML), Infrastructure

Version 2.4.1

OMG Document Number: formal/2011-08-05
Standard document URL: http://www.omg.org/spec/UML/2.4.1/Infrastructure
Associated Normative Machine-Readable Files:
http://www.omg.org/spec/UML/20110701/Infrastucture.xmi
http://www.omg.org/spec/UML/20110701/L0.xmi
http://www.omg.org/spec/UML/20110701/LM.xmi
http://www.omg.org/spec/UML/20110701/PrimitiveTypes.xmi

Version 2.4.1 supersedes formal/2010-05-04.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.
IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 140 Kendrick Street, Needham, MA 02494, U.S.A.

TRADEMARKS

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® and XMI® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™ , Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™ , IMM™, MOF™ , OMG Interface Definition Language (IDL)™, and OMG Systems Modeling Language (OMG SysML)™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

OMG’s Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

Table of Contents

1 Scope

This specification defines the Unified Modeling Language (UML), revision 2. The objective of UML is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes.

The initial versions of UML (UML 1) originated with three leading object-oriented methods (Booch, OMT, and OOSE), and incorporated a number of best practices from modeling language design, object-oriented programming, and architectural description languages. Relative to UML 1, this revision of UML has been enhanced with significantly more precise definitions of its abstract syntax rules and semantics, a more modular language structure, and a greatly improved capability for modeling large-scale systems.

One of the primary goals of UML is to advance the state of the industry by enabling object visual modeling tool interoperability. However, to enable meaningful exchange of model information between tools, agreement on semantics and notation is required. UML meets the following requirements:

- A formal definition of a common MOF-based metamodel that specifies the abstract syntax of the UML. The abstract syntax defines the set of UML modeling concepts, their attributes and their relationships, as well as the rules for combining these concepts to construct partial or complete UML models.
- A detailed explanation of the semantics of each UML modeling concept. The semantics define, in a technology-independent manner, how the UML concepts are to be realized by computers.
- A specification of the human-readable notation elements for representing the individual UML modeling concepts as well as rules for combining them into a variety of different diagram types corresponding to different aspects of modeled systems.
- A detailed definition of ways in which UML tools can be made compliant with this specification. This is supported (in a separate specification) with an XML-based specification of corresponding model interchange formats (XMI) that must be realized by compliant tools.

2 Conformance

UML is a language with a very broad scope that covers a large and diverse set of application domains. Not all of its modeling capabilities are necessarily useful in all domains or applications. This suggests that the language should be structured modularly, with the ability to select only those parts of the language that are of direct interest. On the other hand, an excess of this type of flexibility increases the likelihood that two different UML tools will be supporting different subsets of the language, leading to interchange problems between them. Consequently, the definition of compliance for UML requires a balance to be drawn between modularity and ease of interchange.

Experience with previous versions of UML has indicated that the ability to exchange models between tools is of paramount interest to a large community of users. For that reason, this specification defines a small number of *compliance levels* thereby increasing the likelihood that two or more compliant tools will support the same or compatible language subsets. However, in recognition of the need for flexibility in learning and using the language, UML also provides the concept of *language units*.

2.1 Language Units

The modeling concepts of UML are grouped into *language units*. A language unit consists of a collection of tightly-coupled modeling concepts that provide users with the power to represent aspects of the system under study according to a particular paradigm or formalism. For example, the State Machines language unit enables modelers to specify discrete event-driven behavior using a variant of the well-known statecharts formalism, while the Activities language unit provides for modeling behavior based on a workflow-like paradigm. From the user's perspective, this partitioning of UML means that they need only be concerned with those parts of the language that they consider necessary for their models. If those needs change over time, further language units can be added to the user's repertoire as required. Hence, a UML user does not have to know the full language to use it effectively.

In addition, most language units are partitioned into multiple increments, each adding more modeling capabilities to the previous ones. This fine-grained decomposition of UML serves to make the language easier to learn and use, but the individual segments within this structure do not represent separate compliance points. The latter strategy would lead to an excess of compliance points and result to the interoperability problems described above. Nevertheless, the groupings provided by language units and their increments do serve to simplify the definition of UML compliance as explained below.

2.2 Compliance Levels

The stratification of language units is used as the foundation for defining compliance in UML. Namely, the set of modeling concepts of UML is partitioned into horizontal layers of increasing capability called *compliance levels*. Compliance levels cut across the various language units, although some language units are only present in the upper levels. As their name suggests, each compliance level is a distinct compliance point.

For ease of model interchange, there are just two compliance levels defined for UML Infrastructure:

- *Level 0 (L0)* - This contains a single language unit that provides for modeling the kinds of class-based structures encountered in most popular object-oriented programming languages. As such, it provides an entry-level modeling capability. More importantly, it represents a low-cost common denominator that can serve as a basis for interoperability between different categories of modeling tools.
- *Metamodel Constructs (LM)* - This adds an extra language unit for more advanced class-based structures used for building metamodels (using CMOF) such as UML itself.

As noted, compliance levels build on supporting compliance levels. The principal mechanism used in this specification for achieving this is *package merge* (see Section 11.9.3, "PackageMerge," on page 162). Package merge allows modeling concepts defined at one level to be extended with new features. Most importantly, this is achieved *in the context of the same namespace*, which enables interchange of models at different levels of compliance as described in "Meaning and Types of Compliance."

For this reason, all compliance levels are defined as extensions to a single core "UML" package that defines the common namespace shared by all the compliance levels. Level 0 is defined by the top-level metamodel shown below.

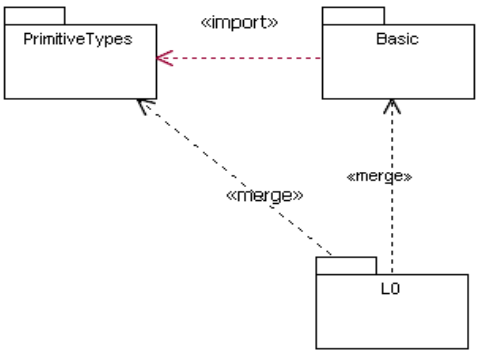


Figure 2.1 - Level 0 package diagram

In this model, ”UML” is originally an empty package that simply merges in the contents of the Basic package from the UML Infrastructure. This package, contains elementary concepts such as Class, Package, DataType, Operation, etc.

At the next level (Level LM), the contents of the “UML” package, now including the packages merged into Level 0 and their contents, are extended with the Constructs package.

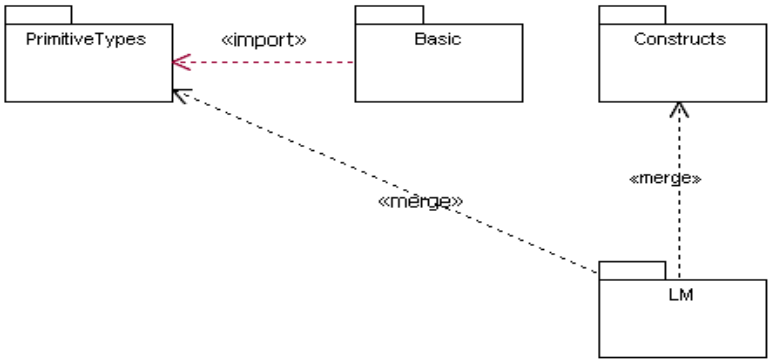


Figure 2.2 - Level M package diagram

Note that LM does not explicitly merge Basic, since the elements in Basic are already incorporated into the corresponding elements in Constructs.

2.3 Meaning and Types of Compliance

Compliance to a given level entails full realization of *all language units* that are defined for that compliance level. This also implies full realization of all language units in all the levels below that level. “Full realization” for a language unit at a given level means supporting the *complete set of modeling concepts* defined for that language unit *at that level*.

Thus, it is not meaningful to claim compliance to, say, Level 2 without also being compliant with the Level 0 and Level 1. A tool that is compliant at a given level must be able to import models from tools that are compliant to lower levels without loss of information.

There are two distinct types of compliance. They are:

- *Abstract syntax compliance*. For a given compliance level, this entails:
  - compliance with the metaclasses, their structural relationships, and any constraints defined as part of the merged UML metamodel for that compliance level, and
  - the ability to output models and to read in models based on the XMI schema corresponding to that compliance level.
- *Concrete syntax compliance*. For a given compliance level, this entails:
  - compliance to the notation defined in the "Notation" sub clauses in this specification for those metamodel elements that are defined as part of the merged metamodel for that compliance level and, by implication, the diagram types in which those elements may appear; and optionally
  - the ability to output diagrams and to read in diagrams based on the XMI schema defined by the Diagram Interchange specification for notation at that level. This option requires abstract syntax and concrete syntax compliance.

    Concrete syntax compliance does not require compliance to any presentation options that are defined as part of the notation.

Compliance for a given level can be expressed as:

- abstract syntax compliance
- concrete syntax compliance
- abstract syntax with concrete syntax compliance
- abstract syntax with concrete syntax and diagram interchange compliance

Table 2.1 - Example compliance statement

Compliance Summary			
Compliance level	**Abstract Syntax**	**Concrete Syntax**	**Diagram Interchange Option**
L0	YES	YES	NO
LM	NO	YES	NO

In case of tools that generate program code from models or those that are capable of executing models, it is also useful to understand the level of support for the run-time semantics described in the various "Semantics" sub clauses of the specification. However, the presence of numerous variation points in these semantics (and the fact that they are defined informally using natural language), make it impractical to define this as a formal compliance type, since the number of possible combinations is very large.

A similar situation exists with presentation options, since different implementers may make different choices on which ones to support. Finally, it is recognized that some implementers and profile designers may want to support only a subset of features from levels that are above their formal compliance level. (Note, however, that they can only claim compliance to the level that they fully support, even if they implement significant parts of the capabilities of higher levels.) Given this potential variability, it is useful to be able to specify clearly and efficiently, which capabilities are supported by a given implementation. To this end, in addition to a formal statement of compliance, implementers and profile designers may

also provide informal *feature support statements*. These statements identify support for additional features in terms of language units and/or individual metamodel packages, as well as for less precisely defined dimensions such as presentation options and semantic variation points.

An example feature support statement is shown in Table 2.2 for an implementation whose compliance statement is given in Table 2.1. In this case, the implementation adds two new language units from higher levels.

Table 2.2 - Example feature support statement

Feature Support Statement	
Language Unit	**Feature**
Constructs	An Association A1 specializes another Association A2 if each end of A1 subsets the corresponding end of A2.
Constructs	A redefining property must have the same name as the redefined property.

2.4 Compliance Level Contents

Table 2.3 identifies the packages by individual compliance levels in addition to those that are defined in lower levels (as a rule, Level (N) includes all the packages supported by Level (N-1)). The set of actual modeling features added by each of the packages are described in the appropriate clauses of the related language unit.

Table 2.3 - Metamodel packages added to compliance levels

Level	**Metamodel Package Added**
L0	Basic
LM	Constructs

3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

- RFC2119, http://ietf.org/rfc/rfc2119, Key words for use in RFCs to Indicate Requirement Levels, S. Bradner, March 1997.
- ISO/IEC 19505-2 , Information technology — OMG Unified Modeling Language (OMG UML) Version 2.4 — Part 2: Superstructure; pas/2011-08-12
- OMG Specification formal/2011-08-06, UML Superstructure, v2.4.1
- OMG Specification formal/2010-02-01, Object Constraint Language, v2.2
- OMG Specification formal/2011-08-07, Meta Object Facility (MOF) Core, v2.4.1
- OMG Specification formal/2011-08-09, XMI Metadata Interchange (XMI) v2.4.1
- OMG Specification formal/06-04-04 , UML 2.0 Diagram Interchange

Note – UML 2 is based on a different generation of MOF and XMI than that specified in ISO/IEC 19502:2005 Information technology - Meta Object Facility (MOF) and ISO/IEC 19503:2005 Information technology - XML Metadata Interchange (XMI) that are compatible with ISO/IEC 19501 UML version 1.4.1.

4 Terms and Definitions

There are no formal definitions in this specification that are taken from other documents.

5 Notational Conventions

The keywords “must,” “must not,” “shall,” “shall not,” “should,” “should not,” and “may” in this specification are to be interpreted as described in RFC 2119.

6 Additional Information

6.1 Architectural Alignment and MDA Support

Clause 7, “Language Architecture,” explains how the *UML 2: Infrastructure* is architecturally aligned with the *UML 2: Superstructure* that complements it. It also explains how the InfrastructureLibrary defined in the *UML 2: Infrastructure* can be strictly reused by MOF 2 specifications.

The *MOF 2: Core* Specification is architecturally aligned with this specification.

The OMG’s Model Driven Architecture (MDA) initiative is an evolving conceptual architecture for a set of industry-wide technology specifications that will support a model-driven approach to software development. Although MDA is not itself a technology specification, it represents an important approach and a plan to achieve a cohesive set of model-driven technology specifications. This specification’s support for MDA is discussed in *Annex B: “Support for Model Driven Architecture,” on page 211*.

6.2 How to Proceed

The rest of this document contains the technical content of this specification. Readers are encouraged to first read “Subpart I - Introduction” to familiarize themselves with the structure of the language and the formal approach used for its specification. Afterwards the reader may choose to either explore the InfrastructureLibrary, described in “Subpart II - Infrastructure Library” or the UML::Classes::Kernel package that reuses it, described in the *UML 2: Superstructure*. The former specifies the flexible metamodel library that is reused by the latter.

Readers who want to explore the user level constructs that are built upon the infrastructural constructs specified here should investigate the specification that complements this, the *UML 2: Superstructure*.

Although the clauses are organized in a logical manner and can be read sequentially, this is a reference specification intended to be read in a non-sequential manner. Consequently, extensive cross-references are provided to facilitate browsing and search.

6.2.1 Diagram format

The following conventions are adopted for all metamodel diagrams throughout this specification:

- An association with one end marked by a navigability arrow means that:
  - the association is navigable in the direction of that end,
  - the marked association end is owned by the classifier, and
  - the opposite (unmarked) association end is owned by the association.
- An association with neither end marked by navigability arrows means that:
  - the association is navigable in both directions,
  - each association end is owned by the classifier at the opposite end (i.e., neither end is owned by the association).
- Association specialization and redefinition are indicated by appropriate constraints situated in the proximity of the association ends to which they apply. Thus:
  - the constraint {subsets endA} means that the association end to which this constraint is applied is a specialization of association end endA that is part of the association being specialized.
  - a constraint {redefines endA} means that the association end to which this constraint is applied redefines the association end endA that is part of the association being specialized.
- If no multiplicity is shown on an association end, it implies a multiplicity of exactly 1.
- If an association end is unlabeled, the default name for that end is the name of the class to which the end is attached, modified such that the first letter is a lowercase letter. (Note that, by convention, non-navigable association ends are often left unlabeled since, in general, there is no need to refer to them explicitly either in the text or in formal constraints – although they may be needed for other purposes, such as MOF language bindings that use the metamodel.)
- Associations that are not explicitly named, are given names that are constructed according to the following production rule:

  "A_" <*association-end-name1*> *"_"* <*association-end-name2*>

  where <*association-end-name1*> is the name of the first association end and <*association-end-name2*> is the name of the second association end.
- An unlabeled dependency between two packages is interpreted as a package import relationship.

Note that some of these conventions were adopted to contend with practical issues related to the mechanics of producing this specification, such as the unavailability of conforming modeling tools at the time the specification itself was being defined. Therefore, they should not necessarily be deemed as recommendations for general use.

Subpart I - Introduction

The Unified Modeling Language is a visual language for specifying, constructing, and documenting the artifacts of systems. It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains (e.g., health, finance, telecom, aerospace) and implementation platforms (e.g., J2EE, .NET).

The OMG adopted the UML 1.1 specification in November 1997. Since then UML Revision Task Forces have produced several minor revisions, the most recent being the UML 1.4 specification, which was adopted in May 2001.

Under the stewardship of the OMG, the UML has emerged as the software industry's dominant modeling language. It has been successfully applied to a wide range of domains, ranging from health and finance to aerospace to e-commerce. As should be expected, its extensive use has raised numerous application and implementation issues by modelers and vendors. As of the time of this writing over 500 formal usage and implementation issues have been submitted to the OMG for consideration.

Although many of the issues have been resolved in minor revisions by Revision Task Forces, other issues require major changes to the language that are outside the scope of an RTF. Consequently, the OMG issued four complementary and architecturally aligned RFPs to define UML: UML Infrastructure, UML Superstructure, UML Object Constraint Language, and UML Diagram Interchange.

This UML specification is organized into two volumes (*UML 2: Infrastructure* and *UML 2: Superstructure*), consistent with the breakdown of modeling language requirements into two RFPs (*UML Infrastructure RFP* and *UML Superstructure RFP*). Since the two volumes cross-reference each other and the specifications are fully integrated, these two volumes could easily be combined into a single volume at a later time.

The next two clauses describe the language architecture and the specification approach used to define UML 2.

7 Language Architecture

The UML specification is defined using a metamodeling approach (i.e., a metamodel is used to specify the model that comprises UML) that adapts formal specification techniques. While this approach lacks some of the rigor of a formal specification method, it offers the advantages of being more intuitive and pragmatic for most implementers and practitioners.[1] This clause explains the architecture of the UML metamodel.

The following sub clauses summarize the design principles followed, and show how they are applied to organize UML's Infrastructure and Superstructure. The last sub clause explains how the UML metamodel conforms to a 4-layer metamodel architectural pattern.

7.1 Design Principles

The UML metamodel has been architected with the following design principles in mind:

- Modularity — This principle of strong cohesion and loose coupling is applied to group constructs into packages and organize features into metaclasses.
- Layering — Layering is applied in two ways to the UML metamodel. First, the package structure is layered to separate the metalanguage core constructs from the higher-level constructs that use them. Second, a 4-layer metamodel architectural pattern is consistently applied to separate concerns (especially regarding instantiation) across layers of abstraction.
- Partitioning — Partitioning is used to organize conceptual areas within the same layer. In the case of the InfrastructureLibrary, fine-grained partitioning is used to provide the flexibility required by current and future metamodeling standards. In the case of the UML metamodel, the partitioning is coarser-grained in order to increase the cohesion within packages and loosening the coupling across packages.
- Extensibility — The UML can be extended in two ways:
  - A new dialect of UML can be defined by using Profiles to customize the language for particular platforms (e.g., J2EE/EJB, .NET/COM+) and domains (e.g., finance, telecommunications, aerospace).
  - A new language related to UML can be specified by reusing part of the InfrastructureLibrary package and augmenting with appropriate metaclasses and metarelationships. The former case defines a new dialect of UML, while the latter case defines a new member of the UML family of languages.
- Reuse — A fine-grained, flexible metamodel library is provided that is reused to define the UML metamodel, as well as other architecturally related metamodels, such as the Meta Object Facility (MOF) and the Common Warehouse Metamodel (CWM).

7.2 Infrastructure Architecture

The Infrastructure of the UML is defined by the InfrastructureLibrary, which satisfies the following design requirements:

---

1. It is important to note that the specification of UML as a metamodel does not preclude it from being specified via a mathematically formal language (e.g., Object-Z or VDM) at a later time.

- Define a metalanguage core that can be reused to define a variety of metamodels, including UML, MOF, and CWM.
- Architecturally align UML, MOF, and XMI so that model interchange is fully supported.
- Allow customization of UML through Profiles and creation of new languages (family of languages) based on the same metalanguage core as UML.

As shown in Figure 7.1, Infrastructure is represented by two packages: *InfrastructureLibrary* and *PrimitiveTypes*. The package *InfrastructureLibrary* consists of the packages *Core* and *Profiles*, where the latter defines the mechanisms that are used to customize metamodels and the former contains core concepts used when metamodeling. The package *PrimitiveTypes* consists of a few predefined primitive types that are commonly used when metamodeling, and is designed specifically with the needs of UML and MOF in mind.

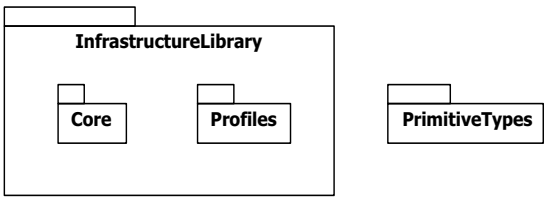


Figure 7.1 - The InfrastructureLibrary packages

7.3 Core

In its first capacity, the *Core* package is a complete metamodel particularly designed for high reusability, where other metamodels at the same metalevel (see Section 7.6, “Superstructure Architecture,” on page 15) either import or specialize its specified metaclasses. This is illustrated in Figure 7.2, where it is shown how UML, CWM, and MOF each depends on a *common* core. Since these metamodels are at the very heart of the Model Driven Architecture (MDA), the common core may also be considered the architectural kernel of MDA. The intent is for UML and other MDA metamodels to reuse all or parts of the *Core* package, which allows other metamodels to benefit from the abstract syntax and semantics that have already been defined.

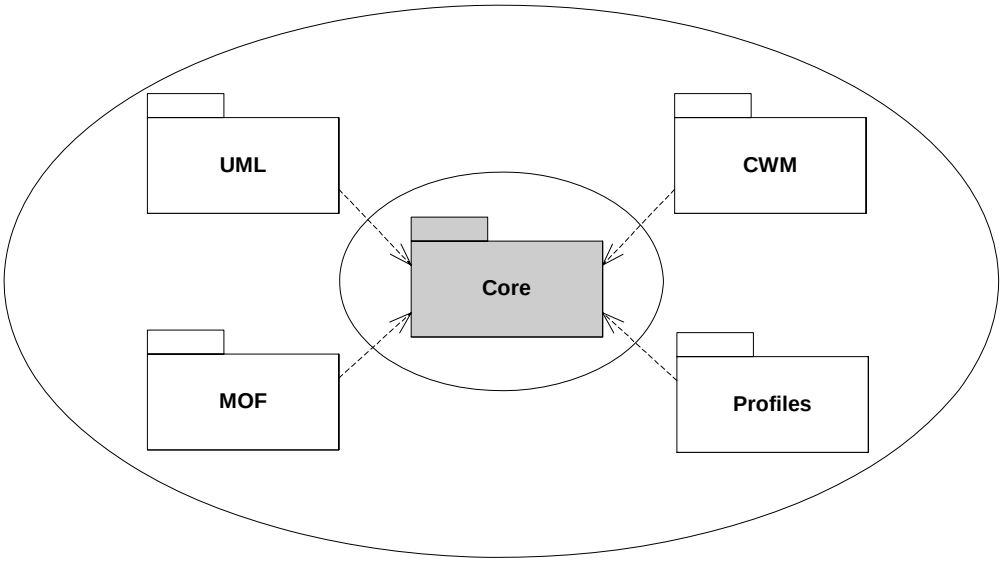


Figure 7.2 - The role of the common Core

In order to facilitate reuse, the *Core* package is subdivided into a number of packages: *Abstractions*, *Basic*, and *Constructs* as shown in Figure 7.3. As we will see in subsequent clauses, some of these are then further divided into even more fine-grained packages to make it possible to pick and choose the relevant parts when defining a new metamodel. Note, however, that choosing a specific package also implies choosing the dependent packages. There are minor differences in the design rationale for the other three packages. The package *Abstractions* mostly contains abstract metaclasses that are intended to be further specialized or that are expected to be commonly reused by many metamodels. Very few assumptions are made about the metamodels that may want to reuse this package; for this reason, the package *Abstractions* is also subdivided into several smaller packages. The package *Constructs*, on the other hand, mostly contains concrete metaclasses that lend themselves primarily to object-oriented modeling; this package in particular is reused by both MOF and UML, and represents a significant part of the work that has gone into aligning the two metamodels. The package *Basic* represents a few constructs that are used as the basis for the produced XMI for UML, MOF, and other metamodels based on the *InfrastructureLibrary.*

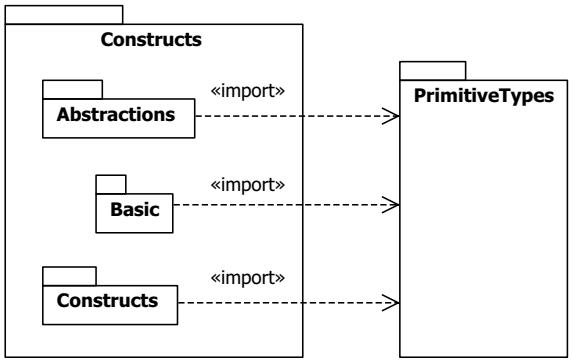


Figure 7.3 - The Core packages

In its second capacity, the *Core* package is used to define the modeling constructs used to create metamodels. This is done through instantiation of metaclasses in the *InfrastructureLibrary* (see Section 7.9, “Metamodel Layering,” on page 17). While instantiation of metaclasses is carried out through MOF, the *InfrastructureLibrary* defines the actual metaclasses that are used to instantiate the elements of UML, MOF, CWM, and indeed the elements of the *InfrastructureLibrary* itself. In this respect, the *InfrastructureLibrary* is said to be self-describing, or *reflective*.

7.4 Profiles

As was depicted in Figure 7.1, the *Profiles* package depends on the *Core* package, and defines the mechanisms used to tailor existing metamodels towards specific platforms, such as C++, CORBA, or EJB; or domains such as real-time, business objects, or software process modeling. The primary target for profiles is UML, but it is possible to use profiles together with any metamodel that is based on (i.e., instantiated from) the common core. A profile must be based on a metamodel such as the UML that it extends, and is not very useful standalone.

Profiles have been aligned with the extension mechanism offered by MOF, but provide a more light-weight approach with restrictions that are enforced to ensure that the implementation and usage of profiles should be straightforward and more easily supported by tool vendors.

7.5 Architectural Alignment between UML and MOF

One of the major goals of the Infrastructure has been to architecturally align UML and MOF. The first approach to accomplish this has been to define the common core, which is realized as the package *Core*, in such a way that the model elements are shared between UML and MOF. The second approach has been to make sure that UML is defined as a model that is based on MOF used as a metamodel, as is illustrated in Figure 7.4. Note that MOF is used as the metamodel for not only UML, but also for other languages such as CWM.

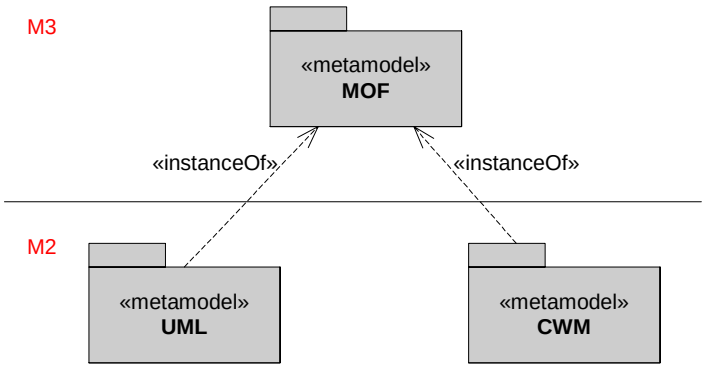


Figure 7.4 - UML and MOF are at different metalevels

How these metalevel hierarchies work is explained in more detail in Section 7.6, “Superstructure Architecture,” on page 15. An important aspect that deserves mentioning here is that every model element of UML is an instance of exactly one model element in MOF. Note that the *InfrastructureLibrary* is used at both the M2 and M3 metalevels, since it is being reused by UML and MOF, respectively, as was shown in Figure 7.2. In the case of MOF, the metaclasses of the *InfrastructureLibrary* are used as is, while in the case of UML these model elements are given additional properties. The reason for these differences is that the requirements when metamodeling differ slightly from the requirements when modeling applications of a very diverse nature.

MOF defines for example how UML models are interchanged between tools using XML Metadata Interchange (XMI). MOF also defines reflective interfaces (MOF::Reflection) for introspection that work for not only MOF itself, but also for CWM, UML, and for any other metamodel that is an instance of MOF. It further defines an extension mechanism that can be used to extend metamodels as an alternative to or in conjunction with profiles (as described in Clause 13, “Core::Profiles”). In fact, profiles are defined to be a subset of the MOF extension mechanism.

7.6 Superstructure Architecture

The UML Superstructure metamodel is specified by the *UML* package, which is divided into a number of packages that deal with structural and behavioral modeling, as shown in Figure 7.5.

Each of these areas is described in a separate clause of the *UML 2: Superstructure* specification. Note that there are some packages that are dependent on each other in circular dependencies. This is because the dependencies between the top-level packages show a summary of all relationships between their subpackages; there are no circular dependencies between subpackages of those packages.

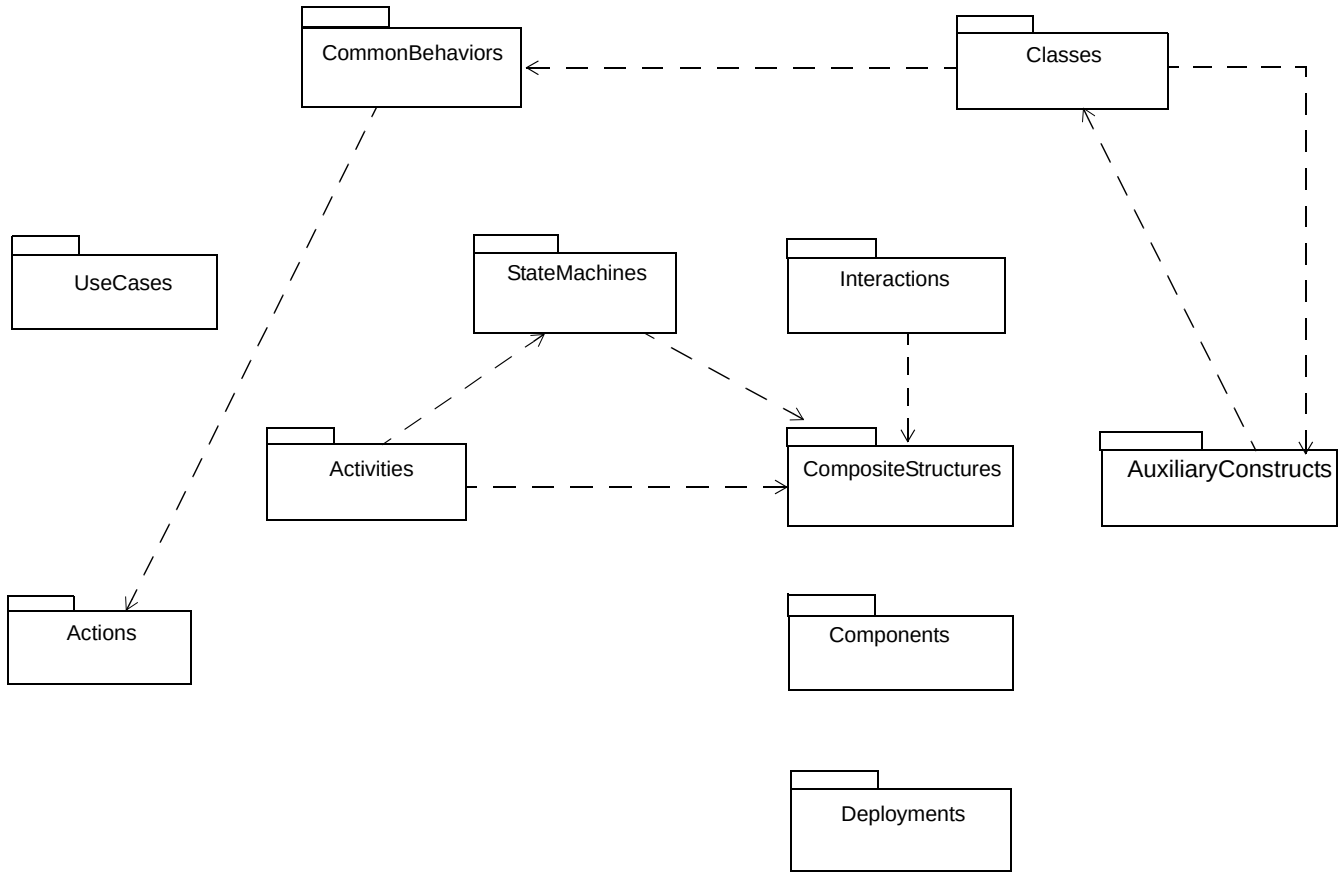


Figure 7.5 - The top-level package structure of the UML 2 Superstructure

7.7 Reusing Infrastructure

One of the primary uses of the UML 2 Infrastructure specification is that it should be reused when creating other metamodels. The UML metamodel reuses the *InfrastructureLibrary* in two different ways:

- All of the UML metamodel is instantiated from meta-metaclasses that are defined in the *InfrastructureLibrary.*
- The UML metamodel imports and specializes all metaclasses in the *InfrastructureLibrary.*

As was discussed earlier, it is possible for a model to be used as a metamodel, and here we make use of this fact. The *InfrastructureLibrary* is in one capacity used as a meta-metamodel and in the other aspect as a metamodel, and is thus reused in two dimensions.

7.8 The Kernel Package

The *InfrastructureLibrary* is primarily reused in the *Kernel* package of *Classes* in *UML 2: Superstructure*; this is done by bringing together the different packages of the Infrastructure using package merge. The *Kernel* package is at the very heart of UML, and the metaclasses of every other package are directly or indirectly dependent on it. The *Kernel* package is very similar to the *Constructs* package of the *InfrastructureLibrary*, but adds more capabilities to the modeling constructs that were not necessary to include for purposes of reuse or alignment with MOF.

Because the Infrastructure has been designed for reuse, there are metaclasses—particularly in *Abstractions*—that are partially defined in several different packages. These different aspects are for the most part brought together into a single metaclass already in *Constructs*, but in some cases this is done only in *Kernel*. In general, if metaclasses with the same name occur in multiple packages, they are meant to represent the same metaclass, and each package where it is defined (specialized) represents a specific factorization. This same pattern of partial definitions also occurs in Superstructure, where some aspects of, for example, the metaclass *Class* are factored out into separate packages to form compliance points (see below).

7.9 Metamodel Layering

The architecture that is centered around the *Core* package is a complementary view of the four-layer metamodel hierarchy on which the UML metamodel has traditionally been based. When dealing with meta-layers to define languages there are generally three layers that always have to be taken into account:

1. the language specification, or the metamodel,
2. the user specification, or the model, and
3. objects of the model.

This structure can be applied recursively many times so that we get a possibly infinite number of meta-layers; what is a metamodel in one case can be a model in another case, and this is what happens with UML and MOF. UML is a language specification (metamodel) from which users can define their own models. Similarly, MOF is also a language specification (metamodel) from which users can define their own models. From the perspective of MOF, however, UML is viewed as a user (i.e., the members of the OMG that have developed the language) specification that is based on MOF as a language specification. In the four-layer metamodel hierarchy, MOF is commonly referred to as a meta-metamodel, even though strictly speaking it is a metamodel.

7.10 The Four-layer Metamodel Hierarchy

The meta-metamodeling layer forms the foundation of the metamodeling hierarchy. The primary responsibility of this layer is to define the language for specifying a metamodel. The layer is often referred to as M3, and MOF is an example of a meta-metamodel. A meta-metamodel is typically more compact than a metamodel that it describes, and often defines several metamodels. It is generally desirable that related metamodels and meta-metamodels share common design philosophies and constructs. However, each layer can be viewed independently of other layers, and needs to maintain its own design integrity.

A metamodel is an instance of a meta-metamodel, meaning that every element of the metamodel is an instance of an element in the meta-metamodel. The primary responsibility of the metamodel layer is to define a language for specifying models. The layer is often referred to as M2; UML and the OMG Common Warehouse Metamodel (CWM) are examples

of metamodels. Metamodels are typically more elaborate than the meta-metamodels that describe them, especially when they define dynamic semantics. The UML metamodel is an instance of the MOF (in effect, each UML metaclass is an instance of an element in *InfrastructureLibrary*).

A model is an instance of a metamodel. The primary responsibility of the model layer is to define languages that describe semantic domains, i.e., to allow users to model a wide variety of different problem domains, such as software, business processes, and requirements. The things that are being modeled reside outside the metamodel hierarchy. This layer is often referred to as M1. A user model is an instance of the UML metamodel. Note that the user model contains both model elements and snapshots (illustrations) of instances of these model elements.

The metamodel hierarchy bottoms out at M0, which contains the run-time instances of model elements defined in a model. The snapshots that are modeled at M1 are constrained versions of the M0 run-time instances.

When dealing with more than three meta-layers, it is usually the case that the ones above M2 gradually get smaller and more compact the higher up they are in the hierarchy. In the case of MOF, which is at M3, it consequently only shares some of the metaclasses that are defined in UML. A specific characteristic about metamodeling is the ability to define languages as being reflective, i.e., languages that can be used to define themselves. The *InfrastructureLibrary* is an example of this, since it contains all the metaclasses required to define itself. MOF is reflective since it is based on the *InfrastructureLibrary.* This allows it to be used to define itself. For this reason, no additional meta-layers above MOF are defined.

7.11 Metamodeling

When metamodeling, we primarily distinguish between metamodels and models. As already stated, a model that is instantiated from a metamodel can in turn be used as a metamodel of another model in a recursive manner. A model typically contains model elements. These are created by instantiating model elements from a metamodel, i.e., metamodel elements.

The typical role of a metamodel is to define the semantics for how model elements in a model get instantiated. As an example, consider Figure 7.6, where the metaclasses Association and Class are both defined as part of the UML metamodel. These are instantiated in a user model in such a way that the classes Person and Car are both instances of the metaclass Class, and the association Person.car between the classes is an instance of the metaclass Association. The semantics of UML defines what happens when the user defined model elements are instantiated at M0, and we get an instance of Person, an instance of Car, and a link (i.e., an instance of the association) between them.

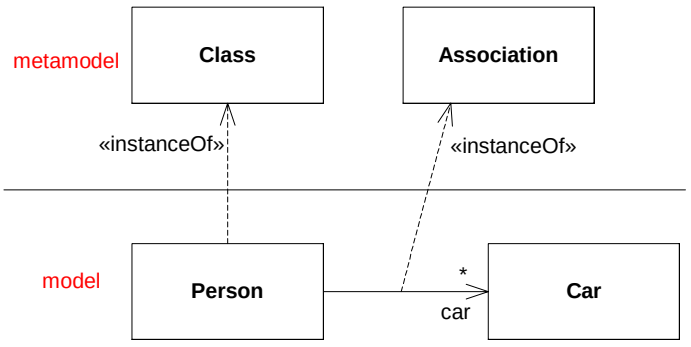


Figure 7.6 - An example of metamodeling; note that not all instance-of relationships are shown

The instances, which are sometimes referred to as “run-time” instances, that are created at M0 from for example Person should not be confused with instances of the metaclass InstanceSpecification that are also defined as part of the UML metamodel. An instance of an InstanceSpecification is defined in a model at the same level as the model elements that it illustrates, as is depicted in Figure 7.7, where the instance specification Mike is an illustration (or a snapshot) of an instance of class Person.

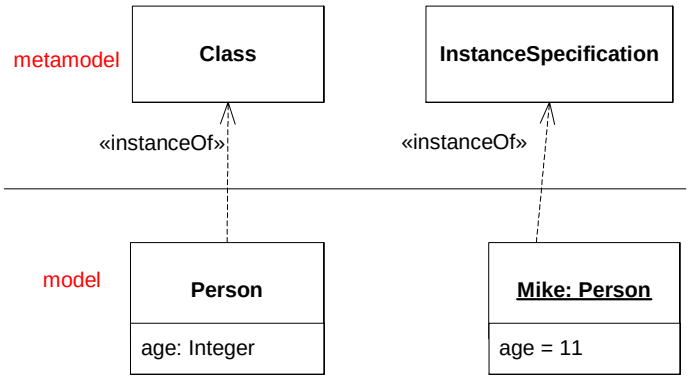


Figure 7.7 - Giving an illustration of a class using an instance specification

7.12 An Example of the Four-level Metamodel Hierarchy

An illustration of how these meta-layers relate to each other is shown in Figure 7.8. It should be noted that we are by no means restricted to only these four meta-layers, and it would be possible to define additional ones. As is shown, the meta-layers are usually numbered from M0 and upwards, depending on how many meta-layers are used. In this particular case, the numbering goes up to M3, which corresponds to MOF.

.

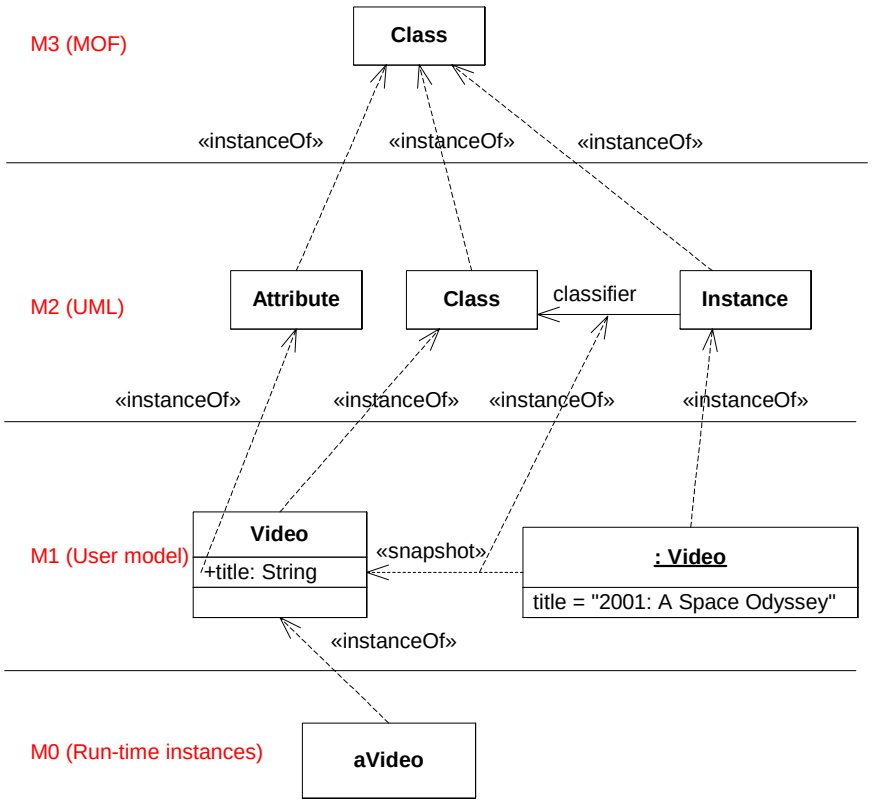


Figure 7.8 - An example of the four-layer metamodel hierarchy

8 Language Formalism

The UML specification is defined by using a metamodeling approach that adapts formal specification techniques. The formal specification techniques are used to increase the precision and correctness of the specification. This clause explains the specification techniques used to define UML.

The following are the goals of the specification techniques used to define UML:

- Correctness — The specification techniques should improve the correctness of the metamodel by helping to validate it. For example, the well-formedness rules should help validate the abstract syntax and help identify errors.
- Precision — The specification techniques should increase the precision of both the syntax and semantics. The precision should be sufficient so that there is no syntactic nor semantic ambiguity for either implementors or users.[1]
- Conciseness — The specification techniques should be parsimonious, so that the precise syntax and semantics are defined without superfluous detail.
- Consistency — The specification techniques should complement the metamodeling approach by adding essential detail in a consistent manner.
- Understandability — While increasing the precision and conciseness, the specification techniques should also improve the readability of the specification. For this reason a less than strict formalism is applied, since a strict formalism would require formal techniques.

The specification technique used describes the metamodel in three views using both text and graphic presentations.

It is important to note that the current description is not a completely formal specification of the language because to do so would have added significant complexity without clear benefit.

The structure of the language is nevertheless given a precise specification, which is required for tool interoperability. The detailed semantics are described using natural language, although in a precise way so they can easily be understood. Currently, the semantics are not considered essential for the development of tools; however, this will probably change in the future.

8.1 Levels of Formalism

A common technique for specification of languages is to first define the syntax of the language and then to describe its static and dynamic semantics. The syntax defines what constructs exist in the language and how the constructs are built up in terms of other constructs. Sometimes, especially if the language has a graphic syntax, it is important to define the syntax in a notation independent way (i.e., to define the abstract syntax of the language). The concrete syntax is then defined by mapping the notation onto the abstract syntax.

The static semantics of a language define how an instance of a construct should be connected to other instances to be meaningful, and the dynamic semantics define the meaning of a well formed construct. The meaning of a description written in the language is defined only if the description is well formed (i.e., if it fulfills the rules defined in the static semantics).

---

1. By definition semantic variation points are an exception to this.

The specification uses a combination of languages - a subset of UML, an object constraint language, and precise natural language to describe the abstract syntax and semantics of the full UML. The description is self-contained; no other sources of information are needed to read the document[2]. Although this is a metacircular description[3], understanding this document is practical since only a small subset of UML constructs are needed to describe its semantics.

In constructing the UML metamodel different techniques have been used to specify language constructs, using some of the capabilities of UML. The main language constructs are reified into metaclasses in the metamodel. Other constructs, in essence being variants of other ones, are defined as stereotypes of metaclasses in the metamodel. This mechanism allows the semantics of the variant construct to be significantly different from the base metaclass. Another more “lightweight” way of defining variants is to use metaattributes. As an example, the aggregation construct is specified by an attribute of the metaclass Property, which is used to indicate if an association is an ordinary aggregate, a composite aggregate, or a common association.

8.2 Package Specification Structure

This sub clause provides information for each package and each class in the UML metamodel. Each package has one or more of the following sub clauses.

8.2.1 Class Descriptions

The sub clause contains an enumeration of the classes specifying the constructs defined in the package. It begins with one diagram or several diagrams depicting the abstract syntax of the constructs (i.e., the classes and their relationships) in the package, together with some of the well-formedness requirements (multiplicity and ordering). Then follows a specification of each class in alphabetic order (see below).

8.2.2 Diagrams

If a specific kind of diagram usually presents the constructs that are defined in the package, a sub clause describing this kind of diagram is included.

8.2.3 Instance Model

An example may be provided to show how an instance model of the contained classes may be populated. The elements in the example are instances of the classes contained in the package (or in an imported package).

8.3 Class Specification Structure

The specification of a class starts with a presentation of the general meaning of the concept that sets the context for the definition.

---

2. Although a comprehension of the UML’s four-layer metamodel architecture and its underlying meta-metamodel is helpful, it is not essential to understand the UML semantics.
3. In order to understand the description of the UML semantics, you must understand some UML semantics.

8.3.1 Description

The sub clause includes an informal definition of the metaclass specifying the construct in UML. The sub clause states if the metaclass is abstract. This sub clause, together with the following two, constitutes a description of the abstract syntax of the construct.

8.3.2 Attributes

Each of the attributes of the class are enumerated together with a short explanation. The sub clause states if the attribute is derived, or if it is a specialization of another attribute. The multiplicity of the attribute is suppressed it defaults to ‘1’ (default in UML).

8.3.3 Associations

The member ends of associations connected to the class are also listed in the same way. The sub clause states if the property is derived, or if it subsets or redefines another end.

8.3.4 Constraints

The well-formedness rules of the metaclass, except for multiplicity and ordering constraints that are defined in the diagram at the beginning of the package sub clause, are defined as a (possibly empty) set of invariants for the metaclass, which must be satisfied by all instances of that metaclass for the model to be meaningful. The rules thus specify constraints over attributes and associations defined in the metamodel. Most invariants are defined by OCL expressions together with an informal explanation of the expression, but in some cases invariants are expressed by other means (in exceptional cases with natural language). The statement ‘No additional constraints’ means that all well-formedness rules are expressed in the superclasses together with the multiplicity and type information expressed in the diagrams.

8.3.5 Additional Operations (optional)

In many cases, additional operations on the classes are needed for the OCL expressions. These are then defined in a separate sub clause after the constraints for the construct, using the same approach as the Constraints sub clause: an informal explanation followed by the OCL expression defining the operation.

8.3.6 Semantics

The meaning of a well formed construct is defined using natural language.

8.3.7 Semantic Variation Points (optional)

The term *semantic variation point* is used throughout this document to denote a part of the UML specification whose purpose in the overall specification is known but whose form or semantics may be varied in some way. The objective of a semantic variation point is to enable specialization of that part of UML for a particular situation or domain.

There are several forms in which semantic variation points appear in the standard:

- *Changeable default* — in this case, a single default specification for the semantic variation point is provided in the standard but it may be replaced. For example, the standard provides a default set of rules for specializing state

machines, but this default can be overridden (e.g., in a profile) by a different set of rules (the choice typically depends on which definition of behavioral compatibility is used).

- *Multiple choice* — in this case, the standard explicitly specifies a number of possible mutually exclusive choices, one of which may be marked as the default. Language designers may either select one of those alternatives or define a new one. An example of this type of variation point can be found in the handling of unexpected events in state machines; the choices include (a) ignoring the event (the default), (b) explicitly rejecting it, or (c) deferring it.
- *Undefined* — in this case, the standard does not provide any pre-defined specifications for the semantic variation point. For instance, the rules for selecting the method to be executed when a polymorphic operation is invoked are not defined in the standard.

8.3.8 Notation

The notation of the construct is presented in this sub clause.

8.3.9 Presentation Options (optional)

If there are different ways to show the construct (e.g., it is not necessary to show all parts of the construct in every occurrence), these possibilities are described in this sub clause.

8.3.10 Style Guidelines (optional)

Often non-normative conventions are used in representing some part of a model. For example, one such convention is to always have the name of a class in bold and centered within the class rectangle.

8.3.11 Examples (optional)

In this sub clause, examples of how the construct is to be depicted are given.

8.3.12 Rationale (optional)

If there is a reason why a construct is defined like it is, or why its notation is defined as it is, this reason is given in this sub clause.

8.3.13 Changes from UML 1.4

Here, changes compared with UML 1.4 are described and a migration approach from 1.4 to 2 is specified.

8.4 Use of a Constraint Language

The specification uses the Object Constraint Language (OCL), as defined in Clause 6, “Object Constraint Language Specification” of the UML 1.4 specification, for expressing well-formedness rules. The following conventions are used to promote readability:

- Self — which can be omitted as a reference to the metaclass defining the context of the invariant, has been kept for clarity.

- In expressions where a collection is iterated, an iterator is used for clarity, even when formally unnecessary. The type of the iterator is usually omitted, but included when it adds to understanding.
- The ‘collect’ operation is left implicit where this is practical.
- The context part of an OCL constraint is not included explicitly, as it is well defined in the sub clause where the constraint appears.

8.5 Use of Natural Language

We strove to be precise in our use of natural language, in this case English. For example, the description of UML semantics includes phrases such as “X provides the ability to…” and “X is a Y.” In each of these cases, the usual English meaning is assumed, although a deeply formal description would demand a specification of the semantics of even these simple phrases.

The following general rules apply:

- When referring to an instance of some metaclass, we often omit the word “instance.” For example, instead of saying “a Class instance” or “an Association instance,” we just say “a Class” or “an Association.” By prefixing it with an “a” or “an,” assume that we mean “an instance of.” In the same way, by saying something like “Elements” we mean “a set (or the set) of instances of the metaclass Element.”
- Every time a word coinciding with the name of some construct in UML is used, that construct is meant.
- Terms including one of the prefixes sub, super, or meta are written as one word (e.g., metamodel, subclass).

8.6 Conventions and Typography

In the description of UML, the following conventions have been used:

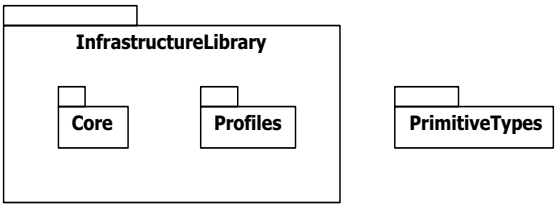
- When referring to constructs in UML, not their representation in the metamodel, normal text is used.
- Metaclass names that consist of appended nouns/adjectives, initial embedded capitals are used (e.g., ‘ModelElement,’ ‘StructuralFeature’).
- Names of metaassociations are written in the same manner as metaclasses (e.g., ‘ElementReference’).
- Initial embedded capital is used for names that consist of appended nouns/adjectives (e.g., ‘ownedElement,’ ‘allContents’).
- Boolean metaattribute names always start with ‘is’ (e.g., ‘isAbstract’).
- Enumeration types always end with “Kind” (e.g., ‘AggregationKind’).
- While referring to metaclasses, metaassociations, metaattributes, etc. in the text, the exact names as they appear in the model are always used.
- No visibilities are presented in the diagrams, as all elements are public.
- If a mandatory section does not apply for a metaclass, the text ‘No additional XXX’ is used, where ‘XXX’ is the name of the heading. If an optional section is not applicable, it is not included.

For textual notations a variant of the Backus-Naur Form (BNF) is often used to specify the legal formats. The conventions of this BNF are:

- All non-terminals are in italics and enclosed between angle brackets (e.g. ,*<non-terminal>*).
- All terminals (keywords, strings, etc.), are enclosed between single quotes (e.g., *'or'*).
- Non-terminal production rule definitions are signified with the '::=' operator.
- Repetition of an item is signified by an asterisk placed after that item: '*'
- Alternative choices in a production are separated by the '|' symbol; e.g., *<alternative-A>* | *<alternative-B>*.
- Items that are optional are enclosed in square brackets (e.g., *[<item-x>]*).
- Where items need to be grouped they are enclosed in simple parenthesis. For example, *(<item-1>* | *<item-2>*) * signifies a sequence of one or more items, each of which is *<item-1>* or *<item-2>*.

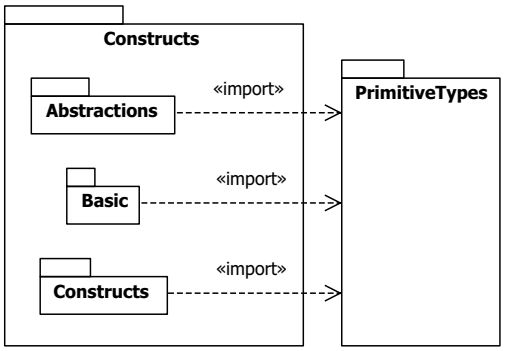
Subpart II - Infrastructure Library

This part describes the structure and contents of the Infrastructure packages for the UML metamodel and related metamodels, such as the Meta Object Facility (MOF) and the Common Warehouse Metamodel (CWM). The first top level package is *InfrastructureLibrary*, which defines a reusable metalanguage kernel and a metamodel extension mechanism for UML. The metalanguage kernel can be used to specify a variety of metamodels, including UML, MOF, and CWM. In addition, the library defines a profiling extension mechanism that can be used to customize UML for different platforms and domains without supporting a complete metamodeling capability. The nested packages of the *InfrastructureLibrary* are *Core* and *Profile*. The other top level package is *PrimitiveTypes*, which consists of a small number of primitive types that are commonly used for metamodeling. The *PrimitiveTypes* package is imported by nested packages in the *InfrastructureLibrary* and can be imported by other packages, libraries and metamodels that need to define primitive data. The high level architecture of the Infrastructure packages is shown below.



The Metamodel Library package contains the packages Core and Profiles

The *Core* package is the central reusable part of the InfrastructureLibrary, and is further subdivided as shown in the figure below.



The Core package contains the packages PrimitiveTypes, Abstractions, Basic, and Constructs

The package *PrimitiveTypes* is a simple package that contains a number of predefined types that are commonly used when metamodeling, and as such they are used both in the infrastructure library itself, but also in metamodels like MOF and UML. The package *Abstractions* contains a number of fine-grained packages with only a few metaclasses each, most of which are abstract. The purpose of this package is to provide a highly reusable set of metaclasses to be specialized when defining new metamodels. The package *Constructs* also contains a number of fine-grained packages, and brings together many of the aspects of the *Abstractions*. The metaclasses in *Constructs* tend to be concrete rather than abstract, and are geared towards an object-oriented modeling paradigm. Looking at metamodels such as MOF and UML, they typically import the *Constructs* package since the contents of the other packages of Core are then automatically included. The package *Basic* contains a subset of *Constructs* that is used primarily for XMI purposes.

The *Profiles* package contains the mechanisms used to create profiles of specific metamodels, and in particular of UML. This extension mechanism subsets the capabilities offered by the more general MOF extension mechanism.

The detailed structure and contents of the *PrimitiveTypes*, *Abstractions*, *Basic*, *Constructs*, and *Profiles* packages are further described in subsequent clauses.

9 Core::Abstractions

The Abstractions package of InfrastructureLibrary::Core is divided into a number of finer-grained packages to facilitate flexible reuse when creating metamodels.

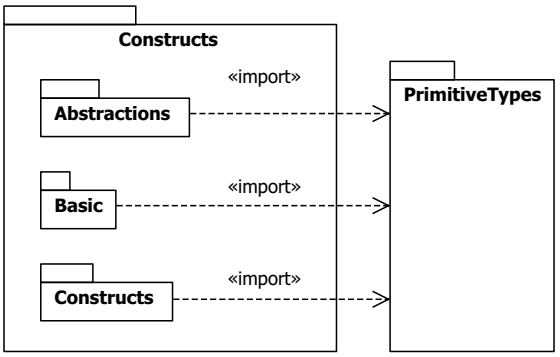


Figure 9.1 - The Core package is owned by the InfrastructureLibrary pack and contains several subpackages

The subpackages of Abstractions are all shown in Figure 9.2.

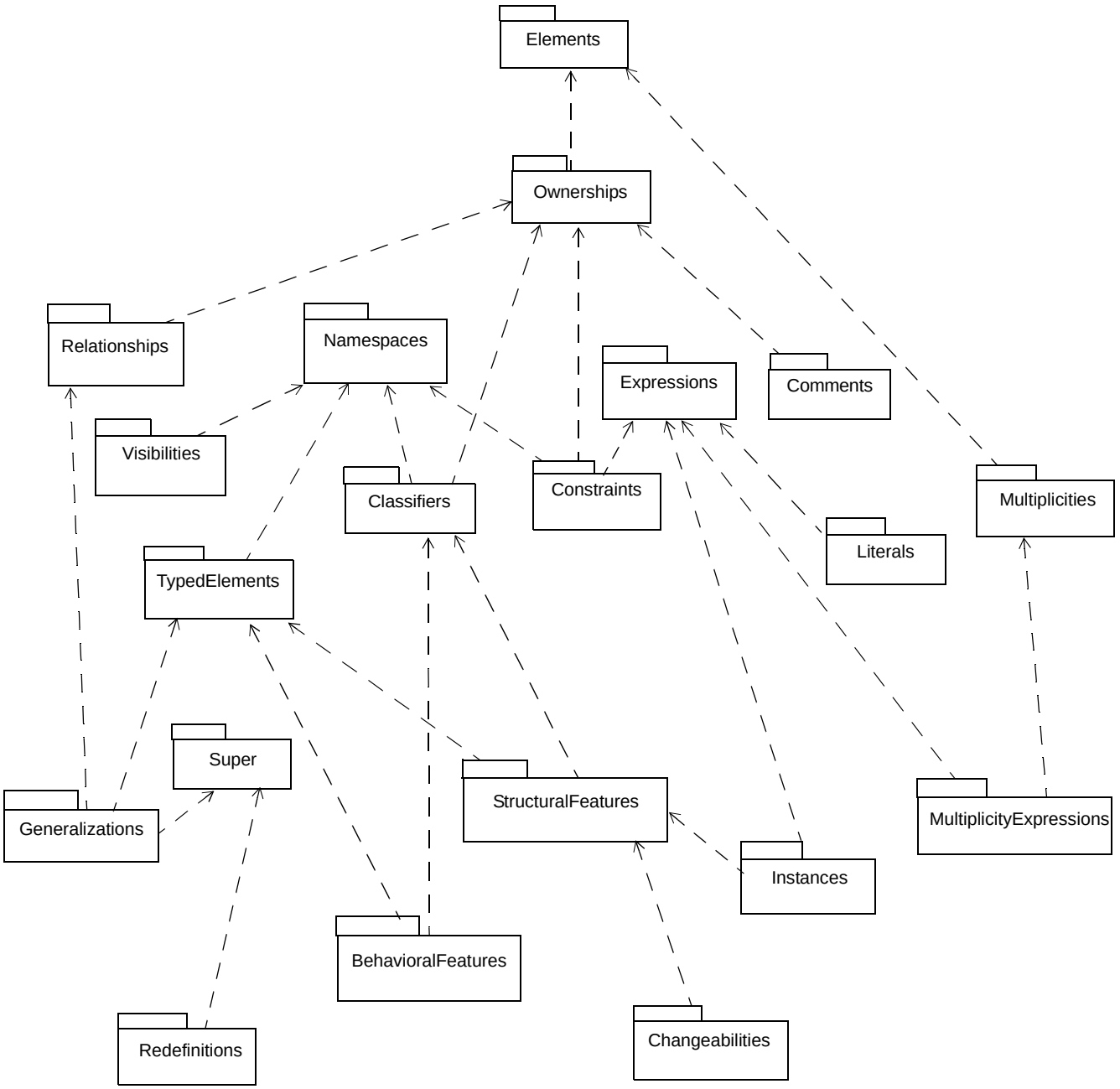


Figure 9.2 - The Abstractions package contains several subpackages, all of which are specified in this clause

The contents of each subpackage of Abstractions is described in a separate sub clause below.

9.1 BehavioralFeatures Package

The BehavioralFeatures subpackage of the Abstractions package specifies the basic classes for modeling dynamic features of model elements.

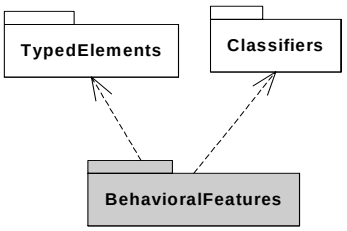


Figure 9.3 - The BehavioralFeatures package

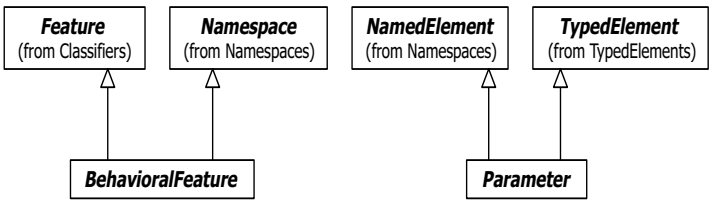


Figure 9.4 - The elements defined in the BehavioralFeatures package

9.1.1 BehavioralFeature

A behavioral feature is a feature of a classifier that specifies an aspect of the behavior of its instances.

Description

A behavioral feature is a feature of a classifier that specifies an aspect of the behavior of its instances. BehavioralFeature is an abstract metaclass specializing *Feature* and *Namespace*. Kinds of behavioral aspects are modeled by subclasses of BehavioralFeature.

Generalizations

- “Feature” on page 36
- “Namespace” on page 73

Attributes

No additional attributes

Constraints

No additional constraints

Additional Operations

[1] The query isDistinguishableFrom() determines whether two BehavioralFeatures may coexist in the same Namespace. It specifies that they have to have different signatures.

```
BehavioralFeature::isDistinguishableFrom(n: NamedElement, ns: Namespace): Boolean;
isDistinguishableFrom =
    if n.oclIsKindOf(BehavioralFeature)
    then
        if ns.getNamesOfMember(self)->intersection(ns.getNamesOfMember(n))->notEmpty()
        then Set{}->including(self)->including(n)->isUnique( bf | bf.parameter->collect(type))
        else true
        endif
    else true
    endif
```

Semantics

The list of parameters describes the order and type of arguments that can be given when the BehavioralFeature is invoked.

Notation

No additional notation

9.2 Parameter

A parameter is a specification of an argument used to pass information into or out of an invocation of a behavioral feature.

Description

Parameter is an abstract metaclass specializing *TypedElement* and *NamedElement*.

Generalizations

- “TypedElement” on page 87
- “NamedElement” on page 71

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

A parameter specifies arguments that are passed into or out of an invocation of a behavioral element like an operation. A parameter’s type restricts what values can be passed.

A parameter may be given a name, which then identifies the parameter uniquely within the parameters of the same behavioral feature. If it is unnamed, it is distinguished only by its position in the ordered list of parameters.

Notation

No general notation. Specific subclasses of BehavioralFeature will define the notation for their parameters.

Style Guidelines

A parameter name typically starts with a lowercase letter.

9.3 Changeabilities Package

The Changeabilities subpackage of the Abstractions package defines when a structural feature may be modified by a client.

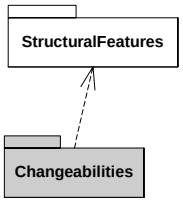


Figure 9.5 - The Changeabilities package

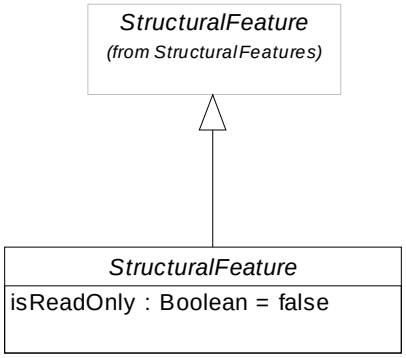


Figure 9.6 - The elements defined in the Changeabilities package

9.3.1 StructuralFeature (as specialized)

Description

StructuralFeature is specialized to add an attribute that determines whether a client may modify its value.

Generalizations

- “StructuralFeature” on page 81

Attributes

- isReadOnly: Boolean — States whether the feature’s value may be modified by a client. Default is false.

Associations

No additional associations

Constraints

No additional constraints

Semantics

No additional semantics

Notation

A read only structural feature is shown using {readOnly} as part of the notation for the structural feature. This annotation may be suppressed, in which case it is not possible to determine its value from the diagram.

Presentation Option

It is possible to only allow suppression of this annotation when isReadOnly=false. In this case it is possible to assume this value in all cases where {readOnly} is not shown.

9.4 Classifiers Package

The Classifiers package in the Abstractions package specifies an abstract generalization for the classification of instances according to their features.

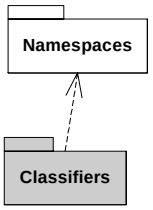


Figure 9.7 - The Classifiers package

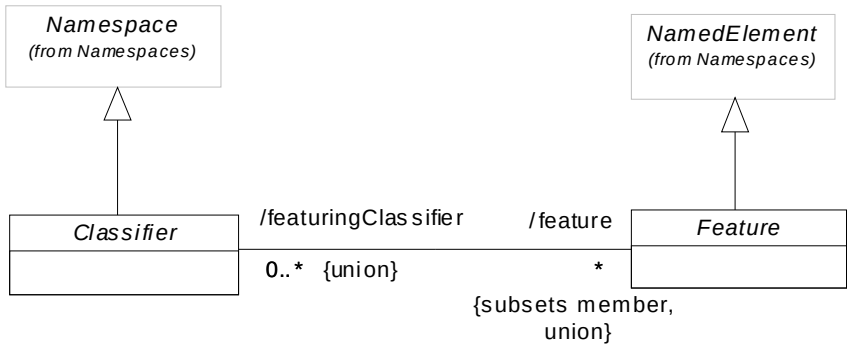


Figure 9.8 - The elements defined in the Classifiers package

9.4.1 Classifier

A classifier is a classification of instances — it describes a set of instances that have features in common.

Description

A classifier is a namespace whose members can include features. Classifier is an abstract metaclass.

Generalizations

- “Namespace” on page 73

Attributes

No additional attributes

Associations

- / feature : Feature [*]
  Specifies each feature defined in the classifier. Subsets *Namespace::member.* This is a derived union.

Additional Operations

[1] The query allFeatures() gives all of the features in the namespace of the classifier. In general, through mechanisms such as inheritance, this will be a larger set than feature.

```
Classifier::allFeatures(): Set(Feature);
allFeatures = member->select(oclIsKindOf(Feature))
```

Constraints

No additional constraints

Semantics

A classifier is a classification of instances according to their features.

Notation

The default notation for a classifier is a solid-outline rectangle containing the classifier's name, and optionally with compartments separated by horizontal lines containing features or other members of the classifier. The specific type of classifier can be shown in guillemets above the name. Some specializations of Classifier have their own distinct notations.

Presentation Options

Any compartment may be suppressed. A separator line is not drawn for a suppressed compartment. If a compartment is suppressed, no inference can be drawn about the presence or absence of elements in it. Compartment names can be used to remove ambiguity, if necessary.

9.4.2 Feature

A feature declares a behavioral or structural characteristic of instances of classifiers.

Description

A feature declares a behavioral or structural characteristic of instances of classifiers. Feature is an abstract metaclass.

Generalizations

- "NamedElement" on page 71

Attributes

No additional attributes

Associations

- / featuringClassifier: Classifier [0..*]
  The Classifiers that have this Feature as a feature. This is a derived union.

Constraints

No additional constraints

Semantics

A Feature represents some characteristic for its featuring classifiers. A Feature can be a feature of multiple classifiers.

Notation

No general notation. Subclasses define their specific notation.

9.5 Comments Package

The Comments package of the Abstractions package defines the general capability of attaching comments to any element.

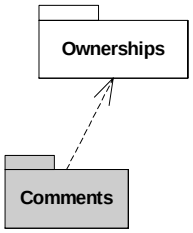


Figure 9.9 - The Comments package

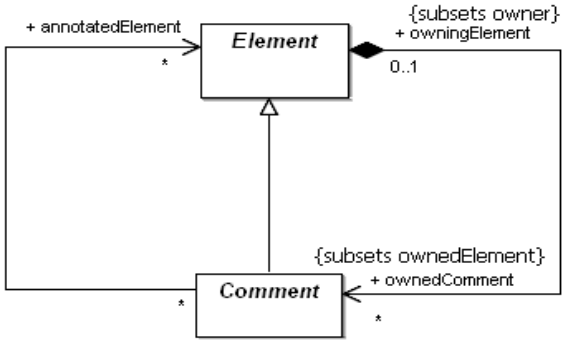


Figure 9.10 - The elements defined in the Comments package

9.5.1 Comment

A comment is a textual annotation that can be attached to a set of elements.

Description

A comment gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler.

A comment may be owned by any element.

Generalizations

None

Attributes

- body: String
  Specifies a string that is the comment.

Associations

- annotatedElement: Element[*]
  References the Element(s) being commented.

Constraints

No additional constraints

Semantics

A Comment adds no semantics to the annotated elements, but may represent information useful to the reader of the model.

Notation

A Comment is shown as a rectangle with the upper right corner bent (this is also known as a “note symbol”). The rectangle contains the body of the Comment. The connection to each annotated element is shown by a separate dashed line.

Presentation Options

The dashed line connecting the note to the annotated element(s) may be suppressed if it is clear from the context, or not important in this diagram.

Examples

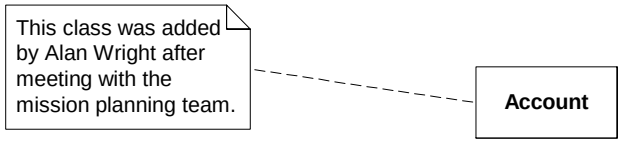


Figure 9.11 - Comment notation

9.5.2 Element

Description

An element can own comments.

Attributes

No additional attributes

Generalizations

- “Element (as specialized)” on page 75

Associations

- ownedComment: Comment[*]
  The Comments owned by this element. Subsets *Element::ownedElement*.

Constraints

No additional constraints

Semantics

The comments for an Element add no semantics but may represent information useful to the reader of the model.

Notation

No additional notation

9.6 Constraints Package

The Constraints subpackage of the Abstractions package specifies the basic building blocks that can be used to add additional semantic information to an element.

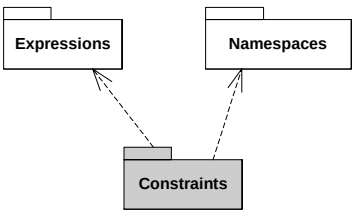


Figure 9.12 - The Constraints package

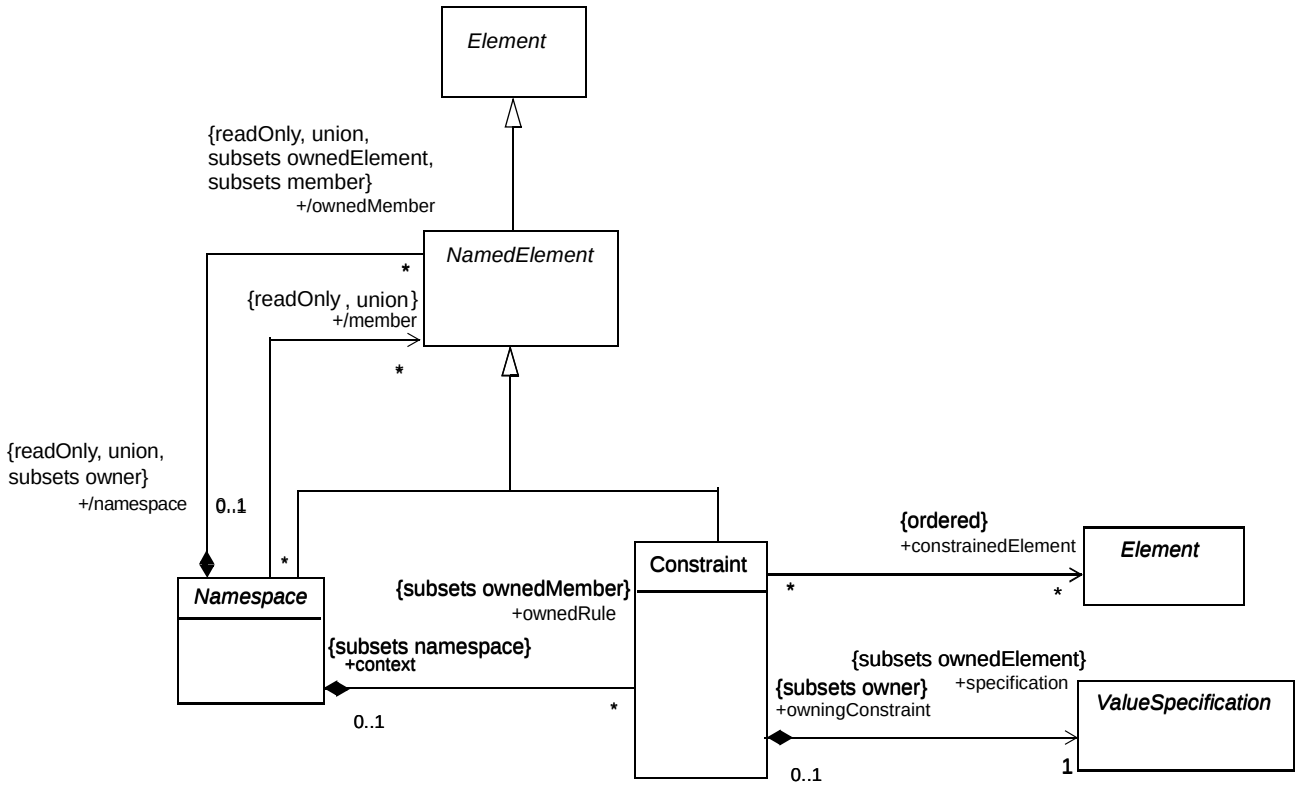


Figure 9.13 - The elements defined in the Constraints package

9.6.1 Constraint

A constraint is a condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element.

Description

Constraint contains a ValueSpecification that specifies additional semantics for one or more elements. Certain kinds of constraints are predefined in UML, others may be user-defined. A user-defined Constraint is described using a specified language, whose syntax and interpretation is a tool responsibility. One predefined language for writing constraints is OCL. In some situations, a programming language such as Java may be appropriate for expressing a constraint. In other situations natural language may be used.

Constraint is a condition (a Boolean expression) that restricts the extension of the associated element beyond what is imposed by the other language constructs applied to the element.

Constraint contains an optional name, although they are commonly unnamed.

Generalizations

- “NamedElement” on page 71

Attributes

No additional attributes

Associations

- constrainedElement: Element[*]
  The ordered set of Elements referenced by this Constraint.
- context: Namespace [0..1]
  Specifies the Namespace that is the context for evaluating this constraint. Subsets *NamedElement::namespace*.
- specification: ValueSpecification[1]
  A condition that must be true when evaluated in order for the constraint to be satisfied. Subsets *Element::ownedElement*.

Constraints

[1] The value specification for a constraint must evaluate to a Boolean value.

Cannot be expressed in OCL.

[2] Evaluating the value specification for a constraint must not have side effects.

Cannot be expressed in OCL.

[3] A constraint cannot be applied to itself.

not constrainedElement->includes(self)

Semantics

A Constraint represents additional semantic information attached to the constrained elements. A constraint is an assertion that indicates a restriction that must be satisfied by a correct design of the system. The constrained elements are those elements required to evaluate the constraint specification. In addition, the context of the Constraint may be accessed, and may be used as the namespace for interpreting names used in the specification. For example, in OCL ‘self’ is used to refer to the context element.

Constraints are often expressed as a text string in some language. If a formal language such as OCL is used, then tools may be able to verify some aspects of the constraints.

In general there are many possible kinds of owners for a Constraint. The only restriction is that the owning element must have access to the constrainedElements.

The owner of the Constraint will determine when the constraint specification is evaluated. For example, this allows an Operation to specify if a Constraint represents a precondition or a postcondition.

Notation

A Constraint is shown as a text string in braces ({}) according to the following BNF:

constraint ::= ‘{‘ [<name> ‘:’] <Boolean expression>’ }’

For an element whose notation is a text string (such as an attribute, etc.) the constraint string may follow the element text string in braces. Figure 9.14 shows a constraint string that follows an attribute within a class symbol.

For a Constraint that applies to a single element (such as a class or an association path), the constraint string may be placed near the symbol for the element, preferably near the name, if any. A tool must make it possible to determine the constrained element.

For a Constraint that applies to two elements (such as two classes or two associations), the constraint may be shown as a dashed line between the elements labeled by the constraint string (in braces). Figure 9.15 shows an {xor} constraint between two associations.

Presentation Options

The constraint string may be placed in a note symbol and attached to each of the symbols for the constrained elements by a dashed line. Figure 9.16 shows an example of a constraint in a note symbol.

If the constraint is shown as a dashed line between two elements, then an arrowhead may be placed on one end. The direction of the arrow is relevant information within the constraint. The element at the tail of the arrow is mapped to the first position and the element at the head of the arrow is mapped to the second position in the constrainedElements collection.

For three or more paths of the same kind (such as generalization paths or association paths), the constraint may be attached to a dashed line crossing all of the paths.

Examples

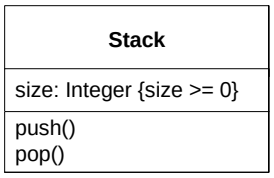


Figure 9.14 - Constraint attached to an attribute

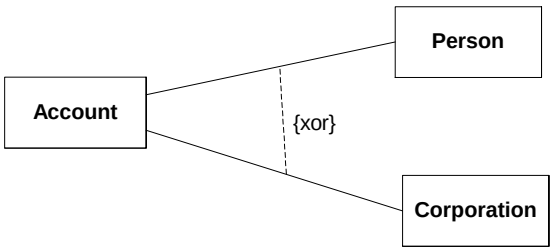


Figure 9.15 - {xor} constraint

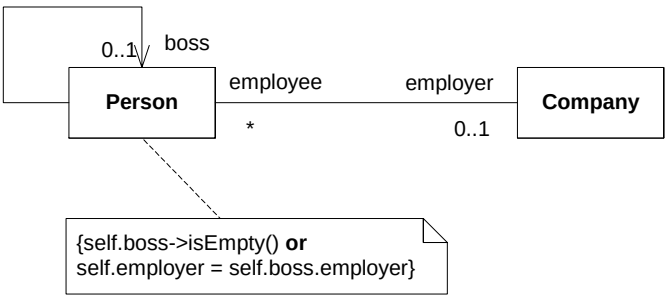


Figure 9.16 - Constraint in a note symbol

9.6.2 Namespace (as specialized)

Description

A namespace can own constraints. A constraint associated with a namespace may either apply to the namespace itself, or it may apply to elements in the namespace.

Generalizations

- “Namespace” on page 73

Attributes

No additional attributes

Associations

- ownedRule: Constraint[*]
  Specifies a set of Constraints owned by this Namespace. Subsets *Namespace::ownedMember.*

Constraints

No additional constraints

Semantics

The ownedRule constraints for a Namespace represent well-formedness rules for the constrained elements. These constraints are evaluated when determining if the model elements are well-formed.

Notation

No additional notation

9.7 Elements Package

The Elements subpackage of the Abstractions package specifies the most basic abstract construct, Element.

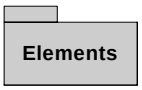


Figure 9.17 - The Elements package

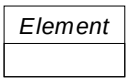


Figure 9.18 - The elements defined in the Elements package

9.7.1 Element

An element is a constituent of a model.

Description

Element is an abstract metaclass with no superclass. It is used as the common superclass for all metaclasses in the infrastructure library.

Generalizations

- None

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Subclasses of Element provide semantics appropriate to the concept they represent.

Notation

There is no general notation for an Element. The specific subclasses of Element define their own notation.

9.8 Expressions Package

The Expressions package in the Abstractions package specifies the general metaclass supporting the specification of values, along with specializations for supporting structured expression trees and opaque, or uninterpreted, expressions. Various UML constructs require or use expressions, which are linguistic formulas that yield values when evaluated in a context.

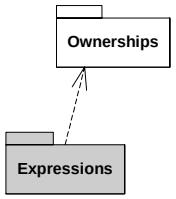


Figure 9.19 - The Expressions package

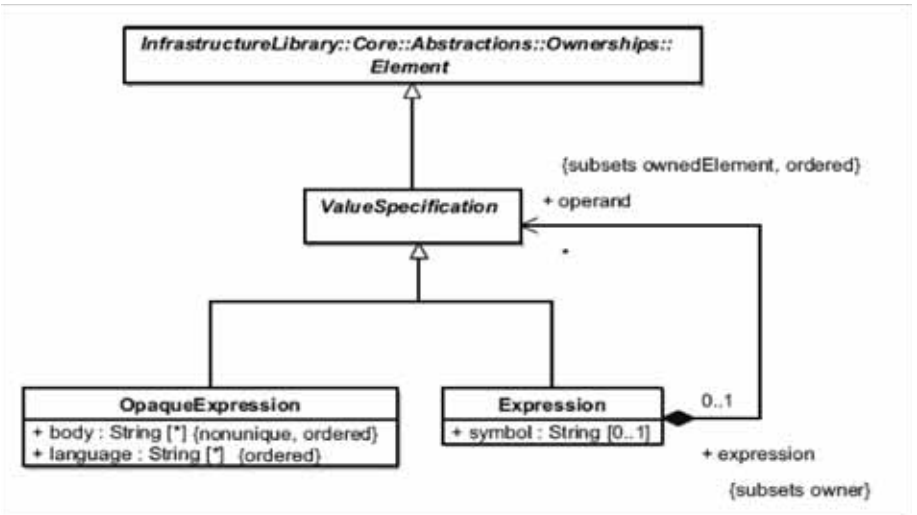


Figure 9.20 - The elements defined in the Expressions package

9.8.1 Expression

An expression is a structured tree of symbols that denotes a (possibly empty) set of values when evaluated in a context.

Description

An expression represents a node in an expression tree, which may be non-terminal or terminal. It defines a symbol, and has a possibly empty sequence of operands that are value specifications.

- “ValueSpecification” on page 47

Attributes

- symbol: String [1]
  The symbol associated with the node in the expression tree.

Associations

- operand: ValueSpecification[*]
  Specifies a sequence of operands. Subsets *Element::ownedElement*.

Constraints

No additional constraints

Semantics

An expression represents a node in an expression tree. If there is no operand, it represents a terminal node. If there are operands, it represents an operator applied to those operands. In either case there is a symbol associated with the node. The interpretation of this symbol depends on the context of the expression.

Notation

By default an expression with no operands is notated simply by its symbol, with no quotes. An expression with operands is notated by its symbol, followed by round parentheses containing its operands in order. In particular contexts special notations may be permitted, including infix operators.

Examples

xor
else
plus(x,1)
x+1

9.8.2 OpaqueExpression

An opaque expression is an uninterpreted textual statement that denotes a (possibly empty) set of values when evaluated in a context.

Description

An opaque expression contains language-specific text strings used to describe a value or values, and an optional specification of the languages.

One predefined language for specifying expressions is OCL. Natural language or programming languages may also be used.

Generalizations

- “ValueSpecification” on page 47

Attributes

- body: String [0..*] {nonunique, ordered}
  The text of the expression, possibly in multiple languages.

- language: String [0..*] {ordered}
  Specifies the languages in which the expression is stated. The interpretation of the expression body depends on the language. If languages are unspecified, it might be implicit from the expression body or the context. Languages are matched to body strings by order.

Associations

No additional associations

Constraints

[1] If the language attribute is not empty, then the size of the body and language arrays must be the same.

```
language->notEmpty() implies
        (body->size() = language->size())
```

Semantics

The expression body may consist of a sequence of text strings – each in a different language – representing alternative representations of the same content. When multiple language strings are provided, the language of each separate string is determined by its corresponding entry in the “language” attribute (by sequence order). The interpretation of the text strings is language specific. Languages are matched to body strings by order. If the languages are unspecified, it might be implicit from the expression bodies or the context.

It is assumed that a linguistic analyzer for the specified languages will evaluate the bodies. The time at which the bodies will be evaluated is not specified.

Notation

An opaque expression is displayed as text string in particular languages. The syntax of the strings are the responsibility of a tool and linguistic analyzers for the language.

An opaque expression is displayed as a part of the notation for its containing element.

The languages of an opaque expression, if specified, are often not shown on a diagram. Some modeling tools may impose a particular language or assume a particular default language. The language is often implicit under the assumption that the form of the expression makes its purpose clear. If the language name is shown, it should be displayed in braces ({}) before the expression string to which it corresponds.

Style Guidelines

A language name should be spelled and capitalized exactly as it appears in the document defining the language. For example, use OCL, not ocl.

Examples

a > 0
{OCL} i > j and self.size > i
average hours worked per week

9.8.3 ValueSpecification

A value specification is the specification of a (possibly empty) set of instances, including both objects and data values.

Description

ValueSpecification is an abstract metaclass used to identify a value or values in a model. It may reference an instance or it may be an expression denoting an instance or instances when evaluated.

Generalizations

- “Element (as specialized)” on page 75

Attributes

No additional attributes

Associations

- expression: Expression[0..1]
  If this value specification is an operand, the owning expression. Subsets *Element::owner.*

Constraints

No additional constraints

Additional Operations

These operations are introduced here. They are expected to be redefined in subclasses. Conforming implementations may be able to compute values for more expressions that are specified by the constraints that involve these operations.

[1] The query isComputable() determines whether a value specification can be computed in a model. This operation cannot be fully defined in OCL. A conforming implementation is expected to deliver true for this operation for all value specifications that it can compute, and to compute all of those for which the operation is true. A conforming implementation is expected to be able to compute the value of all literals.

```
ValueSpecification::isComputable(): Boolean;
isComputable = false
```

[2] The query integerValue() gives a single Integer value when one can be computed.

```
ValueSpecification::integerValue() : [Integer];
integerValue = Set{}
```

[3] The query realValue() gives a single Real value when one can be computed.

```
ValueSpecification::realValue() : [Real];
realValue = Set{}
```

[4] The query booleanValue() gives a single Boolean value when one can be computed.

```
ValueSpecification::booleanValue() : [Boolean];
booleanValue = Set{}
```

[5] The query stringValue() gives a single String value when one can be computed.

```
ValueSpecification::stringValue() : [String];
stringValue = Set{}
```

[6] The query unlimitedValue() gives a single UnlimitedNatural value when one can be computed.

```
ValueSpecification::unlimitedValue() : [UnlimitedNatural];
unlimitedValue = Set{}
```

[7] The query isNull() returns true when it can be computed that the value is null.

```
ValueSpecification::isNull() : Boolean;
isNull = false
```

Semantics

A value specification yields zero or more values. It is required that the type and number of values is suitable for the context where the value specification is used.

Notation

No specific notation

9.9 Generalizations Package

The Generalizations package of the Abstractions package provides mechanisms for specifying generalization relationships between classifiers.

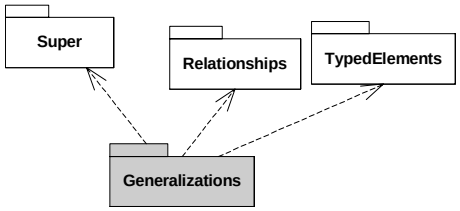


Figure 9.21 - The Generalizations package

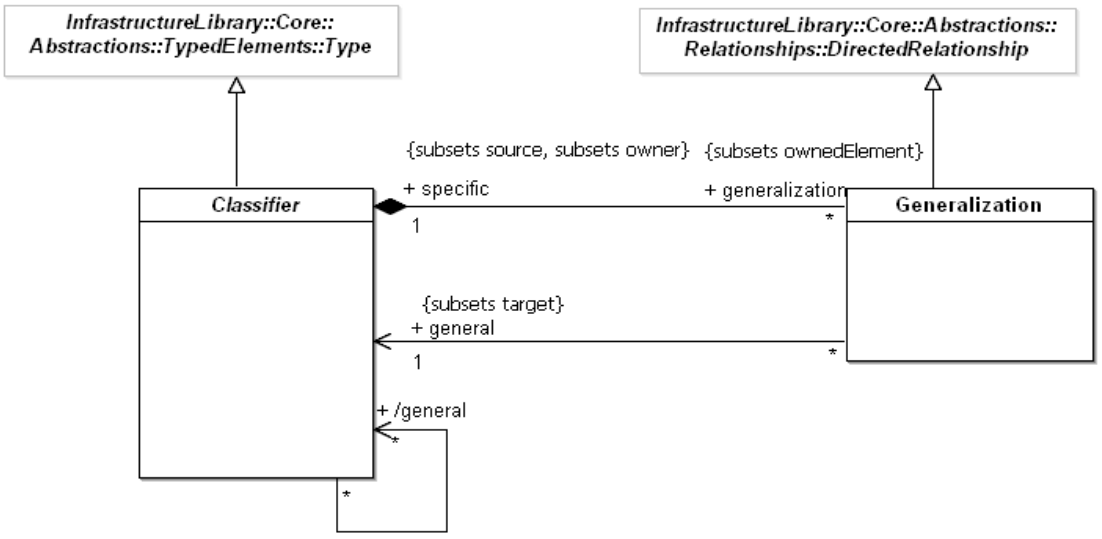


Figure 9.22 - The elements defined in the Generalizations package

9.9.1 Classifier (as specialized)

Description

A classifier is a type and can own generalizations, thereby making it possible to define generalization relationships to other classifiers.

Attributes

No additional attributes

Generalizations

- “Type” on page 86
- “Classifier (as specialized)” on page 83

Associations

- generalization: Generalization[*]
  Specifies the Generalization relationships for this Classifier. These Generalizations navigate to more general classifiers in the generalization hierarchy. Subsets *Element::ownedElement*.
- / general : Classifier[*]
  Specifies the general Classifiers for this Classifier. This is derived.

Constraints

[1] The general classifiers are the classifiers referenced by the generalization relationships.

```
general = self.parents()
```

Additional Operations

[1] The query parents() gives all of the immediate ancestors of a generalized Classifier.

```
Classifier::parents(): Set(Classifier);
parents = generalization.general
```

[2] The query conformsTo() gives true for a classifier that defines a type that conforms to another. This is used, for example, in the specification of signature conformance for operations.

```
Classifier::conformsTo(other: Classifier): Boolean;
conformsTo = (self=other) or (self.allParents()->includes(other))
```

Semantics

A Classifier may participate in generalization relationships with other Classifiers. An instance of a specific Classifier is also an (indirect) instance of the general Classifier. The specific semantics of how generalization affects each concrete subtype of Classifier varies. A Classifier defines a type. Type conformance between generalizable Classifiers is defined so that a Classifier conforms to itself and to all of its ancestors in the generalization hierarchy.

Notation

No additional notation

Examples

See Generalization

9.9.2 Generalization

A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an instance of the general classifier. Thus, the specific classifier indirectly has features of the more general classifier.

Description

A generalization relates a specific classifier to a more general classifier, and is owned by the specific classifier.

Generalizations

- “DirectedRelationship” on page 79

Attributes

No additional attributes

Associations

- general: Classifier [1]
  References the general classifier in the Generalization relationship. Subsets *DirectedRelationship::target*.
- specific: Classifier [1]
  References the specializing classifier in the Generalization relationship. Subsets *DirectedRelationship::source* and *Element::owner*.

Constraints

No additional constraints

Semantics

Where a generalization relates a specific classifier to a general classifier, each instance of the specific classifier is also an instance of the general classifier. Therefore, features specified for instances of the general classifier are implicitly specified for instances of the specific classifier. Any constraint applying to instances of the general classifier also applies to instances of the specific classifier.

Notation

A Generalization is shown as a line with a hollow triangle as an arrowhead between the symbols representing the involved classifiers. The arrowhead points to the symbol representing the general classifier. This notation is referred to as the “separate target style.” See the example sub clause below.

Presentation Options

Multiple Generalization relationships that reference the same general classifier can be connected together in the “shared target style.” See the example sub clause below.

Examples

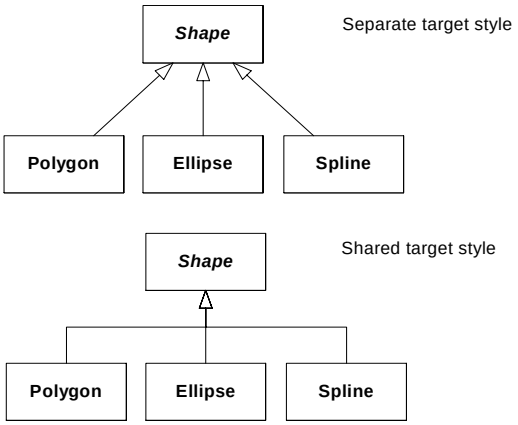


Figure 9.23 - Examples of generalizations between classes

9.10 Instances Package

The Instances package in the Abstractions package provides for modeling instances of classifiers.

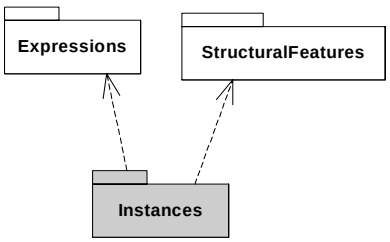


Figure 9.24 - The Instances package

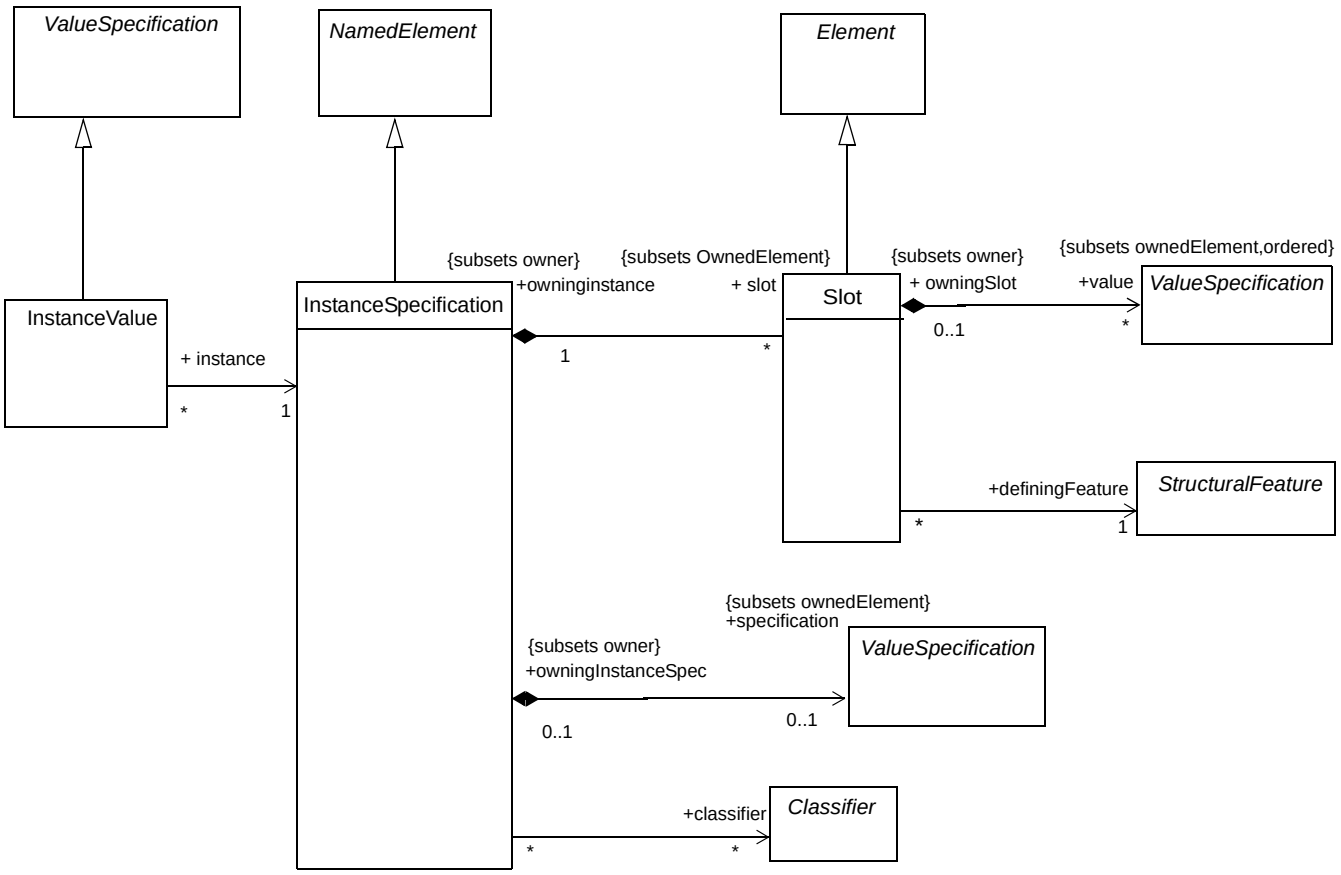


Figure 9.25 - The elements defined in the Instances package

9.10.1 InstanceSpecification

An instance specification is a model element that represents an instance in a modeled system.

Description

An instance specification specifies existence of an entity in a modeled system and completely or partially describes the entity. The description includes:

- Classification of the entity by one or more classifiers of which the entity is an instance. If the only classifier specified is abstract, then the instance specification only partially describes the entity.
- The kind of instance, based on its classifier or classifiers. For example, an instance specification whose classifier is a class describes an object of that class, while an instance specification whose classifier is an association describes a link of that association.

- Specification of values of structural features of the entity. Not all structural features of all classifiers of the instance specification need be represented by slots, in which case the instance specification is a partial description.
- Specification of how to compute, derive, or construct the instance (optional).

InstanceSpecification is a concrete class.

Generalizations

- “NamedElement” on page 71

Attributes

No additional attributes

Associations

- classifier : Classifier [0..*]
  The classifier or classifiers of the represented instance. If multiple classifiers are specified, the instance is classified by all of them.
- slot : Slot [*]
  A slot giving the value or values of a structural feature of the instance. An instance specification can have one slot per structural feature of its classifiers, including inherited features. It is not necessary to model a slot for each structural feature, in which case the instance specification is a partial description. Subsets *Element::ownedElement*
- specification : ValueSpecification [0..1]
  A specification of how to compute, derive, or construct the instance. Subsets *Element::ownedElement*

Constraints

[1] The defining feature of each slot is a structural feature (directly or inherited) of a classifier of the instance specification.

```
slot->forAll(s |
    classifier->exists(c | c.allFeatures()->includes(s.definingFeature))
    )
```

[2] One structural feature (including the same feature inherited from multiple classifiers) is the defining feature of at most one slot in an instance specification.

```
classifier->forAll(c |
    (c.allFeatures()->forAll(f | slot->select(s | s.definingFeature = f)->size() <= 1)
    )
```

Semantics

An instance specification may specify the existence of an entity in a modeled system. An instance specification may provide an illustration or example of a possible entity in a modeled system. An instance specification describes the entity. These details can be incomplete. The purpose of an instance specification is to show what is of interest about an entity in the modeled system. The entity conforms to the specification of each classifier of the instance specification, and has features with values indicated by each slot of the instance specification. Having no slot in an instance specification for some feature does not mean that the represented entity does not have the feature, but merely that the feature is not of interest in the model.

An instance specification can represent an entity at a point in time (a snapshot). Changes to the entity can be modeled using multiple instance specifications, one for each snapshot.

It is important to keep in mind that InstanceSpecification is a model element and should not be confused with the dynamic element that it is modeling. Therefore, one should not expect the dynamic semantics of InstanceSpecification model elements in a model repository to conform to the semantics of the dynamic elements that they represent.

When used to provide an illustration or example of an entity in a modeled system, an InstanceSpecification class does not depict a precise run-time structure. Instead, it describes information about such structures. No conclusions can be drawn about the implementation detail of run-time structure. When used to specify the existence of an entity in a modeled system, an instance specification represents part of that system. Instance specifications can be modeled incompletely, required structural features can be omitted, and classifiers of an instance specification can be abstract, even though an actual entity would have a concrete classification.

Notation

An instance specification is depicted using the same notation as its classifier, but in place of the classifier name appears an underlined concatenation of the instance name, a colon (‘:’), and the classifier name or names. The convention for showing multiple classifiers is to separate their names by commas.

Names are optional for UML 2 classifiers and instance specifications. The absence of a name in a diagram may reflect its absence in the underlying model.

The standard notation for an anonymous instance specification of an unnamed classifier is an underlined colon (' :').

If an instance specification has a value specification as its specification, the value specification is shown either after an equal sign (“=”) following the name, or without an equal sign below the name. If the instance specification is shown using an enclosing shape (such as a rectangle) that contains the name, the value specification is shown within the enclosing shape.

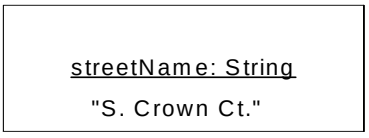


Figure 9.26 - Specification of an instance of String

Slots are shown using similar notation to that of the corresponding structural features. Where a feature would be shown textually in a compartment, a slot for that feature can be shown textually as a feature name followed by an equal sign (‘=’) and a value specification. Other properties of the feature, such as its type, optionally can be shown.

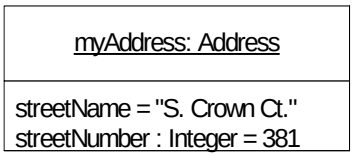


Figure 9.27 - Slots with values

An instance specification whose classifier is an association represents a link and is shown using the same notation as for an association, but the solid path or paths connect instance specifications rather than classifiers. It is not necessary to show an underlined name where it is clear from its connection to instance specifications that it represents a link and not an association. End names can adorn the ends. Navigation arrows can be shown, but if shown, they must agree with the navigation of the association ends.

Figure 9.28 - Instance specifications representing two objects connected by a link

Presentation Options

A slot value for an attribute can be shown using a notation similar to that for a link. A solid path runs from the owning instance specification to the target instance specification representing the slot value, and the name of the attribute adorns the target end of the path. Navigability, if shown, must be only in the direction of the target.

9.10.2 InstanceValue

An instance value is a value specification that identifies an instance.

Description

An instance value specifies the value modeled by an instance specification.

Generalizations

- “ValueSpecification” on page 47

Attributes

No additional attributes

Associations

- instance: InstanceSpecification [1]
  The instance that is the specified value.

Constraints

No additional constraints

Semantics

The instance specification is the specified value.

Notation

An instance value can appear using textual or graphical notation. When textual, as can appear for the value of an attribute slot, the name of the instance is shown. When graphical, a reference value is shown by connecting to the instance (see “InstanceSpecification”).

9.10.3 Slot

A slot specifies that an entity modeled by an instance specification has a value or values for a specific structural feature.

Description

A slot is owned by an instance specification. It specifies the value or values for its defining feature, which must be a structural feature of a classifier of the instance specification owning the slot.

Generalizations

- “Element (as specialized)” on page 75

Attributes

No additional attributes

Associations

- definingFeature : StructuralFeature [1]
  The structural feature that specifies the values that may be held by the slot.
- owningInstance : InstanceSpecification [1]
  The instance specification that owns this slot. Subsets *Element.owner*
- value : ValueSpecification [*]
  The value or values corresponding to the defining feature for the owning instance specification. This is an ordered association. Subsets *Element.ownedElement*

Constraints

No additional constraints

Semantics

A slot relates an instance specification, a structural feature, and a value or values. It represents that an entity modeled by the instance specification has a structural feature with the specified value or values. The values in a slot must conform to the defining feature of the slot (in type, multiplicity, etc.).

Notation

See “InstanceSpecification”

9.11 Literals Package

The Literals package in the Abstractions package specifies metaclasses for specifying literal values.

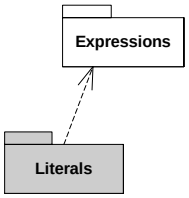


Figure 9.29 - The Literals package

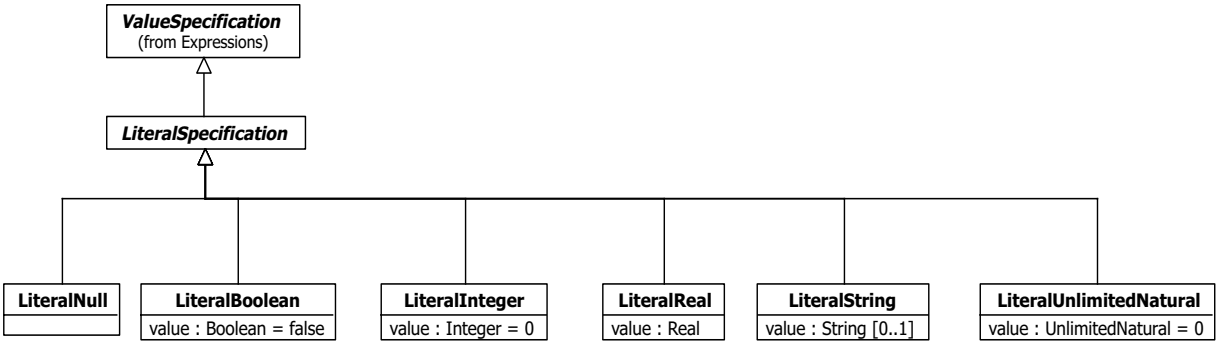


Figure 9.30 - The elements defined in the Literals package

9.11.1 LiteralBoolean

A literal Boolean is a specification of a Boolean value.

Description

A literal Boolean contains a Boolean-valued attribute.

Generalizations

- “LiteralSpecification” on page 62

Attributes

- value: Boolean
  The specified Boolean value. Redefines *ValueSpecification::value*.

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

[1] The query isComputable() is redefined to be true.

```
LiteralBoolean::isComputable(): Boolean;
isComputable = true
```

[2] The query booleanValue() gives the value.

```
LiteralBoolean::booleanValue() : [Boolean];
booleanValue = value
```

Semantics

A LiteralBoolean specifies a constant Boolean value.

Notation

A LiteralBoolean is shown as either the word ‘true’ or the word ‘false,’ corresponding to its value.

9.11.2 LiteralInteger

A literal integer is a specification of an integer value.

Description

A literal integer contains an Integer-valued attribute.

Generalizations

- “LiteralSpecification” on page 62

Attributes

- value: Integer
  The specified Integer value. Redefines *ValueSpecification::value*

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

[1] The query isComputable() is redefined to be true.

```
LiteralInteger::isComputable(): Boolean;
```

isComputable = true

[2] The query integerValue() gives the value.

LiteralInteger::integerValue() : [Integer];
integerValue = value

Semantics

A LiteralInteger specifies a constant Integer value.

Notation

A LiteralInteger is typically shown as a sequence of digits.

9.11.3 LiteralNull

A literal null specifies the lack of a value.

Description

A literal null is used to represent null (i.e., the absence of a value).

Generalizations

- “LiteralSpecification” on page 62

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

[1] The query isComputable() is redefined to be true.

LiteralNull::isComputable(): Boolean;
isComputable = true

[2] The query isNull() returns true.

LiteralNull::isNull() : Boolean;
isNull = true

Semantics

LiteralNull is intended to be used to explicitly model the lack of a value.

Notation

Notation for LiteralNull varies depending on where it is used. It often appears as the word ‘null.’ Other notations are described for specific uses.

9.11.4 LiteralReal

A literal real is a specification of a real value.

Description

A literal real contains a Real-valued attribute.

Generalizations

- “LiteralSpecification” on page 62

Attributes

- value: Real
  The specified Real value. Redefines *ValueSpecification::value*

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

[1] The query isComputable() is redefined to be true.

```
LiteralReal::isComputable(): Boolean;
isComputable = true
```

[2] The query realValue() gives the value.

```
LiteralString::realValue() : [Real];
realValue = value
```

Semantics

A LiteralReal specifies a constant Real value.

Notation

A LiteralReal is shown in the decimal notation or scientific notation. Decimal notation consists of an optional sign character (+/-) followed by zero or more digits followed optionally by a dot (.) followed by one or more digits. Scientific notation consists of decimal notation followed by either the letter “e” or “E” and an exponent consisting of an optional sign character followed by one or more digits. The scientific notation expresses a real number equal to that given by the decimal notation before the exponent, times 10 raised to the power of the exponent.

This notation is specified by the following BNF rules:

<natural-literal> ::= ('0'..'9')+

<decimal-literal> ::= ['+' | '-'] <natural-literal>

| ['+' | '-'] [<natural-literal>] '.' <natural-literal>

<real-literal> ::= <decimal-literal> [('e' | 'E') ['+' | '-'] <natural-literal>]

9.11.5 LiteralSpecification

A literal specification identifies a literal constant being modeled.

Description

A literal specification is an abstract specialization of ValueSpecification that identifies a literal constant being modeled.

Generalizations

- “ValueSpecification” on page 47

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

No additional semantics. Subclasses of LiteralSpecification are defined to specify literal values of different types.

Notation

No specific notation

9.11.6 LiteralString

A literal string is a specification of a string value.

Description

A literal string contains a String-valued attribute.

Generalizations

- “LiteralSpecification” on page 62

Attributes

- value: String
  The specified String value. Redefines *ValueSpecification::value*.

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

[1] The query isComputable() is redefined to be true.

```
LiteralString::isComputable(): Boolean;
isComputable = true
```

[2] The query stringValue() gives the value.

```
LiteralString::stringValue() : [String];
stringValue = value
```

Semantics

A LiteralString specifies a constant String value.

Notation

A LiteralString is shown as a sequence of characters within double quotes. The character set used is unspecified.

9.11.7 LiteralUnlimitedNatural

A literal unlimited natural is a specification of an unlimited natural number.

Description

A literal unlimited natural contains an UnlimitedNatural-valued attribute.

Generalizations

- “LiteralSpecification” on page 62

Attributes

- value: UnlimitedNatural
  The specified UnlimitedNatural value. Redefines *ValueSpecification::value*

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

[1] The query isComputable() is redefined to be true.

```
LiteralUnlimitedNatural::isComputable(): Boolean;
isComputable = true
```

[2] The query unlimitedValue() gives the value.

```
LiteralUnlimitedNatural::unlimitedValue() : [UnlimitedNatural];
unlimitedValue = value
```

Semantics

A LiteralUnlimitedNatural specifies a constant UnlimitedNatural value.

Notation

A LiteralUnlimitedNatural is shown either as a sequence of digits or as an asterisk (*), where the asterisk denotes unlimited (and not infinity).

9.12 Multiplicities Package

The Multiplicities subpackage of the Abstractions package defines the metamodel classes used to support the specification of multiplicities for typed elements (such as association ends and attributes), and for specifying whether multivalued elements are ordered or unique.

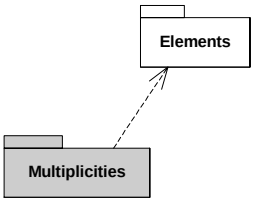


Figure 9.31 - The Multiplicities package

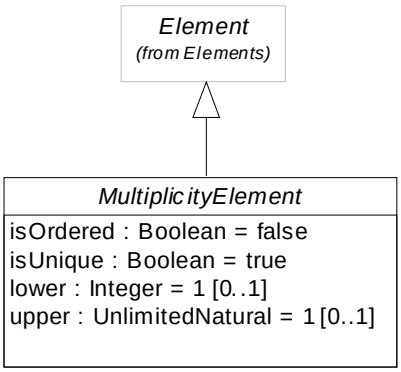


Figure 9.32 - The elements defined in the Multiplicities package

9.12.1 MultiplicityElement

A multiplicity is a definition of an inclusive interval of non-negative integers beginning with a lower bound and ending with a (possibly infinite) upper bound. A multiplicity element embeds this information to specify the allowable cardinalities for an instantiation of this element.

Description

A MultiplicityElement is an abstract metaclass which includes optional attributes for defining the bounds of a multiplicity. A MultiplicityElement also includes specifications of whether the values in an instantiation of this element must be unique or ordered.

Generalizations

- "Element" on page 44

Attributes

- isOrdered: Boolean
  For a multivalued multiplicity, this attribute specifies whether the values in an instantiation of this element are sequentially ordered (Default is *false*).
- isUnique : Boolean
  For a multivalued multiplicity, this attributes specifies whether the values in an instantiation of this element are unique (Default is *true*).
- lower : Integer [0..1]
  Specifies the lower bound of the multiplicity interval (Default is one).
- upper : UnlimitedNatural [0..1]
  Specifies the upper bound of the multiplicity interval (Default is one).

Associations

No additional associations

Constraints

These constraints must handle situations where the upper bound may be specified by an expression not computable in the model. In this package such situations cannot arise but they can in subclasses.

[1] The lower bound must be a non-negative integer literal.

lowerBound()->notEmpty() **implies** lowerBound() >= 0

[2] The upper bound must be greater than or equal to the lower bound.

(upperBound()->notEmpty() **and** lowerBound()->notEmpty()) **implies** upperBound() >= lowerBound()

Additional Operations

[1] The query isMultivalued() checks whether this multiplicity has an upper bound greater than one.
MultiplicityElement::isMultivalued() : Boolean;
pre: upperBound()->notEmpty()
isMultivalued = (upperBound() > 1)

[2] The query includesCardinality() checks whether the specified cardinality is valid for this multiplicity.

MultiplicityElement::includesCardinality(C : Integer) : Boolean;

pre: upperBound()->notEmpty() **and** lowerBound()->notEmpty()
includesCardinality = (lowerBound() <= C) **and** (upperBound() >= C)

[3] The query includesMultiplicity() checks whether this multiplicity includes all the cardinalities allowed by the specified multiplicity.

MultiplicityElement::includesMultiplicity(M : MultiplicityElement) : Boolean;
pre: self.upperBound()->notEmpty() **and** self.lowerBound()->notEmpty()
and M.upperBound()->notEmpty() **and** M.lowerBound()->notEmpty()
includesMultiplicity = (self.lowerBound() <= M.lowerBound()) **and** (self.upperBound() >= M.upperBound())

[4] The query lowerBound() returns the lower bound of the multiplicity as an integer.

MultiplicityElement::lowerBound() : [Integer];
lowerBound = **if** lower->notEmpty() **then** lower **else** 1 **endif**

[5] The query upperBound() returns the upper bound of the multiplicity for a bounded multiplicity as an unlimited natural.

MultiplicityElement::upperBound() : [UnlimitedNatural];
upperBound = **if** upper->notEmpty() **then** upper **else** 1 **endif**

Semantics

A multiplicity defines a set of integers that define valid cardinalities. Specifically, cardinality C is valid for multiplicity M if M.includesCardinality(C).

A multiplicity is specified as an interval of integers starting with the lower bound and ending with the (possibly infinite) upper bound.

If a MultiplicityElement specifies a multivalued multiplicity, then an instantiation of this element has a set of values. The multiplicity is a constraint on the number of values that may validly occur in that set.

If the MultiplicityElement is specified as ordered (i.e., isOrdered is true), then the set of values in an instantiation of this element is ordered. This ordering implies that there is a mapping from positive integers to the elements of the set of values. If a MultiplicityElement is not multivalued, then the value for isOrdered has no semantic effect.

If the MultiplicityElement is specified as unordered (i.e., isOrdered is false), then no assumptions can be made about the order of the values in an instantiation of this element.

If the MultiplicityElement is specified as unique (i.e., isUnique is true), then the set of values in an instantiation of this element must be unique. If a MultiplicityElement is not multivalued, then the value for isUnique has no semantic effect.

The lower and upper bounds for the multiplicity of a MultiplicityElement may be specified by value specifications, such as (side-effect free, constant) expressions. A MultiplicityElement can define a [0..0] multiplicity. This restricts cardinality to be 0; that is, it forces the collection to be empty. This is useful in the context of generalizations - to constrain the cardinalities of a more general classifier. It applies to (but is not limited to) redefining properties existing in more general classifiers.

Notation

The specific notation for a MultiplicityElement is defined by the concrete subclasses. In general, the notation will include a multiplicity specification, which is shown as a text string containing the bounds of the interval, and a notation for showing the optional ordering and uniqueness specifications.

The multiplicity bounds are typically shown in the format:

<lower-bound>'..' *<upper-bound>*

where *<lower-bound>* is a non-negative integer and *<upper-bound>* is an unlimited natural number. The asterisk (*) is used as part of a multiplicity specification to represent the unlimited (or infinite) upper bound.

If the Multiplicity is associated with an element whose notation is a text string (such as an attribute, etc.), the multiplicity string will be placed within square brackets ([]) as part of that text string. Figure 9.33 shows two multiplicity strings as part of attribute specifications within a class symbol.

If the Multiplicity is associated with an element that appears as a symbol (such as an association end), the multiplicity string is displayed without square brackets and may be placed near the symbol for the element. Figure 9.34 shows two multiplicity strings as part of the specification of two association ends.

The specific notation for the ordering and uniqueness specifications may vary depending on the specific subclass of MultiplicityElement. A general notation is to use a property string containing ordered or unordered to define the ordering, and unique or nonunique to define the uniqueness.

Presentation Options

If the lower bound is equal to the upper bound, then an alternate notation is to use the string containing just the upper bound. For example, “1” is semantically equivalent to “1..1.”

A multiplicity with zero as the lower bound and an unspecified upper bound may use the alternative notation containing a single asterisk “*” instead of “0..*”.

The following BNF defines the syntax for a multiplicity string, including support for the presentation options listed above.

<multiplicity> ::= <multiplicity-range>
[[‘{‘ <order-designator> [‘,’ <uniqueness-designator>] ‘}’] |
[‘{‘ <uniqueness-designator> [‘,’ <order-designator>] ‘}’]]
<multiplicity-range> ::= [<lower> ‘..’] <upper>
<lower> ::= <integer> | <value-specification>
<upper> ::= ‘’ | <value-specification>*
<order-designator> ::= ‘ordered’ | ‘unordered’
<uniqueness-designator> ::= ‘unique’ | ‘nonunique’

Examples

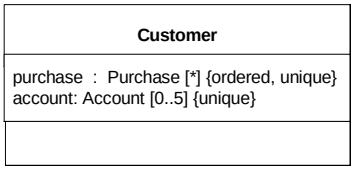


Figure 9.33 - Multiplicity within a textual specification

Figure 9.34 - Multiplicity as an adornment to a symbol

Rationale

MultiplicityElement represents a design trade-off to improve some technology mappings (such as XMI).

9.13 MultiplicityExpressions Package

The MultiplicityExpressions subpackage of the Abstractions package extends the multiplicity capabilities to support the use of value expressions for the bounds.

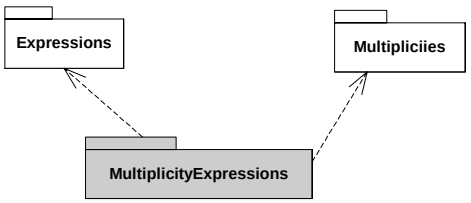


Figure 9.35 - The MultiplicityExpressions package

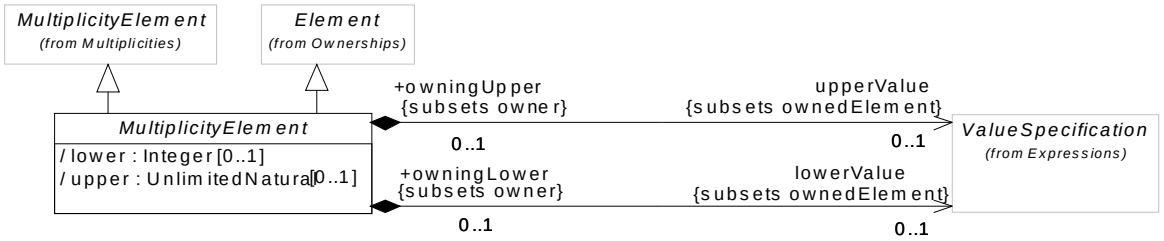


Figure 9.36 - The elements defined in the MultiplicityExpressions package

9.13.1 MultiplicityElement (specialized)

Description

MultiplicityElement is specialized to support the use of value specifications to define each bound of the multiplicity.

Generalizations

- “MultiplicityElement” on page 65
- “Element (as specialized)” on page 75

Attributes

- / lower : Integer [0..1]
  Specifies the lower bound of the multiplicity interval, if it is expressed as an integer. This is a redefinition of the corresponding property from Multiplicities.
- / upper : UnlimitedNatural [0..1]
  Specifies the upper bound of the multiplicity interval, if it is expressed as an unlimited natural. This is a redefinition of the corresponding property from Multiplicities.

Associations

- lowerValue: ValueSpecification [0..1]
  The specification of the lower bound for this multiplicity. Subsets *Element::ownedElement*
- upperValue: ValueSpecification [0..1]
  The specification of the upper bound for this multiplicity. Subsets *Element::ownedElement*

Constraints

[1] If a ValueSpecification is used for the lower or upper bound, then evaluating that specification must not have side effects.
Cannot be expressed in OCL.

[2] If a ValueSpecification is used for the lower or upper bound, then that specification must be a constant expression.
Cannot be expressed in OCL.

[3] The derived lower attribute must equal the lowerBound.
lower = lowerBound()

[4] The derived upper attribute must equal the upperBound.
upper = upperBound()

Additional Operations

[1] The query lowerBound() returns the lower bound of the multiplicity as an integer.

```
MultiplicityElement::lowerBound() : [Integer];
lowerBound =
    if lowerValue->isEmpty() then
        1
    else
        lowerValue.integerValue()
    endif
```

[2] The query upperBound() returns the upper bound of the multiplicity as an unlimited natural.

```
MultiplicityElement::upperBound() : [UnlimitedNatural];
upperBound =
    if upperValue->isEmpty() then
        1
    else
        upperValue.unlimitedValue()
    endif
```

Semantics

The lower and upper bounds for the multiplicity of a MultiplicityElement may be specified by value specifications, such as (side-effect free, constant) expressions.

Notation

The notation for Multiplicities::MultiplicityElement (see page 65) is extended to support value specifications for the bounds.

The following BNF defines the syntax for a multiplicity string, including support for the presentation options.

<multiplicity> ::= <multiplicity-range>
[['{' <order-designator> [',' <uniqueness-designator>] '}'] |
['{' <uniqueness-designator> [',' <order-designator>] '}']]
<multiplicity-range> ::= [<lower> '..'] <upper>
<lower> ::= <integer> | <value-specification>
<upper> ::= '' | <value-specification>*
<order-designator> ::= 'ordered' | 'unordered'
<uniqueness-designator> ::= 'unique' | 'nonunique'

9.14 Namespaces Package

The Namespaces subpackage of the Abstractions package specifies the concepts used for defining model elements that have names, and the containment and identification of these named elements within namespaces.

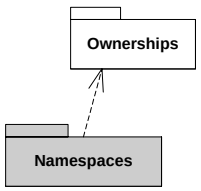


Figure 9.37 - The Namespaces package

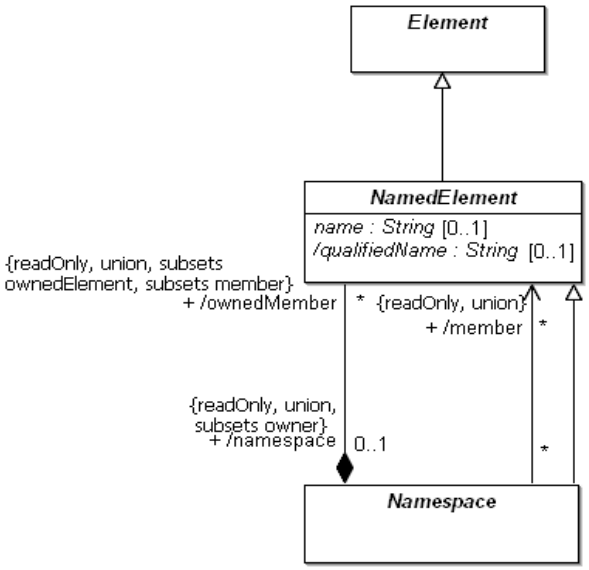


Figure 9.38 - The elements defined in the Namespaces package

9.14.1 NamedElement

A named element is an element in a model that may have a name.

Description

A named element represents elements that may have a name. The name is used for identification of the named element within the namespace in which it is defined. A named element also has a qualified name that allows it to be unambiguously identified within a hierarchy of nested namespaces. NamedElement is an abstract metaclass.

Generalizations

- “Element (as specialized)” on page 75

Attributes

- name: String [0..1]
  The name of the NamedElement.
- / qualifiedName: String [0..1]
  A name which allows the NamedElement to be identified within a hierarchy of nested Namespaces. It is constructed from the names of the containing namespaces starting at the root of the hierarchy and ending with the name of the NamedElement itself. This is a derived attribute.

Associations

- / namespace: Namespace [0..1]
  Specifies the namespace that owns the NamedElement. Subsets *Element::owner.* This is a derived union.

Constraints

[1] If there is no name, or one of the containing namespaces has no name, there is no qualified name.

```
(self.name->isEmpty() or self.allNamespaces()->select(ns | ns.name->isEmpty())->notEmpty())
    implies self.qualifiedName->isEmpty()
```

[2] When there is a name, and all of the containing namespaces have a name, the qualified name is constructed from the names of the containing namespaces.

```
(self.name->notEmpty() and self.allNamespaces()->select(ns | ns.name->isEmpty())->isEmpty()) implies
    self.qualifiedName = self.allNamespaces()->iterate( ns : Namespace; result: String = self.name |
        ns.name->union(self.separator())->union(result))
```

Additional Operations

[1] The query allNamespaces() gives the sequence of namespaces in which the NamedElement is nested, working outwards.

```
NamedElement::allNamespaces(): Sequence(Namespace);
allNamespaces =
  if self.namespace->isEmpty()
  then Sequence{}
  else self.namespace.allNamespaces()->prepend(self.namespace)
  endif
```

[2] The query isDistinguishableFrom() determines whether two NamedElements may logically co-exist within a Namespace. By default, two named elements are distinguishable if (a) they have unrelated types or (b) they have related types but different names.

```
NamedElement::isDistinguishableFrom(n:NamedElement, ns: Namespace): Boolean;
isDistinguishable =
  if self.oclIsKindOf(n.oclType) or n.oclIsKindOf(self.oclType)
  then ns.getNamesOfMember(self)->intersection(ns.getNamesOfMember(n))->isEmpty()
  else true
  endif
```

[3] The query separator() gives the string that is used to separate names when constructing a qualified name.

```
NamedElement::separator(): String;
separator = '::'
```

Semantics

The name attribute is used for identification of the named element within namespaces where its name is accessible. Note that the attribute has a multiplicity of [0..1], which provides for the possibility of the absence of a name (which is different from the empty name).

9.14.2 Namespace

A namespace is an element in a model that contains a set of named elements that can be identified by name.

Description

A namespace is a named element that can own other named elements. Each named element may be owned by at most one namespace. A namespace provides a means for identifying named elements by name. Named elements can be identified by name in a namespace either by being directly owned by the namespace or by being introduced into the namespace by other means (e.g., importing or inheriting). Namespace is an abstract metaclass.

Generalizations

- “Namespace” on page 73

Attributes

No additional attributes

Associations

- / member: NamedElement [*]
  A collection of NamedElements identifiable within the Namespace, either by being owned or by being introduced by importing or inheritance. This is a derived union.
- / ownedMember: NamedElement [*]
  A collection of NamedElements owned by the Namespace. Subsets *Element::ownedElement* and *Namespace::member*. This is a derived union.

Constraints

[1] All the members of a Namespace are distinguishable within it.
membersAreDistinguishable()

Additional Operations

[1] The query getNamesOfMember() gives a set of all of the names that a member would have in a Namespace. In general a member can have multiple names in a Namespace if it is imported more than once with different aliases. Those semantics are specified by overriding the getNamesOfMember operation. The specification here simply returns a set containing a single name, or the empty set if no name.

```
Namespace::getNamesOfMember(element: NamedElement): Set(String);
getNamesOfMember =
    if member->includes(element) then Set{}->including(element.name) else Set{} endif
```

[2] The Boolean query membersAreDistinguishable() determines whether all of the namespace’s members are distinguishable within it.

```
Namespace::membersAreDistinguishable() : Boolean;
membersAreDistinguishable =
self.member->forAll( memb |
```

```
self.member->excluding(memb)->forAll(other |
    memb.isDistinguishableFrom(other, self)))
```

Semantics

A namespace provides a container for named elements. It provides a means for resolving composite names, such as name1::name2::name3. The *member* association identifies all named elements in a namespace called N that can be referred to by a composite name of the form N::<x>. Note that this is different from all of the names that can be referred to unqualified within N, because that set also includes all unhidden members of enclosing namespaces.

Named elements may appear within a namespace according to rules that specify how one named element is distinguishable from another. The default rule is that two elements are distinguishable if they have unrelated types, or related types but different names. This rule may be overridden for particular cases, such as operations that are distinguished by their signature.

Notation

No additional notation. Concrete subclasses will define their own specific notation.

9.15 Ownerships Package

The Ownerships subpackage of the Abstractions package extends the basic element to support ownership of other elements.

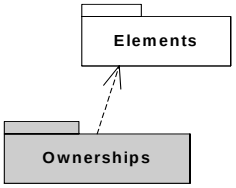


Figure 9.39 - The Ownerships package

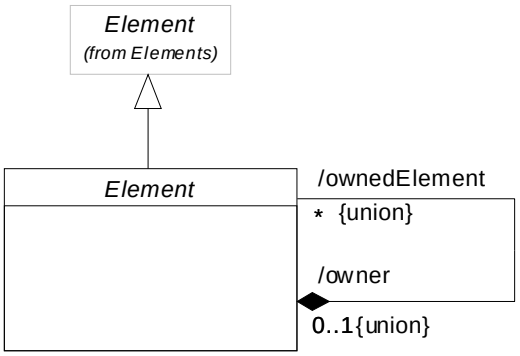


Figure 9.40 - The elements defined in the Ownerships package

9.15.1 Element (as specialized)

An element is a constituent of a model. As such, it has the capability of owning other elements.

Description

Element has a derived composition association to itself to support the general capability for elements to own other elements.

Generalizations

- “Element” on page 44

Attributes

No additional attributes

Associations

- / ownedElement: Element[*]
  The Elements owned by this element. This is a derived union.
- / owner: Element [0..1]
  The Element that owns this element. This is a derived union.

Constraints

[1] An element may not directly or indirectly own itself.
not self.allOwnedElements()->includes(self)

[2] Elements that must be owned must have an owner.
self.mustBeOwned() **implies** owner->notEmpty()

Additional Operations

[1] The query allOwnedElements() gives all of the direct and indirect owned elements of an element.
Element::allOwnedElements(): Set(Element);

allOwnedElements = ownedElement->union(ownedElement->collect(e | e.allOwnedElements()))

[2] The query mustBeOwned() indicates whether elements of this type must have an owner. Subclasses of Element that do not require an owner must override this operation.

Element::mustBeOwned() : Boolean;
mustBeOwned = true

Semantics

Subclasses of Element will provide semantics appropriate to the concept they represent.

The derived *ownedElement* association is subsetted (directly or indirectly) by all composed association ends in the metamodel. Thus ownedElement provides a convenient way to access all the elements that are directly owned by an Element.

Notation

There is no general notation for an Element. The specific subclasses of Element define their own notation.

9.16 Redefinitions Package

The Redefinitions package in the Abstractions package specifies the general capability of redefining model elements in the context of a generalization hierarchy.

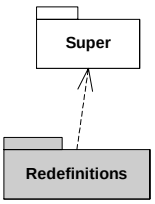


Figure 9.41 - The Redefinitions package

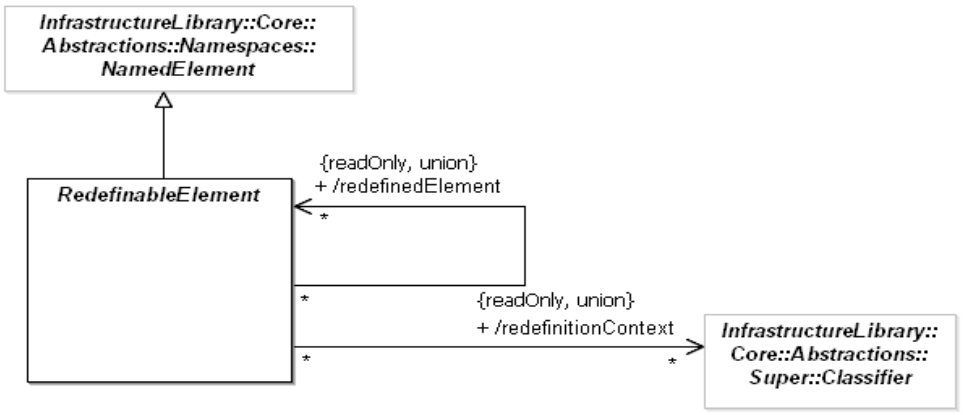


Figure 9.42 - The elements defined in the Redefinitions package

9.16.1 RedefinableElement

A redefinable element is an element that, when defined in the context of a classifier, can be redefined more specifically or differently in the context of another classifier that specializes (directly or indirectly) the context classifier.

Description

A redefinable element is a named element that can be redefined in the context of a generalization. RedefinableElement is an abstract metaclass.

Generalizations

- “NamedElement” on page 71

Attributes

No additional attributes

Associations

- / redefinedElement: RedefinableElement[*]
  The redefinable element that is being redefined by this element. This is a derived union.
- / redefinitionContext: Classifier[*]
  References the contexts that this element may be redefined from. This is a derived union.

Constraints

[1] At least one of the redefinition contexts of the redefining element must be a specialization of at least one of the redefinition contexts for each redefined element.

```
self.redefinedElement->forAll(e | self.isRedefinitionContextValid(e))
```

[2] A redefining element must be consistent with each redefined element.

```
self.redefinedElement->forAll(re | re.isConsistentWith(self))
```

Additional Operations

[1] The query isConsistentWith() specifies, for any two RedefinableElements in a context in which redefinition is possible, whether redefinition would be logically consistent. By default, this is false; this operation must be overridden for subclasses of RedefinableElement to define the consistency conditions.

```
RedefinableElement::isConsistentWith(redefinee: RedefinableElement): Boolean;
pre: redefinee.isRedefinitionContextValid(self)
isConsistentWith = false
```

[2] The query isRedefinitionContextValid() specifies whether the redefinition contexts of this RedefinableElement are properly related to the redefinition contexts of the specified RedefinableElement to allow this element to redefine the other. By default at least one of the redefinition contexts of this element must be a specialization of at least one of the redefinition contexts of the specified element.

```
RedefinableElement::isRedefinitionContexValid(redefinable: RedefinableElement): Boolean;
    isRedefinitionContextValid =
        redefinitionContext->exists(c | c.allparents()->
            includes (redefined.redefinitionContext))
```

Semantics

A RedefinableElement represents the general ability to be redefined in the context of a generalization relationship. The detailed semantics of redefinition varies for each specialization of RedefinableElement.

A redefinable element is a specification concerning instances of a classifier that is one of the element's redefinition contexts. For a classifier that specializes that more general classifier (directly or indirectly), another element can redefine the element from the general classifier in order to augment, constrain, or override the specification as it applies more specifically to instances of the specializing classifier.

A redefining element must be consistent with the element it redefines, but it can add specific constraints or other details that are particular to instances of the specializing redefinition context that do not contradict invariant constraints in the general context.

A redefinable element may be redefined multiple times. Furthermore, one redefining element may redefine multiple inherited redefinable elements.

Semantic Variation Points

There are various degrees of compatibility between the redefined element and the redefining element, such as name compatibility (the redefining element has the same name as the redefined element), structural compatibility (the client visible properties of the redefined element are also properties of the redefining element), or behavioral compatibility (the redefining element is substitutable for the redefined element). Any kind of compatibility involves a constraint on redefinitions. The particular constraint chosen is a semantic variation point.

Notation

No general notation. See the subclasses of RedefinableElement for the specific notation used.

9.17 Relationships Package

The Relationships subpackage of the Abstractions package adds support for directed relationships.

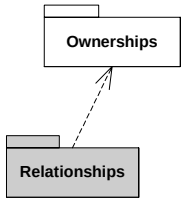


Figure 9.43 - The Relationships package

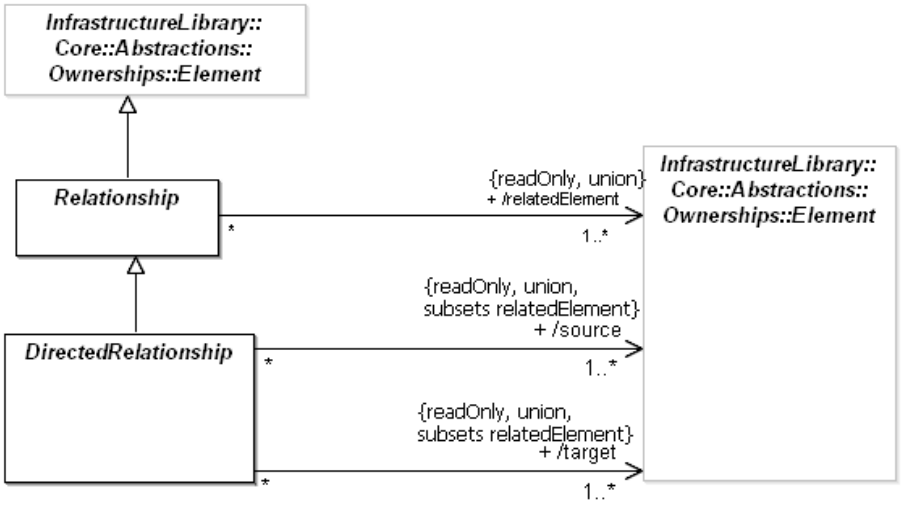


Figure 9.44 - The elements defined in the Relationships package

9.17.1 DirectedRelationship

A directed relationship represents a relationship between a collection of source model elements and a collection of target model elements.

Description

A directed relationship references one or more source elements and one or more target elements. DirectedRelationship is an abstract metaclass.

Generalizations

- “Relationship” on page 80

Attributes

No additional attributes

Associations

- / source: Element [1..*]
  Specifies the sources of the DirectedRelationship. Subsets *Relationship::relatedElement*. This is a derived union.
- / target: Element [1..*]
  Specifies the targets of the DirectedRelationship. Subsets *Relationship::relatedElement*. This is a derived union.

Constraints

No additional constraints

Semantics

DirectedRelationship has no specific semantics. The various subclasses of DirectedRelationship will add semantics appropriate to the concept they represent.

Notation

There is no general notation for a DirectedRelationship. The specific subclasses of DirectedRelationship will define their own notation. In most cases the notation is a variation on a line drawn from the source(s) to the target(s).

9.17.2 Relationship

Relationship is an abstract concept that specifies some kind of relationship between elements.

Description

A relationship references one or more related elements. Relationship is an abstract metaclass.

Generalizations

- “Element (as specialized)” on page 75

Attributes

No additional attributes.

Associations

- / relatedElement: Element [1..*]
  Specifies the elements related by the Relationship. This is a derived union.

Constraints

No additional constraints

Semantics

Relationship has no specific semantics. The various subclasses of Relationship will add semantics appropriate to the concept they represent.

Notation

There is no general notation for a Relationship. The specific subclasses of Relationship will define their own notation. In most cases the notation is a variation on a line drawn between the related elements.

9.18 StructuralFeatures Package

The StructuralFeatures package of the Abstractions package specifies an abstract generalization of structural features of classifiers.

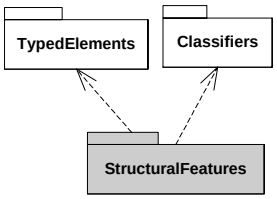


Figure 9.45 - The StructuralFeatures package

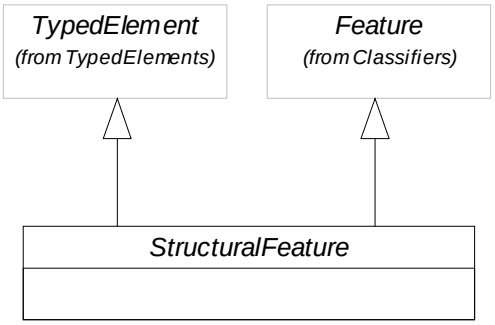


Figure 9.46 - The elements defined in the StructuralFeatures package

9.18.1 StructuralFeature

A structural feature is a typed feature of a classifier that specifies the structure of instances of the classifier.

Description

A structural feature is a typed feature of a classifier that specifies the structure of instances of the classifier. Structural feature is an abstract metaclass.

Generalizations

- “TypedElement” on page 87
- “Feature” on page 36

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

A structural feature specifies that instances of the featuring classifier have a slot whose value or values are of a specified type.

Notation

No additional notation

9.19 Super Package

The Super package of the Abstractions package provides mechanisms for specifying generalization relationships between classifiers.

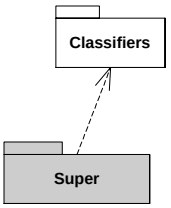


Figure 9.47 - The Super package

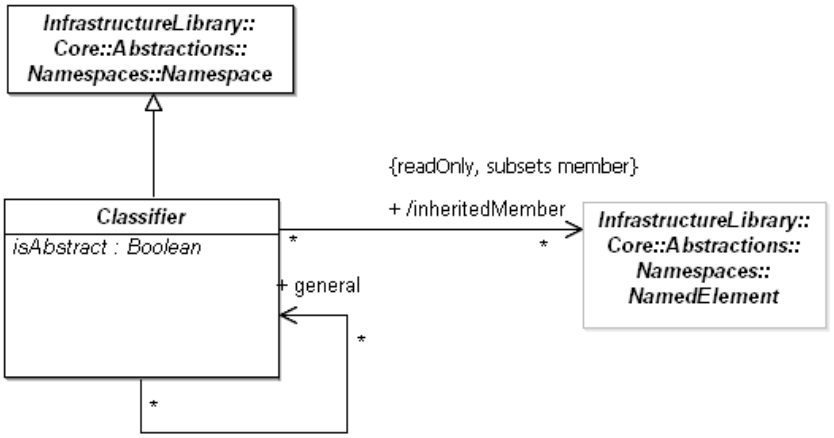


Figure 9.48 - The elements defined in the Super package

9.19.1 Classifier (as specialized)

Description

A classifier can specify a generalization hierarchy by referencing its general classifiers.

Generalizations

- “Classifier” on page 35

Attributes

- isAbstract: Boolean
  If *true*, the Classifier does not provide a complete declaration and can typically not be instantiated. An abstract classifier is intended to be used by other classifiers (e.g., as the target of general metarelationships or generalization relationships). Default value is *false*.

Associations

- general: Classifier[*]
  Specifies the more general classifiers in the generalization hierarchy for this Classifier.
- / inheritedMember: NamedElement[*]
  Specifies all elements inherited by this classifier from the general classifiers. Subsets *Namespace::member*. This is derived.

Constraints

[1] Generalization hierarchies must be directed and acyclical. A classifier cannot be both a transitively general and transitively specific classifier of the same classifier.

not self.allParents()->includes(self)

[2] A classifier may only specialize classifiers of a valid type.

self.parents()->forAll(c | self.maySpecializeType(c))

[3] The inheritedMember association is derived by inheriting the inheritable members of the parents.

self.inheritedMember = self.inherit(self.parents()->collect(p | p.inheritableMembers(self))->asSet())

Additional Operations

[1] The query parents() gives all of the immediate ancestors of a generalized Classifier.

Classifier::parents(): Set(Classifier);
parents = general

[2] The query allParents() gives all of the direct and indirect ancestors of a generalized Classifier.

Classifier::allParents(): Set(Classifier);
allParents = self.parents()->union(self.parents()->collect(p | p.allParents()))

[3] The query inheritableMembers() gives all of the members of a classifier that may be inherited in one of its descendants, subject to whatever visibility restrictions apply.

Classifier::inheritableMembers(c: Classifier): Set(NamedElement);
pre: c.allParents()->includes(self)
inheritableMembers = member->select(m | c.hasVisibilityOf(m))

[4] The query hasVisibilityOf() determines whether a named element is visible in the classifier. It is only called when the argument is something owned by a parent.

Classifier::hasVisibilityOf(n: NamedElement) : Boolean;

pre: self.allParents()->collect(c | c.member)->includes(n)

hasVisibilityOf = (n.visibility <> #private)

[5] The query inherit() defines how to inherit a set of elements. Here the operation is defined to inherit them all. It is intended to be redefined in circumstances where inheritance is affected by redefinition.

Classifier::inherit(inhs: Set(NamedElement)): Set(NamedElement);
inherit = inhs

[6] The query maySpecializeType() determines whether this classifier may have a generalization relationship to classifiers of the specified type. By default a classifier may specialize classifiers of the same or a more general type. It is intended to be redefined by classifiers that have different specialization constraints.

Classifier::maySpecializeType(c : Classifier) : Boolean;
maySpecializeType = self.oclIsKindOf(c.oclType)

Semantics

The specific semantics of how generalization affects each concrete subtype of Classifier varies.

An instance of a specific Classifier is also an (indirect) instance of each of the general Classifiers. Therefore, features specified for instances of the general classifier are implicitly specified for instances of the specific classifier. Any constraint applying to instances of the general classifier also applies to instances of the specific classifier.

Notation

The name of an abstract Classifier is shown in italics.

Generalization is shown as a line with a hollow triangle as an arrowhead between the symbols representing the involved classifiers. The arrowhead points to the symbol representing the general classifier. This notation is referred to as “separate target style.” See the example sub clause below.

Presentation Options

Multiple Classifiers that have the same general classifier can be shown together in the “shared target style.” See the example sub clause below.

An abstract Classifier can be shown using the keyword {abstract} after or below the name of the Classifier.

Examples

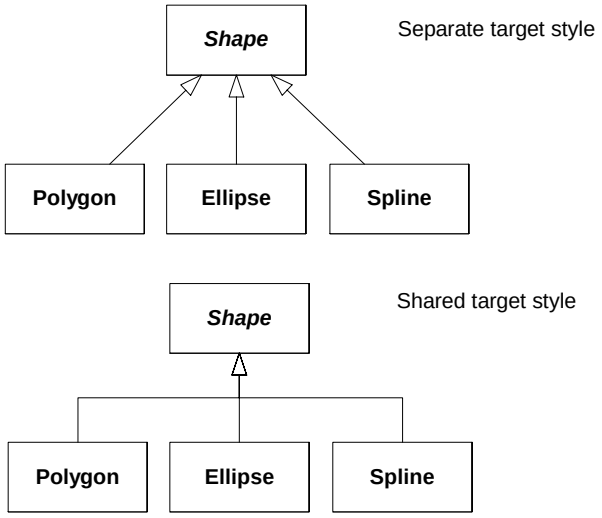


Figure 9.49 - Example class generalization hierarchy

9.20 TypedElements Package

The TypedElements subpackage of the Abstractions package defines typed elements and their types.

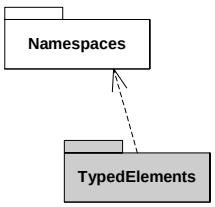


Figure 9.50 - The TypedElements package

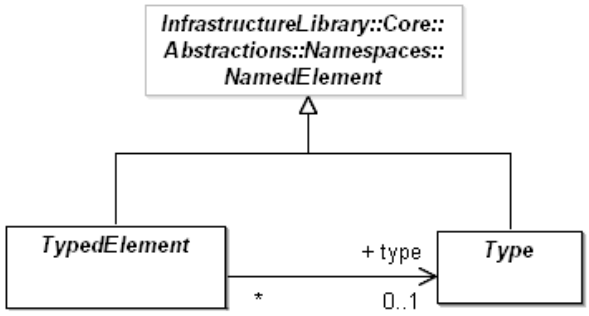


Figure 9.51 - The elements defined in the TypedElements package

9.20.1 Type

A type constrains the values represented by a typed element.

Description

A type serves as a constraint on the range of values represented by a typed element. Type is an abstract metaclass.

Generalizations

- “NamedElement” on page 71

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

[1] The query conformsTo() gives true for a type that conforms to another. By default, two types do not conform to each other. This query is intended to be redefined for specific conformance situations.

```
conformsTo(other: Type): Boolean;
conformsTo = false
```

Semantics

A type represents a set of values. A typed element that has this type is constrained to represent values within this set.

Notation

No general notation

9.20.2 TypedElement

A typed element has a type.

Description

A typed element is an element that has a type that serves as a constraint on the range of values the element can represent. Typed element is an abstract metaclass.

Generalizations

- “NamedElement” on page 71

Attributes

No additional attributes

Associations

- type: Type [0..1]
  The type of the TypedElement.

Constraints

No additional constraints.

Semantics

Values represented by the element are constrained to be instances of the type. A typed element with no associated type may represent values of any type.

Notation

No general notation

9.21 Visibilities Package

The Visibility subpackage of the Abstractions package provides basic constructs from which visibility semantics can be constructed.

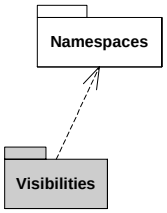


Figure 9.52 - The Visibilities package

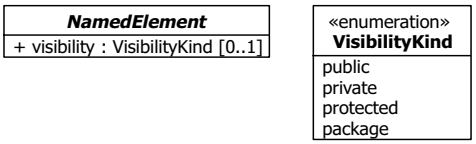


Figure 9.53 - The elements defined in the Visibilities package

9.21.1 NamedElement (as specialized)

Description

NamedElement has a visibility attribute.

Attributes

- visibility: VisibilityKind [0..1]
  Determines where the NamedElement appears within different Namespaces within the overall model, and its accessibility.

Generalizations

- “NamedElement” on page 71

Associations

No additional associations

Constraints

[1] If a NamedElement is not owned by a Namespace, it does not have a visibility.
namespace->isEmpty() **implies** visibility->isEmpty()

Semantics

The visibility attribute provides the means to constrain the usage of a named element either in namespaces or in access to the element. It is intended for use in conjunction with import, generalization, and access mechanisms.

9.21.2 VisibilityKind

VisibilityKind is an enumeration type that defines literals to determine the visibility of elements in a model.

Generalizations

- None

Description

VisibilityKind is an enumeration of the following literal values:

- public
- private
- protected
- package

Semantics

VisibilityKind is intended for use in the specification of visibility in conjunction with, for example, the Imports, Generalizations, Packages, and Classifiers packages. Detailed semantics are specified with those mechanisms. If the Visibility package is used without those packages, these literals will have different meanings, or no meanings.

- A public element is visible to all elements that can access the contents of the namespace that owns it.
- A private element is only visible inside the namespace that owns it.
- A protected element is visible to elements that have a generalization relationship to the namespace that owns it.
- A package element is owned by a namespace that is not a package, and is visible to elements that are in the same package as its owning namespace. Only named elements that are not owned by packages can be marked as having package visibility. Any element marked as having package visibility is visible to all elements within the nearest enclosing package (given that other owning elements have proper visibility). Outside the nearest enclosing package, an element marked as having package visibility is not visible.

In circumstances where a named element ends up with multiple visibilities, for example by being imported multiple times, public visibility overrides private visibility, i.e., if an element is imported twice into the same namespace, once using public import and once using private import, it will be public.

Notation

The following visual presentation options are available for representing VisibilityKind enumeration literal values:

“+” public

“-“ private

“#” protected

“~” package

10 Core::Basic

The Basic package of InfrastructureLibrary::Core provides a minimal class-based modeling language on top of which more complex languages can be built. It is intended for reuse by the Essential layer of the Meta-Object Facility (MOF). The metaclasses in Basic are specified using four diagrams: Types, Classes, DataTypes, and Packages. Basic can be viewed as an instance of itself. More complex versions of the Basic constructs are defined in Constructs, which is intended for reuse by the Complete layer of MOF as well as the UML Superstructure.

Figure 10.1 illustrates the relationships between the Core packages and how they contribute to the origin and evolution of package Basic. Package Basic imports model elements from package PrimitiveTypes. Basic also contains metaclasses derived from shared metaclasses defined in packages contained in Abstractions. These shared metaclasses are included in Basic by copy.

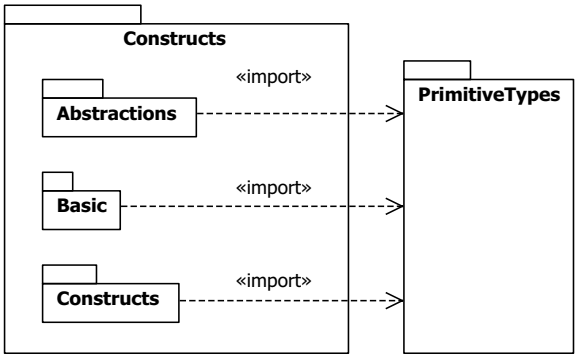


Figure 10.1 - The Core package is owned by the InfrastructureLibrary package and contains several subpackages

10.1 Types Diagram

The Types diagram defines abstract metaclasses that deal with naming and typing of elements.

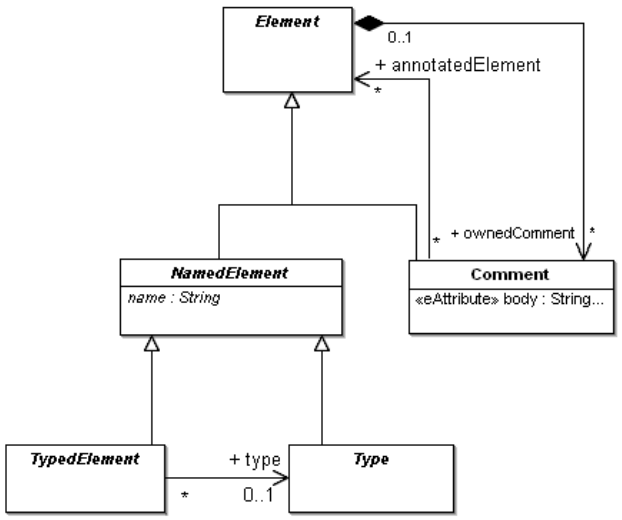


Figure 10.2 - The classes defined in the Types diagram

10.1.1 Comment

Description

Basic::Comment reuses the definition of Comment from *Abstractions::Comments*.

Generalizations

- “Element” on page 93

Attributes

- body: String [0..1]
  Specifies a string that is the comment.

Associations

- annotatedElement: Element[*]
  Redefines the corresponding property in *Abstractions.*

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

10.1.2 Element

An element is a constituent of a model.

Description

Element is an abstract metaclass with no superclass. It is used as the common superclass for all metaclasses in the infrastructure library.

Generalizations

- None

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Subclasses of Element provide semantics appropriate to the concept they represent.

Notation

There is no general notation for an Element. The specific subclasses of Element define their own notation.

10.1.3 NamedElement

Description

A named element represents elements with names.

Generalizations

- “Element” on page 93

Attributes

- name: String [0..1]
  The name of the element.

Semantics

Elements with names are instances of NamedElement. The name for a named element is optional. If specified, then any valid string, including the empty string, may be used.

Notation

As an abstract class, Basic::NamedElement has no notation.

10.1.4 Type

Description

A type is a named element that is used as the type for a typed element.

Generalizations

- “NamedElement” on page 93

Attributes

No additional attributes

Semantics

Type is the abstract class that represents the general notion of the type of a typed element and constrains the set of values that the typed element may refer to.

Notation

As an abstract class, Basic::Type has no notation.

10.1.5 TypedElement

Description

A typed element is a kind of named element that represents elements with types.

Generalizations

- “NamedElement” on page 93

Attributes

- type: Type [0..1]
  The type of the element.

Semantics

Elements with types are instances of TypedElement. A typed element may optionally have no type. The type of a typed element constrains the set of values that the typed element may refer to.

Notation

As an abstract class, Basic::TypedElement has no notation.

10.2 Classes Diagram

The Classes diagram defines the constructs for class-based modeling.

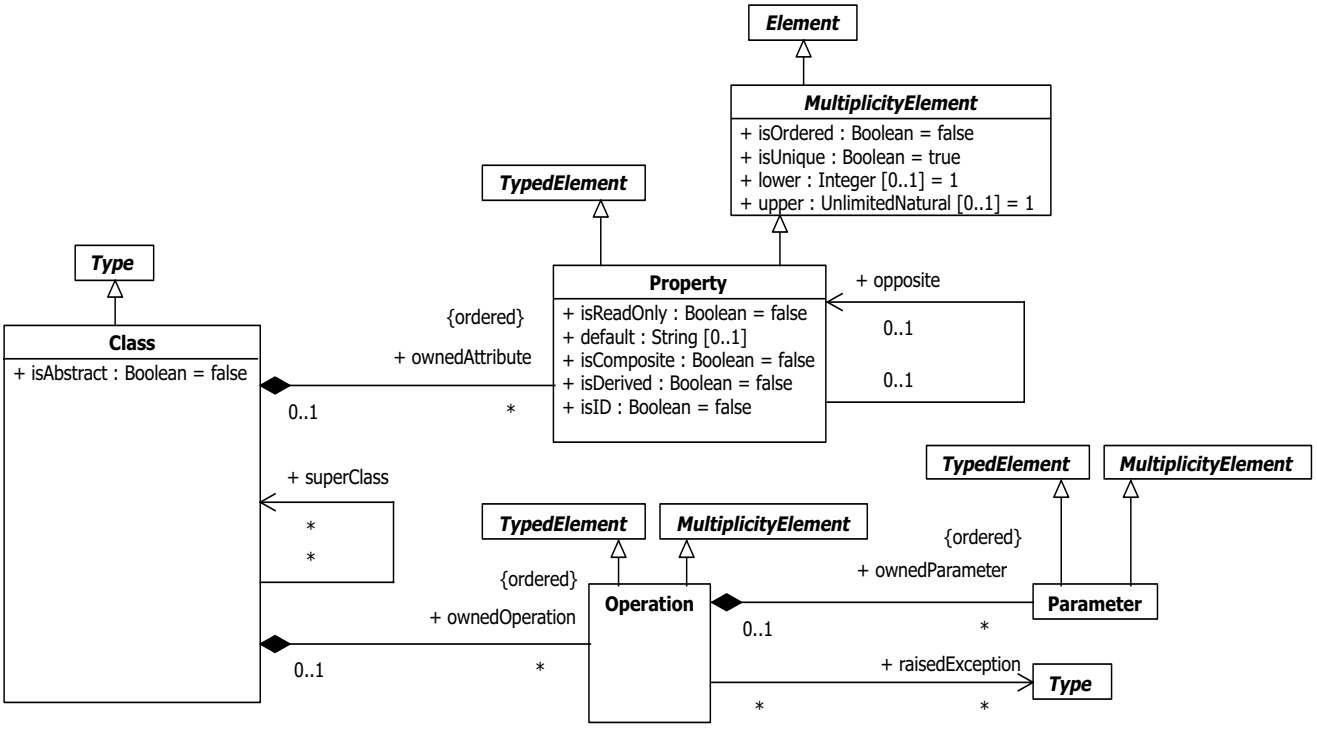


Figure 10.3 - The classes defined in the Classes diagram

10.2.1 Class

Description

A class is a type that has objects as its instances.

Generalizations

- “Type” on page 94

Attributes

- isAbstract : Boolean
  True when a class is abstract. The default value is false.
- ownedAttribute : Property [*]
  The attributes owned by a class. These do not include the inherited attributes. Attributes are represented by instances of Property.

- ownedOperation : Operation [*]
  The operations owned by a class. These do not include the inherited operations.
- superClass : Class[*]
  The immediate superclasses of a class, from which the class inherits.

Semantics

Classes have attributes and operations and participate in inheritance hierarchies. Multiple inheritance is allowed. The instances of a class are objects. When a class is abstract it cannot have any direct instances. Any direct instance of a concrete (i.e., non-abstract) class is also an indirect instance of its class' superclasses. An object has a slot for each of its class's direct and inherited attributes. An object permits the invocation of operations defined in its class and its class' superclasses. The context of such an invocation is the invoked object.

A class cannot access private features of another class, or protected features on another class that is not its supertype. When creating and deleting associations, at least one end must allow access to the class.

Notation

The notation for Basic::Class is the same as that for *Constructs::Class* with the omission of those aspects of the notation that cannot be represented by the Basic model.

10.2.2 MultiplicityElement

Description

Basic::MultiplicityElement reuses the definition from *Abstractions::MultiplicityElement*.

Generalizations

- "Element" on page 93

Description

Constructs::Relationship reuses the definition of *Relationship* from *Abstractions::Relationships*. It adds a specialization to *Constructs::Element*.

Generalizations

- "Element" on page 93

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

10.2.3 Operation

Description

An operation is owned by a class and may be invoked in the context of objects that are instances of that class. It is a typed element and a multiplicity element.

Generalizations

- “TypedElement” on page 94
- “TypedElement” on page 94 - MultiplicityElement.

Attributes

- class : Class [0..1]
  The class that owns the operation.
- ownedParameter : Parameter [*] {ordered, composite }
  The parameters to the operation.
- raisedException : Type [*]
  The exceptions that are declared as possible during an invocation of the operation.

Semantics

An operation belongs to a class. It is possible to invoke an operation on any object that is directly or indirectly an instance of the class. Within such an invocation the execution context includes this object and the values of the parameters. The type of the operation, if any, is the type of the result returned by the operation, and the multiplicity is the multiplicity of the result. An operation can be associated with a set of types that represent possible exceptions that the operation may raise.

Notation

The notation for Basic::Class is the same as that for *Constructs::Class* with the omission of those aspects of the notation that cannot be represented by the Basic model.

10.2.4 Parameter

Description

A parameter is a typed element that represents a parameter of an operation.

Attributes

- operation: Operation [0..1]
  The operation that owns the parameter.

Semantics

When an operation is invoked, an argument may be passed to it for each parameter. Each parameter has a type and a multiplicity. Every Basic::Parameter is associated with an operation, although subclasses of Parameter elsewhere in the UML model do not have to be associated with an operation, hence the 0..1 multiplicity.

Notation

The notation for Basic::Parameter is the same as that for Constructs::Parameter with the omission of those aspects of the notation that cannot be represented by the Basic model.

10.2.5 Property

Description

A property is a typed element that represents an attribute of a class.

Generalizations

- “TypedElement” on page 94
- “TypedElement” on page 94 - MultiplicityElement.

Attributes

- class : Class [0..1]
  The class that owns the property, and of which the property is an attribute.
- default : String [0..1]
  A string that is evaluated to give a default value for the attribute when an object of the owning class is instantiated.
- isComposite : Boolean
  If isComposite is true, the object containing the attribute is a container for the object or value contained in the attribute. The default value is false.
- isDerived : Boolean
  If isDerived is true, the value of the attribute is derived from information elsewhere. The default value is false.
- isReadOnly : Boolean
  If isReadOnly is true, the attribute may not be written to after initialization. The default value is false.
- opposite : Property [0..1]
  Two attributes attr1 and attr2 of two objects o1 and o2 (which may be the same object) may be paired with each other so that o1.attr1 refers to o2 if and only if o2.attr2 refers to o1. In such a case attr1 is the opposite of attr2 and attr2 is the opposite of attr1.
- isID : Boolean
  True indicates this property can be used to uniquely identify an instance of the containing Class. Default value is *false*.

Semantics

A property represents an attribute of a class. A property has a type and a multiplicity. When a property is paired with an opposite they represent two mutually constrained attributes. The semantics of two properties that are mutual opposites are the same as for bidirectionally navigable associations in Constructs, with the exception that the association has no explicit links as instances, and has no name.

A property may be marked as being (part of) the identifier (if any) for classes of which it is a member. The interpretation of this is left open but this could be mapped to implementations such as primary keys for relational database tables or ID attributes in XML. If multiple properties are marked (possibly in superclasses), then it is the combination of the (property, value) tuples that will logically provide the uniqueness for any instance. Hence there is no need for any specification of order and it is possible for some (but not all) of the property values to be empty. If the property is multivalued, then all values are included.

Notation

When a Basic::Property has no opposite, its notation is the same for *Constructs::Property* when used as an attribute with the omission of those aspects of the notation that cannot be represented by the Basic model. Normally if the type of the property is a data type, the attribute is shown within the attribute compartment of the class box, and if the type of the property is a class, it is shown using the association-like arrow notation.

When a property has an opposite, the pair of attributes are shown using the same notation as for a *Constructs::Association* with two navigable ends, with the omission of those aspects of the notation that cannot be represented by the Basic model.

10.3 DataTypes Diagram

The DataTypes diagram defines the metaclasses that define data types.

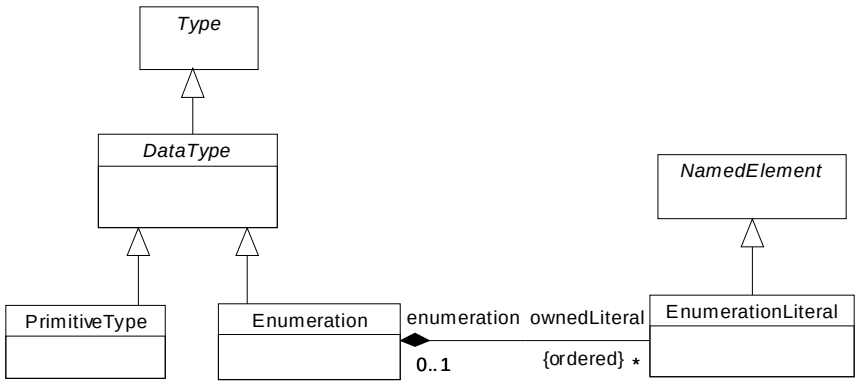


Figure 10.4 - The classes defined in the DataTypes diagram

10.3.1 DataType

Description

DataType is an abstract class that acts as a common superclass for different kinds of data types.

Generalizations

- “Type” on page 94

Attributes

No additional attributes

Semantics

DataType is the abstract class that represents the general notion of being a data type (i.e., a type whose instances are identified only by their value).

Notation

As an abstract class, Basic::DataType has no notation.

10.3.2 Enumeration

Description

An enumeration defines a set of literals that can be used as its values.

Generalizations

- “DataType” on page 99

Attributes

- ownedLiteral: EnumerationLiteral [*] {ordered, composite} — The ordered collection of literals for the enumeration.

Semantics

An enumeration defines a finite ordered set of values, such as {red, green, blue}.

The values denoted by typed elements whose type is an enumeration must be taken from this set.

Notation

The notation for Basic::Enumeration is the same as that for *Constructs::Enumeration* with the omission of those aspects of the notation that cannot be represented by the Basic model.

10.3.3 EnumerationLiteral

Description

An enumeration literal is a value of an enumeration.

Generalizations

- “NamedElement” on page 93

Attributes

- enumeration: Enumeration [0..1]
  The enumeration that this literal belongs to.

Semantics

See Enumeration

Notation

See Enumeration

10.3.4 PrimitiveType

Description

A primitive type is a data type implemented by the underlying infrastructure and made available for modeling.

Generalizations

- “DataType” on page 99

Attributes

No additional attributes

Semantics

Primitive types used in the Basic model itself are Integer, Boolean, String, and UnlimitedNatural. Their specific semantics is given by the tooling context, or in extensions of the metamodel (e.g., OCL).

Notation

The notation for a primitive type is implementation-dependent. Notation for the primitive types used in the UML metamodel is given in the “PrimitiveTypes Package” on page 201.

10.4 Packages Diagram

The Packages diagram defines the Basic constructs related to Packages and their contents.

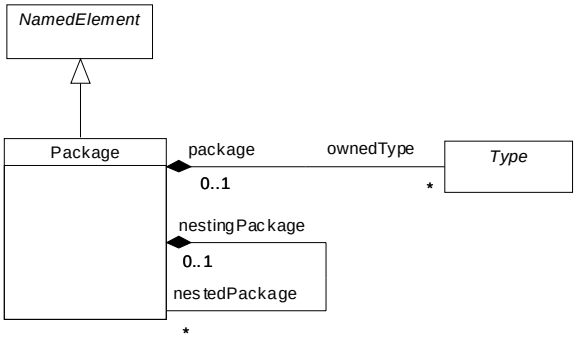


Figure 10.5 - The classes defined in the Packages diagram

10.4.1 Package

Description

A package is a container for types and other packages.

Generalizations

- “NamedElement” on page 93

Attributes

- nestedPackage : Package [*]
  The set of contained packages.
- nestingPackage : Package [0..1]
  The containing package.
- ownedType : Type [*]
  The set of contained types.
- URI: String [0..1] {id}
  Provides an identifier for the package that can be used for many purposes. A URI is the universally unique identification of the package following the IETF URI specification, RFC 2396 http://www.ietf.org/rfc/rfc2396.txt and it must comply with those syntax rules.

Semantics

Packages provide a way of grouping types and packages together, which can be useful for understanding and managing a model. A package cannot contain itself. The URI can be specified to provide a unique identifier for a Package.

Notation

Containment of packages and types in packages uses the same notation as for *Constructs::Packages* with the omission of those aspects of the notation that cannot be represented by the Basic model.

10.4.2 Type

Note – (additional properties - see “Type” on page 94).

Description

A type can be contained in a package.

Generalizations

- “NamedElement” on page 93

Attributes

- package : Package [0..1]
  The containing package.

Semantics

No additional semantics

Notation

Containment of types in packages uses the same notation as for Constructs::Packages with the omission of those aspects of the notation that cannot be represented by the Basic model.

11 Core::Constructs

This clause describes the Constructs package of InfrastructureLibrary::Core. The Constructs package is intended to be reused by the Meta-Object Facility.

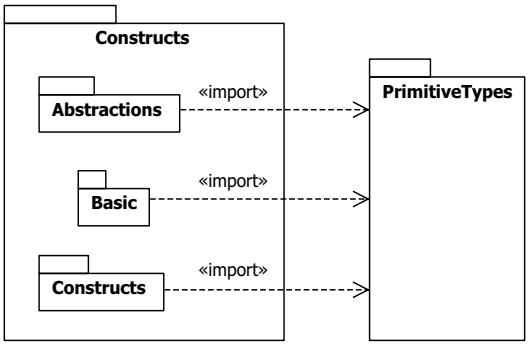


Figure 11.1 -The Core package is owned by the InfrastructureLibrary package, and contains several subpackages

The Constructs package is specified by a number of diagrams each of which is described in a separate sub clause below.

The constructs package is dependent on several other packages, notably Basic and various packages from Abstractions, as depicted in Figure 11.2.

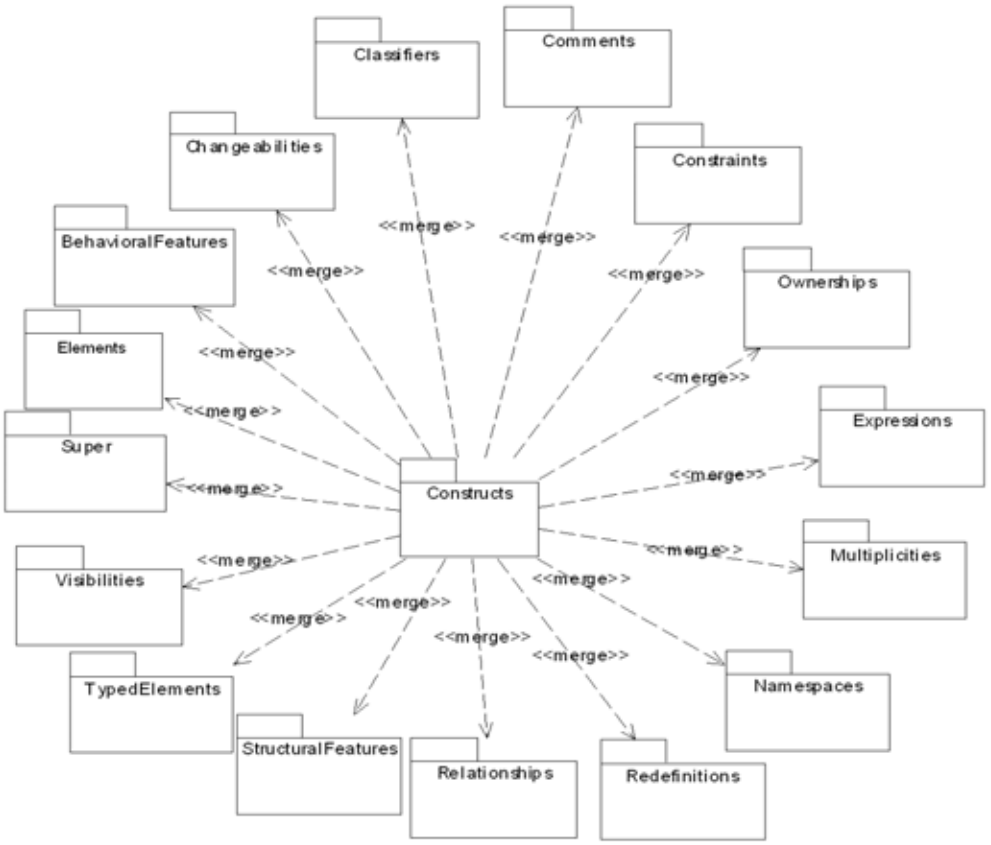


Figure 11.2 - The Constructs package depends on several other packages

Figure 11.2 illustrates the relationships between the Core packages and how they contribute to the origin and evolution of package Constructs. Package Constructs imports model elements from package PrimitiveTypes. Constructs also contains metaclasses from Basic and shared metaclasses defined in packages contained in Abstractions. These shared metaclasses are included in Constructs by copy. Figure 11.2 uses PackageMerge to illustrate the packages that contribute to the definition of Constructs and how. *The InfrastructureLibrary metamodel does not actually include these package merges as Constructs is a complete metamodel that already includes all the metaclasses in the referenced packages.* This allows Constructs to be understood and used without requiring the use of PackageMerge.

11.1 Root Diagram

The Root diagram in the Constructs package specifies the Element, Relationship, DirectedRelationship, and Comment constructs.

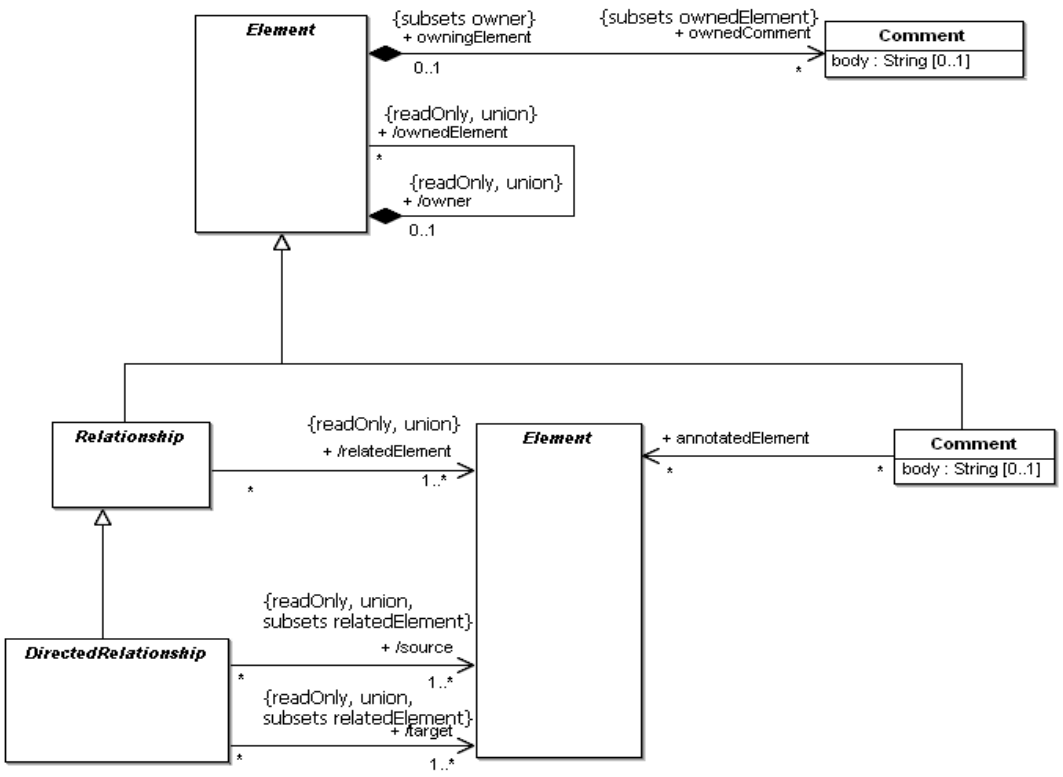


Figure 11.3 - The Root diagram of the Constructs package

11.1.1 Comment

Description

Generalizations

- “Element” on page 106

Attributes

- body: String
  Specifies a string that is the comment.

Associations

- annotatedElement: Element[*]
  Redefines the corresponding property in *Abstractions*.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

11.1.2 DirectedRelationship

Description

Constructs::DirectedRelationship reuses the definition of *DirectedRelationship* from *Abstractions::Relationships*. It adds a specialization to *Constructs::Relationship*.

Generalizations

- “Relationship” on page 107

Attributes

No additional attributes

Associations

- /source: Element[1..*]
  Redefines the corresponding property in *Abstractions*. Subsets *Relationship::relatedElement*. This is a derived union.
- /target: Element[1..*]
  Redefines the corresponding property in *Abstractions*. Subsets *Relationship::relatedElement*. This is a derived union.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

11.1.3 Element

Description

Constructs::Element reuses the definition of *Element* from *Abstractions::Comments*.

Generalizations

- None

Attributes

No additional attributes

Associations

- /ownedComment: Comment[*]
  Redefines the corresponding property in *Abstractions*. Subsets *Element::ownedElement*.
- /ownedElement: Element[*]
  Redefines the corresponding property in *Abstractions*. This is a derived union.
- /owner: Element[0..1]
  Redefines the corresponding property in *Abstractions*. This is a derived union.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

11.1.4 Relationship

Description

Constructs::Relationship reuses the definition of *Relationship* from *Abstractions::Relationships*. It adds a specialization to *Constructs::Element*.

Generalizations

- “Element” on page 106

Attributes

No additional attributes

Associations

- /relatedElement: Element[1..*]
  Redefines the corresponding property in *Abstractions*. This is a derived union.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

11.2 Expressions Diagram

The Expressions diagram in the Constructs package specifies the ValueSpecification, Expression, and OpaqueExpression constructs.

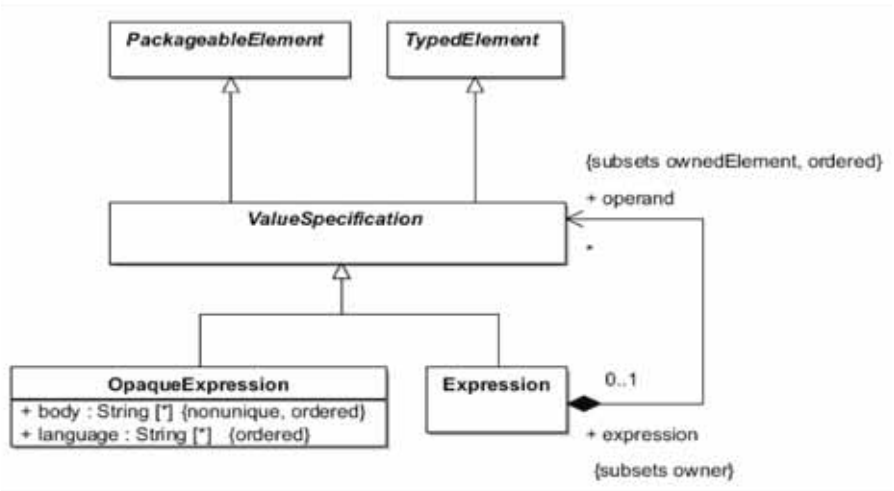


Figure 11.4 - The Expressions diagram of the Constructs package

11.2.1 Expression

Description

Constructs::Expression reuses the definition of *Expression* from *Abstractions::Expressions*. It adds a specialization to *Constructs::ValueSpecification*.

Generalizations

- “ValueSpecification” on page 109

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

11.2.2 OpaqueExpression

Description

Constructs::OpaqueExpression reuses the definition of *OpaqueExpression* from *Abstractions::Expressions*. It adds a specialization to *Constructs::ValueSpecification*.

Generalizations

- “PackageableElement” on page 149
- “TypedElement” on page 135

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

11.2.3 ValueSpecification

Description

Constructs::ValueSpecification reuses the definition of *ValueSpecification* from *Abstractions::Expressions*. It adds a specialization to *Constructs::TypedElement*.

Generalizations

- “Relationship” on page 107

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

11.3 Classes Diagram

The Classes diagram of the Constructs package specifies the Association, Class, and Property constructs and adds features to the Classifier and Operation constructs.

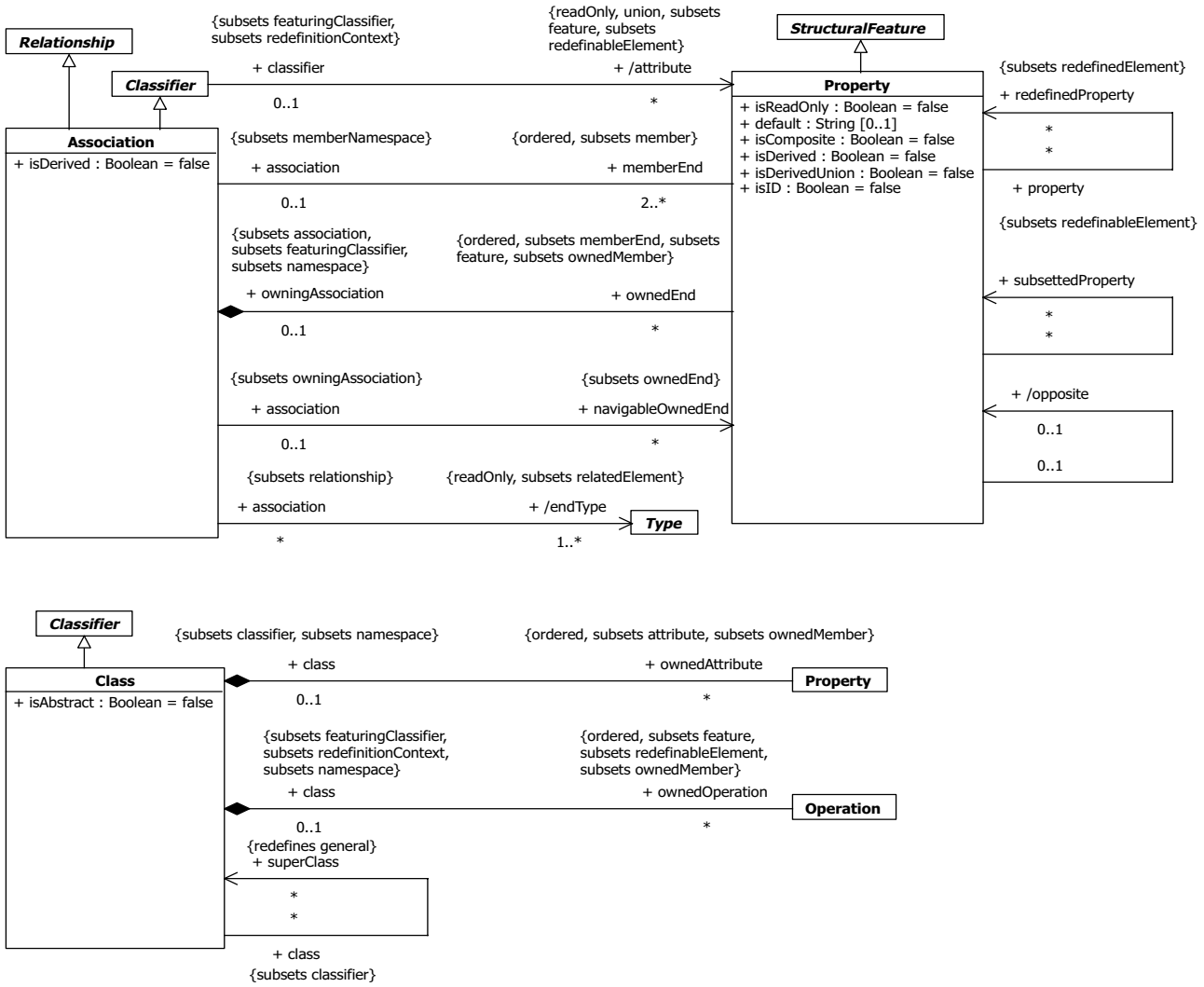


Figure 11.5 - The Classes diagram of the Constructs package

11.3.1 Association

An association describes a set of tuples whose values refer to typed instances. An instance of an association is called a link.A link is a tuple with one value for each end of the association, where each value is an instance of the type of the end.

Description

An association specifies a semantic relationship that can occur between typed instances. It has at least two ends represented by properties, each of which is connected to the type of the end. More than one end of an association may have the same type.

An end property of an association that is owned by an end class or that is a navigable owned end of the association indicates that the association is navigable from the opposite ends, otherwise the association is not navigable from the opposite ends.

Generalizations

- "Classifier" on page 130
- "Relationship" on page 107

Attributes

- isDerived : Boolean
  Specifies whether the association is derived from other model elements such as other associations or constraints. The default value is *false*.

Associations

- memberEnd : Property [2..*]
  Each end represents participation of instances of the classifier connected to the end in links of the association. This is an ordered association. Subsets *Namespace::member.*
- ownedEnd : Property [*]
  The ends that are owned by the association itself. This is an ordered association. Subsets *Association::memberEnd, Classifier::feature*, and *Namespace::ownedMember.*
- / endType: Type [1..*]
  References the classifiers that are used as types of the ends of the association.
- navigableOwnedEnd : Property [*]
  The navigable ends that are owned by the association itself. Subsets Association.ownedEnd.

Constraints

[1] An association specializing another association has the same number of ends as the other association.

```
parents()->select(oclIsKindOf(Association)).oclAsType(Association)->
    forAll(p | p.memberEnd->size() = self.memberEnd->size())
```

[2] When an association specializes another association, every end of the specific association corresponds to an end of the general association, and the specific end reaches the same type or a subtype of the more general end.

```
Sequence{1..self.memberEnd->size()}->
    forAll(i | self.general->select(oclIsKindOf(Association)).oclAsType(Association)->
        forAll(ga |self.memberEnd->at(i).type.conformsTo(ga.memberEnd->at(i).type)))
```

[3] endType is derived from the types of the member ends.

```
self.endType = self.memberEnd->collect(e | e.type)
```

[4] Only binary associations can be aggregations

```
self.memberEnd->exists(isComposite) implies self.memberEnd->size() = 2
```

[5] Association ends of associations with more than two ends must be owned by the association.

```
if memberEnd->size() > 2
then ownedEnd->includesAll(memberEnd)
```

Semantics

An association declares that there can be links between instances of the associated types. A link is a tuple with one value for each end of the association, where each value is an instance of the type of the end.

When one or more ends of the association have isUnique=false, it is possible to have several links associating the same set of instances. In such a case, links carry an additional identifier apart from their end values.

When one or more ends of the association are ordered, links carry ordering information in addition to their end values.

For an association with N ends, choose any N-1 ends and associate specific instances with those ends. Then the collection of links of the association that refer to these specific instances will identify a collection of instances at the other end. The multiplicity of the association end constrains the size of this collection. If the end is marked as ordered, this collection will be ordered. If the end is marked as unique, this collection is a set; otherwise, it allows duplicate elements.

Subsetting represents the familiar set-theoretic concept. It is applicable to the collections represented by association ends, not to the association itself. It means that the subsetting association end is a collection that is either equal to the collection that it is subsetting or a proper subset of that collection. (Proper subsetting implies that the superset is not empty and that the subset has fewer members.) Subsetting is a relationship in the domain of extensional semantics.

Specialization is in contrast to *Subsetting* a relationship in the domain of intensional semantics, which is to say it characterized the criteria whereby membership in the collection is defined, not by the membership. One classifier may specialize another by adding or redefining features; a set cannot *specialize* another set. A naïve but popular and useful view has it that as the classifier becomes more specialized, the extent of the collection(s) of classified objects narrows. In the case of associations, subsetting ends, according to this view, correlates positively with specializing the association. This view falls down because it ignores the case of classifiers which, for whatever reason, denote the empty set. Adding new criteria for membership does not narrow the extent if the classifier already has a null denotation.

Redefinition is a relationship between features of classifiers within a specialization hierarchy. Redefinition may be used to change the definition of a feature, and thereby introduce a specialized classifier in place of the original featuring classifier, but this usage is incidental. The difference in domain (that redefinition applies to *features*) differentiates redefinition from specialization.

The combination of constraints [1,2] above with the semantics of property subsetting and redefinition specified in section 11.3.5 in constraints [3,4,5] imply that any association end that subsets or redefines another association end forces the association of the subsetting or redefining association end to be a specialization of the association of the subsetted or redefined association end respectively.

For n-ary associations, the lower multiplicity of an end is typically 0. A lower multiplicity for an end of an n-ary association of 1 (or more) implies that one link (or more) must exist for every possible combination of values for the other ends.

An association may represent a composite aggregation (i.e., a whole/part relationship). Only binary associations can be aggregations. Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it. Note that a part can (where allowed) be removed from a composite before the composite is deleted, and thus not be deleted as part of the composite. Compositions may be linked in a directed acyclic graph with transitive deletion characteristics; that is, deleting an element in one part of the graph will also result in the deletion of all elements of the subgraph below that element. Composition is represented by the *isComposite* attribute on the part end of the association being set to *true*.

Navigability means instances participating in links at runtime (instances of an association) can be accessed efficiently from instances participating in links at the other ends of the association. The precise mechanism by which such access is achieved is implementation specific. If an end is not navigable, access from the other ends may or may not be possible, and if it is, it might not be efficient. Note that tools operating on UML models are not prevented from navigating associations from non-navigable ends.

Semantic Variation Points

The order and way in which part instances in a composite are created is not defined.

The logical relationship between the derivation of an association and the derivation of its ends is not defined.

Notation

Any association may be drawn as a diamond (larger than a terminator on a line) with a solid line for each association end connecting the diamond to the classifier that is the end's type. An association with more than two ends can only be drawn this way.

A binary assocation is normally drawn as a solid line connecting two classifiers, or a solid line connecting a single classifier to itself (the two ends are distinct). A line may consist of one or more connected segments. The individual segments of the line itself have no semantic significance, but they may be graphically meaningful to a tool in dragging or resizing an association symbol.

An association symbol may be adorned as follows:

- The association's name can be shown as a name string near the association symbol, but not near enough to an end to be confused with the end's name.
- A slash appearing in front of the name of an association, or in place of the name if no name is shown, marks the association as being derived.
- A property string may be placed near the association symbol, but far enough from any end to not be confused with a property string on an end. A property string is a comma-delimited list of property expressions enclosed in curly braces. A property expression is, in the simplest case, a name such as 'redefines' or 'subsets.'
- On a binary association drawn as a solid line, a solid triangular arrowhead next to or in place of the name of the association and pointing along the line in the direction of one end indicates that end to be the last in the order of the ends of the association. The arrow indicates that the association is to be read as associating the end away from the direction of the arrow with the end to which the arrow is pointing (see Figure 11.6). This notation is for documentation purposes only and has no general semantic interpretation. It is used to capture some application-specific detail of the relationship between the associated classifiers.
- Generalizations between associations can be shown using a generalization arrow between the association symbols.

An association end is the connection between the line depicting an association and the icon (often a box) depicting the connected classifier. A name string may be placed near the end of the line to show the name of the association end. The name is optional and suppressible.

Various other notations can be placed near the end of the line as follows:

- A multiplicity.
- The BNF for property strings on association ends is:
  <property-string> ::= '{' <end-property> [',' <end-property>]* '}'
  <end-property> ::=

(
'subsets' <property-name> | 'redefines' <end-name>
)

where <*property-name*> and <*end-name*> are names of user-provided properties and association ends found in the model context.

If an association end is navigable, attribute-properties defined for attributes are legal as end-properties in the property string for that association end.

Note that by default an association end represents a set.

A stick arrowhead on the end of an association indicates the end is navigable. A small x on the end of an association indicates the end is not navigable. A visibility symbol can be added as an adornment on a navigable end to show the end's visibility as an attribute of the featuring classifier.

If the association end is derived, this may be shown by putting a slash in front of the name, or in place of the name if no name is shown.

The notation for an attribute can be applied to a navigable end name as specified in the Notation sub clause of "Property" on page 124.

A composite aggregation is shown using the same notation as a binary association, but with a solid, filled diamond at the aggregate end.

Presentation Options

When two lines cross, the crossing may optionally be shown with a small semicircular jog to indicate that the lines do not intersect (as in electrical circuit diagrams).

Various options may be chosen for showing navigation arrows on a diagram. In practice, it is often convenient to suppress some of the arrows and crosses and just show exceptional situations:

- Show all arrows and xs. Navigation and its absence are made completely explicit.
- Suppress all arrows and xs. No inference can be drawn about navigation. This is similar to any situation in which information is suppressed from a view.
- Suppress arrows for associations with navigability in both directions, and show arrows only for associations with one-way navigability. In this case, the two-way navigability cannot be distinguished from situations where there is no navigation at all; however, the latter case occurs rarely in practice.

If there are two or more aggregations to the same aggregate, they may be drawn as a tree by merging the aggregation ends into a single segment. Any adornments on that single segment apply to all of the aggregation ends.

Style Guidelines

Lines may be drawn using various styles, including orthogonal segments, oblique segments, and curved segments. The choice of a particular set of line styles is a user choice.

Generalizations between associations are best drawn using a different color or line width than what is used for the associations.

Examples

Figure 11.6 shows a binary association from *Player* to *Year* named *PlayedInYear*. The solid triangle indicates the order of reading: *Player PlayedInYear Year*. The figure further shows a ternary association between *Team*, *Year*, and *Player* with ends named *team*, *season*, and *goalie* respectively.

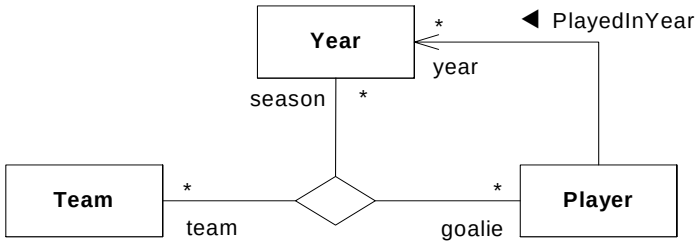


Figure 11.6 - Binary and ternary associations

The following example shows association ends with various adornments.

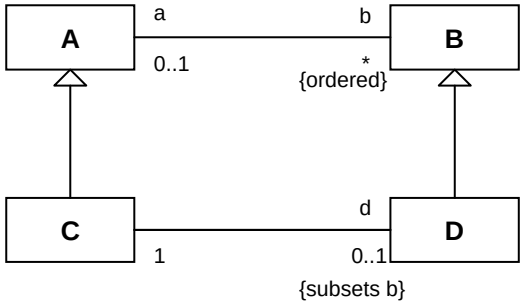


Figure 11.7 - Association ends with various adornments

The following adornments are shown on the four association ends in Figure 11.7.

- Names *a*, *b*, and *d* on three of the ends.
- Multiplicities 0..1 on *a*, * on *b*, 1 on the unnamed end, and 0..1 on *d*.
- Specification of ordering on *b*.
- Subsetting on *d*. For an instance of class C, the collection d is a subset of the collection b. This is equivalent to the OCL constraint:

  context C inv: b->includesAll(d)

The following examples show notation for navigable ends.

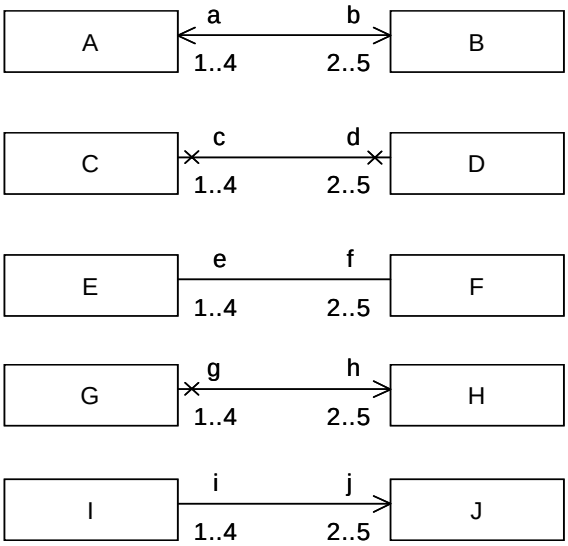


Figure 11.8 - Examples of navigable ends

In Figure 11.8:

- The top pair AB shows a binary association with two navigable ends.
- The second pair CD shows a binary association with two non-navigable ends.
- The third pair EF shows a binary association with unspecified navigability.
- The fourth pair GH shows a binary association with one end navigable and the other non-navigable.
- The fifth pair IJ shows a binary association with one end navigable and the other having unspecified navigability.

Figure 11.9 shows that the attribute notation can be used for an association end owned by a class, because an association end owned by a class is also an attribute. This notation may be used in conjunction with the line-arrow notation to make it perfectly clear that the attribute is also an association end.

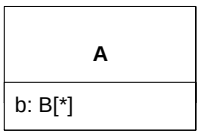


Figure 11.9 - Example of attribute notation for navigable end owned by an end class

Figure 11.10 shows the notation for a derived union. The attribute A::b is derived by being the strict union of all of the attributes that subset it. In this case there is just one of these, A1::b1. So for an instance of the class A1, b1 is a subset of b, and b is derived from b1.

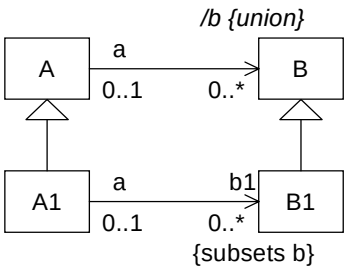


Figure 11.10 - Example of a derived union

Figure 11.11 shows the black diamond notation for composite aggregation.

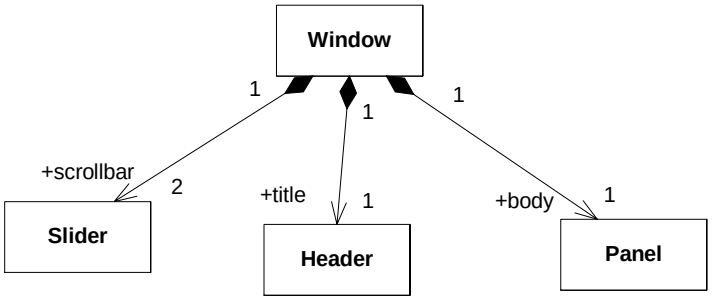


Figure 11.11 - Composite aggregation is depicted as a black diamond

11.3.2 Class

A class describes a set of objects that share the same specifications of features, constraints, and semantics. Constructs::Class merges the definition of Basic::Class with Constructs::Classifier.

Description

Class is a kind of classifier whose features are attributes and operations. Attributes of a class are represented by instances of *Property* that are owned by the class. Some of these attributes may represent the navigable ends of binary associations.

Generalizations

- “Classifier” on page 130.

Attributes

- isAbstract : Boolean
  This redefines the corresponding attributes in *Basic::Class* and *Abstractions::Classifier.*

Associations

- ownedAttribute : Property [*]
  The attributes (i.e., the properties) owned by the class. This is an ordered association. Subsets *Classifier::attribute* and *Namespace::ownedMember.*
- ownedOperation : Operation [*]
  The operations owned by the class. This is an ordered association. Subsets *Classifier::feature* and *Namespace::ownedMember.*
- superClass : Class [*]
  This gives the superclasses of a class. It redefines *Classifier::general.*

Constraints

No additional constraints

Additional Operations

[1] The inherit operation is overridden to exclude redefined properties.

```
Class::inherit(inhs: Set(NamedElement)) : Set(NamedElement);
inherit = inhs->excluding(inh |
    ownedMember->select(oclIsKindOf(RedefinableElement))->select(redefinedElement->includes(inh)))
```

Semantics

The purpose of a class is to specify a classification of objects and to specify the features that characterize the structure and behavior of those objects.

Objects of a class must contain values for each attribute that is a member of that class, in accordance with the characteristics of the attribute, for example its type and multiplicity.

When an object is instantiated in a class, for every attribute of the class that has a specified default, if an initial value of the attribute is not specified explicitly for the instantiation, then the default value specification is evaluated to set the initial value of the attribute for the object.

Operations of a class can be invoked on an object, given a particular set of substitutions for the parameters of the operation. An operation invocation may cause changes to the values of the attributes of that object. It may also return a value as a result, where a result type for the operation has been defined. Operation invocations may also cause changes in value to the attributes of other objects that can be navigated to, directly or indirectly, from the object on which the operation is invoked, to its output parameters, to objects navigable from its parameters, or to other objects in the scope of the operation's execution. Operation invocations may also cause the creation and deletion of objects.

Notation

A class is shown using the classifier symbol. As class is the most widely used classifier, the word "class" need not be shown in guillemets above the name. A classifier symbol without a metaclass shown in guillemets indicates a class.

Presentation Options

A class is often shown with three compartments. The middle compartment holds a list of attributes while the bottom compartment holds a list of operations.

Attributes or operations may be presented grouped by visibility. A visibility keyword or symbol can then be given once for multiple features with the same visibility.

Additional compartments may be supplied to show other details, such as constraints, or to divide features.

Style Guidelines

- Center class name in boldface.
- Capitalize the first letter of class names (if the character set supports uppercase).
- Left justify attributes and operations in plain face.
- Begin attribute and operation names with a lowercase letter.
- Put the class name in italics if the class is abstract.
- Show full attributes and operations when needed and suppress them in other contexts or when merely referring to a class.

Examples

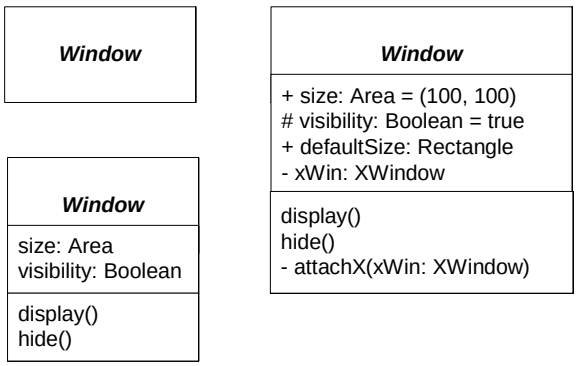


Figure 11.12 -Class notation: details suppressed, analysis-level details, implementation-level details

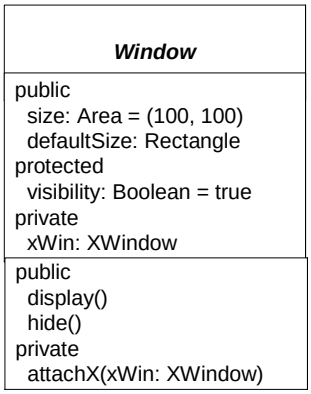


Figure 11.13 - Class notation: attributes and operations grouped according to visibility

11.3.3 Classifier

Note – (additional properties - see “Classifier” on page 130).

Description

Constructs::Classifier is defined in the Classifiers diagram. A Classifier is a Type. The Classes diagram adds the association between Classifier and Property that represents the *attributes* of the classifier.

Generalizations

- “Type” on page 134
- “Namespace” on page 147

Attributes

No additional attributes

Associations

- attribute: Property [*]
  Refers to all of the Properties that are direct (i.e., not inherited or imported) attributes of the classifier. Subsets *Classifier::feature* and is a derived union.

Constraints

No additional constraints

Semantics

All instances of a classifier have values corresponding to the classifier’s attributes.

Semantic Variation Points

The precise lifecycle semantics of aggregation is a semantic variation point.

Notation

An attribute can be shown as a text string. The format of this string is specified in the Notation sub clause of “Property” on page 124.

All redefinitions should be made explicit with the use of a {redefines <x>} property string. Matching features in subclasses without an explicit redefinition result in a redefinition that need not be shown in the notation. Redefinition prevents inheritance of a redefined element into the redefinition context thereby making the name of the redefined element available for reuse, either for the redefining element, or for some other.

Presentation Options

The type, visibility, default, multiplicity, property string may be suppressed from being displayed, even if there are values in the model.

The individual properties of an attribute can be shown in columns rather than as a continuous string.

The attribute compartment is often suppressed, especially when a data type does not contain attributes. The operation compartment may be suppressed. A separator line is not drawn for a missing compartment. If a compartment is suppressed, no inference can be drawn about the presence or absence of elements in it. Compartment names can be used to remove ambiguity, if necessary.

Additional compartments may be supplied to show other predefined or user-defined model properties (for example, to show business rules, responsibilities, variations, events handled, exceptions raised, and so on). Most compartments are simply lists of strings, although more complicated formats are also possible. Appearance of each compartment should preferably be implicit based on its contents. Compartment names may be used, if needed.

A data-type symbol with a stereotype icon may be “collapsed” to show just the stereotype icon, with the name of the data type either inside the rectangle or below the icon. Other contents of the data type are suppressed.

Style Guidelines

- Center the name of the data type in boldface.
- Center keyword (including stereotype names) in plain face within guillemets above data-type name.
- For those languages that distinguish between uppercase and lowercase characters, capitalize names (i.e., begin them with an uppercase character).
- Left justify attributes and operations in plain face.
- Begin attribute and operation names with a lowercase letter.
- Show full attributes and operations when needed and suppress them in other contexts or references.

Attribute names typically begin with a lowercase letter. Multiword names are often formed by concatenating the words and using lowercase for all letters, except for upcasing the first letter of each word but the first.

Examples

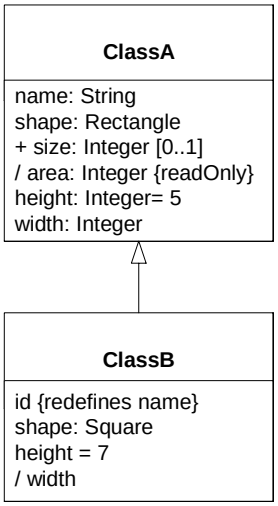


Figure 11.14 - Examples of attributes

The attributes in *Figure 11.14* are explained below.

- ClassA::name is an attribute with type String.
- ClassA::shape is an attribute with type Rectangle.
- ClassA::size is a public attribute with type Integer with multiplicity 0..1.
- ClassA::area is a derived attribute with type Integer. It is marked as read-only.
- ClassA::height is an attribute of type Integer with a default initial value of 5.
- ClassA::width is an attribute of type Integer.
- ClassB::id is an attribute that redefines ClassA::name.
- ClassB::shape is an attribute that redefines ClassA::shape. It has type Square, a specialization of Rectangle.
- ClassB::height is an attribute that redefines ClassA::height. It has a default of 7 for ClassB instances which overrides the ClassA default of 5.
- ClassB::width is a derived attribute that redefines ClassA::width, which is not derived.

An attribute may also be shown using association notation, with no adornments at the tail of the arrow as shown in Figure 11.15.

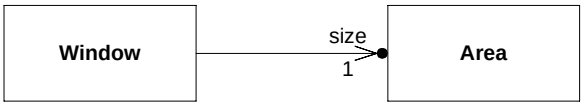


Figure 11.15 - Association-like notation for attribute

11.3.4 Operation

Note – (additional properties - see “Operation” on page 153).

Description

Constructs::Operation is defined in the Operations diagram. The Classes diagram adds the association between Operation and *Class* that represents the ownership of the operation by a class.

Generalizations

- “BehavioralFeature” on page 152

Attributes

No additional attributes

Associations

- class : Class [0..1]
  Redefines the corresponding association in Basic. Subsets *RedefinableElement::redefinitionContext*, *NamedElement::namespace* and *Feature::featuringClassifier*.

Constraints

No additional constraints

Semantics

An operation may be owned by and in the namespace of a class that provides the context for its possible redefinition.

11.3.5 Property

A property is a structural feature of a classifier that characterizes instances of the classifier. Constructs::Property merges the definition of *Basic::Property* with *Constructs::StructuralFeature*.

A property related by ownedAttribute to a classifier (other than an association) represents an attribute and might also represent an association end. It relates an instance of the class to a value or set of values of the type of the attribute.

A property related by memberEnd or its specializations to an association represents an end of the association. The type of the property is the type of the end of the association.

Description

Property represents a declared state of one or more instances in terms of a named relationship to a value or values. When a property is an attribute of a classifier, the value or values are related to the instance of the classifier by being held in slots of the instance. When a property is an association end, the value or values are related to the instance or instances at the other end(s) of the association (see semantics of Association).

Property is indirectly a subclass of *Constructs::TypedElement*. The range of valid values represented by the property can be controlled by setting the property's type.

Generalizations

- "StructuralFeature" on page 133

Attributes

- isDerivedUnion : Boolean
  Specifies whether the property is derived as the union of all of the properties that are constrained to subset it. The default value is *false*.
- isReadOnly : Boolean
  This redefines the corresponding attribute in *Basic::Property* and *Abstractions::StructuralFeature*. The default value is *false*.
- isID : Boolean
  True indicates this property can be used to uniquely identify an instance of the containing Class. Default value is *false.*

Associations

- association: Association [0..1]
  References the association of which this property is a member, if any.
- owningAssociation: Association [0..1]
  References the owning association of this property, if any. Subsets *Property::association*, *NamedElement::namespace*, and *Feature::featuringClassifier.*
- redefinedProperty : Property [*]
  References the properties that are redefined by this property. Subsets *RedefinableElement::redefinedElement*.
- subsettedProperty : Property [*]
  References the properties of which this property is constrained to be a subset.
- / opposite : Property [0..1]
  In the case where the property is one navigable end of a binary association with both ends navigable, this gives the other end.
- class : Class [0..1]
  References the Class that owns the Property. Subsets *NamedElement::namespace*, *Feature::featuringClassifier*

Constraints

[1] If this property is owned by a class, associated with a binary association, and the other end of the association is also owned by a class, then opposite gives the other end.

opposite =

if owningAssociation->isEmpty() **and** association.memberEnd->size() = 2 **then**

```
        let otherEnd = (association.memberEnd - self)->any() in
            if otherEnd.owningAssociation->isEmpty() then otherEnd else Set{} endif
    else Set {}
    endif
```

[2] A multiplicity of a composite aggregation must not have an upper bound greater than 1.

```
isComposite implies (upperBound()->isEmpty() or upperBound() <= 1)
```

[3] Subsetting may only occur when the context of the subsetting property conforms to the context of the subsetted property.

```
subsettedProperty->notEmpty() implies
    (subsettingContext()->notEmpty() and subsettingContext()->forAll (sc |
        subsettedProperty->forAll(sp |
            sp.subsettingContext()->exists(c | sc.conformsTo(c)))))
```

[4] A redefined property must be inherited from a more general classifier containing the redefining property.

```
if (redefinedProperty->notEmpty()) then
  (redefinitionContext->notEmpty() and
     redefinedProperty->forAll(rp|
    ((redefinitionContext->collect(fc|
      fc.allParents()))->asSet())->
        collect(c| c.allFeatures())->asSet()->
          includes(rp))
```

[5] A subsetting property may strengthen the type of the subsetted property, and its upper bound may be less.

```
subsettedProperty->forAll(sp |
    type.conformsTo(sp.type) and
        ((upperBound()->notEmpty() and sp.upperBound()->notEmpty()) implies
            upperBound()<=sp.upperBound() ))
```

[6] A derived union is derived.

```
isDerivedUnion implies isDerived
```

[7] A derived union is read only

```
isDerivedUnion implies isReadOnly
```

[8] The value of isComposite is true only if aggregation is composite.

```
isComposite = (self.aggregation = #composite)
```

[9] A Property cannot be subset by a Property with the same name

```
if (self.subsettedProperty->notEmpty()) then
    self.subsettedProperty->forAll(sp | sp.name <> self.name)
```

Additional Operations

[1] The query isConsistentWith() specifies, for any two Properties in a context in which redefinition is possible, whether redefinition would be logically consistent. A redefining property is consistent with a redefined property if the type of the redefining property conforms to the type of the redefined property, and the multiplicity of the redefining property (if specified) is contained in the multiplicity of the redefined property.

```
Property::isConsistentWith(redefinee : RedefinableElement) : Boolean

pre: redefinee.isRedefinitionContextValid(self)
isConsistentWith = redefinee.oclIsKindOf(Property) and
    let prop : Property = redefinee.oclAsType(Property) in
        (prop.type.conformsTo(self.type) and
        ((prop.lowerBound()->notEmpty() and self.lowerBound()->notEmpty()) implies
```

```
        prop.lowerBound() >= self.lowerBound()) and
        ((prop.upperBound()->notEmpty() and self.upperBound()->notEmpty()) implies
            prop.lowerBound() <= self.lowerBound()) and
            and (self.isComposite implies prop.isComposite))
```

[2] The query subsettingContext() gives the context for subsetting a property. It consists, in the case of an attribute, of the corresponding classifier, and in the case of an association end, all of the classifiers at the other ends.

```
Property::subsettingContext() : Set(Type)
subsettingContext =
    if association->notEmpty()
    then association.endType-type
    else if classifier->notEmpty() then Set{classifier} else Set{} endif
    endif
```

[3] The query isNavigable indicates whether it is possible to navigate across the property.

```
Property::isNavigable() : Boolean
IsNavigable = not classifier->isEmpty() or
            association.owningAssociation.navigableOwnedEnd->includes(self)
```

[4] The query isAttribute() is true if the Property is defined as an attribute of some classifier

```
context Property::isAttribute(p : Property) : Boolean
    post: result = Classifier.allInstances->exists(c| c.attribute->includes(p))
```

Semantics

When a property is owned by a classifier other than an association via ownedAttribute, then it represents an *attribute* of the class or data type. When related to an association via memberEnd or one of its specializations, it represents an end of the association. In either case, when instantiated a property represents a value or collection of values associated with an instance of one (or in the case of a ternary or higher-order association, more than one) type. This set of types is called the context for the property; in the case of an attribute the context is the owning classifier, and in the case of an association end the context is the set of types at the other end or ends of the association.

The value or collection of values instantiated for a property in an instance of its context conforms to the property's type. Property inherits from *MultiplicityElement* and thus allows multiplicity bounds to be specified. These bounds constrain the size of the collection. Typically and by default the maximum bound is 1.

Property also inherits the *isUnique* and *isOrdered* meta-attributes. When *isUnique* is *true* (the default) the collection of values may not contain duplicates. When *isOrdered* is *true* (*false* being the default) the collection of values is ordered. In combination these two allow the type of a property to represent a collection in the following way:

Table 11.1 - Collection types for properties

isOrdered	isUnique	Collection type
false	true	Set
true	true	OrderedSet
false	false	Bag
true	false	Sequence

If there is a default specified for a property, this default is evaluated when an instance of the property is created in the absence of a specific setting for the property or a constraint in the model that requires the property to have a specific value. The evaluated default then becomes the initial value (or values) of the property.

If a property is derived, then its value or values can be computed from other information. Actions involving a derived property behave the same as for a nonderived property. Derived properties are often specified to be read-only (i.e., clients cannot directly change values). But where a derived property is changeable, an implementation is expected to make appropriate changes to the model in order for all the constraints to be met, in particular the derivation constraint for the derived property. The derivation for a derived property may be specified by a constraint.

The name and visibility of a property are not required to match those of any property it redefines.

A derived property can redefine one that is not derived. An implementation must ensure that the constraints implied by the derivation are maintained if the property is updated.

If a property has a specified default, and the property redefines another property with a specified default, then the redefining property's default is used in place of the more general default from the redefined property.

If a navigable property is marked as readOnly, then it cannot be updated once it has been assigned an initial value.

A property may be marked as a subset of another, as long as every element in the context of the subsetting property conforms to the corresponding element in the context of the subsetted property. In this case, the collection associated with an instance of the subsetting property must be included in (or the same as) the collection associated with the corresponding instance of the subsetted property.

A property may be marked as being a derived union. This means that the collection of values denoted by the property in some context is derived by being the strict union of all of the values denoted, in the same context, by properties defined to subset it. If the property has a multiplicity upper bound of 1, then this means that the values of all the subsets must be null or the same.

A property may be marked as being (part of) the identifier (if any) for classes of which it is a member. The interpretation of this is left open but this could be mapped to implementations such as primary keys for relational database tables or ID attributes in XML. If multiple properties are marked (possibly in superclasses), then it is the combination of the (property, value) tuples that will logically provide the uniqueness for any instance. Hence there is no need for any specification of order and it is possible for some (but not all) of the property values to be empty. If the property is multivalued then all values are included.

Notation

The following general notation for properties is defined. Note that some specializations of Property may also have additional notational forms. These are covered in the appropriate Notation sub clauses of those classes.

<property> ::= [<visibility>] ['/'] <name> [':' <prop-type>] ['[' <multiplicity> ']'] ['=' <default>]

['{' <prop-modifier > [',' <prop-modifier >] '}']*

where:

- *<visibility>* is the visibility of the property (See "VisibilityKind" on page 89).

  <visibility> ::= '+' | '-'

- '/' signifies that the property is derived.
- *<name>* is the name of the property.
- *<prop-type>* is the name of the Classifier that is the type of the property.

- <*multiplicity*> is the multiplicity of the property. If this term is omitted, it implies a multiplicity of 1 (exactly one). (See “MultiplicityElement” on page 132.)
- <*default*> is an expression that evaluates to the default value or values of the property.
- <*prop-modifier* > indicates a modifier that applies to the property.

<*prop-modifier*> *::= ‘readOnly’* | *‘union’* | *‘subsets‘* <*property-name*> |

‘redefines’ <*property-name*> | *‘ordered’* | *‘unique’* | *‘id’* | <*prop-constraint*>

where:

- *readOnly* means that the property is read only.
- *union* means that the property is a derived union of its subsets.
- *subsets* <*property-name*> means that the property is a proper subset of the property identified by <*property-name*>.
- *redefines* <*property-name*> means that the property redefines an inherited property identified by <*property-name*>.
- *ordered* means that the property is ordered.
- *unique* means that there are no duplicates in a multi-valued property.
- *id* means that the property is part of the identifier for the class.
- <*prop-constraint*> is an expression that specifies a constraint that applies to the property.

All redefinitions shall be made explicit with the use of a {redefines <x>} property string. Redefinition prevents inheritance of a redefined element into the redefinition context thereby making the name of the redefined element available for reuse, either for the redefining element, or for some other.

11.4 Classifiers Diagram

The Classifiers diagram of the Constructs package specifies the concepts Classifier, TypedElement, MultiplicityElement, RedefinableElement, Feature, and StructuralFeature. In each case these concepts are extended and redefined from their corresponding definitions in Basic and Abstractions.

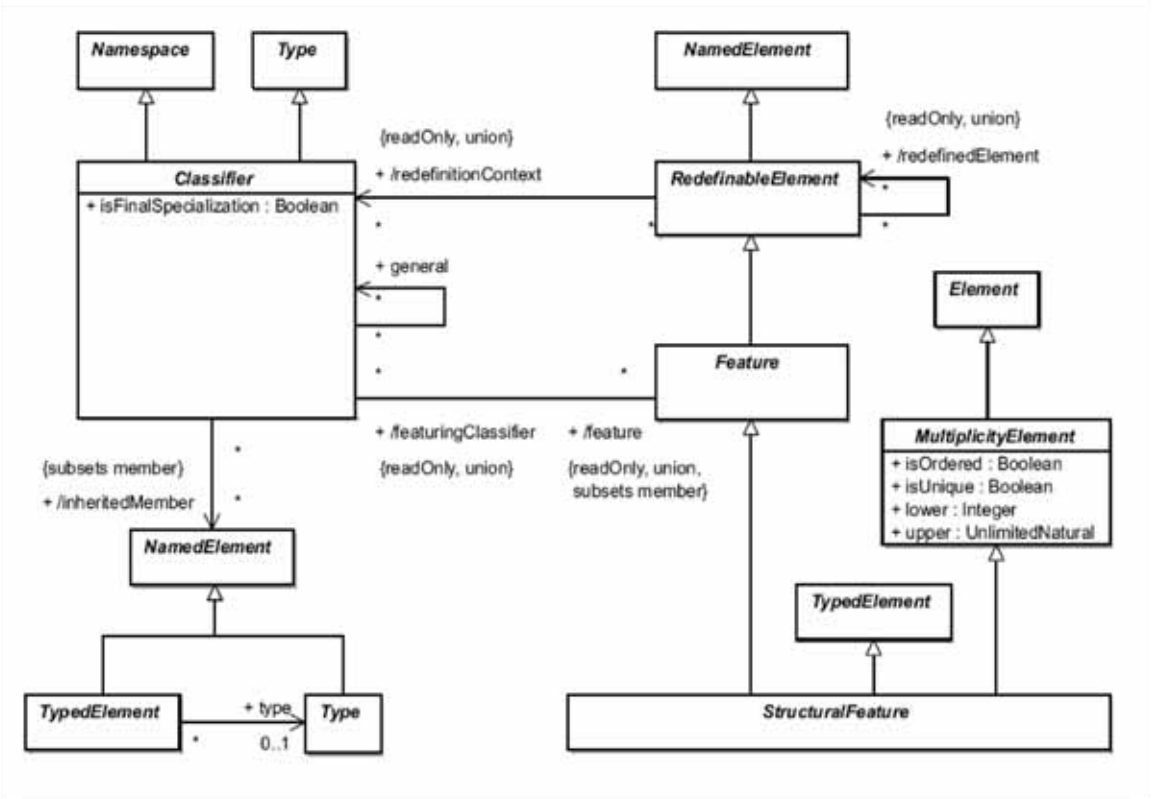


Figure 11.16 - The Classifiers diagram of the Constructs package

11.4.1 Classifier

Description

Constructs::Classifier merges the definitions of Classifier from *Basic* and *Abstractions*. It adds specializations from *Constructs::Namespace* and *Constructs::Type* and the capability to specify that a classifier cannot be specialized by generalization.

Generalizations

- “Type” on page 134
- “Namespace” on page 147

Attributes

- isFinalSpecialization: Boolean

  if *true*, the Classifier cannot be specialized by generalization. Note that this property is preserved through package merge operations; that is, the capability to specialize a Classifier (i.e., isFinalSpecialization =*false*) must be preserved in the resulting Classifier of a package merge operation where a Classifier with isFinalSpecialization =*false* is merged with a matching Classifier with isFinalSpecialization =*true*: the resulting Classifier will have isFinalSpecialization =*false*. Default is *false*.

Associations

- /feature : Feature [*]
  Redefines the corresponding association in *Abstractions*. Subsets *Namespace::member* and is a derived union. Note that there may be members of the Classifier that are of the type Feature but are not included in this association (e.g., inherited features).

Constraints

[1] The parents of a classifier must be non-final.

```
self.parents()->forAll(not isFinalSpecialization)
```

Semantics

No additional semantics

Notation

As defined in Abstractions

11.4.2 Feature

Description

Constructs::Feature reuses the definition of Feature from *Abstractions*. It adds a specialization from *Constructs::RedefinableElement*.

Generalizations

- “RedefinableElement” on page 132

Attributes

No additional attributes

Associations

- /featuringClassifier : Classifier [1..*]
  Redefines the corresponding association in *Abstractions*. This is a derived union.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

As defined in Abstractions

11.4.3 MultiplicityElement

Description

Constructs::MultiplicityElement reuses the definition of MultiplicityElement from *Abstractions*. It adds a specialization from Constructs::Element.

Generalizations

- “Element” on page 106

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

No additional semantics

Notation

As defined in Abstractions

11.4.4 RedefinableElement

Description

Constructs::RedefinableElement reuses the definition of *RedefineableElement* from *Abstractions*. It adds a specialization from *Constructs::NamedElement* and the capability for indicating whether it is possible to further redefine a RedefinableElement.

Generalizations

- “NamedElement” on page 147

Attributes

- isLeaf: Boolean

  Indicates whether it is possible to further redefine a RedefinableElement. If the value is *true*, then it is not possible to further redefine the RedefinableElement. Note that this property is preserved through package merge operations. That is, the capability to redefine a RedefinableElement (i.e., isLeaf=*false*) must be preserved in the resulting RedefinableElement of a package merge operation where a RedefinableElement with isLeaf=*false* is merged with a matching RedefinableElement with isLeaf=*true*: the resulting RedefinableElement will have isLeaf=*false*. Default value is *false*.

Associations

- /redefinedElement: RedefinableElement[*]
  This derived union is redefined from *Abstractions*.
- /redefinitionContext: Classifier[*]
  This derived union is redefined from *Abstractions*.

Constraints

[1] At least one of the redefinition contexts of the redefining element must be a specialization of at least one of the redefinition contexts for each redefined element.

```
self.redefinedElement->forAll(e | self.isRedefinitionContextValid(e))
```

[2] A redefining element must be consistent with each redefined element.

[3] self.redefinedElement->forAll(re | re.isConsistentWith(self))
A redefinable element can only redefine non-leaf redefinable elements

```
self.redefinedElement->forAll(not isLeaf)
```

Additional Operations

[1] The query isConsistentWith() specifies, for any two RedefinableElements in a context in which redefinition is possible, whether redefinition would be logically consistent. By default, this is false; this operation must be overridden for subclasses of RedefinableElement to define the consistency conditions.

```
RedefinableElement::isConsistentWith(redefinee: RedefinableElement): Boolean;
```

pre: redefinee.isRedefinitionContextValid(self)

result = false

[2] The query isRedefinitionContextValid() specifies whether the redefinition contexts of this RedefinableElement are properly related to the redefinition contexts of the specified RedefinableElement to allow this element to redefine the other. By default at least one of the redefinition contexts of this element must be a specialization of at least one of the redefinition contexts of the specified element.

```
RedefinableElement::isRedefinitionContextValid(redefined: RedefinableElement): Boolean;
result = self.redefinitionContext->exists(c | c.allParents()->includes(redefined.redefinitionContext))
```

Semantics

No additional semantics

Notation

As defined in Abstractions

11.4.5 StructuralFeature

Description

Constructs::StructuralFeature reuses the definition of *StructuralFeature* from *Abstractions*. It adds specializations from *Constructs::Feature*, *Constructs::TypedElement*, and *Constructs::MultiplicityElement*.

By specializing *MultiplicityElement*, it supports a multiplicity that specifies valid cardinalities for the set of values associated with an instantiation of the structural feature.

Generalizations

- “Feature” on page 131
- “TypedElement” on page 135
- “MultiplicityElement” on page 132

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

No additional semantics

Notation

As defined in Abstractions

11.4.6 Type

Description

Constructs::Type merges the definitions of *Type* from *Basic* and *Abstractions*. It adds a specialization from *Constructs::NamedElement*.

Generalizations

- “NamedElement” on page 147
- “PackageableElement” on page 149

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

No additional semantics

Notation

As defined in Abstractions

11.4.7 TypedElement

Description

Constructs::TypedElement merges the definitions of *TypedElement* from *Basic* and *Abstractions*. It adds a specialization from *Constructs::NamedElement*.

Generalizations

- “NamedElement” on page 147

Attributes

- type: Classifier [1]
  Redefines the corresponding attributes in both *Basic* and *Abstractions*.

Associations

No additional associations

Constraints

No additional constraints

Semantics

No additional semantics

Notation

As defined in Abstractions

11.5 Constraints Diagram

The Constraints diagram of the Constructs package specifies the Constraint construct and adds features to the Namespace construct.

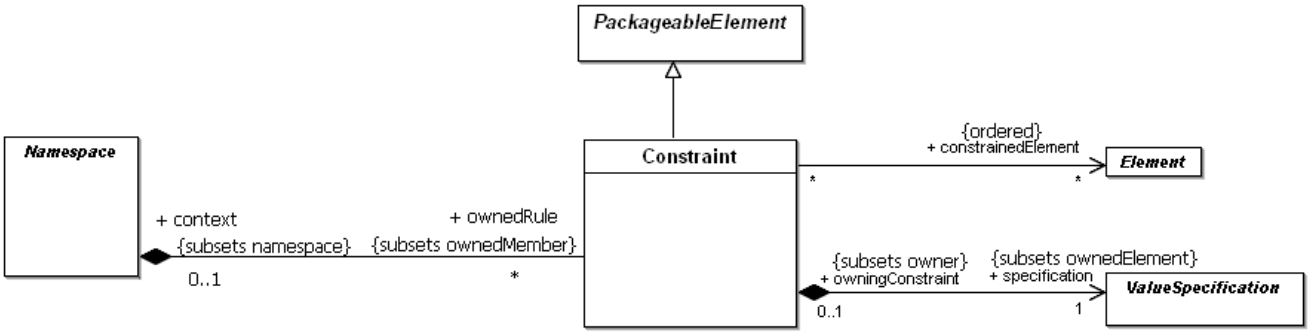


Figure 11.17 - The Constraints diagram of the Constructs package

11.5.1 Constraint

Description

Constructs::Constraint reuses the definition of *Constraint* from *Abstractions::Constraints*. It adds a specialization to *PackageableElement*.

Generalizations

- “PackageableElement” on page 149

Attributes

No additional attributes

Associations

- constrainedElement: Element
  Redefines the corresponding property in *Abstractions*.
- context: Namespace [0..1]
  Specifies the Namespace that is the context for evaluating this constraint. Subsets *NamedElement::namespace*.
- specification: ValueSpecification
  Redefines the corresponding property in *Abstractions*. Subsets *Element.ownedElement*

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

11.5.2 Namespace

Note – (additional properties - see “Namespace” on page 147).

Description

Constructs::Namespace is defined in the *Namespaces* diagram. The Constraints diagram shows the association between Namespace and *Constraint* that represents the ownership of the constraint by a namespace.

Generalizations

- “NamedElement” on page 147

Attributes

No additional attributes

Associations

- ownedRule : Constraint [*]
  Redefines the corresponding property in *Abstractions*. Subsets *Namespace::ownedMember*

Constraints

No additional constraints

Semantics

No additional semantics

11.6 DataTypes Diagram

The DataTypes diagram of the Constructs package specifies the DataType, Enumeration, EnumerationLiteral, and PrimitiveType constructs, and adds features to the Property and Operation constructs. These constructs are used for defining primitive data types (such as Integer and String) and user-defined enumeration data types. The data types are typically used for declaring the types of the class attributes.

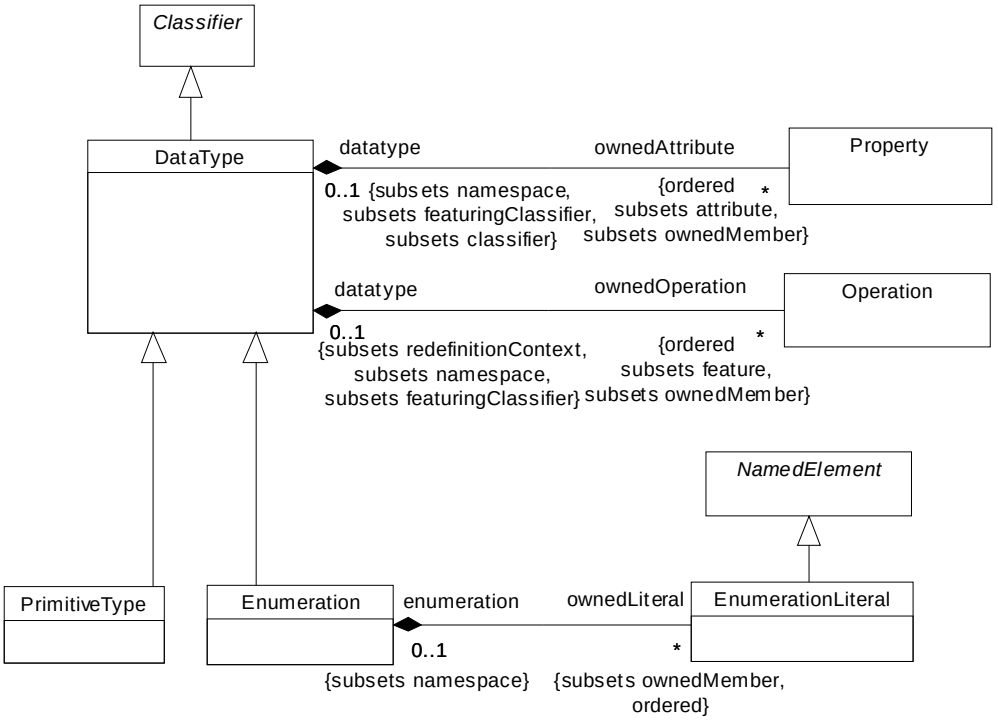


Figure 11.18 - The classes defined in the DataTypes diagram

11.6.1 DataType

Description

A data type is a type whose instances are identified only by their value. A DataType may contain attributes to support the modeling of structured data types.

A typical use of data types would be to represent programming language primitive types or CORBA basic types. For example, integer and string types are often treated as data types.

Generalizations

- “Classifier” on page 130

Attributes

No additional attributes

Associations

- ownedAttribute: Property[*]
  The Attributes owned by the DataType. This is an ordered collection. Subsets *Classifier::attribute* and *Namespace::ownedMember*

- ownedOperation: Operation[*]
  The Operations owned by the DataType. This is an ordered collection. Subsets *Classifier::feature* and *Namespace::ownedMember*

Constraints

No additional constraints

Additional Operations

[1] The *inherit* operation is overriden to exclude redefined properties

```
DataType::inherit(inhs: Set(NamedElement)): Set(NamedElement);
inherit=inhs->excluding(inh | ownedMember->
    select(oclIsKindOf(RedefinableElement))->select(redefinedElement->includes(inh)))
```

Semantics

A data type is a special kind of classifier, similar to a class. It differs from a class in that instances of a data type are identified only by their value.

All copies of an instance of a data type and any instances of that data type with the same value are considered to be equal instances. Instances of a data type that have attributes (i.e., is a structured data type) are considered to be equal if the structure is the same and the values of the corresponding attributes are equal. If a data type has attributes, then instances of that data type will contain attribute values matching the attributes.

Semantic Variation Points

Any restrictions on the capabilities of data types, such as constraining the types of their attributes, is a semantic variation point.

Notation

A data type is shown using the classifier symbol with keyword «dataType» or when it is referenced by e.g., an attribute, shown as a string containing the name of the data type.

Examples

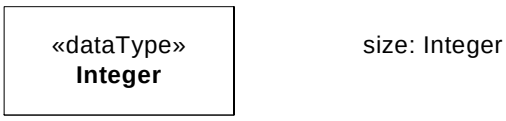


Figure 11.19 - Notation of data type: to the left is an icon denoting a data type and to the right is a reference to a data type that is used in an attribute.

11.6.2 Enumeration

An enumeration is a data type whose values are enumerated in the model as enumeration literals.

Description

Constructs::Enumeration reuses the definition of Enumeration from Basic. It adds a specialization to Constructs::DataType.

Enumeration is a kind of data type, whose instances may be any of a number of predefined enumeration literals.

It is possible to extend the set of applicable enumeration literals in other packages or profiles.

Generalizations

- “DataType” on page 138.

Attributes

No additional attributes

Associations

- ownedLiteral: EnumerationLiteral[*]
  The ordered set of literals for this Enumeration. Subsets *Namespace::ownedMember*

Constraints

No additional constraints

Semantics

The run-time instances of an Enumeration are data values. Each such value corresponds to exactly one EnumerationLiteral.

Notation

An enumeration may be shown using the classifier notation (a rectangle) with the keyword «enumeration». The name of the enumeration is placed in the upper compartment. A compartment listing the attributes for the enumeration is placed below the name compartment. A compartment listing the operations for the enumeration is placed below the attribute compartment. A list of enumeration literals may be placed, one to a line, in the bottom compartment. The attributes and operations compartments may be suppressed, and typically are suppressed if they would be empty.

Examples

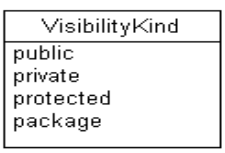


Figure 11.20 - Example of an enumeration

11.6.3 EnumerationLiteral

An enumeration literal is a user-defined data value for an enumeration.

Description

Constructs::EnumerationLiteral reuses the definition of Enumeration from Basic. It adds a specialization to Constructs::NamedElement.

Generalizations

- “NamedElement” on page 147

Attributes

No additional attributes

Associations

- enumeration: Enumeration[0..1]
  The Enumeration that this EnumerationLiteral is a member of. Subsets *NamedElement::namespace*

Constraints

No additional constraints

Semantics

An EnumerationLiteral defines an element of the run-time extension of an enumeration data type.

An EnumerationLiteral has a name that can be used to identify it within its enumeration datatype. The enumeration literal name is scoped within and must be unique within its enumeration. Enumeration literal names are not global and must be qualified for general use.

The run-time values corresponding to enumeration literals can be compared for equality.

Notation

An EnumerationLiteral is typically shown as a name, one to a line, in the compartment of the enumeration notation (see “Enumeration”).

Examples

See “Enumeration”

11.6.4 Operation

Note – (additional properties - see “Operation” on page 153)

Description

Constructs::Operation is defined in the *Operations* diagram. The DataTypes diagram shows the association between Operation and *DataType* that represents the ownership of the operation by a data type.

Generalizations

- “BehavioralFeature” on page 152

Attributes

No additional attributes

Associations

- datatype : DataType [0..1]
  The DataType that owns this Operation. Subsets *NamedElement::namespace*, *Feature::featuringClassifier*, and *RedefinableElement::redefinitionContext*.

Constraints

No additional constraints

Semantics

An operation may be owned by and in the namespace of a datatype that provides the context for its possible redefinition.

11.6.5 PrimitiveType

A primitive type defines a predefined data type, without any relevant substructure (i.e., it has no parts in the context of UML). A primitive datatype may have an algebra and operations defined outside of UML, for example, mathematically.

Description

Constructs::PrimitiveType reuses the definition of *PrimitiveType* from *Basic*. It adds a specialization to *Constructs::DataType*.

The instances of primitive type used in UML itself include Boolean, Integer, UnlimitedNatural, and String (see Clause 12, “Core::PrimitiveTypes”).

Generalizations

- “DataType” on page 138

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

The run-time instances of a primitive type are data values. The values are in many-to-one correspondence to mathematical elements defined outside of UML (for example, the various integers).

Instances of primitive types do not have identity. If two instances have the same representation, then they are indistinguishable.

Notation

A primitive type has the keyword «primitive» above or before the name of the primitive type.

Instances of the predefined primitive types (see Clause 12, “Core::PrimitiveTypes”) may be denoted with the same notation as provided for references to such instances (see the subtypes of “ValueSpecification”).

Examples

See Clause 12, “Core::PrimitiveTypes” for examples

11.6.6 Property

Note – (additional properties - see “Property” on page 124)

Description

Constructs::Property is defined in the *Classes* diagram. The DataTypes diagram shows the association between Property and *DataType* that represents the ownership of the property by a data type.

Generalizations

- “StructuralFeature” on page 133

Attributes

No additional attributes

Associations

- datatype : DataType [0..1]
  The DataType that owns this Property. Subsets *NamedElement::namespace*, *Feature::featuringClassifier*, and *Property::classifier.*

Constraints

No additional constraints

Semantics

A property may be owned by and in the namespace of a datatype.

11.7 Namespaces Diagram

The Namespaces diagram of the Constructs package specifies Namespace and related constructs. It specifies how named elements are defined as members of namespaces, and also specifies the general capability for any namespace to import all or individual members of packages.

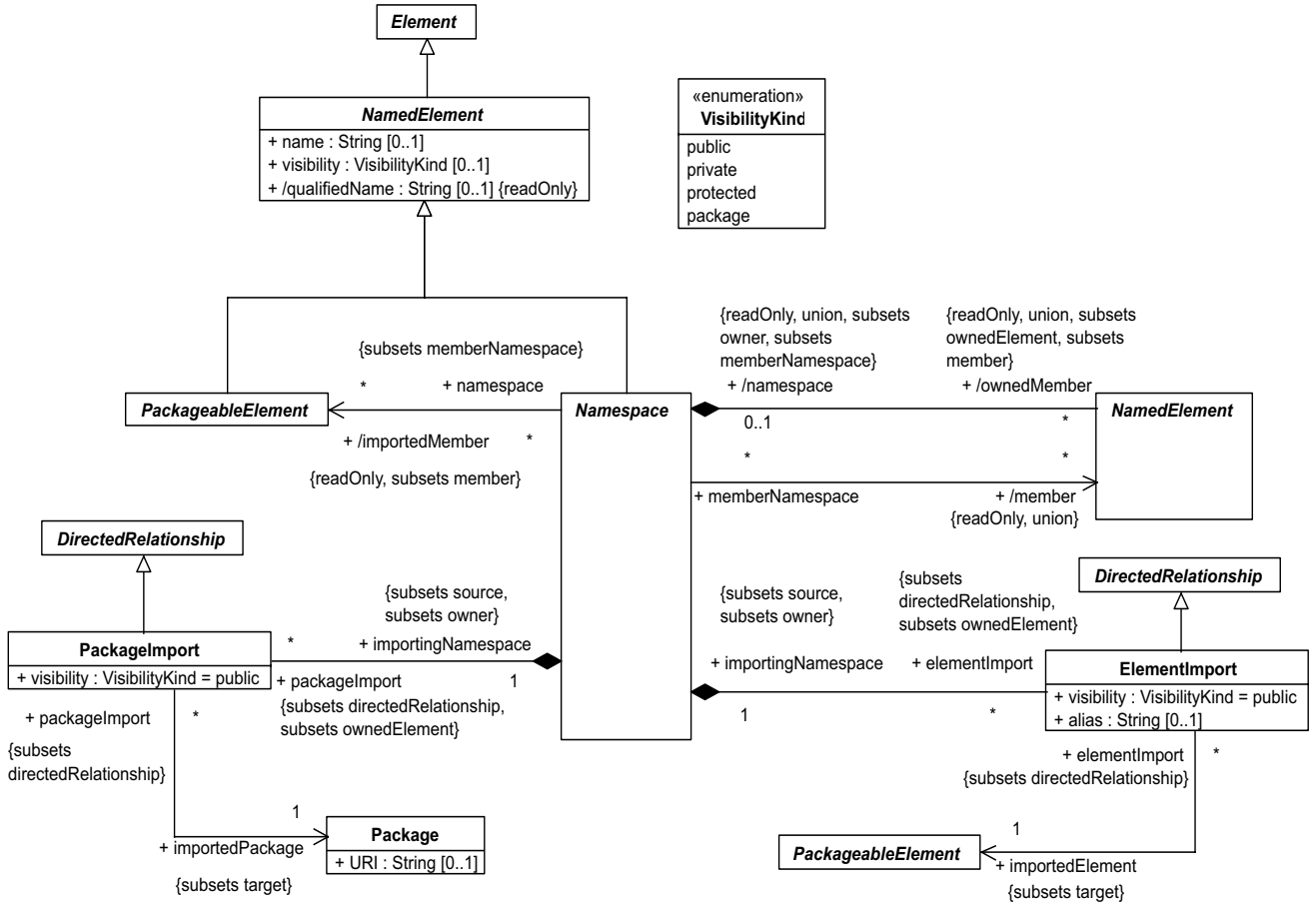


Figure 11.21 - The Namespaces diagram of the Constructs package

11.7.1 ElementImport

An element import identifies an element in another package, and allows the element to be referenced using its name without a qualifier.

Description

An element import is defined as a directed relationship between an importing namespace and a packageable element. The name of the packageable element or its alias is to be added to the namespace of the importing namespace. It is also possible to control whether the imported element can be further imported.

Generalizations

- “DirectedRelationship” on page 106

Attributes

- visibility: VisibilityKind
  Specifies the visibility of the imported PackageableElement within the importing Package. The default visibility is the same as that of the imported element. If the imported element does not have a visibility, it is possible to add visibility to the element import; default value is *public*.
- alias: String [0..1]
  Specifies the name that should be added to the namespace of the importing Package in lieu of the name of the imported PackagableElement. The aliased name must not clash with any other member name in the importing Package. By default, no alias is used.

Associations

- importedElement: PackageableElement [1]
  Specifies the PackageableElement whose name is to be added to a Namespace. Subsets *DirectedRelationship::target*
- importingNamespace: Namespace [1]
  Specifies the Namespace that imports a PackageableElement from another Package. Subsets *DirectedRelationship::source* and *Element::owner*

Constraints

[1] The visibility of an ElementImport is either public or private.

self.visibility = #public **or** self.visibility = #private

[2] An importedElement has either public visibility or no visibility at all.

self.importedElement.visibility.notEmpty() **implies** self.importedElement.visibility = #public

Additional Operations

[1] The query getName() returns the name under which the imported PackageableElement will be known in the importing namespace.

```
ElementImport::getName(): String;
getName =
    if self.alias->notEmpty() then
        self.alias
    else
        self.importedElement.name
    endif
```

Semantics

An element import adds the name of a packageable element from a package to the importing namespace. It works by reference, which means that it is not possible to add features to the element import itself, but it is possible to modify the referenced element in the namespace from which it was imported. An element import is used to selectively import individual elements without relying on a package import.

In case of a nameclash with an outer name (an element that is defined in an enclosing namespace is available using its unqualified name in enclosed namespaces) in the importing namespace, the outer name is hidden by an element import, and the unqualified name refers to the imported element. The outer name can be accessed using its qualified name.

If more than one element with the same name would be imported to a namespace as a consequence of element imports or package imports, the elements are not added to the importing namespace and the names of those elements must be qualified in order to be used in that namespace. If the name of an imported element is the same as the name of an element owned by the importing namespace, that element is not added to the importing namespace and the name of that element

must be qualified in order to be used. If the name of an imported element is the same as the name of an element owned by the importing namespace, the name of the imported element must be qualified in order to be used and is not added to the importing namespace.

An imported element can be further imported by other namespaces using either element or package imports.

The visibility of the ElementImport may be either the same or more restricted than that of the imported element.

Notation

An element import is shown using a dashed arrow with an open arrowhead from the importing namespace to the imported element. The keyword «import» is shown near the dashed arrow if the visibility is public, otherwise the keyword «access» is shown to indicate private visibility.

If an element import has an alias, this is used in lieu of the name of the imported element. The aliased name may be shown after or below the keyword «import».

Presentation Options

If the imported element is a package, the keyword may optionally be preceded by element (i.e., «element import»).

As an alternative to the dashed arrow, it is possible to show an element import by having a text that uniquely identifies the imported element within curly brackets either below or after the name of the namespace. The textual syntax is then:

‘{element import ‘ <qualifiedName> ‘} | ‘{element access ‘ <qualifiedName> ‘}’

Optionally, the aliased name may be shown as well:

‘{element import ‘ <qualifiedName> ‘as’ <alias> ‘} | ‘{element access ‘ <qualifiedName> ‘as’ <alias> ‘}’

Examples

The element import that is shown in Figure 11.22 allows elements in the package Program to refer to the type Time in Types without qualification. However, they still need to refer explicitly to Types::Integer, since this element is not imported. Type String can be used in the Program package but cannot be further imported from Program to other packages.

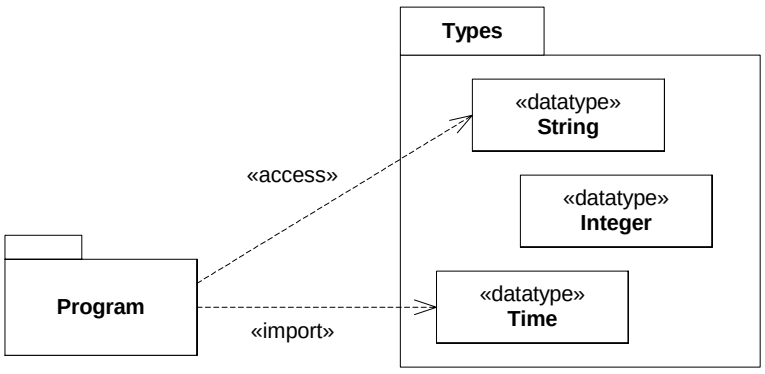


Figure 11.22 - Example of element import

In Figure 11.23, the element import is combined with aliasing, meaning that the type Types::Real will be referred to as Double in the package Shapes.

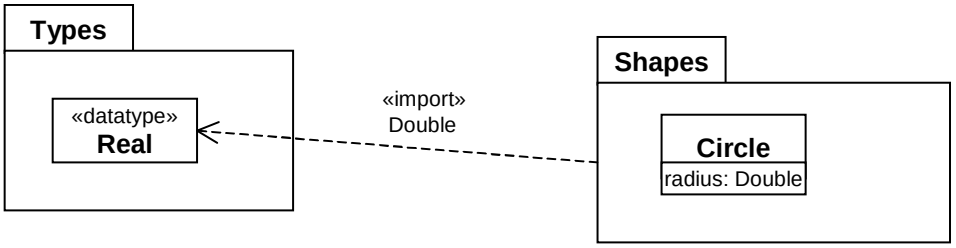


Figure 11.23 - Example of element import with aliasing

11.7.2 NamedElement

Description

Constructs::NamedElement reuses the definition of *NamedElement* from *Abstractions::Visibilitites*. It adds specializations from *Constructs::Element* and *Basic::NamedElement*.

Generalizations

- “Element” on page 106

Attributes

- name: String [0..1]
  The name of the NamedElement.

Associations

- namespace: NamedElement [0..1]
  The Namespace that owns this NamedElement. Redefines the corresponding property from *Abstractions::Namespaces::NamedElement*.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

11.7.3 Namespace

Description

Constructs::Namespace reuses the definition of *Abstractions::Constraints::Namespace*.

A namespace has the ability to import either individual members or all members of a package, thereby making it possible to refer to those named elements without qualification in the importing namespace. In the case of conflicts, it is necessary to use qualified names or aliases to disambiguate the referenced elements.

Generalizations

- "NamedElement" on page 147

Attributes

No additional attributes

Associations

- elementImport: ElementImport [*]
  References the ElementImports owned by the Namespace.Subsets *Element::ownedElement.*
- /importedMember: PackageableElement [*]
  References the PackageableElements that are members of this Namespace as a result of either PackageImports or ElementImports. Subsets *Namespace::member.*
- /member: NamedElement [*]
  Redefines the corresponding property of *Abstractions::Namespaces::Namespace*.
- /ownedMember: NamedElement [*]
  Redefines the corresponding property of *Abstractions::Namespaces::Namespace*.
- packageImport: PackageImport [*]
  References the PackageImports owned by the Namespace. Subsets *Element::ownedElement.*

Constraints

[1] The importedMember property is derived from the ElementImports and the PackageImports.

```
importedMember = self.elementImport.importedElement.asSet()->union(self.packageImport.importedPackage->collect(p |
p.visibleMembers()))
```

Additional operations

[1] The query getNamesOfMember() is overridden to take account of importing. It gives back the set of names that an element would have in an importing namespace, either because it is owned; or if not owned, then imported individually; or if not individually, then from a package.

```
Namespace::getNamesOfMember(element: NamedElement): Set(String);
getNamesOfMember=
    if self.ownedMember ->includes(element)
        then Set{}->include(element.name)
    else let elementImports: ElementImport = self.elementImport->select(ei | ei.importedElement = element) in
        if elementImports->notEmpty()
            then elementImports->collect(el | el.getName())
        else
            self.packageImport->select(pi | pi.importedPackage.visibleMembers()->includes(element))->
                collect(pi | pi.importedPackage.getNamesOfMember(element))
        endif
    endif
```

[2] The query importMembers() defines which of a set of PackageableElements are actually imported into the namespace. This excludes hidden ones, i.e., those which have names that conflict with names of owned members, and also excludes elements that would have the same name when imported.

Namespace::importMembers(imps: Set(PackageableElement)): Set(PackageableElement);
importMembers = self.excludeCollisions(imps)->select(imp | self.ownedMember->forAll(mem |
mem.imp.isDistinguishableFrom(mem, self)))

[3] The query excludeCollisions() excludes from a set of PackageableElements any that would not be distinguishable from each other in this namespace.

Namespace::excludeCollisions(imps: Set(PackageableElements)): Set(PackageableElements);
excludeCollisions = imps->reject(imp1 | imps.exists(imp2 | not imp1.isDistinguishableFrom(imp2, self)))

Semantics

No additional semantics

Notation

No additional notation

11.7.4 PackageableElement

A packageable element indicates a named element that may be owned directly by a package.

Description

A packageable element indicates a named element that may be owned directly by a package.

Generalizations

- “NamedElement” on page 147

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

11.7.5 PackageImport

A package import is a relationship that allows the use of unqualified names to refer to package members from other namespaces.

Description

A package import is defined as a directed relationship that identifies a package whose members are to be imported by a namespace.

Generalizations

- “DirectedRelationship” on page 106

Attributes

- visibility: VisibilityKind
  Specifies the visibility of the imported PackageableElements within the importing Namespace, i.e., whether imported elements will in turn be visible to other packages that use that importingPackage as an importedPackage. If the PackageImport is public, the imported elements will be visible outside the package, while if it is private they will not. By default, the value of visibility is *public*.

Associations

- importedPackage: Package [1]
  Specifies the Package whose members are imported into a Namespace. Subsets *DirectedRelationship::target*
- importingNamespace: Namespace [1]
  Specifies the Namespace that imports the members from a Package. Subsets *DirectedRelationship::source* and *Element::owner*

Constraints

[1] The visibility of a PackageImport is either public or private.

self.visibility = #public **or** self.visibility = #private

Semantics

A package import is a relationship between an importing namespace and a package, indicating that the importing namespace adds the names of the members of the package to its own namespace. Conceptually, a package import is equivalent to having an element import to each individual member of the imported namespace, unless there is already a separately-defined element import.

Notation

A package import is shown using a dashed arrow with an open arrowhead from the importing package to the imported package. A keyword is shown near the dashed arrow to identify which kind of package import that is intended. The predefined keywords are «import» for a public package import, and «access» for a private package import.

Presentation options

As an alternative to the dashed arrow, it is possible to show a package import by having a text that uniquely identifies the imported package within curly brackets either below or after the name of the namespace. The textual syntax is then:

‘{‘import ‘ <qualifiedName> ‘}’ | ‘{access ‘ <qualifiedName> ‘}’

Examples

In Figure 11.24, a number of package imports are shown. The elements in Types are imported to ShoppingCart, and then further imported WebShop. However, the elements of Auxiliary are only accessed from ShoppingCart, and cannot be referenced using unqualified names from WebShop.

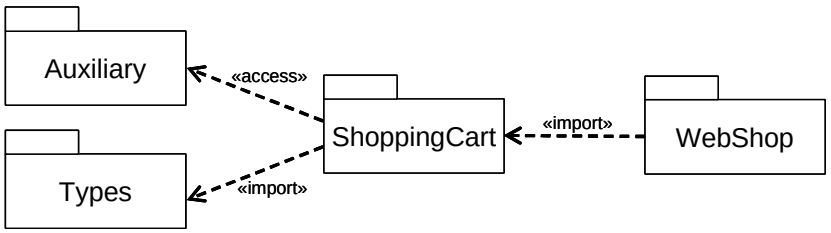


Figure 11.24 - Examples of public and private package imports

11.8 Operations Diagram

The Operations diagram of the Constructs package specifies the BehavioralFeature, Operation, and Parameter constructs.

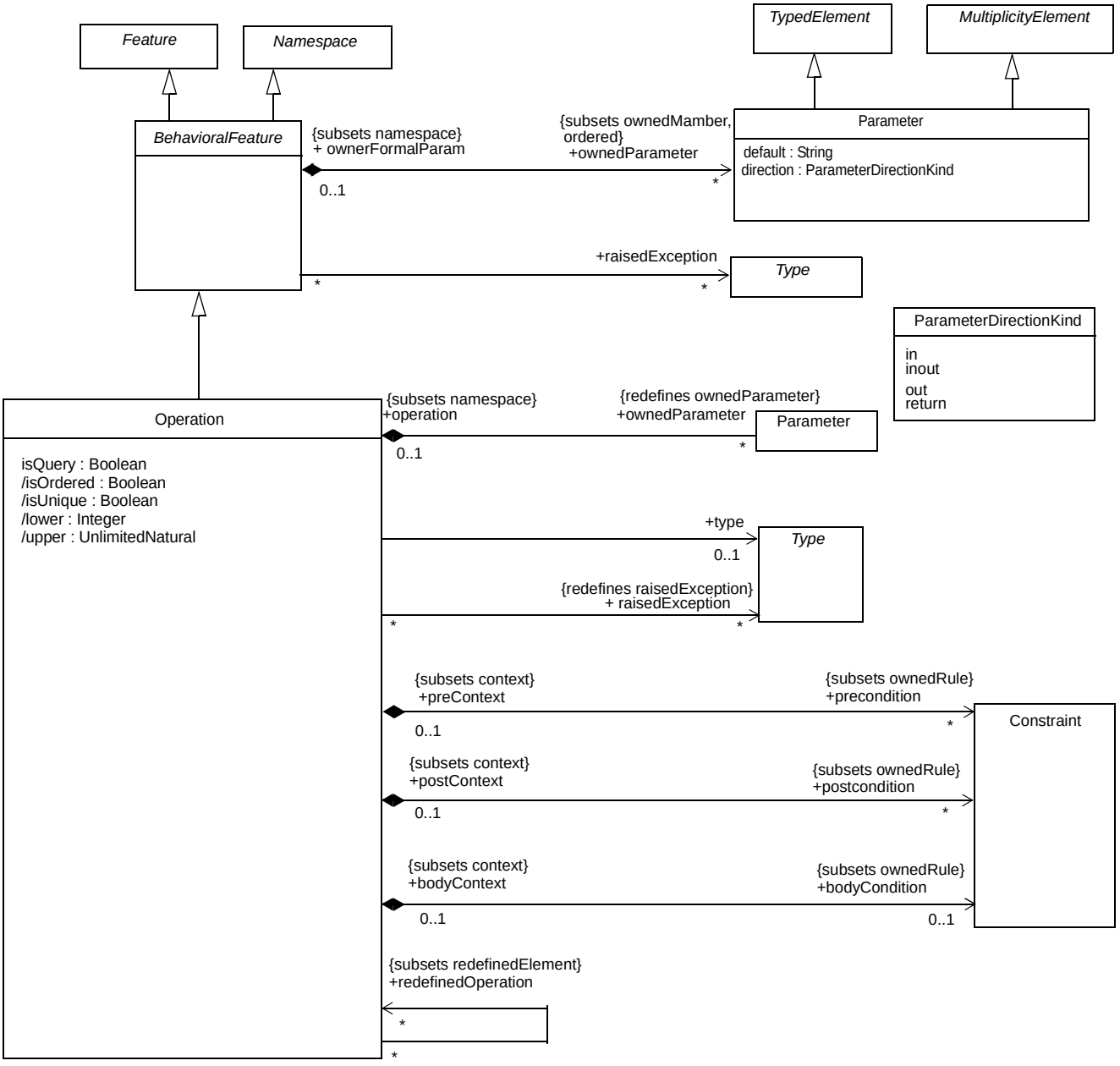


Figure 11.25 - The Operations diagram of the Constructs package

11.8.1 BehavioralFeature

Description

Constructs::BehavioralFeature reuses the definition of *BehavioralFeature* from *Abstractions::BehavioralFeatures*. It adds specializations to *Constructs::Namespace* and *Constructs::Feature*.

Generalizations

- “Feature” on page 131
- “Namespace” on page 147

Attributes

No additional attributes

Associations

- ownedParameter: Parameter[*]
  Specifies the ordered set of formal parameters of this BehavioralFeature. Subsets *Namespace::ownedMember*.
- raisedException: Type[*]
  References the Types representing exceptions that may be raised during an invocation of this feature.

Constraints

No additional constraints

Additional Operations

[1] The query isDistinguishableFrom() determines whether two BehavioralFeatures may coexist in the same Namespace. It specifies that they have to have different signatures.

```
BehavioralFeature::isDistinguishableFrom(n: NamedElement, ns: Namespace): Boolean;
isDistinguishableFrom =
    if n.oclIsKindOf(BehavioralFeature)
    then
        if ns.getNamesOfMember(self)->intersection(ns.getNamesOfMember(n))->notEmpty()
        then Set{}->include(self)->include(n)->isUnique( bf | bf.ownedParameter->collect(type))
        else true
        endif
    else true
    endif
```

Semantics

The list of owned parameters describes the order, type, and direction of arguments that can be given when the BehavioralFeature is invoked or which are returned when the BehavioralFeature terminates.

The owned parameters with direction in or inout define the type, and number, of arguments that must be provided when invoking the BehavioralFeature. An owned parameter with direction out, inout, or return defines the type of the argument that will be returned from a successful invocation. A BehavioralFeature may raise an exception during its invocation.

Notation

No additional notation

11.8.2 Operation

An operation is a behavioral feature of a classifier that specifies the name, type, parameters, and constraints for invoking an associated behavior.

Description

Constructs::Operation reuses the definition of *Operation* from *Basic*. It adds a specialization to *Constructs::BehavioralFeature*.

The specification of an operation defines what service it provides, not how this is done, and can include a list of pre- and postconditions.

Generalizations

- “BehavioralFeature” on page 152

Attributes

- /isOrdered : Boolean
  Redefines the corresponding property from *Basic* to derive this information from the return result for this Operation.
- isQuery : Boolean
  Specifies whether an execution of the Operation leaves the state of the system unchanged (isQuery=true) or whether side effects may occur (isQuery=false). The default value is false.
- /isUnique : Boolean
  Redefines the corresponding property from *Basic* to derive this information from the return result for this Operation.
- /lower : Integer[0..1]
  Redefines the corresponding property from *Basic* to derive this information from the return result for this Operation.
- /upper : UnlimitedNatural[0..1]
  Redefines the corresponding property from *Basic* to derive this information from the return result for this Operation.

Associations

- bodyCondition: Constraint[0..1]
  An optional Constraint on the result values of an invocation of this Operation. Subsets *Namespace.ownedRule*
- postcondition: Constraint[*]
  An optional set of Constraints specifying the state of the system when the Operation is completed. Subsets *Namespace.ownedRule*
- precondition: Constraint[*]
  An optional set of Constraints on the state of the system when the Operation is invoked. Subsets *Namespace.ownedRule*
- raisedException: Type[*]
  References the Types representing exceptions that may be raised during an invocation of this operation. Redefines *Basic::Operation.raisedException* and *BehavioralFeature.raisedException.*
- redefinedOperation: Operation[*]
  References the Operations that are redefined by this Operation. Subsets *RedefinableElement.redefinedElement.*
- /type: Type[0..1]
  Redefines the corresponding property from *Basic* to derive this information from the return result for this Operation.

Constraints

[1] n operation can have at most one return parameter (i.e., an owned parameter with the direction set to ‘return’).

```
ownedParameter->select(par | par.direction = #return)->size() <= 1
```

[2] If this operation has a return parameter, isOrdered equals the value of isOrdered for that parameter. Otherwise isOrdered is false.

isOrdered = **if** returnResult()->notEmpty() **then** returnResult()->any().isOrdered **else** false **endif**

[3] If this operation has a return parameter, isUnique equals the value of isUnique for that parameter. Otherwise isUnique is true.

isUnique = **if** returnResult()->notEmpty() **then** returnResult()->any().isUnique **else** true **endif**

[4] If this operation has a return parameter, lower equals the value of lower for that parameter. Otherwise lower is not defined.

lower = **if** returnResult()->notEmpty() **then** returnResult()->any().lower **else** Set{} **endif**

[5] If this operation has a return parameter, upper equals the value of upper for that parameter. Otherwise upper is not defined.

upper = **if** returnResult()->notEmpty() **then** returnResult()->any().upper **else** Set{} **endif**

[6] If this operation has a return parameter, type equals the value of type for that parameter. Otherwise type is not defined.

type = **if** returnResult()->notEmpty() **then** returnResult()->any().type **else** Set{} **endif**

[7] A bodyCondition can only be specified for a query operation.

bodyCondition->notEmpty() **implies** isQuery

Additional Operations

[1] The query isConsistentWith() specifies, for any two Operations in a context in which redefinition is possible, whether redefinition would be logically consistent. A redefining operation is consistent with a redefined operation if it has the same number of owned parameters, and the type of each owned parameter conforms to the type of the corresponding redefined parameter.

A redefining operation is consistent with a redefined operation if it has the same number of formal parameters, the same number of return results, and the type of each formal parameter and return result conforms to the type of the corresponding redefined parameter or return result.

```
Operation::isConsistentWith(redefinee: RedefinableElement): Boolean;
pre: redefinee.isRedefinitionContextValid(self)
result = (redefinee.oclIsKindOf(Operation) and
        let op: Operation = redefinee.oclAsType(Operation) in
        self.ownedParameter->size() = op.ownedParameter->size() and
        Sequence{1..self.ownedParameter->size()}->
            forAll(i | op.ownedParameter->at(1).type.conformsTo(self.ownedParameter->at(1).type))
```

[2] The query returnResult() returns the set containing the return parameter of the Operation if one exists, otherwise, it returns an empty set

```
Operation::returnResult() : Set(Parameter);
returnResult = ownedParameter->select (par | par.direction = #return)
```

Semantics

An operation is invoked on an instance of the classifier for which the operation is a feature. A static operation is invoked on the classifier owning the operation, hence it can be invoked without an instance.

The preconditions for an operation define conditions that must be true when the operation is invoked. These preconditions may be assumed by an implementation of this operation.

The postconditions for an operation define conditions that will be true when the invocation of the operation completes successfully, assuming the preconditions were satisfied. These postconditions must be satisfied by any implementation of the operation.

The bodyCondition for an operation constrains the return result. The bodyCondition differs from postconditions in that the bodyCondition may be overridden when an operation is redefined, whereas postconditions can only be added during redefinition.

An operation may raise an exception during its invocation. When an exception is raised, it should not be assumed that the postconditions or bodyCondition of the operation are satisfied.

An operation may be redefined in a specialization of the featured classifier. This redefinition may specialize the types of the formal parameters or return results, add new preconditions or postconditions, add new raised exceptions, or otherwise refine the specification of the operation.

Each operation states whether or not its application will modify the state of the instance or any other element in the model (isQuery).

Semantic Variation Points

The behavior of an invocation of an operation when a precondition is not satisfied is a semantic variation point.

When operations are redefined in a specialization, rules regarding invariance, covariance, or contravariance of types and preconditions determine whether the specialized classifier is substitutable for its more general parent. Such rules constitute semantic variation points with respect to redefinition of operations.

Notation

If shown in a diagram, an operation is shown as a text string of the form:

[<visibility>] <name> '(' [<parameter-list>] ')' [':' [<return-type>] ['[' <multiplicity> ']']
['{' <oper-property> [',' <oper-property>] '}']]*

where:

- <*visibility*> is the visibility of the operation (See "VisibilityKind" on page 89.)
  <*visibility*> *::=* '+' | '-'
- <*name*> is the name of the operation.
- <*return-type*> is the type of the return result parameter if the operation has one defined.
- <*multiplicity*> is the multiplicity of the return type. (See "MultiplicityElement" on page 132).
- <*oper-property*> indicates the properties of the operation

  <*oper-property*> *::= 'redefines'* <*oper-name*> | *'query'* | *'ordered'* | *'unique'* | <*oper-constraint*>

  where:

  - *redefines* <*oper-name*> means that the operation redefines an inherited operation identified by <*oper-name*>.
  - *query* means that the operation does not change the state of the system.
  - *ordered* means that the values of the return parameter are ordered.
  - *unique* means that the values returned by the parameter have no duplicates.
  - <*oper-constraint*> is a constraint that applies to the operation.

- <*parameter-list*> is a list of parameters of the operation in the following format:
  <*parameter-list*> *::= <parameter> [',' <parameter>]**
  <*parameter*> *::= [<direction>] <parameter-name> ':' <type-expression>*
  ['['<multiplicity>']'] ['=' <default>] ['{' <parm-property> [',' <parm-property>] '}']*

  where:
  - <*direction*> *::= 'in'* | *'out'* | *'inout'* (defaults to '*in*' if omitted).
  - <*parameter-name*> is the name of the parameter.
  - <*type-expression*> is an expression that specifies the type of the parameter.
  - <*multiplicity*> is the multiplicity of the parameter. (See "MultiplicityElement" on page 65.)
  - <*default*> is an expression that defines the value specification for the default value of the parameter.
  - <*parm-property*> indicates additional property values that apply to the parameter.

Presentation Options

The parameter list can be suppressed. The return result of the operation can be expressed as a return parameter, or as the type of the operation. For example:

```
toString(return : String)
```

means the same thing as

```
toString() : String
```

Style Guidelines

An operation name typically begins with a lowercase letter.

Examples

```
display ()
-hide ()
+createWindow (location: Coordinates, container: Container [0..1]): Window
+toString (): String
```

11.8.3 Parameter

A parameter is a specification of an argument used to pass information into or out of an invocation of a behavioral feature.

Description

Constructs::Parameter merges the definitions of *Parameter* from *Basic* and *Abstractions::BehavioralFeatures*. It adds specializations to *TypedElement* and *MultiplicityElement*.

A parameter is a kind of typed element in order to allow the specification of an optional multiplicity on parameters. In addition, it supports the specification of an optional default value.

Generalizations

- “TypedElement” on page 135
- “MultiplicityElement” on page 132

Attributes

- default: String [0..1]
  Specifies a String that represents a value to be used when no argument is supplied for the Parameter.
- direction: ParameterDirectionKind [1]
  Indicates whether a parameter is being sent into or out of a behavioral element. The default value is *in*.

Associations

- /operation: Operation[0..1]
  References the Operation for which this is a formal parameter. Subsets *NamedElement::namespace* and redefines *Basic::Parameter::operation*.

Constraints

No additional constraints

Semantics

A parameter specifies how arguments are passed into or out of an invocation of a behavioral feature like an operation. The type and multiplicity of a parameter restrict what values can be passed, how many, and whether the values are ordered.

If a default is specified for a parameter, then it is evaluated at invocation time and used as the argument for this parameter if and only if no argument is supplied at invocation of the behavioral feature.

Notation

See Operation

11.8.4 ParameterDirectionKind

Parameter direction kind is an enumeration type that defines literals used to specify direction of parameters.

Generalizations

- none

Description

ParameterDirectionKind is an enumeration of the following literal values:

- in — Indicates that parameter values are passed into the behavioral element by the caller.
- inout — Indicates that parameter values are passed into a behavioral element by the caller and then back out to the caller from the behavioral element.
- out — Indicates that parameter values are passed from a behavioral element out to the caller.
- return — Indicates that parameter values are passed as return values from a behavioral element back to the caller.

11.9 Packages Diagram

The Packages diagram of the Constructs package specifies the Package and PackageMerge constructs.

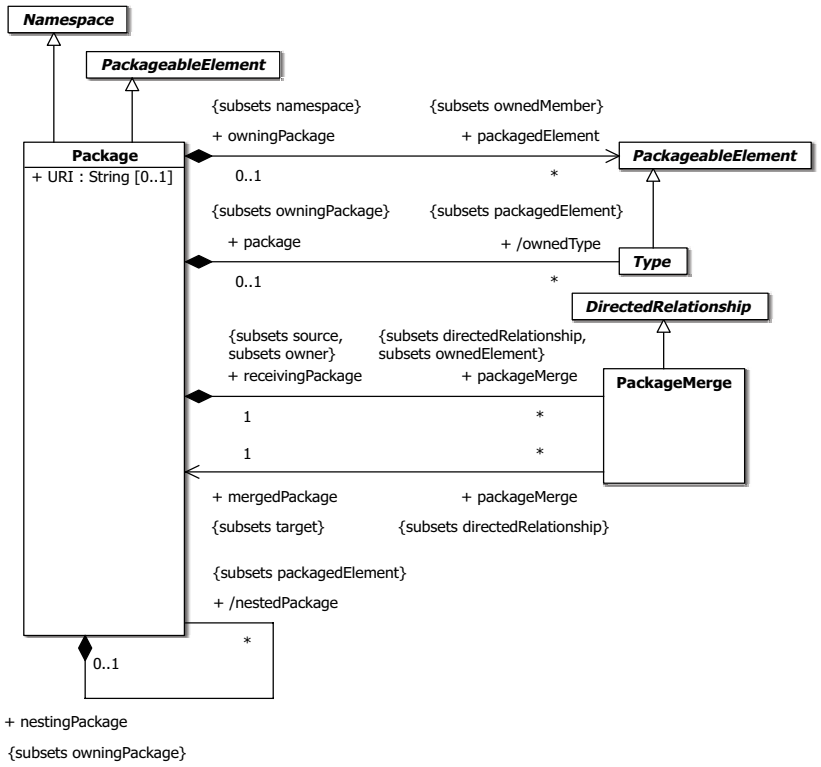


Figure 11.26 - The Packages diagram of the Constructs package

11.9.1 Type

Note – (additional properties - see “Type” on page 134).

Description

Constructs::Type is defined in the *Classifiers* diagram. The Packages diagram adds the association between Type and *Package* that represents the ownership of the type by a package.

Generalizations

- “NamedElement” on page 147
- “PackageableElement” on page 149

Attributes

No additional attributes

Associations

- package: Package [0..1]
  Specifies the owning package of this classifier, if any. Subsets *Package::owningPackage*.

Constraints

No additional constraints

Semantics

No additional semantics

11.9.2 Package

A package is used to group elements, and provides a namespace for the grouped elements.

Description

A package is a namespace for its members, and may contain other packages. Only packageable elements can be owned members of a package. By virtue of being a namespace, a package can import either individual members of other packages, or all the members of other packages.

In addition a package can be merged with other packages.

Generalizations

- “PackageableElement” on page 149
- “Namespace” on page 147

Attributes

- URI: String [0..1] {id}
  Provides an identifier for the package that can be used for many purposes. A URI is the universally unique identification of the package following the IETF URI specification, RFC 2396 http://www.ietf.org/rfc/rfc2396.txt and it must comply with those syntax rules.

Associations

- /nestedPackage: Package [*]
  References the owned members that are Packages. Subsets *Package::ownedMember* and redefines *Basic::Package::nestedPackage*.
- ownedMember: PackageableElement [*]
  Specifies the members that are owned by this Package. Redefines *Namespace::ownedMember.*
- ownedType: Type [*]
  References the owned members that are Types. Subsets *Package::ownedMember* and redefines *Basic::Package::ownedType*.
- package: Package [0..1]
  References the owning package of a package. Subsets *NamedElement::namespace* and redefines *Basic::Package::nestingPackage*.

- packageMerge: Package [*]
  References the PackageMerges that are owned by this Package. Subsets *Element::ownedElement*.

Constraints

[1] If an element that is owned by a package has visibility, it is public or private.

self.ownedElements->forAll(e | e.visibility->notEmpty() **implies** e.visibility = #public **or** e.visibility = #private)

Additional Operations

[1] The query mustBeOwned() indicates whether elements of this type must have an owner.

Package::mustBeOwned() : Boolean
mustBeOwned = false

[2] The query visibleMembers() defines which members of a Package can be accessed outside it.

Package::visibleMembers() : Set(PackageableElement);
visibleMembers = member->select(m | self.makesVisible(m))

[3] The query makesVisible() defines whether a Package makes an element visible outside itself. Elements with no visibility and elements with public visibility are made visible.

```
Package::makesVisible(el: Namespaces::NamedElement) : Boolean;
pre: self.member->includes(el)
makesVisible =
    -- the element is in the package
    (ownedMember->includes(el)) or
    -- it is imported individually with public visibility
    (elementImport->
        select(ei|ei.visibility = #public)->
            collect(ei|ei.importedElement)->includes(el)) or
    -- it is imported through a package with public visibility
    (packageImport->
        select(pi|pi.visibility = #public)->
        collect(pi|
            pi.importedPackage.member->includes(el))->notEmpty())
```

Semantics

A package is a namespace and is also a packageable element that can be contained in other packages.

The elements that can be referred to using non-qualified names within a package are owned elements, imported elements, and elements in enclosing (outer) namespaces. Owned and imported elements may each have a visibility that determines whether they are available outside the package.

A package owns its owned members, with the implication that if a package is removed from a model, so are the elements owned by the package.

The public contents of a package are always accessible outside the package through the use of qualified names.

The URI can be specified to provide a unique identifier for a Package. Within UML there is no predetermined usage for this, with the exception of profiles (see Using XMI to exchange Profiles in section 18.3.6). It may, for example, be used by model management facilities for model identification. The URI should hence be unique and unchanged once assigned. There is no requirement that the URI be dereferenceable (though this is of course permitted).

Notation

A package is shown as a large rectangle with a small rectangle (a “tab”) attached to the left side of the top of the large rectangle. The members of the package may be shown within the large rectangle. Members may also be shown by branching lines to member elements, drawn outside the package. A plus sign (+) within a circle is drawn at the end attached to the namespace (package).

- If the members of the package are not shown within the large rectangle, then the name of the package should be placed within the large rectangle.
- If the members of the package are shown within the large rectangle, then the name of the package should be placed within the tab.

The visibility of a package element may be indicated by preceding the name of the element by a visibility symbol (‘+’ for public and ‘-’ for private). Package elements with defined visibility may not have protected or package visibility.

The URI for a Package may be indicated with the text {uri=<uri>} following the package name.

Presentation Options

A tool may show visibility by a graphic marker, such as color or font. A tool may also show visibility by selectively displaying those elements that meet a given visibility level (e.g., only public elements). A diagram showing a package with contents must not necessarily show all its contents; it may show a subset of the contained elements according to some criterion.

Elements that become available for use in an importing package through a package import or an element import may have a distinct color or be dimmed to indicate that they cannot be modified.

Examples

There are three representations of the same package Types in Figure 11.27. The one on the left just shows the package without revealing any of its members. The middle one shows some of the members within the borders of the package, and the one to the right shows some of the members using the alternative membership notation.

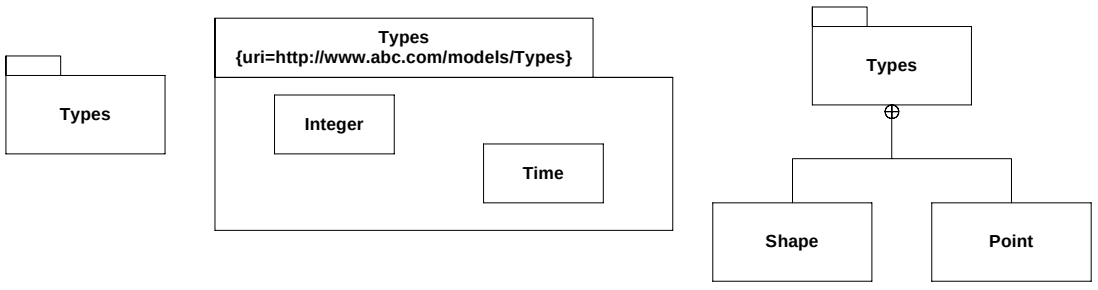


Figure 11.27 - Examples of a package with members

11.9.3 PackageMerge

A package merge defines how the contents of one package are extended by the contents of another package.

Generalizations

- “DirectedRelationship” on page 106

Description

A package merge is a directed relationship between two packages, that indicates that the contents of the two packages are to be combined. It is very similar to Generalization in the sense that the source element conceptually adds the characteristics of the target element to its own characteristics resulting in an element that combines the characteristics of both.

This mechanism should be used when elements defined in different packages have the same name and are intended to represent the same concept. Most often it is used to provide different definitions of a given concept for different purposes, starting from a common base definition. A given base concept is extended in increments, with each increment defined in a separate merged package. By selecting which increments to merge, it is possible to obtain a custom definition of a concept for a specific end. Package merge is particularly useful in meta-modeling and is extensively used in the definition of the UML metamodel.

Conceptually, a package merge can be viewed as an operation that takes the contents of two packages and produces a new package that combines the contents of the packages involved in the merge. In terms of model semantics, there is no difference between a model with explicit package merges, and a model in which all the merges have been performed.

Attributes

No additional attributes

Associations

- mergedPackage: Package [1]
  References the Package that is to be merged with the receiving package of the PackageMerge. Subsets *DirectedRelationship::target*.
- receivingPackage: Package [1]
  References the Package that is being extended with the contents of the merged package of the PackageMerge. Subsets *Element::owner* and *DirectedRelationship::source*.

Constraints

No additional constraints

Semantics

A package merge between two packages *implies* a set of transformations, whereby the contents of the package to be merged are combined with the contents of the receiving package. In cases in which certain elements in the two packages represent the same entity, their contents are (conceptually) merged into a single resulting element according to the formal rules of package merge specified below.

As with Generalization, a package merge between two packages in a model merely implies these transformations, but the results are not themselves included in the model. Nevertheless, the receiving package and its contents are deemed to represent the result of the merge, in the same way that a subclass of a class represents the aggregation of features of all of its superclasses (and not merely the increment added by the class). Thus, within a model, any reference to a model element contained in the receiving package implies a reference to the results of the merge rather than to the increment that is physically contained in that package. This is illustrated by the example in Figure 11.28 in which package P1 and package P2 both define different increments of the same class A (identified as P1::A and P2::A respectively). Package P2 merges the contents of package P1, which implies the merging of increment P1::A into increment P2::A. Package P3

imports the contents of P2 so that it can define a subclass of A called SubA. In this case, element A in package P3 (P3::A) represents the *result* of the merge of P1::A into P2::A and not just the increment P2::A. Note that, if another package were to *import* P1, then a reference to A in the importing package would represent the increment P1::A rather than the A resulting from merge.

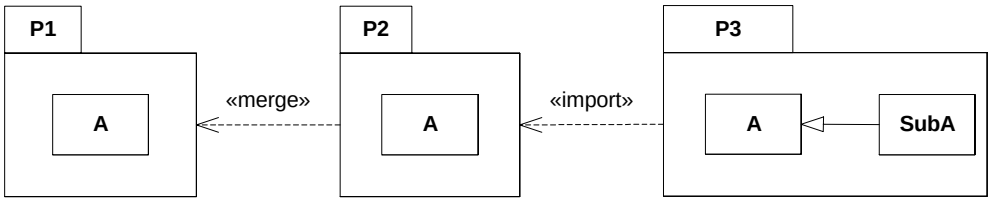


Figure 11.28 - Illustration of the meaning of package merge

To understand the rules of package merge, it is necessary to clearly distinguish between three distinct entities: the merged increment (e.g., P1::A in Figure 11.28), the receiving increment (e.g., P2::A), and the result of the merge transformations. The main difficulty comes from the fact that the receiving package and its contents represents both the operand and the results of the package merge, depending on the context in which they are considered. For example, in Figure 11.28 , with respect to the package merge operation, P2 represents the increment that is an operand for the merge. However, with respect to the import operation, P2 represents the result of the merge. This dual interpretation of the same model element can be confusing, so it is useful to introduce the following terminology that aids understanding:

- *merged package* - the first operand of the merge, that is, the package that is to be merged into the receiving package (this is the package that is the target of the merge arrow in the diagrams).
- *receiving package* - the second operand of the merge, that is, the package that, conceptually, contains the results of the merge (and which is the source of the merge arrow in the diagrams). However, this term is used to refer to the package and its contents *before* the merge transformations have been performed.
- *resulting package* - the package that, conceptually, contains the results of the merge. In the model, this is, of course, the same package as the receiving package, but this particular term is used to refer to the package and its contents *after* the merge has been performed.
- *merged element* - refers to a model element that exists in the merged package.
- *receiving element* - is a model element in the receiving package. If the element has a *matching* merged element, the two are combined to produce the resulting element (see below). This term is used to refer to the element *before* the merge has been performed (i.e., the increment itself rather than the result).
- *resulting element* - is a model element in the resulting package *after* the merge was performed. For receiving elements that have a matching merged element, this is the same element as the receiving element, but in the state *after* the merge was performed. For merged elements that have no matching receiving element, this is the merged element. For receiving elements that have no matching merged element, this is the same as the receiving element.
- *element type* - refers to the type of any kind of TypedElement, such as the type of a Parameter or StructuralFeature.
- *element metatype* - is the MOF type of a model element (e.g., Classifier, Association, Feature).

This terminology is based on a conceptual view of package merge that is represented by the schematic diagram in Figure 11.29 (NOTE: this is not a UML diagram). The owned elements of packages A and B are all incorporated into the namespace of package B. However, it is important to emphasize that this view is merely a convenience for describing the semantics of package merge and is not reflected in the repository model, that is, the *physical* model itself is not transformed in any way by the presence of package merges.

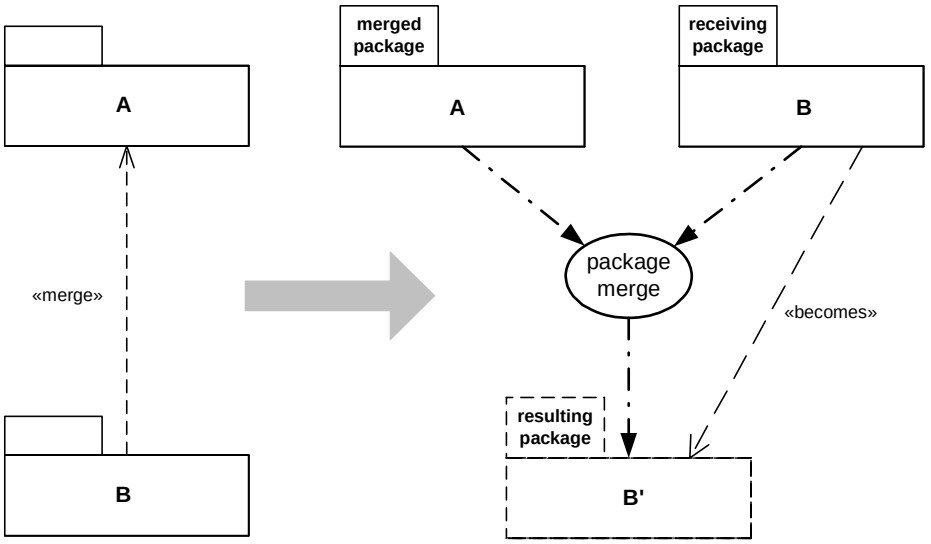


Figure 11.29 - Conceptual view of the package merge semantics

The semantics of package merge are defined by a set of constraints and transformations. The constraints specify the preconditions for a valid package merge, while the transformations describe its semantic effects (i.e., postconditions). If any constraints are violated, the package merge is ill-formed and the resulting model that contains it is invalid. Different metatypes have different semantics, but the general principle is always the same: a resulting element will not be any less capable than it was prior to the merge. This means, for instance, that the resulting navigability, multiplicity, visibility, etc. of a receiving model element will not be reduced as a result of a package merge. One of the key consequences of this is that model elements in the resulting package are compatible extensions of the corresponding elements in the (unmerged) receiving package *in the same namespace*. This capability is particularly useful in defining metamodel compliance levels such that each successive level is compatible with the previous level, including their corresponding XMI representations.

In this specification, explicit merge transformations are only defined for certain general metatypes found mostly in metamodels (Packages, Classes, Associations, Properties, etc.), since the semantics of merging other kinds of metatypes (e.g., state machines, interactions) are complex and domain specific. Elements of all other kinds of metatypes are transformed according to the default rule: they are simply deep copied into the resulting package. (This rule can be superseded for specific metatypes through profiles or other kinds of language extensions.)

General package merge rules

A merged element and a receiving element *match* if they satisfy the matching rules for their metatype.

CONSTRAINTS:

1. There can be no cycles in the «merge» dependency graph.

2. A package cannot merge a package in which it is contained.
3. A package cannot merge a package that it contains.
4. A merged element whose metatype is not a kind of Package, Class, DataType, Property, Association, Operation, Constraint, Enumeration, or EnumerationLiteral, cannot have a receiving element with the same name and metatype unless that receiving element is an exact copy of the merged element (i.e., they are the same).
5. A package merge is valid if and only if all the constraints required to perform the merge are satisfied.
6. Matching typed elements (e.g., Properties, Parameters) must have conforming types. For types that are classes or data types, a conforming type is either the same type or a common supertype. For all other cases, conformance means that the types must be the same.
7. A receiving element cannot have explicit references to any merged element.
8. Any redefinitions associated with matching redefinable elements must not be conflicting.

TRANSFORMATIONS:

1. (*The default rule*) Merged or receiving elements for which there is no matching element are deep copied into the resulting package.
2. The result of merging two elements with matching names and metatypes that are exact copies of each other is the receiving element.
3. Matching elements are combined according to the transformation rules specific to their metatype and the results included in the resulting package.
4. All type references to typed elements that end up in the resulting package are transformed into references to the corresponding resulting typed elements (i.e., not to their respective increments).
5. For all matching elements: if both matching elements have private visibility, the resulting element will have private visibility; otherwise, the resulting element will have public visibility.
6. For all matching classifier elements: if both matching elements are abstract, the resulting element is abstract; otherwise, the resulting element is non-abstract.
7. For all matching classifier elements: if both matching elements are final specializations, the resulting element is a final specialization; otherwise, the resulting element is a non-final specialization.
8. For all matching elements: if both matching elements are not derived, the resulting element is also not derived; otherwise, the resulting element is derived.
9. For all matching multiplicity elements: the lower bound of the resulting multiplicity is the lesser of the lower bounds of the multiplicities of the matching elements.
10. For all matching multiplicity elements: the upper bound of the resulting multiplicity is the greater of the upper bounds of the multiplicities of the matching elements.
11. Any stereotypes applied to a model element in either a merged or receiving element are also applied to the corresponding resulting element.
12. For matching redefinable elements: different redefinitions of matching redefinable elements are combined conjunctively.

13. For matching redefinable elements: if both matching elements have isLeaf=true, the resulting element also has isLeaf=true; otherwise, the resulting element has isLeaf=false.

Package rules

Elements that are a kind of Package match by name and metatype (e.g., profiles match with profiles and regular packages with regular packages).

CONSTRAINTS:

1. All classifiers in the merged package must have a non-empty qualified name and be distinguishable in the merged package.
2. All classifiers in the receiving package must have a non-empty qualified name and be distinguishable in the receiving package.

TRANSFORMATIONS:

1. A nested package from the merged package is transformed into a nested package with the same name in the resulting package, unless the receiving package already contains a matching nested package. In the latter case, the merged nested package is recursively merged with the matching receiving nested package.
2. An element import owned by the receiving package is transformed into a corresponding element import in the resulting package. Imported elements are not merged (unless there is also a package merge to the package owning the imported element or its alias).

Class and DataType rules

Elements that are kinds of Class or DataType match by name and metatype.

TRANSFORMATIONS:

1. All properties from the merged classifier are merged with the receiving classifier to produce the resulting classifier according to the property transformation rules specified below.
2. Nested classifiers are merged recursively according to the same rules.

Property rules

Elements that are kinds of Property match by name and metatype.

CONSTRAINTS:

1. The static (or non-static) characteristic of matching properties must be the same.
2. The uniqueness characteristic of matching properties must be the same.
3. Any constraints associated with matching properties must not be conflicting.

TRANSFORMATIONS:

1. For merged properties that do not have a matching receiving property, the resulting property is a newly created property in the resulting classifier that is the same as the merged property.

2. For merged properties that have a matching receiving property, the resulting property is a property with the same name and characteristics except where these characteristics are different. Where these characteristics are different, the resulting property characteristics are determined by application of the appropriate transformation rules.
3. For matching properties: if both properties are designated read-only, the resulting property is also designated read-only; otherwise, the resulting property is designated as not read-only.
4. For matching properties: if both properties are unordered, then the resulting property is also unordered; otherwise, the resulting property is ordered.
5. For matching properties: if neither property is designated as a subset of some derived union, then the resulting property will not be designated as a subset; otherwise, the resulting property will be designated as a subset of that derived union.
6. For matching properties: different constraints of matching properties are combined conjunctively.
7. For matching properties: if either the merged and/or receiving elements are non-unique, the resulting element is non-unique; otherwise, the resulting element is designated as unique.
8. The resulting property type is transformed to refer to the corresponding type in the resulting package.

Association rules

Elements that are a kind of Association match by name and metatype.

CONSTRAINTS:

1. These rules only apply to binary associations. (The default rule is used for merging n-ary associations.)
2. The receiving association end must be a composite if the matching merged association end is a composite.
3. The receiving association end must be owned by the association if the matching merged association end is owned by the association

TRANSFORMATIONS:

1. A merge of matching associations is accomplished by merging the Association classifiers (using the merge rules for classifiers) and merging their corresponding owned end properties according to the rules for properties and association ends.
2. For matching association ends: if neither association end is navigable, then the resulting association end is also not navigable. In all other cases, the resulting association end is navigable.

Operation rules

Elements that are a kind of Operation match by name, parameter order, and parameter types, not including any return type.

CONSTRAINTS:

1. Operation parameters and types must conform to the same rules for type and multiplicity as were defined for properties.
2. The receiving operation must be a query if the matching merged operation is a query.

TRANSFORMATIONS:

1. For merged operations that do not have a matching receiving operation, the resulting operation is an operation with the same name and signature in the resulting classifier.
2. For merged operations that have a matching receiving operation, the resulting operation is the outcome of a merge of the matching merged and receiving operations, with parameter transformations performed according to the property transformations defined above.

Enumeration rules

Elements that are a kind of EnumerationLiteral match by owning enumeration and literal name.

CONSTRAINTS:

1. Matching enumeration literals must be in the same order.

TRANSFORMATIONS:

1. Non-matching enumeration literals from the merged enumeration are concatenated to the receiving enumeration.

Constraint Rules

CONSTRAINTS:

1. Constraints must be mutually non-contradictory.

TRANSFORMATIONS:

1. The constraints of the merged model elements are conjunctively added to the constraints of the matching receiving model elements.

Notation

A PackageMerge is shown using a dashed line with an open arrowhead pointing from the receiving package (the source) to the merged package (the target). In addition, the keyword «merge» is shown near the dashed line.

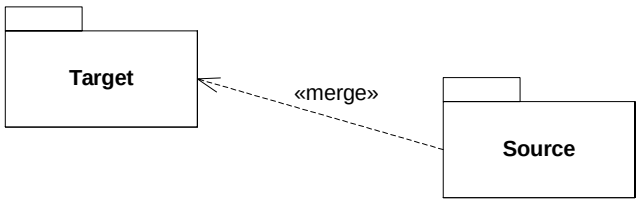


Figure 11.30 - Notation for package merge

Examples

In Figure 11.31, packages P and Q are being merged by package R, while package S merges only package Q.

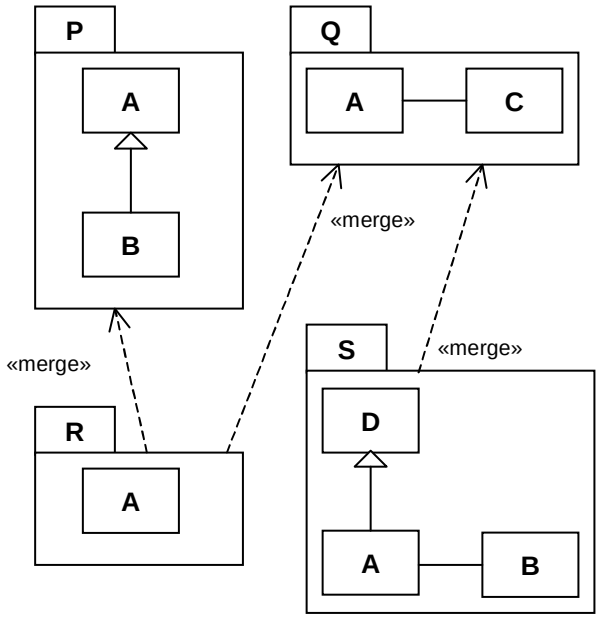


Figure 11.31 - Simple example of package merges

The transformed packages R and S are shown in Figure 11.32. The expressions in square brackets indicate which individual increments were merged to produce the final result, with the "@" character denoting the merge operator (note that these expressions are not part of the standard notation, but are included here for explanatory purposes).

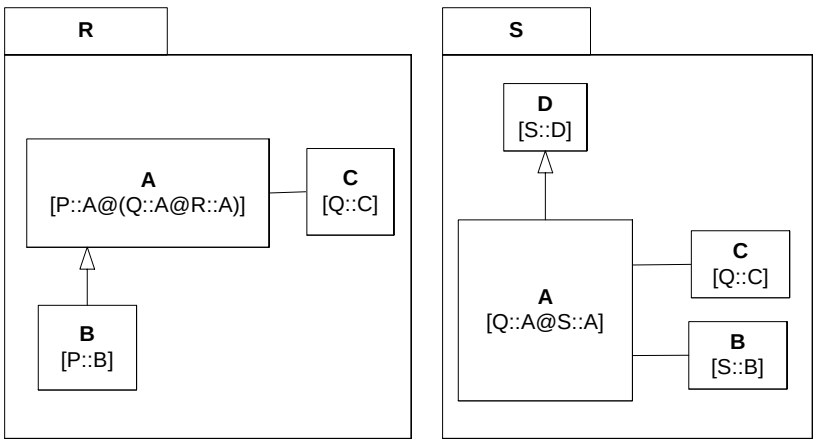


Figure 11.32 - Simple example of transformed packages following the merges in Figure 11.31

In Figure 11.33, additional package merges are introduced by having package T, which is empty prior to execution of the merge operation, merge packages R and S defined previously.

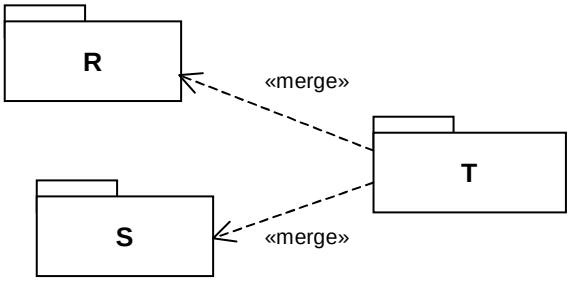


Figure 11.33 - Introducing additional package merges

In Figure 11.34, the transformed version of package T is depicted. In this package, the partial definitions of A, B, C, and D have all been brought together. Note that the types of the ends of the associations that were originally in the packages Q and S have all been updated to refer to the appropriate elements in package T.

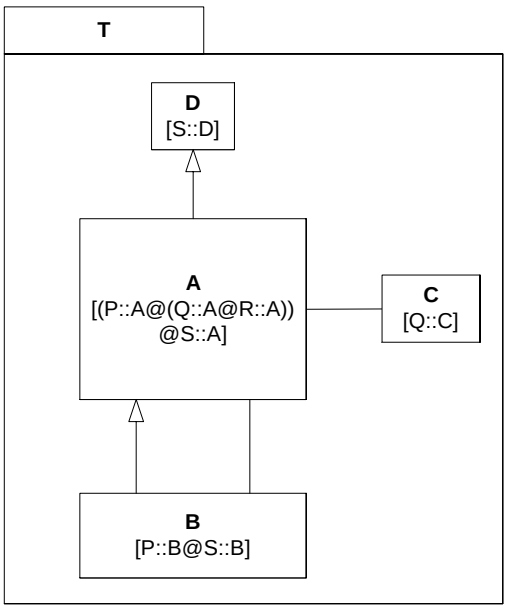


Figure 11.34 - The result of the additional package merges in Figure 11.33

12 Core::Profiles

The Profiles package contains mechanisms that allow metaclasses from existing metamodels to be extended to adapt them for different purposes. This includes the ability to tailor the UML metamodel for different platforms (such as J2EE or .NET) or domains (such as real-time or business process modeling). The profiles mechanism is consistent with the OMG Meta Object Facility (MOF).

Positioning profiles versus metamodels, MOF and UML

The infrastructure specification is reused at several meta-levels in various OMG specifications that deal with modeling. For example, MOF uses it to provide the ability to model metamodels, whereas the UML superstructure uses it to model the UML model. This clause deals with use cases comparable to the MOF at the meta-meta-level, which is one level higher than the UML metamodel specification. In order to allow this, the reference metamodel must be defined as an instance of UML that corresponds to its definition using MOF. Thus when defining a UML profile, the profile?s stereotypes are defined to extend the UML classes in the normative version of the UML metamodel merged at the highest level of compliance, L3, defined in UML whose xmi serialization is listed in Annex C. This approach allows defining a UML profile for arbitrary subsets of the UML at lower levels of compliance, which can be further restricted using constraints defined in the profile.

Profiles History and design requirements

The Profile mechanism has been specifically defined for providing a lightweight extension mechanism to the UML standard. In UML 1.1, stereotypes and tagged values were used as string-based extensions that could be attached to UML model elements in a flexible way. In subsequent revisions of UML, the notion of a Profile was defined in order to provide more structure and precision to the definition of Stereotypes and Tagged values. The UML 2 infrastructure and superstructure specifications have carried this further, by defining it as a specific meta-modeling technique.

The following requirements have driven the definition of profile semantics from inception:

1. A profile must provide mechanisms for specializing a reference metamodel (such as a set of UML packages) in such a way that the specialized semantics do not contradict the semantics of the reference metamodel. That is, profile constraints may typically define well-formedness rules that are more constraining (but consistent with) those specified by the reference metamodel.

2. It must be possible to interchange profiles between tools, together with models to which they have been applied, by using the UML XMI interchange mechanisms. A profile must therefore be defined as an interchangeable UML model. In addition to exchanging profiles together with models between tools, profile application should also be definable “by reference” (e.g., “import by name”); that is, a profile does not need to be interchanged if it is already present in the importing tool.

3. A profile must be able to reference domain-specific UML libraries where certain model elements are pre-defined.

4. It must be possible to specify which profiles are being applied to a given Package (or any specializations of that concept). This is particularly useful during model interchange so that an importing environment can interpret a model correctly.

5. It should be possible to define a UML extension that combines profiles and model libraries (including template libraries) into a single logical unit. However, within such a unit, for definitional clarity and for ease of interchange (e.g., ‘reference by name’), it should still be possible to keep the libraries and the profiles distinct from each other.

6. A profile should be able to specialize the semantics of standard UML metamodel elements. For example, in a model with the profile “Java model,” generalization of classes should be able to be restricted to single inheritance without having to explicitly assign a stereotype «Java class» to each and every class instance.

7. A notational convention for graphical stereotype definitions as part of a profile should be provided.

8. In order to satisfy requirement [1] above, UML Profiles should form a metamodel extension mechanism that imposes certain restrictions on how the UML metamodel can be modified. The reference metamodel is considered as a “read only” model, that is extended without changes by profiles. It is therefore forbidden to insert new metaclasses in the UML metaclass hierarchy (i.e., new super-classes for standard UML metaclasses) or to modify the standard UML metaclass definitions (e.g., by adding meta-associations). Such restrictions do not apply in a MOF context where in principle any metamodel can be reworked in any direction.

9. The vast majority of UML case tools should be able to implement Profiles. The design of UML profiles should therefore not constrain these tools to have an internal implementation based on a meta-metamodel/metamodel architecture.

10. Profiles can be dynamically applied to or retracted from a model. It is possible on an existing model to apply new profiles, or to change the set of applied profiles.

11. Profiles can be dynamically combined. Frequently, several profiles will be applied at the same time on the same model. This profile combination may not be foreseen at profile definition time.

12. Models can be exchanged regardless of the profiles known by the destination target. The destination of the exchange of a model extended by a profile may not know the profile, and is not required to interpret a specific profile description. The destination environment interprets extensions only if it possesses the required profiles.

Extensibility

The profiles mechanism is not a first-class extension mechanism (i.e., it does not allow for modifying existing metamodels). Rather, the intention of profiles is to give a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method. Each such adaptation is grouped in a profile. It is not possible to take away any of the constraints that apply to a metamodel such as UML using a profile, but it is possible to add new constraints that are specific to the profile. The only other restrictions are those inherent in the profiles mechanism; there is nothing else that is intended to limit the way in which a metamodel is customized.

First-class extensibility is handled through MOF, where there are no restrictions on what you are allowed to do with a metamodel: you can add and remove metaclasses and relationships as you find necessary. Of course, it is then possible to impose methodology restrictions that you are not allowed to modify existing metamodels, but only extend them. In this case, the mechanisms for first-class extensibility and profiles start coalescing.

There are several reasons why you may want to customize a metamodel:

- Give a terminology that is adapted to a particular platform or domain (such as capturing EJB terminology like home interfaces, enterprise java beans, and archives).
- Give a syntax for constructs that do not have a notation (such as in the case of actions).
- Give a different notation for already existing symbols (such as being able to use a picture of a computer instead of the ordinary node symbol to represent a computer in a network).
- Add semantics that is left unspecified in the metamodel (such as how to deal with priority when receiving signals in a statemachine).

- Add semantics that does not exist in the metamodel (such as defining a timer, clock, or continuous time).
- Add constraints that restrict the way you may use the metamodel and its constructs (such as disallowing actions from being able to execute in parallel within a single transition).
- Add information that can be used when transforming a model to another model or code (such as defining mapping rules between a model and Java code).

Profiles and Metamodels

There is no simple answer for when you should create a new metamodel and when you instead should create a new profile.

12.1 Profiles package

The Profiles package is dependent on the *Constructs* package from *Core*, as is depicted in Figure 12.1.

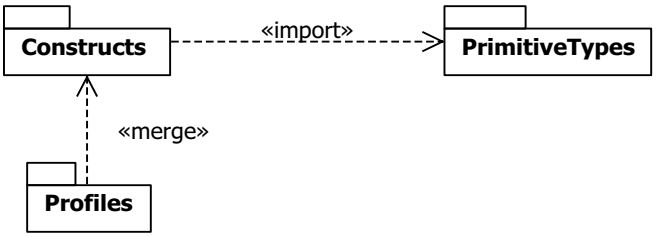


Figure 12.1 - Dependencies between packages described in this clause

The classes of the Profiles package are depicted in Figure 12.2, and subsequently specified textually.

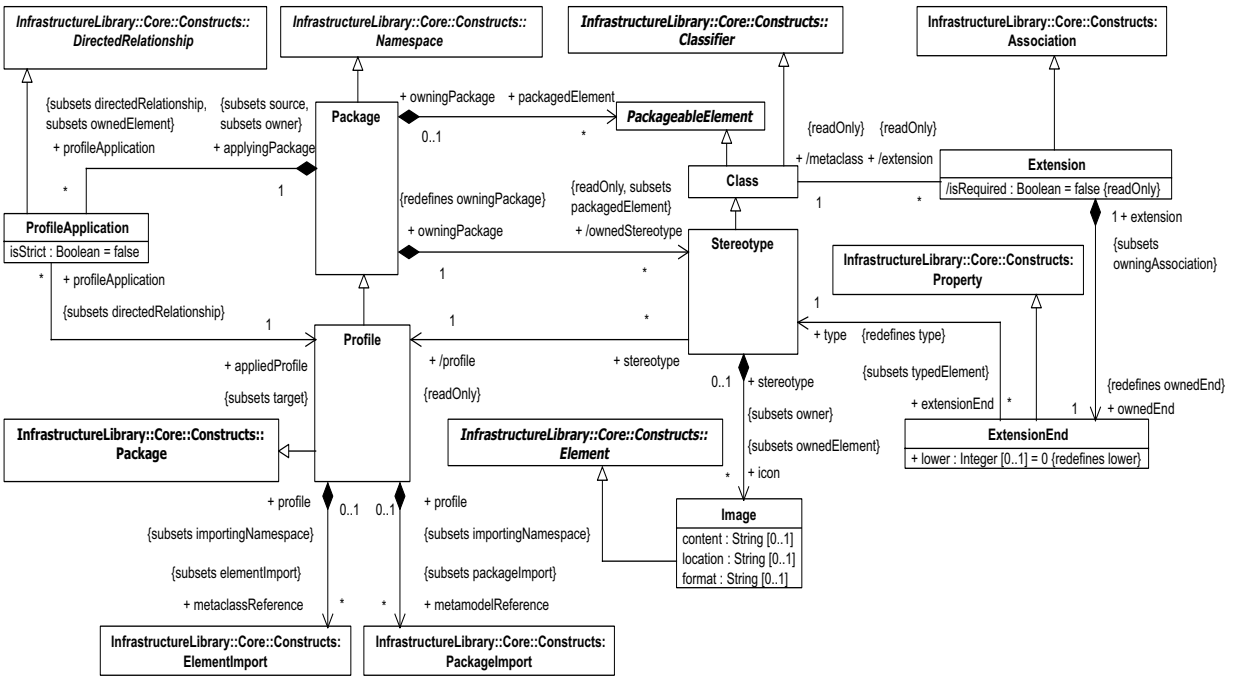


Figure 12.2 - The classes defined in the Profiles package

12.1.1 Class (from Profiles)

Generalizations

- “Classifier” on page 121
- “PackageableElement (from Profiles)” on page 184

Description

Class has derived association that indicates how it may be extended through one or more stereotypes.

Stereotype is the only kind of metaclass that cannot be extended by stereotypes.

Attributes

No additional attributes

Associations

- / extension: Extension [*]

  References the Extensions that specify additional properties of the metaclass. The property is derived from the extensions whose memberEnds are typed by the Class.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

Presentation Options

A Class that is extended by a Stereotype may be extended by the optional keyword «Metaclass» shown above or before its name.

Examples

In Figure 12.3, an example is given where it is made explicit that the extended class *Interface* is in fact a metaclass (from a reference metamodel).

Figure 12.3 - Showing that the extended class is a metaclass

Changes from previous UML

A link typed by UML 1.4 ModelElement::stereotype is mapped to a link that is typed by Class::extension.

12.1.2 Extension (from Profiles)

An extension is used to indicate that the properties of a metaclass are extended through a stereotype, and gives the ability to flexibly add (and later remove) stereotypes to classes.

Generalizations

- “Association” on page 111

Description

Extension is a kind of Association. One end of the Extension is an ordinary Property and the other end is an ExtensionEnd. The former ties the Extension to a Class, while the latter ties the Extension to a Stereotype that extends the Class.

Attributes

- / isRequired: Boolean
  Indicates whether an instance of the extending stereotype must be created when an instance of the extended class is created. The attribute value is derived from the value of the lower property of the ExtensionEnd referenced by *Extension::ownedEnd*; a lower value of 1 means that isRequired is *true*, but otherwise it is *false*. Since the default value of ExtensionEnd::lower is 0, the default value of isRequired is *false*.

Associations

- ownedEnd: ExtensionEnd [1]
  References the end of the extension that is typed by a Stereotype. {Redefines *Association::ownedEnd*}
- / metaclass: Class [1]
  References the Class that is extended through an Extension. The property is derived from the type of the memberEnd that is not the ownedEnd.

Constraints

[1] The non-owned end of an Extension is typed by a Class.

```
metaclassEnd()->notEmpty() and metaclass()->oclIsKindOf(Class)
```

[2] An Extension is binary (i.e., it has only two memberEnds).

```
memberEnd->size() = 2
```

Additional Operations

[1] The query metaclassEnd() returns the Property that is typed by a metaclass (as opposed to a stereotype).

```
Extension::metaclassEnd(): Property;
metaclassEnd = memberEnd->reject(ownedEnd)
```

[2] The query metaclass() returns the metaclass that is being extended (as opposed to the extending stereotype).

```
Extension::metaclass(): Class;
metaclass = metaclassEnd().type
```

[3] The query isRequired() is true if the owned end has a multiplicity with the lower bound of 1.

```
Extension::isRequired(): Boolean;
isRequired = (ownedEnd->lowerBound() = 1)
```

Semantics

A required extension means that an instance of a stereotype must always be linked to an instance of the extended metaclass. The instance of the stereotype is typically deleted only when either the instance of the extended metaclass is deleted, or when the profile defining the stereotype is removed from the applied profiles of the package. The model is not well-formed if an instance of the stereotype is not present when isRequired is true. If the extending stereotype has subclasses, then at most one instance of the stereotype or one of its subclasses is required.

A non-required extension means that an instance of a stereotype can be linked to an instance of an extended metaclass at will, and also later deleted at will; however, there is no requirement that each instance of a metaclass be extended. An instance of a stereotype is further deleted when either the instance of the extended metaclass is deleted, or when the profile defining the stereotype is removed from the applied profiles of the package.

The equivalence to a MOF construction is shown in Figure 12.4. This figure illustrates the case shown in Figure 12.6, where the “Home” stereotype extends the “Interface” metaclass. In this figure, Interface is an instance of a CMOF::Class and Home is an instance of a CMOF::Stereotype. The MOF construct equivalent to an extension is an aggregation from

the extended metaclass to the extension stereotype, navigable from the extension stereotype to the extended metaclass. When the extension is required, then the cardinality on the extension stereotype is "1." The role names are provided using the following rule: The name of the role of the extended metaclass is:

'base_' extendedMetaclassName

The name of the role of the extension stereotype is:

'extension$_' stereotypeName

Constraints are frequently added to stereotypes. The role names will be used for expressing OCL navigations. For example, the following OCL expression states that a Home interface shall not have attributes:

```
self.base_Interface.ownedAttributes->size() = 0
```

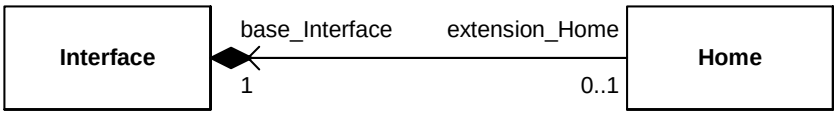


Figure 12.4 - MOF model equivalent to extending "interface" by the "Home" stereotype

Notation

The notation for an Extension is an arrow pointing from a Stereotype to the extended Class, where the arrowhead is shown as a filled triangle. An Extension may have the same adornments as an ordinary association, but navigability arrows are never shown. If isRequired is true, the property {required} is shown near the ExtensionEnd.

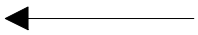


Figure 12.5 - The notation for an Extension

Presentation Options

It is possible to use the multiplicities 0..1 or 1 on the ExtensionEnd as an alternative to the property {required}. Due to how isRequired is derived, the multiplicity 0..1 corresponds to isRequired being *false*.

Style Guidelines

Adornments of an Extension are typically elided.

Examples

In Figure 12.6, a simple example of using an extension is shown, where the stereotype *Home* extends the metaclass *Interface*.

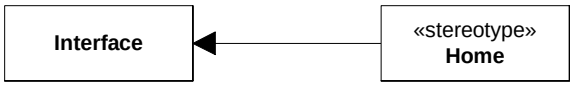


Figure 12.6 - An example of using an Extension

An instance of the stereotype *Home* can be added to and deleted from an instance of the class *Interface* at will, which provides for a flexible approach of dynamically adding (and removing) information specific to a profile to a model.

In Figure 12.7, an instance of the stereotype *Bean* always needs to be linked to an instance of class *Component* since the Extension is defined to be required. (Since the stereotype *Bean* is abstract, this means that an instance of one of its concrete subclasses always has to be linked to an instance of class *Component*.) The model is not well-formed unless such a stereotype is applied. This provides a way to express extensions that should always be present for all instances of the base metaclass depending on which profiles are applied.

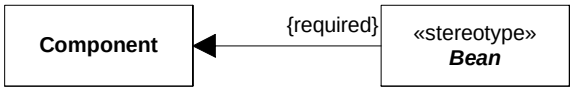


Figure 12.7 - An example of a required Extension

Changes from previous UML

Extension did not exist as a metaclass in UML 1.x.

Occurrences of Stereotype::baseClass of UML 1.4 is mapped to an instance of Extension, where the ownedEnd is typed by Stereotype and the other end is typed by the metaclass that is indicated by the baseClass.

12.1.3 ExtensionEnd (from Profiles)

An extension end is used to tie an extension to a stereotype when extending a metaclass.

Generalizations

- “Property” on page 124

Description

ExtensionEnd is a kind of Property that is always typed by a Stereotype.

An ExtensionEnd is a navigable ownedEnd, owned by an Extension. This allows a stereotype instance to be attached to an instance of the extended classifier without adding a property to the classifier.

The aggregation of an ExtensionEnd is always composite.

The default multiplicity of an ExtensionEnd is 0..1.

Attributes

- /lower : integer [0..1] = 0
  This redefinition changes the default multiplicity of association ends, since model elements are usually extended by 0 or 1 instance of the extension stereotype. The value of lower is derived from the redefined lowerBound() operation, via a constraint inherited from MultiplicityElement. {*Redefines MultiplicityElement::lower*}

Associations

- type: Stereotype [1]
  References the type of the ExtensionEnd. Note that this association restricts the possible types of an ExtensionEnd to only be Stereotypes. {Redefines *TypedElement::type*}.

Constraints

[1] The multiplicity of ExtensionEnd is 0..1 or 1.

```
(self->lowerBound() = 0 or self->lowerBound() = 1) and self->upperBound() = 1
```

[2] The aggregation of an ExtensionEnd is composite.

```
self.aggregation = #composite
```

Additional Operations

[1] The query lowerBound() returns the lower bound of the multiplicity as an Integer. This is a redefinition of the default lower bound, which, if empty, normally evaluates to 1 for MulticplicityElements.

```
ExtensionEnd::lowerBound() : Set(Integer);
lowerBound = if lowerValue->isEmpty() then 0 else lowerValue->IntegerValue() endif
```

Semantics

No additional semantics

Notation

No additional notation

Examples

See “Extension (from Profiles)” on page 177.

Changes from previous UML

ExtensionEnd did not exist as a metaclass in UML 1.4. See “Extension (from Profiles)” on page 177 for further details.

12.1.4 Image (from Profiles)

Physical definition of a graphical image.

Generalizations

- “Element” on page 106

Description

The Image class provides the necessary information to display an Image in a diagram. Icons are typically handled through the Image class.

Attributes

- content : String [0..1]
  This contains the serialization of the image according to the imageFormat. The value could represent a bitmap, image such as a GIF file, or drawing ‘instructions’ using a standard such as Scalable Vector Graphics (SVG) (which is XML based).
- format : String [0..1]
  This indicates the format of the imageContent - which is how the string imageContent should be interpreted. The following values are reserved:

  SVG, GIF, PNG, JPG, WMF, EMF, BMP.

  In addition the prefix ‘MIME:’ is also reserved: this must be followed by a valid MIME type as defined by RFC3023. This option can be used as an alternative to express the reserved values above, for example “SVG” could instead be expressed “MIME: image/svg+xml.”
- location : String [0..1]
  This contains a location that can be used by a tool to locate the image as an alternative to embedding it in the stereotype.

Associations

No additional associations

Constraints

No additional constraints

Semantics

Information such as physical localization or format is provided by the Image class. The Image class provides a generic way of representing images in different formats. Although some predefined values are specified for imageFormat for convenience and interoperability, the set of possible formats is open-ended. However there is no requirement for an implementation to be able to interpret and display any specific format, including those predefined values.

12.1.5 Package (from Profiles)

Generalizations

- “Namespace” on page 137

Description

A package can have one or more ProfileApplications to indicate which profiles have been applied.

Because a profile is a package, it is possible to apply a profile not only to packages, but also to profiles.

Attributes

No additional attributes

Associations

- profileApplication : ProfileApplication [*]
  References the ProfileApplications that indicate which profiles have been applied to the Package. Subsets *Element::ownedElement.*
- /ownedStereotype: Stereotype [*]
  References the Stereotypes that are owned by the Package. Subsets *Package::packagedElement*
- packagedElement: PackageableElement [*]
  Specifies the PackageableElements that are owned by this Package.

Constraints

No additional constraints

Additional Operations

[1] The query allApplicableStereotypes() returns all the directly or indirectly owned stereotypes, including stereotypes contained in sub-profiles.

```
Package::allApplicableStereotypes(): Set(Stereotype)
result = self.ownedStereotype->union(self.ownedMember->
    select(oclIsKindOf(Package)).oclAsType(Package).allApplicableStereotypes()->flatten())->asSet()
```

[2] The query containingProfile() returns the closest profile directly or indirectly containing this package (or this package itself, if it is a profile).

```
Package::containingProfile(): Profile
result =
    if self.oclIsKindOf(Profile) then
        self.oclAsType(Profile)
    else
        self.namespace.oclAsType(Package).containingProfile()
    endif
```

Semantics

The association “appliedProfile” between a package and a profile conceptually crosses metalevels: It links one element from a model (a kind of package) to an element of its metamodel and represents the set of profiles that define the extensions applicable to the package. In order to represent this conceptual crossing of metalevels, the UML metamodel is available as an instance of UML.

Notation

No additional notation

Changes from previous UML

In UML 1.4, it was not possible to indicate which profiles were applied to a package.

12.1.6 PackageableElement (from Profiles)

Generalizations

- “PackageableElement” on page 149 (merge increment)

Description

See InfrastructureLibrary::Core::Constructs::PackageableElement.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

12.1.7 Profile (from Profiles)

A profile defines limited extensions to a reference metamodel with the purpose of adapting the metamodel to a specific platform or domain.

Generalizations

- “Package (from Profiles)” on page 182

Description

A Profile is a kind of Package that extends a reference metamodel. The primary extension construct is the Stereotype, which is defined as part of Profiles.

A profile introduces several constraints, or restrictions, on ordinary metamodeling through the use of the metaclasses defined in this package.

A profile is a restricted form of a metamodel that must always be related to a *reference metamodel*, such as UML, as described below. A profile cannot be used without its reference metamodel, and defines a limited capability to extend metaclasses of the reference metamodel. The extensions are defined as stereotypes that apply to existing metaclasses.

Attributes

No additional attributes

Associations

- metaclassReference: ElementImport [*]
  References a metaclass that may be extended. Subsets *Package::elementImport*
- metamodelReference: PackageImport [*]
  References a package containing (directly or indirectly) metaclasses that may be extended. Subsets *Package::packageImport*.

Constraints

[1] An element imported as a metaclassReference is not specialized or generalized in a Profile.

```
self.metaclassReference.importedElement->
    select(c | c.oclIsKindOf(Classifier) and
    (c.generalization.namespace = self or
    c.specialization.namespace = self))->isEmpty()
```

[2] All elements imported either as metaclassReferences or through metamodelReferences are members of the same base reference metamodel.

```
self.metamodelReference.importedPackage.elementImport.importedElement.allOwningPackages()->
    union(self.metaclassReference.importedElement.allOwningPackages())->notEmpty()
```

Additional Operations

[1] The query allOwningPackages() returns all the directly or indirectly owning packages.

```
NamedElement::allOwningPackages(): Set(Package)
allOwningPackages = self.namespace->select(p | p.oclIsKindOf(Package))->
    union(p.allOwningPackages())
```

Semantics

A profile by definition extends a reference metamodel. It is not possible to define a standalone profile that does not directly or indirectly extend an existing metamodel. The profile mechanism may be used with any metamodel that is created from MOF, including UML and CWM.

A reference metamodel typically consists of metaclasses that are either imported or locally owned. All metaclasses that are extended by a profile have to be members of the same reference metamodel. The “metaclassReference” element imports and “metamodelReference” package imports serve two purposes: (1) they identify the reference metamodel elements that are imported by the profile and (2) they specify the profile’s filtering rules. The filtering rules determine which elements of the metamodel are *visible* when the profile is applied and which ones are *hidden*. Note that applying a profile does not change the underlying model in any way; it merely defines a view of the underlying model.

In general, only model elements that are instances of imported reference metaclasses will be visible when the profile is applied. All other metaclasses will be hidden. By default, model elements whose metaclasses are public and owned by the reference metamodel are visible. This applies transitively to any subpackages of the reference metamodel according to the default rules of package import. If any metaclasses is imported using a *metaclass reference* element import, then model elements whose metaclasses are the same as that metaclass are visible. Note, however, that a metaclass reference overrides a *metamodel reference* whenever an element or package of the referenced metamodel is also referenced by a metaclass reference. In such cases, only the elements that are explicitly referenced by the metaclass reference will be visible, while all other elements of the metamodel package will be hidden.

The following rules are used to determine whether a model element is visible or hidden when a profile has been applied. Model elements are *visible* if they are instances of metaclasses that are:

1. referenced by an explicit MetaclassReference, or
2. contained (directly or transitively) in a package that is referenced by an explicit MetamodelReference; unless there are other elements of subpackages of that package that are explicitly referenced by a MetaclassReference, or
3. extended by a stereotype owned by the applied profile (even if the extended metaclass itself is not visible).

All other model elements are *hidden* (not visible) when the profile is applied.

The most common case is when a profile just imports an entire metamodel using a MetamodelReference. In that case, every element of the metamodel is visible.

In the example in Figure 12.8, MyMetamodel is a metamodel containing two metaclasses: Metaclass1 and Metaclass2. MyProfile is a profile that references MyMetamodel and Metaclass2. However, there is also an explicit metaclass reference to Metaclass2, which overrides the metamodel reference. An application of MyProfile to some model based on MyMetamodel will show instances of Metaclass2 (because it is referenced by an explicit metaclass reference). Also, those instances of Metaclass1 that are extended by an instance of MyStereotype will be visible. However, instances of Metaclass1 that are not extended by MyStereotype remain hidden.

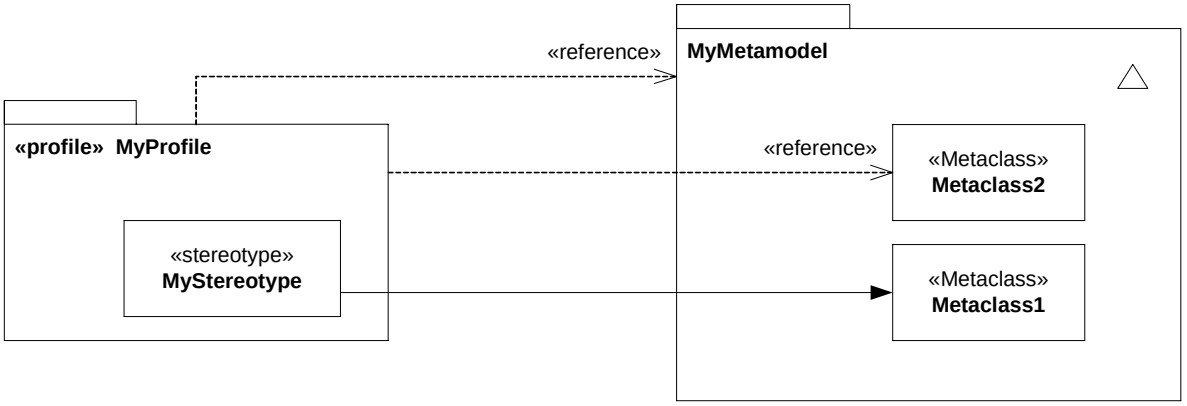


Figure 12.8 Specification of an accessible metaclass

If a profile P1 imports another profile P2, then all metaclassReference and metamodelReference associations will be combined at the P2 level, and the filtering rules apply to this union.

The filtering rules defined at the profile level are, in essence, merely a suggestion to modeling tools on what to do when a profile is applied to a model.

The “isStrict” attribute on a profileApplication specifies that the filtering rules have to be applied strictly. If isStrict is true on a ProfileApplication, then no other metaclasses than the accessible one defined by the profile shall be accessible when the profile is applied on a model. This prohibits the combination of applied profiles that specify different accessible metaclasses.

In addition to these import and filtering mechanisms, profile designers can select the appropriate metamodel by selecting the appropriate subpackages, and using the package merge mechanism. For example, they can build a specific reference metamodel by merging UML2 superstructure packages and classes, and or import packages from one of the UML2 compliance packages (L0-L4).

A Profile can define Classes, DataTypes, PrimitiveTypes, and Enumerations as well as Stereotypes since Profiles imports Constructs. However, these types can only be used as the type of properties in the profile, they cannot be used as types in models the profile is applied to since they apply at the meta-model level, not the model level. It is however possible to define these types in separate packages and import them as needed in both profiles and models in order to use them for both purposes.

Stereotypes can participate in binary associations. The opposite class can be another stereotype, a non-stereotype class that is owned by a profile, or a metaclass of the reference metamodel. For these associations there must be a property owned by the Stereotype to navigate to the opposite class. Where the opposite class is not a stereotype, the opposite property must be owned by the Association itself rather than the other class/metaclass.

The most direct implementation of the Profile mechanism that a tool can provide is by having a metamodel based implementation, similar to the Profile metamodel. However, this is not a requirement of the current standard, which requires only the support of the specified notions, and the standard XMI based interchange capacities. The profile mechanism has been designed to be implementable by tools that do not have a metamodel-based implementation. Practically any mechanism used to attach new values to model elements can serve as a valid profile implementation. As an example, the UML1.4 profile metamodel could be the basis for implementing a UML 2-compliant profile tool.

Integrating and extending Profiles

Integrating and extending existing profiles can result in improved reuse, better integration between OMG specifications, and less gaps and overlaps between metamodel extensions. There are a number of ways to create, extend and integrate profiles. These are described briefly in this section in order to foster better profile integration and reuse.

The simplest form of profile integration is to simply apply multiple profiles to the same model. This requires no integration between the profiles at all. Such profiles might be designed to complement each other, addressing different concerns.

It is also possible to one profile to reuse all of or parts of another, and to extend existing profiles. Like any other class, Stereotypes can be defined in packages or profiles that can be factored for reuse. These stereotypes can be directly reused through PackageImport or ElementImport in other profiles. Importing profiles can also use Generalizations to extend referenced and reused stereotypes for their unique purposes.

However, referencing stereotypes in other packages and profiles can sometimes result in undue coupling. This is because the referenced stereotypes may be associated with other stereotypes resulting in the need for the referencing profile to include large, perhaps unpredictable parts of the referenced profile in order to maintain semantic consistency. This can lead to unexpected burden on profile implementers and users because more is reused than intended resulting in coupling complex profiles together so they cannot be easily reused separately.

PackageMerge can be used to address this problem. A profile can define stereotypes that intentionally overlap with stereotypes in another profile. The profiles can be designed in such a way that they can stand alone. But they can also be merged together using PackageMerge to create a new profile that combines the capabilities of both. Vendors and users can then decide whether to apply one profile or the other, or the merged profile to get capabilities resulting from the combination of two or more profiles.

For example, the UPDM profile could integrate with SysML to reuse stereotypes such as Requirement and ViewPoint. UPDM could be designed to use ViewPoint is a manner that is semantically consistent with SysML since SysML already existed. However UPDM could extend ViewPoint with additional properties and associations for its purposes. The UPDM specification could note to users that ViewPoint is a stereotype in UPDM that represents a “placeholder” to ViewPoint in SysML. Users could then apply UPDM to a model, and get UPDM’s ViewPoint capabilities without any coupling with, or need for SysML. UPDM could then provide another compliance point that merges with the SysML profile resulting in stereotypes Requirement and ViewPoint having the capabilities of both profiles. The SysML::ViewPoint would be merged with the UPDM::ViewPoint allowing the shared semantics to be supported without making any changes to the existing model. Users who want UPDM with SysML would then apply this merged profile.

Using XMI to exchange Profiles

A profile is an instance of a UML2 metamodel, not a CMOF metamodel. Therefore the MOF to XMI mapping rules do not directly apply for instances of a profile. Figure 12.4 on page 179 is an example of a mapping between a UML2 Profile and an equivalent CMOF model. This mapping is used as a means to explain and formalize how profiles are serialized and exchanged as XMI. Using this Profile to CMOF mapping, rules for mapping CMOF to XMI can be used indirectly to specify mappings from Profiles to XMI. In the mapping:

- A Profile maps to a CMOF Package.
- The metaclass, Extension::Class, is an instance of a MOF class; Extension::Class maps to itself (that is, a Class in the profile is also the same Class in the corresponding CMOF model; the mapping is the identity transformation).
- A Stereotype maps to a MOF class with the same name and properties.
- An Extension maps to an Association as described in the Semantics sub clause of Profile::Class on page 177.

For a Profile the URI attribute (inherited from package) is used to determine the nsURI to be used to identify instances of the profile in XMI.

[Note: by default the name attribute of the Profile is used for the nsPrefix in XMI but this can be overridden by the CMOF tag org.omg.xmi.nsPrefix].

OMG normative profiles, such as those described in Annex H, follow an OMG normative naming scheme for URIs. For non-standard profiles a recommended convention is:

```
uri = http://<profileParentQualifiedName>/<version>/<profileName>.xmi
```

where:

- `<profileParentQualifiedName>`is the qualified name of the package containing the Profile (if any) with / (forward slash) substituted for ::, and all other illegal XML QName characters removed,
- <*version*> is a version identifier (note that for OMG normative profiles this is a date in the format YYYYMMDD),
- `<profileName>` is the name of the Profile,
- nsPrefix = <*profileName*>,

A profile can be exchanged just like any model, as an XMI schema definition, and models that are extended by a profile can also be interchanged.

Figure 12.6 on page 180 shows a “Home” stereotype extending the “Interface” UML2 metaclass. Figure 12.4 on page 179 illustrates the MOF correspondence for that example, basically by introducing an association from the “Home” MOF class to the “Interface” MOF class. For illustration purposes, we add a property (tagged value definition in UML1.4) called “magic:String” to the “Home” stereotype.

The first serialization below shows how the model in Figure 12.6 on page 180 (instance of the profile and UML2 metamodel) can be exchanged. Note that [UML version number], [MOF version number] and [XMI version number] are placeholders for the published normative version numbers of the referenced normative UML, MOF and XMI specifications respectively, which follow the format YYYYMMDD.

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI
  xmlns:xmi="http:// www.omg.org/spec/XMI/[XMI version number]"
  xmlns:mofext="http://www.omg.org/spec/MOF/[MOF version number]"
  xmlns:uml="http://www.omg.org/spec/UML/[UML version number]">
  <uml:Profile xmi:id="id0" name="HomeExample" metamodelReference="id2"
```

```
      URI="http://HomeExample/20101201/HomeExample.xmi">
      <packageImport xmi:id="id2">
            <importedPackage href="http://www.omg.org/spec/UML/[UML version number]/UML.xmi#_0"/>
      </packageImport>
      <packagedElement xmi:type="uml:Stereotype" xmi:id="id3" name="Home">
            <ownedAttribute xmi:type="uml:Property" xmi:id="id5" name="base_Interface" association="id6">
                  <type href="http://www.omg.org/spec/UML/[UML version number]/UML.xmi#Interface"/>
            </ownedAttribute>
      </packagedElement>
      <packagedElement xmi:type="uml:Extension" xmi:id="id6" name="A_Interface_Home"
            memberEnd="id7 id5">
            <ownedEnd xmi:type="uml:ExtensionEnd" xmi:id="id7" name="extension_Home" type="id3"
                  aggregation="composite">
            </ownedEnd>
      </packagedElement>
  </uml:Profile>
  <mofext:Tag name="org.omg.xmi.nsPrefix" value="HomeExample"/>
</xmi:XMI>
```

Figure 12.9 is an example model that includes an instance of Interface extended by the Home stereotype.

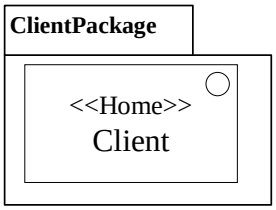


Figure 12.9 - Using the “HomeExample” profile to extend a model

Now the XMI below shows how this model extended by the profile is serialized. A tool importing that XMI file can filter out the elements related to the “HomeExample” schema, if the tool does not have this profile definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI
    xmlns:xmi="http:// www.omg.org/spec/XMI/[XMI version number]"
    xmlns:uml="http://www.omg.org/spec/UML/[UML version number]"
    xmlns:HomeExample="http://HomeExample/20101201/HomeExample.xmi">
    <uml:Package xmi:id="id1" name="ClientPackage">
        <profileApplication xmi:id="id3">
            <appliedProfile
               href="http://HomeExample/20101201/HomeExample.xmi#id0"/>
        </profileApplication>
        <packagedElement xmi:type="uml:Interface" xmi:id="id2" name="Client"/>
    </uml:Package>
<!-- applied stereotypes -->
    <HomeExample:Home xmi:id= "id4" base_Interface="id2"/>
</xmi:XMI>
```

Notation

A Profile uses the same notation as a Package, with the addition that the keyword «profile» is shown before or above the name of the Package. *Profile::metaclassReference* and *Profile::metamodelReference* uses the same notation as *Package::elementImport* and *Package::packageImport*, respectively.

Examples

In Figure 12.10, a simple example of an EJB profile is shown.

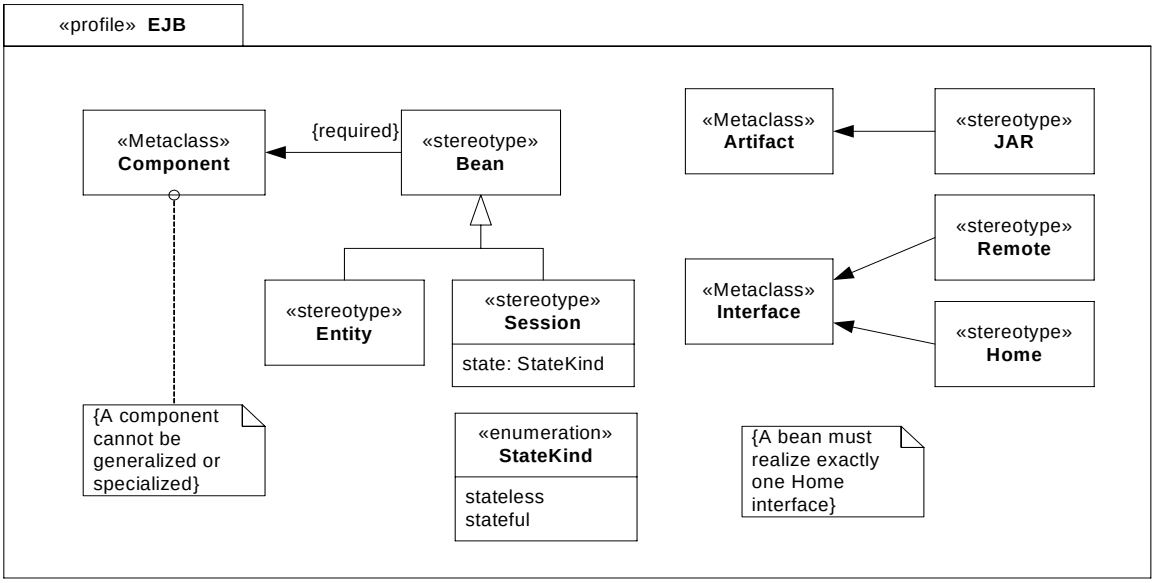


Figure 12.10 - Defining a simple EJB profile

The profile states that the abstract stereotype *Bean* is required to be applied to metaclass *Component*, which means that an instance of either of the concrete subclasses *Entity* and *Session* of *Bean* must be linked to each instance of *Component*. The constraints that are part of the profile are evaluated when the profile has been applied to a package, and these constraints need to be satisfied in order for the model to be well-formed.

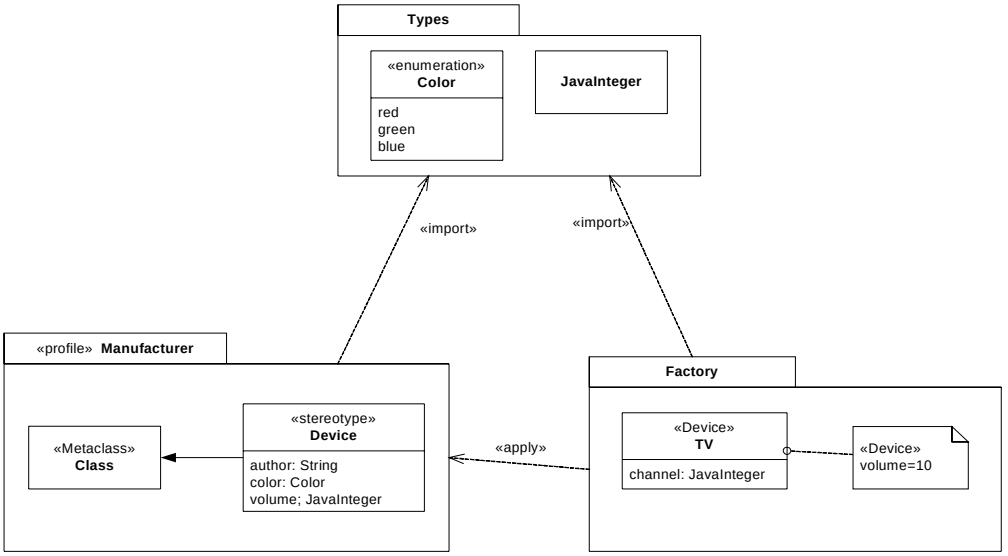


Figure 12.11 - Importing a package from a profile

In Figure 12.11, the package *Types* is imported from the profile *Manufacturer*. The data type *Color* is then used as the type of one of the properties of the stereotype *Device*, just like the predefined type String is also used. Note that the class *JavaInteger* may also be used as the type of a property.

If the profile *Manufacturer* is later applied to a package, then the types from *Types* are not available for use in the package to which the profile is applied unless package *Types* is explicitly imported. This means that for example the class *JavaInteger* can be used as a metaproperty (as part of the stereotype *Device*) but not as an ordinary property (as part of the class *TV*) unless package *Factory* also imports package *Types*. Note how the metaproperty is given a value when the stereotype *Device* is applied to the class *TV*.

12.1.8 ProfileApplication (from Profiles)

A profile application is used to show which profiles have been applied to a package.

Generalizations

- "DirectedRelationship" on page 106

Description

ProfileApplication is a kind of DirectedRelationship that adds the capability to state that a Profile is applied to a Package.

Attributes

- isStrict : Boolean [1] = false
  Specifies whether or not the Profile filtering rules for the metaclasses of the referenced metamodel shall be strictly applied. See the semantics sub clause of "Profile (from Profiles)" on page 184 for further details.

Associations

- importedProfile: Profile [1]
  References the Profiles that are applied to a Package through this ProfileApplication.
  Subsets *PackageImport::importedPackage*
- applyingPackage : Package [1]
  The package that owns the profile application. {*Subsets Element::owner* and *DirectedRelationship::source*}

Constraints

No additional constraints

Semantics

One or more profiles may be applied at will to a package that is created from the same metamodel that is extended by the profile. Applying a profile means that it is allowed, but not necessarily required, to apply the stereotypes that are defined as part of the profile. It is possible to apply multiple profiles to a package as long as they do not have conflicting constraints. Applying a profile means recursively applying all its nested and imported. Stereotypes imported from another profile using ElementImport or PackageImport are added to the namespace members of the importing profile. Stereotypes that are public namespace members of a profile may be used to extend the applicable model elements in packages to which the profile has been applied.

When a profile is applied, instances of the appropriate stereotypes should be created for those elements that are instances of metaclasses with required extensions. The model is not well-formed without these instances.

Once a profile has been applied to a package, it is allowed to remove the applied profile at will. Removing a profile implies that all elements that are instances of elements defined in a profile are deleted. A profile that has been applied cannot be removed unless other applied profiles that depend on it are first removed.

A nested profile can be applied individually. However, the nested profile must specify any required metaclass and/or metamodel references if it contains any stereotypes and may use PackageImport to indicate other dependent packages. Metaclass and/or metamodel references are not inherited from a containing profile.

ProfileApplication makes stereotype names visible to the referenced metamodel, not the model the profile is applied to. ProfileApplication is not a kind of PackageImport because of this crossing of metamodel levels. As with package import, profile application does not expose the names of nested profiles.

Note – The removal of an applied profile leaves the instances of elements from the referenced metamodel intact. It is only the instances of the elements from the profile that are deleted. This means that for example a profiled UML model can always be interchanged with another tool that does not support the profile and be interpreted as a pure UML model.

Notation

The names of Profiles are shown using a dashed arrow with an open arrowhead from the package to the applied profile. The keyword «apply» is shown near the arrow.

If multiple applied profiles have stereotypes with the same name, it may be necessary to qualify the name of the stereotype (with the profile name).

Examples

Given the profiles *Java* and *EJB*, Figure 12.12 shows how these have been applied to the package *WebShopping*.

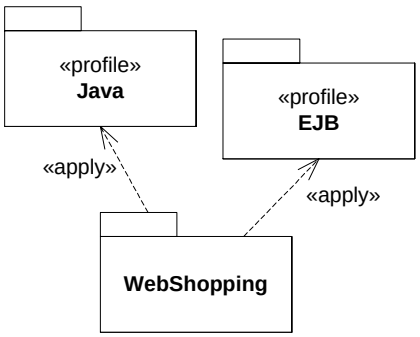


Figure 12.12 - Profiles applied to a package

12.1.9 Stereotype (from Profiles)

A stereotype defines how an existing metaclass may be extended, and enables the use of platform or domain specific terminology or notation in place of, or in addition to, the ones used for the extended metaclass.

Generalizations

- InfrastructureLibrary::Constructs::Class

- “Class (from Profiles)” on page 176

Description

Stereotype is a kind of Class that extends Classes through Extensions.

Just like a class, a stereotype may have properties, which may be referred to as tag definitions. When a stereotype is applied to a model element, the values of the properties may be referred to as tagged values.

Attributes

No additional attributes

Associations

- icon : Image [*]
  Stereotype can change the graphical appearance of the extended model element by using attached icons. When this association is not null, it references the location of the icon content to be displayed within diagrams presenting the extended model elements. {Subsets Element::ownedElement}
- /profile : Profile [1]
  The profile that directly or indirectly contains this stereotype.

Constraints

[1] A Stereotype may only generalize or specialize another Stereotype.

```
generalization.general->forAll(e | e.oclIsKindOf(Stereotype)) and
generalization.specific->forAll(e | e.oclIsKindOf(Stereotype))
```

[2] Stereotype names should not clash with keyword names for the extended model element.

[3] A stereotype must be contained, directly or indirectly, in a profile.

```
profile = self.containingProfile()
```

Additional Operations

[1] The query containingProfile() returns the closest profile directly or indirectly containing this package (or this package itself, if it is a profile).

```
Package::containingProfile(): Profile
    result = self.namespace.oclAsType(Package).containingProfile()
```

Semantics

A stereotype is like a limited kind of metaclass that cannot be used by itself, but must always be used in conjunction with one of the metaclasses it extends. Each stereotype may extend one or more classes through extensions as part of a profile. Similarly, a class may be extended by one or more stereotypes.

An instance “S” of Stereotype is like a kind of metaclass that extends other metaclasses through association (Extension) rather than generalization/specialization. Relating it to a metaclass “C” from the reference metamodel (typically UML) using an “Extension” (which is a specific kind of association), signifies that model elements of type C can be extended by an instance of “S” (see example in Figure 12.13). At the model level (such as in Figure 12.18) instances of “S” are related to “C” model elements (instances of “C”) by links (occurrences of the association/extension from “S’ to “C”).

Any metaclassReference or model element from the metamodelReference of the closest profile directly or indirectly containing a stereotype can be extended by the stereotype. For example in UML, States, Transitions, Activities, Use cases, Components, Attributes, Dependencies, etc. can all be extended with stereotypes if the metamodelReference is UML. A stereotype may be contained in a package in which case the metaclass and/or metamodel references available for extension are those of the closest parent profile containing the package.

Stereotype specializes Class which may be merged with Class from InfrastructureLibrary and Superstructure in different UML2 compliance levels. This adds a number of additional properties to class such as isLeaf, isAbstract, and ownedAttributes needed to provide the properties of the stereotype. These properties have the same meaning in Stereotypes as they do in Class. Tool vendors may choose to support extensibility that includes owned operations and behaviors, but are not required to do so. Tools must however support Stereotype ownedAttributes.

A Stereotype may be contained in a Profile or Package which defines the namespace for the Stereotype. When profiles are applied to a Package, the available stereotypes for use as extensions are defined by the applied profiles, and theses stereotypes can be identified by the fully qualified name if needed in order to distinguish stereotypes with the same name in different profiles or packages. Package and element import can be used to allow the use of unqualified names. Stereotypes directly owned by an applied profile (ownedStereotype) may be used without qualified names.

Notation

A Stereotype uses the same notation as a Class, with the addition that the keyword «stereotype» is shown before or above the name of the Class.

When a stereotype is applied to a model element (an instance of a stereotype is linked to an instance of a metaclass), the name of the stereotype is shown within a pair of guillemets above or before the name of the model element, or where the name would appear if the name is omitted or not displayed. For model elements that do not have names but do have a graphical representation, unless specifically stated elsewhere, the stereotypes can be displayed within a pair of guillemets near the upper right corner of the graphical representation. If multiple stereotypes are applied, the names of the applied stereotypes are shown as a comma-separated list with a pair of guillemets. When the extended model element has a keyword, then the stereotype name will be displayed close to the keyword, within separate guillemets (example: «interface» «Clock»).

Normally a stereotype?s name and the name of its applications start with upper-case letters, to follow the convention for naming classes. Domain-specific profiles may use different conventions. Matching between the names of stereotype definitions and uses is case-insensitive, so naming stereotype applications with lower-case letters where the stereotypes are defined using upper-case letters is valid, although stylistically obsolete.

Presentation Options

If multiple stereotypes are applied to an element, it is possible to show this by enclosing each stereotype name within a pair of guillemets and listing them after each other. A tool can choose whether it will display stereotypes or not. In particular, some tools can choose not to display “required stereotypes,” but to display only their attributes (tagged values) if any.

The values of a stereotyped element can be shown in one of the following three ways:

- As part of a comment symbol connected to the graphic node representing the model element
- In separate compartments of a graphic node representing that model element.
- Above the name string within the graphic node or, else, before the name string.

In the case where a compartment or comment symbol is used, the stereotype name may shown in guillemets before the name string in addition to being included in the compartment or comment.

The values are displayed as name-value pairs:

<namestring> '=' <valuestring>

If a stereotype property is multi-valued then the <valuestring> is displayed as a comma-separated list:

<valuestring> ::= <value> [',' <value>]*

Certain values have special display rules:

- As an alternative to a name-value pair, when displaying the values of Boolean properties, diagrams may use the convention that if the <namestring> is displayed then the value is True, otherwise the value is False.
- If the value is the name of a NamedElement, then, optionally, its qualifiedName can be used.

If compartments are used to display stereotype values, then an additional compartment is required for each applied stereotype whose values are to be displayed. Each such compartment is headed by the name of the applied stereotype in guillemets. Any graphic node may have these compartments.

Within a comment symbol, or, if displayed before or above the symbols's <namestring>, the values from a specific stereotype are optionally preceded with the name of the applied stereotype within a pair of guillemets. This is useful if values of more than one applied stereotype should be shown.

When displayed in compartments or in a comment symbol, at most one name-value pair can appear on a single line. When displayed above or before a <namestring>, the name-value pairs are separated by semicolons and all pairs for a given stereotype are enclosed in braces.

If the extension end is given a name, this name can be used in lieu of the stereotype name within the pair of guillemets when the stereotype is applied to a model element.

It is possible to attach a specific notation to a stereotype that can be used in lieu of the notation of a model element to which the stereotype is applied.

Icon presentation

When a stereotype includes the definition of an icon, this icon can be graphically attached to the model elements extended by the stereotype. Every model element that has a graphical presentation can have an attached icon. When model elements are graphically expressed as:

- Boxes (see Figure 12.14 on page 196): the box can be replaced by the icon, and the name of the model element appears below the icon. This presentation option can be used only when a model element is extended by one single stereotype and when properties of the model element (i.e., attributes, operations of a class) are not presented. As another option, the icon can be presented in a reduced shape, inside and on top of the box representing the model element. When several stereotypes are applied, several icons can be presented within the box.
- Links: the icon can be placed close to the link.
- Textual notation: the icon can be presented to the left of the textual notation.

Several icons can be attached to a stereotype. The interpretation of the different attached icons in that case is a semantic variation point. Some tools may use different images for the icon replacing the box, for the reduced icon inside the box, for icons within explorers, etc. Depending on the image format, other tools may choose to display one single icon into different sizes.

Some model elements are already using an icon for their default presentation. A typical example of this is the Actor model element, which uses the “stickman” icon. In that case, when a model element is extended by a stereotype with an icon, the stereotype’s icon replaces the default presentation icon within diagrams.

Examples

In Figure 12.13, a simple stereotype *Clock* is defined to be applicable at will (dynamically) to instances of the metaclass Class.

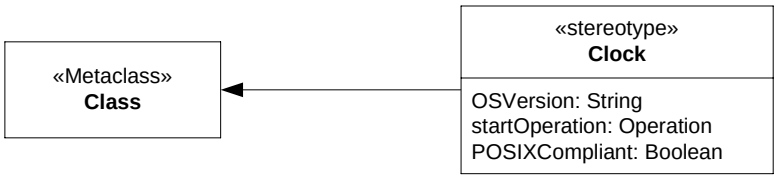


Figure 12.13 - Defining a stereotype

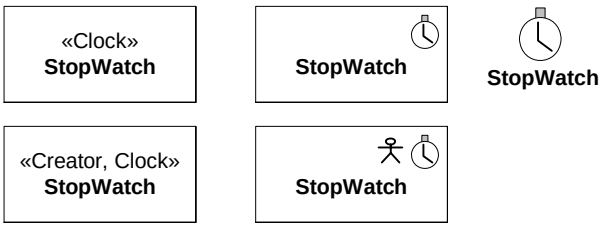


Figure 12.14 - Presentation options for an extended class

In Figure 12.15, an instance specification of the example in Figure 12.13 is shown. Note that the extension end must be composite, and that the derived isRequired” attribute in this case is false. Figure 12.15 shows the repository schema of the stereotype “clock” defined in Figure 12.13. In this schema, the extended instance (:Class; “name = Class”) is defined in the UML 2 (reference metamodel) repository. In a UML modeling tool these extended instances referring to the UML 2 standard would typically be in a “read only” form, or presented as proxies to the metaclass being extended.

(It is therefore still at the same meta-level as UML, and does not show the instance model of a model extended by the stereotype. An example of this is provided in Figure 12.17 and Figure 12.18.) The Semantics sub clause of the Extension concept explains the MOF equivalent, and how constraints can be attached to stereotypes.

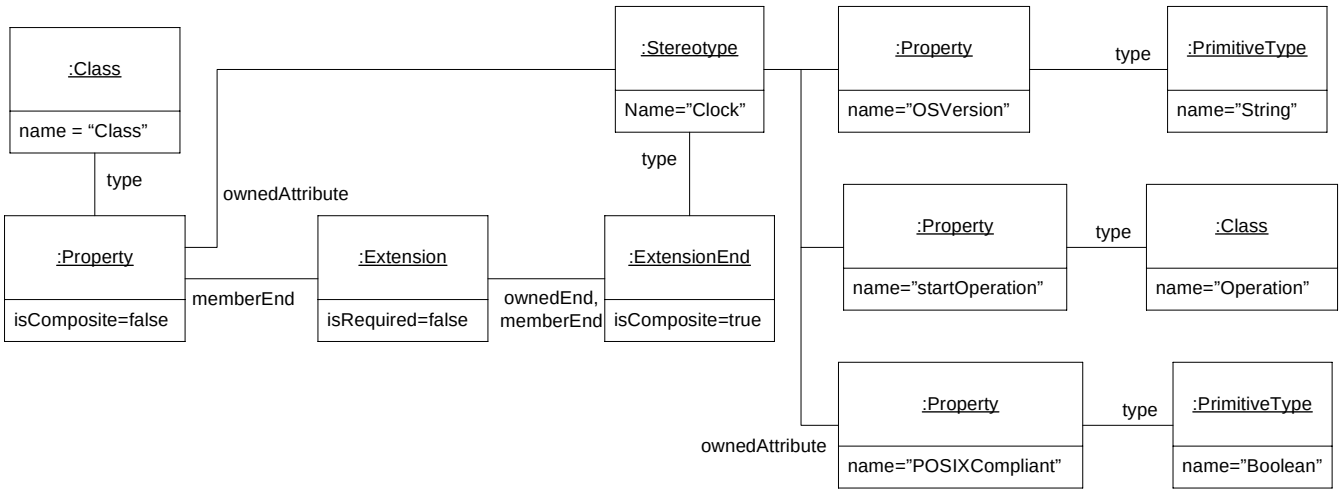


Figure 12.15 - An instance specification when defining a stereotype

Figure 12.16 shows how the same stereotype *Clock* can extend either the metaclass Component or the metaclass Class. It also shows how different stereotypes can extend the same metaclass.

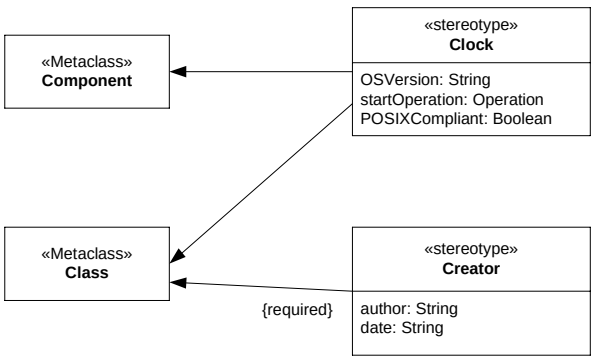


Figure 12.16 - Defining multiple stereotypes on multiple stereotypes

Figure 12.17 shows how the stereotype *Clock*, as defined in Figure 12.16, is applied to a class called *StopWatch*.

Figure 12.17 - Using a stereotype

Figure 12.18 shows an example instance model for when the stereotype *Clock* is applied to a class called *StopWatch*. The extension between the stereotype and the metaclass results in a link between the instance of stereotype *Clock* and the (user-defined) class *StopWatch*.

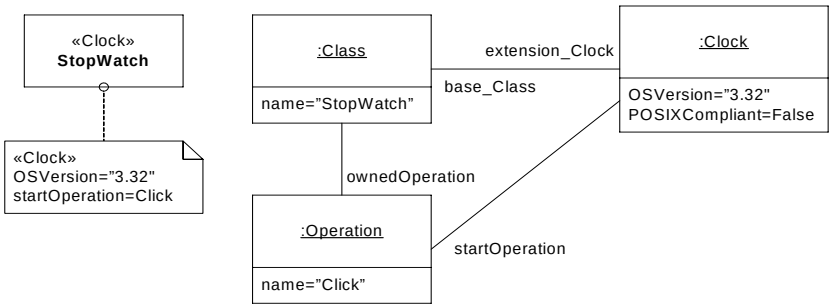


Figure 12.18 - Showing values of stereotypes and a simple instance specification

Next, two stereotypes, *Clock* and *Creator*, are applied to the same model element, as shown in Figure 12.19. Note that the attribute values of each of the applied stereotypes can be shown in a comment symbol attached to the model element.

Figure 12.19 - Using stereotypes and showing values

Finally, two more alternative notational forms are shown in Figure 12.20.

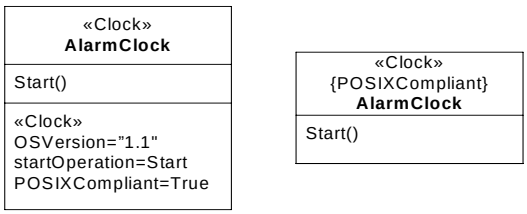


Figure 12.20 - Other notational forms for depicting stereotype values

Changes from previous UML

In UML 1.3, tagged values could extend a model element without requiring the presence of a stereotype. In UML 1.4, this capability, although still supported, was deprecated, to be used only for backward compatibility reasons. In UML 2, a tagged value can only be represented as an attribute defined on a stereotype. Therefore, a model element must be extended by a stereotype in order to be extended by tagged values. However, the “required” extension mechanism can, in effect, provide the 1.3 capability, since a tool can in those circumstances automatically define a stereotype to which “unattached” attributes (tagged values) would be attached.

13 PrimitiveTypes

The PrimitiveTypes package is a top level package that contains a number of predefined types that can be imported and used when defining the abstract syntax of metamodels.

Figure 13.1 - The Core package is owned by the InfrastructureLibrary package, and contains several subpackages

13.1 PrimitiveTypes Package

The PrimitiveTypes package is a top level package that is independent of any other package. The package defines a set of reusable primitive types that are commonly used in the definition of metamodels. The InfrastructureLibrary and the UML metamodel are examples of metamodels using the PrimitiveTypes package to define their primitive data.

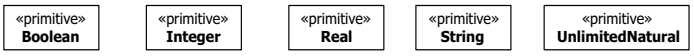


Figure 13.2 - The classes defined in the PrimitiveTypes package

13.1.1 Boolean

A Boolean type is used for logical expression, consisting of the predefined values *true* and *false*.

Description

Boolean is an instance of PrimitiveType. In the metamodel, Boolean defines an enumeration that denotes a logical condition. Its enumeration literals are:

- true — The Boolean condition is satisfied.
- false — The Boolean condition is not satisfied.

It is used for Boolean attribute and Boolean expressions in the metamodel, such as OCL expression.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Boolean is an instance of PrimitiveType.

Notation

Boolean will appear as the type of attributes in the metamodel. Boolean instances will be values associated to slots, and can have literally the following values: *true* or *false*.

Examples

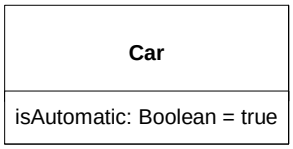


Figure 13.3 - An example of a Boolean attribute

13.1.2 Integer

An integer is a primitive type representing integer values.

Description

An instance of Integer is an element in the (infinite) set of integers (…-2, -1, 0, 1, 2…). It is used for integer attributes and integer expressions in the metamodel.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Integer is an instance of PrimitiveType.

Notation

Integer will appear as the type of attributes in the metamodel. Integer instances will be values associated to slots such as 1, -5, 2, 34, 26524, etc.

Examples

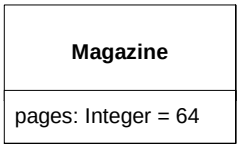


Figure 13.4 - An example of an integer attribute

13.1.3 Real

A real is a primitive type representing the mathematical concept of real.

Description

An instance of Real is an element in the infinite set of real numbers. It is used for real attributes and real expressions in the models.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Real is an instance of PrimitiveType.

Notation

Real will appear as the type of attributes in the metamodel. Real instances will be literals associated to real-typed slots such as 1, -5, 0.3, 34.75, 26524.2339, 2.99E5, etc.

Examples

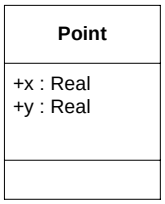


Figure 13.5 - An example of a real attribute

13.1.4 String

A string is a sequence of characters in some suitable character set used to display information about the model. Character sets may include non-Roman alphabets and characters.

Description

An instance of String defines a piece of text. The semantics of the string itself depends on its purpose, it can be a comment, computational language expression, OCL expression, etc. It is used for String attributes and String expressions in the metamodel.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

String is an instance of PrimitiveType.

Notation

String appears as the type of attributes in the metamodel. String instances are values associated to slots. The value is a sequence of characters surrounded by double quotes ("). It is assumed that the underlying character set is sufficient for representing multibyte characters in various human languages; in particular, the traditional 8-bit ASCII character set is insufficient. It is assumed that tools and computers manipulate and store strings correctly, including escape conventions for special characters, and this document will assume that arbitrary strings can be used.

A string is displayed as a text string graphic. Normal printable characters should be displayed directly. The display of nonprintable characters is unspecified and platform-dependent.

Examples

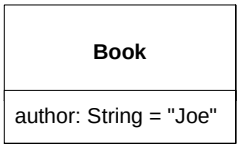


Figure 13.6 - An example of a string attribute

13.1.5 UnlimitedNatural

An unlimited natural is a primitive type representing unlimited natural values.

Description

An instance of UnlimitedNatural is an element in the (infinite) set of naturals (0, 1, 2…). The value of infinity is shown using an asterisk (‘*’).

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

UnlimitedNatural is an instance of PrimitiveType.

Notation

UnlimitedNatural will appear as the type of upper bounds of multiplicities in the metamodel. UnlimitedNatural instances will be values associated to slots such as 1, 5, 398475, etc. The value infinity may be shown using an asterisk (‘*’).

Examples

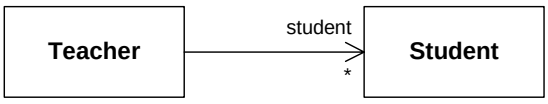


Figure 13.7 - An example of an unlimited natural

Subpart III - Annexes

Annexes include:

A - XMI Serialization and Schema

B - Support for Model Driven Architecture

Annex A: XMI Serialization and Schema

(normative)

UML 2 models are serialized in XMI according to the rules specified by the MOF 2: XMI Mapping Specification.

XMI allows the use of tags to tailor the schemas that are produced and the documents that are produced using XMI. The following are the tag settings that appear in the XMI of the *InfrastructureLibrary*:

- tag “org.omg.xmi.nsPrefix” set to “uml” (for packages L0, LM)

The following are the tag settings that appear in the XMI of the *PrimitiveTypes* package:

- tag “org.omg.xmi.nsPrefix” set to “primitives”
- tag “org.omg.xmi.schemaType” set to “http://www.w3.org/2001/XMLSchema#integer” (for element PrimitiveTypes-Integer)
- tag “org.omg.xmi.schemaType” set to “http://www.w3.org/2001/XMLSchema#boolean” (for element PrimitiveTypes-Boolean)
- tag “org.omg.xmi.schemaType” set to “http://www.w3.org/2001/XMLSchema#double” (for element PrimitiveTypes-Real)

No other tags are explicitly set, which means they assume their default values as documented in the MOF 2 XMI Mappings Specification.

Annex B: Support for Model Driven Architecture

(normative)

The OMG's Model Driven Architecture (MDA) initiative is an evolving conceptual architecture for a set of industry-wide technology specifications that will support a model-driven approach to software development. Although MDA is not itself a technology specification, it represents an approach and a plan to achieve a cohesive set of model-driven technology specifications.

The MDA initiative was initiated after the UML 2 RFPs were issued. However, as noted in the OMG's *Executive Overview* of MDA (www.omg.org/mda/executive_overview.htm): "[MDA] is built on the solid foundation of well-established OMG standards, including: Unified Modeling Language™ (UML™), the ubiquitous modeling notation used and supported by every major company in the software industry; XML Metadata Interchange (XMI™), the standard for storing and exchanging models using XML; and CORBA™, the most popular open middleware standard." Consequently, it is expected that this major revision to UML will play an important role in furthering the goals of MDA.

The OMG Object Reference Model Subcommittee has produced MDA Guide (the latest version of which is referenced from www.omg.org/mda) which is the official, commonly agreed upon, definition of MDA. This MDA Guide draft characterizes MDA as follows:

> *"MDA provides an approach for and enables tools to be provided for:*
> *- specifying a system independently of the platform that supports it,*
> *- specifying platforms or selecting existing platform specifications,*
> *- choosing a particular platform for the system, and*
> *- transforming the system specification into one for the chosen particular platform."*

In addition, this MDA Guide draft and many other MDA documents commonly refer to a "UML family of languages," which is described in the MDA Guide as: "Extensions to the UML language [that] will be standardized for specific purposes. Many of these will be designed specifically for use in MDA."

The following sections explain how UML 2 supports the most prominent concepts in the evolving MDA vision.

- **Family of languages:** UML is a general purpose language, that is expected to be customized for a wide variety of domains, platforms and methods. Towards that end, this UML specification refines UML 1.x's Profile mechanism so that it is more robust and flexible, and significantly easier to implement and apply. Consequently, it can be used to customize UML dialects for various domains (e.g., finance, telecommunications, aerospace), platforms (e.g., J2EE, .NET), and methods (e.g., Unified Process, Agile methods). For those whose customization requirements exceed these common anticipated usages, and who want to define their new languages via metamodels, the InfrastructureLibrary is intended to be reused by MOF 2. Tools that implement MOF 2 will allow users to define entirely new languages via metamodels.
- **Specifying a system independently of the platform that supports it:** As was the case with its predecessor, the general purpose UML 2 specification is intended to be used with a wide range of software methods. Consequently, it includes support for software methods that distinguish between analysis or logical models, and design or physical models. Since analysis or logical models are typically independent of implementation and platform specifics, they can be considered "Platform Independent Models" (PIMs), consistent with the evolving MDA terminology. Some of the proposed improvements to UML that will make it easier for modelers to specify Platform Independent Models include the ability to model logical as well as physical Classes and Components, consistent with either a class-based or component-based approach.

- **Specifying platforms:** Although UML 1.x provided extremely limited support for modeling Platform Specific Models (PSMs, the complement of PIMs), this specification offers two significant improvements. First, the revised Profile mechanism allows modelers to more efficiently customize UML for target platforms, such as J2EE or .NET. (Examples of J2EE/EJB or .NET/COM micro-profiles can be found in the UML Superstructure Specification.) Secondly, the constructs for specifying component architectures, component containers (execution runtime environments), and computational nodes are significantly enhanced, allowing modelers to fully specify target implementation environments.
- **Choosing a particular platform for the system:** This is considered a method or approach requirement, rather than a modeling requirement. Consequently, we will not address it here.
- **Transforming the system specification into one for a particular platform:** This refers to the transformation of a Platform Independent Model into a Platform Specific Model. The UML Superstructure Specification specifies various relationships that can be used to specify the transformation of a PIM to a PSM, including Realization, Refine, and Trace. However, the specific manner in which these transformations are used will depend upon the profiles used for the PSMs involved, as well as the method or approach applied to guide the transformation process. Consequently, we will not address it further here.

Annex C: UML XMI Documents

(normative)

UML defined in UML:

- http://www.omg.org/spec/UML/20110701/UML.xmi

INDEX