UML 2.0 Infrastructure Specification

This OMG document replaces the submission document (ad/03-01-01) and the Draft Adopted specification (ptc/03-07-05). It is an OMG Final Adopted Specification and is currently in the finalization phase. Comments on the content of this document are welcomed, and should be directed to *issues@omg.org* by November 7, 2003.

You may view the pending issues for this specification from the OMG revision issues web page *http://www.omg.org/issues/*; however, at the time of this writing there were no pending issues.

The FTF Recommendation and Report for this specification will be published on April 30, 2004. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

OMG Adopted Specification
ptc/03-09-15

Date: September 2003

Unified Modeling Language (UML) Specification: Infrastructure

version 2.0

ptc/03-09-15

Copyright © 2002, Adaptive Ltd.
Copyright © 1997-2003, Object Management Group
Copyright © 2001-2003 U2 Partners (www.u2-partners.org).

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

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

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

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, 250 First Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI® and IIOP® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBAservices™, CORBAfacilities™, CORBAmed™, CORBAnet™, Integrate 2002™, Middleware That's Everywhere™, UML™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ and the XMI Logo™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

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.

ISSUE REPORTING

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

Table of Contents

1 Scope

This *UML 2.0: Infrastructure* is the first of two complementary specifications that represent a major revision to the Object Management Group's Unified Modeling Language (UML), for which the previous current version was UML v1.5. The second specification, which uses the architectural foundation provided by this specification, is the *UML 2.0: Superstructure (ptc/03-08-02)*.

The *UML 2.0: Infrastructure* defines the foundational language constructs required for UML 2.0. It is complemented by *UML 2.0: Superstructure*, which defines the user level constructs required for UML 2.0.

Editorial Comment: The FTF needs to review and complete this section -- this version was derived from the "Introduction" section of the Preface in the Draft Adopted Specification

---

2 Conformance

Editorial Comment: The FTF needs to review and complete this section -- this version was derived from the "Introduction" section of the Preface in the Draft Adopted Specification

---

The basic units of compliance for UML are the packages which define the UML metamodel. Unless otherwise qualified, complying with a package requires complying with its abstract syntax, well-formedness rules, semantics and notation.

In the case of the UML Infrastructure, its InfrastructureLibrary is intended to be flexibly reused by UML2, MOF2, CWM2 and future metmodels. In order to maximize its flexibility for reuse, each subpackage in the InfrastructureLibrary constitutes a separate compliance point.

All metamodels that reuse the InfrastructureLibrary should clearly specify which packages they reuse, and further clarify which packages are imported without change, and which packages are imported and extended via specialization. For example, the UML::AuxiliaryConstructs::Profiles package imports the InfrastructureLibrary::Profiles package without change, whereas the UML::Classes::Kernel package imports all the Infrastructure::Core subpackages and extends some of their classes via specialization. In the latter case, specific specializations needs to be clearly defined.

The following table summarizes the compliance points of the *UML 2.0: Infrastructure*, where the following compliance options are valid:

no compliance	Implementation does not comply with the abstract syntax, well-formedness rules, semantics and notation of the package.
partial compliance	Implementation partially complies with the abstract syntax, well-formedness rules, semantics and notation of the package.
compliant compliance	Implementation fully complies with the abstract syntax, well-formedness rules, semantics and notation of the package.
interchange compliance	Implementation fully complies with the abstract syntax, well-formedness rules, semantics, notation and XMI schema of the package.

Table 1 - Summary of Compliance Points

Compliance Point	Valid Options
Core::Abstractions::BehavioralFeatures	no, partial, complete, interchange
Core::Abstractions::Changeabilities	no, partial, complete, interchange
Core::Abstractions::Classifiers	no, partial, complete, interchange
Core::Abstractions::Comments	no, partial, complete, interchange
Core::Abstractions::Constraints	no, partial, complete, interchange
Core::Abstractions::Elements	no, partial, complete, interchange
Core::Abstractions::Expressions	no, partial, complete, interchange
Core::Abstractions::Generalizations	no, partial, complete, interchange
Core::Abstractions::Instances	no, partial, complete, interchange
Core::Abstractions::Literals	no, partial, complete, interchange
Core::Abstractions::Multiplicities	no, partial, complete, interchange
Core::Abstractions::MultiplicityExpressions	no, partial, complete, interchange
Core::Abstractions::Namespaces	no, partial, complete, interchange
Core::Abstractions::Ownerships	no, partial, complete, interchange
Core::Abstractions::Redefinitions	no, partial, complete, interchange
Core::Abstractions::Relationships	no, partial, complete, interchange
Core::Abstractions::StructuralFeatures	no, partial, complete, interchange
Core::Abstractions::Super	no, partial, complete, interchange
Core::Abstractions::Visibilities	no, partial, complete, interchange
Core::Basic	no, partial, complete, interchange
Core::Constructs	no, partial, complete, interchange
Core::PrimitiveTypes	no, partial, complete, interchange
Core::Profiles	no, partial, complete, interchange

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.

- UML 2.0: Superstructure Specification
- MOF 2.0: Core Specification
- MOF 2.0: XMI Mapping Specification

Editorial Comment: The FTF needs to review and complete this section

---

4 Terms and Definitions

Editorial Comment: The FTF needs to review and complete this section -- the version in this document was produced by literal copying of the contents of the Glossary from the UML 2.0 Superstructure Draft Adopted Spec into this section.

For the purposes of this specification, the terms and definitions given in the normative references and the following apply.

(Note: The following conventions are used in the term definitions below:

- The entries usually begin with a lowercase letter. An initial uppercase letter is used when a word is usually capitalized in standard practice. Acronyms are all capitalized, unless they traditionally appear in all lowercase.
- When one or more words in a multi-word term is enclosed in brackets, it indicates that those words are optional when referring to the term. For example, *use case [class]* may be referred to as simply *use case*.
- A phrase of the form “Contrast: <term>” refers to a term that has an opposed or substantively different meaning.
- A phrase of the form “See: <term>” refers to a related term that has a similar, but not synonymous meaning.
- A phrase of the form “Synonym: <term>” indicates that the term has the same meaning as another term, which is referenced.
- A phrase of the form “Acronym: <term>” indicates that the term is an acronym. The reader is usually referred to the spelled-out term for the definition, unless the spelled-out term is rarely used.)

abstract class
A class that cannot be directly instantiated. Contrast: *concrete class*.

abstraction
The result of empasizing certain features of a thing while de-emphasizing other features that are not relative. An abstraction is definedrelative to the perspective of the viewer.

action
A fundamental unit of behavior specification that represents some transformation or processing in the modeled system, be it a computer system or a real-world system. Actions are contained in activities, which provide their context. See: *activity.*

action sequence
An expression that resolves to a sequence of actions.

action state
A state that represents the execution of an atomic action, typically the invocation of an operation.

activation
The initiation of an action execution.

active class
A class whose instances are active objects. See: *active object.*

active object
An object that may execute its own behavior without requiring method invocation. This is sometimes referred to as "the object having its own thread of control." The points at which an active object responds to communications from other objects are determined solely by the behavior of the active object and not by the invoking object. This implies that an active object is both autonomous and interactive to some degree. See: *active class, thread.*

activity
A specification of parameterized behavior that is expressed as a flow of execution via a sequencing of subordinate units (whose primitive elements are individual actions). See *actions*.

activity diagram
A diagram that depicts behavior using a control and data-flow model.

actor
A construct that is employed in use cases that define a role that a user or any other system plays when interacting with the system under consideration. It is a type of entity that interacts, but which is itself external to the subject. Actors may represent human users, external hardware, or other subjects. An actor does not necessarily represent a specific physical entity. For instance, a single physical entity may play the role of several different actors and, conversely, a given actor may be played by multiple physical entities.

actual parameter
Synonym: *argument*.

aggregate
A class that represents the "whole" in an aggregation (whole-part) relationship. See: *aggregation*.

aggregation
A special form of association that specifies a whole-part relationship between the aggregate (whole) and a component part. See: *composition*.

analysis
The phase of the system development process whose primary purpose is to formulate a model of the problem domain that is independent of implementation considerations. Analysis focuses on what to do; design focuses on how to do it. Contrast: *design*.

analysis time
Refers to something that occurs during an analysis phase of the software development process. See: *design time, modeling time.*

argument
A binding for a parameter that is resolved later. An independent variable.

artifact
A physical piece of information that is used or produced by a development process. Examples of Artifacts include models, source files, scripts, and binary executable files. An artifact may constitute the implementation of a deployable component. Synonym: *product*. Contrast: *component*.

association
A relationship that may occur between instances of classifiers.

association class
A model element that has both association and class properties. An association class can be seen as an association that also has class properties, or as a class that also has association properties.

association end
The endpoint of an association, which connects the association to a classifier.

attribute
A structural feature of a classifier that characterizes instances of the classifier. An attribute relates an instance of a classifier to a value or values through a named relationship.

auxiliary class
A stereotyped class that supports another more central or fundamental class, typically by implementing secondary logic or control flow. Auxiliary classes are typically used together with focus classes, and are particularly useful for specifying the secondary business logic or control flow of components during design. See also: *focus*.

behavior
The observable effects of an operation or event, including its results. It specifies the computation that generates the effects of the behavioral feature. The description of a behavior can take a number of forms: interaction, statemachine, activity, or procedure (a set of actions).

behavior diagram
A form of diagram that depict behavioral features.

behavioral feature
A dynamic feature of a model element, such as an operation or method.

behavioral model aspect
A model aspect that emphasizes the behavior of the instances in a system, including their methods, collaborations, and state histories.

binary association
An association between two classes. A special case of an n-ary association.

binding
The creation of a model element from a template by supplying arguments for the parameters of the template.

boolean
An enumeration whose values are true and false.

boolean expression
An expression that evaluates to a boolean value.

cardinality
The number of elements in a set. Contrast: *multiplicity*.

child
In a generalization relationship, the specialization of another element, the parent. See: *subclass*, *subtype*. Contrast: *parent*.

call
An action state that invokes an operation on a classifier.

class
A classifier that desctibes of a set of objects that share the same specifications of features, constraints, and semantics.

classifier
A collection of instances that have something in common. A classifier can have features that characterize its instances. Classifiers include interfaces, classes, datatypes, and components.

classification
The assignment of an instance to a classifier. See *dynamic classification*, *multiple classification* and *static classification*.

class diagram
A diagram that shows a collection of declarative (static) model elements, such as classes, types, and their contents and relationships.

client
A classifier that requests a service from another classifier. Contrast: *supplier.*

collaboration
The specification of how an operation or classifier, such as a use case, is realized by a set of classifiers and associations playing specific roles used in a specific way. The collaboration defines an interaction. See: *interaction*.

collaboration occurrence
A particular use of a collaboration to explain the relationships between the parts of a classifier or the properties of an operation. It may also be used to indicate how a collaboration represents a classifier or an operation. A collaboration occurrence indicates a set of roles and connectors that cooperate within the classifier or operation according to a given collaboration, indicated by the type of the collaboration occurrence. There may be multiple occurrences of a given collaboration within a classifier or operation, each involving a different set of roles and connectors. A given role or connector may be involved in multiple occurrences of the same or different collaborations. See: *collaboration*.

communication diagram
A diagram that focuses on the interaction between lifelines where the architecture of the internal structure and how this corresponds with the message passing is central. The sequencing of messages is given through a sequence numberering scheme. Sequence diagrams and communication diagrams express similar information, but show it in different ways. See: *sequence diagram*.

compile time
Refers to something that occurs during the compilation of a software module. See: *modeling time, run time*.

component
A modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment. A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics).

component diagram
A diagram that shows the organizations and dependencies among components.

composite
A class that is related to one or more classes by a composition relationship. See: *composition*.

composite aggregation
Synonym: *composition*.

composite state
A state that consists of either concurrent (orthogonal) substates or sequential (disjoint) substates. See: *substate*.

composite structure diagram
A diagram that depicts the internal structure of a classifier, including the interaction points of the classifier to other

parts of the system. It shows the configuration of parts that jointly perform the behavior of the containing classifier. The architecture diagram specifies a set of instances playing parts (roles), as well as their required relationships given in a particular context.

composition
A form of aggregation which requires that a part instance be included in at most one composite at a time, and that the composite object is responsible for the creation and destruction of the parts. Composition may be recursive. Synonym: *composite aggregation*.

concrete class
A class that can be directly instantiated. Contrast: *abstract class.*

concurrency
The occurrence of two or more activities during the same time interval. Concurrency can be achieved by interleaving or simultaneously executing two or more threads. See: *thread.*

concurrent substate
A substate that can be held simultaneously with other substates contained in the same composite state. See: *composite state.* Contrast: *disjoint substate.*

connectable element
An abstract metaclass representing model elements which may be linked via connector. See: *connector*.

connector
A link that enables communication between two or more instances. The link may be realized by something as simple as a pointer or by something as complex as a network connection.

constraint
A semantic condition or restriction. It can be expressed in natural language text, mathematically formal notation, or in a machine-readable language for the purpose of declaring some of the semantics of a model element.

container
1. An instance that exists to contain other instances, and that provides operations to access or iterate over its contents. (for example, arrays, lists, sets).
2. A component that exists to contain other components.

containment hierarchy
A namespace hierarchy consisting of model elements, and the containment relationships that exist between them. A containment hierarchy forms a graph.

context
A view of a set of related modeling elements for a particular purpose, such as specifying an operation.

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

delegation
The ability of an object to issue a message to another object in response to a message. Delegation can be used as an alternative to inheritance. Contrast: *inheritance*.

dependency
A relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).

deployment diagram
A diagram that depicts the execution architecture of systems. It represents system artifacts as nodes, which are connected through communication paths to create network systems of arbitrary complexity. Nodes are typically

defined in a nested manner, and represent either hardware devices or software execution environments. See: *component diagrams*.

derived element
A model element that can be computed from another element, but that is shown for clarity or that is included for design purposes even though it adds no semantic information.

design
The phase of the system development process whose primary purpose is to decide how the system will be implemented. During design strategic and tactical decisions are made to meet the required functional and quality requirements of a system.

design time
Refers to something that occurs during a design phase of the system development process. See: *modeling time*. Contrast: *analysis time*.

development process
A set of partially ordered steps performed for a given purpose during system development, such as constructing models or implementing models.

diagram
A graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements). UML supports the diagrams listed in Appendix A.

disjoint substate
A substate that cannot be held simultaneously with other substates contained in the same composite state. See: *composite state*. Contrast: *concurrent substate*.

distribution unit
A set of objects or components that are allocated to a process or a processor as a group. A distribution unit can be represented by a run-time composite or an aggregate.

domain
An area of knowledge or activity characterized by a set of concepts and terminology understood by practitioners in that area.

dynamic classification
The assignment of an instance from one classifier to another. Contrast: *multiple classification, static classification*.

element
A constituent of a model.

entry action
An action that a method executes when an object enters a state in a state machine regardless of the transition taken to reach that state.

enumeration
A data type whose instances a list of named values. For example, RGBColor = {red, green, blue}. Boolean is a predefined enumeration with values from the set {false, true}.

event
The specification of a significant occurrence that has a location in time and space and can cause the execution of an associated behavior. In the context of state diagrams, an event is an occurrence that can trigger a transition.

exception
A special kind of signal, typically used to signal fault situations. The sender of the exception aborts execution and execution resumes with the receiver of the exception, which may be the sender itself. The receiver of an exception is determined implicitly by the interaction sequence during execution; it is not explicitly specified.

execution occurrence
A unit of behavior within the lifeline as represented on an interaction diagram.

exit action
An action that a method executes when an object exits a state in a state machine regardless of the transition taken to exit that state.

export
In the context of packages, to make an element visible outside its enclosing namespace. See: *visibility.* Contrast: *export* [OMA], *import*.

expression
A string that evaluates to a value of a particular type. For example, the expression “(7 + 5 * 3)” evaluates to a value of type number.

extend
A relationship from an extension use case to a base use case, specifying how the behavior defined for the extension use case augments (subject to conditions specified in the extension) the behavior defined for the base use case. The behavior is inserted at the location defined by the extension point in the base use case. The base use case does not depend on performing the behavior of the extension use case. See *extension point*, *include*.

extension
An aggregation that is used to indicate that the properties of a metaclass are extended through a stereotype, and that gives the ability to flexibly add and remove stereotypes from classes.

facade
A stereotyped package containing only references to model elements owned by another package. It is used to provide a ‘public view’ of some of the contents of a package.

feature
A property, such as an operation or attribute, that characterizes the instances of a classifier.

final state
A special kind of state signifying that the enclosing
composite state or the entire state machine is completed.

fire
To execute a state transition. See: *transition*.

focus class
A stereotyped class that defines the core logic or control flow for one or more auxiliary classes that support it. Focus classes are typically used together with one or more auxiliary classes, and are particularly useful for specifying the core business logic or control flow of components during design. See also: *auxiliary class*.

focus of control
A symbol on a sequence diagram that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure.

formal parameter
Synonym: *parameter*.

framework
A stereotyped package that contains model elements which specify a reusable architecture for all or part of a system. Frameworks typically include classes, patterns or templates. When frameworks are specialized for an application domain, they are sometimes referred to as application frameworks. See: *pattern*.

generalizable element
A model element that may participate in a generalization relationship. See: *generalization*.

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

guard condition
A condition that must be satisfied in order to enable an associated transition to fire.

implementation
A definition of how something is constructed or computed. For example, a class is an implementation of a type, a method is an implementation of an operation.

implementation class
A stereotyped class that specifies the implementation of a class in some programming language (e.g., C++, Smalltalk, Java) in which an instance may not have more than one class. An Implementation class is said to realize a type if it provides all of the operations defined for the type with the same behavior as specified for the type's operations. See also: *type*.

implementation inheritance
The inheritance of the implementation of a more general element. Includes inheritance of the interface. Contrast: *interface inheritance*.

import
In the context of packages, a dependency that shows the packages whose classes may be referenced within a given package (including packages recursively embedded within it). Contrast: *export*.

include
A relationship from a base use case to an inclusion use case, specifying how the behavior for the base use case contains the behavior of the inclusion use case. The behavior is included at the location which is defined in the base use case. The base use case depends on performing the behavior of the inclusion use case, but not on its structure (i.e., attributes or operations). See *extend*.

inheritance
The mechanism by which more specific elements incorporate structure and behavior of more general elements. See *generalization*.

initial state
A special kind of state signifying the source for a single transition to the default state of the composite state.

instance
An entity that has unique identity, a set of operations that can be applied to it, and state that stores the effects of the operations. See: *object*.

interaction
A specification of how stimuli are sent between instances to perform a specific task. The interaction is defined in the context of a collaboration. See *collaboration.*

interaction diagram
A generic term that applies to several types of diagrams that emphasize object interactions. These include communication diagrams, sequence diagrams, and the interaction overview diagram.

interaction overview diagram
A disgram that depicts interactions through a variant of
activity diagrams in a way that promotes overview of
the control flow. It focuses on the overview of the flow of control where each node can be an interaction diagram.

interface

A named set of operations that characterize the behavior of an element.

interface inheritance
The inheritance of the interface of a more general element. Does not include inheritance of the implementation. Contrast: *implementation inheritance*.

internal transition
A transition signifying a response to an event without changing the state of an object.

layer
The organization of classifiers or packages at the same level of abstraction. A layer may represent a horizontal slice through an architecture, whereas a partition represents a vertical slice. Contrast: *partition*.

lifeline
A modeling element that represents an individual participant in an interaction. A lifeline represents only one interacting entity.

link
A semantic connection among a tuple of objects. An instance of an association. See: *association*.

link end
An instance of an association end. See: *association end*.

message
A specification of the conveyance of information from one instance to another, with the expectation that activity will ensue. A message may specify the raising of a signal or the call of an operation.

metaclass
A class whose instances are classes. Metaclasses are typically used to construct metamodels.

meta-metamodel
A model that defines the language for expressing a metamodel. The relationship between a meta-metamodel and a metamodel is analogous to the relationship between a metamodel and a model.

metamodel
A model that defines the language for expressing a model.

metaobject
A generic term for all metaentities in a metamodeling language. For example, metatypes, metaclasses, metaattributes, and metaassociations.

method
The implementation of an operation. It specifies the algorithm or procedure associated with an operation.

model aspect
A dimension of modeling that emphasizes particular qualities of the metamodel. For example, the structural model aspect emphasizes the structural qualities of the metamodel.

model elaboration
The process of generating a repository type from a published model. Includes the generation of interfaces and implementations which allows repositories to be instantiated and populated based on, and in compliance with, the model elaborated.

model element
An element that is an abstraction drawn from the system being modeled. Contrast: *view element*.

model library
A stereotyped package that contains model elements that are intended to be reused by other packages. A model library differs from a profile in that a model library does not extend the metamodel using stereotypes and tagged

definitions. A model library is analogous to a class library in some programming languages.

modeling time
Refers to something that occurs during a modeling phase of the system development process. It includes analysis time and design time. Usage note: When discussing object systems, it is often important to distinguish between modeling-time and run-time concerns. See: *analysis time, design time*. Contrast: *run time*.

multiple classification
The assignment of an instance directly to more than one classifier at the same time. See: *static classification*, *dynamic classification*.

multiple inheritance
A semantic variation of generalization in which a type may have more than one supertype. Contrast: *single inheritance*.

multiplicity
A specification of the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given for association ends, parts within composites, repetitions, and other purposes. Essentially a multiplicity is a (possibly infinite) subset of the non-negative integers. Contrast: *cardinality*.

n-ary association
An association among three or more classes. Each instance of the association is an n-tuple of values from the respective classes. Contrast: *binary association*.

name
A string used to identify a model element.

namespace
A part of the model in which the names may be defined and used. Within a namespace, each name has a unique meaning. See: *name*.

node
A classifier that represents a run-time computational resource, which generally has at least memory and often processing capability. Run-time objects and components may reside on nodes.

note
An annotation attached to an element or a collection of elements. A note has no semantics. Contrast: *constraint*.

object
An instance of a class. See: *class, instance*.

object diagram
A diagram that encompasses objects and their relationships at a point in time. An object diagram may be considered a special case of a class diagram or a communication diagram. See: *class diagram, communication diagram.*

object flow state
A state in an activity diagram that represents the passing of an object from the output of actions in one state to the input of actions in another state.

object lifeline
A line in a sequence diagram that represents the existence of an object over a period of time. See: *sequence diagram*.

operation
A feature which declares a service that can be performed by instances of the classifier of which they are instances.

package

A general purpose mechanism for organizing elements into groups. Packages may be nested within other packages.

package diagram
A diagram that depicts how model elements are organized into packages and the dependencies among them, including package imports and package extensions.

parameter
An argument of a behavioral feature. 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. Synonyms: *formal parameter*. Contrast: *argument*.

parameterized element
The descriptor for a class with one or more unbound parameters. Synonym: *template*.

parent
In a generalization relationship, the generalization of another element, the child. See: *subclass*, *subtype*. Contrast: *child*.

part
An element representing a set of instances that are owned by a containing classifier instance or role of a classifier. (See *role.)* Parts may be joined by attached connectors and specify configurations of linked instances to be created within an instance of the containing classifier.

participate
The connection of a model element to a relationship or to a reified relationship. For example, a class participates in an association, an actor participates in a use case.

partition
A grouping of any set of model elements based on a set of criteria.
1. activity diagram: A grouping of activity nodes and edges. Partitions divide the nodes and edges to constrain and show a view of the contained nodes. Partitions can share contents. They often correspond to organizational units in a business model. They may be used to allocate characteristics or resources among the nodes of an activity.
2. architecture: A set of related classifiers or packages at the same level of abstraction or across layers in a layered architecture. A partition represents a vertical slice through an architecture, whereas a layer represents a horizontal slice. Contrast: *layer*.

pattern
A template collaboration that describes the structure of
a design pattern. UML patterns are more limited than those used by the design pattern community. In general, design patterns involve many non-structural aspects, such as heuristics for their use and usage trade-offs.

persistent object
An object that exists after the process or thread that created it has ceased to exist.

pin
A model element that represents the data values passed into a behavior upon its invocation as well as the data values returned from a behavior upon completion of its execution.

port
A feature of a classifier that specifies a distinct interaction point between that classifier and its environment or between the (behavior of the) classifier and its internal parts. Ports are connected to other ports through connectors through which requests can be made to invoke the behavioral features of a classifier.

postcondition
A constraint expresses a condition that must be true at the completion of an operation.

powertype
A classifier whose instances are also subclasses of another classifier. Power types, then, are metaclasses with an extra twist: the instances are also subclasses.

precondition
A constraint expresses a condition that must be true when an operation is invoked.

primitive type
A pre-defined data type without any relevant substructure (i.e., is not decomposable) such as an integer or a string. It may have an algebra and operations defined outside of UML, for example, mathematically.

procedure
A set of actions that may be attached as a unit to other parts of a model, for example, as the body of a method. Conceptually a procedure, when executed, takes a set of values as arguments and produces a set of values as results, as specified by the parameters of the procedure.

process
1. A heavyweight unit of concurrency and execution in an operating system. Contrast: *thread*, which includes heavyweight and lightweight processes. If necessary, an implementation distinction can be made using stereotypes.
2. A software development process—the steps and guidelines by which to develop a system.
3. To execute an algorithm or otherwise handle something dynamically.

profile
A stereotyped package that contains model elements that have been customized for a specific domain or purpose using extension mechanisms, such as stereotypes, tagged definitions and constraints. A profile may also specify model libraries on which it depends and the metamodel subset that it extends.

projection
A mapping from a set to a subset of it.

property
A named value denoting a characteristic of an element. A property has semantic impact. Certain properties are predefined in the UML; others may be user defined. See: *tagged value*.

pseudo-state
A vertex in a state machine that has the form of a state, but doesn't behave as a state. Pseudo-states include initial and history vertices.

physical system
1. The subject of a model.
2. A collection of connected physical units, which can include software, hardware and people, that are organized to accomplish a specific purpose. A physical system can be described by one or more models, possibly from different viewpoints. Contrast: *system*.

qualifier
An association attribute or tuple of attributes whose values partition the set of objects related to an object across an association.

realization
A specialized abstraction relationship between two sets of model elements, one representing a specification (the supplier) and the other representing an implementation of the latter (the client). Realization can be used to model stepwise refinement, optimizations, transformations, templates, model synthesis, framework composition, etc.

receive [a message]
The handling of a stimulus passed from a sender instance. See: *sender, receiver*.

receiver
The object handling a stimulus passed from a sender object. Contrast: *sender*.

reception
A declaration that a classifier is prepared to react to the receipt of a signal.

reference
1. A denotation of a model element.
2. A named slot within a classifier that facilitates navigation to other classifiers. Synonym: *pointer*.

refinement
A relationship that represents a fuller specification of something that has already been specified at a certain level of detail. For example, a design class is a refinement of an analysis class.

relationship
An abstract concept that specifies some kind of connection between elements. Examples of relationships include associations and generalizations.

repository
A facility for storing object models, interfaces, and implementations.

requirement
A desired feature, property, or behavior of a system.

responsibility
A contract or obligation of a classifier.

reuse
The use of a pre-existing artifact.

role
The named set of features defined over a collection of entities participating in a particlar context.
Collaboration: The named set of behaviors possessed by a class or part participating in a particular context.
Part: a subset of a particular class which exhibits a subset of features possessed by the class
Associations: A synonym for association end often referring to a subset of classifier instances that are participating in the association.

run time
The period of time during which a computer program or a systemexecutes. Contrast: *modeling time*.

scenario
A specific sequence of actions that illustrates behaviors. A scenario may be used to illustrate an interaction or the execution of a use case instance. See: *interaction.*

semantic variation point
A point of variation in the semantics of a metamodel. It provides an intentional degree of freedom for the interpretation of the metamodel semantics.

send [a message]
The passing of a stimulus from a sender instance to a receiver instance. See: *sender, receiver*.

sender
The object passing a stimulus to a receiver instance. Contrast: *receiver*.

sequence diagram
A diagram that depicts an interaction by focusing on the sequence of messages that are exchanged, along with their corresponding event occurrences on the lifelines.
Unlike a communication diagram, a sequence diagram includes time sequences but does not include object

relationships. A sequence diagram can exist in a generic form (describes all possible scenarios) and in an instance form (describes one actual scenario). Sequence diagrams and communication diagrams express similar information, but show it in different ways. See: *communication diagram*.

signal
The specification of an asynchronous stimulus that triggers a reaction in the receiver in an asynchronous way and without a reply. The receiving object handles the signal as specified by its receptions. The data carried by a send request and passed to it by the occurrence of the send invocation event that caused the request is represented as attributes of the signal instance. A signal is defined independently of the classifiers handling the signal.

signature
The name and parameters of a behavioral feature. A signature may include an optional returned parameter.

single inheritance
A semantic variation of generalization in which a type may have only one supertype. Synonym: *multiple inheritance* [OMA]. Contrast: *multiple inheritance*.

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

software module
A unit of software storage and manipulation. Software modules include source code modules, binary code modules, and executable code modules.

specification
A set of requirements for a system or other classifier. Contrast: *implementation*.

state
A condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. Contrast: *state* [OMA].

state machine diagram
A diagram that depicts discrete behavior modeled through finite state-transition systems. In particular, it specifies the sequences of states that an object or an interaction goes through during its life in response to events, together with its responses and actions. See: *state machine*.

state machine
A behavior that specifies the sequences of states that an object or an interaction goes through during its life in response to events, together with its responses and actions.

static classification
The assignment of an instance to a classifier where the assignment may not change to any other classifier. Contrast: *dynamic classification.*

stereotype
A class that defines how an existing metaclass (or stereotype) may be extended, and enables the use of platform or domain specific terminology or notation in addition to the ones used for the extended metaclass.
Certain stereotypes are predefined in the UML, others may be user defined. Stereotypes are one of the extensibility mechanisms in UML. See: *constraint, tagged value.*

stimulus
The passing of information from one instance to another, such as raising a signal or invoking an operation. The receipt of a signal is normally considered an event. See: *message*.

string
A sequence of text characters. The details of string representation depend on implementation, and may include

character sets that support international characters and graphics.

structural feature
A static feature of a model element, such as an attribute.

structural model aspect
A model aspect that emphasizes the structure of the objects in a system, including their types, classes, relationships, attributes, and operations.

structure diagram
A form of diagram that depicts the elements in a specification that are irrespective of time. Class diagrams and component diagrams are examples of structure diagrams.

subactivity state
A state in an activity diagram that represents the execution of a non-atomic sequence of steps that has some duration.

subclass
In a generalization relationship, the specialization of another class, the superclass. See: *generalization*. Contrast: *superclass*.

submachine state
A state in a state machine that is equivalent to a
composite state but whose contents are described by another state machine.

substate
A state that is part of a composite state. See: *concurrent state, disjoint state*.

subpackage
A package that is contained in another package.

subsystem
A unit of hierarchical decomposition for large systems. A subsystem is commonly instantiated indirectly. Definitions of subsystems vary widely among domains and methods, and it is expected that domain and method profiles will specialize this construct. A subsystem may be defined to have specification and realization elements.

subtype
In a generalization relationship, the specialization of another type, the supertype. See: *generalization*. Contrast: *supertype*.

superclass
In a generalization relationship, the generalization of another class, the subclass. See: *generalization.* Contrast: *subclass*.

supertype
In a generalization relationship, the generalization of another type, the subtype. See: *generalization*. Contrast: *subtype*.

supplier
A classifier that provides services that can be invoked by others. Contrast: *client*.

synch state
A vertex in a state machine used for synchronizing the
concurrent regions of a state machine.

system
An organized array of elements functioning as a unit
Also, a top-level subsystem in a model.

tagged value
The explicit definition of a property as a name-value pair. In a tagged value, the name is referred as the tag. Certain tags are predefined in the UML; others may be user defined. Tagged values are one of three extensibility mechanisms in UML. See: *constraint, stereotype.*

template
Synonym: *parameterized element*.

thread [of control]
A single path of execution through a program, a dynamic model, or some other representation of control flow. Also, a stereotype for the implementation of an active object as lightweight process. See *process*.

time event
An event that denotes the time elapsed since the current state was entered. See: *event*.

time expression
An expression that resolves to an absolute or relative value of time.

timing diagram
An interaction diagram that shows the change in state or condition of a lifeline (representing a Classifier Instance or Classifier Role) over linear time. The most common usage is to show the change in state of an object over time in response to accepted events or stimuli.

top level
A stereotype denoting the top-most package in a containment hierarchy. The topLevel stereotype defines the outer limit for looking up names, as namespaces “see” outwards. For example, opLevel subsystem represents the top of the subsystem containment hierarchy.

trace
A dependency that indicates a historical or process relationship between two elements that represent the same concept without specific rules for deriving one from the other.

transient object
An object that exists only during the execution of the process or thread that created it.

transition
A relationship between two states indicating that an object in the first state will perform certain specified actions and enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state, the transition is said to fire.

type
A stereotyped class that specifies a domain of objects together with the operations applicable to the objects, without defining the physical implementation of those objects. A type may not contain any methods, maintain its own thread of control, or be nested. However, it may have attributes and associations. Although an object may have at most one implementation class, it may conform to multiple different types. See also: *implementation class* Contrast: *interface*.

type expression
An expression that evaluates to a reference to one or more types.

uninterpreted
A placeholder for a type or types whose implementation is not specified by the UML. Every uninterpreted value has a corresponding string representation. See: *any* [CORBA].

usage
A dependency in which one element (the client) requires the presence of another element (the supplier) for its correct functioning or implementation.

use case
The specification of a sequence of actions, including variants, that a system (or other entity) can perform, interacting with actors of the system. See: *use case instances.*

use case diagram
A diagram that shows the relationships among actors and the subject (system), and use cases.

use case instance
The performance of a sequence of actions being specified in a use case. An instance of a use case. See: *use case class*.

use case model
A model that describes a system's functional requirements in terms of use cases.

utility
A stereotype that groups global variables and procedures in the form of a class declaration. The utility attributes and operations become global variables and global procedures, respectively. A utility is not a fundamental modeling construct, but a programming convenience.

value
An element of a type domain.

vertex
A source or a target for a transition in a state machine. A vertex can be either a state or a pseudo-state. See: *state, pseudo-state.*

view
A projection of a model that is seen from a given perspective or vantage point and omits entities that are not relevant to this perspective.

view element
A textual and/or graphical projection of a collection of model elements.

view projection
A projection of model elements onto view elements. A view projection provides a location and a style for each view element.

visibility
An enumeration whose value (public, protected, or private) denotes how the model element to which it refers may be seen outside its enclosing namespace.

5 Symbols

Editorial Comment: The FTF needs to complete this section (or possibly eliminate it)

6 Additional information

6.1 Changes to Adopted OMG Specifications

The specification, in conjunction with the specification that complements this, the *UML 2.0: Superstructure*, completely replaces the current versions of UML 1.4.1 and UML 1.5 with Action Semantics, except for the “Model Interchange Using CORBA IDL” (see Chapter 5, Section 5.3 of the OMG UML Specification v1.4, OMG document ad/01-02-17). “Model Interchange Using CORBA IDL” is retired as an adopted technology because of lack of vendor and user interest.

6.2 Architectural Alignment and MDA Support

Chapter 7, “Language Architecture”, explains how the *UML 2.0: Infrastructure* is architecturally aligned with the *UML 2.0: Superstructure* that complements it. It also explains how the InfrastructureLibrary defined in the *UML 2.0: Infrastructure* can be strictly reused by MOF 2.0 specifications.

The *MOF 2.0: 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 Appendix B.

6.3 How to Read this Specification

The rest of this document contains the technical content of this specification. Readers are encouraged to first read Part “Part 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 Part “Part II - Infrastructure Library”, or the UML::Classes::Kernel package which reuses it, described in the *UML 2.0: 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 2.0: Superstructure*.

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

6.4 Acknowledgments

The following companies submitted and/or supported parts of this specification:

- Adaptive
- Boldsoft
- Borland Software Corporation
- Compuware
- Dresden University of Technology
- International Business Machines Corp.
- IONA

- Kabira Technologies, Inc.
- Kings College
- Klasse Objecten
- Oracle
- Project Technology, Inc.
- Rational Software Corporation
- Softeam
- Syntropy Ltd.
- Telelogic
- University of Bremen
- University of Kent
- University of York

The following persons were members of the core team that designed and wrote this specification: Don Baisley, Morgan Björkander, Conrad Bock, Steve Cook, Philippe Desfray, Nathan Dykman, Anders Ek, David Frankel, Eran Gery, Øystein Haugen, Sridhar Iyengar, Cris Kobryn, Birger Møller-Pedersen, James Odell, Gunnar Övergaard, Karin Palmkvist, Guus Ramackers, Jim Rumbaugh, Bran Selic, Thomas Weigert and Larry Williams.

In addition, the following persons contributed valuable ideas and feedback that significantly improved the content and the quality of this specification: Colin Atkinson, Ken Baclawski, Mariano Belaunde, Steve Brodsky, Roger Burkhart, Bruce Douglass, Sandy Friedenthal, Sébastien Gerard, Dwayne Hardy, Mario Jeckle, Larry Johnson, Allan Kennedy, Stuart Kent, Mitch Kokar, Thomas Kuehne, Michael Latta, Dave Mellor, Jeff Mischkinksky, Hiroshi Miyazaki, Jishnu Mukerji, Ileana Ober, Barbara Price, Tom Rutt, Oliver Sims, Kendall Scott, Cameron Skinner, Jeff Smith, Doug Tolbert, and Ian Wilkie.

Part 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 has issued four complementary and architecturally aligned RFPs to define UML 2.0: UML 2.0 Infrastructure, UML 2.0 Superstructure, UML 2.0 Object Constraint Language and UML 2.0 Diagram Interchange.

This UML 2.0 specification is organized into two volumes (*UML 2.0: Infrastructure* and *UML 2.0: Superstructure*), consistent with the breakdown of modeling language requirements into two RFPs (*UML 2.0 Infrastructure RFP* and *UML 2.0 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 chapters describe the language architecture and the specification approach used to define UML 2.0.

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 implementors and practitioners.[1] This chapter explains the architecture of the UML metamodel.

The following sections summarize the design principles followed, and show how they are applied to organize UML's Infrastructure and Superstructure. The last section 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 loosing the coupling across packages.
- Extensibility. The UML can be extended in two ways: 1) 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); 2) 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 Model (CWM).

7.2 Infrastructure Architecture

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

- 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.

---

1. It is important to note that the specification of UML as a metamodel does note preclude it from being specified via a mathematically formal language (e.g., Object-Z or VDM) at a later time.

As is shown in Figure 1, Infrastructure is represented by the package *InfrastructureLibrary*, which 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.

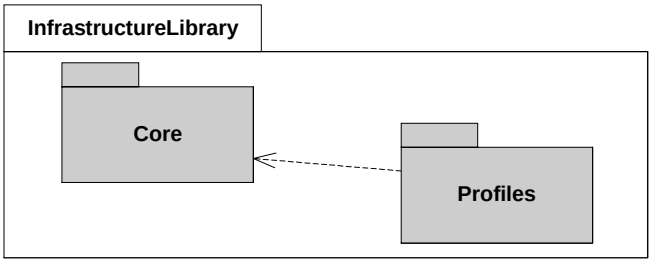

Figure 1 - The InfrastructureLibrary packages

7.2.1 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 “Superstructure Architecture” on page 26) either import or specialize its specified metaclasses. This is illustrated in Figure 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.

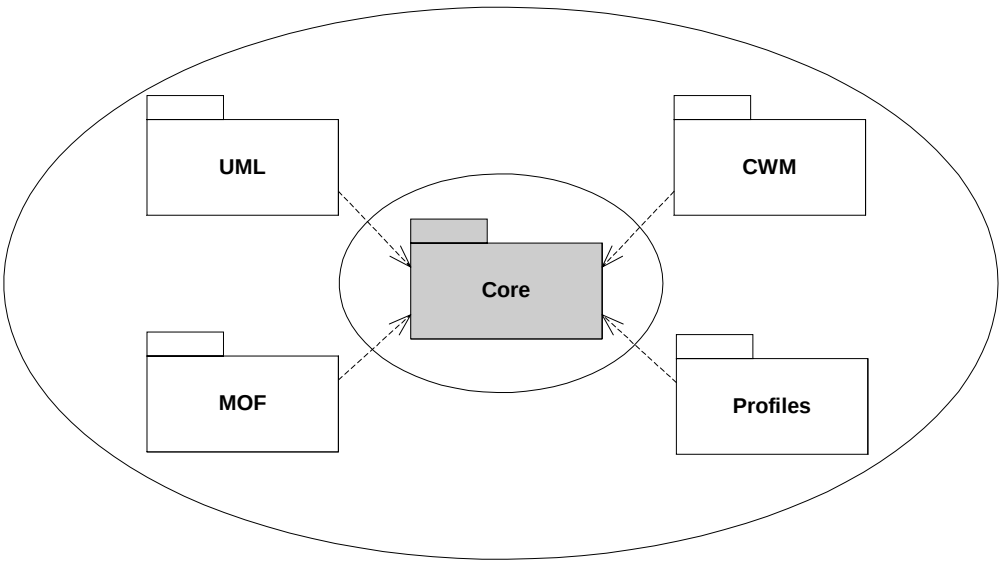

Figure 2 - The role of the common Core

In order to facilitate reuse, the *Core* package is subdivided into a number of packages: *PrimitiveTypes*, *Abstractions*, *Basic*, and *Constructs*, as shown in Figure 3. As we will see in subsequent chapters, 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. The package *PrimitiveTypes* simply contains a few predefined types that are commonly used when metamodeling, and is designed specifically with the needs of UML and MOF in mind. Other metamodels may need other or overlapping sets of primitive types. There are minor differences in the design rationale for the other two 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.*

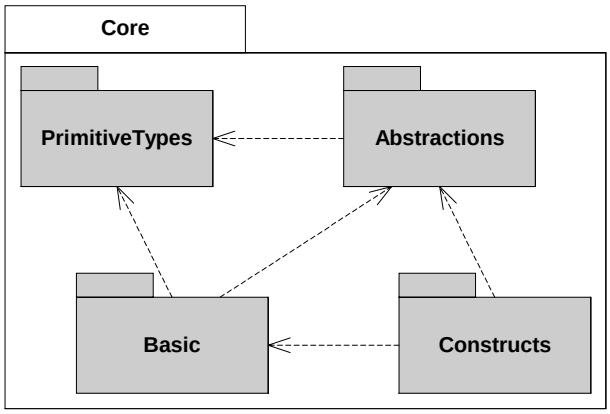

Figure 3 - The Core packages

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 “Metamodel layering” on page 28). 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*.

7.2.2 Profiles

As was depicted in Figure 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.

7.2.3 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 4. Note that MOF is used as the metamodel for not only UML, but also for other languages such as CWM.

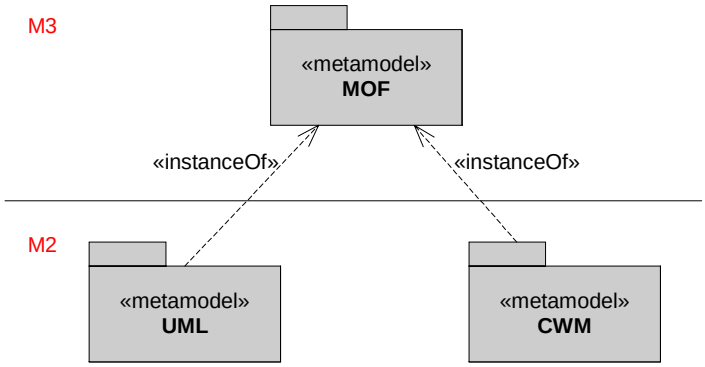

Figure 4 - UML and MOF are at different metalevels

How these metalevel hierarchies work is explained in more detail in “Superstructure Architecture” on page 26. 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 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 MOF itself, but also for CWM, UML, and 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 Chapter 13, “Core::Profiles”). In fact, profiles are defined to be a subset of the MOF extension mechanism.

7.2.4 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 5.

Each of these areas is described in a separate chapter of the *UML 2.0: 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.

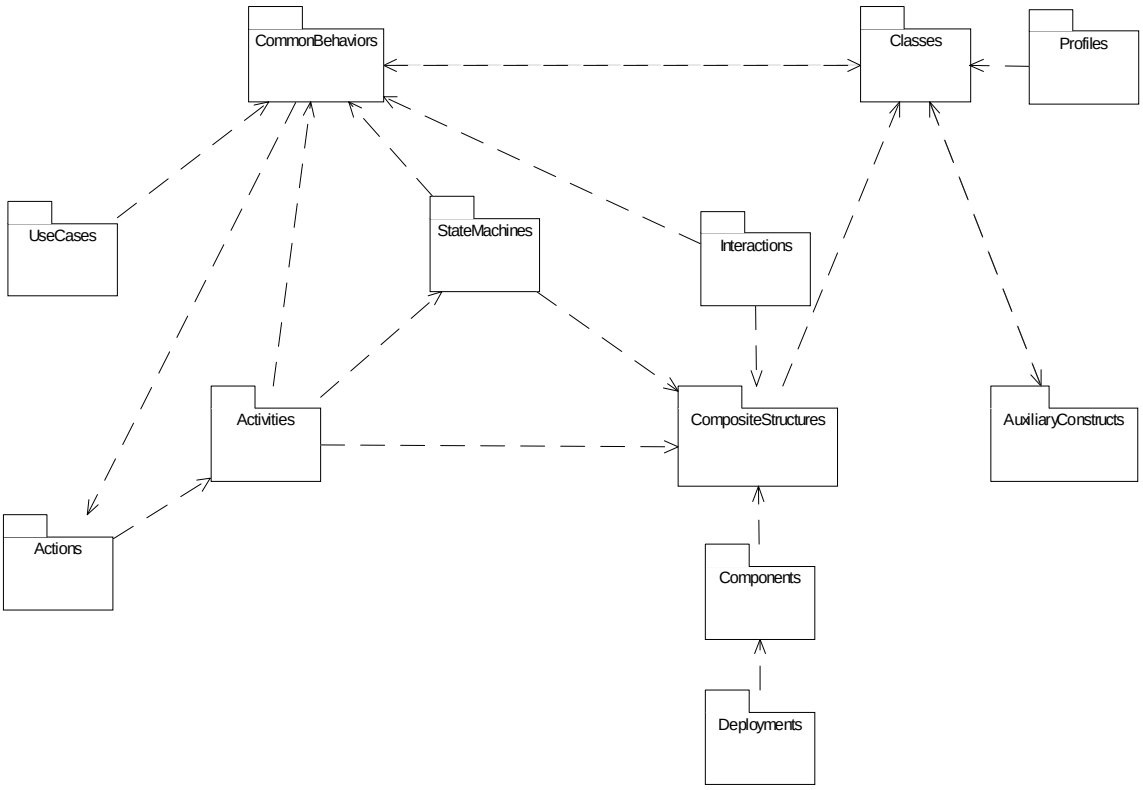

Figure 5 - The top-level package structure of the UML 2.0 Superstructure

7.2.5 Reusing Infrastructure

One of the primary uses of the UML 2.0 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.

7.2.6 The Kernel Package

The *InfrastructureLibrary* is primarily reused in the *Kernel* package of *Classes* in *UML 2.0: 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 occurs 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).

7.2.7 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 has to be taken into account:

- the language specification, or the metamodel,
- the user specification, or the model, and
- 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.

7.2.8 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. When a language is reflective, there is no need to define another language to specify its semantics. MOF is reflective since it is based on the *InfrastructureLibrary*, and there is thus no need to have additional meta-layers above MOF.

7.2.9 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 gets instantiated. As an example, consider Figure 8, 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.

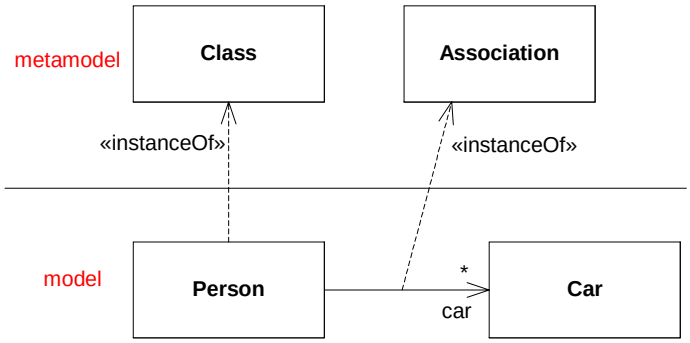

Figure 6 - An example of metamodeling; note that not all instance-of relationships are shown

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, where the instance specification Mike is an illustration (or a snapshot) of an instance of class Person.

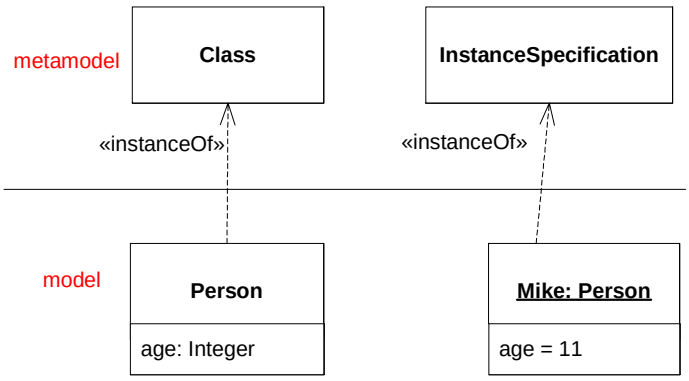

Figure 7 - Giving an illustration of a class using an instance specification

7.2.10 An example of the four-level metamodel hierarchy

An illustration of how these meta-layers relate to each other is shown in Figure 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.

.

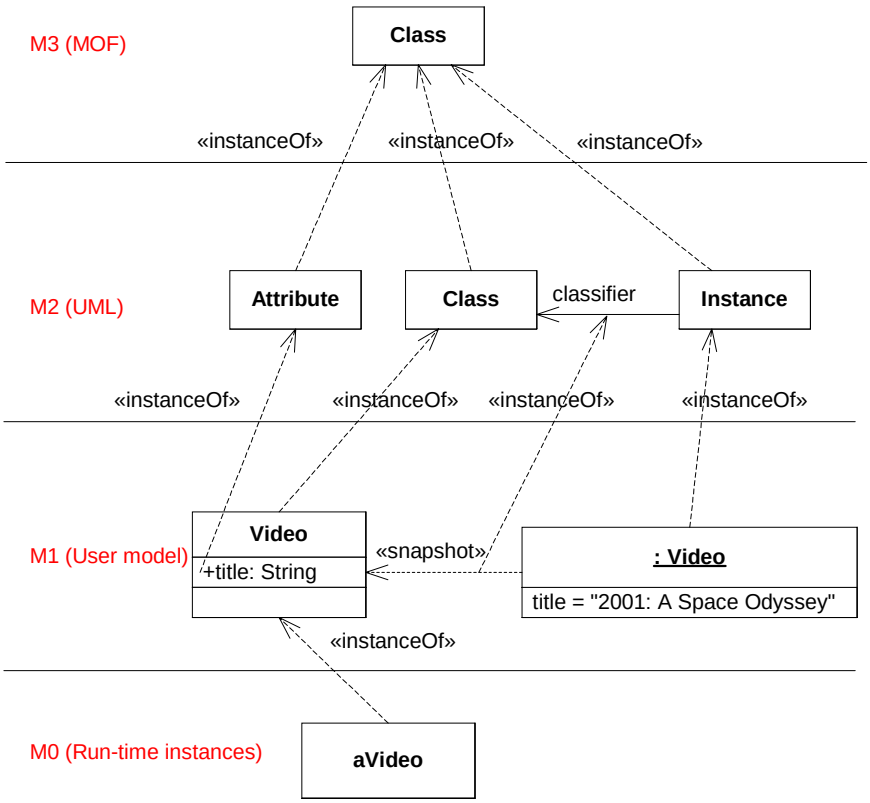

Figure 8 - An example of the four-layer metamodel hierarchy

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 chapter explains the specification techniques used to define UML.

The following are the goals of the specifications 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 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 AssociationEnd, which is used to indicate if an association is an ordinary aggregate, a composite aggregate, or a common association.

Package Specification Structure

This section provides information for each package and each class in the UML metamodel. Each package has one or more of the following subsections:

Class Descriptions

The section 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).

8.1.1 Diagrams

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

8.1.2 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 instance of the classes contained in the package (or in an imported package).

8.2 Class Specification Structure

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

8.2.1 Description

The section includes an informal definition of the metaclass specifying the construct in UML. The section states if the metaclass is abstract.

This section together with the following two constitute a description of the abstract syntax of the construct.

---

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.

Attributes

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

8.2.2 Associations

The opposite ends of associations connected to the class are also listed in the same way. The section states if the association is derived, or if it is a specialization of another association. The multiplicity of an association end is suppressed if it is ‘*’ (default in UML).

When directed associations are specified in lieu of attributes, the multiplicity on the undirected end is assumed to be ‘*’ (default in UML) and the role name should not be used.

8.2.3 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 section, 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 invariant is defined by an OCL expression together with an informal explanation of the expression, but in some cases the invariant is 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.

8.2.4 Additional Operations (optional)

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

8.2.5 Semantics

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

8.2.6 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.

8.2.7 Notation

The notation of the construct is presented in this section.

8.2.8 Presentation Options (optional)

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

8.2.9 Style Guidelines (optional)

Often is an informal convention how to show (a part of) a construct, like the name of a class should be centered and in bold. These conventions are presented in this section.

8.2.10 Examples (optional)

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

8.2.11 Rationale (optional)

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

8.2.12 Changes from UML 1.4

Here, changes compared with UML 1.4 are described and a migration approach from 1.4 to 2.0 is specified.

8.3 Use of a Constraint Language

The specification uses the Object Constraint Language (OCL), as defined in Chapter 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 section where the constraint appears.

8.4 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).

8.5 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.

Part II - Infrastructure Library

This part describes the structure and contents of the Infrastructure Library for the UML metamodel and related metamodels, such as the Meta Object Facility (MOF) and the Common Warehouse Model (CWM). The *InfrastructureLibrary* package 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 top-level packages of the InfrastructureLibrary are shown in Figure 9.

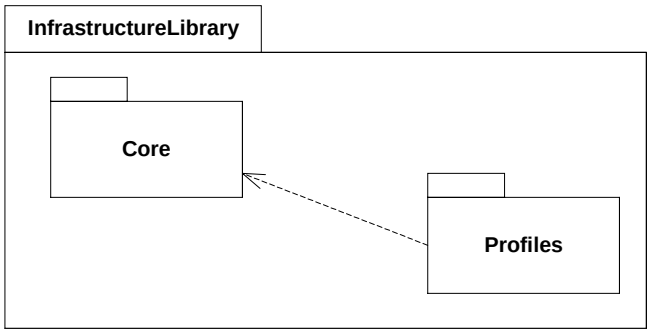

Figure 9 - The Metamodel Library package contains the packages Core and Profiles

The *Core* package is the central reusable part of the InfrastructureLibrary, and is further subdivided as shown in Figure 10.

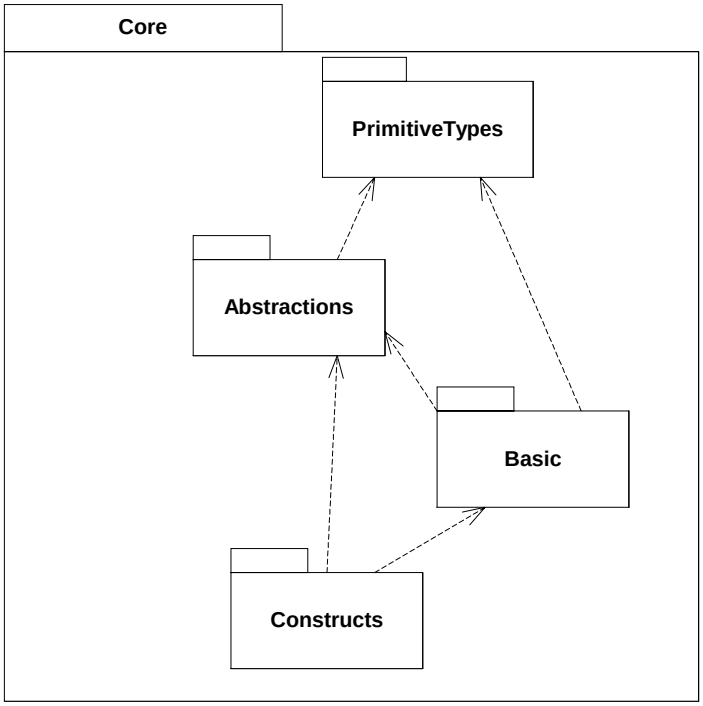

Figure 10 - The Core package contains the packages PrimitiveTypes, Abstractions, Basic, and Constructs

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 chapters.

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.

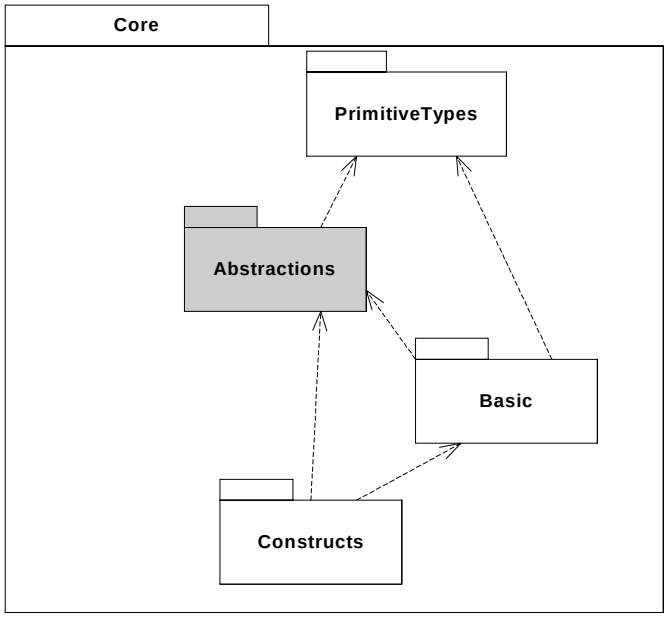

Figure 11 - The Core package is owned by the InfrastructureLibrary pack and contains several subpackages

The subpackages of Abstractions are all shown in Figure 12.

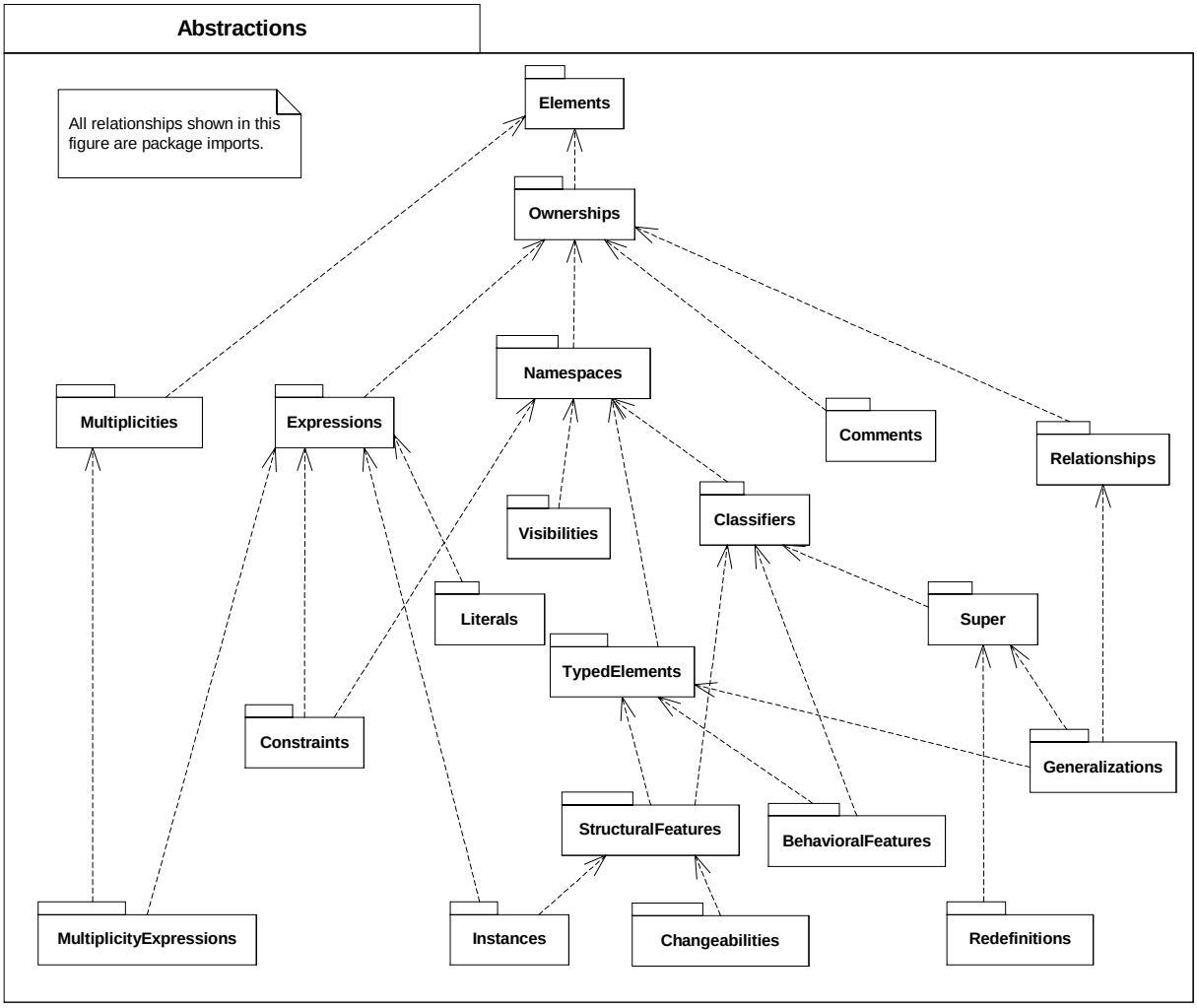

Figure 12 - The Abstractions package contains several subpackages, all of which are specified in this chapter

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

9.1 BehavioralFeatures package

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

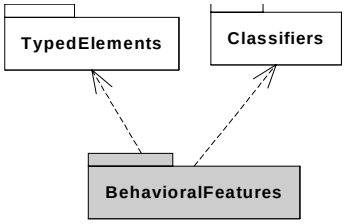

Figure 13 - The BehavioralFeatures package

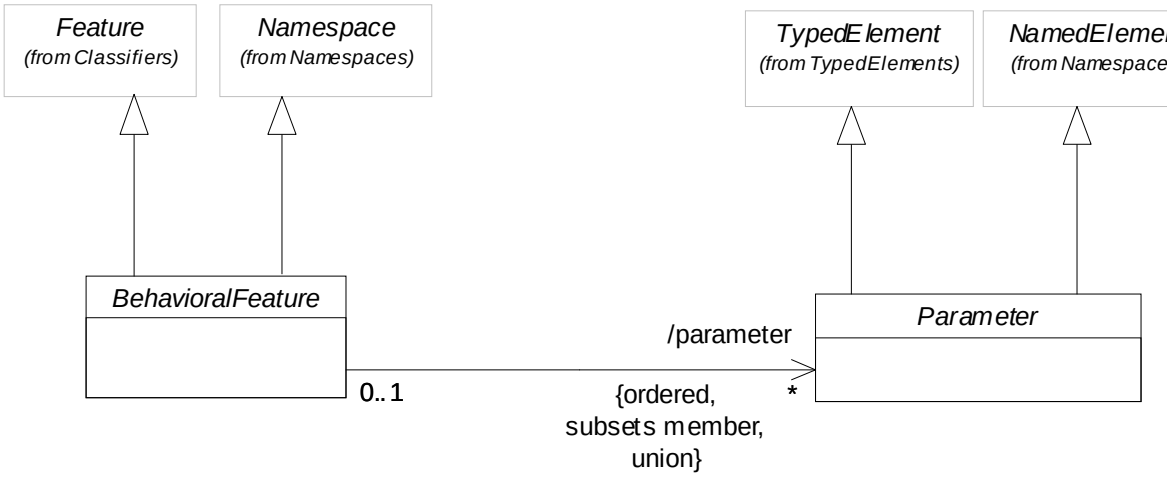

Figure 14 - The elements defined in the BehavioralFeatures package

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.

Attributes

No additional attributes.

Associations

- / parameter: Parameter[*] Specifies the parameters of the BehavioralFeature. Subsets *Namespace::member*. This is a derived union and is ordered.

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.

9.1.2 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*.

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.

9.2 Changeabilities package

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

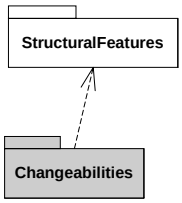

Figure 15 - The Changeabilities package

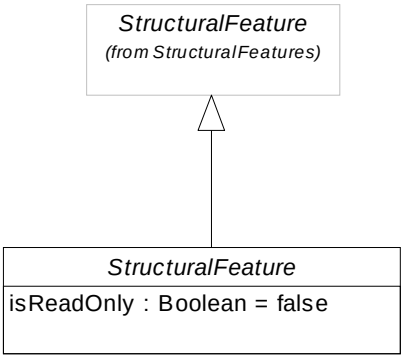

Figure 16 - The elements defined in the Changeabilities package

9.2.1 ChangeabilityKind

ChangeabilityKind is an enumeration type.

Description

ChangeabilityKind is an enumeration of the following literal values:

- unrestricted: Indicates that there is no restriction no adding new values, changing a value, or removing values to an occurrence of a StructuralFeature.

- readOnly: Indicates that adding new values, changing values, and removing values or an occurrence of a StructuralFeature is not permitted.
- addOnly: Indicates that there is no restriction on adding new values to an occurrence of a StructuralFeature, but changing and removing values are restricted.
- removeOnly: Indicates that there is no restriction on removing values from an occurrence of a StructuralFeature, but adding new values and changing values is not permitted.

9.2.2 StructuralFeature (as specialized)

Description

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

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. A modifiable structural feature is shown using {unrestricted} 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.

9.3 Classifiers package

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

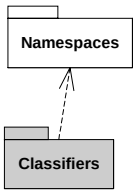

Figure 17 - The Classifiers package

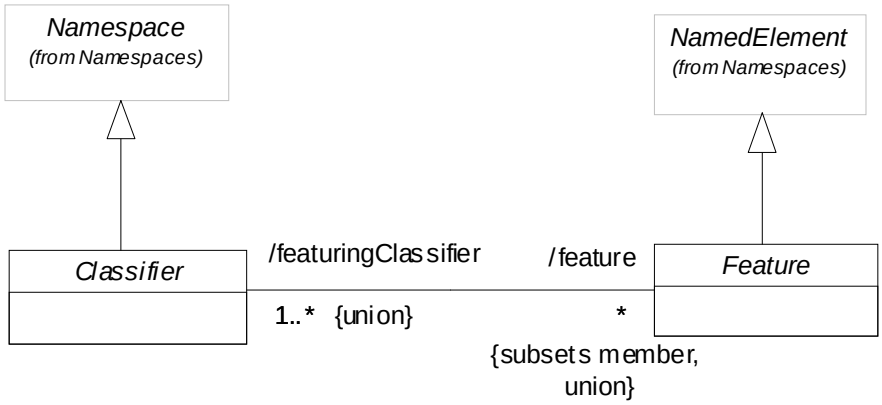

Figure 18 - The elements defined in the Classifiers package

9.3.1 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.

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.

9.3.2 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.

Attributes

No additional attributes.

Associations

- / featuringClassifier: Classifier [1..*]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.

9.4 Comments package

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

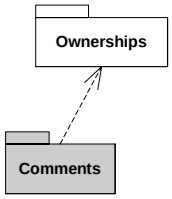

Figure 19 - The Comments package

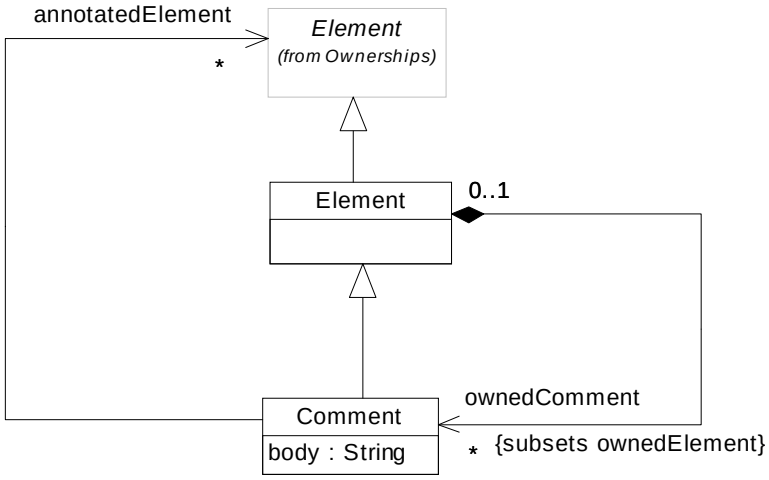

Figure 20 - The elements defined in the Comments package

9.4.1 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.

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

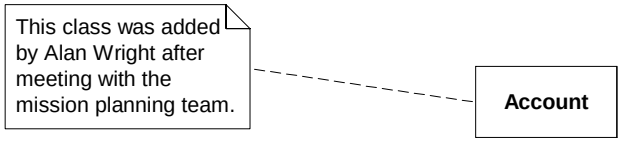

Figure 21 - Comment notation

9.4.2 Element (as specialized)

Description

An element can own comments.

Attributes

No additional attributes.

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.

9.5 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.

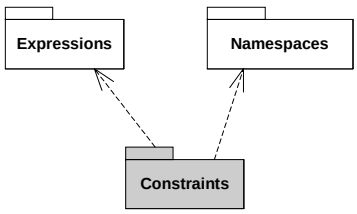

Figure 22 - The Constraints package

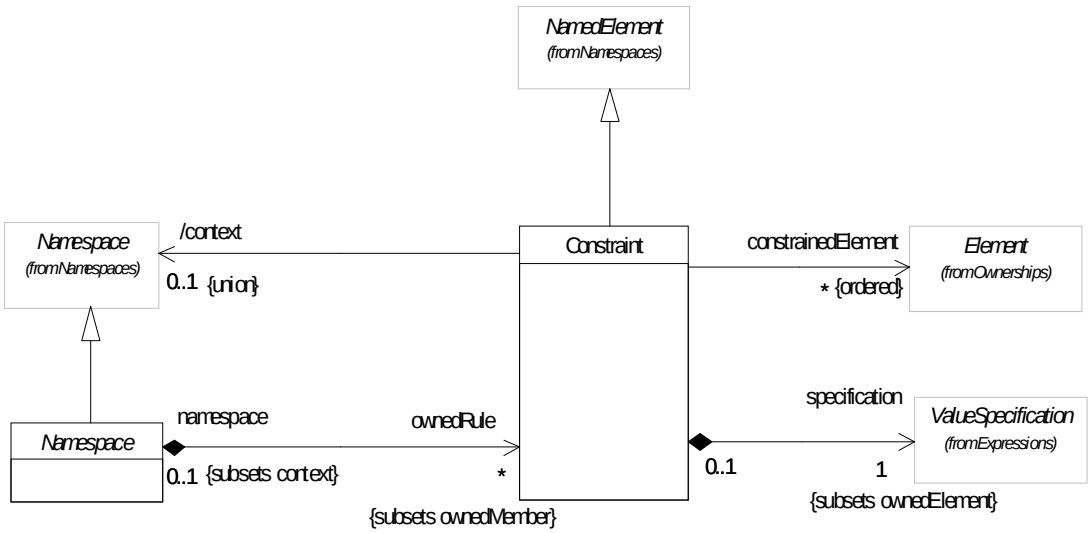

Figure 23 - The elements defined in the Constraints package

9.5.1 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 (such as an association “xor” constraint) 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.

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. This is a derived union.
- specification: ValueSpecification[0..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.

self.specification.isOclKindOf(Boolean)

[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 24 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 25 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 26 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

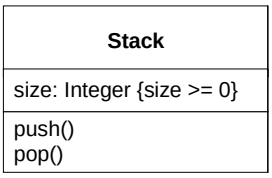

Figure 24 - Constraint attached to an attribute.

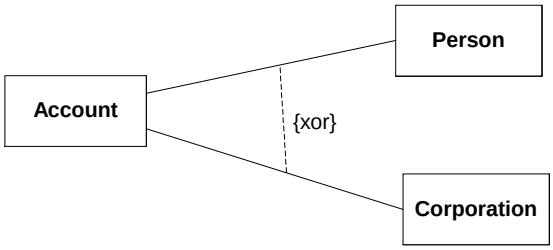

Figure 25 - {xor} constraint

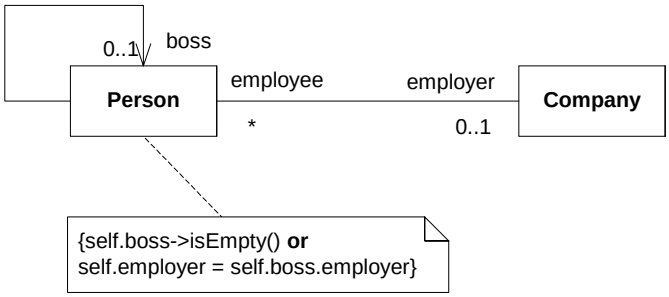

Figure 26 - Constraint in a note symbol

9.5.2 Namespace (as specialized)

Description

A namespace can own constraints. The constraint does not necessarily apply to the namespace itself, but may also apply to elements in the namespace.

Attributes

No additional attributes.

Associations

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

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.

9.6 Elements package

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

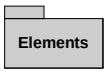

Figure 27 - The Elements package

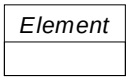

Figure 28 - The elements defined in the Elements package

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.

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.

9.7 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.

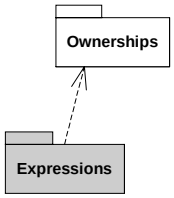

Figure 29 - The Expressions package

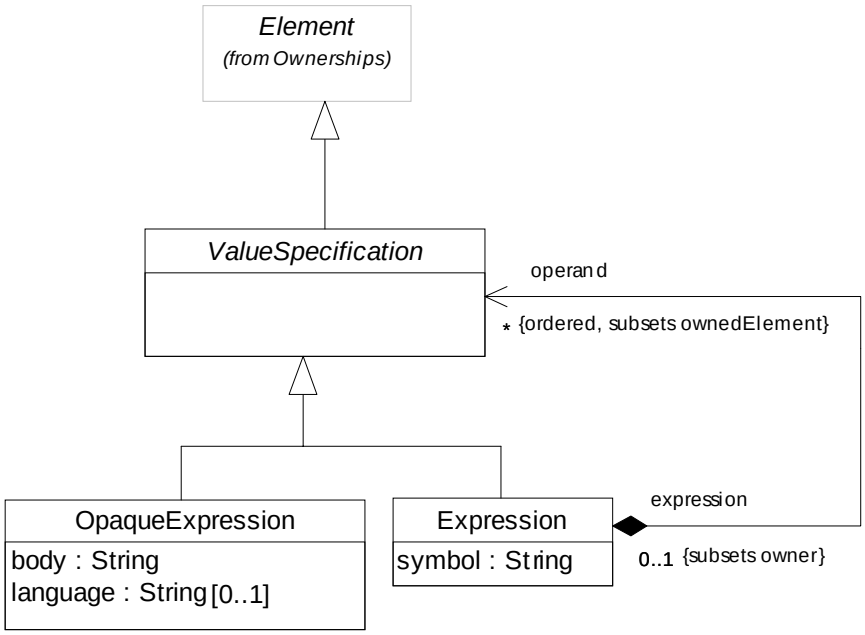

Figure 30 - The elements defined in the Expressions package

9.7.1 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 which are value specifications.

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 are no operands 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

9.7.2 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 a language-specific text string used to describe a value or values, and an optional specification of the language.

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

Attributes

- body: String [1] The text of the expression.
- language: String [0..1] Specifies the language in which the expression is stated. The interpretation of the expression body depends on the language. If language is unspecified, it might be implicit from the expression body or the context.

Associations

No additional associations.

Constraints

No additional constraints.

Semantics

The interpretation of the expression body depends on the language. If the language is unspecified, it might be implicit from the expression body or the context.

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

Notation

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

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

The language of an opaque expression, if specified, is 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.

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

9.7.3 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.

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 booleanValue() gives a single Boolean value when one can be computed.

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

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

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

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

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

[6] 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.

9.8 Generalizations package

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

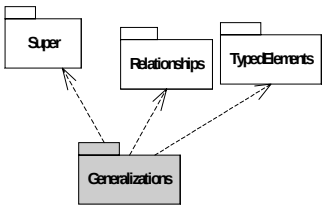

Figure 31 - The Generalizations package

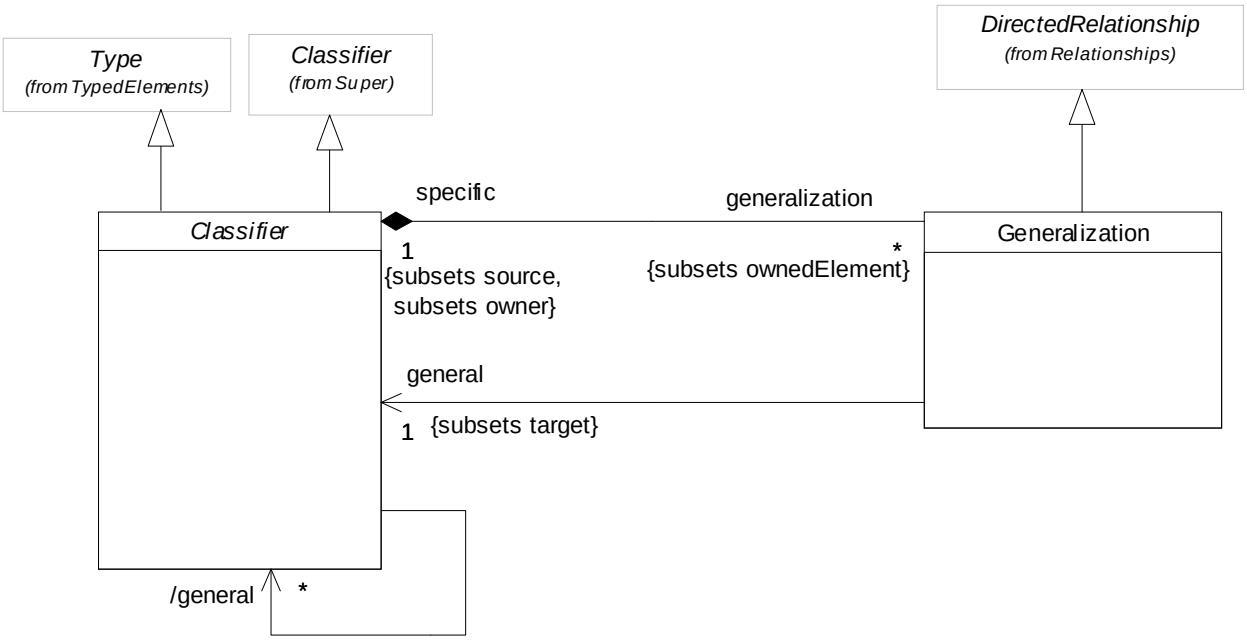

Figure 32 - The elements defined in the Generalizations package

9.8.1 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.

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.

9.8.2 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.

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 an 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 section below.

Presentation Options

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

Examples

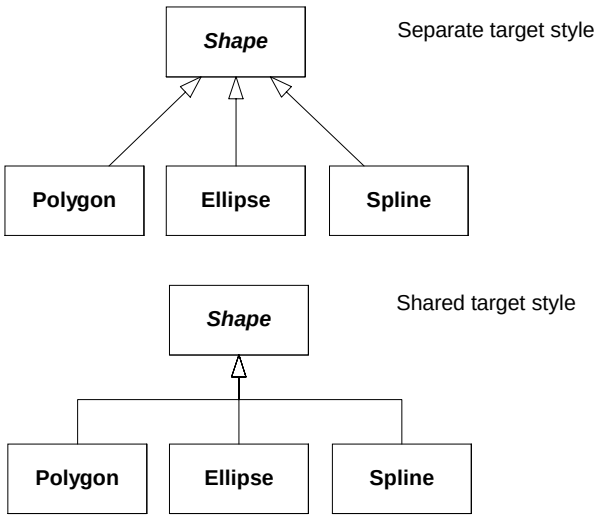

Figure 33 - Examples of generalizations between classes

9.9 Instances package

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

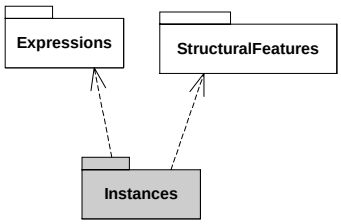

Figure 34 - The Instances package

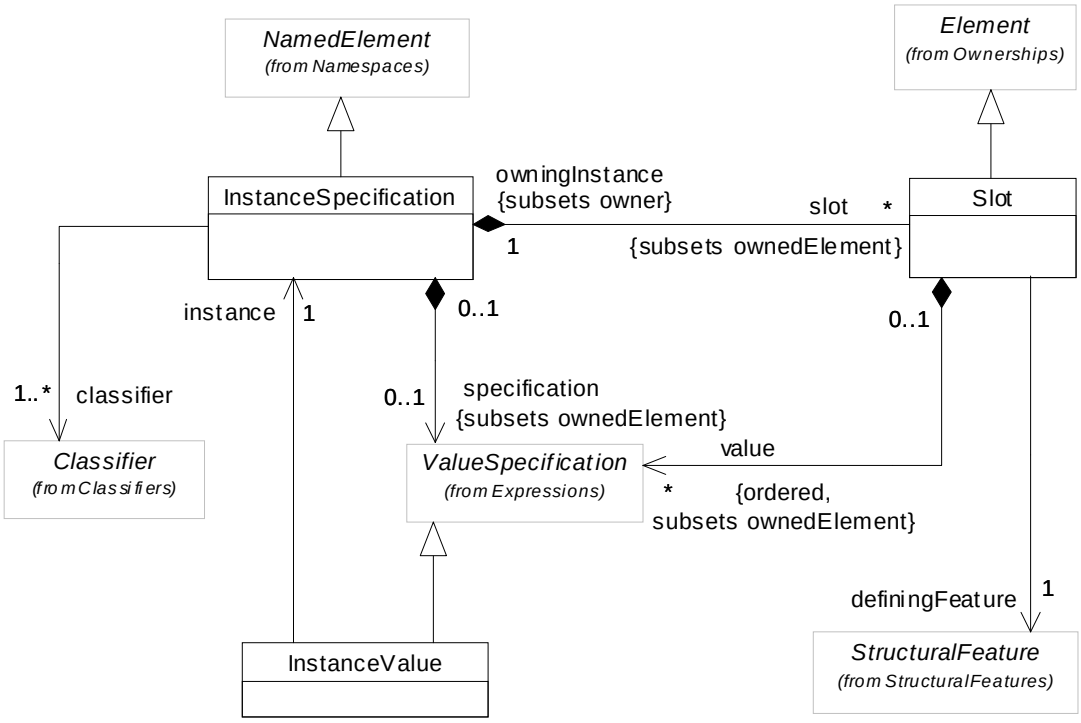

Figure 35 - The elements defined in the Instances package

9.9.1 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.

Attributes

No additional attributes.

Associations

- classifier : Classifier [1..*] 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.

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 (if any), a colon (‘:’) and the classifier name or names. If there are multiple classifiers, the names are all shown separated by commas. Classifier names can be omitted from a diagram.

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.

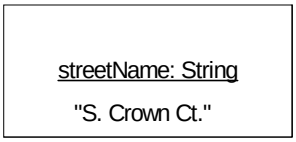

Figure 36 - Specification of an instance of String

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, can optionally be shown.

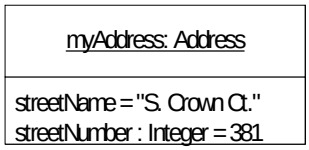

Figure 37 - Slots with values

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.

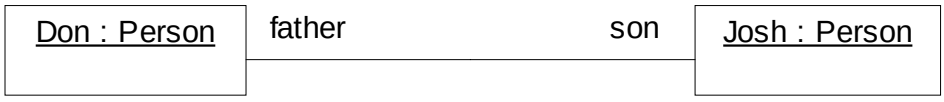

Figure 38 - Instance specifications representing two objects connected by a link

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.

9.9.2 InstanceValue

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

Description

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

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”.

9.9.3 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.

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 : InstanceSpecification [*]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”.

9.10 Literals package

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

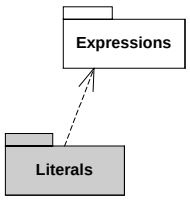

Figure 39 - The Literals package

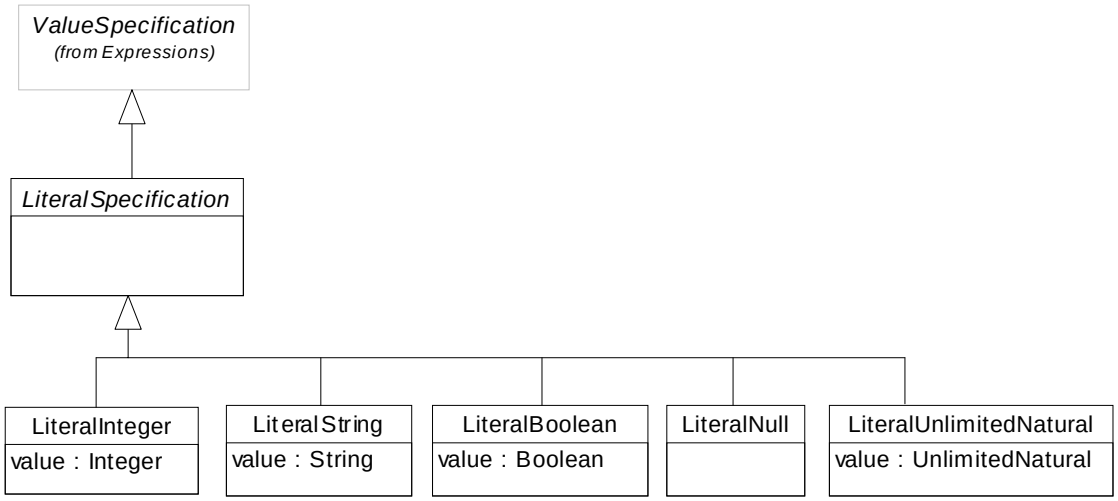

Figure 40 - The elements defined in the Literals package

9.10.1 LiteralBoolean

A literal boolean is a specification of a boolean value.

Description

A literal boolean contains a Boolean-valued attribute.

Attributes

- value: Boolean The specified Boolean 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.

9.10.2 LiteralInteger

A literal integer is a specification of an integer value.

Description

A literal integer contains an Integer-valued attribute.

Attributes

- value: Integer The specified Integer 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.

9.10.3 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.

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.

9.10.4 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.

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.

9.10.5 LiteralString

A literal string is a specification of a string value.

Description

A literal string contains a String-valued attribute.

Attributes

- value: String The specified String 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.

9.10.6 LiteralUnlimitedNatural

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

Description

A literal unlimited natural contains a UnlimitedNatural-valued attribute.

Attributes

- value: UnlimitedNatural The specified UnlimitedNatural 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).

9.11 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.

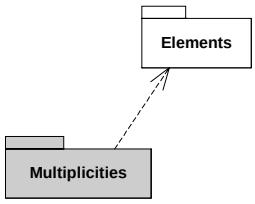

Figure 41 - The Multiplicities package

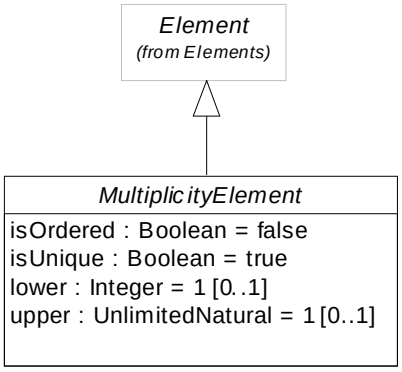

Figure 42 - The elements defined in the Multiplicities package

9.11.1 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.

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 constraint 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] A multiplicity must define at least one valid cardinality that is greater than zero.
upperBound()->notEmpty() **implies** upperBound() > 0

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

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

[3] 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.

Notation

The specific notation for a MultiplicityElement is defined by the concrete subclasses. In general, the notation will include a multiplicity specification 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 an 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 43 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 44 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>
multiplicity_range ::= [lower ‘..’] upper
lower ::= integer
upper ::= unlimited_natural | ‘*’

Examples

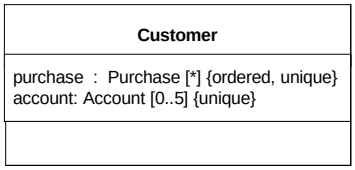

Figure 43 - Multiplicity within a textual specification

Figure 44 - Multiplicity as an adornment to a symbol

Rationale

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

9.12 MultiplicityExpressions package

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

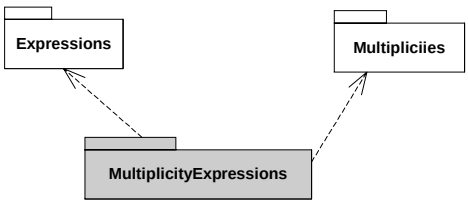

Figure 45 - The MultiplicityExpressions package

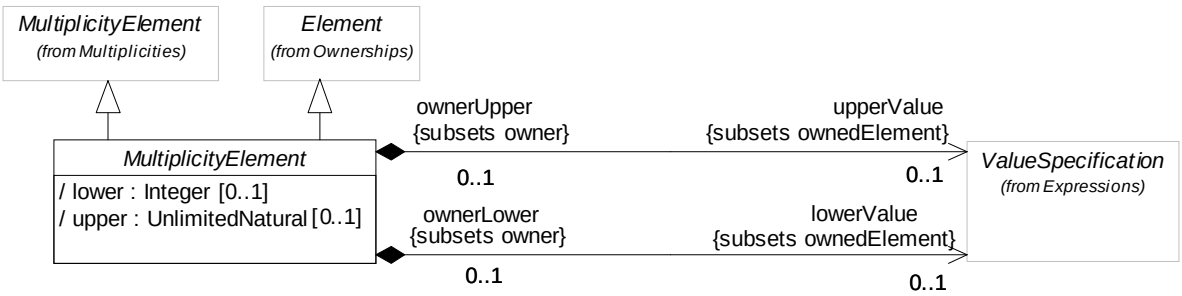

Figure 46 - The elements defined in the MultiplicityExpressions package

9.12.1 MultiplicityElement (specialized)

Description

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

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 71) 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> '}']
multiplicity_range ::= [lower '..'] upper
lower ::= integer | value_specification
upper ::= unlimited_natural | '' | value_specification*
<order_designator> ::= ordered | unordered
<uniqueness_designator> ::= unique | nonunique

9.13 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.

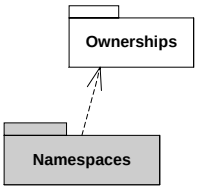

Figure 47 - The Namespaces package

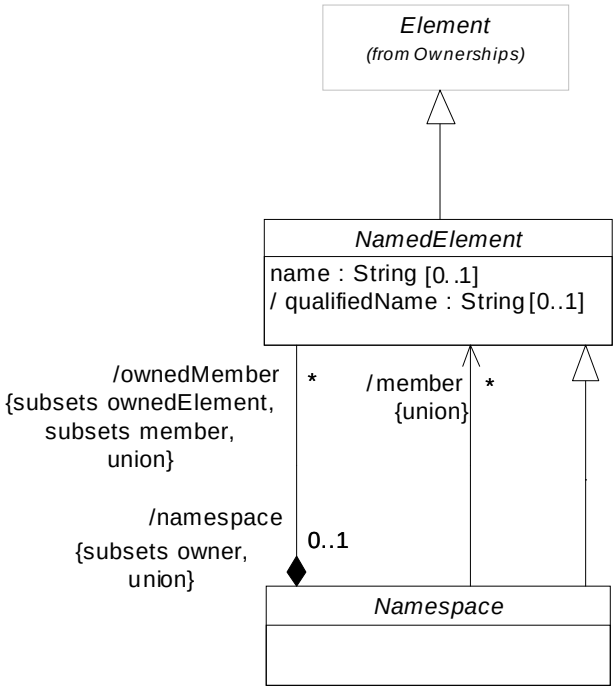

Figure 48 - The elements defined in the Namespaces package

9.13.1 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.

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).

9.13.2 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.

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 which are distinguished by their signature.

Notation

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

9.14 Ownerships package

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

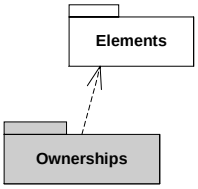

Figure 49 - The Ownerships package

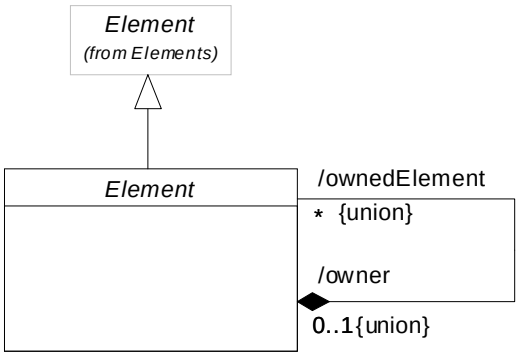

Figure 50 - The elements defined in the Ownerships package

9.14.1 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.

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.

9.15 Redefinitions package

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

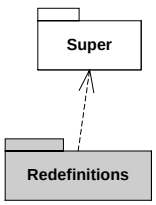

Figure 51 - The Redefinitions package

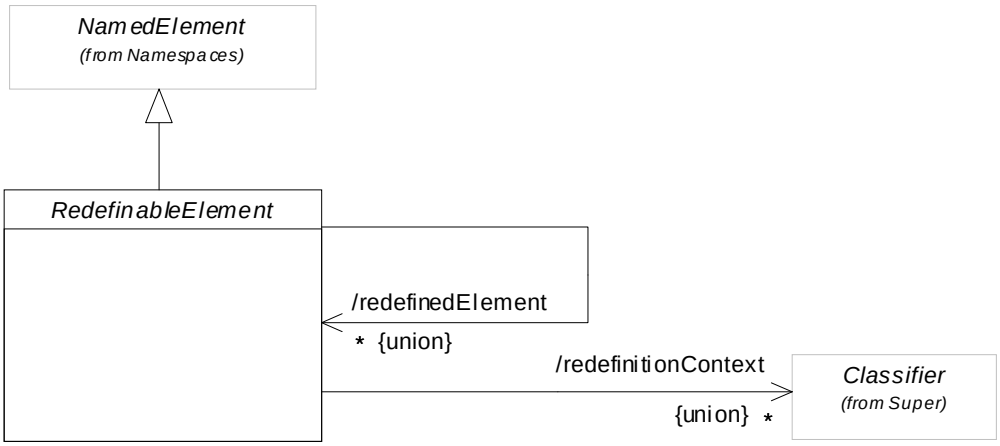

Figure 52 - The elements defined in the Redefinitions package

9.15.1 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.

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 = self.redefinitionContext->exists(c |
                                redefinable.redefinitionContext->exists(c | c.allParents()->includes(r))
                            )
```

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.

9.16 Relationships package

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

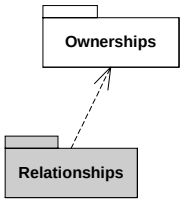

Figure 53 - The Relationships package

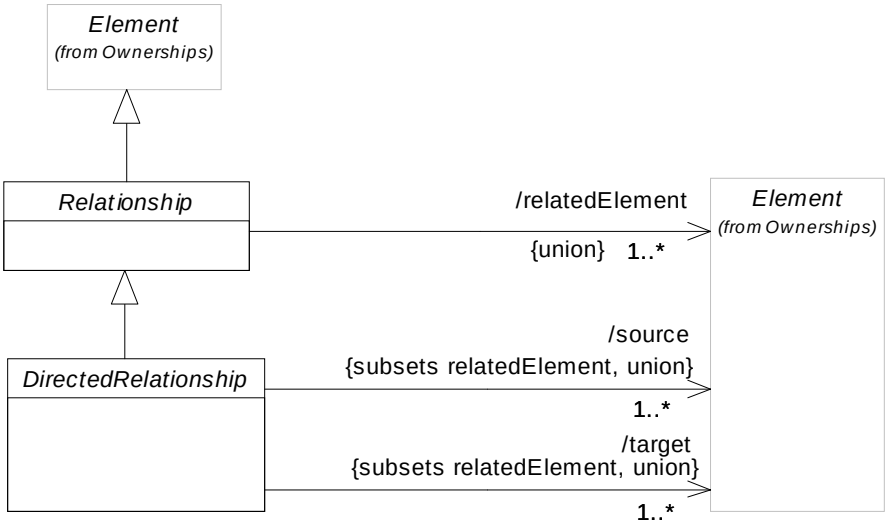

Figure 54 - The elements defined in the Relationships package

9.16.1 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.

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).

9.16.2 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.

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.

9.17 StructuralFeatures package

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

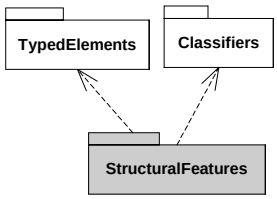

Figure 55 - The StructuralFeatures package

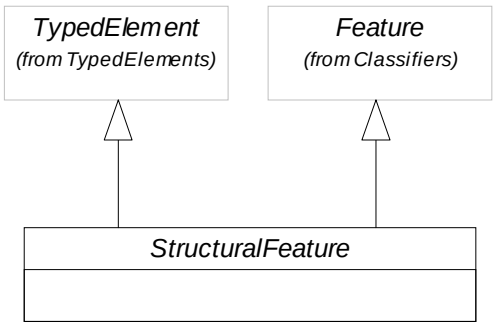

Figure 56 - The elements defined in the StructuralFeatures package

9.17.1 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.

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.

9.18 Super package

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

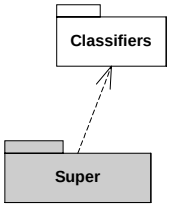

Figure 57 - The Super package

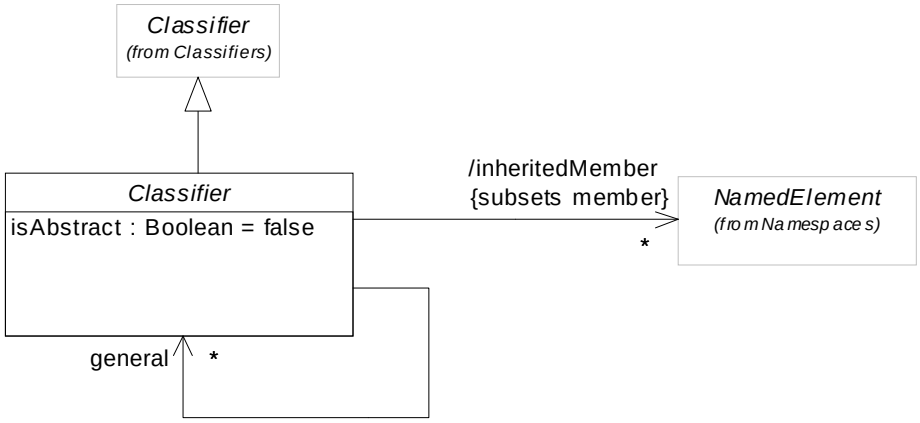

Figure 58 - The elements defined in the Super package

9.18.1 Classifier (as specialized)

Description

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

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 can not 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->includesAll(self.inherit(self.parents()->collect(p | p.inheritableMembers(self)))

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. By default all are visible. 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 =true

[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 an 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 section below.

Presentation Options

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

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

Examples

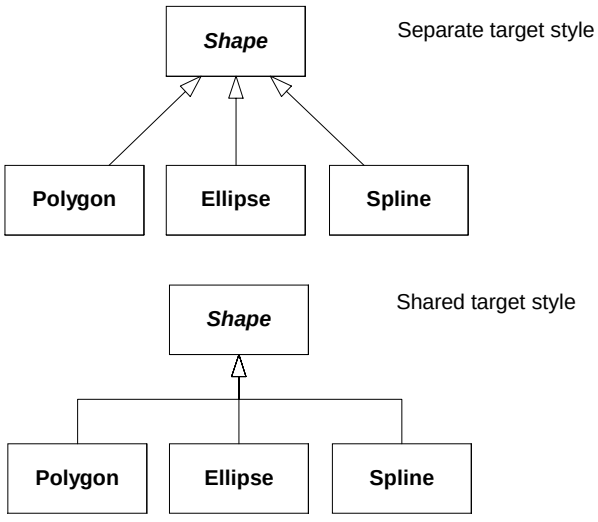

Figure 59 - Example class generalization hierarchy

9.19 TypedElements package

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

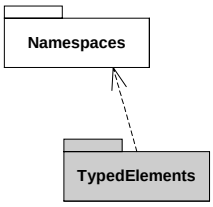

Figure 60 - The TypedElements package

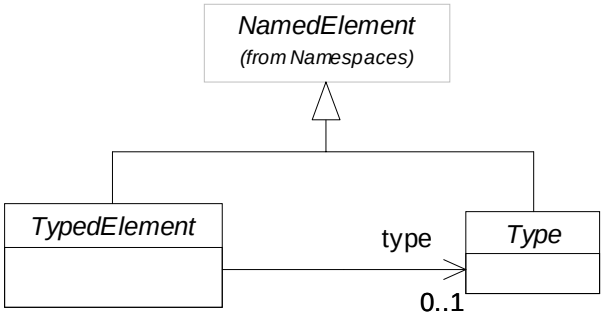

Figure 61 - The elements defined in the TypedElements package

9.19.1 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.

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.

9.19.2 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.

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.

9.20 Visibilities package

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

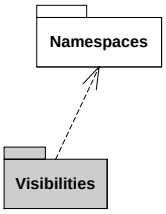

Figure 62 - The Visibilities package

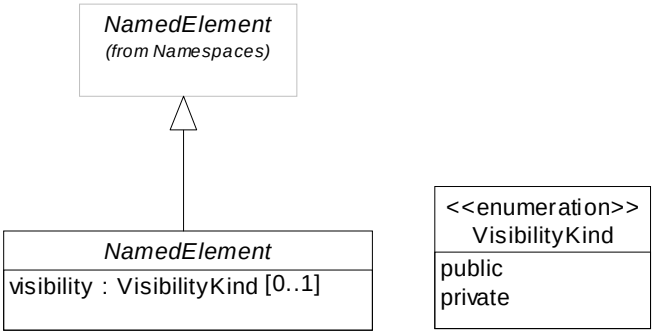

Figure 63 - The elements defined in the Visibilities package

9.20.1 NamedElement (as specialized)

Description

NamedElement has a visibility attribute.

Attributes

- visibility: VisibilityKind [0..1] Determines the visibility of the NamedElement within different Namespaces within the overall model.

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 in different namespaces within a model. It is intended for use in conjunction with import and generalization mechanisms.

9.20.2 VisibilityKind

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

Description

VisibilityKind is an enumeration of the following literal values:

- public
- private

Additional Operations

[1] The query bestVisibility() examines a set of VisibilityKinds, and returns public as the preferred visibility.

```
VisibilityKind::bestVisibility(vis: Set(VisibilityKind)) : VisibilityKind;
bestVisibility = if vis->includes(#public) then #public else #private endif
```

Semantics

VisibilityKind is intended for use in the specification of visibility in conjunction with, for example, the Imports, Generalizations and Packages 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.

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.

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.

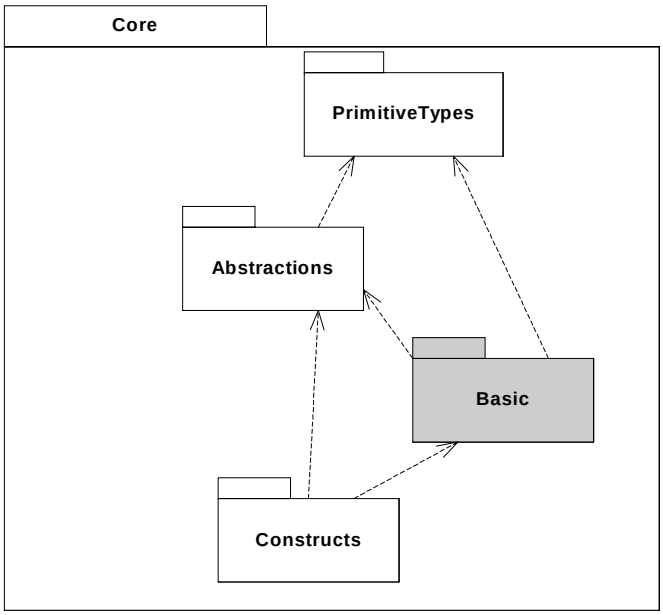

Figure 64 - The Core package is owned by the InfrastructureLibrary package, and contains several subpackages

10.1 Types diagram

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

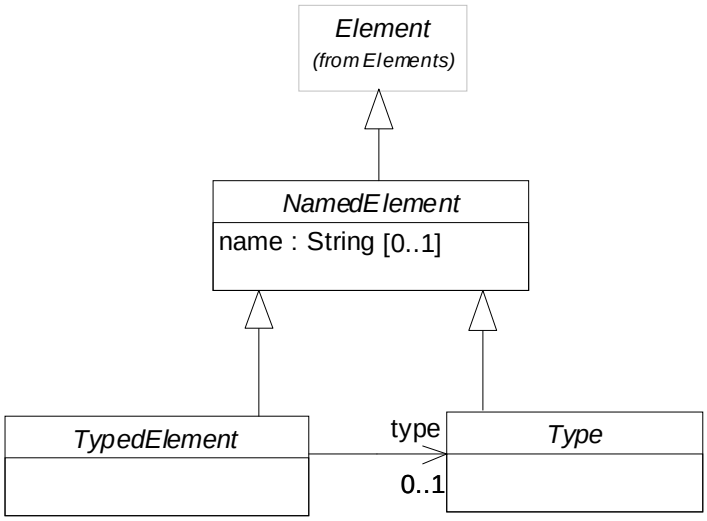

Figure 65 - The classes defined in the Types diagram

10.1.1 Type

Description

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

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.

10.1.2 NamedElement

Description

A named element represents elements with names.

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.

10.1.3 TypedElement

Description

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

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.

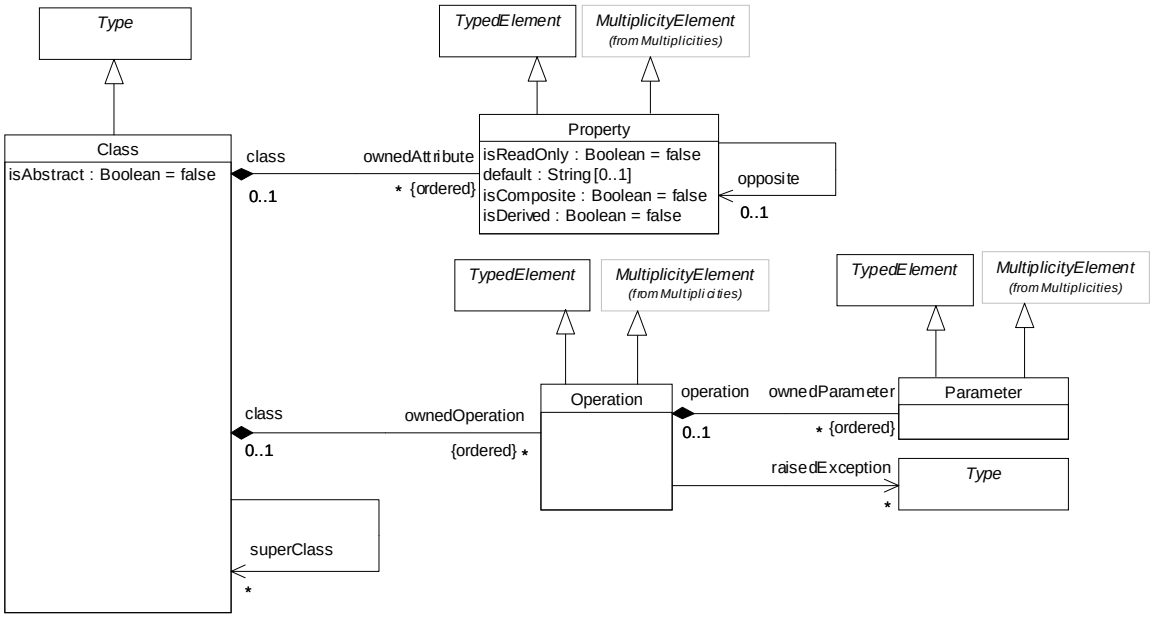

Figure 66 - The classes defined in the Classes diagram

10.2.1 Class

Description

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

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's 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's superclasses. The context of such an invocation is the invoked object.

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 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.

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.

10.2.3 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.

10.2.4 Property

Description

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

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.

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.

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.

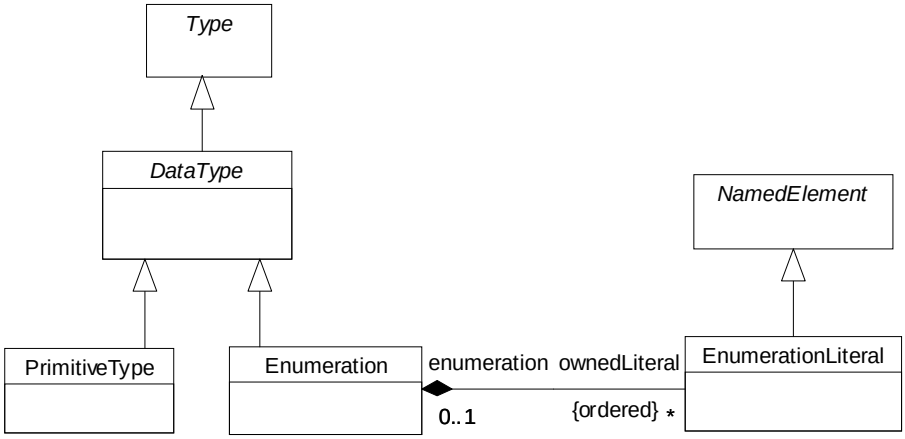

Figure 67 - The classes defined in the DataTypes diagram

10.3.1 DataType

Description

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

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.

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.

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.

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 *Core::PrimitiveTypes* chapter.

10.4 Packages diagram

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

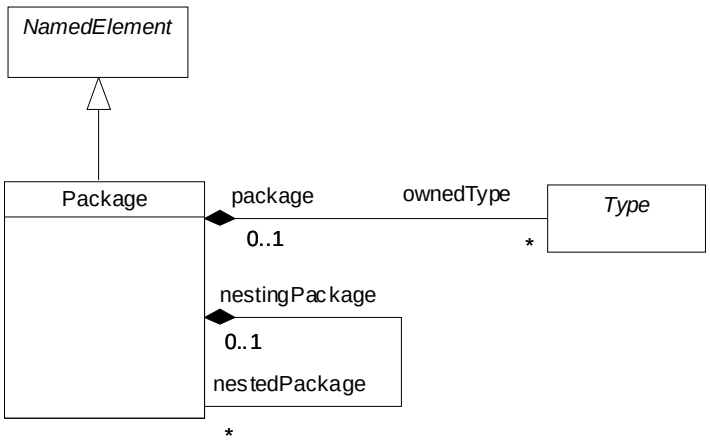

Figure 68 - The classes defined in the Packages diagram

10.4.1 Package

Description

A package is a container for types and other packages.

Attributes

- nestedPackage : Package [*] {composite}The set of contained packages.
- nestingPackage : Package [0..1]The containing package.
- ownedType : Type [*] {composite}The set of contained types.

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.

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 (additional properties)

Description

A type can be contained in a package.

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

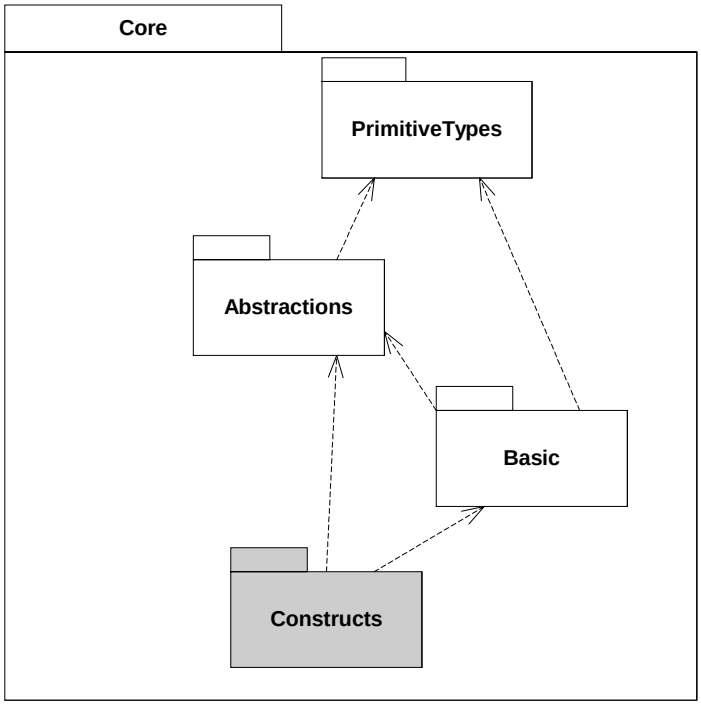

Figure 69 - The Core package is owned by the InfrastructureLibrary package, and contains several subpackages

The Constructs package is specified by a number of diagrams each of which is described in a separate section below. The constructs package is dependent on several other other packages, notably Basic and various packages from Abstractions, as depicted in Figure 70.

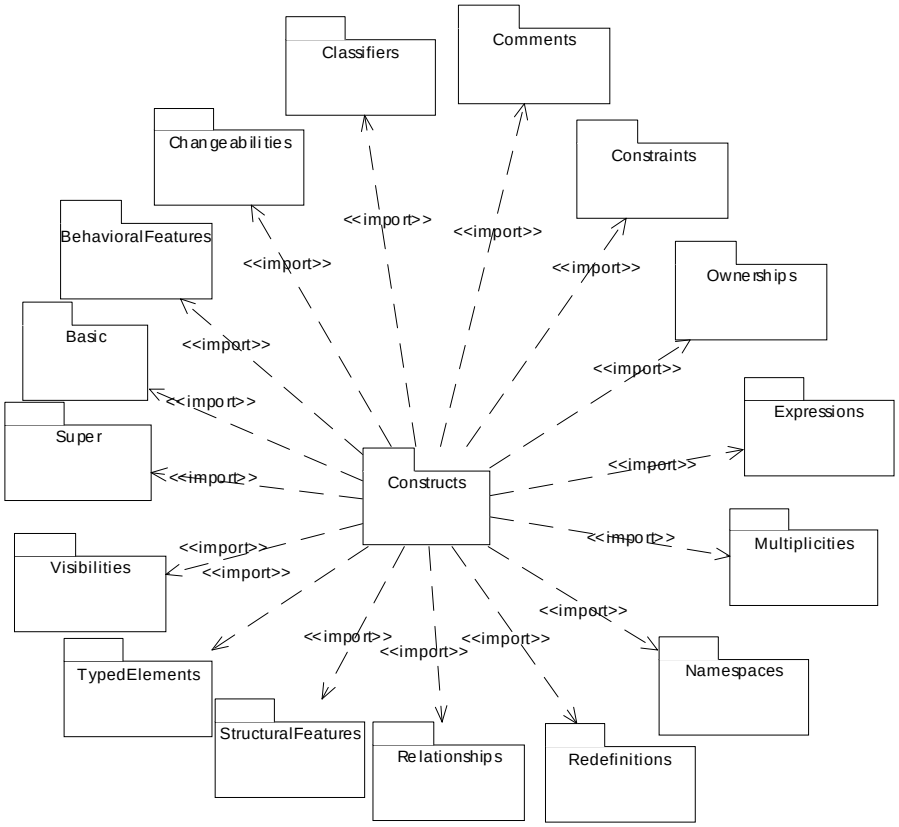

Figure 70 - The Constructs package depends on several other packages

11.1 Root diagram

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

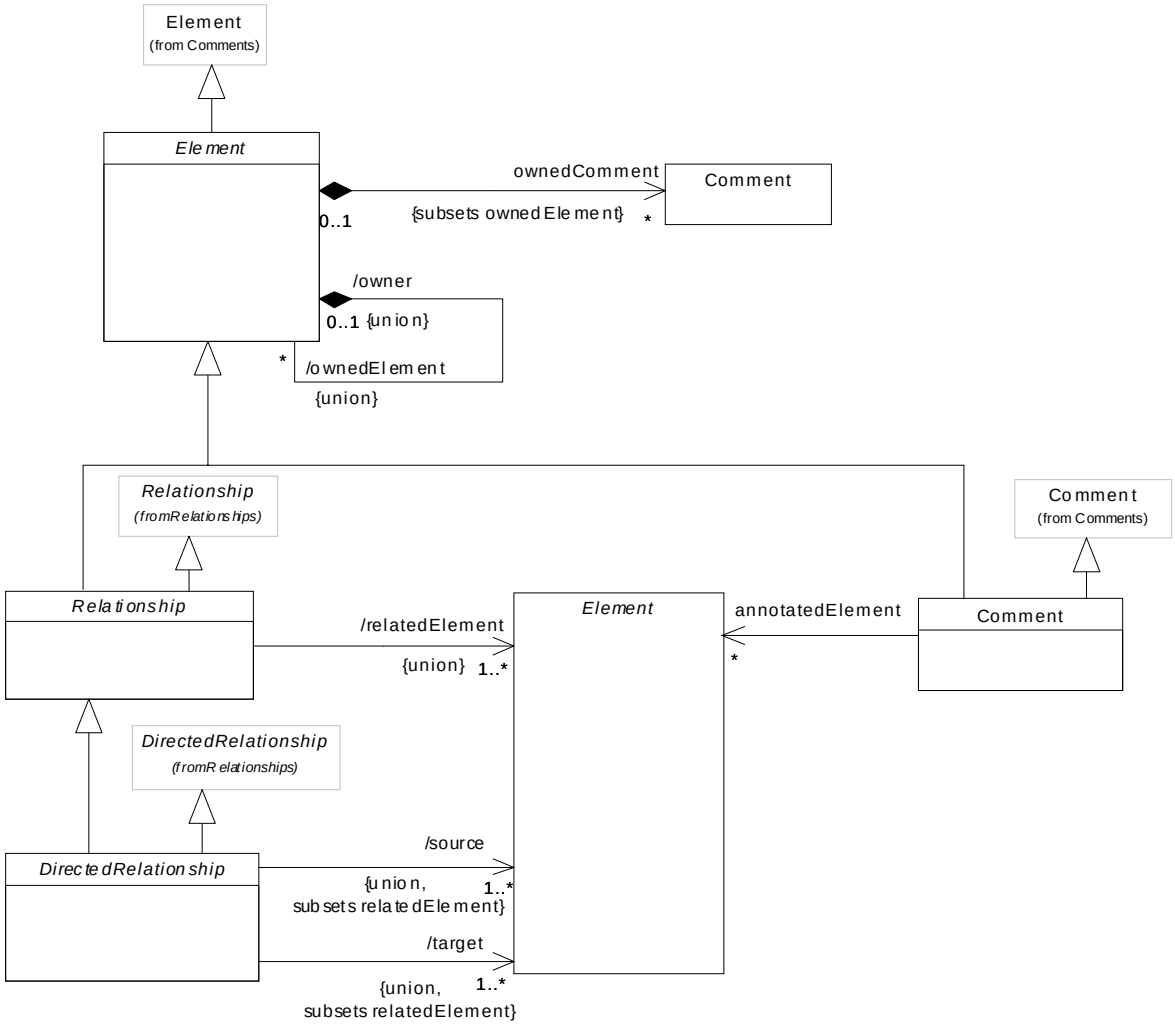

Figure 71 - The Root diagram of the Constructs package

11.1.1 Comment (as specialized)

Description

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

Attributes

No additional attributes.

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 (as specialized)

Description

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

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 (as specialized)

Description

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

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 (as specialized)

Description

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

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.

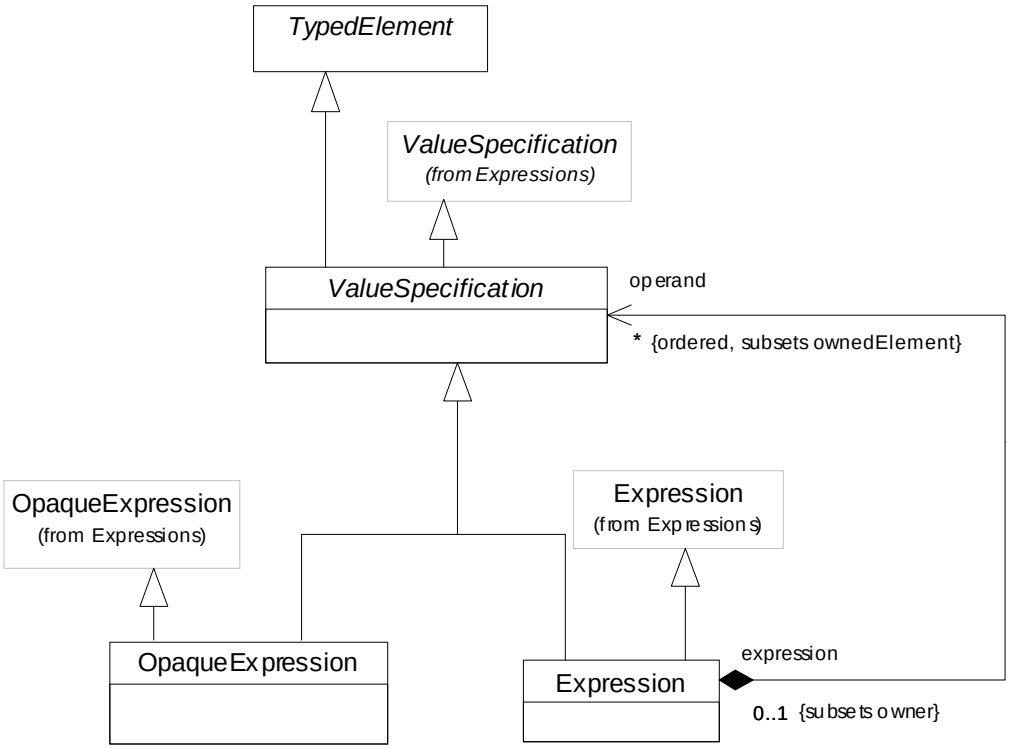

Figure 72 - The Expressions diagram of the Constructs package

11.2.1 Expression (as specialized)

Description

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

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

No additional constraints.

Semantics

No additional semantics.

Notation

No additional notation.

11.2.2 OpaqueExpression (as specialized)

Description

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

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

No additional constraints.

Semantics

No additional semantics.

Notation

No additional notation.

11.2.3 ValueSpecification (as specialized)

Description

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

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.

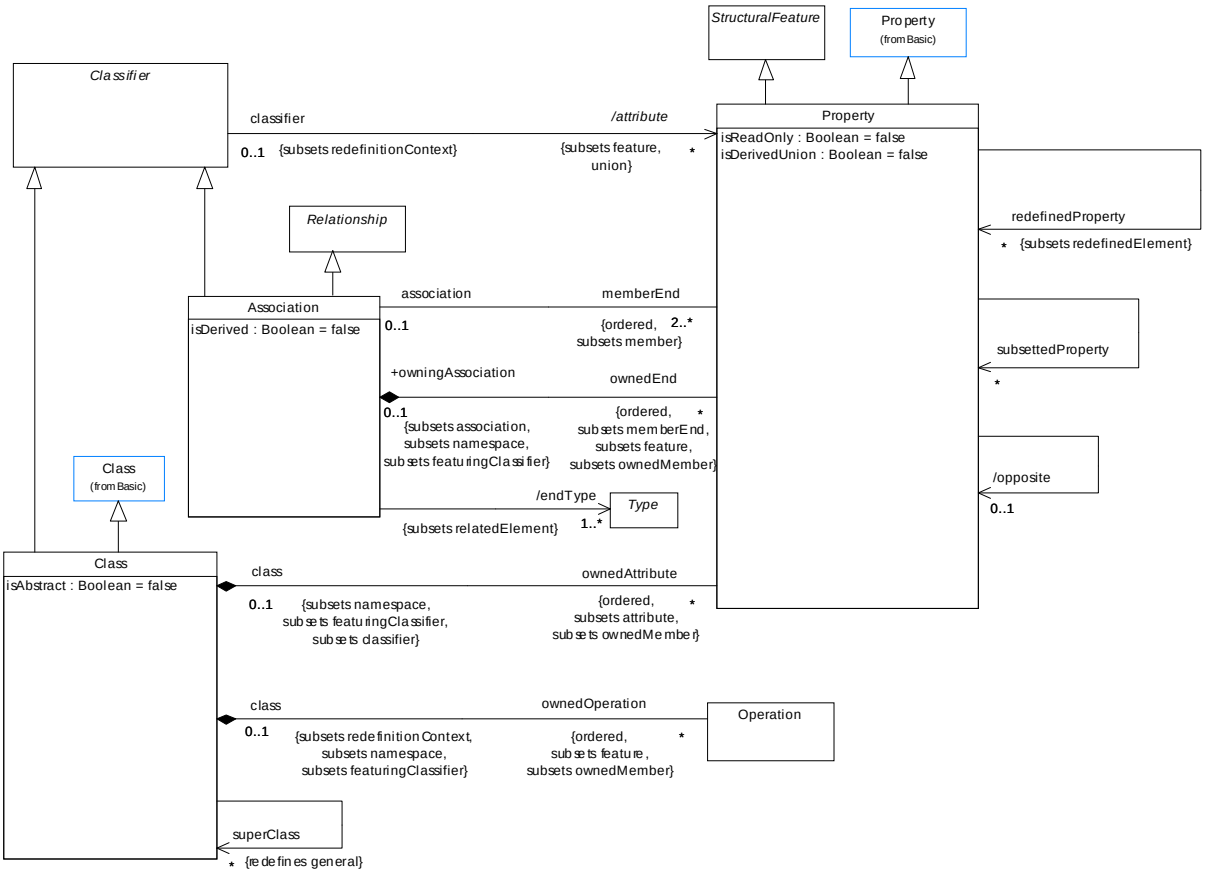

Figure 73 - The Classes diagram of the Constructs package

11.3.1 Association

An association describes a set of tuples whose values refers to typed instances. An instance of an association is called a link.

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.

When a property is owned by an association it represents a non-navigable end of the association. In this case the property does not appear in the namespace of any of the associated classifiers. When a property at an end of an association is owned by one of the associated classifiers it represents a navigable end of the association. In this case the property is also an attribute of the associated classifier. Only binary associations may have navigable ends.

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 non-navigable 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.

Constraints

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

```
self.parents()->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.

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

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

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.

An end of one association may be marked as a *subset* of an end of another in circumstances where (a) both have the same number of ends, and (b) each of the set of types connected by the subsetting association conforms to a corresponding type connected by the subsetted association. In this case, given a set of specific instances for the other ends of both associations, the collection denoted by the subsetting end is fully included in the collection denoted by the subsetted end.

An end of one association may be marked as *redefining* an end of another in circumstances where (a) both have the same number of ends, and (b) each of the set of types connected by the redefing association conforms to a corresponding type connected by the redefined association. In this case, given a set of specific instances for the other ends of both associations, the collections denoted by the redefining and redefined ends are the same.

Associations may be specialized. The existence of a link of a specializing association implies the existence of a link relating the same set of instances in a specialized association.

The semantics of navigable association ends are the same as for attributes.

For n-ary associations, the lower multiplicity of an end is typically 0. If the 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 define transitive asymmetric relationships—their links form a directed, acyclic graph. Composition is represented by the *isComposite* attribute on the part end of the association being set to *true*.

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.

The interaction of association specialization with association end redefinition and subsetting 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.
- 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 74).
- 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.
- A property string enclosed in curly braces. The following property strings can be applied to an association end:
  {subsets <property-name>} to show that the end is a subset of the property called <property-name>.
  {redefines <end-name>} to show that the end redefines the one called <end-name>.
  {union} to show that the end is derived by being the union of its subsets.
  {ordered} to show that the end represents an ordered set.
  {bag} to show that the end represents a collection that permits the same element to appear more than once.
  {sequence} or {seq} to show that the end represents a sequence (an ordered bag).
  if the end is navigable, any property strings that apply to an attribute.

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 assocation end name.

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 74 shows a binary association from *Player* to *Year* named *PlayedInYear*. The solid triangle indicates the order of

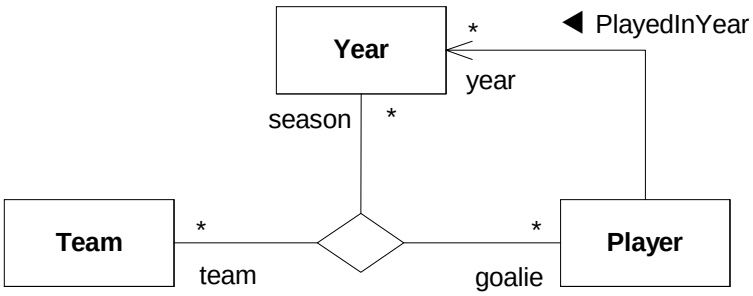

Figure 74 - Binary and ternary associations

reading: *Player PlayedInYear Year*. The figure further shows a ternary association between *Team*, *Year*, and *Player* with ends named *team*, *season*, and *goalie* respectively.

The following example shows association ends with various adornments.

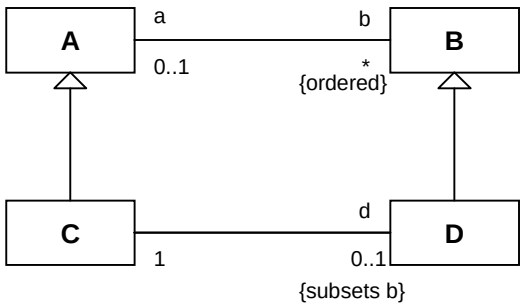

Figure 75 - Association ends with various adornments

The following adornments are shown on the four association ends in Figure 75.

- 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.

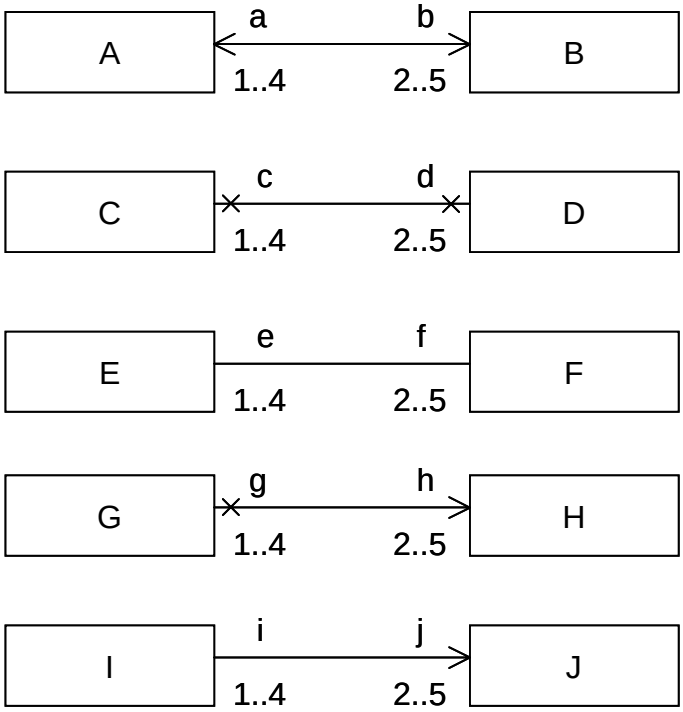

Figure 76 - Examples of navigable ends

In Figure 76:

- 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 77 shows a navigable end using attribute notation. A navigable end is an attribute, so it can be shown using attribute notation. Normally this notation would be used in conjunction with the line-arrow notation to make it perfectly clear that the navigable ends are also attributes.

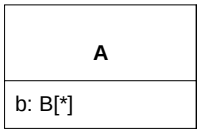

Figure 77 - Example of navigable end shown with attribute notation

Figure 78 shows the notation for a derived union. The attribute A::b is derived by being the strict union of all of the

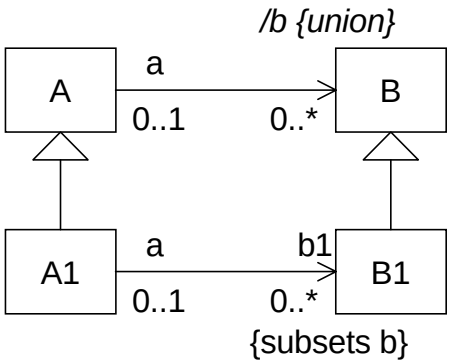

Figure 78 - Example of a derived union.

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 79 shows the black diamond notation for composite aggregation.

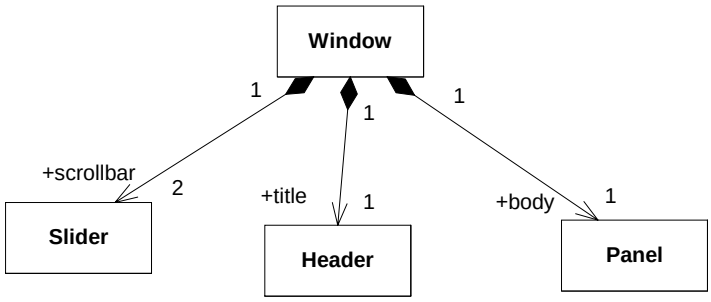

Figure 79 - Composite aggregation is depicted as a black diamond

11.3.2 Class (as specialized)

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.

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

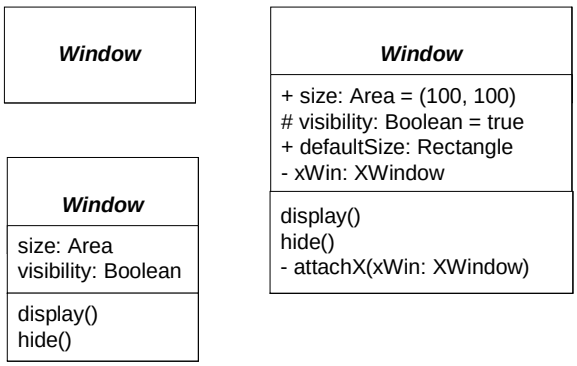

Figure 80 -Class notation: details suppressed, analysis-level details, implementation-level details

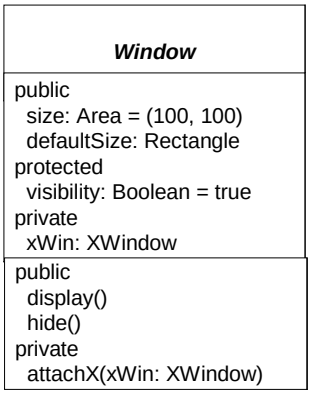

Figure 81 - Class notation: attributes and operations grouped according to visibility.

11.3.3 Classifier (additional properties)

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.

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 that can be parsed into the various properties of an attribute. The basic syntax is (with optional parts shown in braces):

[*visibility***] [***/***] *name* [***: type***] [***multiplicity***] [***= default***] [***{ property-string }***]**

In the following bullets, each of these parts is described:

- *visibility* is a visibility symbol such as + or -. See Section 9.20.2, "VisibilityKind," on page 93.

- / means the attribute is derived.
- *name* is the name of the attribute.
- *type* identifies a classifier that is the attribute's type.
- *multiplicity* shows the attribute's multiplicity in square brackets. The term may be omitted when a multiplicity of 1 (exactly one) is to be assumed. See Section 9.11.1, "MultiplicityElement," on page 71
- *default* is an expression for the default value or values of the attribute.
- *property-string* indicates property values that apply to the attribute. The property string is optional (the braces are omitted if no properties are specified).

The following property strings can be applied to an attribute: {readOnly}, {union}, {subsets <property-name>}, {redefines <property-name>}, {ordered}, {bag}, {seq} or {sequence}, and {composite}.

An attribute with the same name as an attribute that would have been inherited is interpreted to be a redefinition, without the need for a {redefines <x>} property string. Note that a redefined attribute is not inherited into a namespace where it is redefined, so its name can be reused in the featuring classifier, either for the redefining attribute, or alternately for some other attribute.

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.

Style Guidelines

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

Examples

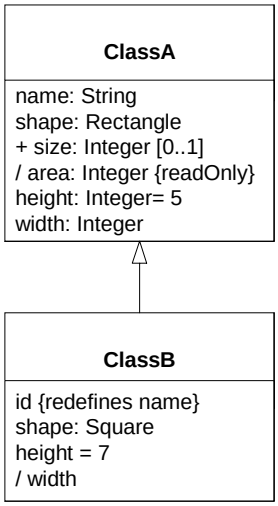

Figure 82 - Examples of attributes

The attributes in *Figure 82* 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 83.

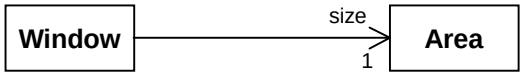

Figure 83 - Association-like notation for attribute

11.3.4 Operation (additional properties)

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.

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 (as specialized)

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*.

When a property is owned by a class it represents an attribute. In this case it relates an instance of the class to a value or set of values of the type of the attribute.

When a property is owned by an association it represents a non-navigable end of the association. In this case 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.

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*.

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.

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->notEmpty() and association.memberEnd->size() = 2 then
        let otherEnd = (association.memberEnd - self)->any() in
            if otherEnd.owningAssociation->notEmpty() then otherEnd else Set{} endif
    else Set {}
    endif
```

[2] A specialization of a composite aggregation is also a composite aggregation.

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

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

[4] 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)))))
```

[5] A navigable property (one that is owned by a class) can only be redefined or subsetted by a navigable property.

```
(subsettedProperty->exists(sp | sp.class->notEmpty())
        implies class->notEmpty())
and
(redefinedProperty->exists(rp | rp.class->notEmpty())
        implies class->notEmpty())
```

[6] 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() ))
```

[7] Only a navigable property can be marked as readOnly.

isReadOnly **implies** class->notEmpty()

[8] A derived union is derived.

isDerivedUnion **implies** isDerived

[9] A derived union is read only
isDerivedUnion **implies** isReadOnly

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, the multiplicity of the redefining property (if.specified) is contained in the multiplicity of the redefined property, and the redefining property is derived if the redefined property is derived.

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

[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
```

Semantics

When a property is owned by a class or data type via ownedAttribute, then it represents an *attribute* of the class or data type. When owned by an association via ownedEnd, it represents a *non-navigable 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 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 appropriately change the source information of the derivation. 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 which 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 (attribute) 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.

Notation

Notation for properties is defined separately for their use as attributes and association ends. Examples of subsetting and derived union are shown for associations.

11.3.6 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.

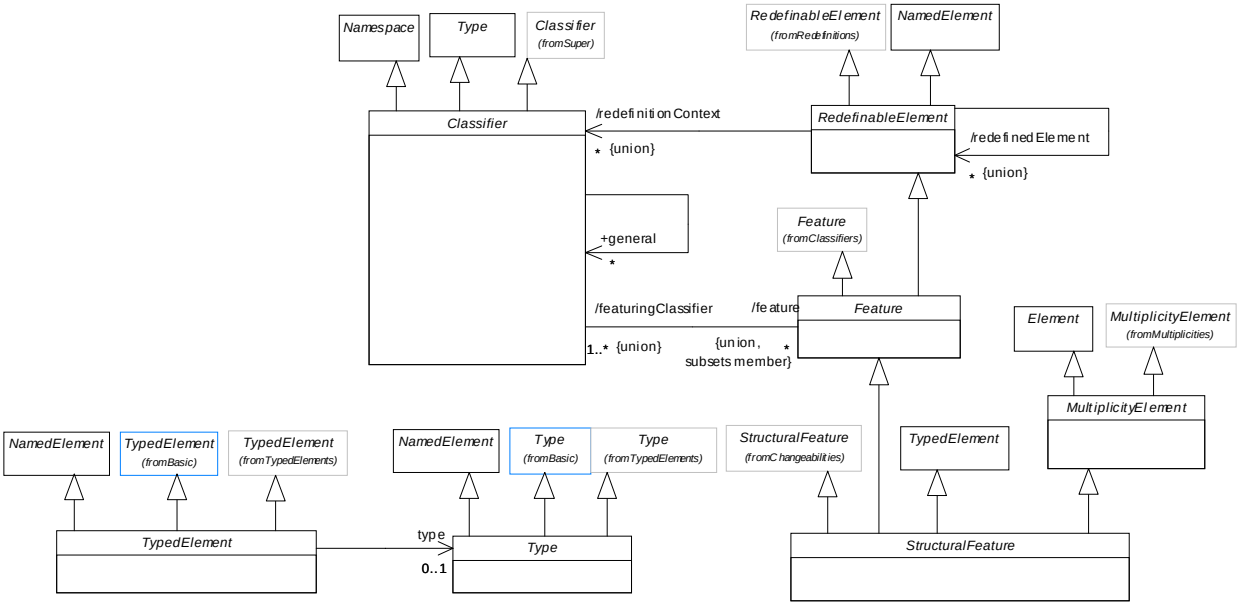

Figure 84 - The Classifiers diagram of the Constructs package

11.3.7 Classifier (as specialized)

Description

Constructs::Classifier merges the definitions of Classifier from *Basic* and *Abstractions*. It adds specializations from *Constructs::Namespace* and *Constructs::Type*.

Attributes

No additional attributes.

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

No additional constraints.

Semantics

No additional semantics.

Notation

As defined in Abstractions.

11.3.8 Feature (as specialized)

Description

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

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.

11.3.9 MultiplicityElement (as specialized)

Description

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

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

No additional constraints.

Semantics

No additional semantics.

Notation

As defined in Abstractions.

11.3.10RedefinableElement (as specialized)

Description

Constructs::RedefineableElement reuses the definition of *RedefineableElement* from *Abstractions*. It adds a specialization from *Constructs::NamedElement*.

Attributes

No additional attributes.

Associations

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

Constraints

No additional constraints.

Semantics

No additional semantics.

Notation

As defined in Abstractions.

11.3.11StructuralFeature (as specialized)

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.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

No additional constraints.

Semantics

No additional semantics.

Notation

As defined in Abstractions.

11.3.12Type (as specialized)

Description

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

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

No additional constraints.

Semantics

No additional semantics.

Notation

As defined in Abstractions.

11.3.13TypedElement (as specialized)

Description

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

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.

11.4 Constraints diagram

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

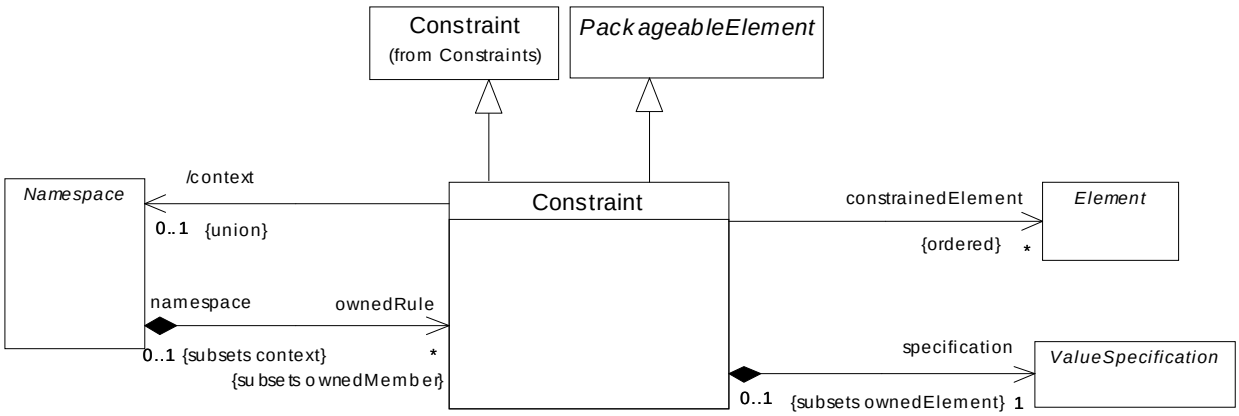

Figure 85 - The Classes diagram of the Constructs package

11.4.1 Constraint

Description

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

Attributes

No additional attributes.

Associations

- constrainedElement: Element Redefines the corresponding property in *Abstractions*.
- / context: Namespace [0..1] Redefines the corresponding property in *Abstractions*. This is a derived union.

- specification: ValueSpecificationRedefines the corresponding property in *Abstractions*. Subsets *Element.ownedElement.*

Constraints

No additional constraints.

Semantics

No additional semantics.

Notation

No additional notation.

11.4.2 Namespace (additional properties)

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.

Attributes

No additional attributes.

Associations

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

Constraints

No additional constraints.

Semantics

No additional semantics.

11.5 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 that 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.

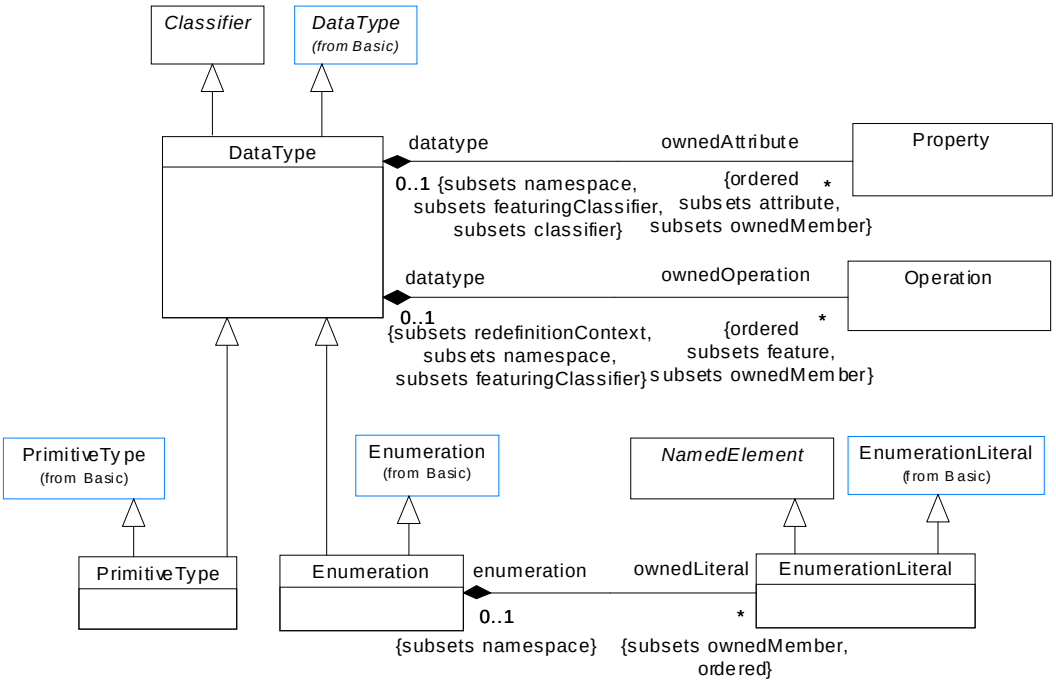

Figure 86 - The classes defined in the DataTypes diagram

11.5.1 DataType (as specialized)

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

Description

Constructs::DataType reuses the definition of *DataType* from *Basic*. It adds a specialization to *Constructs::Classifier.*

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

A DataType may also contain attributes to support the modeling of structured data types.

Attributes

No additional attributes.

Associations

- ownedAttribute: Attribute[*] The Attributes owned by the DataType. Subsets *Classifier::attribute* and *Element::ownedMember.*

- ownedOperation: Operation[*] The Operations owned by the DataType. Subsets *Classifier::feature* and *Element::ownedMember.*

Constraints

No additional constraints.

Semantics

A data type is a special kind of classifier, similar to a class, whose instances are values (not objects). For example, the integers and strings are usually treated as values. A value does not have an identity, so two occurrences of the same value cannot be differentiated. Usually, a data type is used for specification of the type of an attribute. An enumeration type is a user-definable type comprising a finite number of values.

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 denotated using the rectangle symbol with keyword «dataType» or, when it is referenced by e.g. an attribute, denoted by a string containing the name of the data type.

Presentation Options

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

Examples

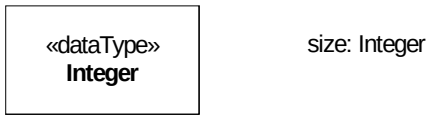

Figure 87 - Notation of data type: to the left is an icon denoting a data type and to the right is a reference to a data type which is used in an attribute

11.5.2 Enumeration (as specialized)

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.

Attributes

No additional attributes.

Associations

- ownedLiteral: EnumerationLiteral[*]The ordered set of literals for this Enumeration. Subsets *Element::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

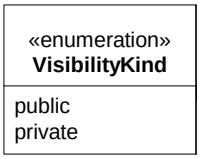

Figure 88 - Example of an enumeration

11.5.3 EnumerationLiteral (as specialized)

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.

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 a compartment of the enumeration notation. See “Enumeration (as specialized)”.

Examples

See “Enumeration (as specialized)”.

11.5.4 Operation (additional properties)

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.

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.

11.5.5 PrimitiveType (as specialized)

A primitive type defines a predefined data type, without any relevant substructure (i.e. it has no parts). 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 Chapter 12, “Core::PrimitiveTypes”).

Attributes

No addtional 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 Chapter 12, “Core::PrimitiveTypes”) may be denoted with the same notation as provided for references to such instances (see the subtypes of “ValueSpecification”).

Examples

See Chapter 12, “Core::PrimitiveTypes” for examples.

11.5.6 Property (additional properties)

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.

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.

11.6 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.

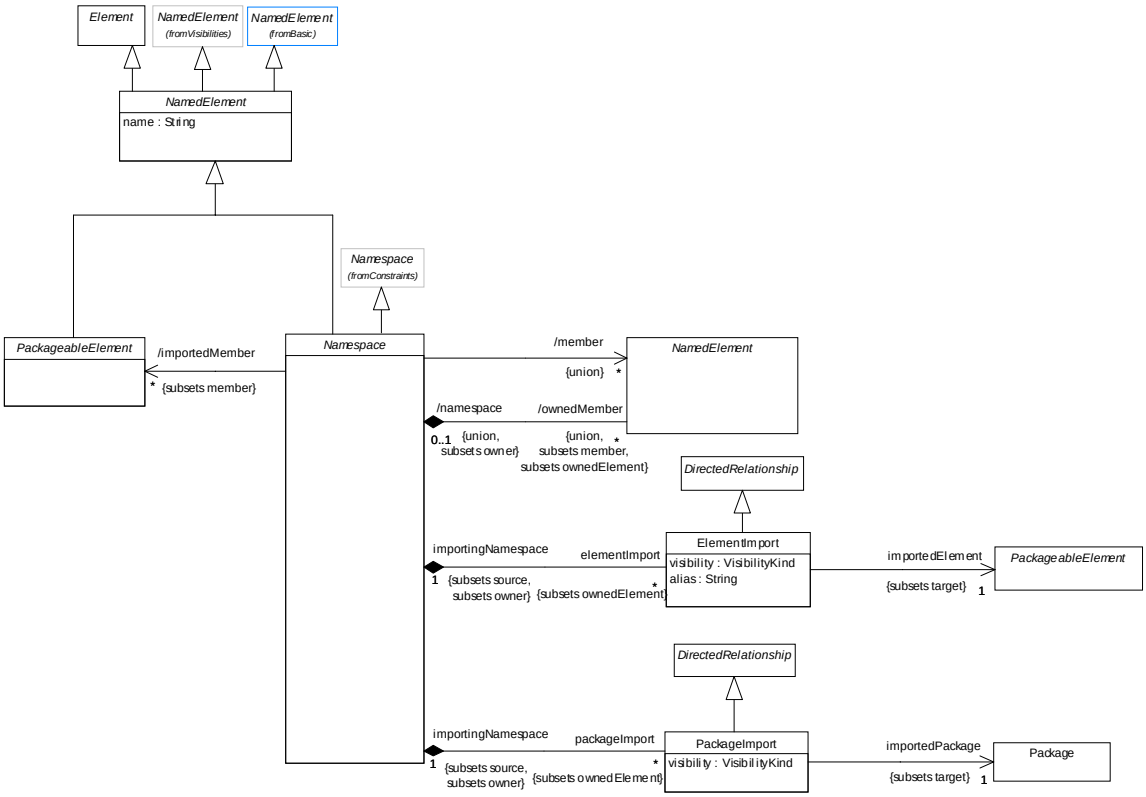

Figure 89 - The Namespaces diagram of the Constructs package

11.6.1 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.

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.

- 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 ele-ment import, and the unqualified name refers to the imported element. The outer name can be accessed using its qual-ified 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 names of the imported elements must be qualified in order to be used and the elements are not added to the importing namespace. 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 member 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 key-word «access» is shown.

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>} or *{element access <qualifiedName>}*

Optionally, the aliased name may be show as well:

{element import <qualifiedName> as <alias>} or *{element access <qualifiedName> as <alias>}*

Examples

The element import that is shown in Figure 90 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.

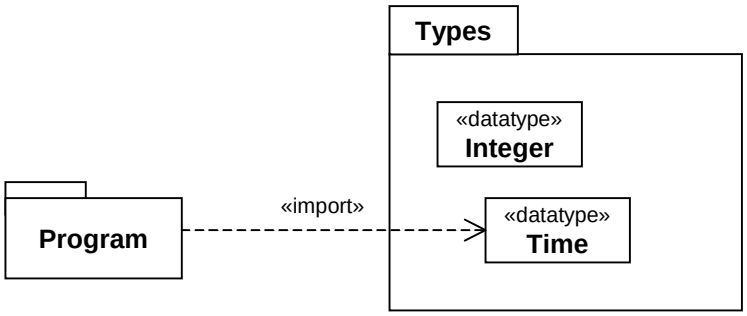

Figure 90 - Example of element import

In Figure 91, the element import is combined with aliasing, meaning that the type Types::Real will be referred to as Double in the package Shapes.

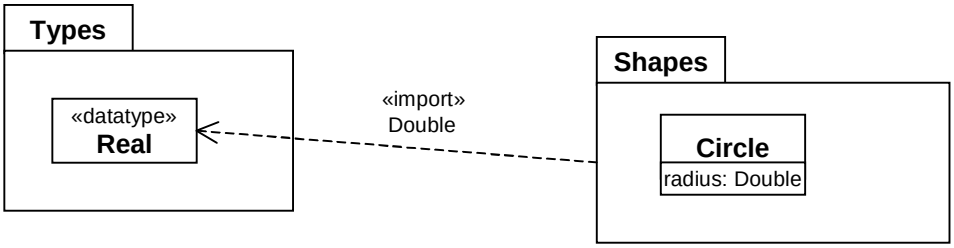

Figure 91 - Example of element import with aliasing

11.6.2 NamedElement (as specialized)

Description

Constructs::NamedElement reuses the definition of *NamedElement* from *Abstractions::Visibilitites*. It adds specializations from *Constructs::Element* and *Basic::NamedElement*.

Attributes

- name: String [0..1] Redefines the corresponding attributes from *Basic::NamedElement* and *Abstractions::Visibilities::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.

11.6.3 Namespace (as specialized)

Description

Constructs::Namespace reuses the definition of *Abstractions::Constraints::Namespace*.

A namespace has the ability to import either individial 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.

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.

```
self.importedMember->includesAll(self.importedMembers(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 which 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.

11.6.4 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.

Attributes

No additional attributes.

Associations

No additional associations.

Constraints

No additional constraints.

Semantics

No additional semantics.

Notation

No additional notation.

11.6.5 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.

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 member import is shown using a dashed arrow with an open arrowhead from the importing namespace to the imported namespace. The keyword «import» is shown near the dashed arrow.

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 an element import by having a text that uniquely identi-fies the imported element within curly brackets either below or after the name of the namespace. The textual syntax is then:

{import <qualifiedName>} or *{access <qualifiedName>}*

Examples

In Figure 92, 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.

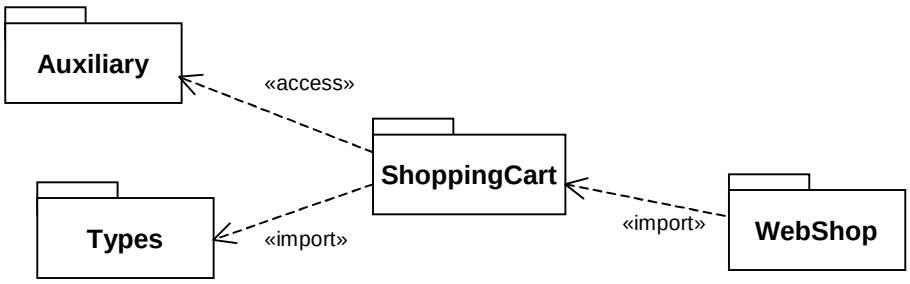

Figure 92 - Examples of public and private package imports

11.7 Operations diagram

The Operations diagram of the Constructs package specifies the BehavioralFeature, Operation, and Parameter constructs.

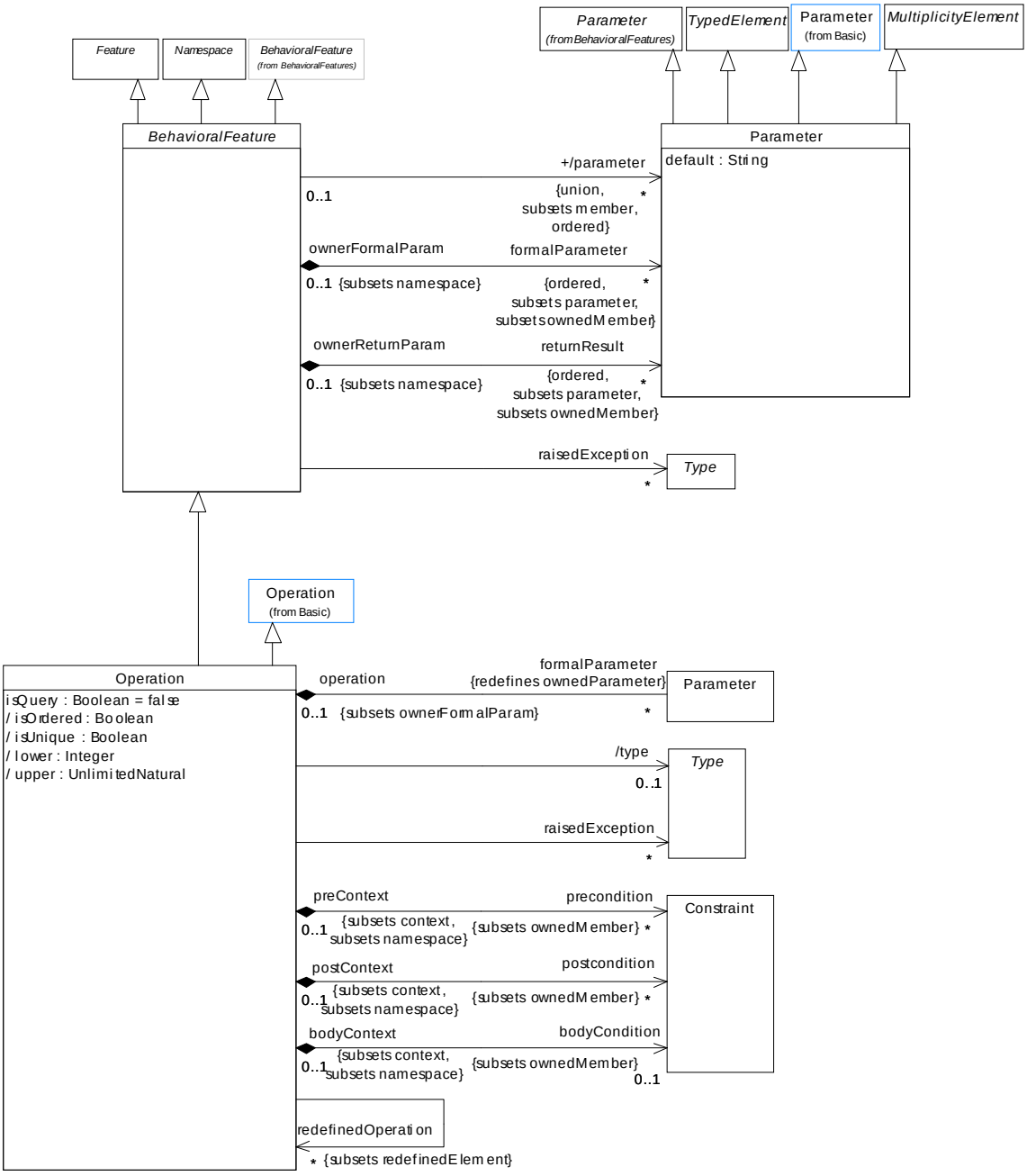

Figure 93 - The Operations diagram of the Constructs package

11.7.1 BehavioralFeature (as specialized)

Description

Constructs::BehavioralFeature reuses the definition of *BehavioralFeature* from *Abstractions::BehavioralFeatures*. It adds specializations to *Constructs::Namespace* and *Constructs::Feature*.

Attributes

No additional attributes.

Associations

- formalParameter: Parameter[*] Specifies the ordered set of formal parameters of this BehavioralFeature. Subsets *BehavioralFeature::parameter* and *Namespace::ownedMember.*
- raisedException: Type[*] References the Types representing exceptions that may be raised during an invocation of this feature.
- returnResult: Parameter[*] Specifies the ordered set of return results of this BehavioralFeature. Subsets *Behavioral-Feature::parameter* and *Namespace::ownedMember.*

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.parameter->collect(type))
        else true
        endif
    else true
    endif
```

Semantics

The formal parameters define the type, and number, of arguments that must be provided when invoking the BehavioralFeature. The return results define the type, and number, of arguments that will be returned from a successful invocation. A BehavioralFeature may raise an exception during its invocation.

Notation

No additional notation.

11.7.2 Operation (as specialized)

An operation is a behavioral feature of a classfier 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.

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 BehavioralFeature 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.ownedMember.*
- formalParameter: Parameter[*] — Specifies the formal parameters for this Operation. Redefines *Basic::Operation.ownedParameter* and *BehavioralFeature.formalParameter.*
- postcondition: Constraint[*] — An optional set of Constraints specifying the state of the system when the Operation is completed. Subsets *Namespace.ownedMember.*
- precondition: Constraint[*] — An optional set of Constraints on the state of the system when the Operation is invoked. Subsets *Namespace.ownedMember.*
- 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] If this operation has a single return result, isOrdered equals the value of isOrdered for that parameter. Otherwise isOrdered is false.

```
isOrdered = if returnResult->size() = 1 then returnResult->any().isOrdered else false endif
```

[2] If this operation has a single return result, isUnique equals the value of isUnique for that parameter. Otherwise isUnique is true.

isUnique = **if** returnResult->size() = 1 **then** returnResult->any().isUnique **else** true **endif**

[3] If this operation has a single return result, lower equals the value of lower for that parameter. Otherwise lower is not defined.

lower = **if** returnResult->size() = 1 **then** returnResult->any().lower **else** Set{} **endif**

[4] If this operation has a single return result, upper equals the value of upper for that parameter. Otherwise upper is not defined.

upper = **if** returnResult->size() = 1 **then** returnResult->any().upper **else** Set{} **endif**

[5] If this operation has a single return result, type equals the value of type for that parameter. Otherwise type is not defined.

type = **if** returnResult->size() = 1 **then** returnResult->any().type **else** Set{} **endif**

[6] 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 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)
isConsistentWith = (redefinee.oclIsKindOf(Operation) and
                    let op: Operation = redefinee.oclAsType(Operation) in
                    self.formalParameter.size() = op.formalParameter.size() and
                    self.returnResult.size() = op.returnResult.size() and
                    forAll(i | op.formalParameter[i].type.conformsTo(self.formalParameter[i].type)) and
                    forAll(i | op.returnResult[i].type.conformsTo(self.returnResult[i].type))
                    )
```

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 is 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.

Notation

An operation is shown as a text string of the form:

visibility name (parameter-list) : property-string

- Where *visibility* is the operation's visibility -- *visibility* may be suppressed.
- Where *name* is the operation's name.
- Where *parameter-list* is a comma-separated list of formal parameters, each specified using the syntax:
  direction name : type-expression [multiplicity] = default-value [{ property-string }]
- Where *direction* is the parameter's direction, with the default of *in* if absent.
- Where *name* is the parameter's name.
- Where *type-expression* identifies the type of the parameter.
- Where *multiplicity* is the parameter's multiplicity in square brackets -- *multiplicity* may be suppressed in which case [1] is assumed.
- Where *default-value* is a value specification for the default value of the parameter. The default value is optional (the equal sign is also omitted if the default value is omitted).
- Where property-string indicates property values that apply to the parameter. The property string is optional (the braces are omitted if no properties are specified).
- Where *property-string* optionally shows other properties of the operation enclosed in braces.

Presentation Options

The parameter list can be suppressed.

Style Guidelines

An operation name typically begins with a lowercase letter.

Examples

display ()

-hide ()

+createWindow (location: Coordinates, container: Container [0..1]): Window

+toString (): String {query}

11.7.3 Parameter (as specialized)

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.

Attributes

- default: String [0..1] — Specifies a String that represents a value to be used when no argument is supplied for the Parameter.

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 Packages diagram

The Packages diagram of the Constructs package specifies the Package and PackageMerge constructs.

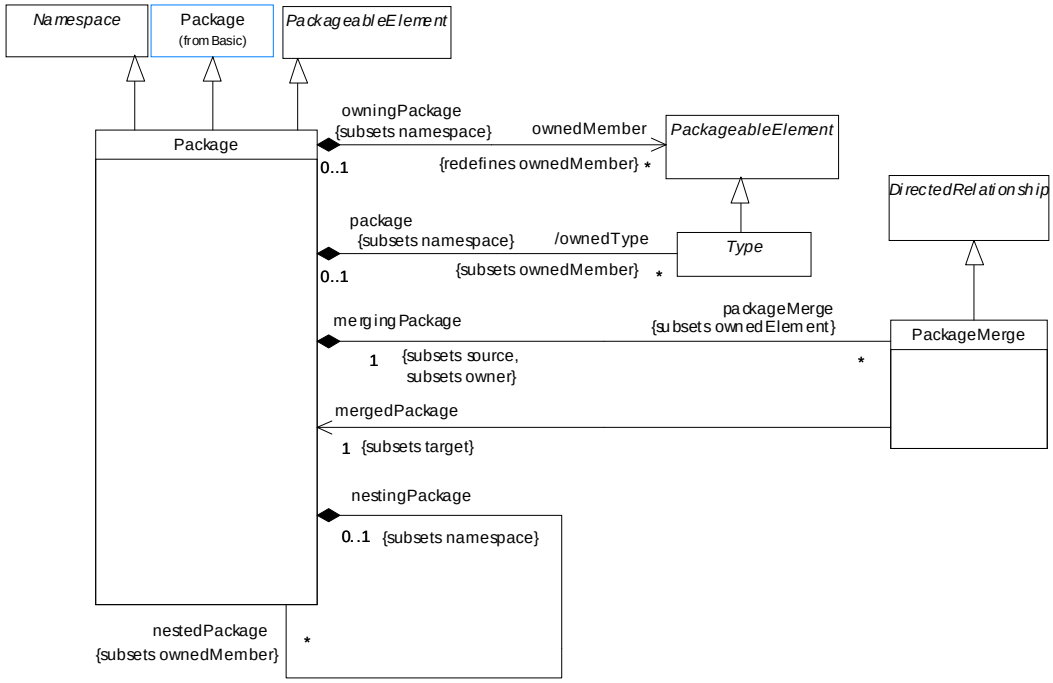

Figure 94 - The Packages diagram of the Constructs package

11.8.1 Type (additional properties)

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.

Attributes

No additional attributes.

Associations

- package: Package [0..1] Specifies the owning package of this classifier, if any. Subsets *NamedElement::namespace* and redefines *Basic::Type::package.*

Constraints

No additional constraints.

Semantics

No additional semantics.

11.8.2 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.

Attributes

No additional attributes.

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.visbility = #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

[1] 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))

[2] 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 = el.visibility->isEmpty() **or** el.visibility = #public

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 is always accessible outside the package through the use of qualified names.

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).

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 a 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 95. 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.

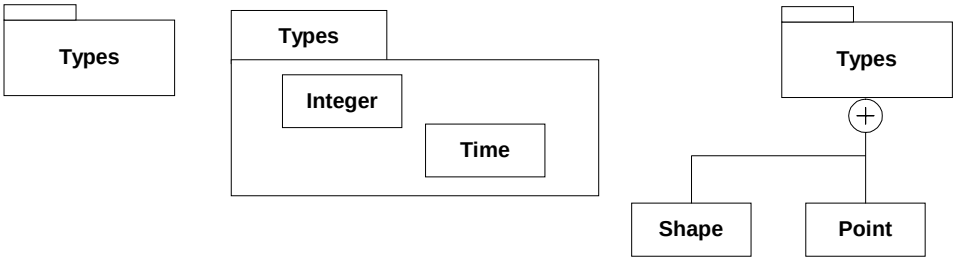

Figure 95 - Examples of a package with members

11.8.3 PackageMerge

A package merge defines how one package extends another package by merging their contents.

Description

A package merge is a relationship between two packages, where the contents of the target package (the one pointed at) is merged with the contents of the source package through specialization and redefinition, where applicable.

This is a mechanism that should be used when elements of the same name are intended to represent the same concept, regardless of the package in which they are defined. A merging package will take elements of the same kind with the same name from one or more packages and merge them together into a single element using generalization and redefinitions.

It should be noted that a package merge can be viewed as a short-hand way of explicitly defining those generalizations and redefinitions. The merged packages are still available, and the elements in those packages can be separately qualified.

From an XMI point of view, it is either possible to exchange a model with all PackageMerges retained or a model where all PackageMerges have been transformed away (in which case package imports, generalizations, and redefinitions are used instead).

Attributes

No additional attributes.

Associations

- mergedPackage: Package [1] References the Package that is to be merged with the source of the PackageMerge. Subsets *DirectedRelationship::target*.
- mergingPackage: Package [1] References the Package that is being extended with the contents of the target 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, where the contents of the merged package is expanded in the merging package. Each element has its own specific expansion rules. The package merge is transformed to a package import having the same source and target packages as the package merge.

An element with private visibility in the merged package is not expanded in the merging package. This applies recursively to all owned elements of the merged package.

A classifier from the target (merged) package is transformed into a classifier with the same name in the source (merging) package, unless the source package already contains a classifier of the same kind with the same name. In the former case, the new classifier gets a generalization to the classifier from the target package. In the latter case, the already existing classifier gets a generalization to the classifier from the target package. In either case, every feature of the general classifier is redefined in the specific classifier in such a way that all types refer to the transformed classifiers. In addition, the classifier in the source package gets generalizations to each transformed superclassifier of the classifier from the target package. This is because the superclassifiers may have merged in additional properties in the source package that need to be propagated properly to the classifier. Classifiers of the same kind with the same name from multiple target packages are transformed into a single classifier in the source package, with generalizations to each target classifier. Nested classifiers are recursively transformed the same way. If features from multiple classifiers are somehow conflicting, the same rules that apply for multiple inheritance are used to resolve conflicts.

Note that having an explicit generalization from a classifier in a source package to a classifier of the same kind with the same name in a target package is redundant, since it will be created as part of the transformation.

A subpackage from the target (merged) package is transformed into a subpackage with the same name in the source (merging) package, unless the source package already contains a subpackage with the same name. In the former case, the new subpackage gets a package merge to the subpackage from the target package. In the latter case, the already existing package gets a package merge to the subpackage from the target package. Subpackages with the same name from multiple target packages are transformed into a single subpackage in the source package, with package merges to each target subpackage. Nested subpackages are recursively transformed the same way.

A package import owned by the target package is transformed into a corresponding new package import in the source package. Elements from imported packages are not merged (unless there is also a package merge to the imported package). The names of merged elements take precedence over the names of imported elements, meaning that names of imported elements are hidden in case of name conflicts and need to be referred to using qualifiers. An element import owned by the target package is transformed into a corresponding new element import in the source package. Imported elements are not merged (unless there is also a package merge to the package owning the imported element or its alias).

A non-generalizable packageable element owned by the target package is copied down to the source package. Any classifiers referenced as part of the packageable element are redirected at transformed classifiers, if any.

Notation

A PackageMerge is shown using a dashed line with a stick arrowhead pointing from the merging package (the source) to the merged package (the target). In addition, the keyword «merge» is shown near the dashed line.

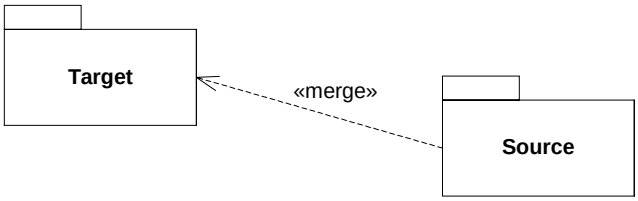

Figure 96 - Notation for package merge

Examples

In Figure 97, packages P and Q are being merged by package R, while package S merges only package Q.

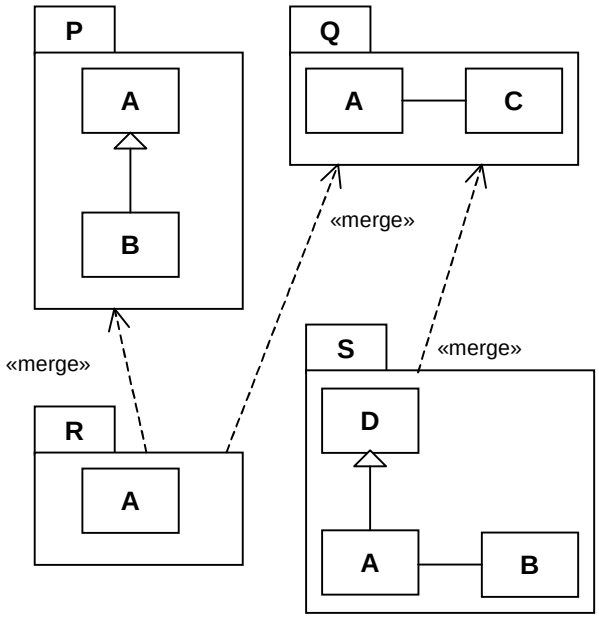

Figure 97 - Simple example of package merges

The transformed packages R and Q are shown in Figure 98. While not shown, the package merges have been transformed into package imports..

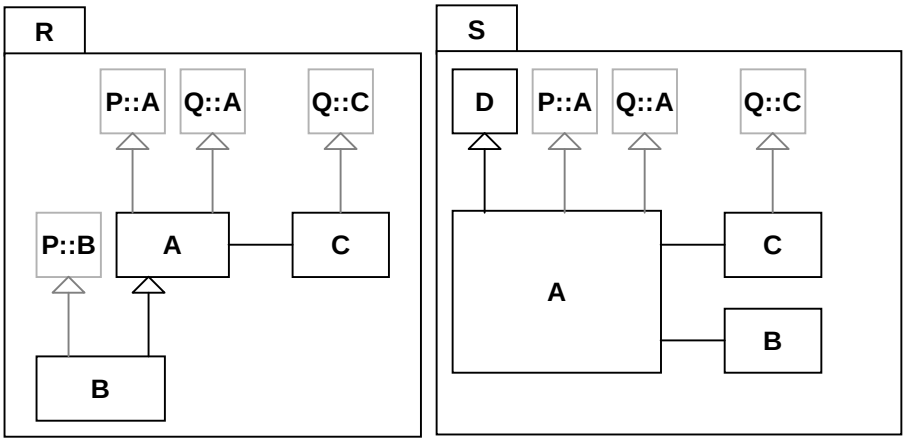

Figure 98 - Simple example of transformed packages

In Figure 99, additional package merges are introduced by having the package T merge the packages R and S that were previously defined. Aside from the package merges, the package T is completely empty.

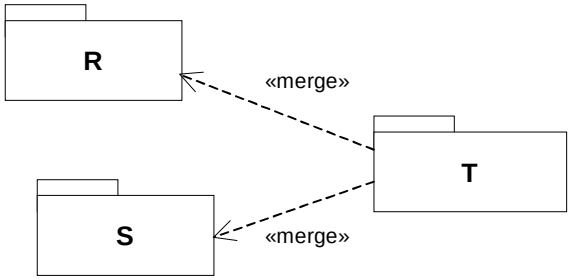

Figure 99 - Introducing additional package merges

In Figure 100, the transformed version of the package T is depicted. In this package, the partial definitions of A, B, C, and D have all been brought together. Again, the package merges have been transformed to package imports. 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 types in package T.

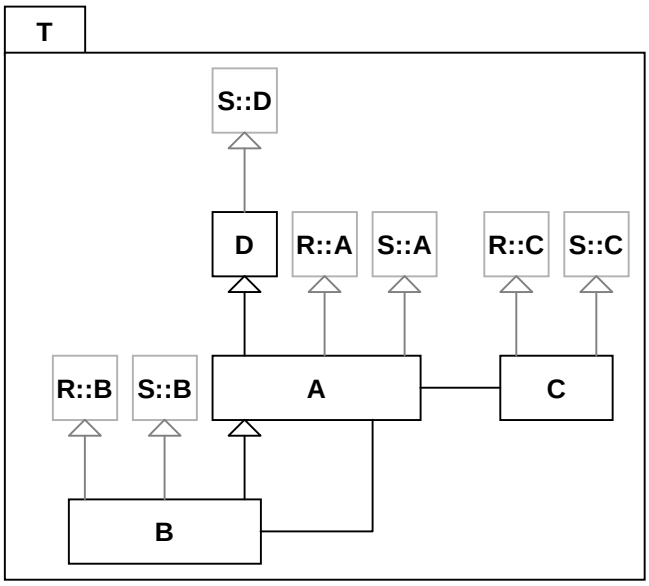

Figure 100 - The result of the additional package merges

It is possible to elide all but the most specific of each classifier, which gives a clearer picture of the end result of the package merge transformations, as is shown in Figure 101.

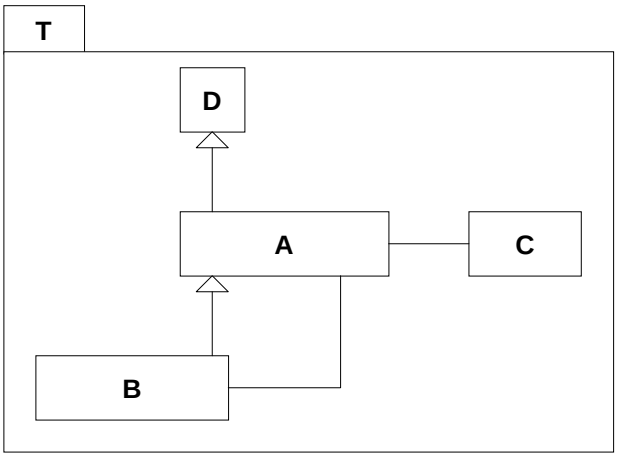

Figure 101 - The result of the additional package merges: elided view

12 Core::PrimitiveTypes

The PrimitiveTypes package of InfrastructureLibrary::Core contains a number of predefined types used when defining the abstract syntax of metamodels.

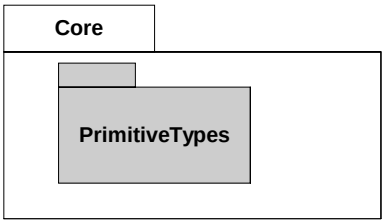

Figure 102 - The Core package is owned by the InfrastructureLibrary package, and contains several subpackages

12.1 PrimitiveTypes package

The PrimitiveTypes subpackage within the Core package defines the different types of primitive values that are used to define the Core metamodel. It is also intended that every metamodel based on Core will reuse the following primitive types.

In Core and the UML metamodel, these primitive types are predefined and available to the Core and UML extensions at all time. These predefined value types are independent of any object model and part of the definition of the Core.

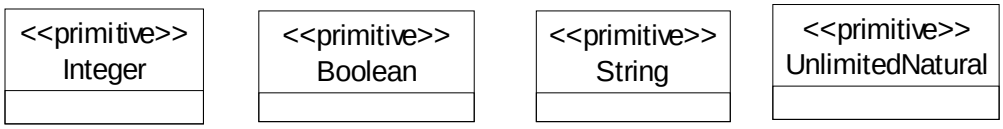

Figure 103 - The classes defined in the PrimitiveTypes package

12.1.1 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

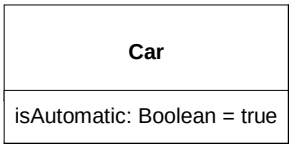

Figure 104 - An example of a boolean attribute

12.1.2 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

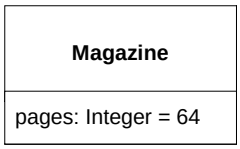

Figure 105 - An example of an integer attribute

12.1.3 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

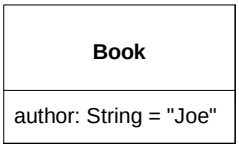

Figure 106 - An example of a string attribute

12.1.4 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

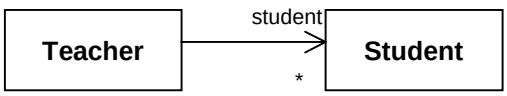

Figure 107 - An example of an unlimited natural

13 Core::Profiles

The Profiles package of the InfrastructureLibrary 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).

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 adaption 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.

13.1 Profiles package

The Profiles package is dependent on the *Constructs* package from *Core*, as is depicted in Figure 108.

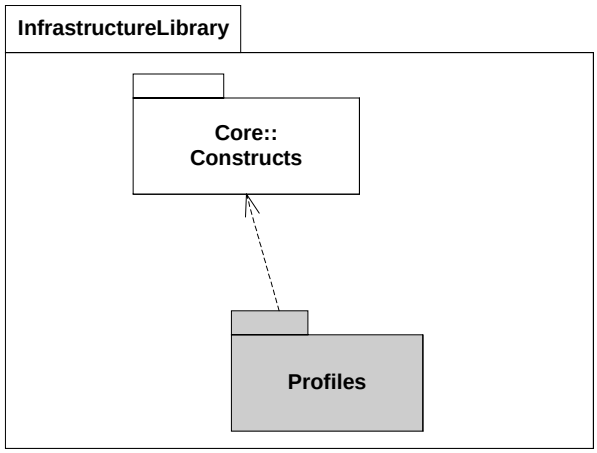

Figure 108 - The Profiles package is owned by the InfrastructureLibrary package

The classes of the Profiles package are depicted in Figure 109, and subsequently specified textually.

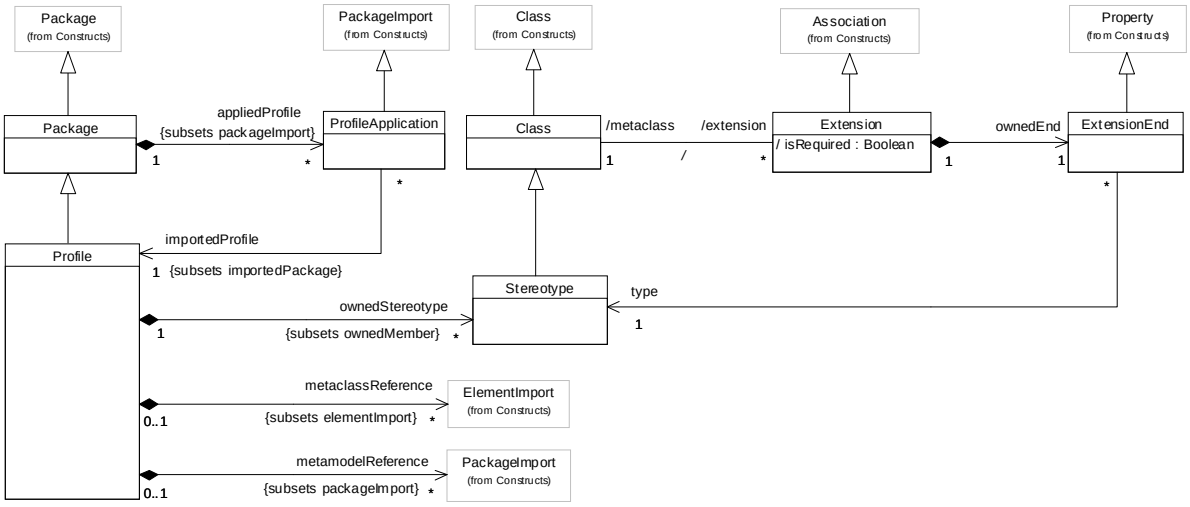

Figure 109 - The classes defined in the Profiles package

13.1.1 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.

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

- package : Package [0..1] — The containing package.
- / 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 multiplicity of *Extension::ownedEnd*; a multiplicity of 1 means that isRequired is *true*, but otherwise it is *false*. Since the default multiplicity of an ExtensionEnd is 0..1, 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.

```
self.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.

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.

In order to be able to navigate to the extended metaclass when writing constraints, the end must have a name. If no name is given, the default name is *baseClass*.

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 110 - The notation for an Extension

Presentation Option

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 111, a simple example of using an extension is shown, where the stereotype *Home* extends the metaclass *Interface*.

Figure 111 - 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 112, 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 for a way to express extensions that should always be present for all instances of the base metaclass depending on which profiles are applied. .

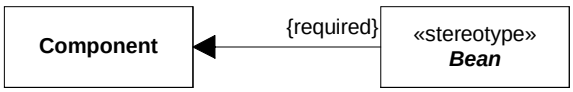

Figure 112 - An example of a required Extension

Changes from UML 1.4

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.

13.1.2 ExtensionEnd (from Profiles)

An extension end is used to tie an extension to a stereotype when extending a metaclass.

Description

ExtensionEnd is a kind of Property that is always typed by a Stereotype.

An ExtensionEnd is never navigable. If it was navigable, it would be a property of the extended classifier. Since a profile is not allowed to changed the referenced metamodel, it is not possible to add properties to the extended classifier. As a consequence, an ExtensionEnd can only be owned by an Extension.

The aggregation of an ExtensionEnd is always composite.

The default multiplicity of an ExtensionEnd is 0..1.

Attributes

No additional attributes.

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 *Property::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 was 1.

```
ExtensionEnd::lowerBound() : [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 165.

Changes from UML 1.4

ExtensionEnd did not exist as a metaclass in UML 1.4. See “Extension (from Profiles)” on page 165 for further details.

13.1.3 Class (from Constructs, Profiles)

Description

Class has derived association that indicates how it may be extended through one or more stereotypes.

Because a stereotype is a class, it is possible to apply a stereotype not only to classes, but also to definitions of 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 Option

A Class that is extended by a Stereotype may have the optional keyword «metaclass» shown above or before its name.

Examples

In Figure 113, an example is given where it is made explicit that the extended class *Interface* is in fact a metaclass (from a reference metamodel).

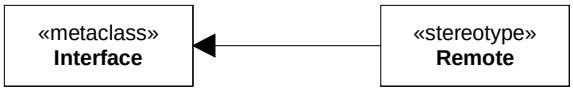

Figure 113 - Showing that the extended class is a metaclass

Changes from UML 1.4

A link typed by UML 1.4 ModelElement::stereotype is mapped to a link that is typed by Class::extension.

13.1.4 Package (from Constructs, Profiles)

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

- appliedProfile: ProfileApplication [*]References the ProfileApplications that indicate which profiles have been applied to the Package. Subsets *Package::packageImport*.

Constraints

No additional constraints.

Semantics

No additional semantics.

Notation

No additional notation.

Changes from UML 1.4

In UML 1.4, it was not possible to indicate which profiles were applied to a package.

13.1.5 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.

Description

A Profile is a kind of Package that extends a reference metamodel. The primary extension construct is the Stereotype, which are 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*.
- ownedStereotype: Stereotype [*] References the Stereotypes that are owned by the Profile. Subsets *Package::owned-Member.*

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 or another profile. 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. A tool can make use of the information about which metaclasses are extended in different ways, for example to filter or hide elements when a profile is applied, or to provide specific tool bars that apply to certain profiles. However, elements of a package or model cannot be deleted simply through the application of a profile. Each profile thus provides a simple viewing mechanism.

As part of a profile, it is not possible to have an association between two stereotypes or between a stereotype and a metaclass unless they are subsets of existing associations in the reference metamodel. However, it is possible to have associations between ordinary classes, and from stereotypes to ordinary classes. Likewise, properties of stereotypes may not be typed by metaclasses or stereotypes.

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 114, a simple example of an EJB profile is shown.

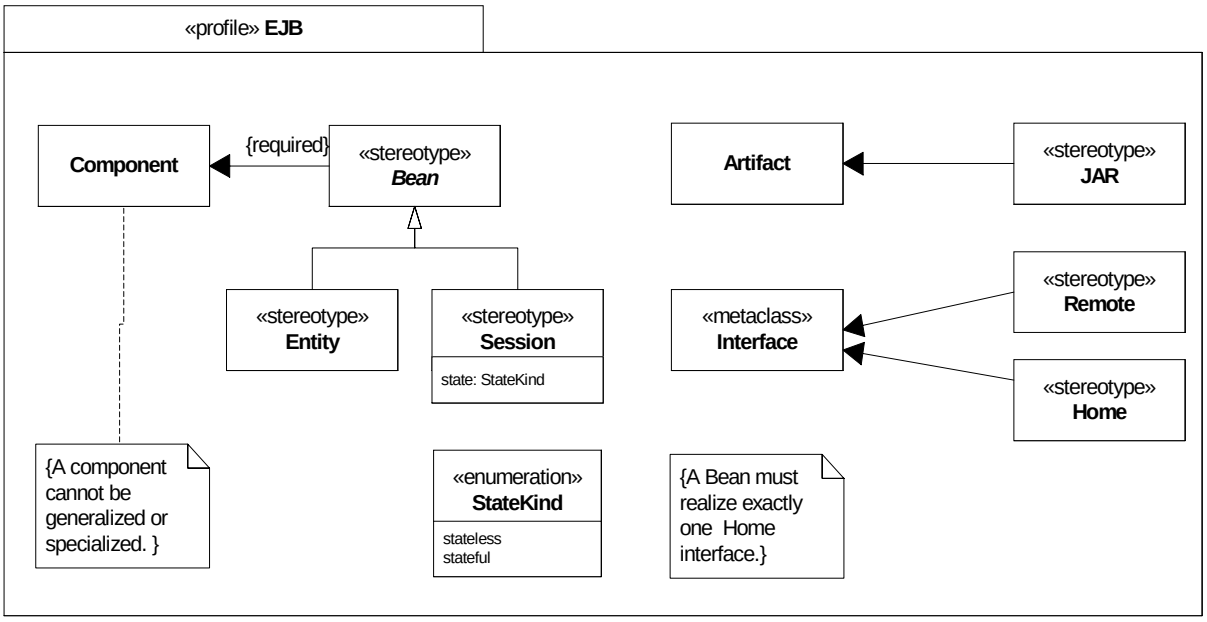

Figure 114 - Defining a simple EJB profile

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 need to be satisfied in order for the model to be well formed.

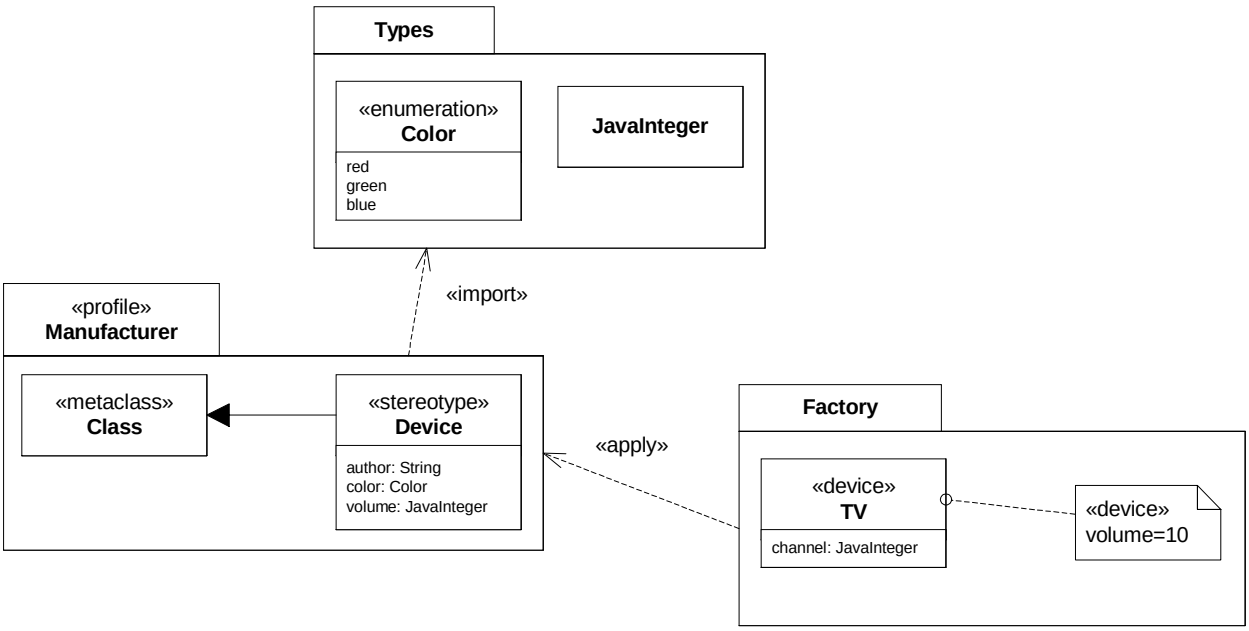

Figure 115 - Importing a package from a profile

In Figure 115, 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 also available for use in the package to which the profile is applied (since profile application is a kind of import). This means that for example the class *JavaInteger* can be used as both a metaproperty (as part of the stereotype *Device*) and an ordinary property (as part of the class *TV*). Note how the metaproperty is given a value when the stereotype *Device* is applied to the class *TV.*

13.1.6 ProfileApplication (from Profiles)

A profile application is used to show which profiles have been applied to a package.

Description

ProfileApplication is a kind of PackageImport that adds the capability to state that a Profile is applied to a Package.

Attributes

No additional attributes.

Associations

- importedProfile: Profile [1] References the Profiles that is applied to a Package through this ProfileApplication.. Subsets *PackageImport::importedPackage*.

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. If a profile that is being applied depends on other profiles, then those profiles must be applied first.

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.

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 stick 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 116 shows how these have been applied to the package *WebShopping*.

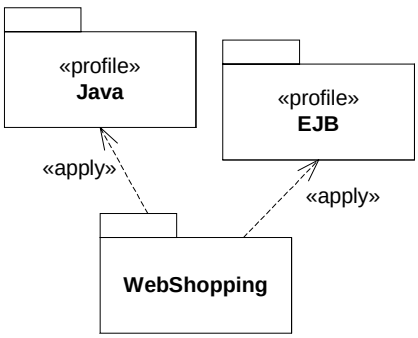

Figure 116 - Profiles applied to a package

13.1.7 Stereotype (from Profiles)

A stereotype defines how an existing metaclass (or stereotype) may be extended, and enables the use of platform or domain specific terminology or notation in addition to the ones used for the extended metaclass.

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

No additional associations.

Constraints

[1] A Stereotype may only generalize or specialize another Stereotype.

```
self.generalization.general->forAll(e | e.oclIsKindOf(Stereotype)) and
self.specialization.specific->forAll(e | e.oclIsKindOf(Stereotype))
```

[2] A concrete Stereotype must directly or indirectly extend a Class.

```
not self.isAbstract implies self.extensionEnd->union(self.parents.extensionEnd)->notEmpty()
```

Semantics

A stereotype is 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 of a stereotype is linked to an instance of an extended metaclass (or stereotype) by virtue of the extension between their types.

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 Stereotype. If multiple stereotypes are applied, the names of the applied stereotypes is shown as a comma-separated list with a pair of guillemets.

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 list them after each other.

The values of a stereotype that has been applied to a model element can be shown as part of a comment symbol tied to the model element. The values from a specific stereotype are optionally preceded with the name of the applied stereotype within a pair of guillemets, which is useful if values of more than one applied stereotype should be shown.

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.

Style Guidelines

The first letter of an applied stereotype should not be capitalized. The values of an applied stereotype are normally not shown.

Examples

In Figure 117, a simple stereotype *Clock* is defined to be applicable at will (dynamically) to instances of the metaclass Class.

Figure 117 - Defining a stereotype

Note that in order to be able to write constraints on the stereotype Clock that should be applied to the metaclass Class or any of its relationships, it is necessary to give the end typed by the metaclass a name for navigation purposes. A typical such name would be for example *base*.

In Figure 118, an instance specification of the example in Figure 117 is shown. Note that the extension must be composite, and that the the derived required attribute in this case is false.

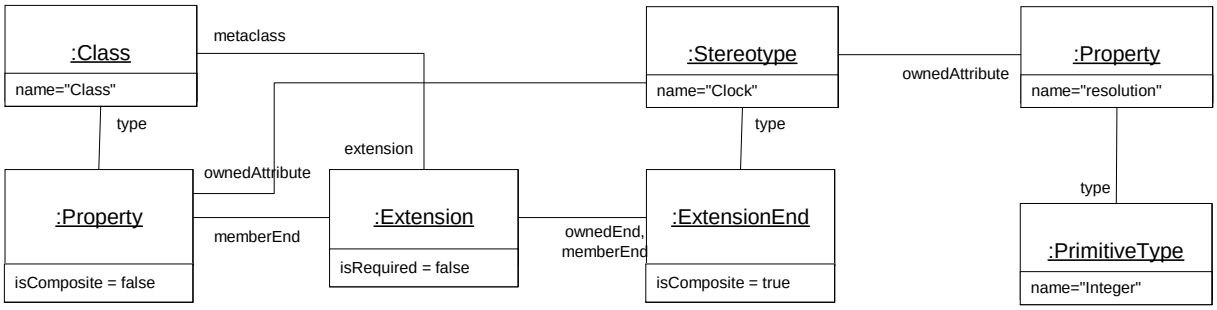

Figure 118 - An instance specification when defining a stereotype

In Figure 119, it is shown how the same stereotype *Clock* can extend either the metaclass Component or the metaclass Class. It is also shown how different stereotypes can extend the same metaclass.

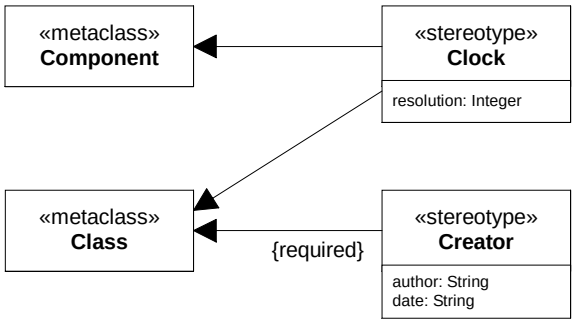

Figure 119 - Defining multiple stereotypes on multiple stereotypes

Figure 120 shows how the stereotype *Clock*, as defined in Figure 119, is applied to a class called *StopWatch*.

Figure 120 - Using a stereotype

Figure 121 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*.

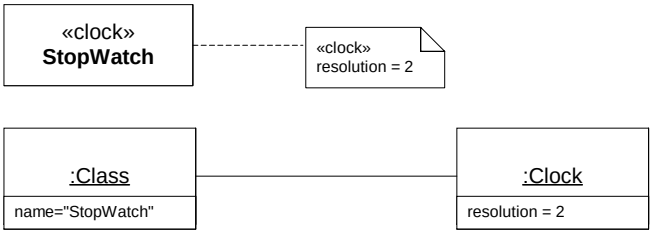

Figure 121 - Showing values of stereotypes and a simple instance specificiation

Next, two stereotypes, *Clock* and *Creator*, are applied to the same model element, as is shown in Figure 122. Note that the attribute values of each of the applied stereotypes can be shown in a comment symbol attached to the model element.

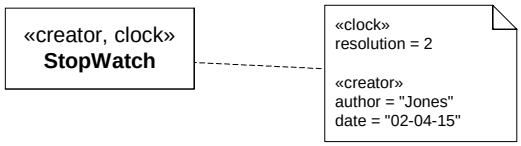

Figure 122 - Using stereotypes and showing values

Part III - Appendices

A XMI Serialization and Schema

UML 2.0 models are serialized in XMI according to the rules specified by the *MOF 2.0: XMI Mapping* Specification. The XML schema for MOF 2.0 models that support the *MOF 2.0: XMI Mappin*g specification is available in OMG document ad/2002-12-09.

The XMI for serializing the *UML 2.0: Infrastructure* as an instance of MOF 2.0 according to the rules specified by the *MOF 2.0: XMI Mapping* Specification is available in OMG document ad/2003-04-04. It is expected that the normative XMI for this specification will be generated by a Finalization Task Force, which will architecturally align and finalize the relevant specifications.

B Support for Model Driven Architecture

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 set of cohesive set of model-driven technology specifications.

The MDA initiative was initiated relatively recently, after the UML 2.0 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 proposed major revision to UML will play an important role in furthering the goals of MDA.

At the time of this writing, there appears to be no official, nor commonly agreed upon definition of MDA or its requirements. However, the OMG provides an executive summary for the initiative, along with a collection of white papers and presentations at www.omg.org/mda. In addition, the OMG Object Reference Model Subcommittee has produced a "Text for an MDA Guide" draft (ormsc/02-10-01) that is intended to be used in a future MDA Guide. This MDA Guide draft characterizes MDA as follows:

> *"MDA provides an approach and tools for:*
> *- specifying a system independently of the platform that supports it,*
> *- specifying platforms,*
> *- choosing a particular platform for the system, and*
> *- transforming the system specification into one for a 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 draft as: "Extensions to the UML language [that] will be standardized for specific purposes. Many of these will be designed specifically for use in MDA."

Given the nascent and evolving state of MDA, and the lack of common and precise definitions and requirements, it is problematic to show strict architectural alignment with it. However, the following sections explain how UML 2.0 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 2.0 proposal 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 proposed InfrastructureLibrary is intended to be reused by MOF 2.0. Tools that implement MOF 2.0 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 2.0 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 2.0 that will make it easier for modelers to specify Platform Independent Modelers 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 proposal 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 Superstructure part of this proposal.) 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 Superstructure part of this proposal 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.

Index

Index

Index

Index