



# OMG Unified Modeling Language™ (OMG UML)

*Version 2.5 FTF – Beta 1*

---

OMG Document Number: ptc/2012-10-24

Normative Reference: <http://www.omg.org/spec/UML/2.5>

Machine Consumable Files:

<http://www.omg.org/spec/UML/20120801/PrimitiveTypes.xmi>

<http://www.omg.org/spec/UML/20120801/UML.xmi>

<http://www.omg.org/spec/UML/20120801/StandardProfile.xmi>

<http://www.omg.org/spec/UML/20120801/UMLDI.xmi>

---

Version 2.5 is a minor revision to the UML 2.4.1 specification. It supersedes formal/2011-08-05 (Infrastructure) and formal/2011-08-06 (Superstructure).

This OMG document replaces the submission document (ad/2012-08-01, Alpha). It is an OMG Adopted Beta Specification and is currently in the finalization phase.

Comments on the content of this document are welcome, and should be directed to [issues@omg.org](mailto:issues@omg.org) by June 10, 2013. You may view the pending issues for this specification from the OMG revision issues web page <http://www.omg.org/issues/>.

The FTF Recommendation and Report for this specification will be published on October 4, 2013. If you are reading this after that date, please download the available specification from the OMG Specifications Catalog.

Copyright © 2009-2012 88Solutions  
Copyright © 2009-2010 Artisan Software Tools  
Copyright © 2001-2012 Adaptive  
Copyright © 2009-2010 Armstrong Process Group, Inc.  
Copyright © 2001-2010 Alcatel  
Copyright © 2001-2010 Borland Software Corporation  
Copyright © 2009-2010 Commissariat à l'Energie Atomique  
Copyright © 2001-2010 Computer Associates International, Inc.  
Copyright © 2009-2010 Computer Sciences Corporation  
Copyright © 2009-2012 Deere & Company  
Copyright © 2009-2010 European Aeronautic Defence and Space Company  
Copyright © 2001-2012 Fujitsu  
Copyright © 2001-2010 Hewlett-Packard Company  
Copyright © 2001-2010 I-Logix Inc.  
Copyright © 2001-2012 International Business Machines Corporation  
Copyright © 2001-2010 IONA Technologies  
Copyright © 2001-2010 Kabira Technologies, Inc.  
Copyright © 2009-2010 Lockheed Martin  
Copyright © 2001-2010 MEGA International  
Copyright © 2009-2010 Mentor Graphics Corporation  
Copyright © 2009-2012 Microsoft Corporation  
Copyright © 2009-2012 Model Driven Solutions  
Copyright © 2001-2010 Motorola, Inc.  
Copyright © 2009-2010 National Aeronautics and Space Administration  
Copyright © 2009-2012 No Magic, Inc.  
Copyright © 2009-2010 oose Innovative Informatik GmbH  
Copyright © 2001-2010 Oracle Corporation  
Copyright © 2009-2010 Oslo Software, Inc.  
Copyright © 2009-2010 Perdue University  
Copyright © 2012 Simula Research Laboratory  
Copyright © 2009-2010 SINTEF  
Copyright © 2001-2010 SOFTEAM  
Copyright © 2009-2012 Sparx Systems Pty Ltd  
Copyright © 2001-2010 Telefonaktiebolaget LM Ericsson  
Copyright © 2009-2010 THALES  
Copyright © 2001-2012 Unisys  
Copyright © 2001-2010 X-Change Technologies Group, LLC

#### 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, 109 Highland Avenue, Needham, MA 02494, U.S.A.

#### TRADEMARKS

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® and XMI® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™, Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™, MOF™, IMM™, OMG Interface Definition Language (IDL™), and OMG Systems Modeling Language

(OMG SysML™) are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

#### COMPLIANCE

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

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

## **OMG's Issue Reporting Procedure**

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

# Table of Contents

1	Scope.....	1
2	Conformance.....	1
3	Normative References.....	2
4	Terms and Definitions.....	2
5	Notational Conventions .....	2
5.1	Key words for Requirement Statements .....	2
5.2	Annotations on Example Diagrams .....	3
6	Additional Information .....	3
6.1	Specification Simplification.....	3
6.2	Architectural Alignment .....	4
6.3	On the Semantics of UML .....	4
6.4	How to Read this Specification.....	7
6.5	Acknowledgements.....	10
7	Common Structure .....	11
7.1	Summary .....	11
7.2	Root.....	11
7.3	Templates.....	13
7.4	Namespaces.....	17
7.5	Types and Multiplicity.....	22
7.6	Constraints .....	25
7.7	Dependencies .....	28
7.8	Classifier Descriptions .....	31
7.9	Association Descriptions .....	53
8	Values .....	63
8.1	Summary .....	63
8.2	Literals .....	63
8.3	Expressions .....	65
8.4	Time .....	67
8.5	Intervals.....	70
8.6	Classifier Descriptions .....	74
8.7	Association Descriptions .....	89
9	Classification.....	95
9.1	Summary .....	95
9.2	Classifiers.....	95
9.3	Classifier Templates.....	100
9.4	Features .....	104
9.5	Properties .....	108
9.6	Operations .....	114
9.7	Generalization Sets .....	118
9.8	Instances.....	127
9.9	Classifier Descriptions .....	131
9.10	Association Descriptions .....	159

10	Simple Classifiers .....	172
10.1	Summary .....	172
10.2	DataTypes .....	172
10.3	Signals.....	174
10.4	Interfaces.....	176
10.5	Classifier Descriptions .....	180
10.6	Association Descriptions .....	185
11	Structured Classifiers .....	190
11.1	Summary .....	190
11.2	Structured Classifiers .....	190
11.3	Encapsulated Classifiers .....	197
11.4	Classes.....	202
11.5	Associations .....	208
11.6	Components .....	217
11.7	Collaborations .....	224
11.8	Classifier Descriptions .....	229
11.9	Association Descriptions .....	244
12	Packages.....	254
12.1	Summary .....	254
12.2	Packages.....	254
12.3	Profiles .....	266
12.4	Classifier Descriptions .....	282
12.5	Association Descriptions .....	291
13	Common Behavior .....	295
13.1	Summary .....	295
13.2	Behaviors .....	295
13.3	Events.....	300
13.4	Classifier Descriptions .....	305
13.5	Association Descriptions .....	312
14	StateMachines .....	316
14.1	Summary .....	316
14.2	Behavior StateMachines .....	316
14.3	StateMachine Redefinition.....	347
14.4	ProtocolStateMachines .....	352
14.5	Classifier Descriptions .....	358
14.6	Association Descriptions .....	379
15	Activities.....	388
15.1	Summary .....	388
15.2	Activities .....	388
15.3	Control Nodes .....	403
15.4	Object Nodes.....	413
15.5	Executable Nodes.....	420
15.6	Activity Groups.....	423
15.7	Classifier Descriptions .....	431

15.8	Association Descriptions .....	452
16	Actions .....	461
16.1	Summary .....	461
16.2	Actions .....	462
16.3	Invocation Actions .....	470
16.4	Object Actions .....	479
16.5	Link End Data .....	482
16.6	Link Actions.....	484
16.7	Link Object Actions .....	486
16.8	Structural Feature Actions .....	488
16.9	Variable Actions .....	490
16.10	Accept Event Actions .....	493
16.11	Structured Actions .....	497
16.12	Expansion Regions.....	502
16.13	Other Actions .....	508
16.14	Classifier Descriptions .....	511
16.15	Association Descriptions .....	569
17	Interactions.....	598
17.1	Summary .....	598
17.2	Interactions.....	600
17.3	Lifelines .....	605
17.4	Messages .....	607
17.5	Occurrences.....	612
17.6	Fragments.....	614
17.7	Interaction Uses .....	625
17.8	Sequence Diagrams.....	630
17.9	Communication Diagrams .....	635
17.10	Interaction Overview Diagrams.....	637
17.11	Timing Diagrams .....	640
17.12	Classifier Descriptions .....	644
17.13	Association Descriptions .....	669
18	UseCases .....	679
18.1	Use Cases .....	679
18.2	Classifier Descriptions .....	688
18.3	Association Descriptions .....	691
19	Deployments .....	694
19.1	Summary .....	694
19.2	Deployments .....	694
19.3	Artifacts.....	698
19.4	Nodes .....	699
19.5	Classifier Descriptions .....	703
19.6	Association Descriptions .....	708
20	InformationFlows.....	712
20.1	Information Flows.....	712



20.2	Classifier Descriptions .....	716
20.3	Association Descriptions .....	718
21	Primitive Types .....	721
21.1	Summary .....	721
21.2	Semantics .....	721
21.3	Notation.....	721
21.4	Examples.....	721
22	Standard Profile .....	723
22.1	Summary .....	723
22.2	Model .....	723
22.3	Standard Stereotypes.....	724
Annex A:	Diagrams .....	727
Annex B:	UML Diagram Interchange .....	731
B.1	Summary .....	731
B.2	Generic .....	732
B.3	Structure .....	737
B.4	Behavior .....	742
B.5	Information Flows.....	748
B.6	Classifier Descriptions .....	749
B.7	Association Descriptions .....	768
Annex C:	Keywords.....	773
Annex D:	Tabular Notations .....	777
Examples.....		778
Annex E:	XMI Serialization and Schema.....	780
Index .....		781



# 1 Scope

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

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

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

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

# 2 Conformance

There are five distinct types of conformance. They are:

1. *Abstract syntax conformance.* A tool demonstrating abstract syntax conformance provides a user interface and/or API that enables instances of concrete UML metaclasses to be created, read, updated, and deleted. The tool must also provide a way to validate the well-formedness of models that corresponds to the constraints defined in the UML metamodel.
2. *Concrete syntax conformance.* A tool demonstrating concrete syntax conformance provides a user interface and/or API that enables instances of UML notation to be created, read, updated, and deleted. Note that a conforming tool may provide the ability to create, read, update and delete additional diagrams and notational elements that are not defined in UML.
3. *Model interchange conformance.* A tool demonstrating model interchange conformance can import and export conformant XMI for all valid UML models, including models with profiles defined and/or applied. Model interchange conformance implies abstract syntax conformance. A conforming UML 2.5 tool shall be able to load and save XMI in UML 2.4.1 format as well as UML 2.5 format.
4. *Diagram interchange conformance.* A tool demonstrating diagram interchange conformance can import and export conformant DI for all valid UML models with diagrams, including models with profiles defined and/or applied. Diagram interchange conformance implies both concrete syntax conformance and abstract syntax conformance.

5. *Semantic conformance.* A tool demonstrating semantic conformance provides a demonstrable way to interpret UML semantics, e.g., code generation, model execution, or semantic model analysis.

Where the UML specification provides options for a conforming tool, these are explicitly stated in the specification. In a number of other cases, certain aspects of the semantics are listed as "undefined", allowing for domain- or application-specific customizations. Only customizations that do not contradict the provisions of this specification will be deemed to conform to it. However, models whose meaning is based on such customizations can only be interchanged without loss with tools that support the same or compatible customizations.

It is recognized that some implementers and profile designers may want to support only a subset of features. Given this potential variability, it is useful to be able to specify clearly and efficiently which capabilities are supported by a given implementation. To this end, UML implementers and profile designers may provide feature support statements.

## 3 Normative References

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

- RFC2119, <http://ietf.org/rfc/rfc2119>, Key words for use in RFCs to Indicate Requirement Levels, S. Bradner, March 1997.
- OMG Specification formal/2012-01-01, Object Constraint Language (OCL), v2.3.1
- OMG Specification formal/2011-08-07, Meta Object Facility (MOF) Core, v2.4.1
- OMG Specification formal/2011-08-09, Meta Object Facility (MOF) 2.4 XMI Mapping Specification, v2.4.1
- OMG Specification ptc/2011-07-13, Diagram Definition 1.0

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

## 4 Terms and Definitions

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

## 5 Notational Conventions

### 5.1 Key words for Requirement Statements

The key words "must," "must not," "shall," "shall not," "should," "should not," and "may" in this specification are to be interpreted as described in RFC 2119.

## 5.2 Annotations on Example Diagrams

Some of the diagram examples in this specification include explanatory annotations, which should not be confused as being part of the formal UML graphical notation.

In these cases, the explanatory text originates outside the UML diagram boundary, and has an arrow pointing at the feature of the diagram which is being explained by the annotation. The color rendition of this spec shows these annotations in red.

# 6 Additional Information

## 6.1 Specification Simplification

This specification has been extensively re-written from its previous version to make it easier to read by removing redundancy and increasing clarity. In particular, the following major changes have been made since UML 2.4.1:

- The UML Infrastructure no longer forms part of the UML specification. The entire UML specification is constituted in this document.
- Package Merge is not used within the specification. Every metaclass is specified completely in one clause.
- The specification is organized to reduce forward references as much as possible. This means that topics such as Templates which are pervasive in their effects appear early in the specification.
- Every clause has a section of documentation generated from the metamodel that contains all of the metaclasses with their properties, and all of the metaassociations with their properties. All cross-references in this generated documentation include hyperlinks to their targets.
- The compliance levels L0, L1, L2, and L3 have been eliminated, because they were not found to be useful in practice. A tool either complies with the whole of UML or it does not. A tool may partially comply with UML by implementing a subset of its metamodel, notation, and semantics, in which case the vendor should declare which subset it implements.

However, the metamodel itself remains unchanged from UML 2.4.1 superstructure, with a few exceptions:

- The metamodel has been partitioned into packages, corresponding to the clause structure of this specification. All of these packages are owned by a top-level package named UML; they are also imported into UML so that metaclasses may be referred to by their unqualified name in UML.
- Many OCL constraints have been corrected or added where they were absent. In order to do this, some names of association-owned properties and the corresponding associations have been changed in order to avoid ambiguity in OCL expressions.
- A small number of lower multiplicities have been relaxed from 1 to 0, in order to represent default values that cannot be formally represented using MOF. In these cases the absence of a value signifies the presence of a default value. These cases could not be represented at all in earlier versions of UML. They all occur in [Clause 15](#): Activities and are made explicit in the text there.
- The property `LoopNode::loopVariable` has been made composite, in order to enable interchange of loop variables, which was not possible in a standard way in UML 2.4.1.
- `{ordered}` has been added to some properties in order to make the semantics consistent.

## 6.2 Architectural Alignment

The OMG's Model Driven Architecture (MDA) initiative is a conceptual architecture for a set of industry-wide technology specifications that 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. UML, MOF, and related specifications play important roles in MDA by providing the languages for creating and transforming models.

In version 2.4.1, the MOF Core specification was defined to use the UML 2.4.1 Superstructure specification directly, albeit with constraints: hence a MOF 2.4.1 metamodel, including the UML 2.4.1 metamodel, is a valid UML 2.4.1 model. This was a substantial simplification and alignment compared to earlier versions. It is expected that future versions of MOF and UML will continue to be aligned in this manner.

## 6.3 On the Semantics of UML

### 6.3.1 Models and What They Model

A model is always a model *of* something. The thing being modeled can generically be considered a *system* within some *domain* of discourse. The model then makes some statements of interest about that system, abstracting from all the details of the system that could possibly be described, from a certain point of view and for a certain purpose. For an existing system, the model may represent an analysis of the properties and behavior of the system. For a planned system, the model may represent a specification of how the system is to be constructed and behave.

A UML model consists of three major categories of model elements, each of which may be used to make statements about different kinds of *individuals* within an incarnation of the system being modeled. These categories are:

- *Classifiers*. A classifier describes a set of objects. An *object* is an individual thing with a state and relationships to other objects.
- *Events*. An event describes a set of possible occurrences. An *occurrence* is something that happens that has some consequence within the system.
- *Behaviors*. A behavior describes a set of possible executions. An *execution* is a performance of a set of actions (potentially over some period of time) that may generate and respond to occurrences of events, including accessing and changing the state of objects.

UML models do *not* contain objects, occurrences, or executions, because such individuals are part of the domain being modeled, not the content of the models themselves. UML does have modeling constructs for directly modeling individuals: value specifications, occurrence specifications, and execution specifications for modeling objects, occurrences, and executions, respectively, within a particular context. However, these are again just model elements, making statements about the individuals being modeled. As for any model, such statements can be incomplete, imprecise, and abstract, according to the purpose of the model, and may turn out to be wrong (or even be asserted as counterfactual). The individuals being modeled, on the other hand, are always complete, precise, and concrete within their domain.

### 6.3.2 Semantic Areas

Clause 2 makes the distinction of the conformance of a tool to the (concrete and abstract) *syntax* of UML from conformance to its *semantics*.

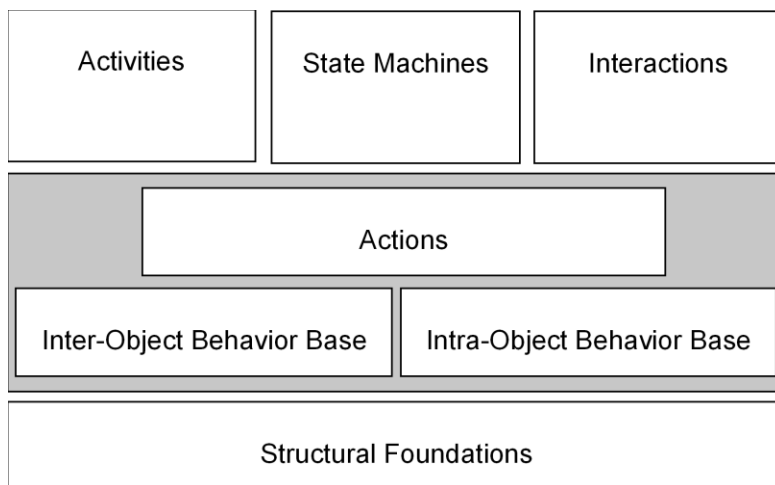
The syntax of UML has to do with how UML models may be constructed, represented and interchanged. The UML specification defines the syntax of UML, but does not specify the semantics of tools that conform to that syntax. Instead, those semantics are given in the relevant MOF Core, XMI and Diagram Interchange specifications, within whose framework the UML syntax is specified.

In contrast, the semantics of UML itself have to do with the standard meaning of the statements made by a UML model about the system being modeled. This is sometimes referred to as the “run-time” semantics of UML, especially in the context of UML models of executable software or other enactable processes. However, not all UML models are executable in this sense and not all UML semantics relate to “running” software or other processes.

Instead, consider the general division of UML modeling constructs into two semantic categories:

- *Structural Semantics* defines the meaning of UML *structural* model elements about individuals in the domain being modeled, which may be true at some specific point in time. (Note that this category is sometimes called “static semantics”. However, in programming language definition, the term “static semantics” is generally used to mean context-sensitive name resolution and type constraints beyond the base context-free syntax of the language, which corresponds to well-formedness constraints in the UML abstract syntax specification. In order to avoid confusion, the term “structural semantics” is used here instead.)
- *Behavioral Semantics* defines the meaning of UML *behavioral* model elements that make statements about how individuals in the domain being modeled change over time. (This is sometimes also called “dynamic semantics.”)

Figure 6.1 shows a more detailed delineation of the semantic areas of UML within these categories and the notional layering of these areas.



**Figure 6.1 Semantic Areas of UML**

The structural semantics of the modeling elements specified in Clause 7 through 12 provide the foundation for the behavioral semantics of UML. This reflects the conception of behavioral semantics in terms of changes in the system state specified through structural modeling. Structural modeling constructs in UML include a number of different kinds of classifiers (e.g., DataTypes, Classes, Signals, Interfaces and Components) and corresponding ValueSpecifications.

The base behavioral semantics of UML builds on this structural foundation, addressing both the basic framework of the execution of behaviors and such execution may result in communication between structural objects with associated behavior (see Clause 13). Note that this framework only deals with *event-driven*, or discrete, behaviors. However, UML semantics do not dictate the amount of time between events (unless this is specifically modeled using timing constraints, see sub clause 8.5). Thus, the intervals between certain events can be considered to be as small as needed by the application; for example, when simulating continuous behaviors.

Actions are the fundamental units of behavior in UML, used to define fine-grained behaviors (see Clause [16](#)). Their resolution and expressive power are comparable to the executable instructions in traditional programming languages. Actions are available for use with any of the higher-level formalisms to be used for describing detailed behaviors. The topmost layer in the semantics hierarchy defines the semantics of these higher-level behavioral formalisms of UML: *StateMachines*, *Activities*, and *Interactions* (see Clauses [14](#), [15](#) and [17](#), respectively).

### 6.3.3 Stable and Transient Behavioral Semantics

Though structural semantics, as defined in sub clause 6.3.2, has to do with modeling things at a specific point in time, the structural modeling constructs in UML still include the ability to model certain behavioral aspects of otherwise primarily structural elements. For example, a classifier may have *behavioral features* that can be invoked to request some behavior from the classifier. Or a class may be modeled as being *active*, meaning that an instance of the class has some autonomous behavior.

The behavioral characteristics of primarily structural modeling constructs make high-level statements about the behavior of a system that may generally be verified when the system is in a *stable* state at some specific point in time. However, they do not define *how* the system actually got into that state from a previous state, just that some behavior must have happened to cause this change. The detailed definition of *transient* behavior over time requires the use of behavioral modeling constructs.

In many cases, a structural element in a UML model will have related behavioral elements that define the detailed behavior to realize the high-level behavior identified for the structural element. For example, an operation owned by a class may have a related *method* that defines its detailed behavior. Or an active class may have a *classifier behavior* that details its autonomous behavior. In these cases, it is the responsibility of the modeler to ensure that the detailed transient behavior specified using the behavioral modeling elements actually results in the high-level stable behavior specified for the corresponding structural elements. (A tool may assist the modeler in this responsibility, but a conforming UML tool is not required to do so.)

The following are some areas in which this semantic distinction is particularly important in UML.

- *Operation behaviors.* An *operation* is a behavioral feature of a class that may be directly invoked on instances of that class (see sub clause [9.6](#)). The definition of an operation includes the types of input and output parameters of the operation and may also include pre- and postconditions on the state of the system being modeled before and after invocation of the operation. The semantics of such a model are that, if the operation is invoked with inputs of the given types and in a state in which the precondition hold, then, when the invoked behavior of the operation completes, it will have produced outputs of the given types and the postcondition will hold in the resulting system state. An operation may also have a *method*, which is a detailed definition of its required behavior (see sub clause [13.2](#)). It is a modeler responsibility to ensure that the detailed behavior modeled by the method of the operation meets the behavioral requirements given by the pre- and postconditions of the operation. Note, however, that the postcondition is not required to hold during the *transient* execution of the method behavior, but only at the *stable* point of the completion of execution of that behavior. A class may also have *invariant* conditions that must be true before and after the execution of the operation but may be violated during the course of the execution of the operation method.
- *Property default values.* The semantics of *properties* specify that, when a property with a default value is instantiated, in the absence of some specific setting for the property, the default value is evaluated to provide the initial values of the property (see sub clause [9.5](#)). Thus, when instantiating a classifier, all its attributes (i.e., properties of the classifier) with default values should be properly initialized once any behavior required to instantiate the classifier completes. However, a create object action is specified to create an object with its attributes initially having no initial values, whether or not those attributes have default values in the classifier of the object (see sub clause [16.4.3](#)). Therefore, when modeling the detailed behavior of the instantiation of a classifier, it is a modeler responsibility to ensure that the modeled behavior carries out the proper initialization of any attributes with default values once the object is created. (This is often done by encapsulating the instantiation behavior for a class in a *constructor* operation – see sub clause [11.4](#) – in which case the initialization of the attributes becomes an implicit postcondition for the constructor.)
- *Active class behaviors.* The semantics of *active classes* specify that, when such a class is instantiated, the new object commences execution of its behavior as a direct consequence of its creation (see sub clause [11.4](#)). However, a create object action is specified to create an object *without* commencing the execution of any associated behaviors (see sub



clause [16.4.3](#)). Instead, it is necessary to use a start object behavior action to execute those behaviors (see sub clause [16.3.3](#)). Therefore, when modeling the detailed behavior of the instantiation of a classifier, it is a modeler responsibility to ensure that the modeled behavior properly starts the classifier behavior of an instance of an active class, after that instance is created. (This behavior may also be encapsulated in a constructor operation for the class.)

## 6.4 How to Read this Specification

### 6.4.1 Specification Format

The rest of this document contains the technical content of this specification.

The concepts of UML are grouped into clauses. A clause typically covers a specific modeling formalism. For instance, all concepts related to state machine modeling are gathered in the State Machines clause and all concepts related to activities modeling are in the Activities clause.

The clauses in the specification as a whole are presented in an order that minimizes forward references. Clauses [7](#) – [12](#) are primarily concerned with the modeling of structure. Clauses [13](#) – [17](#) are primarily concerned with the modeling of behavior. Clauses [18](#) – [20](#) cover supplementary concepts including UseCases, Deployments, and InformationFlows. Clauses [21](#) and [22](#) specify primitive types and the standard profile.

[Annex A](#) discusses UML Diagrams. [Annex B](#) specifies a model for the interchange of UML diagrams: this is a new part of the specification that was absent from earlier versions of UML. [Annex C](#) specifies keywords; [Annex D](#) specifies some alternative tabular notations; [Annex E](#) specifies the format for XMI serialization.

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

Within each clause, there is first a brief informal description of the concepts described in that clause. The clause is then split into sub clauses, each describing a coherent set of concepts that constitute a portion of the formalism specified by the clause. Each sub clause is then split into Abstract Syntax, Semantics, Notation, and Examples.

The Abstract Syntax subdivision contains one or more diagrams that define that capability in terms of a MOF model (i.e., the UML metamodel) with each modeling concept represented by an instance of a metaclass or association. These diagrams are designed to provide information about a related set of concepts. Within such a diagram, all of the metaclasses described in that clause are depicted with their attribute compartments, while metaclasses whose definition appears in another clause are depicted with just their headers and no compartments.

The following stylistic conventions are applied in the Semantics, Notation, and Examples subdivisions:

- Headings without numbers are used to break up the sections into meaningful chunks. These headings are organized by coherent chunks of tightly-coupled semantics. Often these headings will turn out to be pluralized metaclass names (e.g., Comments); they might equally represent particular semantic themes (e.g., Run-to-Completion).
- Italics are used for emphasis.
- Names of metaclasses in the text are capitalized but otherwise used as if they are nouns in English, e.g., “Every Element has the inherent capability of owning other Elements,” pluralizing where necessary.
- Names of properties in the text are styled as 8-point Arial, and used as if they are English nouns pluralizing where necessary, e.g., “the ownedAttributes of the Classifier.”

The Semantics subdivision specifies the semantics of all of the concepts described in the sub clause.

The Notation subdivision specifies the notation corresponding to all of the concepts defined in the sub clause. Only concepts that can appear in diagrams will have a notation specified. For textual notations a variant of the Backus-Naur Form (BNF) is often used to specify the legal formats. The conventions of this BNF are:

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

(<item-1> | <item-2>)\*

signifies a sequence of one or more items, each of which is <item-1> or <item-2>.

The Examples subdivision gives examples intended to illustrate the concepts in the sub clause.

Diagrams appearing in the Notation and Examples subdivisions have been produced by a variety of tools, and may differ in stylistic details such as fonts, line thicknesses, size of arrowheads, etc. Such differences are not material to the specification.

Finally in each clause are machine-generated sub clauses called Classifier Descriptions and Association Descriptions, containing a complete description for all of the classifiers and associations in the metamodel. In Classifier Descriptions, each classifier (Class, Abstract Class, or Enumeration) is documented under the following headings:

- Name [Type]
- Description: a summary of the role played by the classifier in the metamodel.
- Diagrams: a list of links to diagrams in which the classifier appears.
- Generalizations: a list of links to generalizing classifiers, if any.
- Specializations: a list of links to specializing classifiers, if any.
- Attributes: each specified by its name, type, and multiplicity, and any additional properties such as {readOnly}. If no multiplicity is listed, it defaults to 1..1. This is followed by a textual description of the purpose and meaning of the attribute. If an attribute is derived, the name will be preceded by a slash. Where an attribute is derived, the logic of the derivation is in most cases shown using OCL.
- Association Ends: each specified by its name, type, and multiplicity, any additional properties such as {union}, and a link to its opposite end. If the association end subsets or redefines others, this is shown in the additional properties as {subsets <end>} or {redefines <end>}, where <end> is a link to the applicable end. This is followed by a textual description of the purpose and meaning of the association end. If an association end is derived, the name will be preceded by a slash. If the association end is a composition, this is indicated by a small black diamond adjacent to the name of the end.
- Operations: each specified by its signature, a textual description of the logic of the operation, and a specification of the logic of the operation in OCL. Note that in some cases the OCL is absent.

- Constraints: each specified by its name, a textual description of the logic of the constraint, and a specification of the logic of the constraint in OCL. Note that in some cases the OCL is absent.

In Association Descriptions, each association is documented under the following headings:

- Name [Type].
- Diagrams: a list of links to diagrams in which the association appears.
- Generalizations: a list of links to generalizing associations, if any.
- Specializations: a list of links to specializing associations, if any.
- Member Ends: links to each end of the association; this appears if neither of the ends is owned by the association itself.
- Owned Ends: documentation for each association end owned by the association itself, each specified by its name, type and multiplicity, any additional properties such as {union}, and a link to its opposite end. If the association end subsets or redefines others, this is shown in the additional properties as {subsets <end>} or {redefines <end>}, where <end> is a link to the applicable end. If an association end is derived, the name will be preceded by a slash.

## 6.4.2 Diagram Format

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

- A metaclass may appear on many diagrams, but takes a primary role on only one diagram, which is the diagram adjacent to where the semantics of the metaclass are described. A metaclass in a primary role is shown with its attribute compartment expanded; a metaclass in a secondary role is shown as just its header rectangle.
- Dot notation is used to denote association end ownership.
- Arrow notation is used to denote association end navigability. By definition, all class-owned association ends are navigable. By convention, all association-owned ends in the metamodel are not navigable.
- An association with neither end marked by navigability arrows means that the association is navigable in both directions.
- Association specialization and redefinition are indicated by appropriate constraints situated in the proximity of the association ends to which they apply. Thus:
  - The constraint {subsets endA} means that the association end to which this constraint is applied subsets the association end endA.
  - The constraint {redefines endA} means that the association end to which this constraint is applied redefines the association end endA.
- If no multiplicity is shown on an association end, it implies a multiplicity of exactly 1.
- If an association end is unlabeled, the default name for that end is the name of the class to which the end is attached, modified such that the first letter is a lowercase letter. Note that, by convention, non-navigable association ends are often left unlabeled although all association ends have a name which is documented in the Association Description section of each clause.
- Associations that are not explicitly named, are given names that are constructed according to the following production rule:
 

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

where *<association-end-name1>* is the name of the first association end and *<association-end-name2>* is the name of the second association end.

## **6.5 Acknowledgements**

### **6.5.1 Primary Authors**

The following people wrote this specification, incorporating the work of authors of earlier versions of UML:

Conrad Bock, Steve Cook (lead), Pete Rivett, Tom Rutt, Ed Seidewitz, Bran Selic, Doug Tolbert

### **6.5.2 Technical Support**

The following people provided technical support for this specification, including writing tools to generate portions of the document and to validate the OCL:

Peter Denno, Maged Elaasar, Nicolas Rouquette, Ed Willink

### **6.5.3 Reviewers**

In addition to the authors and technical supporters, the following people provided invaluable contributions by reviewing some or all of the specification in detail:

Omar Bahy Badreddin, Neil Capey, Michael Jesse Chonoles (lead), Adriano Comai, Lenny Delligatti, Sanford Friedenthal, Dave Hawkins, Darren Kumasawa, Jim Logan, Sam Mancarella, Milagros Nguyen, Axel Scheithauer, John Watson, Marc-Florian Wendland, Ed Willink.

# 7 Common Structure

## 7.1 Summary

This clause specifies the basic modeling concepts underlying all structural modeling in UML. Many of the metaclasses defined here are abstract, providing the base for specialized, concrete classes defined in subsequent clauses. However, in order to provide examples of how these basic concepts are applied in UML, it is necessary to use these concrete modeling constructs, even though they are specified in later clauses. Appropriate forward references are provided as necessary.

## 7.2 Root

### 7.2.1 Summary

The root concepts of Element and Relationship provide the basis for all other modeling concepts in UML.

### 7.2.2 Abstract Syntax

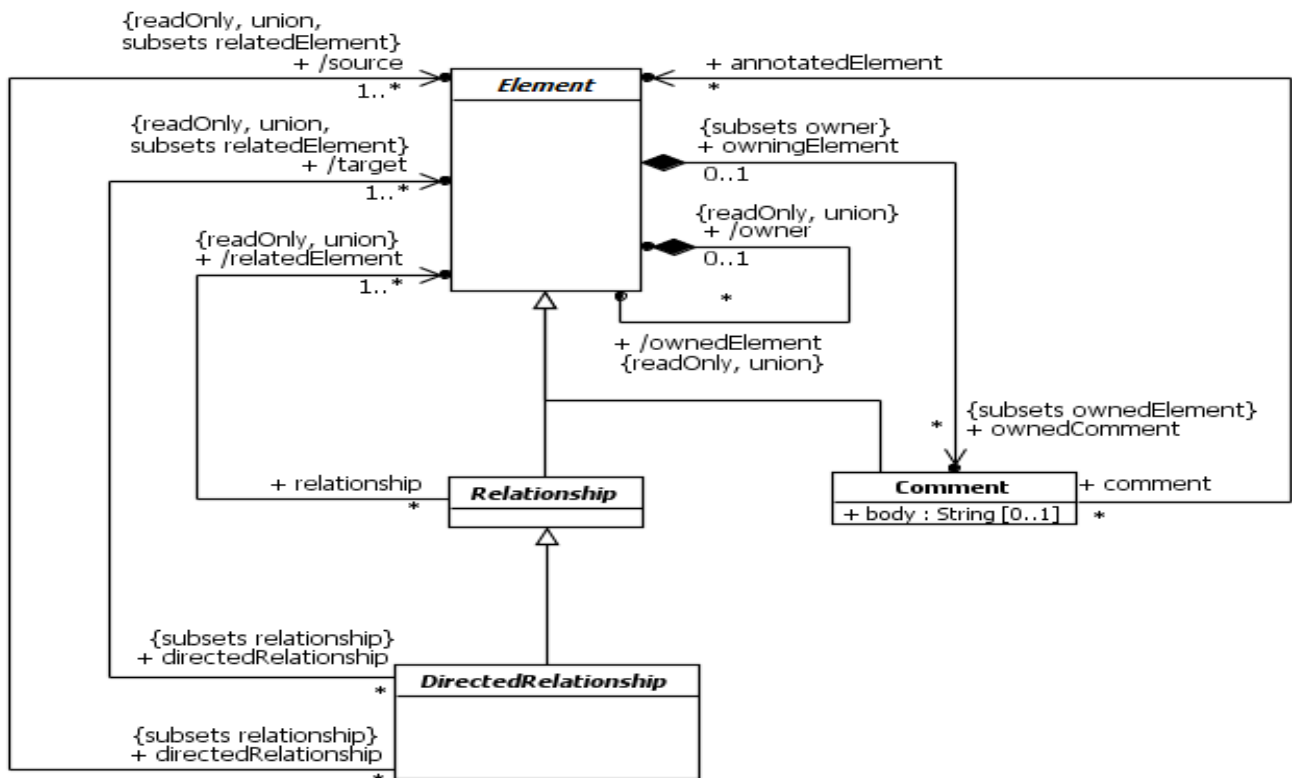


Figure 7.1 Root

## 7.2.3 Semantics

### Elements

An Element is a constituent of a model. Descendants of Element provide semantics appropriate to the concept they represent.

Every Element has the inherent capability of owning other Elements. When an Element is removed from a model, all its ownedElements are also necessarily removed from the model. The abstract syntax for each kind of Element specifies what other kind of Elements it may own. Every Element in a model must be owned by some other Element of that model, with the exception of the top-level Packages of the model.

### Comments

Every kind of Element may own Comments. The ownedComments for an Element add no semantics but may represent information useful to the reader of the model.

### Relationships

A Relationship is an Element that specifies some kind of relationship between other Elements. Descendants of Relationship provide semantics appropriate to the concept they represent.

A DirectedRelationship represents a Relationship between a collection of source model elements and a collection of target model elements. A DirectedRelationship is said to be directed *from* the source elements *to* the target elements.

## 7.2.4 Notation

There is no general notation for Element, Relationships, and DirectedRelationships. The descendants of these classes define their own notation. For Relationships, in most cases the notation is a variation on a line drawn between the relatedElements. For DirectedRelationships, the line is usually directed in some way from the source(s) to the target(s).

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 annotatedElement is shown by a separate dashed line. The dashed line connecting the note symbol to the annotatedElement(s) may be suppressed if it is clear from the context, or not important in this diagram.

## 7.2.5 Examples

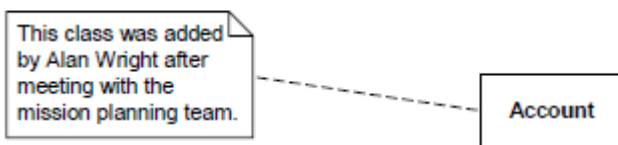


Figure 7.2 Comment notation

## 7.3 Templates

### 7.3.1 Summary

Templates are model Elements that are parameterized by other model Elements. This sub clause specifies the general concepts applicable to all kinds of templates. Further details of specific kinds of templates allowed in UML are discussed in later sub clauses, including Classifier templates (see sub clause 9.3), Operation templates (see sub clause 9.6) and Package templates (see sub clause 12.2).

### 7.3.2 Abstract Syntax

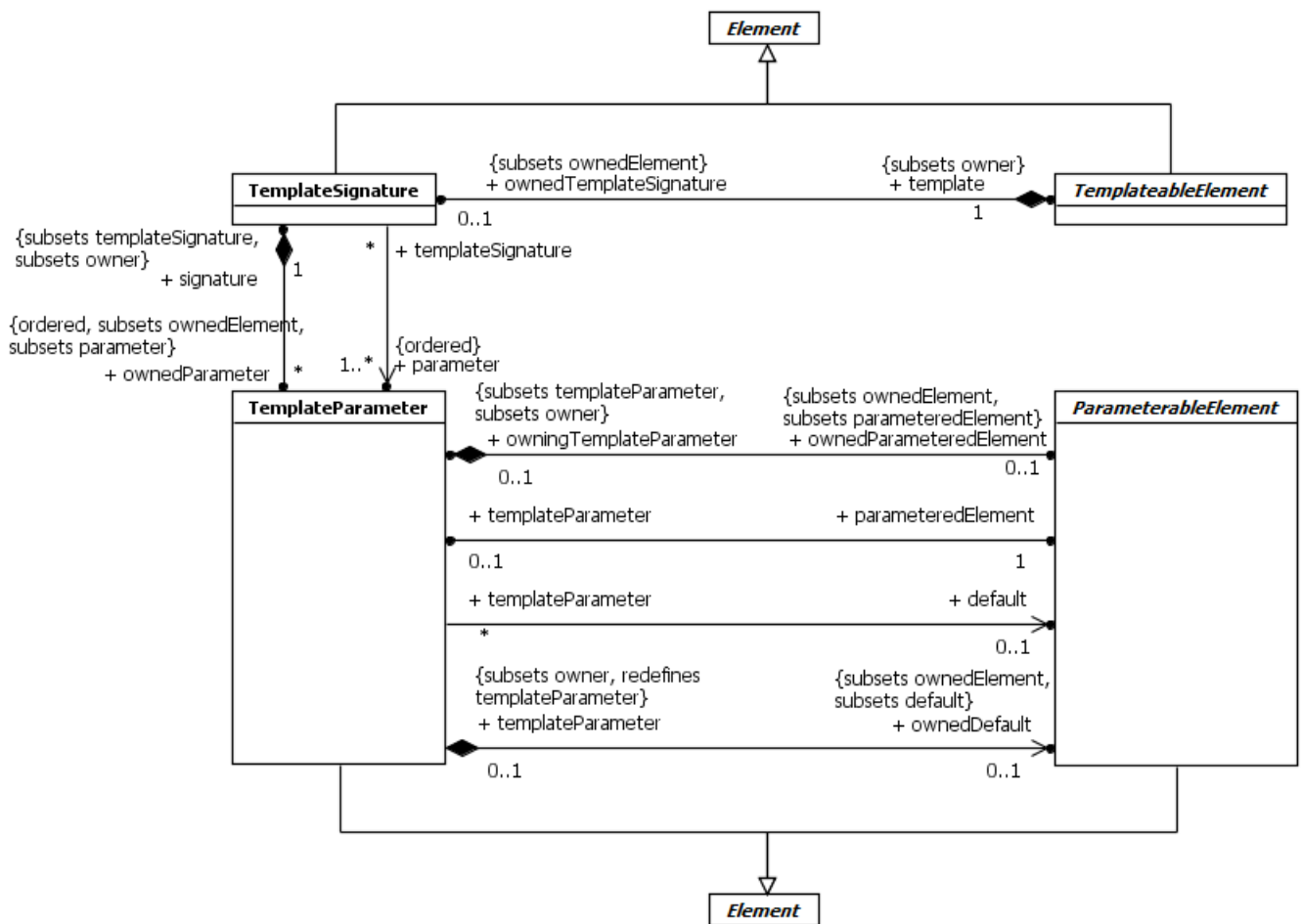


Figure 7.3 Templates

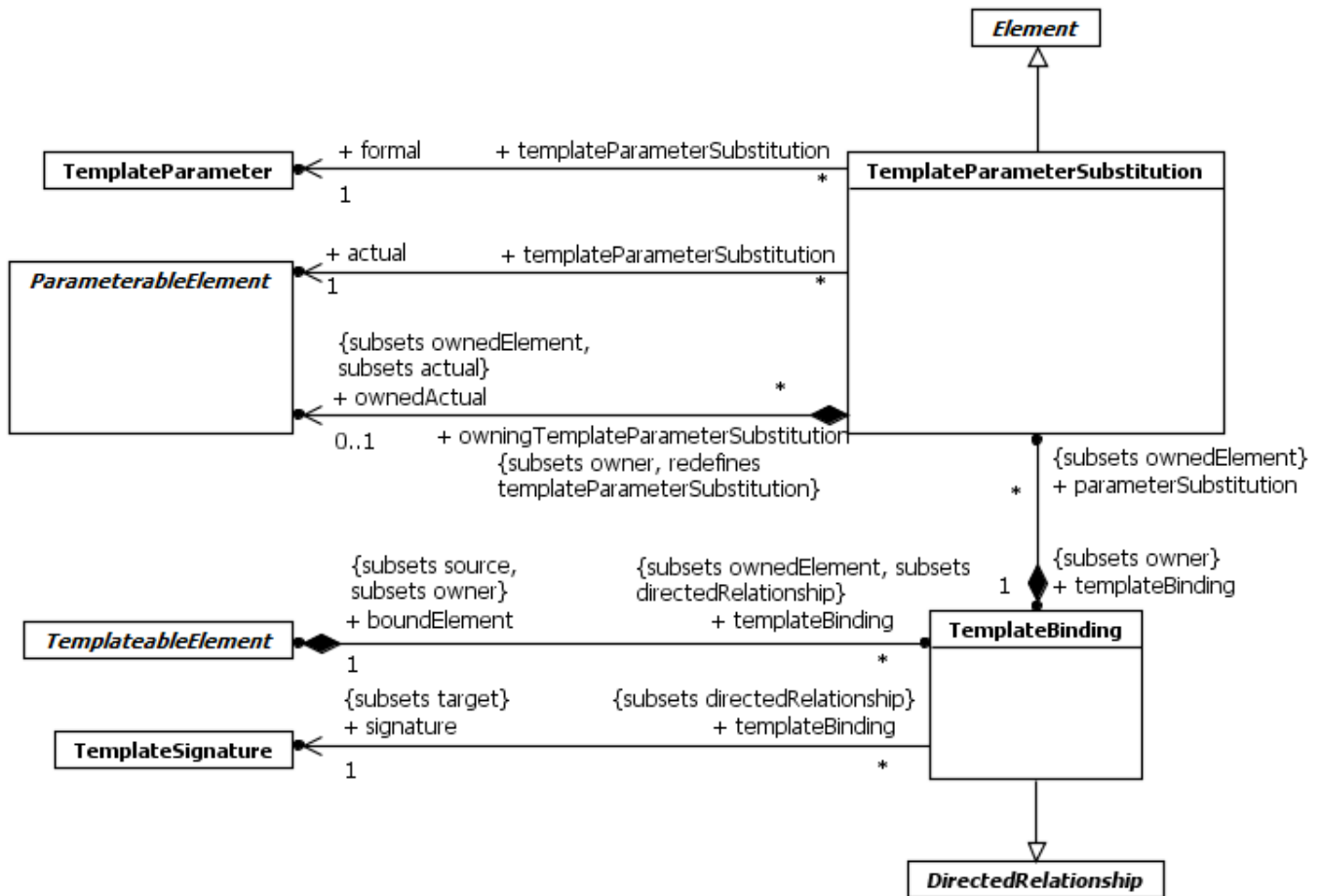


Figure 7.4 Template bindings

### 7.3.3 Semantics

#### Templates

A *TemplateableElement* is an *Element* that can optionally be defined as a template and bound to other templates. A *template* is a *TemplateableElement* that is parameterized using a *TemplateSignature*. Such a template can be used to generate other model Elements using *TemplateBinding* relationships.

A template cannot be used in the same manner as a non-template *Element* of the same kind (e.g., a template *Class* cannot be used as the type of a *TypedElement*). The template *Element* can only be used to generate bound Elements or as part of the specification of another template (e.g., a template *Class* may specialize another template *Class*).

The *TemplateSignature* of a template defines a set of *TemplateParameter*s that may be *bound* to actual model Elements in a bound element for the template. A *bound element* is a *TemplateableElement* that has one or more such *TemplateBinding*s.

A *completely bound element* is a bound element all of whose *TemplateBinding*s bind all the *TemplateParameter* of the template being bound. A completely bound element is an ordinary element and can be used in the same manner as a non-bound (and non-template) element of the same kind. For example, a completely bound element of a *Class* template may be used as the type of a *TypedElement*.



A *partially bound element* is a bound element at least one of whose `TemplateBindings` does not bind a `TemplateParameter` of the template being bound. A partially bound element is still considered to be a template, parameterized by the remaining `TemplateParameters` left unbound by its `TemplateBindings`.

## Template Signatures

The `TemplateParameters` for a `TemplateSignature` specify the formal parameters that will be substituted by actual parameters (or the default) in a binding. A `TemplateParameter` is defined in terms of a `ParameterableElement` contained within the template that owns the `TemplateSignature` of which the `TemplateParameter` is a part. Such an element is said to be *exposed* by the `TemplateParameter`.

An exposed `ParameterableElement` may be owned, directly or indirectly, by the template or it may be owned by the `TemplateParameter` itself, in situations in which the element does not otherwise have an ownership association within the template model. In either case, the `ParameterableElement` is meaningful only within the context of the template—it will be effectively replaced by an actual `Element` in the context of a binding. Thus, a `ParameterableElement` exposed by a `TemplateParameter` cannot be referenced outside its owning template or other templates that have access to the internals of the original template (e.g., if the template is specialized). Subclasses of `TemplateSignature` can also add additional rules that constrain what sort of `ParameterableElement` can be used for a `TemplateParameter` in the context of a particular kind of template.

A `TemplateParameter` may also reference a `ParameterableElement` as the default for this formal parameter in any `TemplateBinding` that does not provide an explicit `TemplateParameterSubstitution` for the parameter. Similarly to an exposed `ParameterableElement`, a default `ParameterableElement` may be owned either directly by the template or by the `TemplateParameter` itself. The `TemplateParameter` may own this default `ParameterableElement` even in situations where the exposed `ParameterableElement` is not owned by the `TemplateParameter`.

## Template Bindings

A `TemplateBinding` is a relationship between a `TemplateableElement` and a template that specifies the substitutions of actual `ParameterableElements` for the formal `TemplateParameters` of the template. A `TemplateParameterSubstitution` specifies the actual parameter to be substituted for a formal `TemplateParameter` within the context of a `TemplateBinding`. If no actual parameter is specified in this binding for a formal parameter, then the default `ParameterableElement` for that formal `TemplateParameter` (if specified) is used.

A bound element may have multiple bindings, possibly to the same template. In addition, the bound element may contain elements other than the bindings. The details of how the expansions of multiple bindings, and any other `Elements` owned by the bound element, are combined together to fully specify the bound element are specific to the subclasses of `TemplateableElement`. The general principle is that one evaluates the bindings in isolation to produce intermediate results (one for each binding), which are then merged to produce the final result. It is the way the merging is done that is specific to each kind of `TemplateableElement`.

A `TemplateableElement` may contain both a `TemplateSignature` and `TemplateBindings`. Thus a `TemplateableElement` may be both a template and a bound element.

A conforming tool may require that all formal `TemplateParameters` must be bound as part of a `TemplateBinding` (complete binding) or may allow just a subset of the formal `TemplateParameters` to be bound (partial binding). In the case of complete binding, the bound element may have its own `TemplateSignature`, and the `TemplateParameters` from this can be provided as actual parameters of the `TemplateBinding`. In the case of partial binding, the unbound formal `TemplateParameters` act as formal `TemplateParameters` of the bound element, which is thus still a template.

**NOTE.** A `TemplateParameter` with a default can never be unbound, as it has an implicit binding to the default, even if an explicit `TemplateParameterSubstitution` is not given for it.

## Bound Element Semantics

A `TemplateBinding` implies that the bound element has the same well-formedness constraints and semantics as if the contents of the template owning the target `TemplateSignature` were copied into the bound element, substituting any `ParameterableElements`

exposed as formal TemplateParameters by the corresponding ParameterableElements specified as actual template parameters in the TemplateBinding. However, a bound element does not explicitly contain the model Elements implied by expanding the templates to which it binds. Nevertheless, it is possible to define an *expanded bound element* that results from actually applying the TemplateParameterSubstitution for a bound element to the target templates.

Formally, an expanded bound element for a bound element with a single TemplateBinding and no Elements other than from that binding is constructed as follows:

1. Copy the template associated with the TemplateSignature that is the target of the TemplateBinding. For the present purposes, a copy of a model Element is an instance of the same metaclass as the original model Element, with:
  - a. Values for all composite properties (owned attributes and owned association ends) that are copies (in the same sense) of the corresponding values from the original Element.
  - b. Values for all non-composite properties that are the same as the corresponding values from the original Element, except that references to Elements owned (directly or indirectly) by the original Element are replaced with references to the copies of those Elements created as specified above and references to the original Element itself are replaced by references to the copy.
2. If the copy specializes any Elements that are templates, then redirect the Generalization relationships to equivalent bound elements for the general elements, using the same TemplateBinding. If the copy is an Operation that has an associated method that is also a template, then replace that method with an equivalent bound element using the same template binding.

**NOTE.** It is necessary for the method of a template Operation to also be a template, presumably with TemplateParameters corresponding to those of the Operation. In particular, Operation TemplateParameters are typically used to parameterize the types of Operation Parameters, but the method of an Operation does not directly reference the Parameters of the Operation that specifies it. Rather, the method has its own ownedParameter list, which should match that of the Operation (see sub clause [13.2](#)). The types of the method Parameters thus need to be separately templated to match the template parameterization of the Operation.

3. For each Element owned directly or indirectly by the copy, replace any reference to the parameteredElement of a TemplateParameter of the copy with a reference to the actual Element associated with the parameter in the TemplateBinding. If an actual Element has a TemplateBinding itself, then reference the equivalent bound element.
4. Remove all TemplateParameters that are referenced in the TemplateBinding from the TemplateSignature of the copy. If this would remove all TemplateParameters from the TemplateSignature, then remove the TemplateSignature entirely.

If a bound element has more than one TemplateBinding, then a specific expanded bound element can be defined based on each TemplateBinding. The overall expanded bound element is then constructed by merging all the TemplateBinding-specific expanded bound elements with any other Elements contained by the original bound element. As noted previously, how this merging is performed depends on the kind of TemplateableElement being bound.

Including a bound element in a model does *not* automatically require that the corresponding expanded bound element be included in the model. However, if the expanded bound element constructed as given above violates any well-formedness constraints, then the original bound element is also considered to not be well formed.

On the other hand, if the bound element is for a Namespace template, then it may be necessary to be able to refer to members of the bound element considered as a Namespace itself. For example, for a bound element of a Class template, it may be necessary to reference Operations of that Class, e.g., from a CallOperationAction.

**NOTE.** Referencing the Operation from the template is not sufficient, as each bound element of the template Class is to be considered to have its own effective copy of the Operations of the template.

In order to accommodate a situation like this, it is allowable to include in a model the expanded bound element for a bound element *in addition to* the bound element itself. In this case, the expanded bound element must have a realization dependency (see sub clause [7.7](#)) to the bound element that it is expanding. The expanded bound element must be constructed (either manually by

the modeler or automatically by a tool) according to the rules given above. References then made as usual from other model elements to visible members of the expanded bound element are considered to be semantically equivalent to effective references made to the corresponding implicit members of the original bound element. Any relationships made directly to the expanded bound element are semantically equivalent to relationships made to the bound element itself.

### 7.3.4 Notation

If a TemplateableElement has TemplateParameters, a small dashed rectangle is superimposed on the symbol for the TemplateableElement, typically on the upper right-hand corner of the notation (if possible). The dashed rectangle contains a list of the formal TemplateParameters. The parameter list must not be empty, although it may be suppressed in the presentation. Any other compartments in the notation of the TemplateableElement appear as normal.

The formal TemplateParameter list may be shown as a comma-separated list, or it may be one formal TemplateParameter per line. The general notation for a TemplateParameter is a string displayed within the TemplateParameter list for the template:

*<template-parameter> ::= <template-param-name> [ ':' <parameter-kind> ] [ '=' <default> ]*

where *<parameter-kind>* is the name of the metaclass for the exposed element. The syntax of *<template-param-name>* and *<default>* depend on the kind of ParameteredElement for this TemplateParameter.

A bound element has the same graphical notation as other Elements of that kind. A TemplateBinding is shown as a dashed arrow with the tail on the bound element and the arrowhead on the template and the keyword «bind». The binding information may be displayed as a comma-separated list of template parameter substitutions:

*<template-param-substitution> ::= <template-param-name> '->' <actual-template-parameter>*

where the syntax of *<template-param-name>* is the name of the formal TemplateParameter and the kind of *<actual-template-parameter>* depends upon the kind of ParameteredElement for that TemplateParameter.

An alternative presentation for the bindings for a bound element is to include the binding information within the notation for the bound element. The name of the bound element is extended to contain binding expressions with the following syntax:

*[<element-name> ':' ] <binding-expression> [ ',' <binding-expression> ]\**

*<binding-expression> ::= <template-element-name> '<' <template-param-substitution> [ ',' <template-param-substitution> ]\* '>'*

and *<template-param-substitution>* is defined as above.

## 7.4 Namespaces

### 7.4.1 Summary

A Namespace is an Element in a model that contains a set of NamedElements that can be identified by name. Packages (see Clause 12) are Namespaces whose specific purpose is to contain other NamedElements in order to organize a model, but many other kinds of model Elements are also Namespaces, including Classifiers (see sub clause 9.2), which contain named Features and nested Classifiers, and BehavioralFeatures (see sub clause 9.4), which contain named Parameters.

## 7.4.2 Abstract Syntax

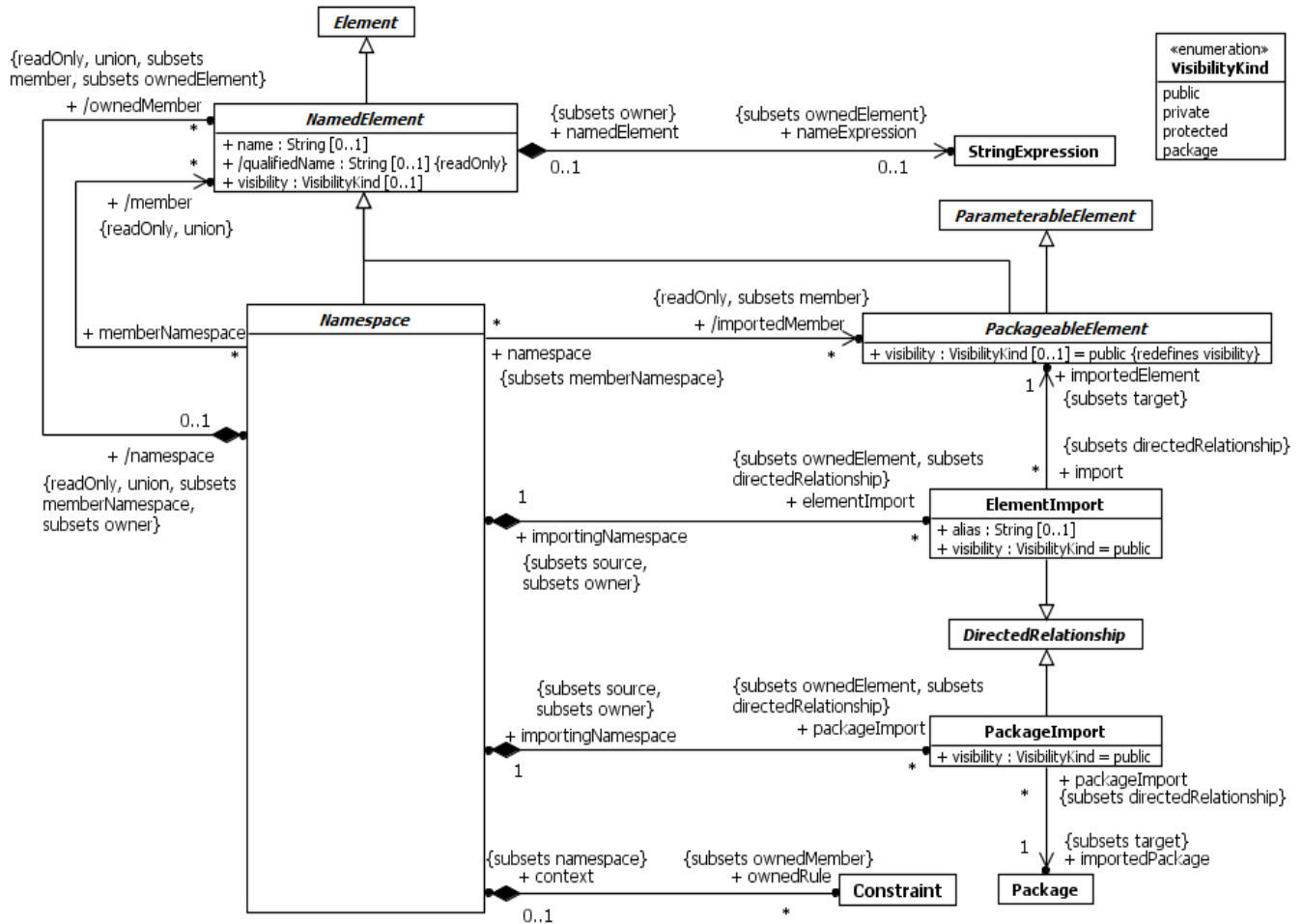


Figure 7.5 Namespaces

## 7.4.3 Semantics

### Namespaces

A Namespace provides a container for NamedElements, which are called its ownedMembers. A namespace may also import NamedElements from other Namespaces, in which case these, along with the ownedMembers, are members of the importing Namespace. If a member of a Namespace with the name *N* is a NamedElement with the name *x*, then the member can be referred to by a *qualified name* of the form *N::x*.

**NOTE.** The set of members of a Namespace is different from the set of all of the names that can be referred to unqualified within the Namespace, because that set also includes all unhidden members of enclosing Namespaces.

As a Namespace is itself a NamedElement, the fully qualified name of a NamedElement may include multiple Namespace names, such as *N1::N2::x*.

The ownedRule Constraints for a Namespace represent well-formedness rules for the constrained elements (see sub clause 7.6 on Constraints). These constraints are evaluated when determining if the constrained elements are well-formed.

## Named Elements

A NamedElement is an Element in a model that may have a name. The name may be used for identification of the NamedElement within Namespaces where its name is accessible.

**NOTE.** The name of a NamedElement is optional, which provides for the possibility of the absence of a name (which is different from the empty name).

NamedElements may appear within a Namespace according to rules that specify how one NamedElement is distinguishable from another. The default rule is that two members are distinguishable if they have different names or if they have the same names, but their metaclasses are different and neither is a (direct or indirect) subclass of the other. This rule may be overridden for particular cases, such as Operations that are distinguished by their signature (see sub clause [9.6](#)).

The visibility of a NamedElement provides a means to constrain the usage of the Element, either in Namespaces or in access to the Element. It is intended for use in conjunction with import, generalization, and access mechanisms.

A NamedElement may, in addition to having an explicit name, be associated with a StringExpression (see sub clause [8.3](#)) that may be used to specify a calculated name for the NamedElement. In a template (see sub clause 7.3), a NamedElement may have an associated whose subexpressions may be ParameteredElements exposed by TemplateParameters. When the template is bound, the exposed subexpressions are substituted with the actuals substituted for the TemplateParameters. The value of the StringExpression is then a string resulting from concatenating the values of the subexpression, which then provides the name of the NamedElement.

A NamedElement may have both a name and a nameExpression associated with it. In this case, the name can be used as an alias for the NamedElement, which may be used, for example, in referencing the element in a Constraint expression. (This avoids the need to use StringExpressions in textual surface notation, which is often cumbersome, although it does not preclude it.)

## Packageable Elements and Imports

A PackageableElement is a NamedElement that may be owned directly by a Package (see Clause [12](#) on Packages). Any such element may serve as a TemplateParameter (see sub clause 7.3 on Templates).

An ElementImport is a DirectedRelationship between an importing Namespace and a PackageableElement. It adds the name of the PackageableElement to the importing Namespace. The visibility of the ElementImport may be either the same or more restricted than that of the imported element.

In case of a name clash with an outer name (an element that is defined in an enclosing Namespace that is available using its unqualified name in enclosed Namespaces) in the importing Namespace, the outer name is hidden by an ElementImport, and the unqualified name refers to the imported element. The outer name can be accessed using its qualified name.

A PackageImport is a DirectedRelationship 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 ElementImport to each individual member of the imported Namespace, unless there is a separately-defined ElementImport. If there is an ElementImport for an element, then this takes precedence over a potential import of the same element via a PackageImport.

If indistinguishable Elements would be imported into a Namespace as a consequence of ElementImports or PackageImports, the Elements are not added to the importing Namespace and the names of those Elements must be qualified in order to be used in that Namespace. If the name of an imported Element is indistinguishable from an element owned by the importing Namespace, that Element is not added to the importing Namespace and the name of that Element must be qualified in order to be used.

An imported Element that is publicly visible can be further imported by other Namespaces using either ElementImports or PackageImports.

**NOTE.** A Namespace may not import itself, nor may it import any of its own ownedMembers. This means that it is not possible for a NamedElement to acquire an alias in its owning Namespace.

## 7.4.4 Notation

### Namespaces

There is no general notation for Namespaces. Specific kinds of Namespace have their own specific notation.

Conforming tools may optionally allow the “circle-plus” notation defined in sub clause [12.2.4](#) to show Package membership to also be used to show membership in other kinds of Namespaces (for example, to show nestedClassifiers and ownedBehaviors of Classes).

### Name Expressions

The nameExpression associated with a NamedElement can be shown in two ways, depending on whether an alias is required or not. Both notations are illustrated in Figure 7.6.

- *No alias:* The StringExpression appears as the name of the model Element.
- *With alias:* Both the StringExpression and the alias are shown wherever the name usually appears. The alias is given first and the StringExpression underneath.

In both cases the StringExpression appears between “\$” signs. The specification of Expressions in UML supports the use of alternative string expression languages in the abstract syntax—they have to have String as their type and can be some structure of operator Expressions with operands. The notation for this is discussed in sub clause [8.3](#) on Expressions. In the context of templates, subexpressions of a StringExpression (usually LiteralStrings) that are parametered in a template are shown between angle brackets.

### Imports

A PackageImport or ElementImport is shown using a dashed arrow with an open arrowhead from the importing Namespace to the imported Package or Element. The keyword «import» is shown near the dashed arrow if the visibility is public; otherwise, the keyword «access» is shown to indicate private visibility. The alias may be shown after or below the keyword «import». If the imported element for an ElementImport 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 a PackageImport or ElementImport by having a text that uniquely identifies the imported Package or Element within curly brackets either below or after the name of the Namespace. The textual syntax for a PackageImport is:

```
{import ' <qualified-name> '} | {access ' <qualified-name> '} | {element import ' <qualified-name> '}
```

The textual syntax for an ElementImport is:

```
{element import ' <qualified-name> '} | {element access ' <qualified-name> '}
```

Optionally, the alias, if any, may be shown as well:

```
{element import ' <qualified-name> ' as ' <alias> '} | {element access ' <qualified-name> ' as ' <alias> '}
```

## 7.4.5 Examples

### Name Expressions

Figure 7.6 shows a ResourceAllocation Package template where the first two formal TemplateParameters are StringExpression parameters. These formal TemplateParameters are used within the Package template to name some of the Classes and Association

ends. The figure also shows a bound Package (named TrainingAdmin) that has two bindings to this ResourceAllocation template. The first binding substitutes the string “Instructor” for Resource, the string “Qualification” for ResourceKind, and the Class TrainingAdminSystem for System. The second binding substitutes the string “Facility” for Resource, the string “FacilitySpecification” for ResourceKind, and the Class TrainingAdminSystem is again substituted for System.

The result of the binding includes Classes Instructor, Qualification, and InstructorAllocation as well as Classes Facility, FacilitySpecification, and FacilityAllocation. The associations are similarly replicated.

**NOTE.** Request will have two attributes derived from the single “the<ResourceKind>” attribute (shown here by an arrow), namely theQualification and theFacilitySpecification.

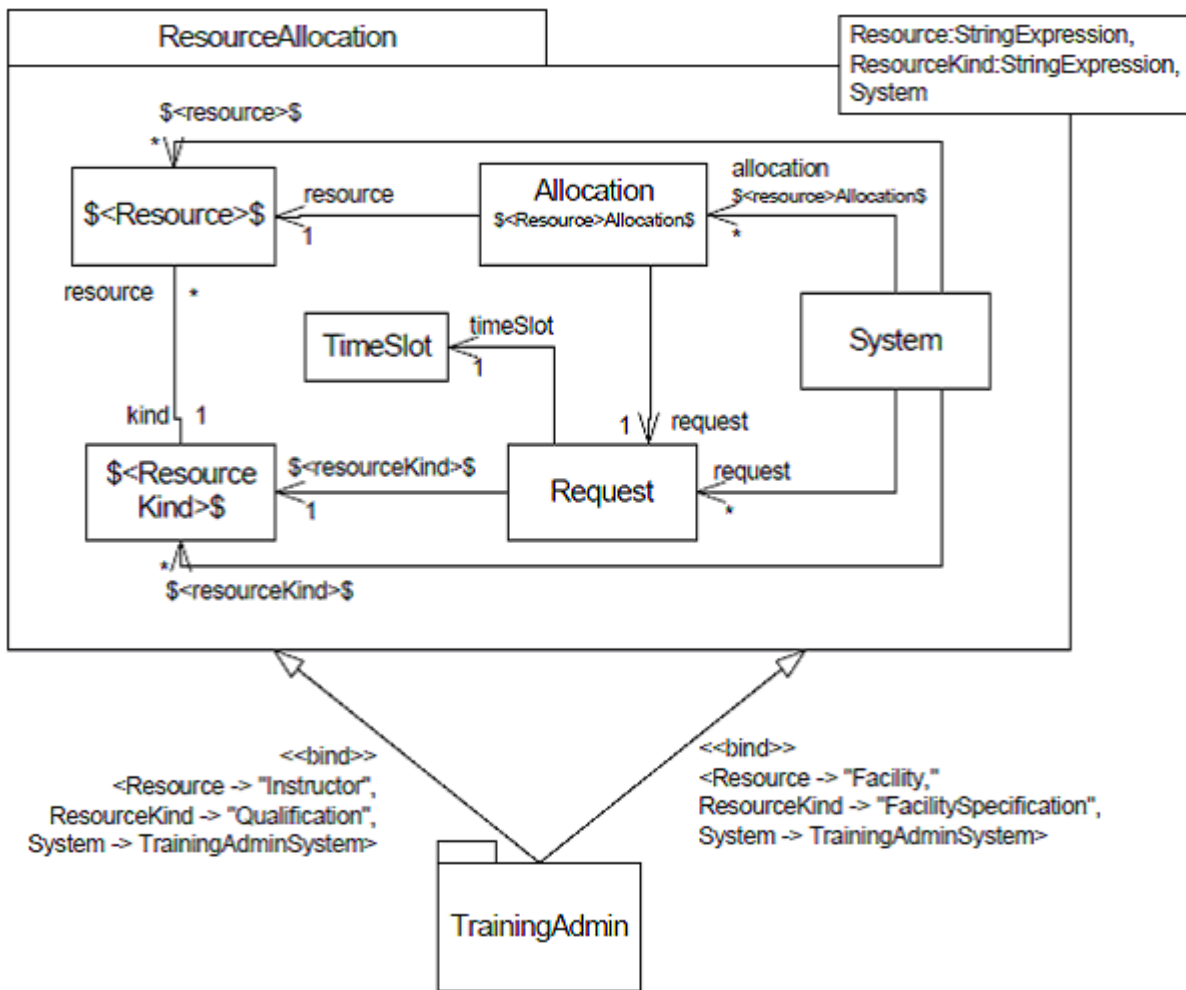


Figure 7.6 Template package with string parameters

### Imports

The ElementImport shown in Figure 7.7 allows Elements in the Package Program to refer by name to the DataType Time in Types without qualification. However, they still need to refer explicitly to Types::Integer, as this Element is not imported. The DataType String is imported into the Program Package but it is not a publicly visible as a member of Program outside of that Package, and it cannot be further imported from the Program Package by other Namespaces.

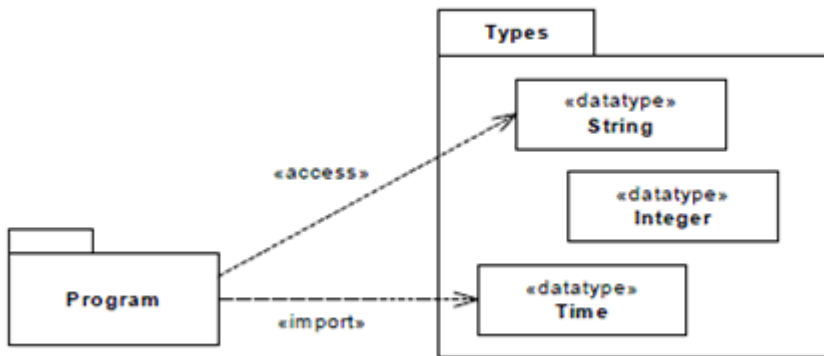


Figure 7.7 Example of element import

In Figure 7.8, the ElementImport is combined with aliasing, meaning that the DataType Types::Real will be referred to by name as Double in the package Shapes.

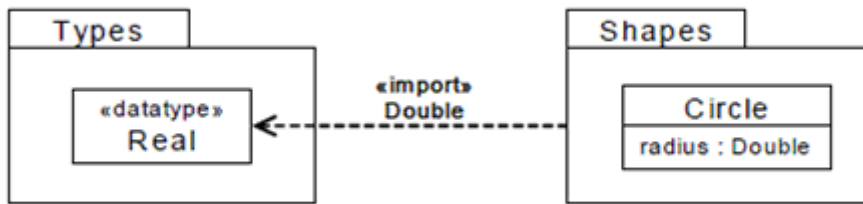


Figure 7.8 Example of element import with aliasing

In Figure 7.9, a number of PackageImports are shown. The public members of Types are imported into ShoppingCart and then further imported into WebShop. However, the members of Auxiliary are only privately imported by ShoppingCart and cannot be referenced using unqualified names from WebShop.

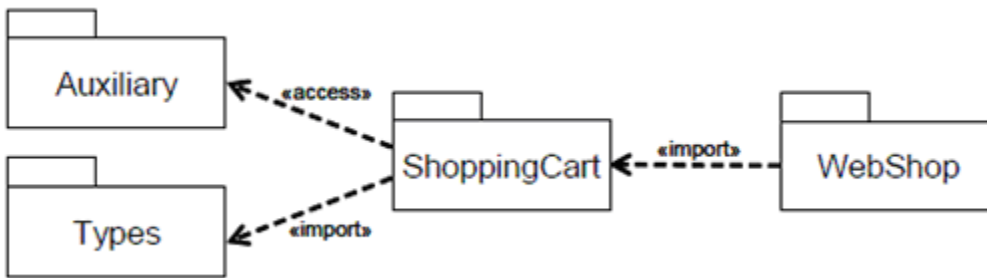


Figure 7.9 Examples of public and private package imports

## 7.5 Types and Multiplicity

### 7.5.1 Summary

Types and multiplicity are used in the declaration of Elements that contain values, in order to constrain the kind and number of values that may be contained.



## 7.5.2 Abstract Syntax

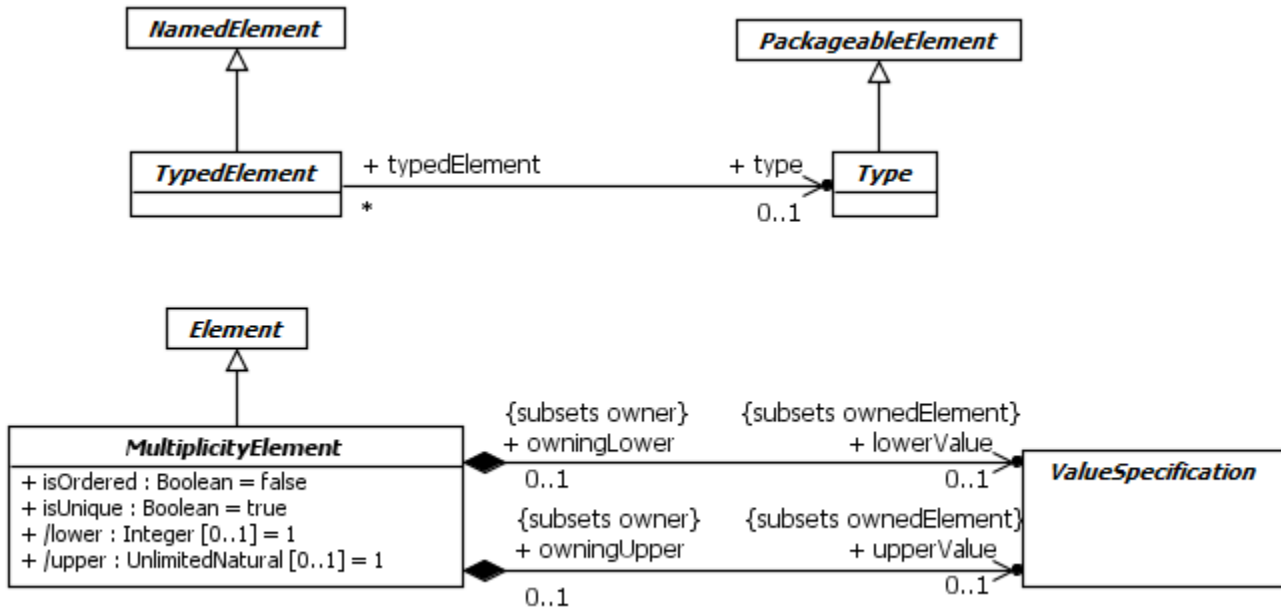


Figure 7.10 Abstract syntax of types and multiplicity elements

## 7.5.3 Semantics

### Types

A Type represents a set of values. A TypedElement that has this Type is constrained to represent values within this set. Values represented by the element are constrained to be instances of the Type. A TypedElement with no associated Type may represent values of any Type.

### Multiplicities

A multiplicity specifies valid cardinalities as an inclusive interval of non-negative integers beginning with a lower bound and ending with a (possibly infinite) upper bound. A MultiplicityElement embeds this information to specify the allowable cardinalities for an instantiation of the Element. The lower and upper bounds for the multiplicity of a MultiplicityElement may be specified by ValueSpecifications (see Clause 8), such as (side-effect free, constant) Expressions.

If a MultiplicityElement specifies a multivalued multiplicity (i.e., upper bound greater than 1), then an instantiation of this element has a collection of values. The multiplicity is a constraint on the number of values that may validly occur in that set.

A MultiplicityElement can define a multiplicity both of whose bounds are zero. This restricts the allowed cardinality to be 0; that is, it requires that an instantiation of this element contain no values. This is useful in the context of Generalizations (see sub clause 9.2) to constrain the cardinalities of a more general Classifier. It applies to (but is not limited to) redefining properties existing in more general Classifiers.

If the MultiplicityElement is specified as ordered (i.e., isOrdered is true), then the collection 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 collection 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 collection of values in an instantiation of this Element must be unique. That is, no two values in the collection may be equal, where equality of objects (instances of Classes) is based on object identity while equality of data values (instances of DataTypes) and Signal instances is based on value (see also sub clauses 10.2, 10.3, and 11.4 on DataTypes, Signals and Classes, respectively ). If a MultiplicityElement is not multivalued, then the value for isUnique has no semantic effect.

Taken together, the isOrdered and isUnique properties can be used to specify that the collection of values in an instantiation of a MultiplicityElement is of one of four types. Table 7.1 shows the traditional names given to each of these collection types.

**Table 7.1 Collection types for MultiplicityElements**

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

## 7.5.4 Notation

### Multiplicity Element

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

The multiplicity bounds may be shown in the format:

*<lower-bound> ‘.’ <upper-bound>*

where <lower-bound> is a ValueSpecification of type Integer and <upper-bound> is a ValueSpecification of type UnlimitedNatural. The star character (\*) is used as part of a multiplicity specification to represent an unlimited upper bound.

If the multiplicity is associated with a MultiplicityElement whose notation is a text string (such as an attribute), the multiplicity string is placed within square brackets ( [ ] ) as part of that text string.

If the multiplicity is associated with a MultiplicityElement 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.

If the lower bound is equal to the upper bound, then an alternate notation is to use a string containing just the upper bound. For example, “1” is semantically equivalent to “1..1” multiplicity. A multiplicity with zero as the lower bound and an unspecified upper bound may use the alternative notation containing a single star “\*” instead of “0..\*” multiplicity.

The specific notation for the ordering and uniqueness specifications may vary depending on the specific kind of MultiplicityElement. A general notation is to use a textual annotation containing “ordered” or “unordered” to define the ordering, and “unique” or “nonunique” to define the uniqueness.

The following BNF defines the general syntax for a multiplicity string, including support order and uniqueness designators:

```
<multiplicity> ::= <multiplicity-range> [ [ ‘{’ <order-designator> [ ‘,’ <uniqueness-designator> ] ‘}’ ] | [ ‘{’ <uniqueness-designator> [ ‘,’ <order-designator> ] ‘}’ ] ]
```

```
<multiplicity-range> ::= [ <lower> ‘.’ ] <upper>
```

```
<lower> ::= <value-specification>
```

$\langle upper \rangle ::= \langle value-specification \rangle$

$\langle order-designator \rangle ::= 'ordered' \mid 'unordered'$

$\langle uniqueness-designator \rangle ::= 'unique' \mid 'nonunique'$

See also Clause 8 on the textual notation for ValueSpecifications.

## 7.5.5 Examples

Figure 7.11 shows two multiplicity strings as part of attribute specifications within a class symbol.

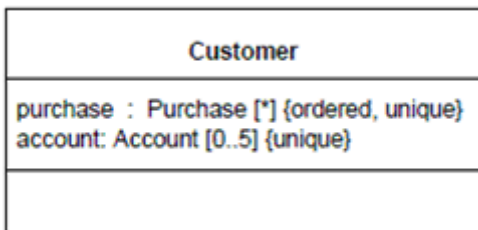


Figure 7.11 – Multiplicity within a textual specification

Figure 7.12 shows two multiplicity strings as part of the specification of two association ends.



Figure 7.12 Multiplicity as an adornment to a symbol

## 7.6 Constraints

### 7.6.1 Summary

A Constraint is an assertion that indicates a restriction that must be satisfied by any valid realization of the model containing the Constraint. A Constraint is attached to a set of constrainedElements, and it represents additional semantic information about those Elements.

## 7.6.2 Abstract Syntax

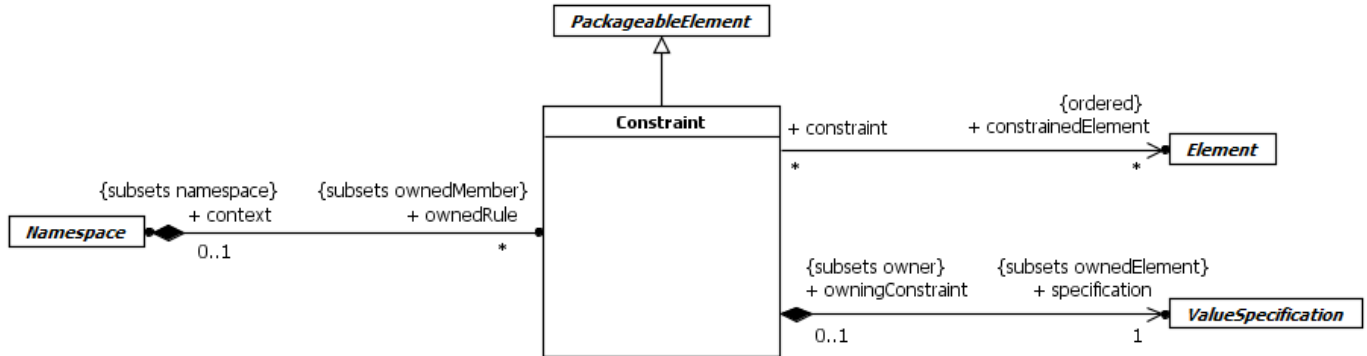


Figure 7.13 Abstract Syntax of Constraints

## 7.6.3 Semantics

The specification of a Constraint is given by a ValueSpecification (see Clause 8) of type Boolean. The computation of the specification may reference the constrainedElements of the Constraint and also the context of the Constraint. In addition, the context of the Constraint 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).

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 determines when the Constraint specification is evaluated. For example, a Constraint that is a precondition of an Operation is evaluated at the start of the invocation of the Operation, while a Constraint that is a postcondition is evaluated at the conclusion of the invocation (see sub clause 9.6 on Operations).

A Constraint is evaluated by evaluating its specification. If the specification evaluates to true, then the Constraint is satisfied at that time. If the specification evaluates to false, then the Constraint is not satisfied, and the realization of the model in which the evaluation occurs is not valid.

## 7.6.4 Notation

Certain kinds of Constraints are predefined in UML, others may be user-defined. The specification of a user-defined Constraint is often expressed as a text string in some language, whose syntax and interpretation is as defined by that language. In some situations, a formal language (such as OCL) or a programming language (such as Java) may be appropriate, in other situations natural language may be used. Such a specification may be represented as an OpaqueExpression with the appropriate language and body (see sub clause 8.3). The Constraint may then be notated textually within braces ({} according to the following BNF:

```
<constraint> ::= '{' [ <name> ':' ] <boolean-expression> '}'
```

where <name> is the name of the Constraint and <boolean-expression> is the appropriate textual notation for the Constraint specification.

Most generally, the constraint string is placed in a note symbol and attached to each of the symbols for the constrainedElements by dashed lines. (See Figure 7.14 for an example.)

For a Constraint that applies to a single constrainedElement (such as a single Class or Association), the constraint string may be directly placed near the symbol for the constrainedElement, preferably near the name, if any. A tool shall make it possible to determine the constrainedElement.

For an Element whose notation is a text string (such as an attribute, etc.), the constraint string may follow the Element text string. The Element so annotated is then the single constrainedElement of the Constraint. (Figure 7.15 shows a Constraint string that follows an attribute within a Class symbol.)

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. (See Figure 7.16 for an example.)

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 constrainedElement collection.

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

### 7.6.5 Examples

Figure 7.14 shows an example of a Constraint in a note symbol.

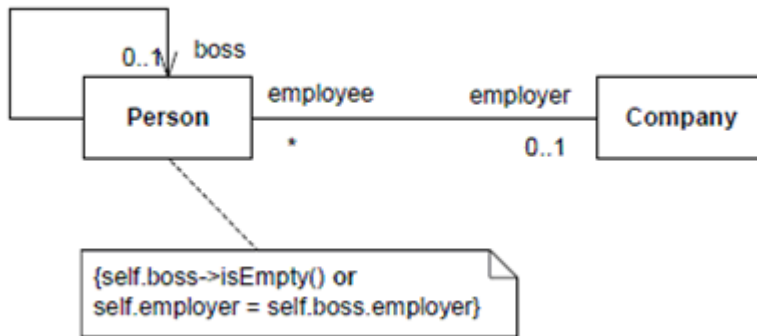


Figure 7.14 Constraint in a note symbol

Figure 7.15 shows a constraint string attached to an attribute.

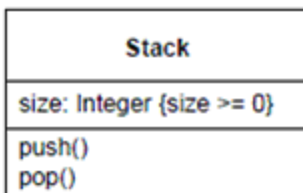


Figure 7.15 Constraint attached to an attribute

Figure 7.16 shows an {xor} constraint between two associations.

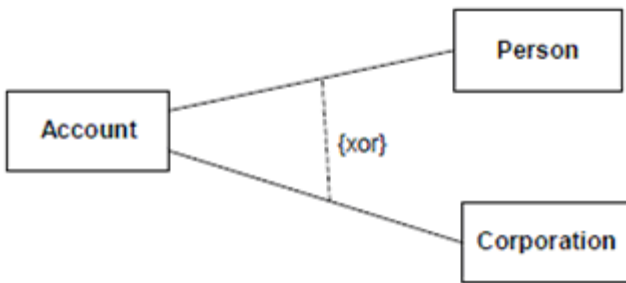


Figure 7.16 {xor} constraint

## 7.7 Dependencies

### 7.7.1 Summary

A Dependency signifies a supplier/client relationship between model elements where the modification of a supplier may impact the client model elements.

### 7.7.2 Abstract Syntax

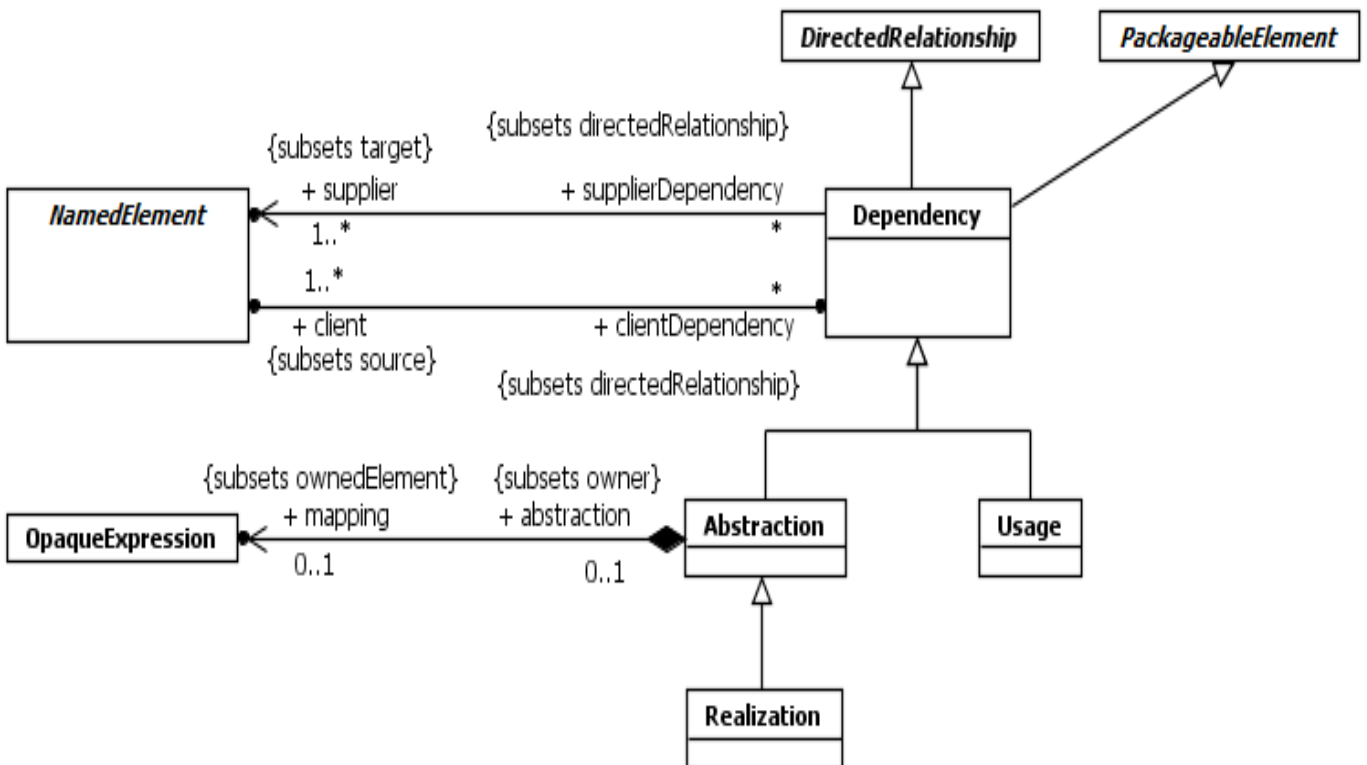


Figure 7.17 Abstract syntax of dependencies

### 7.7.3 Semantics

#### Dependency

A Dependency implies that the semantics of the clients are not complete without the suppliers. The presence of Dependency relationships in a model does not have any runtime semantic implications. The semantics are all given in terms of the NamedElements that participate in the relationship, not in terms of their instances.

#### Usage

A Usage is a Dependency in which one NamedElement requires another NamedElement (or set of NamedElements) for its full implementation or operation. The Usage does not specify how the client uses the supplier other than the fact that the supplier is used by the definition or implementation of the client.

#### Abstraction

An Abstraction is a Dependency that relates two NamedElements or sets of NamedElements that represent the same concept at different levels of abstraction or from different viewpoints. The relationship may be defined as a mapping between the suppliers and the clients. Depending on the specific stereotype of Abstraction, the mapping may be formal or informal, and it may be unidirectional or bidirectional. Abstraction has predefined stereotypes (such as «derive», «refine», and «trace») that are defined in the Standard Profile (see Clause 22). If an Abstraction has more than one client, the supplier maps into the set of clients as a group. For example, an analysis-level Class might be split into several design-level Classes. The situation is similar if there is more than one supplier.

#### Realization

Realization is a specialized Abstraction dependency between two sets of NamedElements, one representing a specification (the supplier) and the other representing an implementation of that specification (the client). Realization can be used to model stepwise refinement, optimizations, transformations, templates, model synthesis, framework composition, etc. A Realization signifies that the set of clients is an implementation of the set of suppliers, which serves as the specification. The meaning of “implementation” is not strictly defined, but rather implies a more refined or elaborate form in respect to a certain modeling context. It is possible to specify a mapping between the specification and implementation elements, although this is not necessarily computable.

### 7.7.4 Notation

A Dependency is shown as a dashed arrow between two model Elements. The model Element at the tail of the arrow (the client) depends on the model Element at the arrowhead (the supplier). The arrow may be labeled with an optional keyword or stereotype and an optional name (see Figure 7.18).

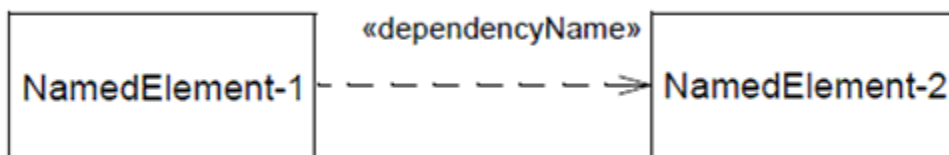


Figure 7.18 Notation for a Dependency between two elements

It is possible to have a set of Elements for the client or supplier. In this case, one or more arrows with their tails on the clients are connected to the tails of one or more arrows with their heads on the suppliers. A small dot can be placed on the junction if desired. A note on the Dependency should be attached at the junction point.

A Usage is shown as a Dependency with a «use» keyword attached to it.

An Abstraction is shown as a Dependency with an «abstraction» keyword or the specific predefined stereotype attached to it. A Realization is shown as a dashed line with a triangular arrowhead at the end that corresponds to the realized Element.

### 7.7.5 Examples

In Figure 7.19, the Car Class has a Dependency on the CarFactory Class. In this case, the Dependency is an instantiate Dependency: an instance of the CarFactory Class creates instances of the Car Class.

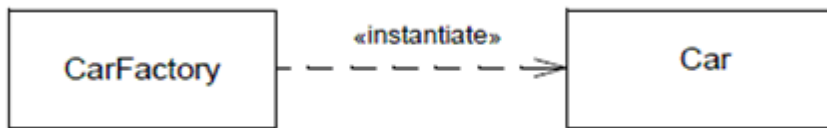


Figure 7.19 An example of an «instantiate» Dependency

In Figure 7.20, an Order Class requires the Line Item Class for its full implementation.

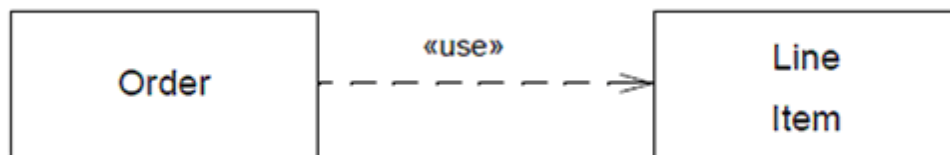


Figure 7.20 – An example of a «use» Dependency

Figure 7.21 illustrates an example in which the Business class is realized by a combination of Owner and Employee classes.

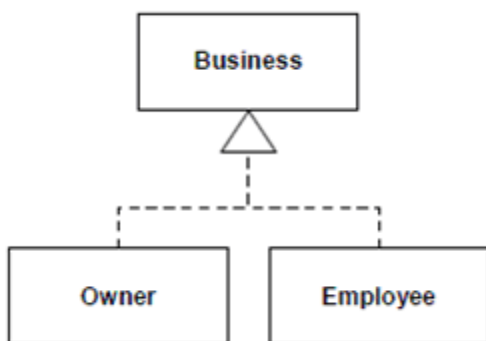


Figure 7.21 – An example of a realization Dependency



## 7.8 Classifier Descriptions

### Abstraction [Class]

#### Description

An Abstraction is a Relationship that relates two Elements or sets of Elements that represent the same concept at different levels of abstraction or from different viewpoints.

#### Diagrams

[Dependencies](#), [Artifacts](#)

#### Generalizations

[Dependency](#)

#### Specializations

[Realization](#), [Manifestation](#)

#### Association Ends

- ◆ mapping : [OpaqueExpression](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A mapping abstraction::abstraction](#))  
An OpaqueExpression that states the abstraction relationship between the supplier(s) and the client(s). In some cases, such as derivation, it is usually formal and unidirectional; in other cases, such as trace, it is usually informal and bidirectional. The mapping expression is optional and may be omitted if the precise relationship between the Elements is not specified.

### Comment [Class]

#### Description

A Comment is a textual annotation that can be attached to a set of Elements.

#### Diagrams

[Root](#)

#### Generalizations

[Element](#)

#### Attributes

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

## Association Ends

- annotatedElement : [Element](#) [0..\*] (opposite [A\\_annotatedElement\\_comment::comment](#))  
References the Element(s) being commented.

## Constraint [Class]

### Description

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 or set of Elements.

### Diagrams

[Namespaces](#), [Constraints](#), [Intervals](#), [Use Cases](#), [Behavior State Machines](#), [Protocol State Machines](#), [Interactions](#), [Fragments](#), [Behaviors](#), [Features](#), [Operations](#), [Actions](#)

### Generalizations

[PackageableElement](#)

### Specializations

[IntervalConstraint](#), [InteractionConstraint](#)

## Association Ends

- constrainedElement : [Element](#) [0..\*]{ordered} (opposite [A\\_constrainedElement\\_constraint::constraint](#))  
The ordered set of Elements referenced by this Constraint.
- context : [Namespace](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [Namespace::ownedRule](#))  
Specifies the Namespace that owns the Constraint.
- ♦ specification : [ValueSpecification](#) [1..1]{subsets [Element::ownedElement](#)} (opposite [A\\_specification\\_owningConstraint::owningConstraint](#))  
A condition that must be true when evaluated in order for the Constraint to be satisfied.

## Constraints

- boolean\_value  
The ValueSpecification for a Constraint must evaluate to a Boolean value.  
  
Cannot be expressed in OCL
- no\_side\_effects  
Evaluating the ValueSpecification for a Constraint must not have side effects.  
  
Cannot be expressed in OCL

- `not_apply_to_self`  
A Constraint cannot be applied to itself.

```
inv: not constrainedElement->includes(self)
```

## Dependency [Class]

### Description

A Dependency is a Relationship that signifies that a single or a set of model Elements requires other model Elements for their specification or implementation. This means that the complete semantics of the client Element(s) are either semantically or structurally dependent on the definition of the supplier Element(s).

### Diagrams

[Dependencies](#), [Collaborations](#), [Deployments](#)

### Generalizations

[DirectedRelationship](#), [PackageableElement](#)

### Specializations

[Abstraction](#), [Usage](#), [Deployment](#)

### Association Ends

- `client` : [NamedElement](#) [1..\*]{subsets [DirectedRelationship::source](#)} (opposite [NamedElement::clientDependency](#))  
The Element(s) dependent on the supplier Element(s). In some cases (such as a trace Abstraction) the assignment of direction (that is, the designation of the client Element) is at the discretion of the modeler and is a stipulation.
- `supplier` : [NamedElement](#) [1..\*]{subsets [DirectedRelationship::target](#)} (opposite [A\\_supplier\\_supplierDependency::supplierDependency](#))  
The Element(s) on which the client Element(s) depend in some respect. The modeler may stipulate a sense of Dependency direction suitable for their domain.

## DirectedRelationship [Abstract Class]

### Description

A DirectedRelationship represents a relationship between a collection of source model Elements and a collection of target model Elements.

### Diagrams

[Root](#), [Template Bindings](#), [Namespaces](#), [Dependencies](#), [Use Cases](#), [Packages](#), [Profiles](#), [Information Flows](#), [Classifiers](#)

### Generalizations

[Relationship](#)

## Specializations

[Dependency](#), [ElementImport](#), [PackageImport](#), [TemplateBinding](#), [Extend](#), [Include](#), [ProtocolConformance](#), [PackageMerge](#), [ProfileApplication](#), [InformationFlow](#), [Generalization](#)

## Association Ends

- /source : [Element](#) [1..\*]{union, subsets [Relationship::relatedElement](#)} (opposite [A\\_source\\_directedRelationship::directedRelationship](#))  
Specifies the source Element(s) of the DirectedRelationship.
- /target : [Element](#) [1..\*]{union, subsets [Relationship::relatedElement](#)} (opposite [A\\_target\\_directedRelationship::directedRelationship](#))  
Specifies the target Element(s) of the DirectedRelationship.

## Element [Abstract Class]

### Description

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

### Diagrams

[Root](#), [Template Bindings](#), [Templates](#), [Namespaces](#), [Types](#), [Constraints](#), [Activity Groups](#), [Executable Nodes](#), [Profiles](#), [Instances](#), [Link End Data](#), [Structured Actions](#)

## Specializations

[Comment](#), [MultiplicityElement](#), [NamedElement](#), [ParameterableElement](#), [Relationship](#), [TemplateableElement](#), [TemplateParameter](#), [TemplateParameterSubstitution](#), [TemplateSignature](#), [ExceptionHandler](#), [Image](#), [Slot](#), [Clause](#), [LinkEndData](#), [QualifierValue](#)

## Association Ends

- ♦ ownedComment : [Comment](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [A\\_ownedComment\\_owningElement::owningElement](#))  
The Comments owned by this Element.
- ♦ /ownedElement : [Element](#) [0..\*]{union} (opposite [Element::owner](#))  
The Elements owned by this Element.
- /owner : [Element](#) [0..1]{union} (opposite [Element::ownedElement](#))  
The Element that owns this Element.

## Operations

- allOwnedElements() : [Element](#) [0..\*]  
The query allOwnedElements() gives all of the direct and indirect ownedElements of an Element.

```
body: ownedElement->union(ownedElement->collect(e | e.allOwnedElements()))->asSet()
```

- `mustBeOwned()` : [Boolean](#)  
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.

```
body: true
```

## Constraints

- `has_owner`  
Elements that must be owned must have an owner.

```
inv: mustBeOwned() implies owner->notEmpty()
```

- `not_own_self`  
An element may not directly or indirectly own itself.

```
inv: not allOwnedElements()->includes(self)
```

## ElementImport [Class]

### Description

An `ElementImport` identifies an `Element` in a different `Namespace`, and allows the `Element` to be referenced using its name without a qualifier in the `Namespace` owning the `ElementImport`.

### Diagrams

[Namespaces](#), [Profiles](#)

### Generalizations

[DirectedRelationship](#)

### Attributes

- `alias` : [String](#) [0..1]  
Specifies the name that should be added to the importing `Namespace` in lieu of the name of the imported `PackageableElement`. The alias must not clash with any other member in the importing `Namespace`. By default, no alias is used.
- `visibility` : [VisibilityKind](#) [1..1] = public  
Specifies the visibility of the imported `PackageableElement` within the importing `Namespace`, i.e., whether the imported `Element` will in turn be visible to other `Namespaces`. If the `ElementImport` is public, the imported `Element` will be visible outside the importing `Namespace` while, if the `ElementImport` is private, it will not.

### Association Ends

- `importedElement` : [PackageableElement](#) [1..1]{subsets [DirectedRelationship::target](#)} (opposite [A importedElement import::import](#))

Specifies the PackageableElement whose name is to be added to a Namespace.

- importingNamespace : [Namespace](#) [1..1]{subsets [DirectedRelationship::source](#), subsets [Element::owner](#)} (opposite [Namespace::elementImport](#))  
Specifies the Namespace that imports a PackageableElement from another Namespace.

## Operations

- getName() : [String](#)  
The query getName() returns the name under which the imported PackageableElement will be known in the importing namespace.

```
body: if alias->notEmpty() then
  alias
else
  importedElement.name
endif
```

## Constraints

- imported\_element\_is\_public  
An importedElement has either public visibility or no visibility at all.

```
inv: importedElement.visibility <> null implies importedElement.visibility = VisibilityKind::public
```

- visibility\_public\_or\_private  
The visibility of an ElementImport is either public or private.

```
inv: visibility = VisibilityKind::public or visibility = VisibilityKind::private
```

## MultiplicityElement [Abstract Class]

### Description

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 MultiplicityElement embeds this information to specify the allowable cardinalities for an instantiation of the Element.

### Diagrams

[Types](#), [Activities](#), [Structured Classifiers](#), [Features](#), [Actions](#)

### Generalizations

[Element](#)

### Specializations

[Variable](#), [ConnectorEnd](#), [Parameter](#), [StructuralFeature](#), [Pin](#)

## Attributes

- isOrdered : [Boolean](#) [1..1] = false  
For a multivalued multiplicity, this attribute specifies whether the values in an instantiation of this MultiplicityElement are sequentially ordered.
- isUnique : [Boolean](#) [1..1] = true  
For a multivalued multiplicity, this attributes specifies whether the values in an instantiation of this MultiplicityElement are unique.
- /lower : [Integer](#) [0..1] = 1  
The lower bound of the multiplicity interval.
- /upper : [UnlimitedNatural](#) [0..1] = 1  
The upper bound of the multiplicity interval.

## Association Ends

- ♦ lowerValue : [ValueSpecification](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_lowerValue\\_owningLower::owningLower](#))  
The specification of the lower bound for this multiplicity.
- ♦ upperValue : [ValueSpecification](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_upperValue\\_owningUpper::owningUpper](#))  
The specification of the upper bound for this multiplicity.

## Operations

- compatibleWith(other : [MultiplicityElement](#)) : [Boolean](#)  
The operation compatibleWith takes another multiplicity as input. It checks if one multiplicity is compatible with another.  
  

```
body: (other.lowerBound() <= self.lowerBound()) and ((other.upperBound() = *) or (self.upperBound() <= other.upperBound()))
```
- includesMultiplicity(M : [MultiplicityElement](#)) : [Boolean](#)  
The query includesMultiplicity() checks whether this multiplicity includes all the cardinalities allowed by the specified multiplicity.  
  

```
pre: self.upperBound()->notEmpty() and self.lowerBound()->notEmpty() and M.upperBound()->notEmpty() and M.lowerBound()->notEmpty()
body: (self.lowerBound() <= M.lowerBound()) and (self.upperBound() >= M.upperBound())
```
- is(lowerbound : [Integer](#), upperbound : [UnlimitedNatural](#)) : [Boolean](#)  
The operation is determines if the upper and lower bound of the ranges are the ones given.  
  

```
body: lowerbound = self.lowerBound() and upperbound = self.upperBound()
```

- `isMultivalued()` : [Boolean](#)  
The query `isMultivalued()` checks whether this multiplicity has an upper bound greater than one.

```
pre: upperBound()->notEmpty()
body: upperBound() > 1
```

- `lower()` : [Integer](#) [0..1]  
The derived lower attribute must equal the `lowerBound`.

```
body: lowerBound()
```

- `lowerBound()` : [Integer](#) [0..1]  
The query `lowerBound()` returns the lower bound of the multiplicity as an integer, which is the `integerValue` of `lowerValue`, if this is given, and 1 otherwise.

```
body: if lowerValue=null then 1 else lowerValue.integerValue() endif
```

- `upper()` : [UnlimitedNatural](#) [0..1]  
The derived upper attribute must equal the `upperBound`.

```
body: upperBound()
```

- `upperBound()` : [UnlimitedNatural](#) [0..1]  
The query `upperBound()` returns the upper bound of the multiplicity for a bounded multiplicity as an unlimited natural, which is the `unlimitedNaturalValue` of `upperValue`, if given, and 1, otherwise.

```
body: if upperValue=null then 1 else upperValue.unlimitedValue() endif
```

## Constraints

- `upper_ge_lower`  
The upper bound must be greater than or equal to the lower bound.

```
inv: (upperBound() <> null and lowerBound() <> null) implies upperBound() >= lowerBound()
```

- `lower_ge_0`  
The lower bound must be a non-negative integer literal.

```
inv: lowerBound() <> null implies lowerBound() >= 0
```

- `value_specification_no_side_effects`  
If a non-literal `ValueSpecification` is used for `lowerValue` or `upperValue`, then evaluating that specification must not have side effects.

Cannot be expressed in OCL

- `value_specification_constant`  
If a non-literal `ValueSpecification` is used for `lowerValue` or `upperValue`, then that specification must be a constant expression.

Cannot be expressed in OCL



## NamedElement [Abstract Class]

### Description

A NamedElement is an Element in a model that may have a name. The name may be given directly and/or via the use of a StringExpression.

### Diagrams

[Namespaces](#), [Types](#), [Dependencies](#), [Activity Groups](#), [Time](#), [Use Cases](#), [Collaborations](#), [Behavior State Machines](#), [Interactions](#), [Messages](#), [Lifelines](#), [Occurrences](#), [Fragments](#), [Information Flows](#), [Deployments](#), [Events](#), [Classifiers](#)

### Generalizations

[Element](#)

### Specializations

[Namespace](#), [PackageableElement](#), [TypedElement](#), [ActivityGroup](#), [Trigger](#), [Extend](#), [Include](#), [CollaborationUse](#), [Vertex](#), [GeneralOrdering](#), [InteractionFragment](#), [Lifeline](#), [Message](#), [MessageEnd](#), [DeployedArtifact](#), [DeploymentTarget](#), [ParameterSet](#), [RedefinableElement](#)

### Attributes

- name : [String](#) [0..1]  
The name of the NamedElement.
- /qualifiedName : [String](#) [0..1]  
A name that 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.
- visibility : [VisibilityKind](#) [0..1]  
Determines whether and how the NamedElement is visible outside its owning Namespace.

### Association Ends

- clientDependency : [Dependency](#) [0..\*]{subsets [A\\_source\\_directedRelationship::directedRelationship](#)} (opposite [Dependency::client](#))  
Indicates the Dependencies that reference this NamedElement as a client.
- ♦ nameExpression : [StringExpression](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_nameExpression\\_namedElement::namedElement](#))  
The StringExpression used to define the name of this NamedElement.
- /namespace : [Namespace](#) [0..1]{union, subsets [A\\_member\\_memberNamespace::memberNamespace](#), subsets [Element::owner](#)} (opposite [Namespace::ownedMember](#))  
Specifies the Namespace that owns the NamedElement.

## Operations

- `allNamespaces() : Namespace [0..*]`  
The query `allNamespaces()` gives the sequence of `Namespaces` in which the `NamedElement` is nested, working outwards.

```
body: if namespace->isEmpty()
then OrderedSet{}
else namespace.allNamespaces()->prepend(namespace)
endif
```

- `allOwningPackages() : Package [0..*]`  
The query `allOwningPackages()` returns all the `Packages` that in which this `NamedElement` is directly or indirectly contained, without any intervening `Namespace` that is not a `Package`.

```
body: if namespace.oclIsKindOf(Package)
then
  let owningPackage : Package = namespace.oclAsType(Package) in
  owningPackage->union(owningPackage.allOwningPackages())
else
  null
endif
```

- `isDistinguishableFrom(n : NamedElement, ns : Namespace) : Boolean`  
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 types neither of which is a kind of the other or (b) they have different names.

```
body: (self.oclIsKindOf(n.oclType()) or n.oclIsKindOf(self.oclType())) implies
  ns.getNamesOfMember(self)->intersection(ns.getNamesOfMember(n))->isEmpty()
```

- `qualifiedName() : String`  
When a `NamedElement` has a name, and all of its containing `Namespaces` have a name, the `qualifiedName` is constructed from the name of the `NamedElement` and the names of the containing `Namespaces`.

```
body: if self.name <> null and self.allNamespaces()->select( ns | ns.name=null )->isEmpty()
then
  self.allNamespaces()->iterate( ns : Namespace; agg: String = self.name |
  ns.name.concat(self.separator()).concat(agg))
else
  null
endif
```

- `separator() : String`  
The query `separator()` gives the string that is used to separate names when constructing a `qualifiedName`.

```
body: '::'
```

## Constraints

- `visibility_needs_ownership`  
If a `NamedElement` is owned by something other than a `Namespace`, it does not have a visibility. One that is not owned by anything (and hence must be a `Package`, as this is the only kind of `NamedElement` that overrides `mustBeOwned()`) may have a visibility.

```
inv: (namespace = null and owner <> null) implies visibility = null
```

- **has\_qualified\_name**  
When there is a name, and all of the containing Namespaces have a name, the qualifiedName is constructed from the name of the NamedElement and the names of the containing Namespaces.

```
inv: (name <> null and allNamespaces()->select(ns | ns.name = null)->isEmpty()) implies  
qualifiedName = allNamespaces()->iterate( ns : Namespace; agg: String = name |  
ns.name.concat(self.separator()).concat(agg))
```

- **has\_no\_qualified\_name**  
If there is no name, or one of the containing Namespaces has no name, there is no qualifiedName.

```
inv: name=null or allNamespaces()->select( ns | ns.name=null )->notEmpty() implies qualifiedName =  
null
```

## Namespace [Abstract Class]

### Description

A Namespace is an Element in a model that owns and/or imports a set of NamedElements that can be identified by name.

### Diagrams

[Namespaces](#), [Constraints](#), [Behavior State Machines](#), [Packages](#), [Fragments](#), [Classifiers](#), [Features](#), [Structured Actions](#)

### Generalizations

[NamedElement](#)

### Specializations

[Region](#), [State](#), [Transition](#), [Package](#), [InteractionOperand](#), [BehavioralFeature](#), [Classifier](#), [StructuredActivityNode](#)

### Association Ends

- ♦ elementImport : [ElementImport](#) [0..\*]{subsets [Element::ownedElement](#), subsets [A\\_source\\_directedRelationship::directedRelationship](#)} (opposite [ElementImport::importingNamespace](#))  
References the ElementImports owned by the Namespace.
- /importedMember : [PackageableElement](#) [0..\*]{subsets [Namespace::member](#)} (opposite [A\\_importedMember\\_namespace::namespace](#))  
References the PackageableElements that are members of this Namespace as a result of either PackageImports or ElementImports.
- /member : [NamedElement](#) [0..\*]{union} (opposite [A\\_member\\_memberNamespace::memberNamespace](#))  
A collection of NamedElements identifiable within the Namespace, either by being owned or by being introduced by importing or inheritance.
- ♦ /ownedMember : [NamedElement](#) [0..\*]{union, subsets [Namespace::member](#), subsets [Element::ownedElement](#)} (opposite [NamedElement::namespace](#))  
A collection of NamedElements owned by the Namespace.

- ♦ ownedRule : [Constraint](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [Constraint::context](#))  
Specifies a set of Constraints owned by this Namespace.
- ♦ packageImport : [PackageImport](#) [0..\*]{subsets [Element::ownedElement](#), subsets [A\\_source\\_directedRelationship::directedRelationship](#)} (opposite [PackageImport::importingNamespace](#))  
References the PackageImports owned by the Namespace.

## Operations

- excludeCollisions(imps : [PackageableElement](#) [0..\*]) : [PackageableElement](#) [0..\*]  
The query excludeCollisions() excludes from a set of PackageableElements any that would not be distinguishable from each other in this Namespace.

```
body: imps->reject(imp1 | imps->exists(imp2 | not imp1.isDistinguishableFrom(imp2, self)))
```

- getNamesOfMember(element : [NamedElement](#)) : [String](#) [0..\*]  
The query getNamesOfMember() gives a set of all of the names that a member would have in a Namespace, taking importing into account. In general a member can have multiple names in a Namespace if it is imported more than once with different aliases.

```
body: if self.ownedMember ->includes(element)
then Set{element.name}
else let elementImports : Set(ElementImport) = self.elementImport->select(ei | ei.importedElement =
element) in
  if elementImports->notEmpty()
  then
    elementImports->collect(el | el.getName())->asSet()
  else
    self.packageImport->select(pi | pi.importedPackage.visibleMembers().oclAsType(NamedElement)-
>includes(element))-> collect(pi | pi.importedPackage.getNamesOfMember(element))->asSet()
  endif
endif
```

- importMembers(imps : [PackageableElement](#) [0..\*]) : [PackageableElement](#) [0..\*]  
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 ownedMembers, and it also excludes PackageableElements that would have the same name as each other when imported.

```
body: self.excludeCollisions(imps)->select(imp | self.ownedMember->forall(mem |
imp.isDistinguishableFrom(mem, self)))
```

- importedMember() : [PackageableElement](#) [0..\*]  
The importedMember property is derived as the PackageableElements that are members of this Namespace as a result of either PackageImports or ElementImports.

```
body: self.importMembers(elementImport.importedElement->asSet()->union(packageImport.importedPackage-
>collect(p | p.visibleMembers()))->asSet()
```

- membersAreDistinguishable() : [Boolean](#)  
The Boolean query membersAreDistinguishable() determines whether all of the Namespace's members are distinguishable within it.

```
body: member->forall( memb |
```

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

## Constraints

- members\_distinguishable  
All the members of a Namespace are distinguishable within it.

```
inv: membersAreDistinguishable()
```

- cannot\_import\_self  
A Namespace cannot have a PackageImport to itself.

```
inv: packageImport.importedPackage.oclassType(Namespace)->excludes(self)
```

- cannot\_import\_ownedMembers  
A Namespace cannot have an ElementImport to one of its ownedMembers.

```
inv: elementImport.importedElement.oclassType(Element)->excludesAll(ownedMember)
```

## PackageImport [Class]

### Description

A PackageImport is a Relationship that imports all the non-private members of a Package into the Namespace owning the PackageImport, so that those Elements may be referred to by their unqualified names in the importingNamespace.

### Diagrams

[Namespaces](#), [Profiles](#)

### Generalizations

[DirectedRelationship](#)

### Attributes

- visibility : [VisibilityKind](#) [1..1] = public  
Specifies the visibility of the imported PackageableElements within the importingNamespace, i.e., whether imported Elements will in turn be visible to other Namespaces. If the PackageImport is public, the imported Elements will be visible outside the importingNamespace, while, if the PackageImport is private, they will not.

### Association Ends

- importedPackage : [Package](#) [1..1]{subsets [DirectedRelationship::target](#)} (opposite [A\\_importedPackage\\_packageImport::packageImport](#))  
Specifies the Package whose members are imported into a Namespace.
- importingNamespace : [Namespace](#) [1..1]{subsets [DirectedRelationship::source](#), subsets [Element::owner](#)} (opposite [Namespace::packageImport](#))

Specifies the Namespace that imports the members from a Package.

## Constraints

- `public_or_private`  
The visibility of a `PackageImport` is either public or private.

```
inv: visibility = VisibilityKind::public or visibility = VisibilityKind::private
```

## PackageableElement [Abstract Class]

### Description

A `PackageableElement` is a `NamedElement` that may be owned directly by a `Package`. A `PackageableElement` is also able to serve as the `parameteredElement` of a `TemplateParameter`.

### Diagrams

[Namespaces](#), [Types](#), [Constraints](#), [Dependencies](#), [Literals](#), [Time](#), [Components](#), [Packages](#), [Information Flows](#), [Deployments](#), [Artifacts](#), [Events](#), [Instances](#), [Generalization Sets](#)

### Generalizations

[ParameterableElement](#), [NamedElement](#)

### Specializations

[Constraint](#), [Dependency](#), [Type](#), [Event](#), [Observation](#), [ValueSpecification](#), [Package](#), [InformationFlow](#), [GeneralizationSet](#), [InstanceSpecification](#)

### Attributes

- `visibility` : [VisibilityKind](#) [0..1] = public  
A `PackageableElement` must have a visibility specified if it is owned by a `Namespace`. The default visibility is public.

### Constraints

- `namespace_needs_visibility`  
A `PackageableElement` owned by a `Namespace` must have a visibility.

```
inv: visibility = null implies namespace = null
```

## ParameterableElement [Abstract Class]

### Description

A `ParameterableElement` is an `Element` that can be exposed as a formal `TemplateParameter` for a template, or specified as an actual parameter in a binding of a template.

## Diagrams

[Template Bindings](#), [Templates](#), [Namespaces](#), [Structured Classifiers](#), [Properties](#), [Operations](#)

## Generalizations

[Element](#)

## Specializations

[PackageableElement](#), [ConnectableElement](#), [Operation](#)

## Association Ends

- owningTemplateParameter : [TemplateParameter](#) [0..1]{subsets [ParameterableElement::templateParameter](#), subsets [Element::owner](#)} (opposite [TemplateParameter::ownedParameteredElement](#))  
The formal TemplateParameter that owns this ParameterableElement.
- templateParameter : [TemplateParameter](#) [0..1] (opposite [TemplateParameter::parameteredElement](#))  
The TemplateParameter that exposes this ParameterableElement as a formal parameter.

## Operations

- isCompatibleWith(p : [ParameterableElement](#)) : [Boolean](#)  
The query isCompatibleWith() determines if this ParameterableElement is compatible with the specified ParameterableElement. By default, ParameterableElement P is compatible with ParameterableElement Q if the kind of P is the same as or a subtype of the kind of Q. Subclasses should override this operation to specify different compatibility constraints.

```
body: p->oclIsKindOf(self.oclType())
```

- isTemplateParameter() : [Boolean](#)  
The query isTemplateParameter() determines if this ParameterableElement is exposed as a formal TemplateParameter.

```
body: templateParameter->notEmpty()
```

## Realization [Class]

### Description

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

### Diagrams

[Dependencies](#), [Components](#), [Interfaces](#), [Classifiers](#)

### Generalizations

[Abstraction](#)

## Specializations

[ComponentRealization](#), [InterfaceRealization](#), [Substitution](#)

## Relationship [Abstract Class]

### Description

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

### Diagrams

[Root](#), [Associations](#), [Information Flows](#)

### Generalizations

[Element](#)

### Specializations

[DirectedRelationship](#), [Association](#)

### Association Ends

- /relatedElement : [Element](#) [1..\*]{union} (opposite [A\\_relatedElement\\_relationship::relationship](#))  
Specifies the elements related by the Relationship.

## TemplateBinding [Class]

### Description

A TemplateBinding is a DirectedRelationship between a TemplateableElement and a template. A TemplateBinding specifies the TemplateParameterSubstitutions of actual parameters for the formal parameters of the template.

### Diagrams

[Template Bindings](#)

### Generalizations

[DirectedRelationship](#)

### Association Ends

- boundElement : [TemplateableElement](#) [1..1]{subsets [DirectedRelationship::source](#), subsets [Element::owner](#)} (opposite [TemplateableElement::templateBinding](#))  
The TemplateableElement that is bound by this TemplateBinding.
- ♦ parameterSubstitution : [TemplateParameterSubstitution](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [TemplateParameterSubstitution::templateBinding](#))  
The TemplateParameterSubstitutions owned by this TemplateBinding.



- signature : [TemplateSignature](#) [1..1]{subsets [DirectedRelationship::target](#)} (opposite [A\\_signature\\_templateBinding::templateBinding](#))  
The TemplateSignature for the template that is the target of this TemplateBinding.

## Constraints

- parameter\_substitution\_formal  
Each parameterSubstitution must refer to a formal TemplateParameter of the target TemplateSignature.  
  
`inv: parameterSubstitution->forall(b | signature.parameter->includes(b.formal))`
- one\_parameter\_substitution  
A TemplateBinding contains at most one TemplateParameterSubstitution for each formal TemplateParameter of the target TemplateSignature.  
  
`inv: signature.parameter->forall(p | parameterSubstitution->select(b | b.formal = p)->size() <= 1)`

## TemplateParameter [Class]

### Description

A TemplateParameter exposes a ParameterableElement as a formal parameter of a template.

### Diagrams

[Template Bindings](#), [Templates](#), [Structured Classifiers](#), [Classifier Templates](#), [Operations](#)

### Generalizations

[Element](#)

### Specializations

[ConnectableElementTemplateParameter](#), [ClassifierTemplateParameter](#), [OperationTemplateParameter](#)

### Association Ends

- default : [ParameterableElement](#) [0..1] (opposite [A\\_default\\_templateParameter::templateParameter](#))  
The ParameterableElement that is the default for this formal TemplateParameter.
- ◆ ownedDefault : [ParameterableElement](#) [0..1]{subsets [Element::ownedElement](#), subsets [TemplateParameter::default](#)} (opposite [A\\_ownedDefault\\_templateParameter::templateParameter](#))  
The ParameterableElement that is owned by this TemplateParameter for the purpose of providing a default.
- ◆ ownedParameteredElement : [ParameterableElement](#) [0..1]{subsets [Element::ownedElement](#), subsets [TemplateParameter::parameteredElement](#)} (opposite [ParameterableElement::owningTemplateParameter](#))  
The ParameterableElement that is owned by this TemplateParameter for the purpose of exposing it as the parameteredElement.

- parameteredElement : [ParameterableElement](#) [1..1] (opposite [ParameterableElement::templateParameter](#))  
The ParameterableElement exposed by this TemplateParameter.
- signature : [TemplateSignature](#) [1..1]{subsets [A\\_parameter\\_templateSignature::templateSignature](#), subsets [Element::owner](#)} (opposite [TemplateSignature::ownedParameter](#))  
The TemplateSignature that owns this TemplateParameter.

## Constraints

- must\_be\_compatible  
The default must be compatible with the formal TemplateParameter.

```
inv: default <> null implies default.isCompatibleWith(parameteredElement)
```

## TemplateParameterSubstitution [Class]

### Description

A TemplateParameterSubstitution relates the actual parameter to a formal TemplateParameter as part of a template binding.

### Diagrams

[Template Bindings](#)

### Generalizations

[Element](#)

### Association Ends

- actual : [ParameterableElement](#) [1..1] (opposite [A\\_actual\\_templateParameterSubstitution::templateParameterSubstitution](#))  
The ParameterableElement that is the actual parameter for this TemplateParameterSubstitution.
- formal : [TemplateParameter](#) [1..1] (opposite [A\\_formal\\_templateParameterSubstitution::templateParameterSubstitution](#))  
The formal TemplateParameter that is associated with this TemplateParameterSubstitution.
- ♦ ownedActual : [ParameterableElement](#) [0..1]{subsets [Element::ownedElement](#), subsets [TemplateParameterSubstitution::actual](#)} (opposite [A\\_ownedActual\\_owningTemplateParameterSubstitution::owningTemplateParameterSubstitution](#))  
The ParameterableElement that is owned by this TemplateParameterSubstitution as its actual parameter.
- templateBinding : [TemplateBinding](#) [1..1]{subsets [Element::owner](#)} (opposite [TemplateBinding::parameterSubstitution](#))  
The TemplateBinding that owns this TemplateParameterSubstitution.

## Constraints

- must\_be\_compatible  
The actual ParameterableElement must be compatible with the formal TemplateParameter, e.g., the actual ParameterableElement for a Class TemplateParameter must be a Class.

```
inv: actual->forall(a | a.isCompatibleWith(formal.parameteredElement))
```

## TemplateSignature [Class]

### Description

A Template Signature bundles the set of formal TemplateParameters for a template.

### Diagrams

[Template Bindings](#), [Templates](#), [Classifier Templates](#)

### Generalizations

[Element](#)

### Specializations

[RedefinableTemplateSignature](#)

### Association Ends

- ♦ ownedParameter : [TemplateParameter](#) [0..\*]{ordered, subsets [Element::ownedElement](#), subsets [TemplateSignature::parameter](#)} (opposite [TemplateParameter::signature](#))  
The formal parameters that are owned by this TemplateSignature.
- parameter : [TemplateParameter](#) [1..\*]{ordered} (opposite [A\\_parameter\\_templateSignature::templateSignature](#))  
The ordered set of all formal TemplateParameters for this TemplateSignature.
- template : [TemplateableElement](#) [1..1]{subsets [Element::owner](#)} (opposite [TemplateableElement::ownedTemplateSignature](#))  
The TemplateableElement that owns this TemplateSignature.

### Constraints

- own\_elements  
Parameters must own the ParameterableElements they parameter or those ParameterableElements must be owned by the TemplateableElement being templated.

```
inv: template.ownedElement->includesAll(parameter.parameteredElement->asSet() -  
parameter.ownedParameteredElement->asSet())
```

- unique\_parameters  
The names of the parameters of a TemplateSignature are unique.

```
inv: parameter->forall( p1, p2 | (p1 <> p2 and p1.parameteredElement.ocIsKindOf(NamedElement) and  
p2.parameteredElement.ocIsKindOf(NamedElement) ) implies  
p1.parameteredElement.ocAsType(NamedElement).name <>  
p2.parameteredElement.ocAsType(NamedElement).name)
```

## TemplateableElement [Abstract Class]

### Description

A TemplateableElement is an Element that can optionally be defined as a template and bound to other templates.

### Diagrams

[Template Bindings](#), [Templates](#), [Expressions](#), [Packages](#), [Classifiers](#), [Classifier Templates](#), [Operations](#)

### Generalizations

[Element](#)

### Specializations

[StringExpression](#), [Package](#), [Classifier](#), [Operation](#)

### Association Ends

- ♦ ownedTemplateSignature : [TemplateSignature](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [TemplateSignature::template](#))  
The optional TemplateSignature specifying the formal TemplateParameters for this TemplateableElement. If a TemplateableElement has a TemplateSignature, then it is a template.
- ♦ templateBinding : [TemplateBinding](#) [0..\*]{subsets [Element::ownedElement](#), subsets [A\\_source\\_directedRelationship::directedRelationship](#)} (opposite [TemplateBinding::boundElement](#))  
The optional TemplateBindings from this TemplateableElement to one or more templates.

### Operations

- isTemplate() : [Boolean](#)  
The query isTemplate() returns whether this TemplateableElement is actually a template.  
  
`body: ownedTemplateSignature <> null`
- parameterableElements() : [ParameterableElement](#) [0..\*]  
The query parameterableElements() returns the set of ParameterableElements that may be used as the parameteredElements for a TemplateParameter of this TemplateableElement. By default, this set includes all the ownedElements. Subclasses may override this operation if they choose to restrict the set of ParameterableElements.

```
body: self.allOwnedElements()->select(oclIsKindOf(ParameterableElement)).oclAsType(ParameterableElement)->asSet()
```

## Type [Abstract Class]

### Description

A Type constrains the values represented by a TypedElement.

## Diagrams

[Types](#), [Associations](#), [Packages](#), [Classifiers](#), [Features](#), [Operations](#)

## Generalizations

[PackageableElement](#)

## Specializations

[Classifier](#)

## Attributes

### Association Ends

- package : [Package](#) [0..1]{subsets [A\\_packageableElement\\_owningPackage::owningPackage](#)} (opposite [Package::ownedType](#))  
Specifies the owning Package of this Type, if any.

## Operations

- conformsTo(other : [Type](#)) : [Boolean](#)  
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.

```
body: false
```

## TypedElement [Abstract Class]

### Description

A TypedElement is a NamedElement that may have a Type specified for it.

### Diagrams

[Types](#), [Object Nodes](#), [Literals](#), [Structured Classifiers](#), [Features](#)

### Generalizations

[NamedElement](#)

### Specializations

[ObjectNode](#), [ValueSpecification](#), [ConnectableElement](#), [StructuralFeature](#)

### Association Ends

- type : [Type](#) [0..1] (opposite [A\\_type\\_typedElement::typedElement](#))  
The type of the TypedElement.

## Usage [Class]

### Description

A Usage is a Dependency in which the client Element requires the supplier Element (or set of Elements) for its full implementation or operation.

### Diagrams

[Dependencies](#)

### Generalizations

[Dependency](#)

## VisibilityKind [Enumeration]

### Description

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

### Diagrams

- [Namespaces](#)

### Literals

- public  
A Named Element with public visibility is visible to all elements that can access the contents of the Namespace that owns it.
- private  
A NamedElement with private visibility is only visible inside the Namespace that owns it.
- protected  
A NamedElement with protected visibility is visible to Elements that have a generalization relationship to the Namespace that owns it.
- package  
A NamedElement with package visibility is visible to all Elements within the nearest enclosing Package (given that other owning Elements have proper visibility). Outside the nearest enclosing Package, a NamedElement marked as having package visibility is not visible. Only NamedElements that are not owned by Packages can be marked as having package visibility.

## 7.9 Association Descriptions

### A\_actual\_templateParameterSubstitution [Association]

#### Diagrams

[Template Bindings](#)

#### Specializations

[A\\_ownedActual\\_owningTemplateParameterSubstitution](#)

#### Owned Ends

- templateParameterSubstitution : [TemplateParameterSubstitution](#) [0..\*] (opposite [TemplateParameterSubstitution::actual](#))

### A\_annotatedElement\_comment [Association]

#### Diagrams

[Root](#)

#### Owned Ends

- comment : [Comment](#) [0..\*] (opposite [Comment::annotatedElement](#))

### A\_clientDependency\_client [Association]

#### Diagrams

[Dependencies](#)

#### Member Ends

- [NamedElement::clientDependency](#)
- [Dependency::client](#)

### A\_constrainedElement\_constraint [Association]

#### Diagrams

[Constraints](#)

## Owned Ends

- constraint : [Constraint](#) [0..\*] (opposite [Constraint::constrainedElement](#))

## A\_default\_templateParameter [Association]

### Diagrams

[Templates](#)

### Specializations

[A\\_ownedDefault\\_templateParameter](#)

## Owned Ends

- templateParameter : [TemplateParameter](#) [0..\*] (opposite [TemplateParameter::default](#))

## A\_elementImport\_importingNamespace [Association]

### Diagrams

[Namespaces](#)

### Member Ends

- [Namespace::elementImport](#)
- [ElementImport::importingNamespace](#)

## A\_formal\_templateParameterSubstitution [Association]

### Diagrams

[Template Bindings](#)

## Owned Ends

- templateParameterSubstitution : [TemplateParameterSubstitution](#) [0..\*] (opposite [TemplateParameterSubstitution::formal](#))



## A\_importedElement\_import [Association]

### Diagrams

[Namespaces](#)

### Owned Ends

- import : [ElementImport](#) [0..\*]{subsets [A\\_target\\_directedRelationship::directedRelationship](#)} (opposite [ElementImport::importedElement](#))

## A\_importedMember\_namespace [Association]

### Diagrams

[Namespaces](#)

### Owned Ends

- namespace : [Namespace](#) [0..\*]{subsets [A\\_member\\_memberNamespace::memberNamespace](#)} (opposite [Namespace::importedMember](#))

## A\_importedPackage\_packageImport [Association]

### Diagrams

[Namespaces](#)

### Owned Ends

- packageImport : [PackageImport](#) [0..\*]{subsets [A\\_target\\_directedRelationship::directedRelationship](#)} (opposite [PackageImport::importedPackage](#))

## A\_lowerValue\_owningLower [Association]

### Diagrams

[Types](#)

### Owned Ends

- owningLower : [MultiplicityElement](#) [0..1]{subsets [Element::owner](#)} (opposite [MultiplicityElement::lowerValue](#))

## A\_mapping\_abstraction [Association]

### Diagrams

[Dependencies](#)

### Owned Ends

- abstraction : [Abstraction](#) [0..1]{subsets [Element::owner](#)} (opposite [Abstraction::mapping](#))

## A\_member\_memberNamespace [Association]

### Diagrams

[Namespaces](#)

### Owned Ends

- memberNamespace : [Namespace](#) [0..\*] (opposite [Namespace::member](#))

## A\_nameExpression\_namedElement [Association]

### Diagrams

[Namespaces](#)

### Owned Ends

- namedElement : [NamedElement](#) [0..1]{subsets [Element::owner](#)} (opposite [NamedElement::nameExpression](#))

## A\_ownedActual\_owningTemplateParameterSubstitution [Association]

### Diagrams

[Template Bindings](#)

### Generalizations

[A\\_actual\\_templateParameterSubstitution](#)

### Owned Ends

- owningTemplateParameterSubstitution : [TemplateParameterSubstitution](#) [0..1]{subsets [Element::owner](#), redefines [A\\_actual\\_templateParameterSubstitution::templateParameterSubstitution](#)} (opposite [TemplateParameterSubstitution::ownedActual](#))

## A\_ownedComment\_owningElement [Association]

### Diagrams

[Root](#)

### Owned Ends

- owningElement : [Element](#) [0..1]{subsets [Element::owner](#)} (opposite [Element::ownedComment](#))

## A\_ownedDefault\_templateParameter [Association]

### Diagrams

[Templates](#)

### Generalizations

[A default templateParameter](#)

### Owned Ends

- templateParameter : [TemplateParameter](#) [0..1]{subsets [Element::owner](#), redefines [A default templateParameter::templateParameter](#)} (opposite [TemplateParameter::ownedDefault](#))

## A\_ownedElement\_owner [Association]

### Diagrams

[Root](#)

### Member Ends

- [Element::ownedElement](#)
- [Element::owner](#)

## A\_ownedMember\_namespace [Association]

### Diagrams

[Namespaces](#)

## Member Ends

- [Namespace::ownedMember](#)
- [NamedElement::namespace](#)

## A\_ownedParameter\_signature [Association]

### Diagrams

[Templates](#)

## Member Ends

- [TemplateSignature::ownedParameter](#)
- [TemplateParameter::signature](#)

## A\_ownedParameteredElement\_owningTemplateParameter [Association]

### Diagrams

[Templates](#)

## Member Ends

- [TemplateParameter::ownedParameteredElement](#)
- [ParameterableElement::owningTemplateParameter](#)

## A\_ownedRule\_context [Association]

### Diagrams

[Namespaces](#), [Constraints](#)

## Member Ends

- [Namespace::ownedRule](#)
- [Constraint::context](#)

## A\_ownedTemplateSignature\_template [Association]

### Diagrams

[Templates](#)

## Member Ends

- [TemplateableElement::ownedTemplateSignature](#)
- [TemplateSignature::template](#)

## A\_packageImport\_importingNamespace [Association]

### Diagrams

[Namespaces](#)

### Member Ends

- [Namespace::packageImport](#)
- [PackageImport::importingNamespace](#)

## A\_parameterSubstitution\_templateBinding [Association]

### Diagrams

[Template Bindings](#)

### Member Ends

- [TemplateBinding::parameterSubstitution](#)
- [TemplateParameterSubstitution::templateBinding](#)

## A\_parameter\_templateSignature [Association]

### Diagrams

[Templates](#)

### Owned Ends

- templateSignature : [TemplateSignature](#) [0..\*] (opposite [TemplateSignature::parameter](#))

## A\_parameteredElement\_templateParameter [Association]

### Diagrams

[Templates](#)

### Member Ends

- [TemplateParameter::parameteredElement](#)
- [ParameterableElement::templateParameter](#)

## A\_relatedElement\_relationship [Association]

### Diagrams

[Root](#)

## Owned Ends

- relationship : [Relationship](#) [0..\*] (opposite [Relationship::relatedElement](#))

## A\_signature\_templateBinding [Association]

### Diagrams

[Template Bindings](#)

## Owned Ends

- templateBinding : [TemplateBinding](#) [0..\*]{subsets [A\\_target\\_directedRelationship::directedRelationship](#)} (opposite [TemplateBinding::signature](#))

## A\_source\_directedRelationship [Association]

### Diagrams

[Root](#)

## Owned Ends

- directedRelationship : [DirectedRelationship](#) [0..\*]{subsets [A\\_relatedElement\\_relationship::relationship](#)} (opposite [DirectedRelationship::source](#))

## A\_specification\_owningConstraint [Association]

### Diagrams

[Constraints](#)

## Specializations

[A\\_specification\\_intervalConstraint](#)

## Owned Ends

- owningConstraint : [Constraint](#) [0..1]{subsets [Element::owner](#)} (opposite [Constraint::specification](#))

## A\_supplier\_supplierDependency [Association]

### Diagrams

[Dependencies](#)

### Owned Ends

- supplierDependency : [Dependency](#) [0..\*]{subsets [A\\_target\\_directedRelationship::directedRelationship](#)} (opposite [Dependency::supplier](#))  
Indicates the dependencies that reference the supplier.

## A\_target\_directedRelationship [Association]

### Diagrams

[Root](#)

### Owned Ends

- directedRelationship : [DirectedRelationship](#) [0..\*]{subsets [A\\_relatedElement\\_relationship::relationship](#)} (opposite [DirectedRelationship::target](#))

## A\_templateBinding\_boundElement [Association]

### Diagrams

[Template Bindings](#)

### Member Ends

- [TemplateableElement::templateBinding](#)
- [TemplateBinding::boundElement](#)

## A\_type\_typedElement [Association]

### Diagrams

[Types](#)

### Owned Ends

- typedElement : [TypedElement](#) [0..\*] (opposite [TypedElement::type](#))

## A\_upperValue\_owningUpper [Association]

### Diagrams

[Types](#)

### Owned Ends

- owningUpper : [MultiplicityElement](#) [0..1]{subsets [Element::owner](#)} (opposite [MultiplicityElement::upperValue](#))



# 8 Values

## 8.1 Summary

This clause describes the specification of values. In general, a ValueSpecification is a model element that is considered semantically to yield zero or more *values*. The type and number of values shall be suitable for the context in which the ValueSpecification is used (as determined by the constraints given in that context).

The following sub clauses describe the various kinds of ValueSpecifications available in UML.

## 8.2 Literals

### 8.2.1 Summary

A LiteralSpecification is a ValueSpecification that specifies a literal value. There is a different kind of LiteralSpecification for each of the UML standard PrimitiveTypes, with a corresponding textual literal notation, plus a “null” literal that represents the “lack of a value.”

### 8.2.2 Abstract Syntax

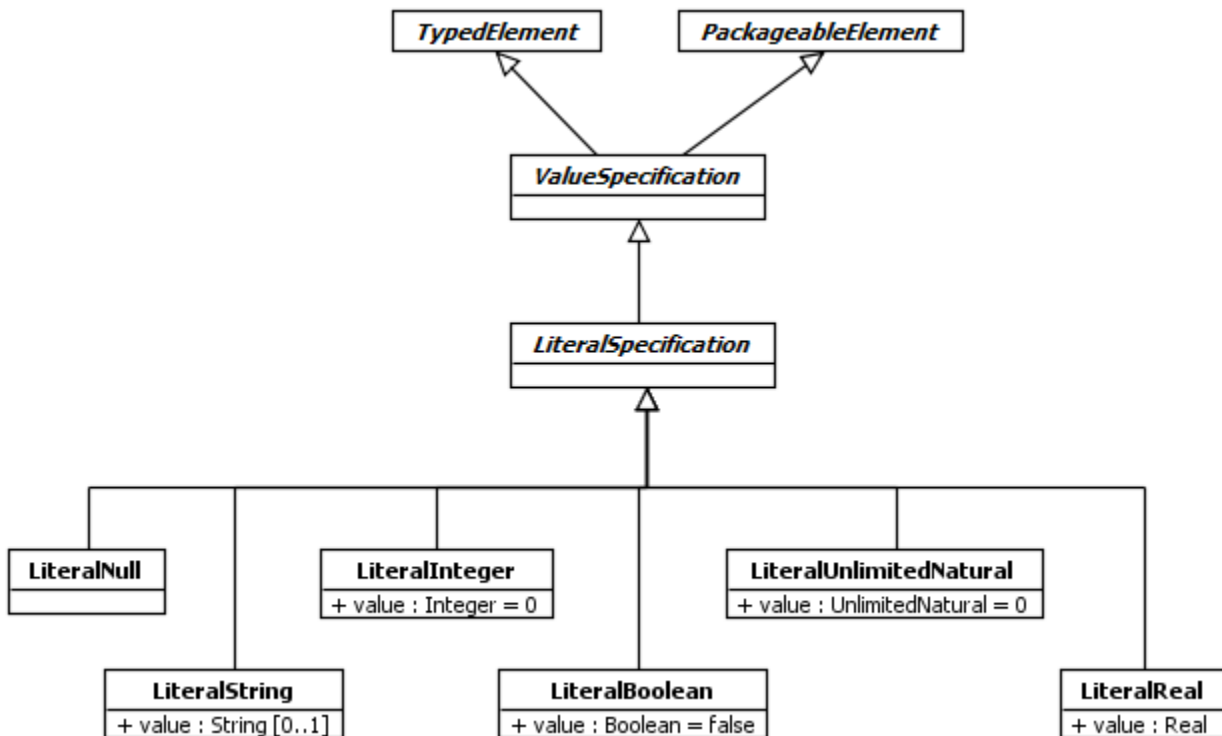


Figure 8.1 Literals

### 8.2.3 Semantics

There are six kinds of LiteralSpecifications:

1. A LiteralNull is intended to be used to explicitly model the lack of a value. In the context of a MultiplicityElement with a multiplicity lower bound of 0, this corresponds to the empty set (i.e., a set of no values). It is equivalent to specifying no values for the Element.
2. A LiteralString specifies a constant value of the PrimitiveType String. Though a String is specified as a sequence of characters, String values are considered to be primitive in UML, so their internal structure is not specified as part of UML semantics.
3. A LiteralInteger specifies a constant value of the PrimitiveType Integer.
4. A LiteralBoolean specifies a constant value of the PrimitiveType Boolean.
5. A LiteralUnlimitedNatural specifies a constant value of the PrimitiveType UnlimitedNatural.
6. A LiteralReal specifies a constant value of the PrimitiveType Real.

See also Clause [21](#) for further discussion of the standard UML primitive types.

### 8.2.4 Notation

LiteralSpecifications are notated textually.

- The notation for a LiteralNull varies depending on where it is used. It often appears as the word “null.” Other notations are described elsewhere for specific uses.
- A LiteralString is shown as a sequence of characters within double quotes. The String value is the sequence of characters, not including the quotes. The character set used is unspecified.
- A LiteralInteger is shown as a sequence of digits representing the decimal numeral for the Integer value.
- A LiteralBoolean is shown as either the word “true” or the word “false,” corresponding to its value.
- A LiteralUnlimitedNatural is shown either as a sequence of digits or as an asterisk (\*), where an asterisk denotes *unlimited*. Note that “unlimited” denotes the lack of a limit on the value of some element (such as a multiplicity upper bound), not a value of “infinity.”
- A LiteralReal is shown in decimal notation or scientific notation. Decimal notation consists of an optional sign character (+/-) followed by zero or more digits followed optionally by a dot (.) followed by one or more digits. Scientific notation consists of decimal notation followed by either the letter “e” or “E” and an exponent consisting of an optional sign character followed by one or more digits. The scientific notation expresses a real number equal to that given by the decimal notation before the exponent, times 10 raised to the power of the exponent.

This notation is specified by the following EBNF rules:

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

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

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

## 8.3 Expressions

### 8.3.1 Summary

Expressions are ValueSpecifications that specify computed values.

### 8.3.2 Abstract Syntax

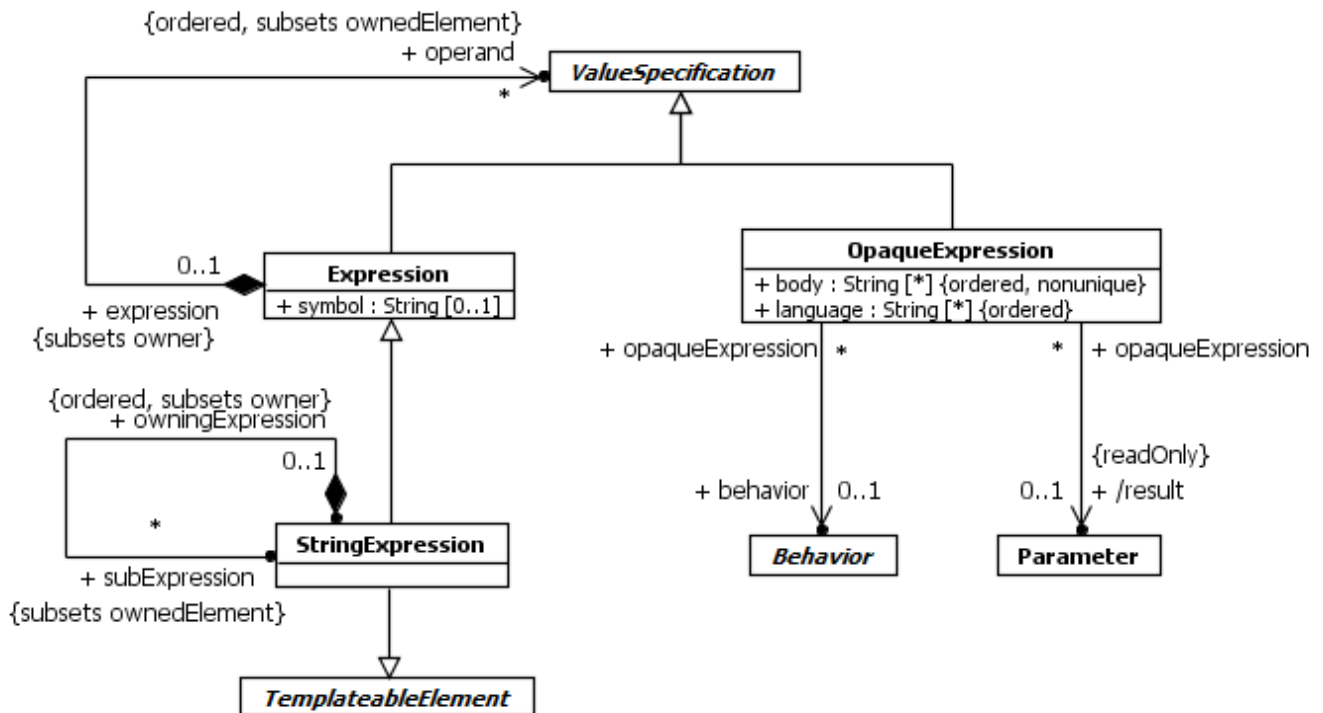


Figure 8.2 Expressions

### 8.3.3 Semantics

#### Expressions

An Expression is specified as a tree structure. Each node in this tree structure consists of a symbol and an optional set of operands. If there are no operands, the Expression represents a terminal node. If there are operands, the Expression represents the operator given by the symbol applied to those operands.

An Expression is evaluated by first evaluating each of its operands and then performing the operation denoted by the Expression symbol to the resulting operand values. However, the actual interpretation of the symbol depends on the context of use of the Expression and this specification does not provide any standard symbol definitions. A conforming tool may define a specific set of symbols for which it provides interpretations or it may simply treat all Expressions as uninterpreted.

## String Expressions

A StringExpression is an Expression that specifies a String value that is derived by concatenating a set of substrings. The substrings are given as either a list of LiteralString operands or as a list of StringExpression subExpressions (but it is not allowed to mix the two). The String value of a StringExpression is obtained by concatenating, in order, the String values of either the operands or the subExpressions, depending on which is given.

StringExpressions are intended to be used to specify the names of NamedElements in the context of Templates. Either the entire StringExpression or one or subExpressions of it may be used as the ParameterableElements of TemplateParameters, allowing the name of a NamedElement to be parameterized within a template. See the semantics of NamedElements in sub clause [7.4.3](#) for further discussion of this.

## Opaque Expressions

An OpaqueExpression specifies the computation of a set of values either in terms of a UML Behavior or based on a textual statement in a language other than UML.

An OpaqueExpression may have a body that consists of a sequence of text Strings representing alternative means of computing the values of the OpaqueExpression. A corresponding sequence of language Strings may be used to specify the languages in which each of the body Strings is to be interpreted. Languages are matched to body Strings by order. The UML specification does not define how body Strings are interpreted relative to any language, though other specifications may define specific language Strings to be used to indicate interpretation with respect to those specifications (e.g., “OCL” for expressions to be interpreted according to the OCL specification). Note also that it is not required to specify the languages. If they are unspecified, then the interpretation of any body Strings must be determined implicitly from the form of the bodies or the context of use of the OpaqueExpression.

An OpaqueExpression may also be defined by a UML Behavior (see sub clause [13.2](#)) that is restricted to have no Parameters other than a return Parameter. The values of the OpaqueExpression are given by invoking the Behavior and returning the values on the return Parameter. Note that the behavior of an OpaqueExpression does not have Parameters other than its return and thus cannot be passed data upon invocation. It must therefore access any input data through elements of its behavioral description.

If an OpaqueExpression has more than one body String, or a behavior in addition to one or more body Strings, then any one of the bodies or the behavior may be used to evaluate the OpaqueExpression. The UML specification does not determine how this choice is made.

## 8.3.4 Notation

### Expressions

An Expression with no operands is notated simply by its symbol (unlike a StringLiteral, the symbol is *not* enclosed in quotes). An Expression with operands may be notated by its symbol, followed by round parentheses containing its operands in order, separated by commas. However, in particular contexts, a conforming tool may permit special notations, including infix operators.

See sub clause [7.4.4](#) for the notation of the use of StringExpressions with NamedElements.

### Opaque Expressions

If an OpaqueExpression has one or more body Strings, then these are used to display the OpaqueExpression in the context of its containing element. The UML Specification does not define the syntax of such Strings, but, if a corresponding language is given for a body String, a conforming tool may enforce the syntax of that language. A conforming tool may also restricted the languages allowed or assume a particular default language.

If languages are specified for an OpaqueExpression, then a language name may be displayed in braces ({} before the body String to which it corresponds. It is not required, however, that the languages of an OpaqueExpression be displayed.

If a language has a specification that defines its language name, then the language name used in an OpaqueExpression should be spelled and capitalized exactly as it appears in the specification for the language. For example, use “OCL,” not “ocl.”

### 8.3.5 Examples

#### Expressions

*xor*

*else*

*plus(x,1)*

*x+1*

#### Opaque Expressions

*a > 0*

*{OCL} i > j and self.size > i*

*average hours worked per week*

## 8.4 Time

### 8.4.1 Summary

This sub clause defines TimeExpressions and Durations that produce values based on a simple model of time. This simple model of time is intended as an approximation for situations in which the more complex aspects of time and time measurement can safely be ignored. For example, in many distributed system there is no global notion of time, only the notion of local time relative to each distributed element of the system. This relativity of time is not accounted for in the simple time model, nor are the effects resulting from imperfect clocks with finite resolution, overflows, drift, skew, etc. It is assumed that applications for which such characteristics are relevant will use a more sophisticated model of time provided by an appropriate profile.

## 8.4.2 Abstract Syntax

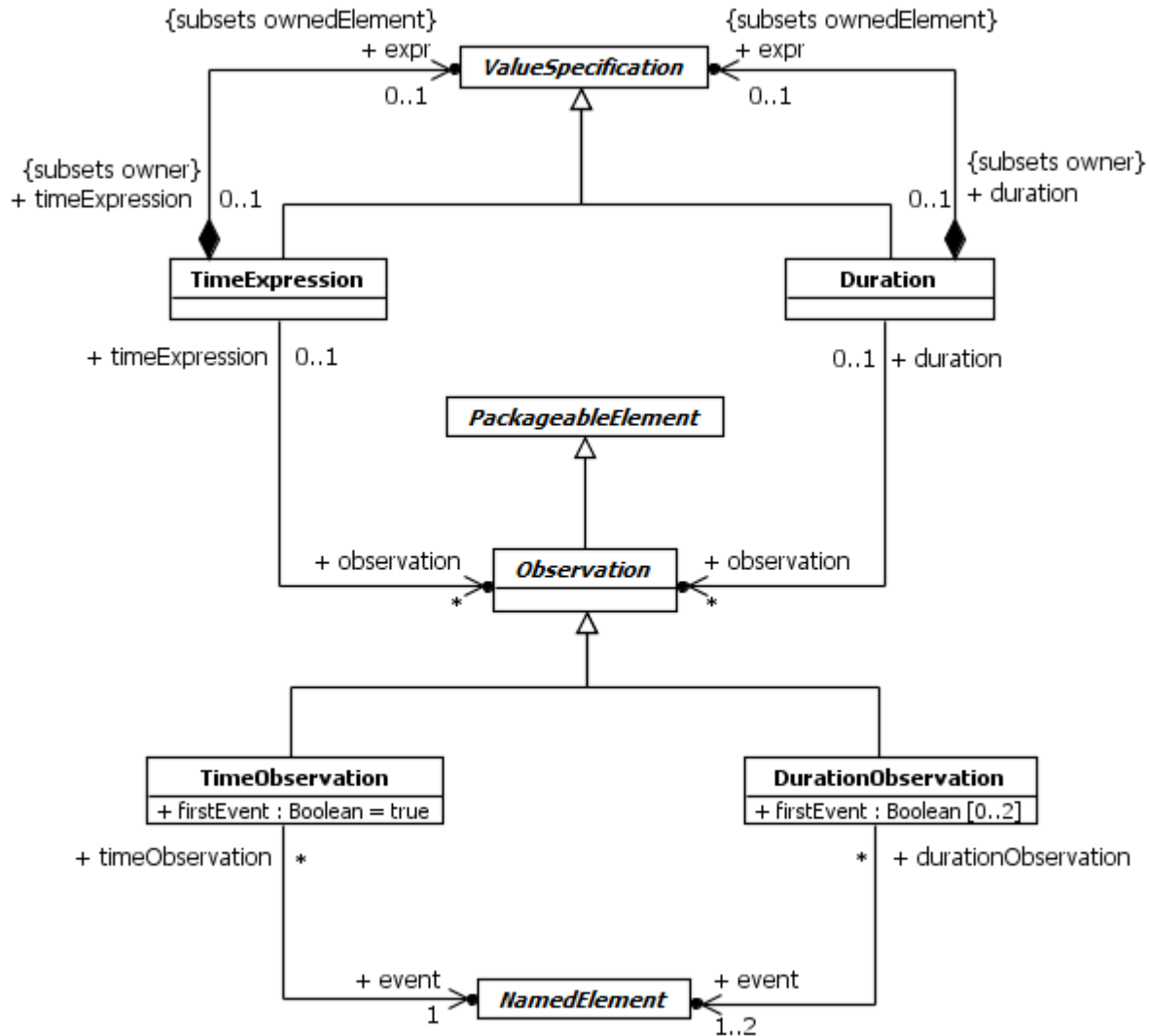


Figure 8.3 Time and Duration

## 8.4.3 Semantics

### Time

The structural modeling constructs of UML are used to model the properties of entities at specific points in time. In contrast, behavioral modeling constructs are used to model how these properties change over time. An *event* is a specification of something that may *occur* at a specific point in time when something of interest happens relative to the properties and behaviors being modeled, such as the change in value of a Property or the beginning of execution of an Activity.

*Time* in this conception is simply a coordinate that orders the occurrence of events. Every event occurrence can be given a time coordinate value and, based on this, can be said to be before, after or at the same time as another event occurrence.

A *duration* is the period of time between two event occurrences, computed as the difference of the time coordinates of those events. If a model Element has a behavioral effect, then this effect may occur over some duration. The starting event of this duration is known as *entering* the element and the ending event is known as *exiting* the Element.

## Observations

An Observation denotes the observation of events that may occur relative to some other part of a model. An Observation is made on a NamedElement within the model. The events of interest are when the reference NamedElement is entered and exited. If the referenced NamedElement is not a behavioral element, then the duration between entering and exiting the NamedElement is considered to be zero, but this specification does not otherwise define what specific events are observed on the Element.

There are two kinds of Observations, TimeObservations and DurationObservations.

A TimeObservation observes either entering or exiting a specific NamedElement. If *firstEvent* is true, then it is the entry event that is observed, otherwise the exit event is observed. The result of a TimeObservation is the time at which the observed event occurs.

A DurationObservation observes a duration relative to either one or two NamedElements. If a single element is observed, then the observed duration is between sequential occurrences of the entry and exit events of the element. If two elements are observed, then the duration is between either the entry or the exit event of the first element and a subsequent entry or exit event of the second element. In the latter case, two corresponding *firstEvent* values must also be given for the DurationObservation, such that, if *firstEvent*=true for an observed element, then it is the entry event that is observed, otherwise it is the exit event that is observed.

## TimeExpression

A TimeExpression is a ValueSpecification that evaluates to the time coordinate for an instant in time, possibly relative to some given set of observations.

If the TimeExpression has an *expr*, then this is evaluated to produce the result of the TimeExpression. The *expr* must evaluate to a single value, but UML does not define any specific type or units that the value must have. The *expr* may reference the observations associated with the TimeExpression but no standard notation is defined for such references. If the TimeExpression has an *expr* but no observations, then the *expr* evaluates to a time constant.

If the TimeExpression does not have an *expr*, then it must have a single TimeObservation and the result of that observation is the value of the TimeExpression.

## Duration

A Duration is a ValueSpecification that evaluates to some duration in time, possibly relative to some given set of observations.

If the Duration has an *expr*, then this is evaluated to produce the result of the DurationExpression. The *expr* must evaluate to a single value, but UML does not define any specific type or units that the value must have. The *expr* may reference the observations associated with the Duration but no standard notation is defined for such references. If the Duration has an *expr* but no observations, then the *expr* evaluates to a duration constant.

If the Duration does not have an *expr*, then it must have a single DurationObservation and the result of that observation is the value of the Duration.

## 8.4.4 Notation

### Observations

An Observation may be denoted by a straight line attached to the NamedElement it references. The Observation is given a name that is shown close to the unattached end of the line. Additional notation conventions on Observations are given elsewhere relative to the modeling constructs in which they are typically used (such as Interactions, see sub clause [17.2](#)).

### Time Expressions and Durations

A TimeExpression or Duration is denoted by the textual representation of its `expr`, if it has one (see sub clause 8.3.5). The representation is of a formula for computing the time or duration value, which may include the names of related Observations and constants. If a TimeExpression or Duration does not have an `expr`, then it is simply represented by its single associated Observation.

A Duration is a value of relative time given in an implementation specific textual format. Often a Duration is a non-negative integer expression representing the number of “time ticks” which may elapse during this duration.

## 8.4.5 Examples

Time is often represented using a numeric coordinate, in which case the `expr` of a TimeExpression should evaluate to a numeric value, the units of which may be assumed by convention in a model (e.g., times are always in seconds). Alternatively, DataTypes may be used to model time values with specific units (e.g., Second, Day, etc.) and the `expr` of a TimeExpression should then have the appropriate one of those types.

A Duration is a value of relative time and, as such, is often represented as a non-negative number, such as an Integer count of the number of “time ticks” on a reference clock that elapsed during the duration. In this case, the `expr` of a DurationExpression should evaluate to a non-negative numeric value. A Duration value may also be used to represent a time coordinate value as a Duration since some fixed “origin” of time.

See also Figure 8.5 in Subclause 8.5.5.

## 8.5 Intervals

### 8.5.1 Summary

An Interval is a range between two values, primarily for use in Constraints that assert that some other Element has a value in the given range. Intervals can be defined for any type of value, but they are especially useful for time and duration values as part of corresponding TimeConstraints and DurationConstraints.



## 8.5.2 Abstract Syntax

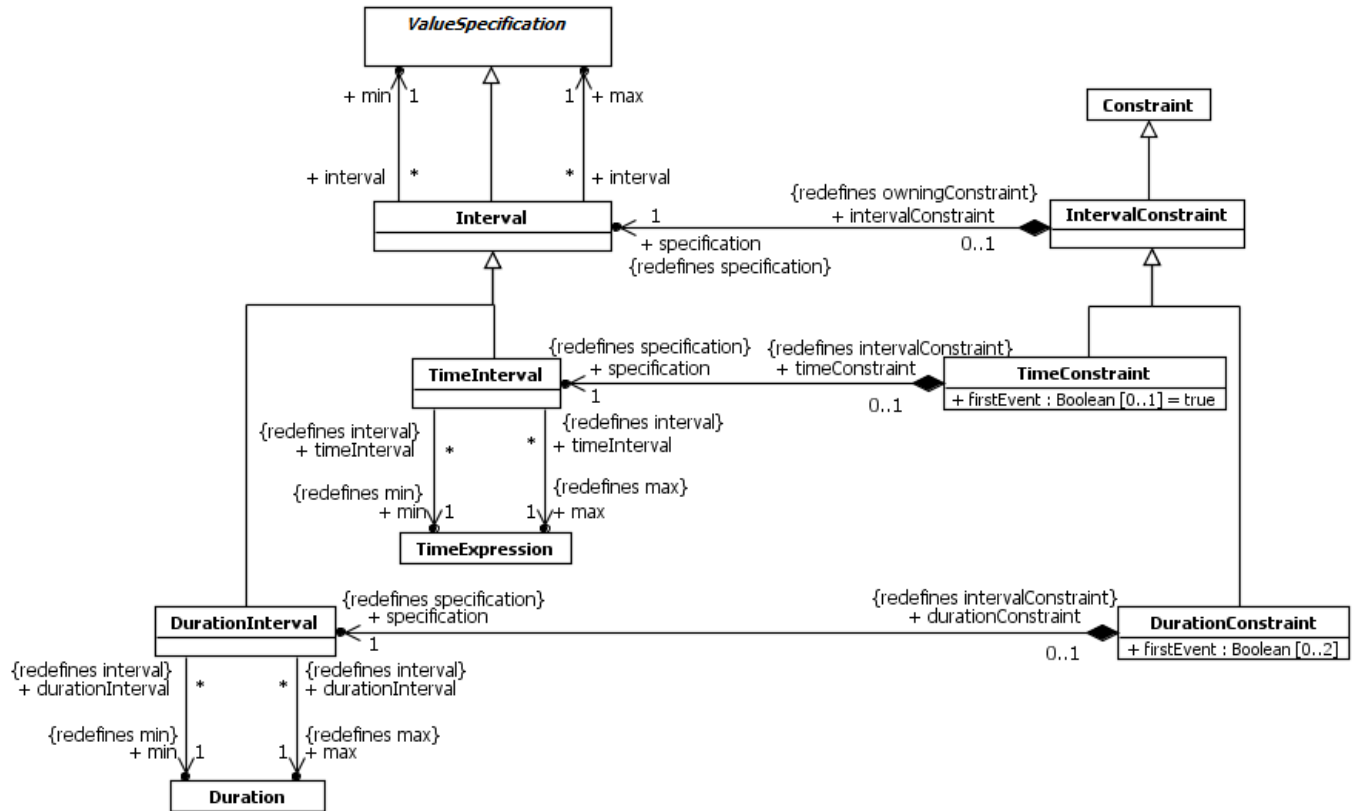


Figure 8.4 Intervals

## 8.5.3 Semantics

### Intervals

An Interval is a ValueSpecification specified using two other ValueSpecifications, the min and the max. An Interval is evaluated by first evaluating each of its constituent ValueSpecifications, which must each evaluate to a single value. The value of the Interval is then the range from the min value to the max value—that is, the set of all values greater than or equal to the min value and less than or equal to the max value (which may be the empty set). Note that, while syntactically any ValueSpecifications of any type are allowed for the min and max of an Interval, a standard semantic interpretation is only given for Intervals for which the min and max ValueSpecifications have the same type and that type has a total ordering defined on it.

There are two specializations of Interval for use with timing constraints. A TimeInterval specifies the range between two time values given by TimeExpressions. A DurationInterval specifies the range between two duration values given by Durations.

### IntervalConstraint

An IntervalConstraint defines a Constraint whose specification is given by an Interval (see also sub clause 7.6 on Constraints). The constrainedElements of an IntervalConstraint are asserted to have values that are within the range specified by the Interval of the IntervalConstraint. If a constrainedElement has a value outside this range, then the IntervalConstraint is violated. If any constrainedElement cannot be interpreted to have a value, or its value is not the same type as the range given by the IntervalConstraint, then the IntervalConstraint has no standard semantic interpretation.

There are two specializations of `IntervalConstraint` for use in specifying timing constraints. A `TimeConstraint` defines an `IntervalConstraint` on a single `constrainedElement` in which the constraining `Interval` is a `TimeInterval`. A `DurationConstraint` defines an `IntervalConstraint` on either one or two `constrainedElements` in which the constraining `Interval` is a `DurationInterval`. If there are two `constrainedElements`, then the start of the duration being observed may be between an event in the first `constrainedElement` and an event in the second.

## 8.5.4 Notation

### Intervals

An `Interval` is denoted textually by the textual representation of its two `ValueSpecification`s separated by “..”:

`<interval> ::= <min-value> ‘..’ <max-value>`

A `TimeInterval` is shown with the notation of `Interval` where each `ValueSpecification` element is a `TimeExpression`. A `DurationInterval` is shown using the notation of `Interval` where each `ValueSpecification` element is a `Duration`. (See sub clause 8.4.4 on the notation for `TimeExpressions` and `Durations`.)

### Interval Constraints

An `IntervalConstraint` is shown as an annotation of its `constrainedElement`. The general notation for `Constraints` (see sub clause 7.6.4) may be used for an `IntervalConstraint`, with the specification `Interval` denoted textually as above. Special notational constructs are defined for `TimeConstraints` and `DurationConstraints`, as given below.

A `TimeConstraint` of a single `constrainedElement` may be shown as a small line between the graphical representation of the `constrainedElement` and the textual representation of the `TimeInterval` of `TimeConstraint`. A `DurationConstraint` may also be shown using a graphical notation relating its `constrainedElements`. However, the notation used is specific to the diagram type on which the `DurationConstraint` appears (see sub clause 17.8 for the notation on `Sequence Diagrams` and sub clause 17.11 for the notation on `Timing Diagrams`).

## 8.5.5 Examples

Figure 8.5 shows a `DurationConstraints` associated with the duration of a `Message` and with the duration between two `OccurrenceSpecifications`. It also shows a `TimeConstraint` associated with the reception of a `Message`. (See also sub clause 17.2.5.)

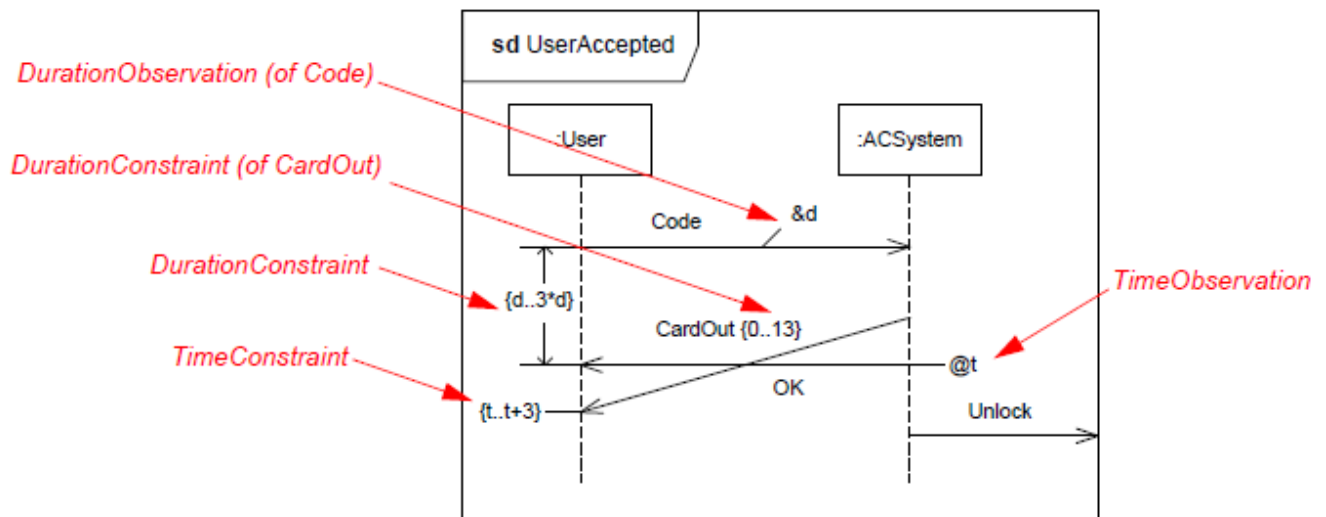


Figure 8.5 Example of DurationConstraints and TimeConstraints

## 8.6 Classifier Descriptions

### Duration [Class]

#### Description

A Duration is a ValueSpecification that specifies the temporal distance between two time instants.

#### Diagrams

[Time](#), [Intervals](#)

#### Generalizations

[ValueSpecification](#)

#### Association Ends

- ♦ expr : [ValueSpecification](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_expr\\_duration::duration](#))  
A ValueSpecification that evaluates to the value of the Duration.
- observation : [Observation](#) [0..\*] (opposite [A\\_observation\\_duration::duration](#))  
Refers to the Observations that are involved in the computation of the Duration value

#### Constraints

- no\_expr\_requires\_observation  
If a Duration has no expr, then it must have a single observation that is a DurationObservation.

```
inv: expr = null implies (observation->size() = 1 and observation->forall(oclIsKindOf(DurationObservation)))
```

### DurationConstraint [Class]

#### Description

A DurationConstraint is a Constraint that refers to a DurationInterval.

#### Diagrams

[Intervals](#)

#### Generalizations

[IntervalConstraint](#)

#### Attributes

- firstEvent : [Boolean](#) [0..2]  
The value of firstEvent[i] is related to constrainedElement[i] (where i is 1 or 2). If firstEvent[i] is true, then the corresponding observation event is the first time instant the execution enters constrainedElement[i]. If firstEvent[i] is

false, then the corresponding observation event is the last time instant the execution is within constrainedElement[i].

## Association Ends

- ♦ specification : [DurationInterval](#) [1..1]{redefines [IntervalConstraint::specification](#)} (opposite [A\\_specification\\_durationConstraint::durationConstraint](#))  
The DurationInterval constraining the duration.

## Constraints

- first\_event\_multiplicity  
The multiplicity of firstEvent must be 2 if the multiplicity of constrainedElement is 2. Otherwise the multiplicity of firstEvent is 0.

```
inv: if (constrainedElement->size() = 2)
    then (firstEvent->size() = 2) else (firstEvent->size() = 0)
endif
```

- has\_one\_or\_two\_constrainedElements  
A DurationConstraint has either one or two constrainedElements.

```
inv: constrainedElement->size() = 1 or constrainedElement->size()=2
```

## DurationInterval [Class]

### Description

A DurationInterval defines the range between two Durations.

### Diagrams

[Intervals](#)

### Generalizations

[Interval](#)

### Association Ends

- max : [Duration](#) [1..1]{redefines [Interval::max](#)} (opposite [A\\_max\\_durationInterval::durationInterval](#))  
Refers to the Duration denoting the maximum value of the range.
- min : [Duration](#) [1..1]{redefines [Interval::min](#)} (opposite [A\\_min\\_durationInterval::durationInterval](#))  
Refers to the Duration denoting the minimum value of the range.

## DurationObservation [Class]

### Description

A DurationObservation is a reference to a duration during an execution. It points out the NamedElement(s) in the model to observe and whether the observations are when this NamedElement is entered or when it is exited.

### Diagrams

[Time](#)

### Generalizations

[Observation](#)

### Attributes

- firstEvent : [Boolean](#) [0..2]  
The value of firstEvent[i] is related to event[i] (where i is 1 or 2). If firstEvent[i] is true, then the corresponding observation event is the first time instant the execution enters event[i]. If firstEvent[i] is false, then the corresponding observation event is the time instant the execution exits event[i].

### Association Ends

- event : [NamedElement](#) [1..2] (opposite [A\\_event\\_durationObservation::durationObservation](#))  
The DurationObservation is determined as the duration between the entering or exiting of a single event Element during execution, or the entering/exiting of one event Element and the entering/exiting of a second.

### Constraints

- first\_event\_multiplicity  
The multiplicity of firstEvent must be 2 if the multiplicity of event is 2. Otherwise the multiplicity of firstEvent is 0.

```
inv: if (event->size() = 2)
    then (firstEvent->size() = 2) else (firstEvent->size() = 0)
endif
```

## Expression [Class]

### Description

An Expression represents a node in an expression tree, which may be non-terminal or terminal. It defines a symbol, and has a possibly empty sequence of operands that are ValueSpecifications. It denotes a (possibly empty) set of values when evaluated in a context.

### Diagrams

[Expressions](#)

## Generalizations

[ValueSpecification](#)

## Specializations

[StringExpression](#)

## Attributes

- symbol : [String](#) [0..1]  
The symbol associated with this node in the expression tree.

## Association Ends

- ♦ operand : [ValueSpecification](#) [0..\*]{ordered, subsets [Element::ownedElement](#)} (opposite [A\\_operand\\_expression::expression](#))  
Specifies a sequence of operand ValueSpecifications.

## Interval [Class]

### Description

An Interval defines the range between two ValueSpecifications.

### Diagrams

[Intervals](#)

## Generalizations

[ValueSpecification](#)

## Specializations

[DurationInterval](#), [TimeInterval](#)

## Association Ends

- max : [ValueSpecification](#) [1..1] (opposite [A\\_max\\_interval::interval](#))  
Refers to the ValueSpecification denoting the maximum value of the range.
- min : [ValueSpecification](#) [1..1] (opposite [A\\_min\\_interval::interval](#))  
Refers to the ValueSpecification denoting the minimum value of the range.

## IntervalConstraint [Class]

### Description

An IntervalConstraint is a Constraint that is specified by an Interval.

### Diagrams

[Intervals](#)

### Generalizations

[Constraint](#)

### Specializations

[DurationConstraint](#), [TimeConstraint](#)

### Association Ends

- ♦ specification : [Interval](#) [1..1]{redefines [Constraint::specification](#)} (opposite [A\\_specification\\_intervalConstraint::intervalConstraint](#))  
The Interval that specifies the condition of the IntervalConstraint.

## LiteralBoolean [Class]

### Description

A LiteralBoolean is a specification of a Boolean value.

### Diagrams

[Literals](#)

### Generalizations

[LiteralSpecification](#)

### Attributes

- value : [Boolean](#) [1..1] = false  
The specified Boolean value.

### Operations

- booleanValue() : [Boolean](#)  
The query booleanValue() gives the value.

body: value



- `isComputable()` : [Boolean](#)  
The query `isComputable()` is redefined to be true.

`body: true`

## LiteralInteger [Class]

### Description

A `LiteralInteger` is a specification of an Integer value.

### Diagrams

[Literals](#)

### Generalizations

[LiteralSpecification](#)

### Attributes

- `value` : [Integer](#) [1..1] = 0  
The specified Integer value.

### Operations

- `integerValue()` : [Integer](#)  
The query `integerValue()` gives the value.

`body: value`

- `isComputable()` : [Boolean](#)  
The query `isComputable()` is redefined to be true.

`body: true`

## LiteralNull [Class]

### Description

A `LiteralNull` specifies the lack of a value.

### Diagrams

[Literals](#)

### Generalizations

[LiteralSpecification](#)

## Operations

- `isComputable()` : [Boolean](#)  
The query `isComputable()` is redefined to be true.

body: true

- `isNull()` : [Boolean](#)  
The query `isNull()` returns true.

body: true

## LiteralReal [Class]

### Description

A `LiteralReal` is a specification of a Real value.

### Diagrams

[Literals](#)

### Generalizations

[LiteralSpecification](#)

### Attributes

- `value` : [Real](#) [1..1]  
The specified Real value.

## Operations

- `isComputable()` : [Boolean](#)  
The query `isComputable()` is redefined to be true.

body: true

- `realValue()` : [Real](#)  
The query `realValue()` gives the value.

body: value

## LiteralSpecification [Abstract Class]

### Description

A `LiteralSpecification` identifies a literal constant being modeled.

## Diagrams

[Literals](#)

## Generalizations

[ValueSpecification](#)

## Specializations

[LiteralBoolean](#), [LiteralInteger](#), [LiteralNull](#), [LiteralReal](#), [LiteralString](#), [LiteralUnlimitedNatural](#)

## LiteralString [Class]

### Description

A LiteralString is a specification of a String value.

### Diagrams

[Literals](#)

### Generalizations

[LiteralSpecification](#)

### Attributes

- value : [String](#) [0..1]  
The specified String value.

### Operations

- isComputable() : [Boolean](#)  
The query isComputable() is redefined to be true.

body: true

- stringValue() : [String](#)  
The query stringValue() gives the value.

body: value

## LiteralUnlimitedNatural [Class]

### Description

A LiteralUnlimitedNatural is a specification of an UnlimitedNatural number.

### Diagrams

[Literals](#)

## Generalizations

[LiteralSpecification](#)

## Attributes

- value : [UnlimitedNatural](#) [1..1] = 0  
The specified UnlimitedNatural value.

## Operations

- isComputable() : [Boolean](#)  
The query isComputable() is redefined to be true.

body: true

- unlimitedValue() : [UnlimitedNatural](#)  
The query unlimitedValue() gives the value.

body: value

## Observation [Abstract Class]

### Description

Observation specifies a value determined by observing an event or events that occur relative to other model Elements.

### Diagrams

[Time](#)

### Generalizations

[PackageableElement](#)

### Specializations

[DurationObservation](#), [TimeObservation](#)

## OpaqueExpression [Class]

### Description

An OpaqueExpression is a ValueSpecification that specifies the computation of a set of values either in terms of a UML Behavior or based on a textual statement in a language other than UML

### Diagrams

[Expressions](#), [Dependencies](#)

## Generalizations

### [ValueSpecification](#)

## Attributes

- body : [String](#) [0..\*]  
A textual definition of the behavior of the OpaqueExpression, possibly in multiple languages.
- language : [String](#) [0..\*]  
Specifies the languages used to express the textual bodies of the OpaqueExpression. Languages are matched to body Strings by order. The interpretation of the body depends on the languages. If the languages are unspecified, they may be implicit from the expression body or the context.

## Association Ends

- behavior : [Behavior](#) [0..1] (opposite [A\\_behavior\\_opaqueExpression::opaqueExpression](#))  
Specifies the behavior of the OpaqueExpression as a UML Behavior.
- /result : [Parameter](#) [0..1]{ } (opposite [A\\_result\\_opaqueExpression::opaqueExpression](#))  
If an OpaqueExpression is specified using a UML Behavior, then this refers to the single required return Parameter of that Behavior. When the Behavior completes execution, the values on this Parameter give the result of evaluating the OpaqueExpression.

## Operations

- isIntegral() : [Boolean](#)  
The query isIntegral() tells whether an expression is intended to produce an Integer.

```
body: false
```

- isNonNegative() : [Boolean](#)  
The query isNonNegative() tells whether an integer expression has a non-negative value.

```
pre: self.isIntegral()  
body: false
```

- isPositive() : [Boolean](#)  
The query isPositive() tells whether an integer expression has a positive value.

```
pre: self.isIntegral()  
body: false
```

- result() : [Parameter](#) [0..1]  
Derivation for OpaqueExpression:/result

```
body: if behavior = null then  
    null  
else  
    behavior.ownedParameter->first()
```

```
endif
```

- `value() : Integer`  
The query `value()` gives an integer value for an expression intended to produce one.

```
pre: self.isIntegral()  
body: 0
```

## Constraints

- `language_body_size`  
If the `language` attribute is not empty, then the size of the `body` and `language` arrays must be the same.

```
inv: language->notEmpty() implies (_'body'->size() = language->size())
```

- `one_return_result_parameter`  
The behavior must have exactly one return result parameter.

```
inv: behavior <> null implies  
    behavior.ownedParameter->select(direction=ParameterDirectionKind::return)->size() = 1
```

- `only_return_result_parameters`  
The behavior may only have return result parameters.

```
inv: behavior <> null implies behavior.ownedParameter-  
>select(direction<>ParameterDirectionKind::return)->isEmpty()
```

## StringExpression [Class]

### Description

A `StringExpression` is an `Expression` that specifies a `String` value that is derived by concatenating a sequence of operands with `String` values or a sequence of `subExpressions`, some of which might be template parameters.

### Diagrams

[Expressions](#), [Namespaces](#)

### Generalizations

[TemplateableElement](#), [Expression](#)

### Association Ends

- `owningExpression : StringExpression [0..1]{ordered, subsets Element::owner}` (opposite [StringExpression::subExpression](#))  
The `StringExpression` of which this `StringExpression` is a `subExpression`.
- `♦ subExpression : StringExpression [0..*]{subsets Element::ownedElement}` (opposite [StringExpression::owningExpression](#))  
The `StringExpressions` that constitute this `StringExpression`.

## Operations

- `stringValue() : String`  
The query `stringValue()` returns the String resulting from concatenating, in order, all the component String values of all the operands or subExpressions that are part of the StringExpression.

```
body: if subExpression->notEmpty()  
then subExpression->iterate(se; stringValue: String = '' | stringValue.concat(se.stringValue()))  
else operand->iterate(op; stringValue: String = '' | stringValue.concat(op.stringValue()))  
endif
```

## Constraints

- operands  
All the operands of a StringExpression must be LiteralStrings  
  
`inv: operand->forall (oclIsKindOf (LiteralString))`
- subexpressions  
If a StringExpression has sub-expressions, it cannot have operands and vice versa (this avoids the problem of having to define a collating sequence between operands and subexpressions).

```
inv: if subExpression->notEmpty() then operand->isEmpty() else operand->notEmpty() endif
```

## TimeConstraint [Class]

### Description

A TimeConstraint is a Constraint that refers to a TimeInterval.

### Diagrams

[Intervals](#)

### Generalizations

[IntervalConstraint](#)

### Attributes

- `firstEvent : Boolean [0..1] = true`  
The value of `firstEvent` is related to the `constrainedElement`. If `firstEvent` is true, then the corresponding observation event is the first time instant the execution enters the `constrainedElement`. If `firstEvent` is false, then the corresponding observation event is the last time instant the execution is within the `constrainedElement`.

### Association Ends

- ♦ `specification : TimeInterval [1..1]{redefines IntervalConstraint::specification}` (opposite [A\\_specification\\_timeConstraint::timeConstraint](#))  
TheTimeInterval constraining the duration.

## Constraints

- `has_one_constrainedElement`  
A `TimeConstraint` has one `constrainedElement`.

```
inv: constrainedElement->size() = 1
```

## TimeExpression [Class]

### Description

A `TimeExpression` is a `ValueSpecification` that represents a time value.

### Diagrams

[Time](#), [Intervals](#), [Events](#)

### Generalizations

[ValueSpecification](#)

### Association Ends

- $\diamond$  `expr` : [ValueSpecification](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_expr\\_timeExpression::timeExpression](#))  
A `ValueSpecification` that evaluates to the value of the `TimeExpression`.
- `observation` : [Observation](#) [0..\*] (opposite [A\\_observation\\_timeExpression::timeExpression](#))  
Refers to the `Observations` that are involved in the computation of the `TimeExpression` value.

## Constraints

- `no_expr_requires_observation`  
If a `TimeExpression` has no `expr`, then it must have a single `observation` that is a `TimeObservation`.

```
inv: expr = null implies (observation->size() = 1 and observation->forall(oclIsKindOf(TimeObservation)))
```

## TimeInterval [Class]

### Description

A `TimeInterval` defines the range between two `TimeExpressions`.

### Diagrams

[Intervals](#)

### Generalizations

[Interval](#)



## Association Ends

- max : [TimeExpression](#) [1..1]{redefines [Interval::max](#)} (opposite [A\\_max\\_timeInterval::timeInterval](#))  
Refers to the TimeExpression denoting the maximum value of the range.
- min : [TimeExpression](#) [1..1]{redefines [Interval::min](#)} (opposite [A\\_min\\_timeInterval::timeInterval](#))  
Refers to the TimeExpression denoting the minimum value of the range.

## TimeObservation [Class]

### Description

A TimeObservation is a reference to a time instant during an execution. It points out the NamedElement in the model to observe and whether the observation is when this NamedElement is entered or when it is exited.

### Diagrams

[Time](#)

### Generalizations

[Observation](#)

### Attributes

- firstEvent : [Boolean](#) [1..1] = true  
The value of firstEvent is related to the event. If firstEvent is true, then the corresponding observation event is the first time instant the execution enters the event Element. If firstEvent is false, then the corresponding observation event is the time instant the execution exits the event Element.

## Association Ends

- event : [NamedElement](#) [1..1] (opposite [A\\_event\\_timeObservation::timeObservation](#))  
The TimeObservation is determined by the entering or exiting of the event Element during execution.

## ValueSpecification [Abstract Class]

### Description

A ValueSpecification is the specification of a (possibly empty) set of values. A ValueSpecification is a ParameterableElement that may be exposed as a formal TemplateParameter and provided as the actual parameter in the binding of a template.

### Diagrams

[Expressions](#), [Literals](#), [Time](#), [Intervals](#), [Object Nodes](#), [Activities](#), [Control Nodes](#), [Messages](#), [Lifelines](#), [Fragments](#), [Interaction Uses](#), [Types](#), [Constraints](#), [Events](#), [Features](#), [Properties](#), [Instances](#), [Actions](#), [Object Actions](#)

## Generalizations

[TypedElement](#), [PackageableElement](#)

## Specializations

[Duration](#), [Expression](#), [Interval](#), [LiteralSpecification](#), [OpaqueExpression](#), [TimeExpression](#), [InstanceValue](#)

## Operations

- `booleanValue() : Boolean [0..1]`  
The query `booleanValue()` gives a single Boolean value when one can be computed.  
  
`body: null`
- `integerValue() : Integer [0..1]`  
The query `integerValue()` gives a single Integer value when one can be computed.  
  
`body: null`
- `isCompatibleWith(p : ParameterableElement) : Boolean`  
The query `isCompatibleWith()` determines if this Parameterable element is compatible with the specified parameterable element. By default parameterable element P is compatible with parameterable element Q if the kind of P is the same or a subtype as the kind of Q. In addition, for a ValueSpecification, the type must be conformant with the type of the specified ParameterableElement (which must have a type, since it must be a kind of ValueSpecification).  
  
`body: p.oc1IsKindOf(self.oc1Type()) and self.type.conformsTo(p.oc1AsType(TypedElement).type)`
- `isComputable() : Boolean`  
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 ValueSpecifications 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 at least the value of all LiteralSpecifications.  
  
`body: false`
- `isNull() : Boolean`  
The query `isNull()` returns true when it can be computed that the value is null.  
  
`body: false`
- `realValue() : Real [0..1]`  
The query `realValue()` gives a single Real value when one can be computed.  
  
`body: null`
- `stringValue() : String [0..1]`  
The query `stringValue()` gives a single String value when one can be computed.  
  
`body: null`

- unlimitedValue() : [UnlimitedNatural](#) [0..1]  
The query unlimitedValue() gives a single UnlimitedNatural value when one can be computed.

body: null

## 8.7 Association Descriptions

### A\_behavior\_opaqueExpression [Association]

#### Diagrams

[Expressions](#)

#### Owned Ends

- opaqueExpression : [OpaqueExpression](#) [0..\*] (opposite [OpaqueExpression::behavior](#))

### A\_event\_durationObservation [Association]

#### Diagrams

[Time](#)

#### Owned Ends

- durationObservation : [DurationObservation](#) [0..\*] (opposite [DurationObservation::event](#))

### A\_event\_timeObservation [Association]

#### Diagrams

[Time](#)

#### Owned Ends

- timeObservation : [TimeObservation](#) [0..\*] (opposite [TimeObservation::event](#))

### A\_expr\_duration [Association]

#### Diagrams

[Time](#)

### Owned Ends

- duration : [Duration](#) [0..1]{subsets [Element::owner](#)} (opposite [Duration::expr](#))

### A\_expr\_timeExpression [Association]

#### Diagrams

[Time](#)

### Owned Ends

- timeExpression : [TimeExpression](#) [0..1]{subsets [Element::owner](#)} (opposite [TimeExpression::expr](#))

### A\_max\_durationInterval [Association]

#### Diagrams

[Intervals](#)

#### Generalizations

[A\\_max\\_interval](#)

### Owned Ends

- durationInterval : [DurationInterval](#) [0..\*]{redefines [A\\_max\\_interval::interval](#)} (opposite [DurationInterval::max](#))

### A\_max\_interval [Association]

#### Diagrams

[Intervals](#)

#### Specializations

[A\\_max\\_timeInterval](#), [A\\_max\\_durationInterval](#)

### Owned Ends

- interval : [Interval](#) [0..\*] (opposite [Interval::max](#))

## A\_max\_timeInterval [Association]

### Diagrams

[Intervals](#)

### Generalizations

[A\\_max\\_interval](#)

### Owned Ends

- timeInterval : [TimeInterval](#) [0..\*]{redefines [A\\_max\\_interval::interval](#)} (opposite [TimeInterval::max](#))

## A\_min\_durationInterval [Association]

### Diagrams

[Intervals](#)

### Generalizations

[A\\_min\\_interval](#)

### Owned Ends

- durationInterval : [DurationInterval](#) [0..\*]{redefines [A\\_min\\_interval::interval](#)} (opposite [DurationInterval::min](#))

## A\_min\_interval [Association]

### Diagrams

[Intervals](#)

### Specializations

[A\\_min\\_timeInterval](#), [A\\_min\\_durationInterval](#)

### Owned Ends

- interval : [Interval](#) [0..\*] (opposite [Interval::min](#))

## A\_min\_timeInterval [Association]

### Diagrams

[Intervals](#)

### Generalizations

[A\\_min\\_interval](#)

### Owned Ends

- timeInterval : [TimeInterval](#) [0..\*]{redefines [A\\_min\\_interval::interval](#)} (opposite [TimeInterval::min](#))

## A\_observation\_duration [Association]

### Diagrams

[Time](#)

### Owned Ends

- duration : [Duration](#) [0..1] (opposite [Duration::observation](#))

## A\_observation\_timeExpression [Association]

### Diagrams

[Time](#)

### Owned Ends

- timeExpression : [TimeExpression](#) [0..1] (opposite [TimeExpression::observation](#))

## A\_operand\_expression [Association]

### Diagrams

[Expressions](#)

### Owned Ends

- expression : [Expression](#) [0..1]{subsets [Element::owner](#)} (opposite [Expression::operand](#))

## A\_result\_opaqueExpression [Association]

### Diagrams

[Expressions](#)

### Owned Ends

- opaqueExpression : [OpaqueExpression](#) [0..\*] (opposite [OpaqueExpression::result](#))

## A\_specification\_durationConstraint [Association]

### Diagrams

[Intervals](#)

### Generalizations

[A\\_specification\\_intervalConstraint](#)

### Owned Ends

- durationConstraint : [DurationConstraint](#) [0..1]{redefines [A\\_specification\\_intervalConstraint::intervalConstraint](#)} (opposite [DurationConstraint::specification](#))

## A\_specification\_intervalConstraint [Association]

### Diagrams

[Intervals](#)

### Generalizations

[A\\_specification\\_owningConstraint](#)

### Specializations

[A\\_specification\\_timeConstraint](#), [A\\_specification\\_durationConstraint](#)

### Owned Ends

- intervalConstraint : [IntervalConstraint](#) [0..1]{redefines [A\\_specification\\_owningConstraint::owningConstraint](#)} (opposite [IntervalConstraint::specification](#))

## A\_specification\_timeConstraint [Association]

### Diagrams

[Intervals](#)

### Generalizations

[A\\_specification\\_intervalConstraint](#)

### Owned Ends

- timeConstraint : [TimeConstraint](#) [0..1]{redefines [A\\_specification\\_intervalConstraint::intervalConstraint](#)} (opposite [TimeConstraint::specification](#))

## A\_subExpression\_owningExpression [Association]

### Diagrams

[Expressions](#)

### Member Ends

- [StringExpression::subExpression](#)
- [StringExpression::owningExpression](#)



# 9 Classification

## 9.1 Summary

Classification is an important technique for organization. This clause specifies concepts relating to classification. The core concept is Classifier, an abstract metaclass whose concrete subclasses are used to classify different kinds of values. The other metaclasses in this clause represent the constituents of Classifiers, models of how Classifiers are instantiated using InstanceSpecifications, and various relationships between all of these concepts.

## 9.2 Classifiers

### 9.2.1 Summary

A Classifier represents a classification of objects according to their Features. Classifiers are organized in hierarchies by Generalizations. RedefinableElements may be redefined in the context of Generalization hierarchies.

### 9.2.2 Abstract Syntax

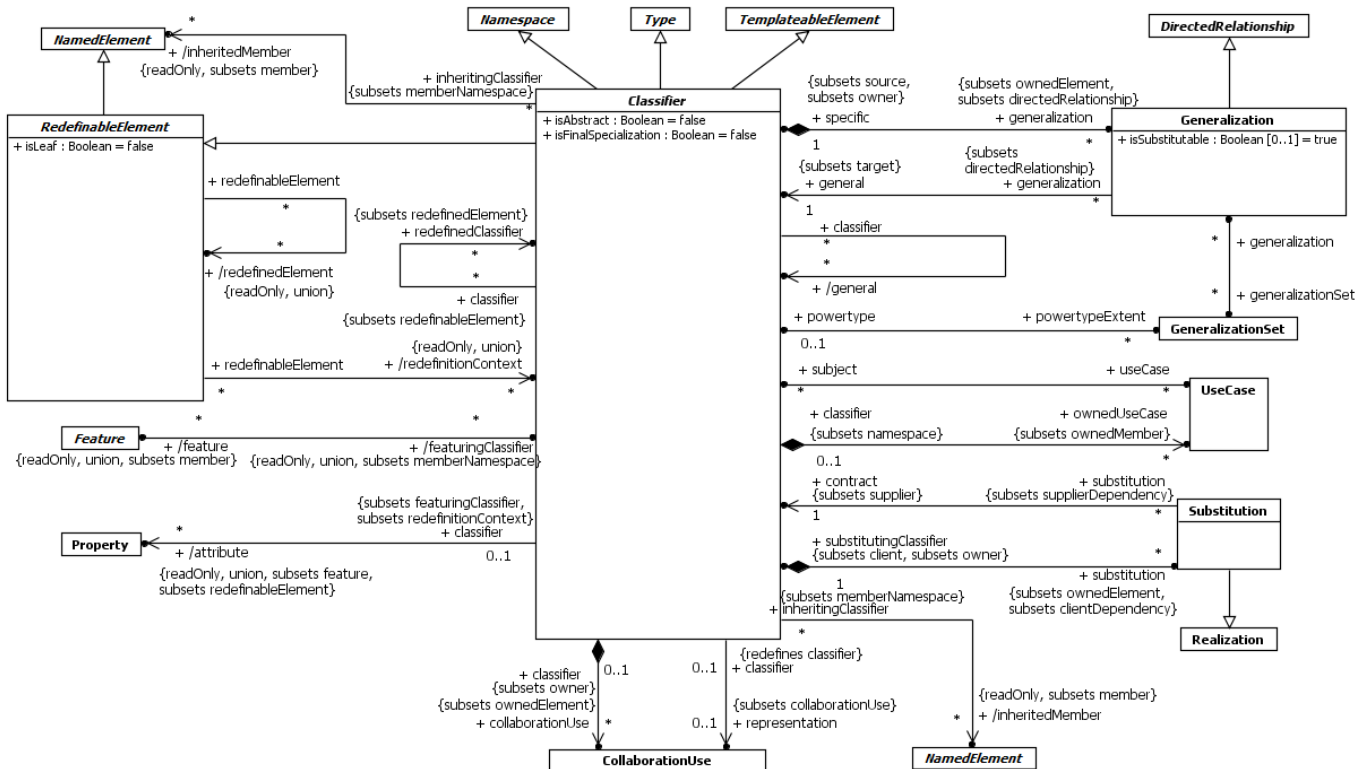


Figure 9.1 Classifiers

## 9.2.3 Semantics

### Classifiers

A Classifier has a set of Features, some of which are Properties called the attributes of the Classifier. Each of the Features is a member of the Classifier (see sub clause [7.4](#) Namespaces).

The values that are classified by a Classifier are called instances of the Classifier.

A Classifier may be redefined (see below).

A Classifier may own CollaborationUses that relate the Classifier to Collaborations. The Collaborations describes aspects of this Classifier. See [11.7](#) Collaborations.

A Classifier may own UseCases. See [18.1](#) Use Cases.

### Generalization

Generalizations define generalization/specialization relationships between Classifiers. Each Generalization relates a specific Classifier to a more general Classifier. Given a Classifier, the transitive closure of its general Classifiers is often called its *generalizations*, and the transitive closure of its specific Classifiers is called its *specializations*. The immediate generalizations are also called the Classifier's parents.

**NOTE.** The parent of a Classifier is not its owner.

An instance of a Classifier is also an (indirect) instance of each of its generalizations. Any Constraints applying to instances of the generalizations also apply to instances of the Classifier.

When a Classifier is generalized, certain members of its generalizations are *inherited*, that is they behave as though they were defined in the inheriting Classifier itself. For example, an inherited member that is an attribute has a value or collection of values in any instance of the inheriting Classifier, and an inherited member that is an Operation may be invoked on an instance of the inheriting Classifier.

The set of members that are inherited is called the *inheritedMembers*. Unless specified differently for a particular kind of Classifier, the *inheritedMembers* are members that do not have private visibility.

Type conformance means that if one Type conforms to another, than any instance of the first Type may be used as the value of a TypedElement whose type is declared to be the second Type. A Classifier is a Type, and conforms to itself and to all of its generalizations.

The *isAbstract* property of Classifier, when true, specifies that the Classifier is abstract, i.e., has no direct instances: every instance of the abstract Classifier shall be an instance of one of its specializations.

### Redefinition

Any member (that is a kind of RedefinableElement) of a generalization of a specializing Classifier may be redefined instead of being inherited. Redefinition is done in order to augment, constrain, or override the redefined member(s) in the context of instances of the specializing Classifier. When this occurs, the redefining member contributes to the structure or behavior of the specializing Classifier in place of the redefined member(s), which are hidden by the redefinition in the sense that any reference to a redefined member in the context of an instance of the specializing Classifier shall resolve to the redefining member.

The Classifier from which the member may be redefined is called the *redefinitionContext*. Although in the metamodel *redefinitionContext* has the multiplicity '\*', there are no cases in the UML specification where there is more than one *redefinitionContext*. The *redefinitionContext* is defined for each kind of RedefinableElement; it is often, but not always, the owner of the member.

A redefining element shall be consistent with the RedefinableElement it redefines, but it may add specific constraints or other details that are particular to instances of the specializing redefinitionContext that do not contradict constraints in the general context.

One redefining element may redefine multiple RedefinableElements. Furthermore, a RedefinableElement may be redefined multiple times, as long as it is unambiguous which definition applies for a particular instance.

The isLeaf property, when true for a particular RedefinableElement, specifies that it shall have no redefinitions.

The detailed semantics of redefinition vary for each specialization of RedefinableElement. There are various kinds of compatibility between a redefined element and its 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.

Classifier is itself a RedefinableElement. This can come into play when a Classifier is nested in a Class or Interface, which becomes the redefinitionContext. Redefining a Classifier in the context of a specializing Class or Interface has the effect of making any references to the redefined Classifier from an instance of the specializing Class or Interface resolve to the redefining Classifier.

## Substitution

A Substitution is a relationship between two Classifiers which signifies that the substitutingClassifier complies with the contract specified by the contract Classifier. This implies that instances of the substitutingClassifier are runtime substitutable where instances of the contract Classifier are expected. The Substitution dependency denotes runtime substitutability that is not based on specialization. Substitution, unlike specialization, does not imply inheritance of structure, but only compliance of publicly available contracts. It requires that:

- Interfaces implemented by the contract Classifier are also implemented by the substitutingClassifier or else the substitutingClassifier implements a more specialized Interface type.
- Any Port owned by the contract Classifier has a matching Port (see [11.3.3](#)) owned by the substitutingClassifier.

## 9.2.4 Notation

### Classifiers

Classifier is an abstract metaclass. It is nevertheless convenient to define in one place a default notation available for any concrete subclass of Classifier. Some specializations of Classifier have their own distinct notations.

The default notation for a Classifier is a solid-outline rectangle containing the Classifier's name, and with compartments separated by horizontal lines below the name. The name of the Classifier should be centered in boldface. For those languages that distinguish between uppercase and lowercase characters, Classifier names should begin with an uppercase character.

If the default notation is used for a Classifier, the metaclass of Classifier shall be shown in guillemets above the name. If no metaclass is shown, the metaclass shall be assumed to be Class. It is not customary to show «Class» explicitly, and a conforming tool is not required to be able to do that.

Any keywords (including stereotype names) should also be centered in plain face within guillemets above the Classifier name. If multiple keywords and/or stereotype names apply to the same model element, each may be enclosed in a separate pair of guillemets and listed one after the other. Alternatively they may all appear between the same pair of guillemets, separated by commas.

The name of an abstract Classifier is shown in italics, where permitted by the font in use. Alternatively or in addition, an abstract Classifier may be shown using the textual annotation {abstract} after or below its name.

Some compartments in Classifier shapes are mandatory and shall be supported by tools that exhibit concrete syntax conformance. Others are optional, in the sense that a conforming tool may not support such compartments.

Any compartment may be suppressed. A separator line is not drawn for a suppressed compartment. If a compartment is suppressed, no inference may be drawn about the presence or absence of elements in it.

The compartment named “attributes” contains notation for the Properties that are reached via the attribute property. The attributes compartment is mandatory and always appears above other compartments, if it is not suppressed.

The compartment named “operations” contains notation for Operations. The operations compartment is mandatory and always appears below the attributes compartment, if it is not suppressed. The operations compartment is used for Classifiers that own Operations, including Class (see [11.4](#)), DataType (see [10.2](#)) and Interface (see [10.4](#)).

The compartment named “receptions” contains notation for Receptions. The receptions compartment is mandatory and always appears below the operations compartment, if it is not suppressed. The receptions compartment is used for Classifiers that own Receptions, including Class (see [11.4](#)).

Any compartment which contains notation for Features may show those Features grouped under the literals public, private and protected, representing their visibility. The visibility literals are left-justified in the compartment with the Features’ notation appearing indented beneath them. The groups may appear in any order. Visibility grouping is optional: a conforming tool need not support it.

A conforming tool may provide the option to suppress individual Features in a compartment containing notation for Features.

A conforming tool may optionally support compartment naming. A compartment’s name may be shown to remove ambiguity, or it may be hidden. Compartment names should be centered and start with lower-case letters. Compartment names may contain spaces and should not contain punctuation (including guillemets).

If a Classifier has ownedMembers that are Classifiers (including Behaviors – see [13.2](#)), a conforming tool may provide the option to show the owned Classifiers, and relationships between them, diagrammatically nested within a separate compartment of the owning Classifier’s rectangle. Unless otherwise specified, the name of such a compartment shall be derived from the corresponding metamodel property, pluralized if that property has multiplicity greater than 1. So, for example, a compartment showing the contents of the property nestedClassifier for a Class (see [11.4.2](#)) shall be called “nested classifiers;” a compartment showing the contents of the property ownedBehavior for a BehaviorClassifier shall be called “owned behaviors.”

If a Classifier owns Constraints, a conforming tool may implement a compartment to show the owned Constraints listed within a separate compartment of the owning Classifier’s rectangle. The name of this optional compartment is “owned rules.”

## Other elements

A Generalization is shown as a line with a hollow triangle as an arrowhead between the symbols representing the involved Classifiers. The arrowhead points to the symbol representing the general Classifier.

Multiple Generalization relationships that reference the same general Classifier may be shown as separate lines with separate arrowheads. This notation is referred to as the “separate target style.” Alternatively they may be connected to the same arrowhead in the “shared target style.”

There is no general notation for RedefinableElement. See the subclasses of RedefinableElement for specific notations.

A Substitution is shown as a Dependency with the keyword «substitute» attached to it.

## 9.2.5 Examples

Examples for Classifier notation are shown under its various concrete subclasses, especially Class (see [11.4.4](#)).

Figure 9.2 illustrates Generalization notation with different target styles.

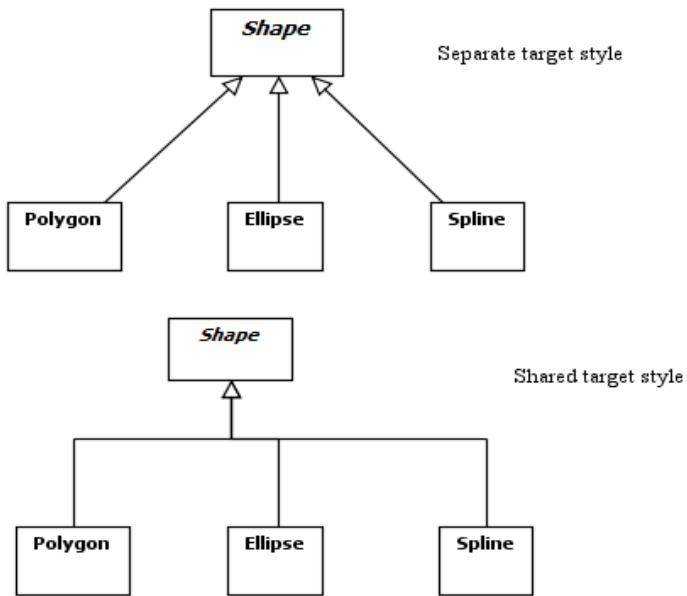


Figure 9.2 Generalization notation showing different target styles

In Figure 9.3, a generic Window class is substitutable in a particular environment by the Resizable Window class.

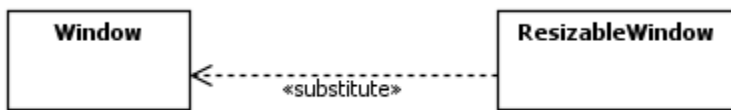


Figure 9.3 Example of Substitution notation

## 9.3 Classifier Templates

### 9.3.1 Summary

Classifier is a kind of TemplateableElement signifying that a Classifier may be parameterized. It is also (via PackageableElement) a kind of ParameterableElement so that a Classifier may be a formal TemplateParameter and may be specified as an actual parameter in a binding of a template. Sub clause 7.3 describes the general semantics of templates and their parameters.

### 9.3.2 Abstract Syntax

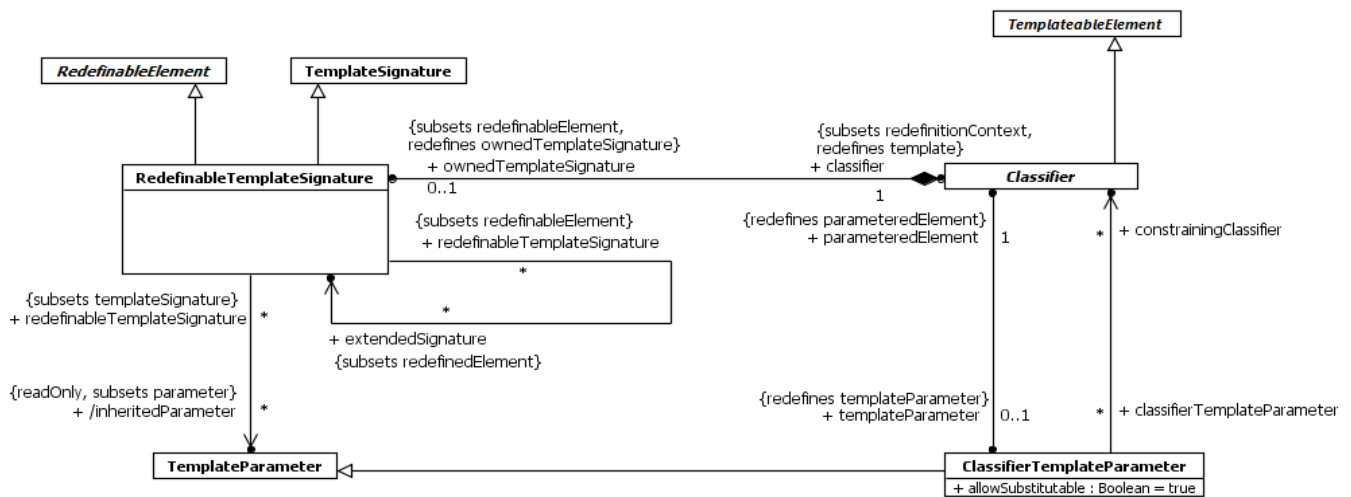


Figure 9.4 Classifier Templates

### 9.3.3 Semantics

#### Template and Bound Classifiers

The meanings of the terms *template* and *bound element* are defined in 7.3 – Templates.

A Classifier that is parameterized using a RedefinableTemplateSignature is called a *template Classifier*, while a Classifier with one or more TemplateBindings is called a *bound Classifier*.

The general semantics of templates as defined in subclause 7.3.3. There the details of how the contents are merged into a bound element are left open. In the case of Classifier the semantics are equivalent to inserting an anonymous general bound Classifier representing the intermediate result for each binding, and specializing all these intermediate results by the bound Classifier.

Members of the expanded bound Classifier may be used as actual parameters in a binding.

A bound Classifier may have contents in addition to those resulting from its bindings.

The parameters of a template Classifier can be any kind of TemplateParameter. Semantics and notation are only defined when the parameter is a Classifier, a LiteralSpecification, a Property or an Operation.

When the parameter is a Classifier, represented by a ClassifierTemplateParameter, the semantics and notation are defined in this clause.

When the parameter is a LiteralSpecification, the semantics and notation are as specified in [7.3](#).

When the parameter is an Operation, the semantics and notation are as specified in 9.6.

When the parameter is a Property, the semantics and notation are as specified in 9.5.

### Template Classifier specialization

RedefinableTemplateSignature specializes both TemplateSignature and RedefinableElement in order to allow the addition of new formal TemplateParameters in the context of a specializing template Classifier.

A RedefinableTemplateSignature redefines the RedefinableTemplateSignatures of all parent Classifiers that are templates. All the formal TemplateParameters of the extended (redefined) signatures are included as formal TemplateParameters of the extending signature, along with any TemplateParameters locally specified for the extending signature.

### Classifier Template Parameters

ClassifierTemplateParameter is a TemplateParameter where the parameteredElement is a Classifier in its capacity of being a kind of ParameterableElement.

All subclasses of Classifier (such as Class, Collaboration, Component, Datatype, Interface, Signal, and UseCases) may be parameterized, bound, and used as TemplateParameters. The same holds for Behavior as a subclass of Class, and thereby all subclasses of Behavior (such as Activity, Interaction, StateMachine).

The constrainingClassifier property of ClassifierTemplateParameter specifies a set of Classifiers that constrain the argument that can be used for the parameter. If there are any Classifiers in this set, then the argument shall be compatible with all of them, in the following sense:

- If allowSubstitutable is false, then compatibility means being the same as or a specialization of all of the constrainingClassifiers.
- If allowSubstitutable is true, then compatibility additionally allows a Substitution whose contract is a constrainingClassifier.

Furthermore, if there are any constrainingClassifiers, the parameteredElement shall be constrained as follows:

- If allowSubstitutable is false, then compatibility means being the same as or a direct specialization of all of the constrainingClassifiers, with no additional features.
- If allowSubstitutable is true, then compatibility additionally allows a Substitution whose contract is a constrainingClassifier.

In all cases, if the parameteredElement is not abstract then the Classifier used as an argument shall not be abstract. Apart from this, if the constrainingClassifier property is empty, there are no constraints on the Classifier that can be used as an argument. In this case the parameteredElement shall have no generalizations and no features, and allowSubstitutable shall be false.

### 9.3.4 Notation

See TemplateableElement for the general notation for displaying a template and a bound element.

When a bound Classifier is used directly as the type of a Property, then *<template-param-name>* acts as the *prop-type* of the Property in its notation (see Property).

The general notation for template parameters specified in 7.3.4 is extended for the parameters of a template Classifier to include the following:

*<template-parameter>* ::= *<classifier-template-parameter>* / *<operation-template-parameter>* / *<connectable-element-template-parameter>*

A ClassifierTemplateParameter extends the notation for a TemplateParameter to include an optional type constraint:

*<classifier-template-parameter>* ::= *<parameter-name>* [ ':' *<parameter-kind>* ] [ '>' *<constraint>* ] [ '=' *<default>* ]

*<constraint>* ::= [ '{contract }' ] *<classifier-name>*\*

*<default>* ::= *<classifier-name>*

The *parameter-kind* indicates the metaclass of the parameteredElement. It may be suppressed if it is 'Class.'

The *classifier-name* of *constraint* designates a constrainingClassifier, of which there may be zero or more, with the meaning specified in the semantics above. The 'contract' option indicates that allowSubstitutable is true.

### 9.3.5 Examples

The example shows a Class template (named FArray) with two formal TemplateParameters. The first formal TemplateParameter (named T) is an unconstrained class TemplateParameter: the metaclass Class has been suppressed from the diagram. The second formal TemplateParameter (named k) is a LiteralInteger that has a default of 10. There is also a bound Class (named AddressList) that substitutes Address for T and 3 for k.

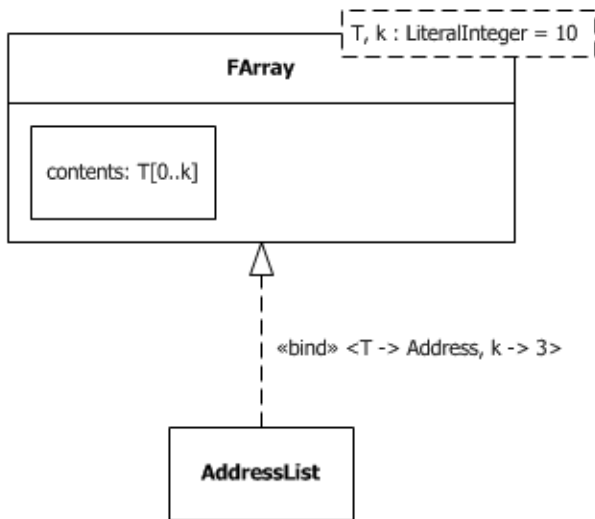
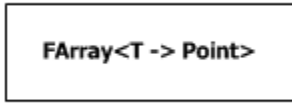


Figure 9.5 Template Class and Bound Class

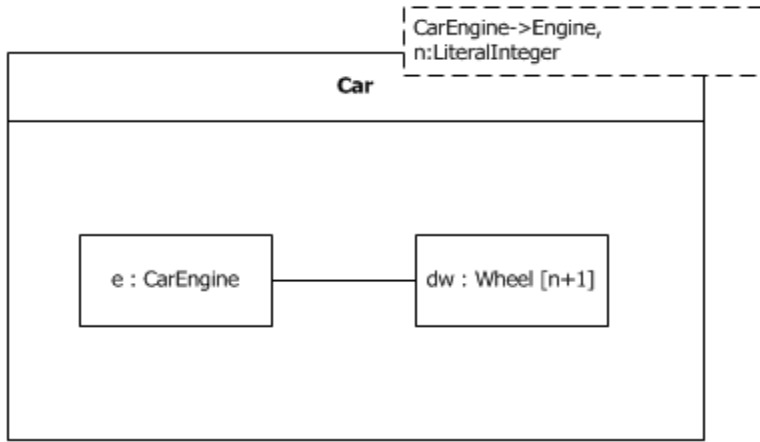
The following figure shows an anonymous bound Class that substitutes the Point class for T. As there is no substitution for k, the default (10) will be used.





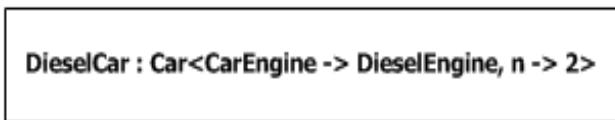
**Figure 9.6 Anonymous Bound Class**

The following figure shows a template Class (named Car) with two formal TemplateParameters. The first formal TemplateParameter (named CarEngine) is a Class that is constrained to conform to the Class called Engine. The second formal TemplateParameter (named n) is a LiteralInteger.



**Figure 9.7 Template Class with constrained Class parameter**

The following figure shows a bound Class (named DieselCar) that binds CarEngine to DieselEngine and n to 2: thus defining a class of 3-wheeled diesel cars.



**Figure 9.8 Bound Class**

## 9.4 Features

### 9.4.1 Summary

Features represent structural and behavioral characteristics of Classifiers.

### 9.4.2 Abstract Syntax

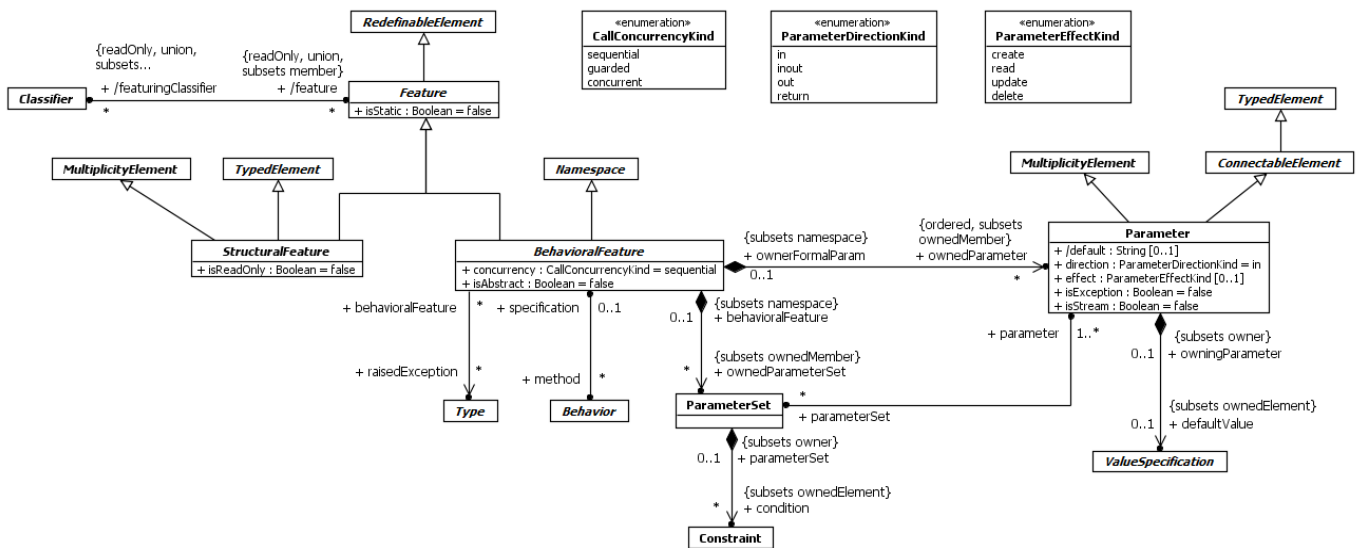


Figure 9.9 Features

### 9.4.3 Semantics

#### Features

Each Feature is associated with a Classifier called its `featuringClassifier`. Although `featuringClassifier` has the multiplicity ‘\*’, in the UML specification there is always one `featuringClassifier` which is the owner of the Feature. The Feature represents some structural or behavioral characteristic for its `featuringClassifier`.

The `isStatic` property specifies whether the characteristic relates to the Classifier’s instances considered individually (`isStatic=false`), or to the Classifier itself (`isStatic=true`). All semantics relating to Features that do not explicitly state whether the feature is static shall be assumed to refer to non-static Features. Where semantics are not explicitly specified for static Features, those semantics are undefined.

#### Structural Features

A **StructuralFeature** is a typed Feature of a Classifier that specifies the structure of instances of the Classifier.

The **StructuralFeatures** of a Classifier that are Properties are called the attributes of the Classifier (see 9.2.3). In UML, Property is the only kind of **StructuralFeature** so all of the **StructuralFeatures** of a Classifier are Properties, and hence attributes.

For each instance of a Classifier there is a value or collection of values for each direct or inherited non-static attribute of the Classifier, as follows:

- If the upper bound of the attribute is 1, there shall be a single value whose Type conforms to the Type of the attribute.
- If the upper bound of the attribute is not 1, there shall be a collection of values whose size is not greater than the upper bound, each of whose Types conforms to the Type of the attribute.

If a StructuralFeature is marked with `isStatic = true`, then the two bullet points above are relative to the Classifier itself considered as an identifiable individual within some execution scope, rather than to individual instances. The execution scope is not defined in UML.

Inherited static StructuralFeatures shall have one of two alternative semantics, within a given execution scope:

1. The value of the StructuralFeature is always the same for any inheriting Classifier as its value for the owning Classifier.
2. The StructuralFeature has a separate and independent value for its owning Classifier and for each Classifier that inherits it.

If a StructuralFeature is marked with `isReadOnly true`, then it may not be updated once it has been assigned an initial value. Conversely, when `isReadOnly` is false, the value may be modified at any time.

## Behavioral Features

A non-static BehavioralFeature specifies that an instance of its featuringClassifier will react to an *invocation* of the BehavioralFeature by carrying out a specific behavioral response. Subclasses of BehavioralFeature model different behavioral aspects of a Classifier.

The list of ownedParameters describes the order, type, and direction of arguments that may be given when the BehavioralFeature is invoked, or which are output and returned when the invocation completes.

The ownedParameters with direction `in` or `inout` define the arguments that shall be provided when invoking the BehavioralFeature. The ownedParameters with direction `out`, `inout`, or `return` define the arguments that will be output and returned from a successful invocation.

A BehavioralFeature may raise an exception during its invocation. Possible exception types may be specified by attaching them to the BehavioralFeature using the `raisedException` association.

One way to define the behavioral response of a BehavioralFeature is to specify one or more Behaviors as methods that implement the BehavioralFeature. An invocation of the BehavioralFeature then results in the execution of one of the associated methods (as further discussed in sub clause [13.2](#) on Behaviors). The `isAbstract` property, when true, specifies that the BehavioralFeature does not have any methods implementing it, with the expectation that an implementation will be supplied by a more specific element.

The concurrency property specifies the semantics of concurrent calls to the same instance. Its type is `CallConcurrencyKind`, an enumeration with the following literals:

sequential	No concurrency management mechanism is associated with the BehavioralFeature and, therefore, concurrency conflicts may occur. Instances that invoke a BehavioralFeature need to coordinate so that only one invocation to a target on any BehavioralFeature occurs at once.
guarded	Multiple invocations of a BehavioralFeature may occur simultaneously to one instance, but only one is allowed to commence. The others are blocked until the performance of the currently executing BehavioralFeature is complete. It is the responsibility of the system designer to ensure that deadlocks do not occur due to simultaneous blocking.
concurrent	Multiple invocations of a BehavioralFeature may occur simultaneously to one instance and all of them may proceed concurrently.

## Parameters

A Parameter is a specification of an argument used to pass information into or out of an invocation of a BehavioralFeature. The Type and Multiplicity of a Parameter restrict what values may be passed, how many, and whether the values are ordered. The

Multiplicity defines a lower and upper bound on the values passed to the Parameter at runtime. A lower bound of zero means the Parameter is optional. Actions using the Parameter may execute without having a value for optional Parameters. A lower bound greater than zero means values for the Parameter are required to arrive sometime during the execution of the action.

If a defaultValue 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 BehavioralFeature.

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

The direction property specifies whether a value is passed into, out of, or both into and out of the owning BehavioralFeature. A single Parameter may be distinguished as a return Parameter. Its type is ParameterDirectionKind, an enumeration of the following literal values:

in	Indicates that Parameter values are passed in by the caller.
inout	Indicates that Parameter values are passed in by the caller and then back out to the caller.
out	Indicates that Parameter values are passed out to the caller.
return	Indicates that Parameter values are passed as return values back to the caller.

The effect property may be used to declare additional indications of the effect on values passed in or out of a Parameter. It is a declaration of the modeler's intent, and does not have execution semantics: the modeler must ensure that the Parameter has the stated effect. Its type is ParameterEffectKind, an enumeration of the following literal values:

create	Indicates that the value passed out of the Parameter is created by the invocation.
read	Indicates that the value passed across the Parameter is read by the invocation.
update	Indicates that the value passed across the Parameter is updated by the invocation.
delete	Indicates that the value passed into the Parameter is deleted by the invocation.

Only in and inout Parameters may have a delete effect. Only out, inout, and return Parameters may have a create effect.

The isException property applies to output Parameters. An output posted to a Parameter with isException true during an invocation of a BehavioralFeature excludes outputs from being posted to any other outputs of the BehavioralFeature during the same invocation. The type of such an exception Parameter may be included in the raisedException set, but does not have to be included.

The isStream property, when true, designates a streaming Parameter. A streaming Parameter expresses the expectation that any Behavior implementing this feature will exhibit streaming behavior on this Parameter – see subclause 13.2. The semantics for a Parameter designated as streaming when the implementing Behavior does not exhibit streaming behavior are undefined.

A ParameterSet owned by a BehavioralFeature is an element that provides alternative sets of inputs or outputs that the Behaviors that implements that BehavioralFeature may use. The Parameters in a ParameterSet shall all be inputs or all outputs of the same BehavioralFeature: a ParameterSet with all inputs is called an input ParameterSet, and one with all outputs is called an output ParameterSet.

A BehavioralFeature with input ParameterSets may only accept inputs from Parameters in one of the sets per invocation. A BehavioralFeature with output ParameterSets may only return outputs to the Parameters in one of the sets per invocation. The semantics of conditions on input and output ParameterSets of BehavioralFeatures is the same as Operation preconditions and postconditions, respectively, but apply to only to invocations that accept inputs to or return outputs from Parameters in the ParameterSet having the condition.

## 9.4.4 Notation

There is no general notation for Feature. Subclasses define their specific notation.

There is no general notation for Parameter. Specific subclasses of BehavioralFeature define notation for their Parameters.

There is no general notation for ParameterSet. Specific subclasses of BehavioralFeature define notation for ParameterSets.

Static Features are underlined.

Where Features are shown in lists, an ellipsis (...) as the final element of a list of Features may be used to indicate that additional Features exist but are not shown in that list.

A read only StructuralFeature is shown using {readOnly} as part of the notation for the StructuralFeature. This annotation may be suppressed, in which case it is not possible to determine its value from the diagram. Alternatively a confirming tool may only allow suppression of the {readOnly} annotation when isReadOnly=false. In this case it is possible to assume this value in all cases where {readOnly} is not shown.

Feature redefinitions may either be explicitly notated with the use of a {redefines <x>} property string on the Feature or implicitly by having a Feature with the same name as another Feature in one of the owning Classifier's more general Classifiers. In both cases, the redefined Feature shall conform to the compatibility constraint on the redefinitions.

## 9.5 Properties

### 9.5.1 Summary

Properties are StructuralFeatures that represent the attributes of Classifiers, the memberEnds of Associations, and the parts of StructuredClassifiers.

### 9.5.2 Abstract Syntax

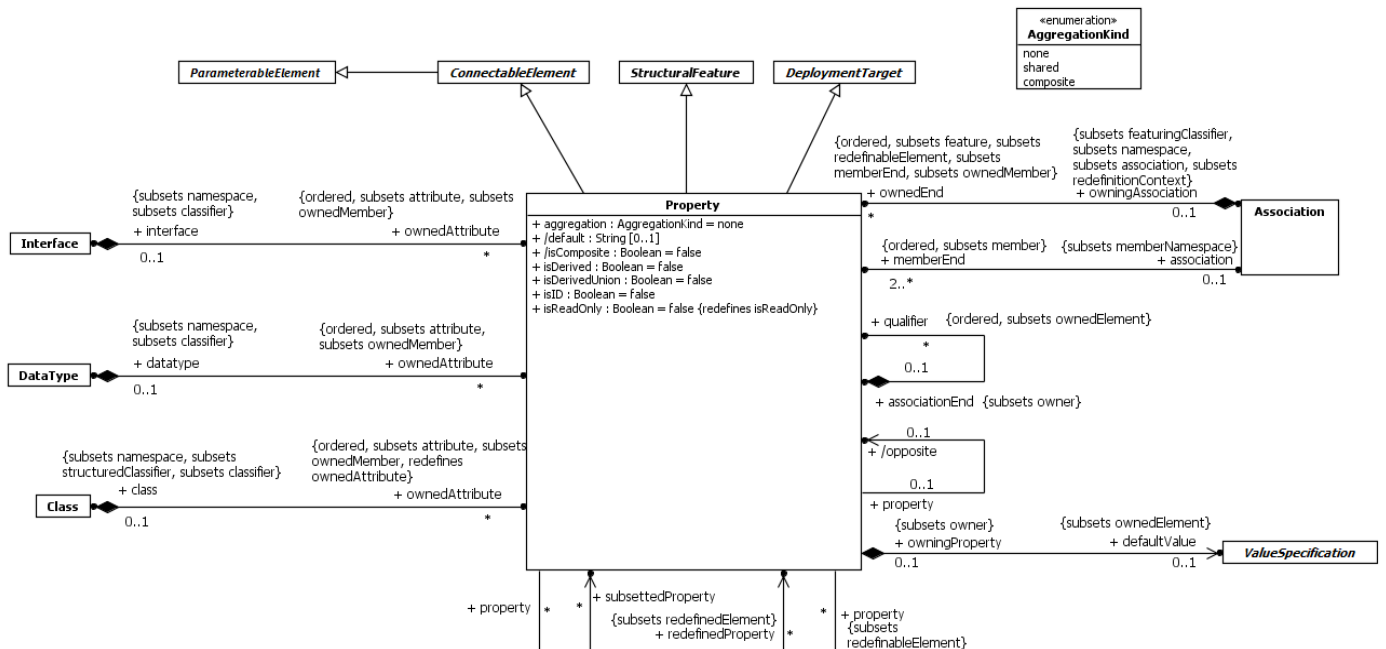


Figure 9.10 Properties

### 9.5.3 Semantics

A Property may represent an attribute of a Classifier, a memberEnd of an Association, or in some cases both simultaneously.

A useful convention for general modeling scenarios is that a Property whose type is a kind of Class is an Association end, while a property whose type is a kind of DataType is not. This convention is not enforced by UML.

A 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 a non-static 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's memberEnd, the value or values are related to the instance or instances at the other end(s) of the association (see 11.5 Associations). When a Property is a static attribute of a Classifier, the value or values are related to the Classifier itself within some execution scope.

A Property that is a memberEnd may itself have other Properties that serve as qualifiers.

When a Property is owned by a Classifier other than an Association via ownedAttribute, then it represents an attribute of the Classifier. When related to an Association via memberEnd it represents an end of the Association. For a binary Association, it may

be both at once. 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) Classifier. This set of Classifiers 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 Classifiers at the other end or ends of the Association.

If there is a `defaultValue` 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 has `isDerived = true`, it is derived and its value or values may 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 may not directly change values). But where a derived Property is changeable, an implementation is expected to make appropriate changes to the model in order for all the constraints to be met, in particular the derivation constraint for the derived Property. The derivation for a derived Property may be specified by a constraint.

Property is indirectly a kind of `RedefinableElement`, so Properties may be redefined. The name and visibility of a Property are not required to match those of any Property it redefines.

A derived Property may redefine one which is not derived. An implementation shall 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.

A Property has an aggregation property, of type `AggregationKind`. This is an enumeration with the following literal values:

none	Indicates that the Property has no aggregation semantics.
shared	Indicates that the Property has shared aggregation semantics. Precise semantics of shared aggregation varies by application area and modeler.
composite	Indicates that the Property is aggregated compositely, i.e., the composite object has responsibility for the existence and storage of the composed objects (parts).

Composite aggregation is a strong form of aggregation that requires a part (see [11.2.3](#)) instance be included in at most one composite instance at a time. If a composite instance is deleted, all of its parts are normally deleted with it.

**NOTE.** A part may (where otherwise allowed) be removed from a composite instance before the composite instance is deleted, and thus not be deleted as part of the composite instance.

Compositions may be linked in a directed acyclic graph with transitive deletion characteristics; that is, deleting an element in one part of the graph will also result in the deletion of all elements of the subgraph below that element. The precise lifecycle semantics of composite aggregation is intentionally not specified. The order and way in which composed instances are created is intentionally not defined. The semantics of composite aggregation when the container or part is typed by a `DataType` are intentionally not specified.

A Property may be marked as the subset of another `subsettingProperty`. In this case, the collection of values denoted by the subsetting property in some context shall be included in (or the same as) the collection of values denoted by the `subsettingProperty` in the same context.

A Property may be marked as being a derived union, by setting `isDerivedUnion` to true. 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 shall be null or the same.

A Property may be marked, via the property `isID`, as being (part of) the identifier (if any) for Classifiers of which it is a member. The interpretation of this is left open but this could be mapped to implementations such as primary keys for relational database tables or ID attributes in XML. If multiple Properties are marked (possibly in generalizing Classifiers) then it is the combination of the (Property, value) tuples that will logically provide the uniqueness for any instance. Hence there is no need for any

specification of order and it is possible for some (but not all) of the Property values to be empty. If the Property is multi valued then all values are included.

A qualifier of a Property that is an Association end declares a partition of the set of associated instances with respect to an instance at the qualified end (the qualified instance is at the end to which the qualifier is attached). A *qualifier value* is a tuple comprising one value for each qualifier.

In the common case in which the multiplicity at the end(s) opposite the qualified end is 0..1, the qualifier value is unique with respect to the qualified instance, and designates at most one associated instance per opposite end. In the general case of multiplicity 0..\*, the set of associated instances is partitioned into subsets, each selected by a given qualifier value. In the case of multiplicity 1 or 0..1, the qualifier has both semantic and implementation consequences. In the case of multiplicity 0..\*, it has no real semantic consequences but suggests an implementation that facilitates easy access of sets of associated instances linked by a given qualifier value.

**NOTE.** A qualified multiplicity whose lower bound is zero indicates that a qualifier value may be absent; while a lower bound of 1 indicates that a qualifier value shall be present. The latter is reasonable only for qualifiers with a finite set of values (such as enumerated values or integer ranges) that represent full tables indexed by the qualifier value.

Property specializes ParameterableElement to specify that a Property may be exposed as a formal ConnectableElementTemplateParameter (see [11.2.3](#)), and provided as an actual parameter in a binding of a template. Within a template a Property TemplateParameter may be used like any other accessible Property. Any references to the Property TemplateParameter within the template will end up being a reference to the actual Property in the bound element.

## 9.5.4 Notation

The following general notation is defined for Properties.

**NOTE.** Some specializations of Property may also have additional notational forms. These are covered in the appropriate Notation sub clauses of those classes.

`<property> ::= [visibility] [ '/' ] <name> [ ':' <prop-type> ] [ '[' <multiplicity> ']' ] [ '=' <default> ] [ '{' <prop-modifier > [ ',' <prop-modifier > ]* '}' ]`

where:

- *visibility* is the visibility of the Property. (See VisibilityKind - sub clause [7.4](#).)  
`<visibility> ::= '+' | '-' | '#' | '~'`
- '/' signifies that the Property is derived.
- *name* is the name of the Property.
- *prop-type* is the name of the Classifier that is the type of the Property.
- *multiplicity* is the multiplicity of the Property. If this term is omitted, it implies a multiplicity of 1 (exactly one). (See MultiplicityElement – sub clause [7.5](#).)
- *default* is an expression that evaluates to the default value or values of the Property.
- *prop-modifier* indicates a modifier that applies to the Property.  
`<prop-modifier> ::= 'readOnly' | 'union' | 'subsets' <property-name> | 'redefines' <property-name> | 'ordered' | 'unique' | 'nonunique' | 'seq' | 'sequence' | 'id' | <prop-constraint>`

where:



- ‘readOnly’ means that the Property is read only.
- ‘union’ means that the Property is a derived union of its subsets.
- ‘subsets’ *<property-name>* means that the Property is a proper subset of the Property identified by *<property-name>*.
- ‘redefines’ *<property-name>* means that the Property redefines an inherited Property identified by *<property-name>*.
- ‘ordered’ means that the Property is ordered, i.e., `isOrdered = true`.
- ‘unique’ means that there are no duplicates in a multi-valued Property, i.e., `isUnique = true`.
- ‘nonunique’ means that there may be duplicates in a multi-valued Property, i.e., `isUnique = false`.
- ‘seq’ or ‘sequence’ means that the property represents an ordered bag, i.e., `isUnique = false` and `isOrdered = true`
- ‘id’ means that the Property is part of the identifier for the class.
- *<prop-constraint>* is an expression that specifies a constraint that applies to the Property.

A qualifier is shown as a small rectangle attached to the end of an association path between the final path segment and the symbol of the Classifier that it connects to. The qualifier rectangle should be smaller than the attached class rectangle, unless this is not practical. The qualifier rectangle is part of the association path, not part of the Classifier. The qualifier rectangle is attached to the end of the association path that represents the memberEnd that owns the qualifier.

The multiplicity attached to the target end denotes the possible cardinalities of the set of target instances selected by the pairing of a qualified instance and a qualifier value.

The qualifier attributes are drawn within the qualifier box. There may be one or more attributes, shown one to a line. Qualifier attributes have the same notation as Classifier attributes, except that initial value expressions are not meaningful.

It is permissible (although somewhat rare), to have a qualifier on every end of a single association.

A qualifier may not be suppressed.

The notation for the aggregation of a Property is defined in [11.5](#) Associations.

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

In a Classifier, the individual properties of an attribute may be shown in columns rather than as a continuous string.

In a Classifier, an attribute may also be shown using association notation, with no adornments at the tail of the arrow.

The notation for a ConnectableElementTemplateParameter used to parameterize a template Classifier by a Property is this:

*<connectable-element-template-parameter> ::= <property-name> ‘: Property’*

### 9.5.5 Examples

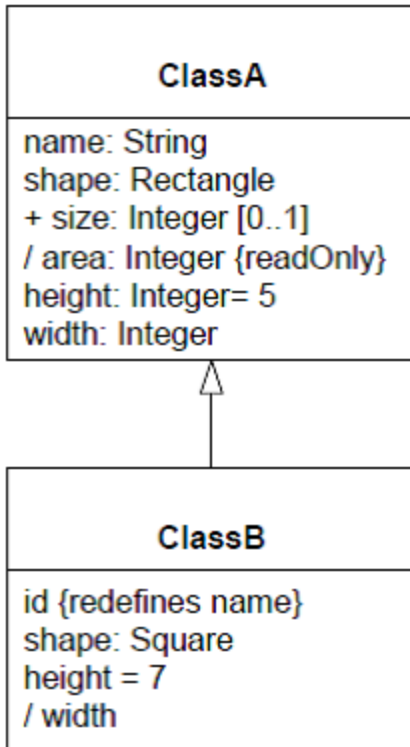


Figure 9.11 Examples of attributes

The attributes in Figure 9.11 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 of 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 that overrides the ClassA default of 5.
- ClassB::width is a derived attribute that redefines ClassA::width, which is not derived.

Figure 9.12 shows how an attribute may be shown using association notation.

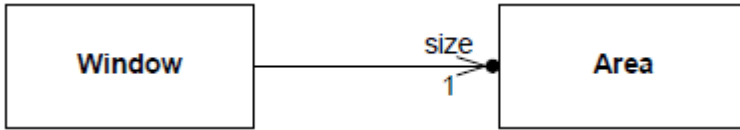


Figure 9.12 Association-like notation for attributes

Figure 9.13 illustrates some qualified associations. The left diagram shows that given a Bank, a particular accountNo identifies zero or one Person. The qualifier is the property accountNo, and the qualified object is the Bank. The qualifier is owned by the unnamed property at the Bank end of the association, i.e., the Property whose type is Bank.

The right diagram shows how an individual Square on the Chessboard may be identified by rank and file; in this case because the multiplicity is 1, the diagram shows that every possible value for Rank and File indicates an individual Square. In this case the qualifiers are owned by the unnamed association end property whose type is Chessboard, while the opposite property whose type is Square is marked with aggregation = composite.



Figure 9.13 Qualified associations

## 9.6 Operations

### 9.6.1 Summary

An Operation is a BehavioralFeature that may be owned by an Interface, DataType or Class. Operations may also be templated and used as template parameters.

### 9.6.2 Abstract Syntax

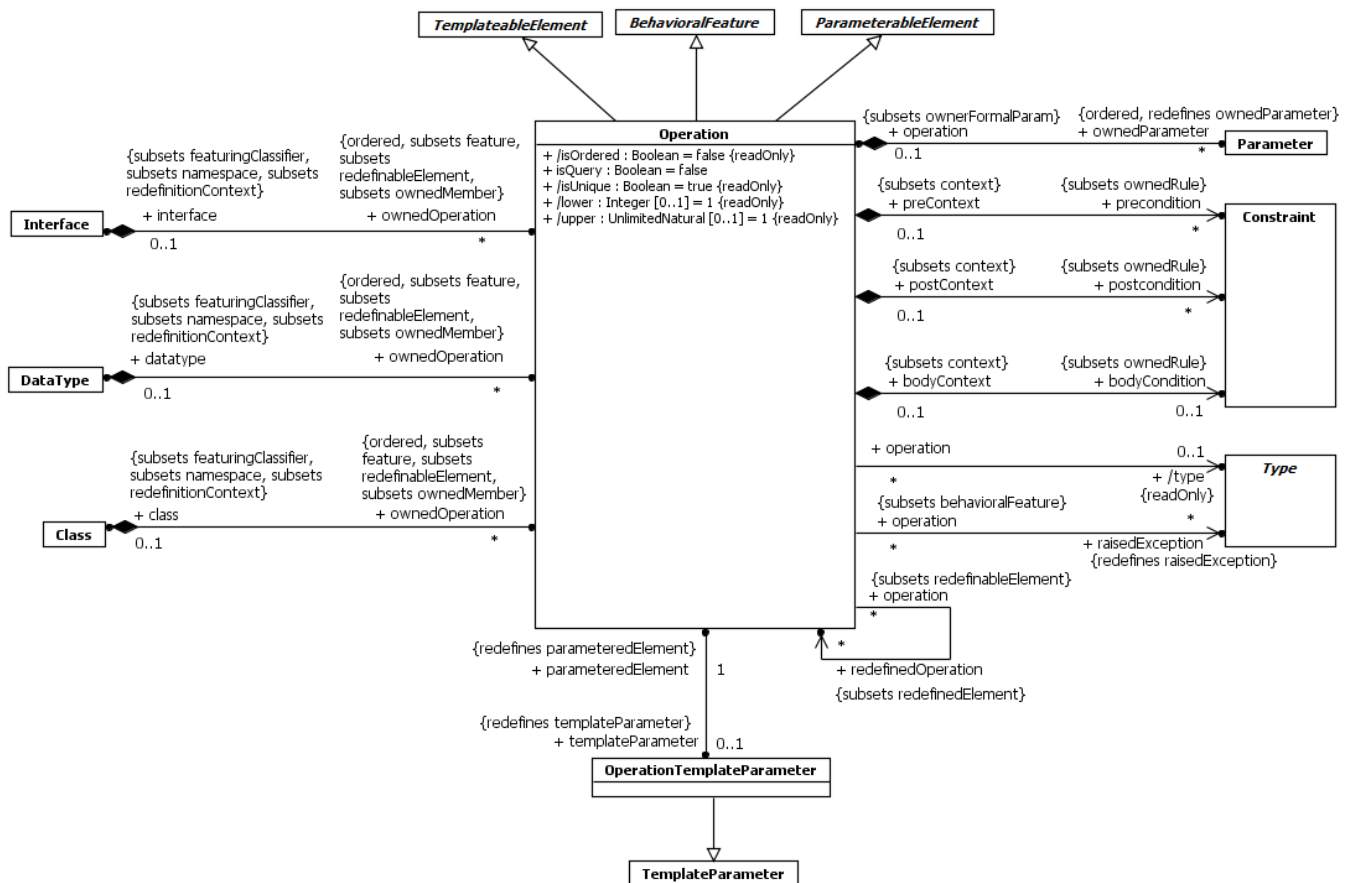


Figure 9.14 Operations

### 9.6.3 Semantics

#### Operations

An Operation is a BehavioralFeature of an Interface, DataType, or Class. An Operation may be directly invoked on instances of its featuringClassifiers. The Operation specifies the name, type, Parameters, and Constraints for such invocations.

If there is a return Parameter, the type and multiplicity of this Parameter is the same as the type and multiplicity of the Operation itself.

The preconditions for an Operation define conditions that shall be true when the Operation is invoked. These preconditions may be assumed by an implementation of this Operation. The behavior of an invocation of an Operation when a precondition is not satisfied is intentionally undefined.

The postconditions for an Operation define conditions that will be true when the invocation of the Operation completes successfully, assuming the preconditions were satisfied. These postconditions shall 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 may 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 featuringClassifier. This redefinition may specialize the types of the owned Parameters, add new preconditions or postconditions, add new raisedExceptions, or otherwise refine the specification of the Operation.

Different type-conformance systems adopt different schemes for how the types of parameters and results may vary when an Operation is redefined in a specialization. When the type may not vary, it is called invariance. When the parameter type may be specialized in a specialized type, it is called covariance. When the parameter type may be generalized in a specialized type, it is called contravariance. In UML, s

If the isQuery property is true, an invocation of the Operation shall not modify the state of the instance or any other element in the model.

An Operation may be owned by and in the namespace of a Class, DataType or Interface that provides the context for its possible redefinition. The owning classifier of the Operation provides its redefinitionContext.

## Template Operations

Operation specializes TemplateableElement in order to support specification of template Operations and bound Operations. Bound Operations must be owned by a Classifier. If the original operation was defined with a Behavior, then the bound element has to be owned by a Classifier that is consistent with that Behavior. This means one of three things: (a) the bound operation appears in the same Classifier as the template; (b) the bound operation appears in a subtype of the template's owner; (c) the template was defined without side-effects in a static class and the bound one can then appear anywhere.

## Operation Template Parameters

An Operation may be exposed by a template as a formal template parameter via an OperationTemplateParameter. OperationTemplateParameter is a kind of TemplateParameter where the parametered element is an Operation. Within a template Classifier an OperationTemplateParameter may be used like any other accessible Operation. Any references to the OperationTemplateParameter within the template will end up being a reference to the actual Operation in the bound Classifier. For example, a call to the OperationTemplateParameter will be a call to the actual Operation.

A default for an OperationTemplateParameter must be an Operation with the same parameter types, directions, and multiplicities as the exposed Operation.

## 9.6.4 Notation

If shown in a diagram, an Operation is shown as a text string of the form:

```
[<visibility>] <name> '(' [<parameter-list> ] ')' [':' [<return-type>] [ '[' <multiplicity> ']' ] ]  
[ '{' <oper-property> [ ';' <oper-property> ]* '}' ] ]
```

where:

- *<visibility>* is the visibility of the Operation (see [7.4](#)).  
*<visibility>* ::= '+' | '-' | '#' | '~'
- *<name>* is the name of the Operation.
- *<return-type>* is the type of the return result Parameter if the Operation has one defined.
- *<multiplicity>* is the multiplicity of the return type (see [7.5](#)).
- *<oper-property>* indicates the properties of the Operation.  
*<oper-property>* ::= 'redefines' *<oper-name>* | 'query' | 'ordered' | 'unique' | *<oper-constraint>*

where:

- 'redefines' *<oper-name>* means that the Operation redefines an inherited Operation identified by *<oper-name>*.
- 'query' means that the Operation does not change the state of the system.
- 'ordered' means that the values of the return Parameter are ordered.
- 'unique' means that the values returned by the Parameter have no duplicates.
- *<oper-constraint>* is a constraint that applies to the Operation.
- *<parameter-list>* is a list of Parameters of the Operation in the following format:  
*<parameter-list>* ::= *<parameter>* ['<parameter>']\*  
*<parameter>* ::= [*<direction>*] *<parameter-name>* ':' *<type-expression>*  
                  ['[<multiplicity>']][ '=' *<default>*]  
                  ['{' *<parm-property>* ['<parm-property>]\* '}' ]

where:

- *<direction>* ::= 'in' | 'out' | 'inout' (defaults to 'in' if omitted).
- *<parameter-name>* is the name of the Parameter.
- *<type-expression>* is an expression that specifies the type of the Parameter.
- *<multiplicity>* is the multiplicity of the Parameter. (See MultiplicityElement – sub clause [7.5](#)).
- *<default>* is an expression that defines the value specification for the default value of the Parameter.
- *<parm-property>* indicates additional property values that apply to the Parameter.

The parameter list may be suppressed.

The return result of the Operation may be expressed as a return parameter, or as the type of the Operation. For example:

```
toString(return : String)
```

means the same thing as

```
toString() : String
```

The TemplateParameters of a template Operation are in a list between the name of the Operation and the Parameters of the Operation.

[<visibility>] <name> ‘< <template-parameter-list> ‘>’ ‘(‘ [<parameter-list>] ‘)’ [‘:’ [<return-type>] [‘[’ <multiplicity> ‘]’] [‘{’ <oper-property> [‘,’ <oper-property>]\* ‘}’]’]

The TemplateParameter bindings of a bound template Operation are in a list between the name of the Operation and the Parameters of the Operation.

[<visibility>] <name> ‘<< <binding-expression-list> ‘>>’ ‘(‘ [<parameter-list>] ‘)’ [‘:’ [<return-type>] [‘[’ <multiplicity> ‘]’] [‘{’ <oper-property> [‘,’ <oper-property>]\* ‘}’]’]

where < binding-expression-list > ::= <binding-expression> [‘,’ <binding-expression>]\*, and <binding-expression> is defined in [7.3.4](#).

Within the notation for formal TemplateParameters and TemplateParameter bindings, an Operation is shown as <operation-name> ‘(‘<parameter-list> ‘)’.

An OperationTemplateParameter extends the notation for a TemplateParameter to include the Parameters for the Operation:

<operation-template-parameter> ::= <parameter> [ ‘: Operation’] [‘=’ <default>]

<parameter> ::= <operation-name> ‘(‘<parameter-list> ‘)’

<default> ::= <operation-name> ‘(‘<parameter-list> ‘)’

The notation in class diagrams for exceptions and streaming Parameters on Operations has the keywords “exception” or “stream” in the property string.

## 9.6.5 Examples

Normal Operations:

display ()

-hide ()

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

+toString (): String

A template Operation:

f <T:Class>(x : T)

A binding of that template Operation.

f << T -> Window >>(x : Window)

**NOTE.** Parameters may be suppressed; they are calculated by the binding.

## 9.7 Generalization Sets

### 9.7.1 Summary

GeneralizationSet provides a way to group Generalizations into orthogonal dimensions. A GeneralizationSet may be associated with a Classifier called its powertype. These techniques provide additional expressive power for organizing classification hierarchies.

### 9.7.2 Abstract Syntax

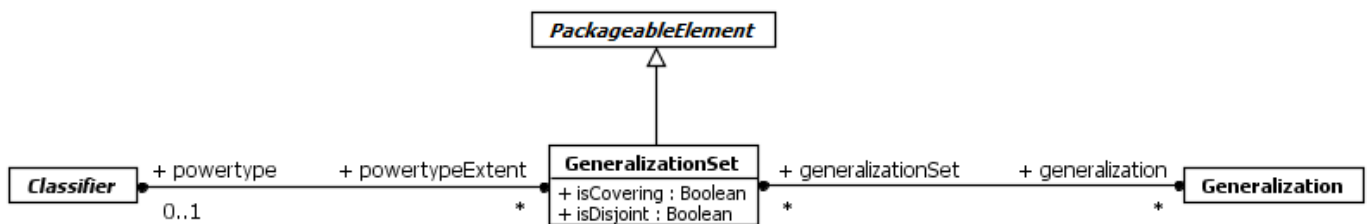


Figure 9.15 Generalization Sets

### 9.7.3 Semantics

Generalizations may be grouped to represent orthogonal dimensions of generalization. Each group is represented by a GeneralizationSet. The generalizationSet property designates the GeneralizationSets to which the Generalization belongs. All of the Generalizations in a particular GeneralizationSet shall have the same general Classifier.

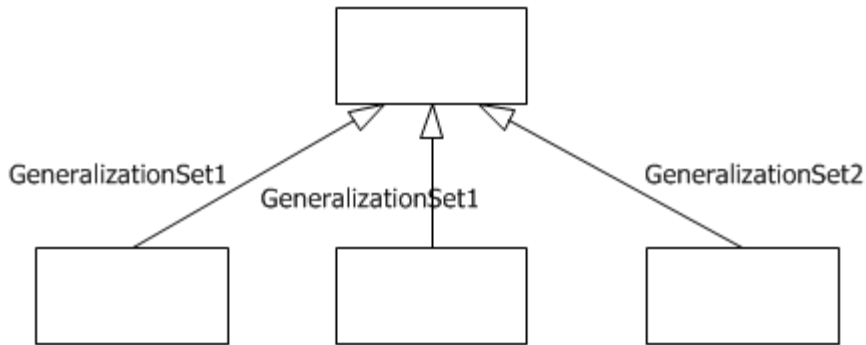
The isCovering property of GeneralizationSet specifies whether the specific Classifiers of the Generalizations in that set are complete, in the following sense: if isCovering is true, then every instance of the general Classifier is an instance of (at least) one of the specific Classifiers. The isDisjoint property specifies whether the specific Classifiers of the Generalizations in that set may overlap, in the following sense: if isDisjoint is true, then no instance of any of the specific Classifiers may also be an instance of any other of the specific Classifiers. By default, both properties are false.

A GeneralizationSet may optionally be associated with a Classifier called its powertype. This means that for every Generalization in the GeneralizationSet, the specializing Classifier is uniquely associated with an instance of the powertype, i.e., there is a 1-1 correspondence between instances of the powertype and specializations in the GeneralizationSet, so that the powertype instances and the corresponding Classifiers may be treated as semantically equivalent. How this semantic equivalence is implemented and how its integrity is maintained is not defined within the scope of UML.

### 9.7.4 Notation

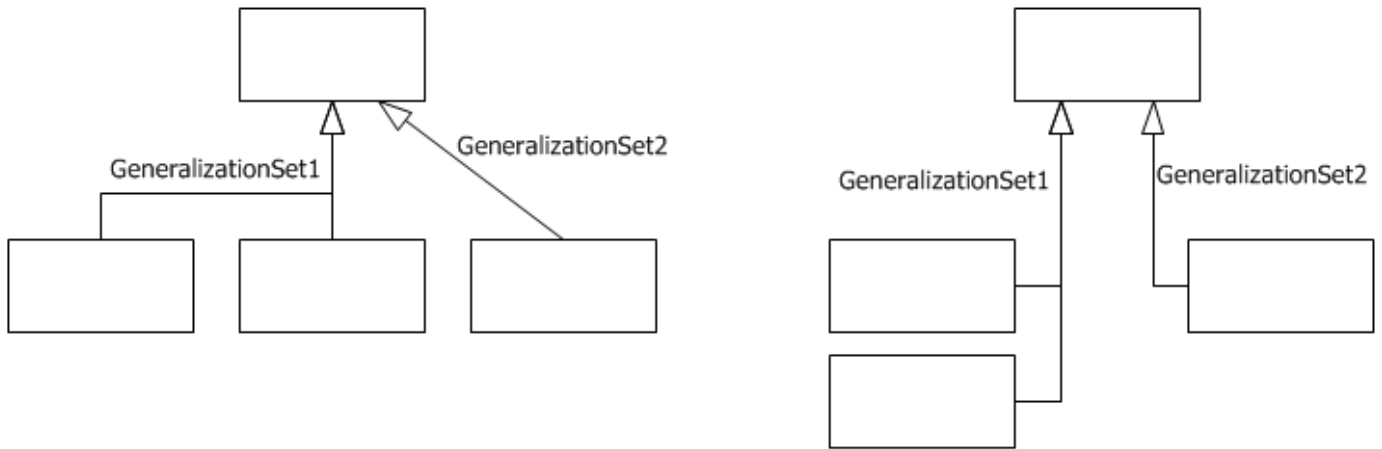
When Generalization relationship lines are named, that name designates the GeneralizationSet to which the Generalization belongs. All Generalization relationships with the same GeneralizationSet name are part of the same GeneralizationSet. This notation form is depicted in Figure 9.16.





**Figure 9.16 GeneralizationSets designated by name**

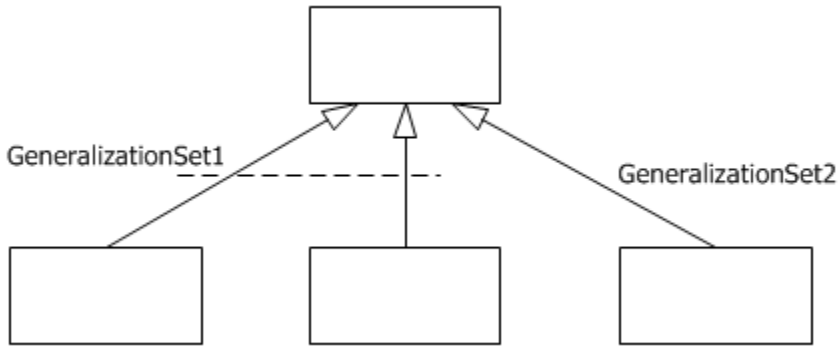
When two or more lines are drawn to the same arrowhead and labeled by a single GeneralizationSet name, i.e., “shared target” style as illustrated in Figure 9.17, the specific Classifiers are part of the same GeneralizationSet.



**Figure 9.17 GeneralizationSets designated by shared target**

With either of the notation forms above, if there are no labels on the Generalization arrows it cannot be determined from the diagram whether there are any GeneralizationSets in the model.

Lastly in Figure 9.18, a GeneralizationSet may be designated by drawing a dashed line across those lines with separate arrowheads that are meant to be part of the same set. Here, as in Figure 9.17, the GeneralizationSet is labeled with a single name, instead of each line labeled separately. This label may be elided.



**Figure 9.18** GeneralizationSet designated by dashed line spanning Generalization arrows

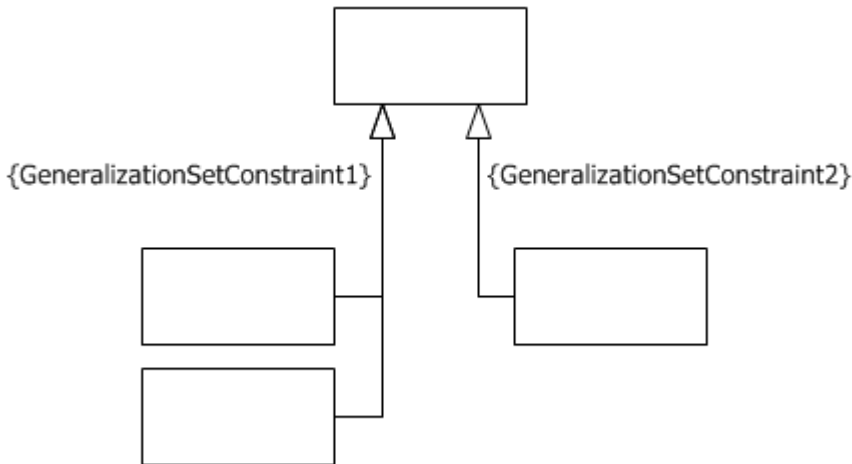
To indicate whether or not a generalization set is covering and disjoint, each set may be labeled with a constraint consisting of one of the keyword pairs indicated below.

**Table 9.1** GeneralizationSet constraints

{complete, disjoint}	Indicates the generalization set is covering and its specific Classifiers have no common instances.
{incomplete, disjoint}	Indicates the generalization set is not covering and its specific Classifiers have no common instances.
{complete, overlapping}	Indicates the generalization set is covering and its specific Classifiers do share common instances.
{incomplete, overlapping}	Indicates the generalization set is not covering and its specific Classifiers do share common instances.

The constraints may appear in either order: {complete, disjoint} is equivalent to {disjoint, complete}. The default values are {incomplete, overlapping}. If only one constraint is shown, the other takes its default value.

Graphically, the GeneralizationSet constraints are placed next to the sets, whether the common arrowhead notation is employed as illustrated in Figure 9.19 below, or the dashed line notation as shown in Figure 9.20.



**Figure 9.19** GeneralizationSet constraint notation with shared target style

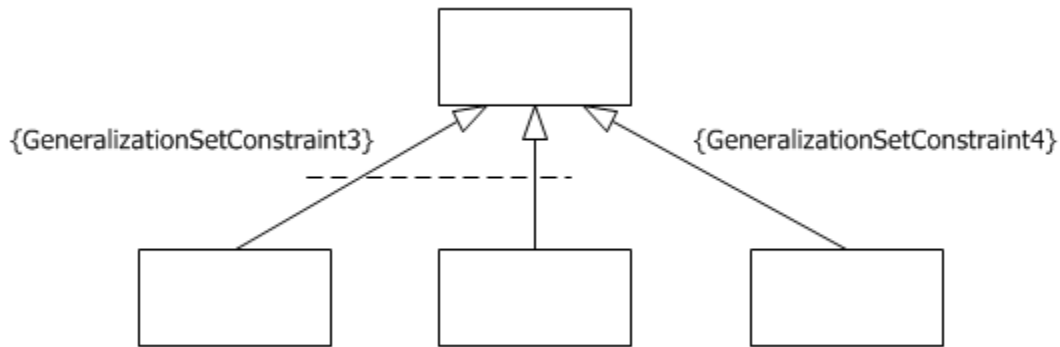


Figure 9.20 GeneralizationSet constraint notation with dashed line style

Power type specification is indicated by placing the name of the powertype Classifier—preceded by a colon—next to the corresponding GeneralizationSet. Figure 9.21 below indicates how this would appear for the shared arrowhead notation, and Figure 9.22 shows it for the dashed-line notation.

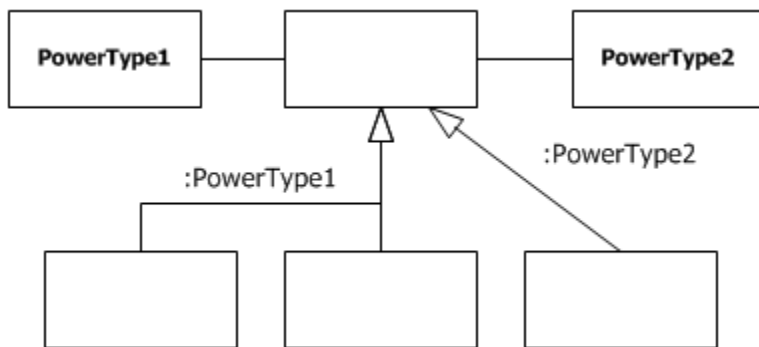


Figure 9.21 Power type notation with shared target style

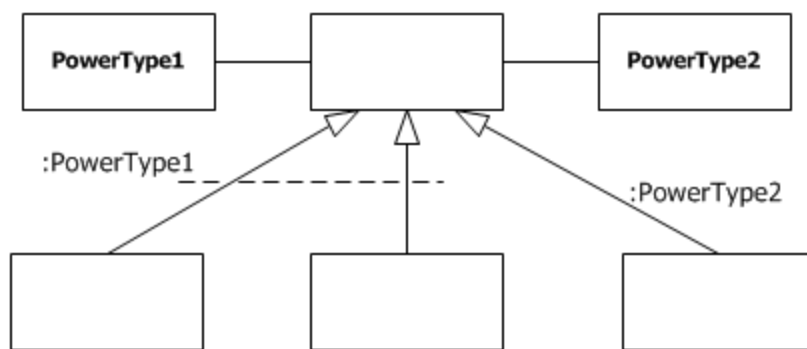
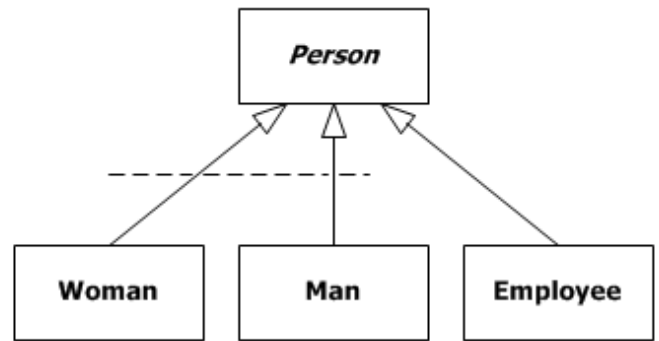
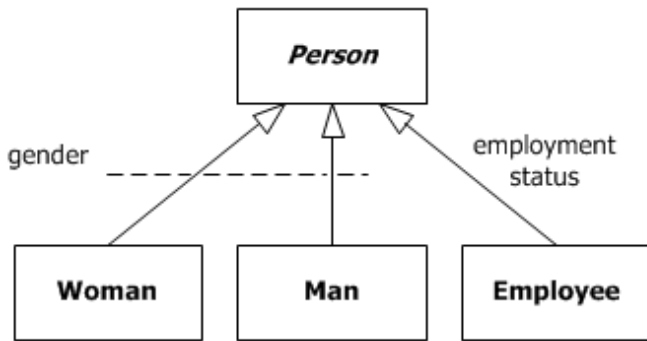
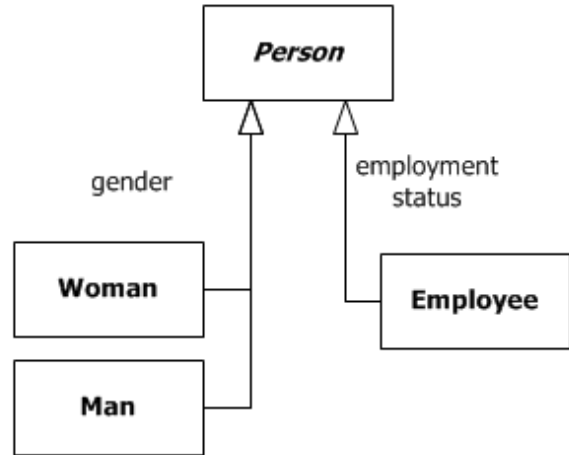
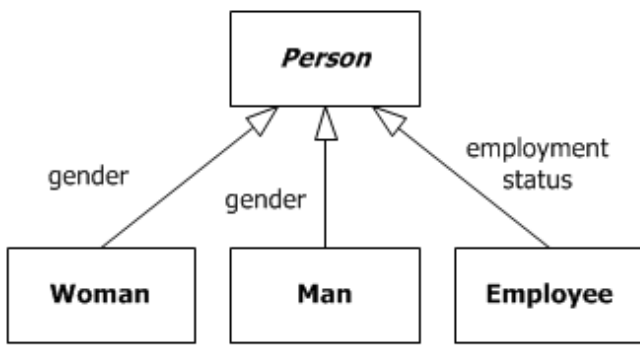


Figure 9.22 Power type notation with dashed line style

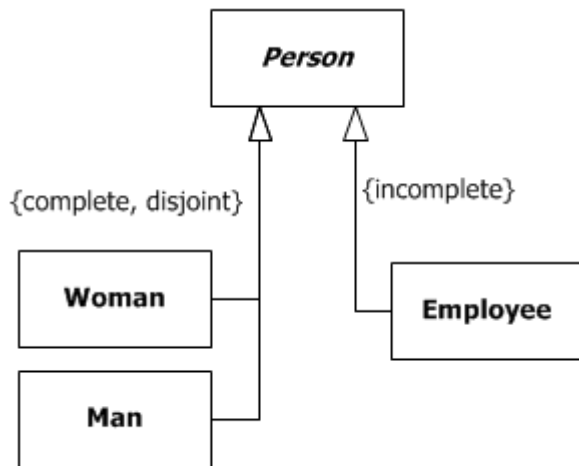
## 9.7.5 Examples

In Figure 9.23, Person is specialized as Woman and Man. Separately, Person is specialized as Employee. Here, the specializations to Woman and Man constitute one GeneralizationSet and that to Employee another. This example employs the various notation forms.



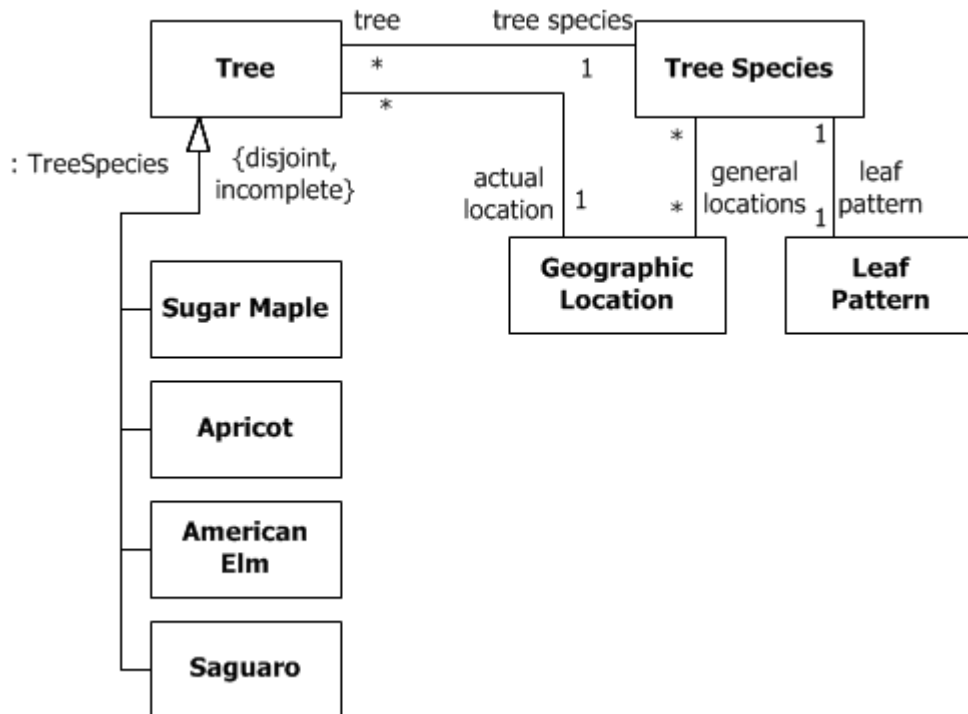
**Figure 9.23 GeneralizationSet notation options**

In Figure 9.24 below, Person is specialized as Woman and Man. Because this GeneralizationSet is partitioned (i.e., is constrained to be complete and disjoint), each instance of Person shall either be a Woman or a Man; that is, it shall be one or the other and not both. (Therefore, Person is an abstract class because a Person object may not exist without being either a Woman or a Man.) Person is also specialized as Employee, and this single specialization is expressed as {incomplete}, which means that a Person may either be an Employee or not. Taken together, the diagram indicates that a Person may be 1) either a Man or Woman, and 2) an Employee or not (a total of four options).



**Figure 9.24 GeneralizationSets and constraints**

One of the ways botanists organize trees is by species. Each tree we see may be classified as an American elm, sugar maple, apricot, saguaro—or some other species of tree. The class diagram below expresses that each Tree Species classifies zero or more instances of Tree, and each Tree is classified as exactly one Tree Species. For example, one of the instances of Tree could be the tree in your front yard, the tree in your neighbor’s backyard, or trees at your local nursery. Furthermore, this figure indicates the relationships that exist between these two sets of objects. For instance, the tree in your front yard might be classified as a sugar maple, your neighbor’s tree as an apricot, and so on. This class diagram indicates that each Tree Species is identified with a Leaf Pattern and has a general location in any number of Geographic Locations. For example, the saguaro cactus has leaves reduced to large spines and is generally found in southern Arizona and northern Sonora. Additionally, this figure indicates each Tree has an actual location at a particular Geographic Location. In this way, a particular tree could be classified as a saguaro and be located in Phoenix, Arizona.

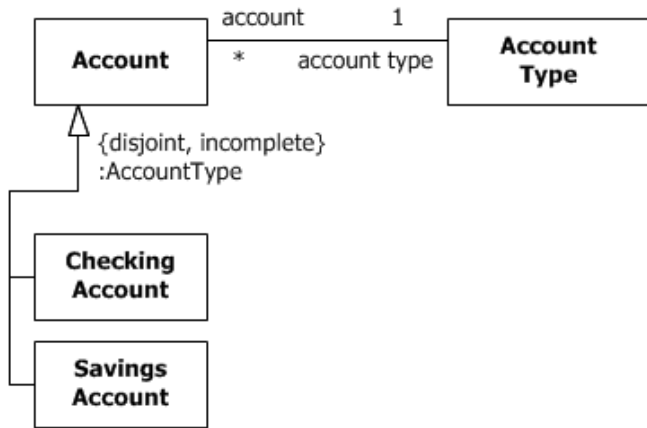


**Figure 9.25 Power type example**

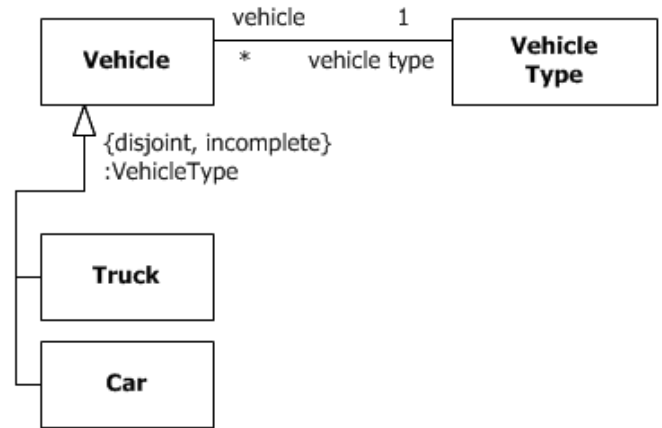
This diagram also illustrates that **Tree** is subtyped as **American Elm**, **Sugar Maple**, **Apricot**, or **Saguaro**—or something else. Each subtype, then, may have its own specialized Properties. For instance, each **Sugar Maple** could have a yearly maple sugar yield of some given quantity, each **Saguaro** could be inhabited by zero or more instances of a **Gila Woodpecker**, and so on.

The powertype designation on the **Tree** GeneralizationSet specifies that the instances of **TreeSpecies** are in one-to-one correspondence to the subclasses of **Tree**.

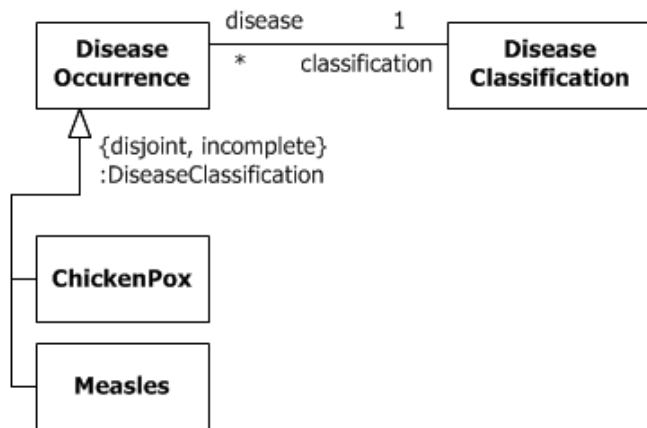
This concept applies to many situations within many lines of business. Figure 9.26 depicts other examples of power types. The name on the GeneralizationSet beginning with a colon indicates the power type.



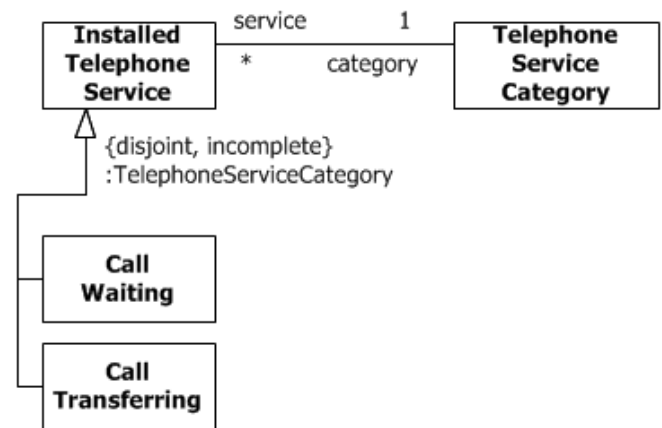
(a) Bank account/account type example



(b) Vehicle/vehicle type example



(c) Disease occurrence/classification example



(d) Telephone service/category example

**Figure 9.26 More power type examples**

In diagram (a), each instance of Checking Account could have its own attributes (including those inherited from Account), such as account number and balance. Additionally, the equivalent instance for Checking Account may have attributes, such as interest rate and maximum delay for withdrawal.

The example (b) depicts a vehicle-modeling example. Here, each Vehicle may be classified as either a Truck or a Car or something else. Furthermore, Truck and Car are equivalent to instances of Vehicle Type. In (c), Disease Occurrence classifies each occurrence of disease (e.g., my chicken pox and your measles). Disease Classification is the power type whose instances are equivalent to classes such as Chicken Pox and Measles.

Labeling collections of subtypes with powertypes becomes increasingly important when a type has more than one powertype. Figure 9.27 illustrates one such example, showing which subtype collection contains Policy Coverage Types and which Insurance Lines. For instance, a Policy may be classified as Life, Health, Property/Casualty, or some other Insurance Line. The same Policy may be classified with its Policy Coverage Type as Group or Individual.

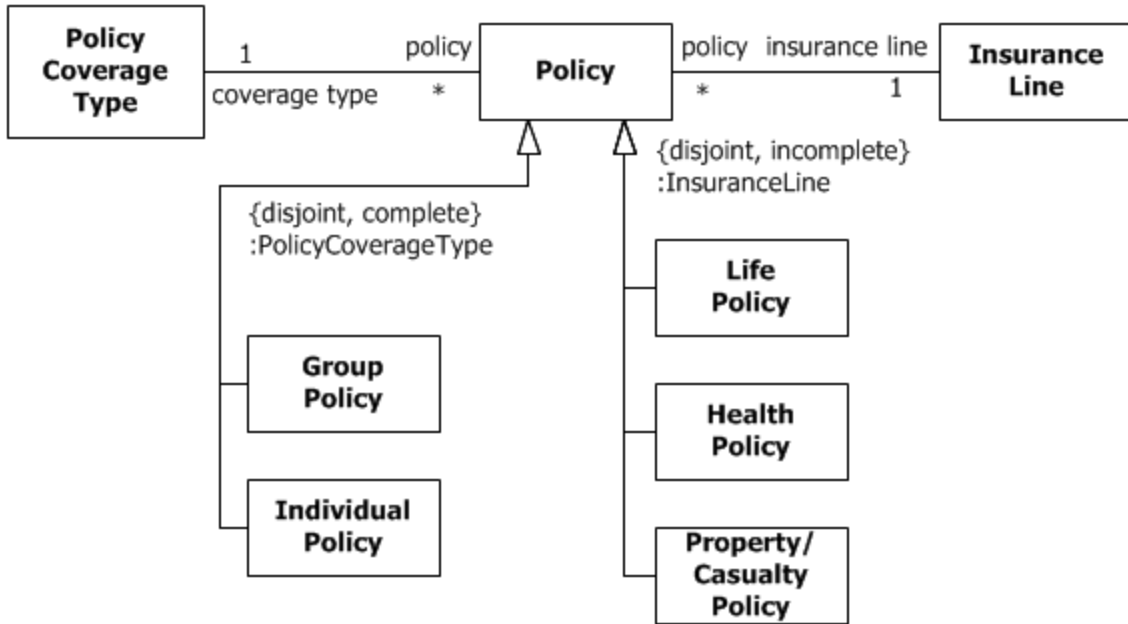


Figure 9.27 More than one powertype



## 9.8 Instances

### 9.8.1 Summary

InstanceSpecifications represent individual instances of Classifiers in a modeled system. They are often used to model example configurations of instances. They may be partial or complete representations of the instances that they correspond to.

### 9.8.2 Abstract Syntax

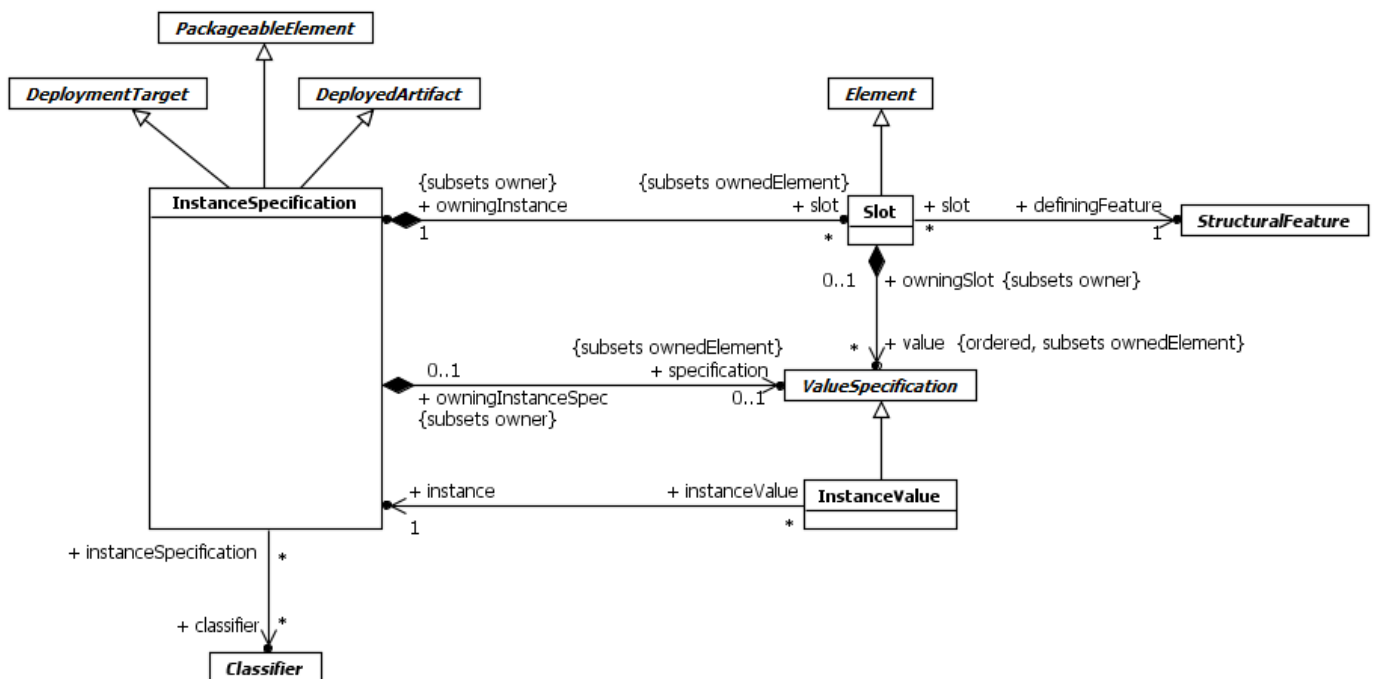


Figure 9.28 Instances

### 9.8.3 Semantics

An InstanceSpecification represents the existence of an instance in a modeled system and completely or partially describes the instance.

A Slot specifies that an instance modeled by an InstanceSpecification has a value or values for a specific StructuralFeature, which shall be a StructuralFeature of a Classifier of the InstanceSpecification owning the Slot. The values in a Slot shall conform to the defining StructuralFeature of the Slot (in type, multiplicity, etc.). The values in a Slot are specified using ValueSpecifications (see Clause 8).

The InstanceSpecification may represent:

- Classification of the instance by one or more Classifiers. If the only Classifier specified is abstract, then the InstanceSpecification only partially describes the instance.

- The kind of instance, based on its classifiers. For example, an InstanceSpecification whose classifier is a Class describes an instance of that Class, while an InstanceSpecification whose classifier is an Association describes a link of that Association. If no classifiers are given, then the InstanceSpecification does not constrain the kind of individual represented. If classifiers of different kinds are given, then the semantics are not defined.
- Specification of values of StructuralFeatures of the instance, where the values are contained in Slots. Not all StructuralFeatures of all Classifiers of the InstanceSpecification need be represented by Slots, in which case the InstanceSpecification is a partial description.
- An optional specification, by a ValueSpecification, of how to compute, derive, or construct the individual. If such a ValueSpecification is given, then the represented individual is equal to the value resulting from the evaluation of the ValueSpecification. If the InstanceSpecification has one or more classifiers, then the type of the ValueSpecification must conform to at least one of those classifiers.

An InstanceSpecification may specify the actual existence of an instance in a modeled system. Or, an InstanceSpecification may provide an illustration or example of a possible instance in a modeled system. The purpose of an InstanceSpecification is to show what is of interest about the instance. The instance conforms to each classifier of the InstanceSpecification, and has properties with values indicated by each slot of the InstanceSpecification. Having no slot in an InstanceSpecification for some properties does not mean that the represented instance does not have the property, but merely that the property is not of interest in the model.

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

It is important to keep in mind that InstanceSpecification is a model element and should not be confused with the instance that it is modeling. As an InstanceSpecification may only partially determine the properties of an individual, there may actually be multiple individuals in the modeled system that satisfy the requirements of the InstanceSpecification. On the other hand, an InstanceSpecification may model a situation which is not actually supposed to occur in the modeled system, in which case no individual meeting the requirements of the InstanceSpecification may ever actually occur in the system.

An InstanceValue is a kind of ValueSpecification whose value is specified using an InstanceSpecification. Each evaluation of the InstanceValue is considered to result in a distinct instance conforming to the InstanceSpecification. If the InstanceSpecification has a specification, then that ValueSpecification is evaluated to give the value of the InstanceValue. Otherwise, an InstanceValue is evaluated by creating a value that is an instance of each of the classifiers identified in the InstanceSpecification. Any slots in the InstanceSpecification then provide values for the corresponding StructuralFeatures of the instance by evaluating the ValueSpecifications associated with those slots. A StructuralFeature for which no slot is given either has the value obtained by evaluating its defaultValue, if it is a Property with a defaultValue, or no value, otherwise.

**NOTE.** An InstanceValue does not own the InstanceSpecification to which it refers; multiple InstanceValues may refer to the same InstanceSpecification.

## 9.8.4 Notation

An InstanceSpecification is depicted using similar notation to its classifiers, 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. The convention for showing multiple classifiers is to separate their names by commas.

An InstanceSpecification 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 InstanceSpecifications 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 may adorn the ends. Navigation arrows may be shown, but if shown, they shall agree with the navigation of the Association's ends.

**NOTE.** Names are optional for Classifiers and InstanceSpecifications. The absence of a name in a diagram does not necessarily reflect its absence in the underlying model.

The standard notation for an anonymous InstanceSpecification of an unnamed Classifier is an underlined colon (‘:’).

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

Slots are shown using similar notation to that of the corresponding StructuralFeatures. Where a StructuralFeature would be shown textually in a compartment, a Slot for that StructuralFeature may be shown textually as a StructuralFeature name followed by an equal sign (“=”) and a value specification. Other properties of the StructuralFeature, such as its type, may optionally be shown.

An InstanceValue may appear using textual or graphical notation. When textual, as may appear for the value of a Slot, the name of the InstanceSpecification is shown. This may be displayed as a qualified name. When graphical, an InstanceValue is represented using the notation for its InstanceSpecification.

A Slot value that is an InstanceValue may alternatively be shown using a graphical notation similar to that for a link. A solid path runs from the owning InstanceSpecification to the symbol representing the InstanceValue that is the Slot’s value, and the name of the attribute adorns the target end of the path. Navigability, if shown, shall be only in the direction of the target. This notation can give rise to visual ambiguity with the link notation when the only adornments are at the target end; in such cases the model should be inspected to determine the presence or absence of an actual Association instance.

Where an InstanceSpecification is classified by a StructuredClassifier (see [11.2.3](#)) it may contain nested rectangles representing the instances playing its roles. The namestring of such a nested InstanceSpecification obeys the following syntax:

```
{<name> [‘/’ <rolename>] | ‘/’ <rolename>} [‘:’ <classifiername> [‘,’ <classifiername>]*]
```

The name of the InstanceSpecification may be followed by the name of the role which the instance plays. The role name may only be present if the instance plays a role.

Where an InstanceSpecification contains both Slot values and nested rectangles depicting roles, it is divided into compartments analogous to the attributes and internal structure compartments of its corresponding StructuredClassifier.

Examples of InstanceSpecifications for StructuredClassifiers are shown in [11.4.5](#).

## 9.8.5 Examples

The example in Figure 9.29 below shows an InstanceSpecification called “streetName,” classified as String, and with a specification that is a LiteralString whose value is “S.Crown Street.”

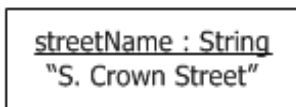


Figure 9.29 Specification of an Instance of String

The example in Figure 9.30 below shows an InstanceSpecification with Slots.

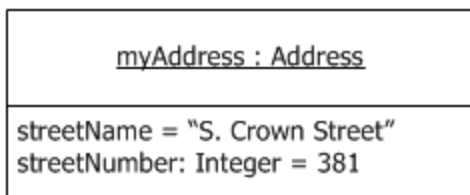
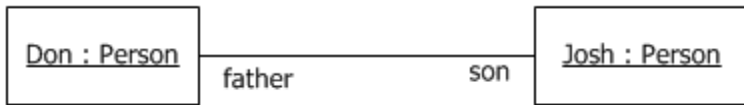


Figure 9.30 Slots with values

The example in Figure 9.31 below shows a link between two InstanceSpecifications.



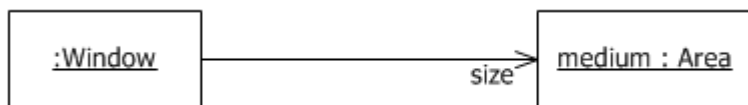
**Figure 9.31 InstanceSpecifications representing two objects connected by a link**

The example in Figure 9.32 below shows an InstanceValue as the value of a Slot represented using textual notation.



**Figure 9.32 InstanceValue represented textually**

The example in Figure 9.33 below shows the same model represented using graphical notation.



**Figure 9.33 InstanceValue represented graphically**

## 9.9 Classifier Descriptions

### AggregationKind [Enumeration]

#### Description

AggregationKind is an Enumeration for specifying the kind of aggregation of a Property.

#### Diagrams

- [Properties](#)

#### Literals

- none  
Indicates that the Property has no aggregation.
- shared  
Indicates that the Property has shared aggregation.
- composite  
Indicates that the Property is aggregated compositely, i.e., the composite object has responsibility for the existence and storage of the composed objects (parts).

### BehavioralFeature [Abstract Class]

#### Description

A BehavioralFeature is a feature of a Classifier that specifies an aspect of the behavior of its instances. A BehavioralFeature is implemented (realized) by a Behavior. A BehavioralFeature specifies that a Classifier will respond to a designated request by invoking its implementing method.

#### Diagrams

[Features](#), [Operations](#), [Signals](#), [Behaviors](#)

#### Generalizations

[Feature](#), [Namespace](#)

#### Specializations

[Operation](#), [Reception](#)

#### Attributes

- concurrency : [CallConcurrencyKind](#) [1..1] = sequential  
Specifies the semantics of concurrent calls to the same passive instance (i.e., an instance originating from a Class with isActive being false). Active instances control access to their own BehavioralFeatures.

- `isAbstract` : [Boolean](#) [1..1] = false  
If true, then the BehavioralFeature does not have an implementation, and one must be supplied by a more specific Classifier. If false, the BehavioralFeature must have an implementation in the Classifier or one must be inherited.

## Association Ends

- method : [Behavior](#) [0..\*] (opposite [Behavior::specification](#))  
A Behavior that implements the BehavioralFeature. There may be at most one Behavior for a particular pairing of a Classifier (as owner of the Behavior) and a BehavioralFeature (as specification of the Behavior).
- ♦ ownedParameter : [Parameter](#) [0..\*]{ordered, subsets [Namespace::ownedMember](#)} (opposite [A\\_ownedParameter\\_ownerFormalParam::ownerFormalParam](#))  
The ordered set of formal Parameters of this BehavioralFeature.
- ♦ ownedParameterSet : [ParameterSet](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [A\\_ownedParameterSet\\_behavioralFeature::behavioralFeature](#))  
The ParameterSets owned by this BehavioralFeature.
- raisedException : [Type](#) [0..\*] (opposite [A\\_raisedException\\_behavioralFeature::behavioralFeature](#))  
The Types representing exceptions that may be raised during an invocation of this BehavioralFeature.

## Operations

- `isDistinguishableFrom(n : NamedElement, ns : Namespace)` : [Boolean](#)  
The query `isDistinguishableFrom()` determines whether two BehavioralFeatures may coexist in the same Namespace. It specifies that they must have different signatures.

```
body: (n.ocIsKindOf(BehavioralFeature) and ns.getNamesOfMember(self)-
>intersection(ns.getNamesOfMember(n))->notEmpty()) implies
  Set{self}->including(n.ocIsType(BehavioralFeature))->isUnique(ownedParameter->collect(type))
```

- `inputParameters()` : [Parameter](#) [0..\*]  
The ownedParameters with direction in and inout.

```
body: ownedParameter->select(direction=ParameterDirectionKind::_'in' or
direction=ParameterDirectionKind::inout)
```

- `outputParameters()` : [Parameter](#) [0..\*]  
The ownedParameters with direction out, inout, or return.

```
body: ownedParameter->select(direction=ParameterDirectionKind::out or
direction=ParameterDirectionKind::inout or direction=ParameterDirectionKind::return)
```

## Constraints

- `abstract_no_method`  
When `isAbstract` is true there are no methods.

```
inv: isAbstract implies method->isEmpty()
```

## CallConcurrencyKind [Enumeration]

### Description

CallConcurrencyKind is an Enumeration used to specify the semantics of concurrent calls to a BehavioralFeature.

### Diagrams

- [Features](#)

### Literals

- sequential  
No concurrency management mechanism is associated with the BehavioralFeature and, therefore, concurrency conflicts may occur. Instances that invoke a BehavioralFeature need to coordinate so that only one invocation to a target on any BehavioralFeature occurs at once.
- guarded  
Multiple invocations of a BehavioralFeature may occur simultaneously to one instance, but only one is allowed to commence. The others are blocked until the performance of the currently executing BehavioralFeature is complete. It is the responsibility of the system designer to ensure that deadlocks do not occur due to simultaneous blocks.
- concurrent  
Multiple invocations of a BehavioralFeature may occur simultaneously to one instance and all of them may proceed concurrently.

## Classifier [Abstract Class]

### Description

A Classifier represents a classification of objects according to their Features. Classifiers are related by Generalizations.

### Diagrams

[Classifiers](#), [Classifier Templates](#), [Features](#), [Instances](#), [Generalization Sets](#), [Executable Nodes](#), [Use Cases](#), [Structured Classifiers](#), [Classes](#), [Associations](#), [Components](#), [Collaborations](#), [State Machine Redefinition](#), [DataTypes](#), [Signals](#), [Interfaces](#), [Information Flows](#), [Artifacts](#), [Actions](#), [Accept Event Actions](#), [Object Actions](#)

### Generalizations

[Namespace](#), [Type](#), [TemplateableElement](#), [RedefinableElement](#)

### Specializations

[Association](#), [StructuredClassifier](#), [BehavioredClassifier](#), [DataType](#), [Interface](#), [Signal](#), [InformationItem](#), [Artifact](#)

### Attributes

- isAbstract : [Boolean](#) [1..1] = false  
If true, the Classifier does not provide a complete declaration and cannot be instantiated. An abstract Classifier is

intended to be used by other Classifiers e.g., as the target of Associations or Generalizations.

- `isFinalSpecialization` : [Boolean](#) [1..1] = false  
If true, the Classifier cannot be specialized.

## Association Ends

- `/attribute` : [Property](#) [0..\*]{ordered, union, subsets [Classifier::feature](#), subsets [A\\_redefinitionContext\\_redefinableElement::redefinableElement](#)} (opposite [A\\_attribute\\_classifier::classifier](#))  
All of the Properties that are direct (i.e., not inherited or imported) attributes of the Classifier.
- `◆ collaborationUse` : [CollaborationUse](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [A\\_collaborationUse\\_classifier::classifier](#))  
The CollaborationUses owned by the Classifier.
- `/feature` : [Feature](#) [0..\*]{union, subsets [Namespace::member](#)} (opposite [Feature::featuringClassifier](#))  
Specifies each Feature directly defined in the classifier. Note that there may be members of the Classifier that are of the type Feature but are not included, e.g., inherited features.
- `/general` : [Classifier](#) [0..\*] (opposite [A\\_general\\_classifier::classifier](#))  
The generalizing Classifiers for this Classifier.
- `◆ generalization` : [Generalization](#) [0..\*]{subsets [Element::ownedElement](#), subsets [A\\_source\\_directedRelationship::directedRelationship](#)} (opposite [Generalization::specific](#))  
The Generalization relationships for this Classifier. These Generalizations navigate to more general Classifiers in the generalization hierarchy.
- `/inheritedMember` : [NamedElement](#) [0..\*]{subsets [Namespace::member](#)} (opposite [A\\_inheritedMember\\_inheritingClassifier::inheritingClassifier](#))  
All elements inherited by this Classifier from its general Classifiers.
- `◆ ownedTemplateSignature` : [RedefinableTemplateSignature](#) [0..1]{subsets [A\\_redefinitionContext\\_redefinableElement::redefinableElement](#), redefines [TemplateableElement::ownedTemplateSignature](#)} (opposite [RedefinableTemplateSignature::classifier](#))  
The optional RedefinableTemplateSignature specifying the formal template parameters.
- `◆ ownedUseCase` : [UseCase](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [A\\_ownedUseCase\\_classifier::classifier](#))  
The UseCases owned by this classifier.
- `powertypeExtent` : [GeneralizationSet](#) [0..\*] (opposite [GeneralizationSet::powertype](#))  
The GeneralizationSet of which this Classifier is a power type.
- `redefinedClassifier` : [Classifier](#) [0..\*]{subsets [RedefinableElement::redefinedElement](#)} (opposite [A\\_redefinedClassifier\\_classifier::classifier](#))  
The Classifiers redefined by this Classifier.
- `representation` : [CollaborationUse](#) [0..1]{subsets [Classifier::collaborationUse](#)} (opposite [A\\_representation\\_classifier::classifier](#))  
A CollaborationUse which indicates the Collaboration that represents this Classifier.



- ♦ substitution : [Substitution](#) [0..\*]{subsets [Element::ownedElement](#), subsets [NamedElement::clientDependency](#)} (opposite [Substitution::substitutingClassifier](#))  
The Substitutions owned by this Classifier.
- templateParameter : [ClassifierTemplateParameter](#) [0..1]{redefines [ParameterableElement::templateParameter](#)} (opposite [ClassifierTemplateParameter::parameteredElement](#))  
TheClassifierTemplateParameter that exposes this element as a formal parameter.
- useCase : [UseCase](#) [0..\*] (opposite [UseCase::subject](#))  
The set of UseCases for which this Classifier is the subject.

## Operations

- allFeatures() : [Feature](#) [0..\*]  
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.

```
body: member->select (oclIsKindOf (Feature) ) ->collect (oclAsType (Feature) ) ->asSet ()
```

- allParents() : [Classifier](#) [0..\*]  
The query allParents() gives all of the direct and indirect ancestors of a generalized Classifier.

```
body: parents () ->union (parents () ->collect (allParents () ) ->asSet () )
```

- conformsTo(other : [Classifier](#)) : [Boolean](#)  
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.

```
body: self = other or allParents () ->includes (other)
```

- general() : [Classifier](#) [0..\*]  
The general Classifiers are the ones referenced by the Generalization relationships.

```
body: parents ()
```

- hasVisibilityOf(n : [NamedElement](#)) : [Boolean](#)  
The query hasVisibilityOf() determines whether a NamedElement is visible in the classifier. By default all are visible. It is only called when the argument is something owned by a parent.

```
pre: allParents () ->including (self) ->collect (member) ->includes (n)
body: n.visibility <> VisibilityKind::private
```

- inherit(inhs : [NamedElement](#) [0..\*]) : [NamedElement](#) [0..\*]  
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.

```
body: inhs
```

- `inheritableMembers(c : Classifier) : NamedElement [0..*]`  
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.

```
pre: c.allParents()->includes(self)
body: member->select(m | c.hasVisibilityOf(m))
```

- `inheritedMember() : NamedElement [0..*]`  
The `inheritedMember` association is derived by inheriting the inheritable members of the parents.

```
body: inherit(parents()->collect(inheritableMembers(self))->asSet())
```

- `isTemplate() : Boolean`  
The query `isTemplate()` returns whether this Classifier is actually a template.

```
body: ownedTemplateSignature <> null or general->exists(g | g.isTemplate())
```

- `maySpecializeType(c : Classifier) : Boolean`  
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.

```
body: self.oclIsKindOf(c.oclType())
```

- `parents() : Classifier [0..*]`  
The query `parents()` gives all of the immediate ancestors of a generalized Classifier.

```
body: generalization.general->asSet()
```

- `directlyRealizedInterfaces() : Interface [0..*]`  
The Interfaces directly realized by this Classifier

```
body: (clientDependency->
  select(oclIsKindOf(Realization) and supplier->forall(oclIsKindOf(Interface))))->
  collect(supplier.oclAsType(Interface))->asSet()
```

- `directlyUsedInterfaces() : Interface [0..*]`  
The Interfaces directly used by this Classifier

```
body: (supplierDependency->
  select(oclIsKindOf(Usage) and client->forall(oclIsKindOf(Interface))))->
  collect(client.oclAsType(Interface))->asSet()
```

- `allRealizedInterfaces() : Interface [0..*]`  
The Interfaces realized by this Classifier and all of its generalizations

```
body: directlyRealizedInterfaces()->union(self.allParents()->collect(directlyRealizedInterfaces()))->asSet()
```

- `allUsedInterfaces() : Interface [0..*]`  
The Interfaces used by this Classifier and all of its generalizations

```
body: directlyUsedInterfaces()->union(self.allParents()->collect(directlyUsedInterfaces()))->asSet()
```

- `isSubstitutableFor(contract : Classifier) : Boolean`

```
body: substitution.contract->includes(contract)
```

- `allAttributes() : Property [0..*]`

The query `allAttributes` gives an ordered set of all owned and inherited attributes of the Classifier. All owned attributes appear before any inherited attributes, and the attributes inherited from any more specific parent Classifier appear before those of any more general parent Classifier. However, if the Classifier has multiple immediate parents, then the relative ordering of the sets of attributes from those parents is not defined.

```
body: attribute->asSequence()->union(parents()->asSequence().allAttributes()->select(p | member->includes(p))->asOrderedSet()
```

## Constraints

- `specialize_type`  
A Classifier may only specialize Classifiers of a valid type.

```
inv: parents()->forall(c | self.maySpecializeType(c))
```

- `maps_to_generalization_set`  
The Classifier that maps to a GeneralizationSet may neither be a specific nor a general Classifier in any of the Generalization relationships defined for that GeneralizationSet. In other words, a power type may not be an instance of itself nor may its instances also be its subclasses.

```
inv: powertypeExtent->forall( gs |  
  gs.generalization->forall( gen |  
    not (gen.general = self) and not gen.general.allParents()->includes(self) and not (gen.specific =  
self) and not self.allParents()->includes(gen.specific)  
  ))
```

- `non_final_parents`  
The parents of a Classifier must be non-final.

```
inv: parents()->forall(not isFinalSpecialization)
```

- `no_cycles_in_generalization`  
Generalization hierarchies must be directed and acyclical. A Classifier can not be both a transitively general and transitively specific Classifier of the same Classifier.

```
inv: not allParents()->includes(self)
```

## ClassifierTemplateParameter [Class]

### Description

A ClassifierTemplateParameter exposes a Classifier as a formal template parameter.

## Diagrams

[Classifier Templates](#)

## Generalizations

[TemplateParameter](#)

## Attributes

- allowSubstitutable : [Boolean](#) [1..1] = true  
Constrains the required relationship between an actual parameter and the parameteredElement for this formal parameter.

## Association Ends

- constrainingClassifier : [Classifier](#) [0..\*] (opposite [A constrainingClassifier classifierTemplateParameter::classifierTemplateParameter](#))  
The classifiers that constrain the argument that can be used for the parameter. If the allowSubstitutable attribute is true, then any Classifier that is compatible with this constraining Classifier can be substituted; otherwise, it must be either this Classifier or one of its specializations. If this property is empty, there are no constraints on the Classifier that can be used as an argument.
- parameteredElement : [Classifier](#) [1..1]{redefines [TemplateParameter::parameteredElement](#)} (opposite [Classifier::templateParameter](#))  
The Classifier exposed by this ClassifierTemplateParameter.

## Constraints

- has\_constraining\_classifier  
If allowSubstitutable is true, then there must be a constrainingClassifier.

```
inv: allowSubstitutable implies constrainingClassifier->notEmpty()
```

- parametered\_element\_no\_features  
The parameteredElement has no direct features, and if constrainedElement is empty it has no generalizations.

```
inv: parameteredElement.feature->isEmpty() and (constrainingClassifier->isEmpty() implies parameteredElement.allParents()->isEmpty())
```

- matching\_abstract  
If the parameteredElement is not abstract, then the Classifier used as an argument shall not be abstract.

```
inv: (not parameteredElement.isAbstract) implies templateParameterSubstitution.actual->forall(a | not a.oclAsType(Classifier).isAbstract)
```

- actual\_is\_classifier  
The argument to a ClassifierTemplateParameter is a Classifier.

```
inv: templateParameterSubstitution.actual->forall(a | a.oclIsKindOf(Classifier))
```

- **constraining\_classifiers\_constrain\_args**

If there are any constrainingClassifiers, then every argument must be the same as or a specialization of them, or if allowSubstitutable is true, then it can also be substitutable.

```
inv: templateParameterSubstitution.actual->forall( a |
  let arg : Classifier = a.oclAsType(Classifier) in
    constrainingClassifier->forall(
      cc |
        arg = cc or arg.conformsTo(cc) or (allowSubstitutable and arg.isSubstitutableFor(cc))
    )
)
```

- **constraining\_classifiers\_constrain\_parametered\_element**

If there are any constrainingClassifiers, then the parameteredElement must be the same as or a specialization of them, or if allowSubstitutable is true, then it can also be substitutable.

```
inv: constrainingClassifier->forall(
  cc | parameteredElement = cc or parameteredElement.conformsTo(cc) or (allowSubstitutable and
parameteredElement.isSubstitutableFor(cc))
)
```

## Feature [Abstract Class]

### Description

A Feature declares a behavioral or structural characteristic of Classifiers.

### Diagrams

[Classifiers](#), [Features](#), [Structured Classifiers](#)

### Generalizations

[RedefinableElement](#)

### Specializations

[BehavioralFeature](#), [StructuralFeature](#), [Connector](#)

### Attributes

- isStatic : [Boolean](#) [1..1] = false  
Specifies whether this Feature characterizes individual instances classified by the Classifier (false) or the Classifier itself (true).

### Association Ends

- /featuringClassifier : [Classifier](#) [0..\*]{union, subsets [A\\_member\\_memberNamespace::memberNamespace](#)} (opposite [Classifier::feature](#))  
The Classifiers that have this Feature as a feature.

## Generalization [Class]

### Description

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. The specific Classifier inherits the features of the more general Classifier. A Generalization is owned by the specific Classifier.

### Diagrams

[Classifiers](#), [Generalization Sets](#)

### Generalizations

[DirectedRelationship](#)

### Attributes

- isSubstitutable : [Boolean](#) [0..1] = true  
Indicates whether the specific Classifier can be used wherever the general Classifier can be used. If true, the execution traces of the specific Classifier shall be a superset of the execution traces of the general Classifier. If false, there is no such constraint on execution traces. If unset, the modeler has not stated whether there is such a constraint or not.

### Association Ends

- general : [Classifier](#) [1..1]{subsets [DirectedRelationship::target](#)} (opposite [A\\_general\\_generalization::generalization](#))  
The general classifier in the Generalization relationship.
- generalizationSet : [GeneralizationSet](#) [0..\*] (opposite [GeneralizationSet::generalization](#))  
Represents a set of instances of Generalization. A Generalization may appear in many GeneralizationSets.
- specific : [Classifier](#) [1..1]{subsets [DirectedRelationship::source](#), subsets [Element::owner](#)} (opposite [Classifier::generalization](#))  
The specializing Classifier in the Generalization relationship.

## GeneralizationSet [Class]

### Description

A GeneralizationSet is a PackageableElement whose instances represent sets of Generalization relationships.

### Diagrams

[Classifiers](#), [Generalization Sets](#)

### Generalizations

[PackageableElement](#)

## Attributes

- `isCovering` : [Boolean](#) [1..1] = false  
Indicates (via the associated Generalizations) whether or not the set of specific Classifiers are covering for a particular general classifier. When `isCovering` is true, every instance of a particular general Classifier is also an instance of at least one of its specific Classifiers for the GeneralizationSet. When `isCovering` is false, there are one or more instances of the particular general Classifier that are not instances of at least one of its specific Classifiers defined for the GeneralizationSet.
- `isDisjoint` : [Boolean](#) [1..1] = false  
Indicates whether or not the set of specific Classifiers in a Generalization relationship have instance in common. If `isDisjoint` is true, the specific Classifiers for a particular GeneralizationSet have no members in common; that is, their intersection is empty. If `isDisjoint` is false, the specific Classifiers in a particular GeneralizationSet have one or more members in common; that is, their intersection is not empty.

## Association Ends

- `generalization` : [Generalization](#) [0..\*] (opposite [Generalization::generalizationSet](#))  
Designates the instances of Generalization that are members of this GeneralizationSet.
- `powertype` : [Classifier](#) [0..1] (opposite [Classifier::powertypeExtent](#))  
Designates the Classifier that is defined as the power type for the associated GeneralizationSet, if there is one.

## Constraints

- `generalization_same_classifier`  
Every Generalization associated with a particular GeneralizationSet must have the same general Classifier.  

```
inv: generalization->collect(general)->asSet()->size() <= 1
```
- `maps_to_generalization_set`  
The Classifier that maps to a GeneralizationSet may neither be a specific nor a general Classifier in any of the Generalization relationships defined for that GeneralizationSet. In other words, a power type may not be an instance of itself nor may its instances be its subclasses.  

```
inv: powertype <> null implies generalization->forall( gen |
    not (gen.general = powertype) and not gen.general.allParents()->includes(powertype) and not
    (gen.specific = powertype) and not powertype.allParents()->includes(gen.specific)
    )
```
- `complete_and_disjoint`  
A complete and disjoint GeneralizationSet implies that the common general Classifier is abstract  

```
inv: isDisjoint and isCovering implies generalization->forall(general.isAbstract)
```

## InstanceSpecification [Class]

### Description

An InstanceSpecification is a model element that represents an instance in a modeled system. An InstanceSpecification can act as a DeploymentTarget in a Deployment relationship, in the case that it represents an instance of a Node. It can also act as a DeployedArtifact, if it represents an instance of an Artifact.

### Diagrams

[Instances](#), [DataTypes](#), [Deployments](#)

### Generalizations

[DeploymentTarget](#), [PackageableElement](#), [DeployedArtifact](#)

### Specializations

[EnumerationLiteral](#)

### Association Ends

- classifier : [Classifier](#) [0..\*] (opposite [A\\_classifier\\_instanceSpecification::instanceSpecification](#))  
The Classifier or Classifiers of the represented instance. If multiple Classifiers are specified, the instance is classified by all of them.
- ♦ slot : [Slot](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [Slot::owningInstance](#))  
A Slot giving the value or values of a StructuralFeature of the instance. An InstanceSpecification can have one Slot per StructuralFeature of its Classifiers, including inherited features. It is not necessary to model a Slot for every StructuralFeature, in which case the InstanceSpecification is a partial description.
- ♦ specification : [ValueSpecification](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_specification\\_owningInstanceSpec::owningInstanceSpec](#))  
A specification of how to compute, derive, or construct the instance.

### Constraints

- deployment\_artifact  
An InstanceSpecification can act as a DeployedArtifact if it represents an instance of an Artifact.  

```
inv: deploymentForArtifact->notEmpty() implies classifier->exists(oclIsKindOf(Artifact))
```
- structural\_feature  
One StructuralFeature (including the same feature inherited from multiple classifiers) is the defining feature of at most one slot in an InstanceSpecification.  

```
inv: classifier->forAll(c | (c.allFeatures()->forAll(f | slot->select(s | s.definingFeature = f)->size() <= 1)))
```
- defining\_feature  
The defining feature of each slot is a StructuralFeature (directly or inherited) of a classifier of the InstanceSpecification.



```
inv: slot->forall(s | classifier->exists (c | c.allFeatures()->includes (s.definingFeature)))
```

- `deployment_target`  
An InstanceSpecification can act as a DeploymentTarget if it represents an instance of a Node and functions as a part in the internal structure of an encompassing Node.

```
inv: deployment->notEmpty() implies classifier->exists(node | node.oclIsKindOf(Node) and  
Node.allInstances()->exists(n | n.part->exists(p | p.type = node)))
```

## InstanceValue [Class]

### Description

An InstanceValue is a ValueSpecification that identifies an instance.

### Diagrams

[Instances](#)

### Generalizations

[ValueSpecification](#)

### Association Ends

- `instance` : [InstanceSpecification](#) [1..1] (opposite [A\\_instance\\_instanceValue::instanceValue](#))  
The InstanceSpecification that represents the specified value.

## Operation [Class]

### Description

An Operation is a BehavioralFeature of a Classifier that specifies the name, type, parameters, and constraints for invoking an associated Behavior. An Operation may invoke both the execution of method behaviors as well as other behavioral responses. Operation specializes TemplateableElement in order to support specification of template operations and bound operations. Operation specializes ParameterableElement to specify that an operation can be exposed as a formal template parameter, and provided as an actual parameter in a binding of a template.

### Diagrams

[Operations](#), [Classes](#), [Protocol State Machines](#), [DataTypes](#), [Interfaces](#), [Artifacts](#), [Events](#), [Invocation Actions](#)

### Generalizations

[TemplateableElement](#), [ParameterableElement](#), [BehavioralFeature](#)

### Attributes

- `/isOrdered` : [Boolean](#) [1..1] = false  
Specifies whether the return parameter is ordered or not, if present. This information is derived from the return result for this Operation.

- isQuery : [Boolean](#) [1..1] = false  
Specifies whether an execution of the BehavioralFeature leaves the state of the system unchanged (isQuery=true) or whether side effects may occur (isQuery=false).
- /isUnique : [Boolean](#) [1..1] = true  
Specifies whether the return parameter is unique or not, if present. This information is derived from the return result for this Operation.
- /lower : [Integer](#) [0..1] = 1  
Specifies the lower multiplicity of the return parameter, if present. This information is derived from the return result for this Operation.
- /upper : [UnlimitedNatural](#) [0..1] = 1  
The upper multiplicity of the return parameter, if present. This information is derived from the return result for this Operation.

## Association Ends

- ♦ bodyCondition : [Constraint](#) [0..1]{subsets [Namespace::ownedRule](#)} (opposite [A\\_bodyCondition\\_bodyContext::bodyContext](#))  
An optional Constraint on the result values of an invocation of this Operation.
- class : [Class](#) [0..1]{subsets [Feature::featuringClassifier](#), subsets [NamedElement::namespace](#), subsets [RedefinableElement::redefinitionContext](#)} (opposite [Class::ownedOperation](#))  
The Class that owns this operation, if any.
- datatype : [DataType](#) [0..1]{subsets [Feature::featuringClassifier](#), subsets [NamedElement::namespace](#), subsets [RedefinableElement::redefinitionContext](#)} (opposite [DataType::ownedOperation](#))  
The DataType that owns this Operation, if any.
- interface : [Interface](#) [0..1]{subsets [Feature::featuringClassifier](#), subsets [NamedElement::namespace](#), subsets [RedefinableElement::redefinitionContext](#)} (opposite [Interface::ownedOperation](#))  
The Interface that owns this Operation, if any.
- ♦ ownedParameter : [Parameter](#) [0..\*]{ordered, redefines [BehavioralFeature::ownedParameter](#)} (opposite [Parameter::operation](#))  
The parameters owned by this Operation.
- ♦ postcondition : [Constraint](#) [0..\*]{subsets [Namespace::ownedRule](#)} (opposite [A\\_postcondition\\_postContext::postContext](#))  
An optional set of Constraints specifying the state of the system when the Operation is completed.
- ♦ precondition : [Constraint](#) [0..\*]{subsets [Namespace::ownedRule](#)} (opposite [A\\_precondition\\_preContext::preContext](#))  
An optional set of Constraints on the state of the system when the Operation is invoked.
- raisedException : [Type](#) [0..\*]{redefines [BehavioralFeature::raisedException](#)} (opposite [A\\_raisedException\\_operation::operation](#))  
The Types representing exceptions that may be raised during an invocation of this operation.

- redefinedOperation : [Operation](#) [0..\*]{subsets [RedefinableElement::redefinedElement](#)} (opposite [A\\_redefinedOperation\\_operation::operation](#))  
The Operations that are redefined by this Operation.
- templateParameter : [OperationTemplateParameter](#) [0..1]{redefines [ParameterableElement::templateParameter](#)} (opposite [OperationTemplateParameter::parameteredElement](#))  
The OperationTemplateParameter that exposes this element as a formal parameter.
- /type : [Type](#) [0..1]{ } (opposite [A\\_type\\_operation::operation](#))  
The return type of the operation, if present. This information is derived from the return result for this Operation.

## Operations

- isConsistentWith(redefinee : [RedefinableElement](#)) : [Boolean](#)  
A redefining operation is consistent with a redefined operation if it has the same number of owned parameters, and the type of each owned parameter conforms to the type of the corresponding redefined parameter. The query isConsistentWith() specifies, for any two Operations in a context in which redefinition is possible, whether redefinition would be consistent in the sense of maintaining type covariance. Other senses of consistency may be required, for example to determine consistency in the sense of contravariance. Users may define alternative queries under names different from isConsistentWith(), as for example, users may define a query named isContravariantWith().

```
pre: redefinee.isRedefinitionContextValid(self)
body: redefinee.oclIsKindOf(Operation) and
let op : Operation = redefinee.oclAsType(Operation) in
  self.ownedParameter->size() = op.ownedParameter->size() and
  Sequence{1..self.ownedParameter->size()->
    forAll(i |op.ownedParameter->at(1).type.conformsTo(self.ownedParameter->at(i).type)}
```

- isOrdered() : [Boolean](#)  
If this operation has a return parameter, isOrdered equals the value of isOrdered for that parameter. Otherwise isOrdered is false.

```
body: if returnResult()->notEmpty() then returnResult()->exists(isOrdered) else false endif
```

- isUnique() : [Boolean](#)  
If this operation has a return parameter, isUnique equals the value of isUnique for that parameter. Otherwise isUnique is true.

```
body: if returnResult()->notEmpty() then returnResult()->exists(isUnique) else true endif
```

- lower() : [Integer](#)  
If this operation has a return parameter, lower equals the value of lower for that parameter. Otherwise lower is not defined.

```
body: if returnResult()->notEmpty() then returnResult()->any(true).lower else null endif
```

- returnResult() : [Parameter](#) [0..\*]  
The query returnResult() returns the set containing the return parameter of the Operation if one exists, otherwise, it returns an empty set

```
body: ownedParameter->select (direction = ParameterDirectionKind::return)
```

- `type() : Type`  
If this operation has a return parameter, type equals the value of type for that parameter. Otherwise type is not defined.

```
body: if returnResult()->notEmpty() then returnResult()->any(true).type else null endif
```

- `upper() : UnlimitedNatural`  
If this operation has a return parameter, upper equals the value of upper for that parameter. Otherwise upper is not defined.

```
body: if returnResult()->notEmpty() then returnResult()->any(true).upper else null endif
```

## Constraints

- `at_most_one_return`  
An Operation can have at most one return parameter; i.e., an owned parameter with the direction set to 'return.'

```
inv: self.ownedParameter->select(direction = ParameterDirectionKind::return)->size() <= 1
```

- `only_body_for_query`  
A bodyCondition can only be specified for a query Operation.

```
inv: bodyCondition <> null implies isQuery
```

## OperationTemplateParameter [Class]

### Description

An OperationTemplateParameter exposes an Operation as a formal parameter for a template.

### Diagrams

[Operations](#)

### Generalizations

[TemplateParameter](#)

### Association Ends

- `parameteredElement : Operation [1..1]{redefines TemplateParameter::parameteredElement}` (opposite [Operation::templateParameter](#))  
The Operation exposed by this OperationTemplateParameter.

### Constraints

- `match_default_signature`

```
inv: default->notEmpty() implies (default.oclIsKindOf(Operation) and (let defaultOp : Operation =
default.oclAsType(Operation) in
  defaultOp.ownedParameter->size() = parameteredElement.ownedParameter->size() and
  Sequence{1.. defaultOp.ownedParameter->size()}->forall( ix |
```

```

let p1: Parameter = defaultOp.ownedParameter->at(ix), p2 : Parameter =
parameteredElement.ownedParameter->at(ix) in
  p1.type = p2.type and p1.upper = p2.upper and p1.lower = p2.lower and p1.direction =
p2.direction and p1.isOrdered = p2.isOrdered and p1.isUnique = p2.isUnique))

```

## Parameter [Class]

### Description

A Parameter is a specification of an argument used to pass information into or out of an invocation of a BehavioralFeature. Parameters can be treated as ConnectableElements within Collaborations.

### Diagrams

[Features](#), [Operations](#), [Object Nodes](#), [Expressions](#), [Behaviors](#)

### Generalizations

[MultiplicityElement](#), [ConnectableElement](#)

### Attributes

- /default : [String](#) [0..1]  
A String that represents a value to be used when no argument is supplied for the Parameter.
- direction : [ParameterDirectionKind](#) [1..1] = in  
Indicates whether a parameter is being sent into or out of a behavioral element.
- effect : [ParameterEffectKind](#) [0..1]  
Specifies the effect that the owner of the parameter has on values passed in or out of the parameter.
- isException : [Boolean](#) [1..1] = false  
Tells whether an output parameter may emit a value to the exclusion of the other outputs.
- isStream : [Boolean](#) [1..1] = false  
Tells whether an input parameter may accept values while its behavior is executing, or whether an output parameter may post values while the behavior is executing.

### Association Ends

- ♦ defaultValue : [ValueSpecification](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_defaultValue\\_owningParameter::owningParameter](#))  
Specifies a ValueSpecification that represents a value to be used when no argument is supplied for the Parameter.
- operation : [Operation](#) [0..1]{subsets [A\\_ownedParameter\\_ownerFormalParam::ownerFormalParam](#)} (opposite [Operation::ownedParameter](#))  
The Operation owning this parameter.
- parameterSet : [ParameterSet](#) [0..\*] (opposite [ParameterSet::parameter](#))  
The ParameterSets containing the parameter. See ParameterSet.

## Operations

- `default() : String [0..1]`  
Derivation for `Parameter::/default`

```
body: if self.type = String then defaultValue.stringValue() else null endif
```

## Constraints

- `in_and_out`  
Only in and inout Parameters may have a delete effect. Only out, inout, and return Parameters may have a create effect.

```
inv: (effect = ParameterEffectKind::delete implies (direction = ParameterDirectionKind::_'in' or
direction = ParameterDirectionKind::inout))
and
(effect = ParameterEffectKind::create implies (direction = ParameterDirectionKind::out or direction =
ParameterDirectionKind::inout or direction = ParameterDirectionKind::return))
```

- `not_exception`  
An input Parameter cannot be an exception.

```
inv: isException implies (direction <> ParameterDirectionKind::_'in' and direction <>
ParameterDirectionKind::inout)
```

- `connector_end`  
A Parameter may only be associated with a Connector end within the context of a Collaboration.

```
inv: end->notEmpty() implies collaboration->notEmpty()
```

- `reentrant_behaviors`  
Reentrant behaviors cannot have stream Parameters.

```
inv: (isStream and behavior <> null) implies not behavior.isReentrant
```

- `stream_and_exception`  
A Parameter cannot be a stream and exception at the same time.

```
inv: not (isException and isStream)
```

## ParameterDirectionKind [Enumeration]

### Description

ParameterDirectionKind is an Enumeration that defines literals used to specify direction of parameters.

### Diagrams

- [Features](#)

## Literals

- in  
Indicates that parameter values are passed into the behavioral element by the caller.
- inout  
Indicates that parameter values are passed into a behavioral element by the caller and then back out to the caller from the behavioral element.
- out  
Indicates that parameter values are passed from a behavioral element out to the caller.
- return  
Indicates that parameter values are passed as return values from a behavioral element back to the caller.

## ParameterEffectKind [Enumeration]

### Description

ParameterEffectKind is an Enumeration that indicates the effect of a Behavior on values passed in or out of its parameters.

### Diagrams

- [Features](#)

## Literals

- create  
Indicates that the behavior creates values.
- read  
Indicates that the behavior reads values.
- update  
Indicates that the behavior updates values.
- delete  
Indicates that the behavior deletes values.

## ParameterSet [Class]

### Description

A ParameterSet designates alternative sets of inputs or outputs that a Behavior may use.

### Diagrams

[Features](#), [Behaviors](#)

## Generalizations

### [NamedElement](#)

## Association Ends

- ♦ condition : [Constraint](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [A\\_condition\\_parameterSet::parameterSet](#))  
A constraint that should be satisfied for the owner of the Parameters in an input ParameterSet to start execution using the values provided for those Parameters, or the owner of the Parameters in an output ParameterSet to end execution providing the values for those Parameters, if all preconditions and conditions on input ParameterSets were satisfied.
- parameter : [Parameter](#) [1..\*] (opposite [Parameter::parameterSet](#))  
Parameters in the ParameterSet.

## Constraints

- same\_parameterized\_entity  
The Parameters in a ParameterSet must all be inputs or all be outputs of the same parameterized entity, and the ParameterSet is owned by that entity.

```
inv: parameter->forall(p1, p2 | self.owner = p1.owner and self.owner = p2.owner and p1.direction = p2.direction)
```

- input  
If a parameterized entity has input Parameters that are in a ParameterSet, then any inputs that are not in a ParameterSet must be streaming. Same for output Parameters.

```
inv: ((parameter->exists(direction = ParameterDirectionKind::_in')) implies behavioralFeature.ownedParameter->select(p | p.direction = ParameterDirectionKind::_in' and p.parameterSet->isEmpty()->forall(isStream)) and ((parameter->exists(direction = ParameterDirectionKind::out)) implies behavioralFeature.ownedParameter->select(p | p.direction = ParameterDirectionKind::out and p.parameterSet->isEmpty()->forall(isStream)))
```

- two\_parameter\_sets  
Two ParameterSets cannot have exactly the same set of Parameters.

```
inv: parameter->forall(parameterSet->forall(s1, s2 | s1->size() = s2->size() implies s1.parameter->exists(p | not s2.parameter->includes(p))))
```

## Property [Class]

### Description

A Property is a StructuralFeature. A Property related by ownedAttribute to a Classifier (other than an association) represents an attribute and might also represent an association end. It relates an instance of the Classifier to a value or set of values of the type of the attribute. A Property related by memberEnd to an Association represents an end of the Association. The type of the Property is the type of the end of the Association. A Property has the capability of being a DeploymentTarget in a Deployment relationship. This enables modeling the deployment to hierarchical nodes that have Properties functioning as internal parts. Property specializes ParameterableElement to specify that a Property can be exposed as a formal template parameter, and provided as an actual parameter in a binding of a template.



## Diagrams

[Classifiers](#), [Properties](#), [Encapsulated Classifiers](#), [Structured Classifiers](#), [Classes](#), [Associations](#), [DataTypes](#), [Signals](#), [Interfaces](#), [Profiles](#), [Deployments](#), [Artifacts](#), [Link End Data](#), [Link Object Actions](#)

## Generalizations

[ConnectableElement](#), [DeploymentTarget](#), [StructuralFeature](#)

## Specializations

[Port](#), [ExtensionEnd](#)

## Attributes

- aggregation : [AggregationKind](#) [1..1] = none  
Specifies the kind of aggregation that applies to the Property.
- /default : [String](#) [0..1]  
A String that is evaluated to give a default value for the Property when an object of the owning Classifier is instantiated.
- /isComposite : [Boolean](#) [1..1] = false  
If isComposite is true, the object containing the attribute is a container for the object or value contained in the attribute. This is a derived value, indicating whether the aggregation of the Property is composite or not.
- isDerived : [Boolean](#) [1..1] = false  
Specifies whether the Property is derived, i.e., whether its value or values can be computed from other information.
- isDerivedUnion : [Boolean](#) [1..1] = false  
Specifies whether the property is derived as the union of all of the Properties that are constrained to subset it.
- isID : [Boolean](#) [1..1] = false  
True indicates this property can be used to uniquely identify an instance of the containing Class.
- isReadOnly : [Boolean](#) [1..1] = false  
If isReadOnly is true, the Property may not be written to after initialization.

## Association Ends

- association : [Association](#) [0..1]{subsets [A\\_member\\_memberNamespace::memberNamespace](#)} (opposite [Association::memberEnd](#))  
The Association of which this Property is a member, if any.
- associationEnd : [Property](#) [0..1]{subsets [Element::owner](#)} (opposite [Property::qualifier](#))  
Designates the optional association end that owns a qualifier attribute.
- class : [Class](#) [0..1]{subsets [NamedElement::namespace](#), subsets [A\\_ownedAttribute\\_structuredClassifier::structuredClassifier](#), subsets [A\\_attribute\\_classifier::classifier](#)} (opposite [Class::ownedAttribute](#))  
The Class that owns this Property, if any.

- datatype : [DataType](#) [0..1]{subsets [NamedElement::namespace](#), subsets [A\\_attribute\\_classifier::classifier](#)} (opposite [DataType::ownedAttribute](#))  
The DataType that owns this Property, if any.
- ♦ defaultValue : [ValueSpecification](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_defaultValue\\_owningProperty::owningProperty](#))  
A ValueSpecification that is evaluated to give a default value for the Property when an object of the owning Classifier is instantiated.
- interface : [Interface](#) [0..1]{subsets [NamedElement::namespace](#), subsets [A\\_attribute\\_classifier::classifier](#)} (opposite [Interface::ownedAttribute](#))  
The Interface that owns this Property, if any.
- /opposite : [Property](#) [0..1] (opposite [A\\_opposite\\_property::property](#))  
In the case where the Property is one end of a binary association this gives the other end.
- owningAssociation : [Association](#) [0..1]{subsets [Feature::featuringClassifier](#), subsets [NamedElement::namespace](#), subsets [Property::association](#), subsets [RedefinableElement::redefinitionContext](#)} (opposite [Association::ownedEnd](#))  
The owning association of this property, if any.
- ♦ qualifier : [Property](#) [0..\*]{ordered, subsets [Element::ownedElement](#)} (opposite [Property::associationEnd](#))  
An optional list of ordered qualifier attributes for the end.
- redefinedProperty : [Property](#) [0..\*]{subsets [RedefinableElement::redefinedElement](#)} (opposite [A\\_redefinedProperty\\_property::property](#))  
The properties that are redefined by this property, if any.
- subsettedProperty : [Property](#) [0..\*] (opposite [A\\_subsettedProperty\\_property::property](#))  
The properties of which this Property is constrained to be a subset, if any.

## Operations

- default() : [String](#) [0..1]  
Derivation for Property::default  
  
`body: if self.type = String then defaultValue.stringValue() else null endif`
- isAttribute(p : [Property](#)) : [Boolean](#)  
The query isAttribute() is true if the Property is defined as an attribute of some Classifier.  
  
`body: Classifier.allInstances()->exists(c | c.attribute->includes(p))`
- isCompatibleWith(p : [ParameterableElement](#)) : [Boolean](#)  
The query isCompatibleWith() determines if this Property is compatible with the specified ParameterableElement. By default ParameterableElement P is compatible with ParameterableElement Q if the kind of P is the same or a subtype as the kind of Q. In addition, for Properties, the type must be conformant with the type of the specified ParameterableElement.  
  
`body: p.ocIsKindOf(self.ocType()) and self.type.conformsTo(p.ocAsType(Property).type)`

- `isComposite()` : [Boolean](#)  
The value of `isComposite` is true only if aggregation is composite.

```
body: aggregation = AggregationKind::composite
```

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

```
pre: redefinee.isRedefinitionContextValid(self)
body: redefinee.ocIsKindOf(Property) and
  let prop : Property = redefinee.ocAsType(Property) in
    (prop.type.conformsTo(self.type) and
      ((prop.lowerBound()->notEmpty() and self.lowerBound()->notEmpty()) implies prop.lowerBound() >=
self.lowerBound()) and
      ((prop.upperBound()->notEmpty() and self.upperBound()->notEmpty()) implies prop.lowerBound() <=
self.lowerBound()) and
      (self.isComposite implies prop.isComposite))
```

- `isNavigable()` : [Boolean](#)  
The query `isNavigable()` indicates whether it is possible to navigate across the property.

```
body: not classifier->isEmpty() or association.navigableOwnedEnd->includes(self)
```

- `opposite()` : [Property](#)  
If this property is a memberEnd of a binary association, then `opposite` gives the other end.

```
body: if association <> null and association.memberEnd->size() = 2
then
  association.memberEnd->any(e | e <> self)
else
  null
endif
```

- `subsettingContext()` : [Type](#) [0..\*]  
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.

```
body: if association <> null
then association.endType->excluding(type)
else
  if classifier<>null
  then classifier->asSet()
  else Set{}
  endif
endif
```

## Constraints

- `subsetting_context_conforms`  
Subsetting may only occur when the context of the subsetting property conforms to the context of the subsetted property.

```
inv: subsettedProperty->notEmpty() implies
  (subsettingContext()->notEmpty() and subsettingContext()->forall(sc |
```

```
subsettingProperty->forall(sp |
  sp.subsettingContext()->exists(c | sc.conformsTo(c))))
```

- **derived\_union\_is\_read\_only**  
A derived union is read only.

```
inv: isDerivedUnion implies isReadOnly
```

- **multiplicity\_of\_composite**  
A multiplicity on the composing end of a composite aggregation must not have an upper bound greater than 1.

```
inv: isComposite and association <> null implies opposite.upperBound() <= 1
```

- **redefined\_property\_inherited**  
A redefined Property must be inherited from a more general Classifier.

```
inv: (redefinedProperty->notEmpty()) implies
  (redefinitionContext->notEmpty() and
   redefinedProperty->forall(rp |
    (redefinitionContext->collect(fc |
     fc.allParents())->asSet()->collect(c | c.allFeatures())->asSet()->includes(rp)))
```

- **subsetting\_rules**  
A subsetting Property may strengthen the type of the subsetting Property, and its upper bound may be less.

```
inv: subsettingProperty->forall(sp |
  self.type.conformsTo(sp.type) and
  ((self.upperBound()->notEmpty() and sp.upperBound()->notEmpty()) implies
   self.upperBound() <= sp.upperBound() ))
```

- **binding\_to\_attribute**  
A binding of a PropertyTemplateParameter representing an attribute must be to an attribute.

```
inv: (isAttribute(self) and (templateParameterSubstitution->notEmpty()))
  implies (templateParameterSubstitution->forall(ts | ts.formal.oclIsKindOf(Property) and
  isAttribute(ts.formal.oclAsType(Property))))
```

- **derived\_union\_is\_derived**  
A derived union is derived.

```
inv: isDerivedUnion implies isDerived
```

- **deployment\_target**  
A Property can be a DeploymentTarget if it is a kind of Node and functions as a part in the internal structure of an encompassing Node.

```
inv: deployment->notEmpty() implies owner.oclIsKindOf(Node) and Node.allInstances()->exists(n |
n.part->exists(p | p = self))
```

- **subsetting\_property\_names**  
A Property may not subset a Property with the same name.

```
inv: subsettedProperty->forall(sp | sp.name <> name)
```

- **type\_of\_opposite\_end**  
If a Property is a classifier-owned end of a binary Association, its owner must be the type of the opposite end.

```
inv: (opposite->notEmpty() and owningAssociation->isEmpty()) implies classifier = opposite.type
```

- **qualified\_is\_association\_end**  
All qualified Properties must be Association ends

```
inv: qualifier->notEmpty() implies association->notEmpty()
```

## RedefinableElement [Abstract Class]

### Description

A RedefinableElement 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.

### Diagrams

[Classifiers](#), [Classifier Templates](#), [Features](#), [Activities](#), [Use Cases](#), [State Machine Redefinition](#)

### Generalizations

[NamedElement](#)

### Specializations

[Classifier](#), [Feature](#), [RedefinableTemplateSignature](#), [ActivityEdge](#), [ActivityNode](#), [ExtensionPoint](#), [Region](#), [State](#), [Transition](#)

### Attributes

- **isLeaf** : [Boolean](#) [1..1] = false  
Indicates whether it is possible to further redefine a RedefinableElement. If the value is true, then it is not possible to further redefine the RedefinableElement.

### Association Ends

- **/redefinedElement** : [RedefinableElement](#) [0..\*]{union} (opposite : [A\\_redefinedElement\\_redefinableElement::redefinableElement](#))  
The RedefinableElement that is being redefined by this element.
- **/redefinitionContext** : [Classifier](#) [0..\*]{union} (opposite : [A\\_redefinitionContext\\_redefinableElement::redefinableElement](#))  
The contexts that this element may be redefined from.

## Operations

- `isConsistentWith(redefinee : RedefinableElement) : Boolean`
- 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.

```
pre: redefinee.isRedefinitionContextValid(self)
body: false
```

- `isRedefinitionContextValid(redefined : RedefinableElement) : Boolean`
- 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.

```
body: redefinitionContext->exists(c | c.allParents()->includesAll(redefined.redefinitionContext))
```

## Constraints

- `redefinition_consistent`  
A redefining element must be consistent with each redefined element.

```
inv: redefinedElement->forAll(isConsistentWith(self))
```

- `non_leaf_redefinition`  
A `RedefinableElement` can only redefine non-leaf `RedefinableElements`.

```
inv: redefinedElement->forAll(not isLeaf)
```

- `redefinition_context_valid`  
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.

```
inv: redefinedElement->forAll(e | isRedefinitionContextValid(e))
```

## RedefinableTemplateSignature [Class]

### Description

A `RedefinableTemplateSignature` supports the addition of formal template parameters in a specialization of a template classifier.

### Diagrams

[Classifier Templates](#)

### Generalizations

[RedefinableElement](#), [TemplateSignature](#)

## Association Ends

- classifier : [Classifier](#) [1..1]{subsets [RedefinableElement::redefinitionContext](#), redefines [TemplateSignature::template](#)} (opposite [Classifier::ownedTemplateSignature](#))  
The Classifier that owns this RedefinableTemplateSignature.
- extendedSignature : [RedefinableTemplateSignature](#) [0..\*]{subsets [RedefinableElement::redefinedElement](#)} (opposite [A\\_extendedSignature\\_redefinableTemplateSignature::redefinableTemplateSignature](#))  
The RedefinableTemplateSignature that is extended by this RedefinableTemplateSignature.
- /inheritedParameter : [TemplateParameter](#) [0..\*]{subsets [TemplateSignature::parameter](#)} (opposite [A\\_inheritedParameter\\_redefinableTemplateSignature::redefinableTemplateSignature](#))  
The formal template parameters of the extended signature.

## Operations

- inheritedParameter() : [TemplateParameter](#) [0..\*]  
Derivation for RedefinableTemplateSignature::/inheritedParameter  

```
body: if extendedSignature->isEmpty() then Set{} else extendedSignature.parameter->asSet() endif
```
- isConsistentWith(redefinee : [RedefinableElement](#)) : [Boolean](#)  
The query isConsistentWith() specifies, for any two RedefinableTemplateSignatures in a context in which redefinition is possible, whether redefinition would be logically consistent. A redefining template signature is always consistent with a redefined template signature, as redefinition only adds new formal parameters.

```
pre: redefinee.isRedefinitionContextValid(self)
body: redefinee.oclIsKindOf(RedefinableTemplateSignature)
```

## Constraints

- redefines\_parents  
If any of the parent Classifiers are a template, then the extendedSignature must include the signature of that Classifier.

```
inv: classifier.allParents()->forall(c | c.ownedTemplateSignature->notEmpty() implies self->closure(extendedSignature)->includes(c.ownedTemplateSignature))
```

## Slot [Class]

### Description

A Slot designates that an entity modeled by an InstanceSpecification has a value or values for a specific StructuralFeature.

### Diagrams

[Instances](#)

### Generalizations

[Element](#)

## Association Ends

- definingFeature : [StructuralFeature](#) [1..1] (opposite [A\\_definingFeature\\_slot::slot](#))  
The StructuralFeature that specifies the values that may be held by the Slot.
- owningInstance : [InstanceSpecification](#) [1..1]{subsets [Element::owner](#)} (opposite [InstanceSpecification::slot](#))  
The InstanceSpecification that owns this Slot.
- ♦ value : [ValueSpecification](#) [0..\*]{ordered, subsets [Element::ownedElement](#)} (opposite [A\\_value\\_owningSlot::owningSlot](#))  
The value or values held by the Slot.

## StructuralFeature [Abstract Class]

### Description

A StructuralFeature is a typed feature of a Classifier that specifies the structure of instances of the Classifier.

### Diagrams

[Features](#), [Properties](#), [Instances](#), [Structural Feature Actions](#)

### Generalizations

[MultiplicityElement](#), [TypedElement](#), [Feature](#)

### Specializations

[Property](#)

### Attributes

- isReadOnly : [Boolean](#) [1..1] = false  
True if the StructuralFeature's value may not be modified by a client.

## Substitution [Class]

### Description

A substitution is a relationship between two classifiers signifying that the substituting classifier complies with the contract specified by the contract classifier. This implies that instances of the substituting classifier are runtime substitutable where instances of the contract classifier are expected.

### Diagrams

[Classifiers](#)

### Generalizations

[Realization](#)



## Association Ends

- contract : [Classifier](#) [1..1]{subsets [Dependency::supplier](#)} (opposite [A\\_contract\\_substitution::substitution](#))  
The contract with which the substituting classifier complies.
- substitutingClassifier : [Classifier](#) [1..1]{subsets [Dependency::client](#), subsets [Element::owner](#)} (opposite [Classifier::substitution](#))  
Instances of the substituting classifier are runtime substitutable where instances of the contract classifier are expected.

## 9.10 Association Descriptions

### A\_attribute\_classifier [Association]

#### Diagrams

[Classifiers](#)

#### Owned Ends

- classifier : [Classifier](#) [0..1]{subsets [Feature::featuringClassifier](#), subsets [RedefinableElement::redefinitionContext](#)}  
(opposite [Classifier::attribute](#))

### A\_bodyCondition\_bodyContext [Association]

#### Diagrams

[Operations](#)

#### Owned Ends

- bodyContext : [Operation](#) [0..1]{subsets [Constraint::context](#)} (opposite [Operation::bodyCondition](#))

### A\_classifier\_instanceSpecification [Association]

#### Diagrams

[Instances](#)

#### Specializations

[A\\_classifier\\_enumerationLiteral](#)

## Owned Ends

- instanceSpecification : [InstanceSpecification](#) [0..\*] (opposite [InstanceSpecification::classifier](#))

## A\_classifier\_templateParameter\_parameteredElement [Association]

### Diagrams

[Classifier Templates](#)

### Member Ends

- [Classifier::templateParameter](#)
- [ClassifierTemplateParameter::parameteredElement](#)

## A\_collaborationUse\_classifier [Association]

### Diagrams

[Classifiers](#), [Collaborations](#)

### Specializations

[A representation classifier](#)

### Owned Ends

- classifier : [Classifier](#) [0..1]{subsets [Element::owner](#)} (opposite [Classifier::collaborationUse](#))

## A\_condition\_parameterSet [Association]

### Diagrams

[Features](#)

### Owned Ends

- parameterSet : [ParameterSet](#) [0..1]{subsets [Element::owner](#)} (opposite [ParameterSet::condition](#))

## A\_constrainingClassifier\_classifierTemplateParameter [Association]

### Diagrams

[Classifier Templates](#)

## Owned Ends

- classifierTemplateParameter : [ClassifierTemplateParameter](#) [0..\*] (opposite [ClassifierTemplateParameter::constrainingClassifier](#))

## A\_contract\_substitution [Association]

### Diagrams

[Classifiers](#)

## Owned Ends

- substitution : [Substitution](#) [0..\*]{subsets [A\\_supplier\\_supplierDependency::supplierDependency](#)} (opposite [Substitution::contract](#))

## A\_defaultValue\_owningParameter [Association]

### Diagrams

[Features](#)

## Owned Ends

- owningParameter : [Parameter](#) [0..1]{subsets [Element::owner](#)} (opposite [Parameter::defaultValue](#))

## A\_defaultValue\_owningProperty [Association]

### Diagrams

[Properties](#)

## Owned Ends

- owningProperty : [Property](#) [0..1]{subsets [Element::owner](#)} (opposite [Property::defaultValue](#))

## A\_definingFeature\_slot [Association]

### Diagrams

[Instances](#)

## Owned Ends

- slot : [Slot](#) [0..\*] (opposite [Slot::definingFeature](#))

## A\_extendedSignature\_redefinableTemplateSignature [Association]

### Diagrams

[Classifier Templates](#)

## Owned Ends

- redefinableTemplateSignature : [RedefinableTemplateSignature](#) [0..\*]{subsets [A\\_redefinedElement\\_redefinableElement::redefinableElement](#)} (opposite [RedefinableTemplateSignature::extendedSignature](#))

## A\_feature\_featuringClassifier [Association]

### Diagrams

[Classifiers, Features](#)

## Member Ends

- [Classifier::feature](#)
- [Feature::featuringClassifier](#)

## A\_general\_classifier [Association]

### Diagrams

[Classifiers](#)

## Owned Ends

- classifier : [Classifier](#) [0..\*] (opposite [Classifier::general](#))

## A\_general\_generalization [Association]

### Diagrams

[Classifiers](#)

## Owned Ends

- generalization : [Generalization](#) [0..\*]{ subsets [A\\_target\\_directedRelationship::directedRelationship](#) } (opposite [Generalization::general](#))

## A\_generalizationSet\_generalization [Association]

### Diagrams

[Classifiers](#), [Generalization Sets](#)

### Member Ends

- [Generalization::generalizationSet](#)
- [GeneralizationSet::generalization](#)

## A\_generalization\_specific [Association]

### Diagrams

[Classifiers](#)

### Member Ends

- [Classifier::generalization](#)
- [Generalization::specific](#)

## A\_inheritedMember\_inheritingClassifier [Association]

### Diagrams

[Classifiers](#)

### Owned Ends

- inheritingClassifier : [Classifier](#) [0..\*]{ subsets [A\\_member\\_memberNamespace::memberNamespace](#) } (opposite [Classifier::inheritedMember](#))

## A\_inheritedParameter\_redefinableTemplateSignature [Association]

### Diagrams

[Classifier Templates](#)

## Owned Ends

- redefinableTemplateSignature : [RedefinableTemplateSignature](#) [0..\*]{subsets [A\\_parameter\\_templateSignature::templateSignature](#)} (opposite [RedefinableTemplateSignature::inheritedParameter](#))

## A\_instance\_instanceValue [Association]

### Diagrams

[Instances](#)

## Owned Ends

- instanceValue : [InstanceValue](#) [0..\*] (opposite [InstanceValue::instance](#))

## A\_method\_specification [Association]

### Diagrams

[Features](#), [Behaviors](#)

## Member Ends

- [BehavioralFeature::method](#)
- [Behavior::specification](#)

## A\_operation\_templateParameter\_parameteredElement [Association]

### Diagrams

[Operations](#)

## Member Ends

- [Operation::templateParameter](#)
- [OperationTemplateParameter::parameteredElement](#)

## A\_opposite\_property [Association]

### Diagrams

[Properties](#)

## Owned Ends

- property : [Property](#) [0..1] (opposite [Property::opposite](#))

## A\_ownedParameterSet\_behavioralFeature [Association]

### Diagrams

[Features](#)

## Owned Ends

- behavioralFeature : [BehavioralFeature](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [BehavioralFeature::ownedParameterSet](#))

## A\_ownedParameter\_operation [Association]

### Diagrams

[Operations](#)

## Member Ends

- [Operation::ownedParameter](#)
- [Parameter::operation](#)

## A\_ownedParameter\_ownerFormalParam [Association]

### Diagrams

[Features](#)

## Owned Ends

- ownerFormalParam : [BehavioralFeature](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [BehavioralFeature::ownedParameter](#))

## A\_ownedTemplateSignature\_classifier [Association]

### Diagrams

[Classifier Templates](#)

## Member Ends

- [Classifier::ownedTemplateSignature](#)
- [RedefinableTemplateSignature::classifier](#)

## A\_ownedUseCase\_classifier [Association]

### Diagrams

[Classifiers](#), [Use Cases](#)

### Owned Ends

- classifier : [Classifier](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [Classifier::ownedUseCase](#))

## A\_parameterSet\_parameter [Association]

### Diagrams

[Features](#)

### Member Ends

- [Parameter::parameterSet](#)
- [ParameterSet::parameter](#)

## A\_postcondition\_postContext [Association]

### Diagrams

[Operations](#)

### Owned Ends

- postContext : [Operation](#) [0..1]{subsets [Constraint::context](#)} (opposite [Operation::postcondition](#))

## A\_powertypeExtent\_powertype [Association]

### Diagrams

[Classifiers](#), [Generalization Sets](#)

### Member Ends

- [Classifier::powertypeExtent](#)
- [GeneralizationSet::powertype](#)



## A\_precondition\_preContext [Association]

### Diagrams

[Operations](#)

### Owned Ends

- precondition : [Operation](#) [0..1]{subsets [Constraint::context](#)} (opposite [Operation::precondition](#))

## A\_qualifier\_associationEnd [Association]

### Diagrams

[Properties, Associations](#)

### Member Ends

- [Property::qualifier](#)
- [Property::associationEnd](#)

## A\_raisedException\_behavioralFeature [Association]

### Diagrams

[Features](#)

### Owned Ends

- behavioralFeature : [BehavioralFeature](#) [0..\*] (opposite [BehavioralFeature::raisedException](#))

## A\_raisedException\_operation [Association]

### Diagrams

[Operations](#)

### Owned Ends

- operation : [Operation](#) [0..\*]{subsets [A\\_raisedException\\_behavioralFeature::behavioralFeature](#)} (opposite [Operation::raisedException](#))

## A\_redefinedClassifier\_classifier [Association]

### Diagrams

[Classifiers](#)

### Owned Ends

- classifier : [Classifier](#) [0..\*]{subsets [A\\_redefinedElement\\_redefinableElement::redefinableElement](#)} (opposite [Classifier::redefinedClassifier](#))

## A\_redefinedElement\_redefinableElement [Association]

### Diagrams

[Classifiers](#)

### Owned Ends

- redefinableElement : [RedefinableElement](#) [0..\*] (opposite [RedefinableElement::redefinedElement](#))

## A\_redefinedOperation\_operation [Association]

### Diagrams

[Operations](#)

### Owned Ends

- operation : [Operation](#) [0..\*]{subsets [A\\_redefinedElement\\_redefinableElement::redefinableElement](#)} (opposite [Operation::redefinedOperation](#))

## A\_redefinedProperty\_property [Association]

### Diagrams

[Properties](#)

### Owned Ends

- property : [Property](#) [0..\*]{subsets [A\\_redefinedElement\\_redefinableElement::redefinableElement](#)} (opposite [Property::redefinedProperty](#))

## A\_redefinitionContext\_redefinableElement [Association]

### Diagrams

[Classifiers](#)

### Specializations

[A\\_redefinitionContext\\_transition](#), [A\\_redefinitionContext\\_state](#), [A\\_redefinitionContext\\_region](#)

### Owned Ends

- redefinableElement : [RedefinableElement](#) [0..\*] (opposite [RedefinableElement::redefinitionContext](#))

## A\_representation\_classifier [Association]

### Diagrams

[Classifiers](#), [Collaborations](#)

### Generalizations

[A\\_collaborationUse\\_classifier](#)

### Owned Ends

- classifier : [Classifier](#) [0..1]{redefines [A\\_collaborationUse\\_classifier::classifier](#)} (opposite [Classifier::representation](#))

## A\_slot\_owningInstance [Association]

### Diagrams

[Instances](#)

### Member Ends

- [InstanceSpecification::slot](#)
- [Slot::owningInstance](#)

## A\_specification\_owningInstanceSpec [Association]

### Diagrams

[Instances](#)

## Owned Ends

- owningInstanceSpec : [InstanceSpecification](#) [0..1]{subsets [Element::owner](#)} (opposite [InstanceSpecification::specification](#))

## A\_subsettedProperty\_property [Association]

### Diagrams

[Properties](#)

## Owned Ends

- property : [Property](#) [0..\*] (opposite [Property::subsettedProperty](#))

## A\_substitution\_substitutingClassifier [Association]

### Diagrams

[Classifiers](#)

## Member Ends

- [Classifier::substitution](#)
- [Substitution::substitutingClassifier](#)

## A\_type\_operation [Association]

### Diagrams

[Operations](#)

## Owned Ends

- operation : [Operation](#) [0..\*] (opposite [Operation::type](#))

## A\_value\_owningSlot [Association]

### Diagrams

[Instances](#)

## Owned Ends

- owningSlot : [Slot](#) [0..1]{subsets [Element::owner](#)} (opposite [Slot::value](#))

# 10 Simple Classifiers

## 10.1 Summary

This clause specifies various kinds of Classifier that do not have complex internal structure.

## 10.2 DataTypes

### 10.2.1 Summary

DataTypes model Types whose instances are distinguished only by their value.

### 10.2.2 Abstract Syntax

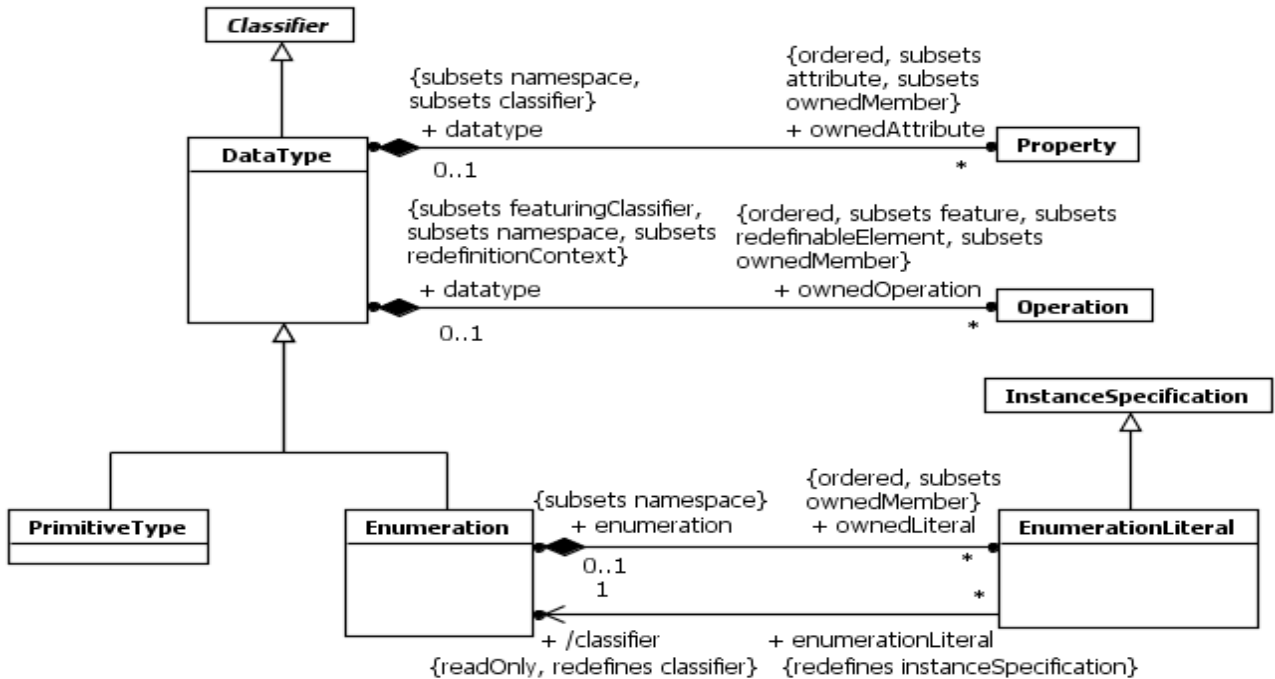


Figure 10.1 DataTypes

### 10.2.3 Semantics

#### DataTypes

A DataType is a kind of Classifier. DataType differs from Class in that instances of a DataType are identified only by their value. All instances of a DataType with the same value are considered to be equal instances.

If a `DataType` has attributes (i.e., Properties owned by it and in its namespace) it is called a *structured* `DataType`. Instances of a structured `DataType` contain attribute values matching its attributes. Instances of a structured `DataType` are considered to be equal if and only if the structure is the same and the values of the corresponding attributes are equal.

A `DataType` may be parameterized, bound, and used as `TemplateParameters`.

## Primitive Types

A `PrimitiveType` defines a predefined `DataType`, without any substructure. A `PrimitiveType` may have algebra and operations defined outside of UML, for example, mathematically. The run-time instances of a `PrimitiveType` are values that correspond to mathematical elements defined outside of UML (for example, the Integers).

## Enumerations

Enumeration is a kind of `DataType`. Each value of an Enumeration corresponds to one of its user-defined `EnumerationLiterals`.

As a specialization of `Classifier`, Enumerations can participate in generalization relationships. An Enumeration that specializes another may define new `EnumerationLiterals`; in such a case the set of applicable literals comprises inherited literals plus locally-defined ones.

An `EnumerationLiteral` defines an element of the run-time extension of an Enumeration. The run-time values corresponding to `EnumerationLiterals` may be compared for equality.

An `EnumerationLiteral` has a name that shall be used to identify it within its Enumeration. The `EnumerationLiteral` name is scoped within and shall be unique within its Enumeration. `EnumerationLiteral` names shall be qualified for general use.

### 10.2.4 Notation

A `DataType` is designated using the `Classifier` notation (a rectangle) with keyword `«dataType»` or, when it is referenced (e.g., by an attribute), by the name of the `DataType`. A compartment listing the attributes is placed below the name compartment. A compartment listing the Operations is placed below the attribute compartment.

A `PrimitiveType` is similarly designated with the keyword `«primitive»` above or before the name of the `PrimitiveType`.

An Enumeration is similarly designated. The name of the Enumeration is placed in the upper compartment with the keyword `«enumeration»` above or before the name. A list of `EnumerationLiterals` may be placed, one to a line, in a compartment below the operations compartment. The attributes and operations compartments may be suppressed, and typically are suppressed and empty.

### 10.2.5 Examples

Figure 10.2 illustrates the notation for defining a `PrimitiveType`.



**Figure 10.2 PrimitiveType Notation**

Figure 10.3 illustrates the notation for defining `DataTypes`. The `FullName` type defined on the left is used as the type of the `fullName` attribute in the `Person` type defined on the right.



Figure 10.3 DataType Notation

Figure 10.4 illustrates the notation for defining Enumerations.



Figure 10.4 Enumeration Notation

## 10.3 Signals

### 10.3.1 Summary

Signals and Receptions are used to model asynchronous communication between objects.

### 10.3.2 Abstract Syntax

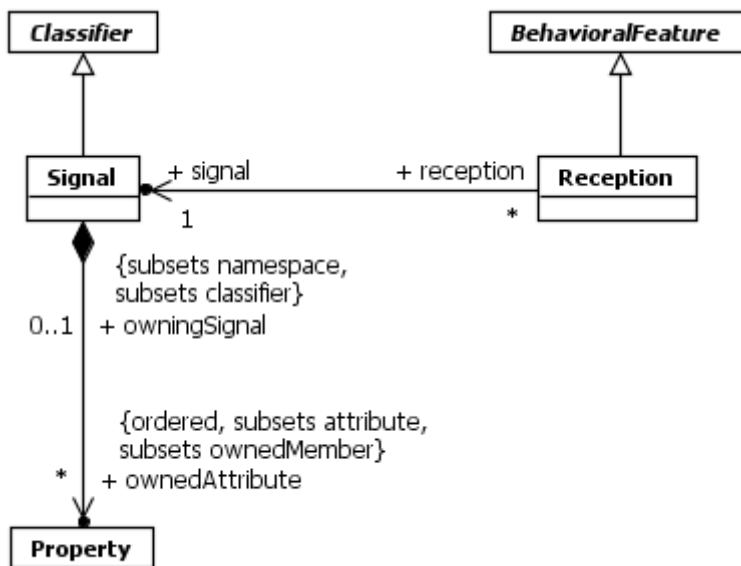


Figure 10.5 Signals



### 10.3.3 Semantics

#### Signals

A Signal is a specification of a kind of communication between objects in which a reaction is asynchronously triggered in the receiver without a reply. The receiving object handles Signals as specified by its Receptions. The data carried by the communication are represented as attributes of the Signal. A Signal is defined independently of the Classifiers handling it.

The sender of a Signal will not block waiting for a reply but continue execution immediately. By declaring a Reception associated to a given Signal, a Classifier specifies that its instances will be able to receive that Signal, or a subtype thereof, and will respond to it with the designated Behavior.

A Signal may be parameterized, bound, and used as TemplateParameters.

#### Receptions

A Reception specifies that its owning Class or Interface is prepared to react to the receipt of a Signal. A Reception matches a Signal if the received Signal is a specialization of the Reception's signal. The details of how the object responds to the received Signal depend on the kind of Behavior associated with the Reception and its owning Class or Interface. See [13.2](#). The name of the Reception is the same as the name of the Signal. A Reception may only have in Parameters (see [9.4.3](#)) that match the attributes of the Signal by name, type, and multiplicity.

### 10.3.4 Notation

A Signal is depicted by a Classifier symbol with the keyword «signal».

Receptions are shown in the receptions compartment using the same notation as for Operations with the keyword «signal».

### 10.3.5 Examples

Figure 10.6 shows an interface IAlarm that defines two Receptions, each referring to a Signal also shown in the example.

**NOTE.** The name of the Reception matches the name of the Signal, and the parameter of the Reception matches the attribute of the Signal.

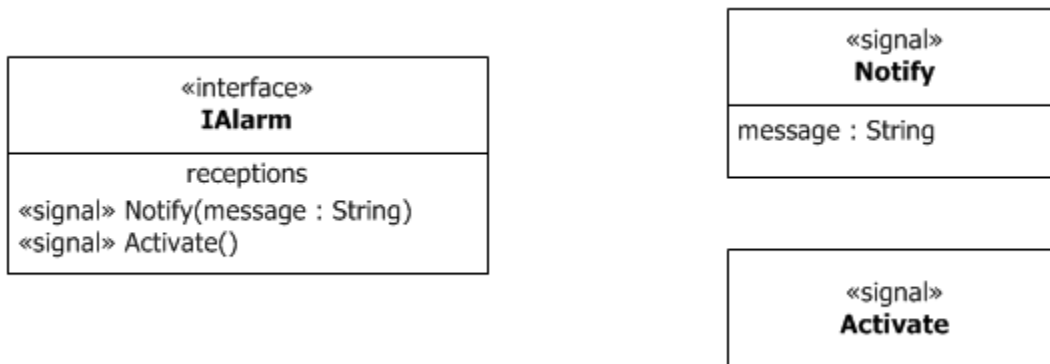


Figure 10.6 Reception Notation

## 10.4 Interfaces

### 10.4.1 Summary

Interfaces declare coherent services that are implemented by BehavedClassifiers that implement the Interfaces via InterfaceRealizations.

### 10.4.2 Abstract Syntax

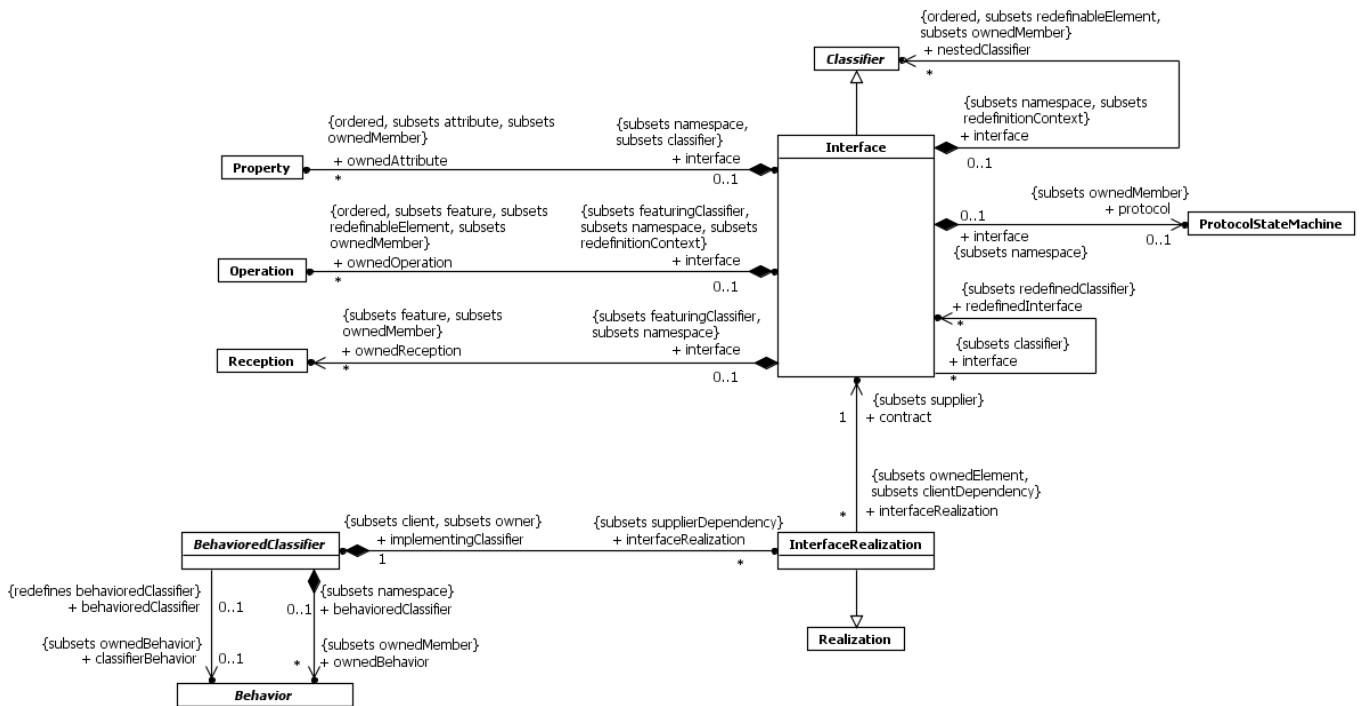


Figure 10.7 Interfaces

### 10.4.3 Semantics

#### Interfaces

An Interface is a kind of Classifier that represents a declaration of a set of public Features and obligations that together constitute a coherent service. An Interface specifies a contract; any instance of a Classifier that realizes the Interface shall fulfill that contract. The obligations associated with an Interface are in the form of constraints (such as pre- and postconditions) or protocol specifications, which may impose ordering restrictions on interactions through the Interface.

Interfaces may not be instantiated. Instead, an Interface specification is *implemented* or *realized* by a BehavedClassifier, which means that the BehavedClassifier presents a public facade that conforms to the Interface specification.

**NOTE.** A given BehavedClassifier may implement more than one Interface and that an Interface may be implemented by a number of different BehavedClassifiers.

Interfaces provide a way to partition and characterize groups of public Features and obligations that realizing BehavoredClassifiers shall possess. An Interface does not specify how it is to be implemented, but merely what needs to be supported by realizing BehavoredClassifiers. That is, such BehavoredClassifiers shall provide a public façade consisting of attributes, Operations, and externally observable Behavior that conforms to the Interface.

**NOTE.** If an Interface declares an attribute, this does not necessarily mean that the realizing BehavoredClassifier will necessarily have such an attribute in its implementation, but only that it will appear so to external observers.

The set of Interfaces realized by a BehavoredClassifier are its *provided* Interfaces, which represent the services and obligations that instances of that BehavoredClassifier offer to their clients. Interfaces may also be used to specify *required* Interfaces, which are specified by a Usage dependency between the BehavoredClassifier and the corresponding Interfaces. Required Interfaces specify services that a BehavoredClassifier needs in order to perform its function and fulfill its own obligations to its clients.

Properties owned by Interfaces (including Association ends) imply that the realizing BehavoredClassifier should maintain information corresponding to the type and multiplicity of the Property and facilitate retrieval and modification of that information. A Property declared on an Interface does not necessarily imply that there will be such a Property on a realizing BehavoredClassifier (e.g., it may be realized by equivalent get and set Operations). Interfaces may also own constraints that impose constraints on the Features of the implementing BehavoredClassifier.

Interfaces may own a ProtocolStateMachine that specifies event sequences and pre/post conditions for the Operations and Receptions described by the Interface. A BehavoredClassifier realizing an Interface shall comply with the ProtocolStateMachine owned by the Interface.

An Interface may be parameterized, bound, and used as TemplateParameters.

An InterfaceRealization relationship between a BehavoredClassifier and an Interface implies that the BehavoredClassifier conforms to the contract specified by the Interface by supporting the set of Features owned by the Interface, and any of its parent Interfaces. For BehavioralFeatures, the implementing BehavoredClassifier will have an Operation or Reception for every Operation or Reception, respectively, defined by the Interface. For Properties, the realizing BehavoredClassifier will provide functionality that maintains the state represented by the Property. While such may be done by direct mapping to a Property of the realizing BehavoredClassifier, it may also be supported by the StateMachine of the BehavoredClassifier or by a pair of Operations that support the retrieval of the state information and an Operation that changes the state information.

#### 10.4.4 Notation

An Interface may be designated using the Classifier notation (a rectangle) with keyword «interface» preceding the name. A compartment listing the attributes is placed below the name compartment. A compartment listing the Operations is placed below the attribute compartment.

Alternatively an InterfaceRealization dependency from a BehavoredClassifier to an Interface may be shown by representing the Interface by a circle or *ball*, often also called *lollipop*, labeled with the name of the Interface, attached by a solid line to the BehavoredClassifier that realizes this Interface.

The Usage dependency from a Classifier to an Interface is shown by representing the Interface by a half-circle or *socket*, labeled with the name of the Interface, attached by a solid line to the Classifier that requires this Interface.

Interfaces inherited from a generalization of the BehavoredClassifier may be notated on a diagram through a lollipop. These Interfaces are indicated on the diagram by preceding the name of the Interface by a forward slash.

If a Dependency is wired from a Usage to an InterfaceRealization that are represented using a socket and a lollipop, the dependency arrow may be shown joining the socket to the lollipop

## 10.4.5 Examples

The InterfaceRealization dependency from ProximitySensor to ISensor is shown using *ball (lollipop)* notation (see Figure 10.8).

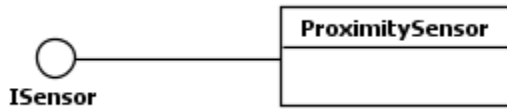


Figure 10.8 ISensor is a provided Interface of ProximitySensor

Figure 10.9 shows the lollipop notation for an inherited provided interface.

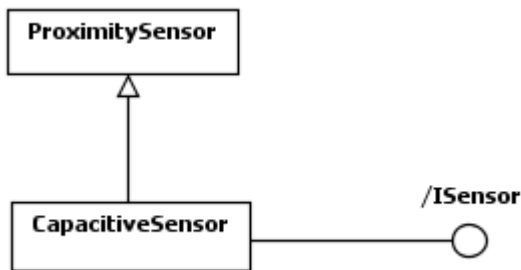


Figure 10.9 ISensor, a provided Interface of ProximitySensor, is shown as inherited by CapacitiveSensor

The Usage dependency from TheftAlarm to ISensor is shown using *socket* notation (see Figure 10.10).



Figure 10.10 ISensor is a required Interface of TheftAlarm

Alternatively, in cases where Interfaces are represented using the rectangle notation, InterfaceRealization and Usage dependencies are denoted with appropriate dependency arrows (see Figure 10.11). The Classifier at the tail of the arrow implements the Interface at the head of the arrow or uses that Interface, respectively.

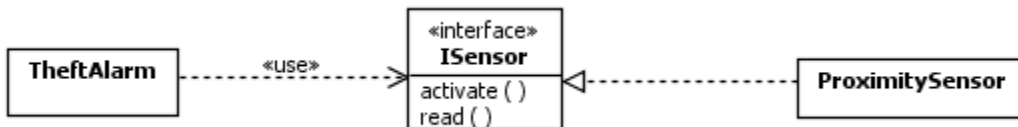


Figure 10.11 Alternative notation for required and provided Interface

It is often the case in practice that two or more Interfaces are mutually coupled through application-specific dependencies. In such situations, each Interface represents a specific role in a multi-party “protocol.” These types of protocol role couplings may be captured by Associations between Interfaces as shown in the example in Figure 10.12. This shows the specification of three Interfaces, *IAlarm*, *ISensor*, and *IBuzzer*. *IAlarm* and *ISensor* are shown as engaged in a bidirectional protocol, meaning that any implementation of *ISensor* must maintain the information needed to realize the *theAlarm* property, and similarly for *IAlarm* and *theSensor*. *IBuzzer* describes an Interface that implementers of *IAlarm* must be able to access.

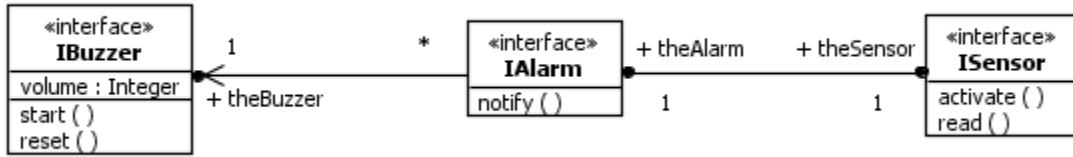


Figure 10.12 A set of collaborating Interfaces

## 10.5 Classifier Descriptions

### BehavedClassifier [Abstract Class]

#### Description

A BehavedClassifier may have InterfaceRealizations, and owns a set of Behaviors one of which may specify the behavior of the BehavedClassifier itself.

#### Diagrams

[Interfaces](#), [Use Cases](#), [Classes](#), [Collaborations](#), [Behaviors](#)

#### Generalizations

[Classifier](#)

#### Specializations

[Actor](#), [UseCase](#), [Class](#), [Collaboration](#)

#### Association Ends

- classifierBehavior : [Behavior](#) [0..1]{subsets [BehavedClassifier::ownedBehavior](#)} (opposite [A\\_classifierBehavior\\_behavedClassifier::behavedClassifier](#))  
A Behavior that specifies the behavior of the BehavedClassifier itself.
- ♦ interfaceRealization : [InterfaceRealization](#) [0..\*]{subsets [Element::ownedElement](#), subsets [NamedElement::clientDependency](#)} (opposite [InterfaceRealization::implementingClassifier](#))  
The set of InterfaceRealizations owned by the BehavedClassifier. Interface realizations reference the Interfaces of which the BehavedClassifier is an implementation.
- ♦ ownedBehavior : [Behavior](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [A\\_ownedBehavior\\_behavedClassifier::behavedClassifier](#))  
Behaviors owned by a BehavedClassifier.

#### Constraints

- class\_behavior  
If a behavior is classifier behavior, it does not have a specification.

```
inv: classifierBehavior->notEmpty() implies classifierBehavior.specification->isEmpty()
```

### DataType [Class]

#### Description

A DataType is a type whose instances are identified only by their value.

## Diagrams

[DataTypes](#), [Properties](#), [Operations](#)

## Generalizations

[Classifier](#)

## Specializations

[Enumeration](#), [PrimitiveType](#)

## Association Ends

- ♦ ownedAttribute : [Property](#) [0..\*]{ordered, subsets [Classifier::attribute](#), subsets [Namespace::ownedMember](#)} (opposite [Property::datatype](#))  
The Attributes owned by the DataType.
- ♦ ownedOperation : [Operation](#) [0..\*]{ordered, subsets [Classifier::feature](#), subsets [A\\_redefinitionContext\\_redefinableElement::redefinableElement](#), subsets [Namespace::ownedMember](#)} (opposite [Operation::datatype](#))  
The Operations owned by the DataType.

## Operations

- inherit(inhs : [NamedElement](#) [0..\*]) : [NamedElement](#) [0..\*]  
The inherit operation is overridden to exclude redefined properties.

```
body: inhs->reject(inh |
    inh.ocIsKindOf(RedefinableElement) and ownedMember->select(ocIsKindOf(RedefinableElement))-
    >select(redefinedElement->includes(inh.ocAsType(RedefinableElement)))->notEmpty())
```

## Enumeration [Class]

### Description

An Enumeration is a DataType whose values are enumerated in the model as EnumerationLiterals.

### Diagrams

[DataTypes](#)

### Generalizations

[DataType](#)

### Association Ends

- ♦ ownedLiteral : [EnumerationLiteral](#) [0..\*]{ordered, subsets [Namespace::ownedMember](#)} (opposite [EnumerationLiteral::enumeration](#))  
The ordered set of literals owned by this Enumeration.

## EnumerationLiteral [Class]

### Description

An EnumerationLiteral is a user-defined data value for an Enumeration.

### Diagrams

[DataTypes](#)

### Generalizations

[InstanceSpecification](#)

### Association Ends

- /classifier : [Enumeration](#) [1..1]{redefines [InstanceSpecification::classifier](#)} (opposite [A classifier enumerationLiteral::enumerationLiteral](#))  
The classifier of this EnumerationLiteral derived to be equal to its Enumeration.
- enumeration : [Enumeration](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [Enumeration::ownedLiteral](#))  
The Enumeration that this EnumerationLiteral is a member of.

### Operations

- classifier() : [Enumeration](#)  
Derivation of Enumeration::/classifier

body: enumeration

## Interface [Class]

### Description

Interfaces declare coherent services that are implemented by BehavioredClassifiers that implement the Interfaces via InterfaceRealizations.

### Diagrams

[Interfaces](#), [Encapsulated Classifiers](#), [Components](#), [Properties](#), [Operations](#)

### Generalizations

[Classifier](#)

### Association Ends

- ♦ nestedClassifier : [Classifier](#) [0..\*]{ordered, subsets [A redefinitionContext redefinableElement::redefinableElement](#), subsets [Namespace::ownedMember](#)} (opposite [A nestedClassifier interface::interface](#))  
References all the Classifiers that are defined (nested) within the Interface.



- ♦ ownedAttribute : [Property](#) [0..\*]{ordered, subsets [Classifier::attribute](#), subsets [Namespace::ownedMember](#)} (opposite [Property::interface](#))  
The attributes (i.e., the Properties) owned by the Interface.
- ♦ ownedOperation : [Operation](#) [0..\*]{ordered, subsets [Classifier::feature](#), subsets [A\\_redefinitionContext\\_redefinableElement::redefinableElement](#), subsets [Namespace::ownedMember](#)} (opposite [Operation::interface](#))  
The Operations owned by the Interface.
- ♦ ownedReception : [Reception](#) [0..\*]{subsets [Classifier::feature](#), subsets [Namespace::ownedMember](#)} (opposite [A\\_ownedReception\\_interface::interface](#))  
Receptions that objects providing this Interface are willing to accept.
- ♦ protocol : [ProtocolStateMachine](#) [0..1]{subsets [Namespace::ownedMember](#)} (opposite [A\\_protocol\\_interface::interface](#))  
References a ProtocolStateMachine specifying the legal sequences of the invocation of the BehavioralFeatures described in the Interface.
- redefinedInterface : [Interface](#) [0..\*]{subsets [Classifier::redefinedClassifier](#)} (opposite [A\\_redefinedInterface\\_interface::interface](#))  
References all the Interfaces redefined by this Interface.

## Constraints

- visibility  
The visibility of all Features owned by an Interface must be public.

```
inv: feature->forAll(visibility = VisibilityKind::public)
```

## InterfaceRealization [Class]

### Description

An InterfaceRealization is a specialized realization relationship between a BehavedClassifier and an Interface. This relationship signifies that the realizing BehavedClassifier conforms to the contract specified by the Interface.

### Diagrams

[Interfaces](#)

### Generalizations

[Realization](#)

### Association Ends

- contract : [Interface](#) [1..1]{subsets [Dependency::supplier](#)} (opposite [A\\_contract\\_interfaceRealization::interfaceRealization](#))  
References the Interface specifying the conformance contract.

- implementingClassifier : [BehavioredClassifier](#) [1..1]{subsets [Dependency::client](#), subsets [Element::owner](#)} (opposite [BehavioredClassifier::interfaceRealization](#))  
References the BehavioredClassifier that owns this InterfaceRealization, i.e., the BehavioredClassifier that realizes the Interface to which it refers.

## PrimitiveType [Class]

### Description

A PrimitiveType defines a predefined DataType, without any substructure. A PrimitiveType may have an algebra and operations defined outside of UML, for example, mathematically.

### Diagrams

[DataTypes](#)

### Generalizations

[DataType](#)

## Reception [Class]

### Description

A Reception is a declaration stating that a Classifier is prepared to react to the receipt of a Signal.

### Diagrams

[Signals](#), [Interfaces](#), [Classes](#)

### Generalizations

[BehavioralFeature](#)

### Association Ends

- signal : [Signal](#) [1..1] (opposite [A signal reception::reception](#))  
The Signal that this Reception handles.

### Constraints

- same\_name\_as\_signal  
A Reception has the same name as its signal  
  
`inv: name = signal.name`
- same\_structure\_as\_signal  
A Reception's parameters match the ownedAttributes of its signal by name, type, and multiplicity  
  
`inv: signal.ownedAttribute->size() = ownedParameter->size() and`

```

Sequence{1..signal.ownedAttribute->size()->forall( i |
  ownedParameter->at(i).direction = ParameterDirectionKind::_in' and
  ownedParameter->at(i).name = signal.ownedAttribute->at(i).name and
  ownedParameter->at(i).type = signal.ownedAttribute->at(i).type and
  ownedParameter->at(i).lowerBound() = signal.ownedAttribute->at(i).lowerBound() and
  ownedParameter->at(i).upperBound() = signal.ownedAttribute->at(i).upperBound()
)

```

## Signal [Class]

### Description

A Signal is a specification of a kind of communication between objects in which a reaction is asynchronously triggered in the receiver without a reply.

### Diagrams

[Signals](#), [Events](#), [Invocation Actions](#)

### Generalizations

[Classifier](#)

### Association Ends

- ♦ ownedAttribute : [Property](#) [0..\*]{ordered, subsets [Classifier::attribute](#), subsets [Namespace::ownedMember](#)} (opposite [A\\_ownedAttribute\\_owningSignal::owningSignal](#))  
The attributes owned by the Signal.

## 10.6 Association Descriptions

### A\_classifierBehavior\_behavoredClassifier [Association]

### Diagrams

[Interfaces](#), [Behaviors](#)

### Generalizations

[A\\_ownedBehavior\\_behavoredClassifier](#)

### Owned Ends

- behavedClassifier : [BehavoredClassifier](#) [0..1]{redefines [A\\_ownedBehavior\\_behavoredClassifier::behavedClassifier](#)} (opposite [BehavoredClassifier::classifierBehavior](#))

## A\_classifier\_enumerationLiteral [Association]

### Diagrams

[DataTypes](#)

### Generalizations

[A\\_classifier\\_instanceSpecification](#)

### Owned Ends

- enumerationLiteral : [EnumerationLiteral](#) [0..\*]{redefines [A\\_classifier\\_instanceSpecification::instanceSpecification](#)} (opposite [EnumerationLiteral::classifier](#))

## A\_contract\_interfaceRealization [Association]

### Diagrams

[Interfaces](#)

### Owned Ends

- interfaceRealization : [InterfaceRealization](#) [0..\*]{subsets [A\\_supplier\\_supplierDependency::supplierDependency](#)} (opposite [InterfaceRealization::contract](#))

## A\_interfaceRealization\_implementingClassifier [Association]

### Diagrams

[Interfaces](#)

### Member Ends

- [BehavioredClassifier::interfaceRealization](#)
- [InterfaceRealization::implementingClassifier](#)

## A\_nestedClassifier\_interface [Association]

### Diagrams

[Interfaces](#)

### Owned Ends

- interface : [Interface](#) [0..1]{subsets [NamedElement::namespace](#), subsets [RedefinableElement::redefinitionContext](#)} (opposite [Interface::nestedClassifier](#))

## A\_ownedAttribute\_datatype [Association]

### Diagrams

[DataTypes](#), [Properties](#)

### Member Ends

- [DataType::ownedAttribute](#)
- [Property::datatype](#)

## A\_ownedAttribute\_interface [Association]

### Diagrams

[Interfaces](#), [Properties](#)

### Member Ends

- [Interface::ownedAttribute](#)
- [Property::interface](#)

## A\_ownedAttribute\_owningSignal [Association]

### Diagrams

[Signals](#)

### Owned Ends

- owningSignal : [Signal](#) [0..1]{subsets [NamedElement::namespace](#), subsets [A\\_attribute\\_classifier::classifier](#)} (opposite [Signal::ownedAttribute](#))

## A\_ownedBehavior\_behavoredClassifier [Association]

### Diagrams

[Interfaces](#), [Behaviors](#)

### Specializations

[A\\_classifierBehavior\\_behavoredClassifier](#)

## Owned Ends

- behavedClassifier : [BehavedClassifier](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [BehavedClassifier::ownedBehavior](#))

## A\_ownedLiteral\_enumeration [Association]

### Diagrams

[DataTypes](#)

### Member Ends

- [Enumeration::ownedLiteral](#)
- [EnumerationLiteral::enumeration](#)

## A\_ownedOperation\_datatype [Association]

### Diagrams

[DataTypes](#), [Operations](#)

### Member Ends

- [DataType::ownedOperation](#)
- [Operation::datatype](#)

## A\_ownedOperation\_interface [Association]

### Diagrams

[Interfaces](#), [Operations](#)

### Member Ends

- [Interface::ownedOperation](#)
- [Operation::interface](#)

## A\_ownedReception\_interface [Association]

### Diagrams

[Interfaces](#)

### Owned Ends

- interface : [Interface](#) [0..1]{subsets [Feature::featuringClassifier](#), subsets [NamedElement::namespace](#)} (opposite [Interface::ownedReception](#))

## A\_protocol\_interface [Association]

### Diagrams

[Interfaces](#)

### Owned Ends

- interface : [Interface](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [Interface::protocol](#))  
Specifies the namespace in which the protocol state machine is defined.

## A\_redefinedInterface\_interface [Association]

### Diagrams

[Interfaces](#)

### Owned Ends

- interface : [Interface](#) [0..\*]{subsets [A\\_redefinedClassifier classifier::classifier](#)} (opposite [Interface::redefinedInterface](#))

## A\_signal\_reception [Association]

### Diagrams

[Signals](#)

### Owned Ends

- reception : [Reception](#) [0..\*] (opposite [Reception::signal](#))

# 11 Structured Classifiers

## 11.1 Summary

StructuredClassifiers are Classifiers that may have an internal structure comprising a network of linked roles (which can themselves be instances of structured classifiers) and an external structure consisting of one or more Ports. The Ports of EncapsulatedClassifiers act as local agents of remote collaborators, allowing EncapsulatedClassifiers to differentiate between them but without being directly coupled to them. Classes, Components, Associations and Collaborations are concrete metaclasses that use these capabilities.

## 11.2 Structured Classifiers

### 11.2.1 Summary

StructuredClassifiers may contain an internal structure of connected elements each of which plays a role in the overall behavior modeled by the StructuredClassifier. It may be helpful to read this sub clause in conjunction with sub clause 11.5 - Associations.

### 11.2.2 Abstract Syntax

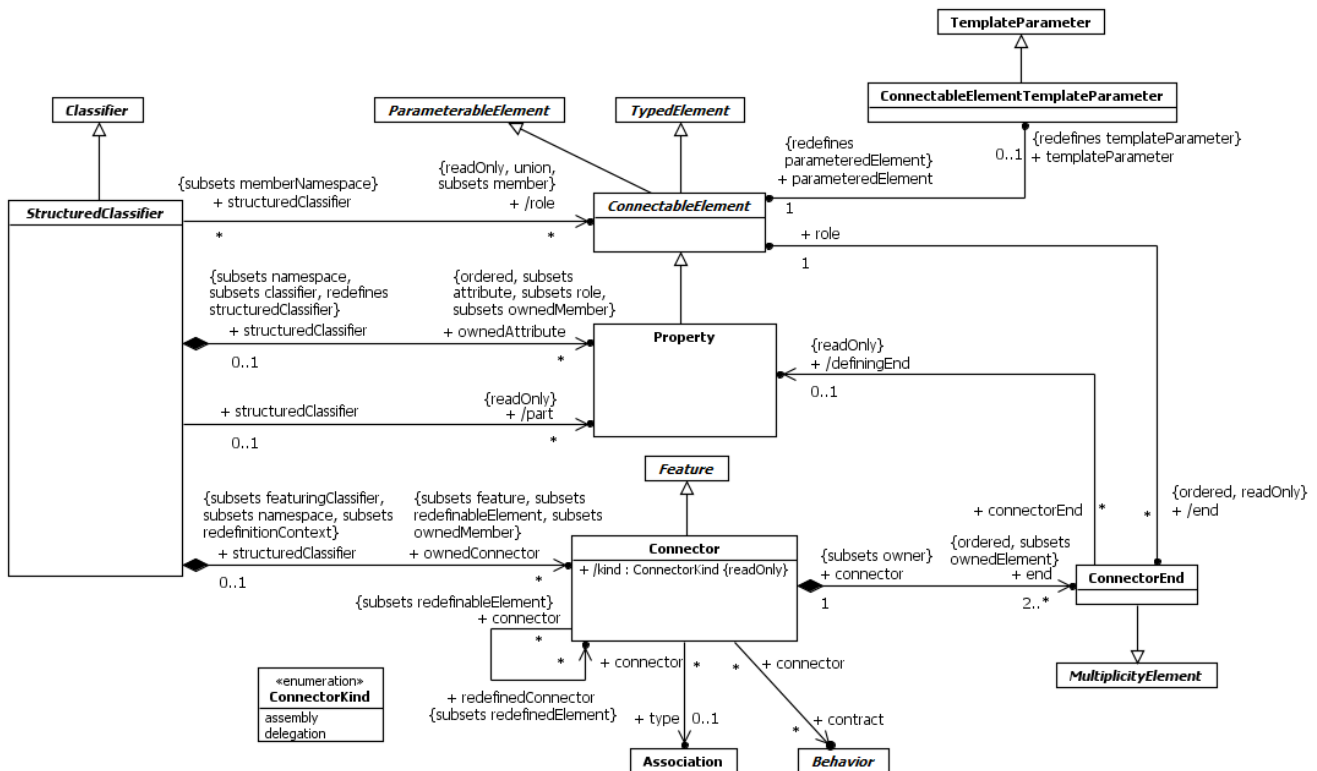


Figure 11.1 Structured Classifiers



## 11.2.3 Semantics

### Connectable Elements

ConnectableElement is an abstract metaclass. Each ConnectableElement represents a participant within the internal structure of a StructuredClassifier; these participants are called roles. Roles may be joined by Connectors, and specify configurations of linked instances contained or referenced within an instance of the containing StructuredClassifier.

The detailed semantics of ConnectableElement is given by its concrete subtypes. In general, each ConnectableElement exhibits a set of *effective required Interfaces* and a set of *effective provided Interfaces*. These sets are used to determine the connectability of ConnectableElements using Connectors, see below.

For ConnectableElements except delegating Ports (see 11.3.3) the effective required Interfaces are the required Interfaces, and the effective provided Interfaces are the provided Interfaces, derived as follows:

- The provided Interfaces comprises the union of the sets of Interfaces realized by the type of the ConnectableElement and its supertypes, or the set containing just its type if it is typed by an Interface.
- The required Interfaces comprises the union of the sets of Interfaces used by the type of the ConnectableElement and its supertypes.

A ConnectableElement may be exposed via a ConnectableElementTemplateParameter as a formal parameter for a template. The semantics and notation for this are only defined when the ConnectableElement is a Property (see the semantics and notation for Property in [9.5](#)).

### Parts and Roles

The Properties of a StructuredClassifier obey the semantics of Property specified in [9.5](#).

Property is a kind of ConnectableElement. All of the ownedAttributes of a StructuredClassifier are roles and can be connected using Connectors.

Those ownedAttributes of a StructuredClassifier that have isComposite = true (see [9.5.3](#)) are called its parts. Hence parts constitute a subset of roles.

### Connectors

A Connector specifies *links* (see 11.5 Associations) between two or more instances playing roles within a StructuredClassifier. Each link may be realized by something as simple as a pointer or by something as complex as a network connection, and may represent the possibility of instances being able to communicate because their identities are known by virtue of being passed in as parameters, held in variables or slots, or even because the communicating instances are the same instance.

In contrast to Associations, which specify links between any suitably-typed instance of the associated Classifiers, Connectors specify links between instances playing the connected roles only.

Each Connector may be attached to two or more ConnectableElements, each representing a set of instances that play a distinct role in the instantiation of the containing StructuredClassifier.

A ConnectorEnd is an endpoint of a Connector, which attaches the Connector to a ConnectableElement.

Links corresponding to Connectors may be created upon the creation of the instance of the containing StructuredClassifier. All such links are destroyed when the containing StructuredClassifier instance is destroyed.

A Connector may be typed by an Association, in which case the links specified by the Connector are instances of the typing Association.

Each feature of each of the *effective required Interfaces* of each ConnectableElement at the end of a Connector must have at least one compatible feature among the features of the *effective provided Interfaces* of ConnectableElements at the other ends. One

feature is compatible with another at least in the cases when the two features are the same or when they are both properties or operations and the second feature is a redefinition of the first. However, conforming tools may allow additional cases of compatible features beyond this.

When there are multiple connectors attached to a single ConnectableElement, the semantics are the same as a single n-ary Connector connecting the ConnectableElement to all of the ConnectableElements connected via the multiple connectors.

Connectors have a kind, whose value is *assembly* or *delegation*. The semantics of *delegation* connectors are only related to Ports and described under Port (see 11.3). All other Connectors are *assembly* connectors.

ConnectorKind is an enumeration of the following literal values:

assembly	Indicates that the Connector is an assembly Connector.
delegation	Indicates that the Connector is a delegation Connector.

Behaviors may be associated with Connectors as contracts to specify valid interaction patterns across the Connector.

### Multiplicities and topologies

The multiplicities on ConnectableElements constrain the number of objects that may be created within an instance of the containing StructuredClassifier, according to the semantics of MultiplicityElement (see [7.5.3](#)).

For a binary Connector, the ConnectorEnd's multiplicity indicates the number of instances that may be linked to each instance of the ConnectableElement on the other end. For an n-ary Connector, the multiplicity of one end constrains the number of links that may refer to a set containing one particular instance for each of the other ends.

When an instance is removed from a role of an instance of a StructuredClassifier, links that exist due to Connectors between that role and others are destroyed.

The topologies that result from matching the multiplicities of ConnectorEnds and those of ConnectableElements they interconnect cannot always be deduced from the model. Specific examples in which the topology can be determined from the multiplicities are shown in Figure 11.6 and Figure 11.7.

### 11.2.4 Notation

The internal structure of a StructuredClassifier is shown in a separate compartment with the name "internal structure." This compartment is mandatory: all tools that conform to the concrete syntax of UML must implement it. The internal structure compartment contains symbols representing the roles and connectors. The internal structure compartment appears below the attributes and operations compartments.

A part may be shown by graphical nesting of a box symbol with a solid outline representing the part within the internal structure compartment. A role that is not a composition may be shown by graphical nesting of a box symbol with a dashed outline. In either case the box may be called a *part box*, even though strictly-speaking only the compositions are parts.

The part box symbol has a name compartment, which contains a string according to the syntax defined in sub clause [9.5.4](#). Detail may also be shown within the part box indicating specific values for Properties of the part's type when instances corresponding to the Property are created.

The multiplicity for a Property may also be shown as a multiplicity mark in the top right corner of the part box.

A part box may be shown containing just a single name (without the colon) in its name string. This implies the definition of an anonymously named Class nested within the namespace of the containing Classifier. The Property has this anonymous Class as its type. Every occurrence of an anonymous Class is different from any other occurrence. A conforming tool may optionally show compartments defining attributes and operations of the anonymous Class within the part box, as though the compartments were in a corresponding Classifier rectangle representing the anonymous Class itself.

When a role is typed by an EncapsulatedClassifier (see 11.3), any Ports of the type may also be shown as small square symbols overlapping the boundary of the part box denoting the role. The name of the Port is shown near the Port; the multiplicity follows the name surrounded by square brackets. Name and multiplicity may be elided. Lollipop and socket symbols may optionally be shown to indicate the provided and required interfaces of the Port, using the same notation as for the Port’s definition (see 11.3.4).

If a role is typed by a classifier other than Class, the name compartment of the part box symbol contains the appropriate keyword (e.g., «component») above the name. For some kinds of Classifiers, optionally in the right hand corner an icon denoting the kind of Classifier can be displayed.

A Connector is drawn using similar notation to that for Association (see 11.5.4). The optional name string of the Connector obeys the following syntax:

$( [ name ] \text{ '}' <classname> ) | <name>$

where  $<name>$  is the name of the Connector, and  $<classname>$  is the name of the Association that is its type. A stereotype keyword within guillemets may be placed above or in front of the Connector name. A property string may be placed after or below the Connector name.

Adornments may be shown on the ConnectorEnd using the same notation as adornments on Association ends. If no multiplicity is shown, the multiplicity matches the multiplicity of the role the end is attached to.

If a ConnectorEnd is attached to a Port on a part or role of the internal structure and no multiplicity is shown, the multiplicity of the ConnectorEnd matches the multiplicity of the Port multiplied by the multiplicity of the role (if any).

The notational specifications in the next three paragraphs are optional: a conforming tool does not need to implement them. They are useful for scalability in complex systems.

If the parts have *simple* Ports (Ports with a single required or provided Interface), then *ball-and-socket notation* may be used to represent assembly Connectors between those Ports. Ball-and-socket notation may not be used to connect complex (i.e., non-simple) Ports or parts without Ports.

When connecting simple Ports, normal Connector notation for assembly or delegation may be shown connected to the ball or socket symbol rather than to the Port symbol itself.

When there is an n-ary Connector connecting more than two simple Ports, and two or more of the Ports provide or require the same or compatible Interfaces, a single symbol representing the Interface can be shown, and lines from the Components can be drawn to that symbol, in a “channeled ball-and-socket” notation.

An internal structure compartment may also contain symbols representing CollaborationUses, following the notation described in 11.7.4.

### 11.2.5 Examples



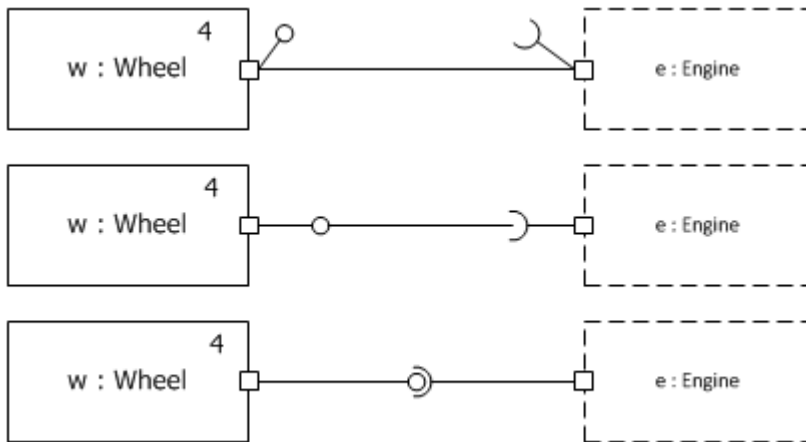
Figure 11.2 Parts and roles

Figure 11.2 shows examples of part boxes. On the left, the part box denotes that the containing instance will own four instances of the *Wheel* class by composition. The multiplicity is shown in the corner of the part box. The part box on the right is not composite, and denotes that the containing instance will reference one or two instances of the *Engine* class.



**Figure 11.3 Parts and roles with Ports**

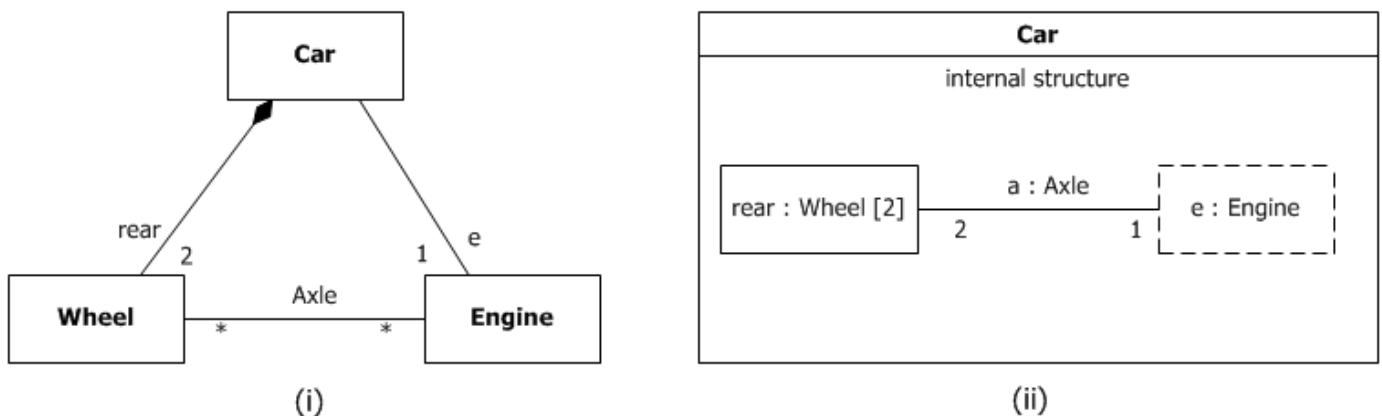
Figure 11.3 shows examples of part boxes for properties typed by EncapsulatedClassifiers with Ports, in this case simple Ports. The notation for more complex Ports can also be used.



**Figure 11.4 Alternative notations for connecting parts and roles with Ports**

Figure 11.4 shows three alternative notations for connecting simple Ports on the parts and roles within a StructuredClassifier. In the top example, the connector is joined to the Port symbols themselves. This is the only mandatory notation for connecting Ports in an internal structure. The lollipops and sockets indicate the provided and required interfaces of the Ports; their appearance is optional.

In the second example, the connector line is attached to the ball and socket symbols; in the third example, ball-and-socket notation is used. These notations correspond to the same model as the top example.

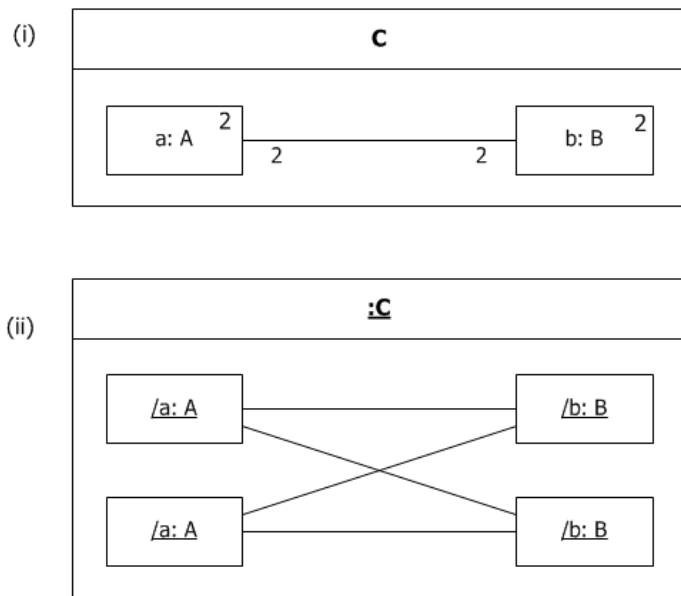


**Figure 11.5 Associations compared with Connectors**

Figure 11.5 shows two possible views of the *Car* Class. In subfigure (i), *Car* is shown as having a composition Association with role name *rear* to a class *Wheel* and an Association with role name *e* to a class *Engine*. In subfigure (ii), the same is specified. However, in addition, in subfigure (ii) it is specified that *rear* and *e* belong to the internal structure of the class *Car*. This allows

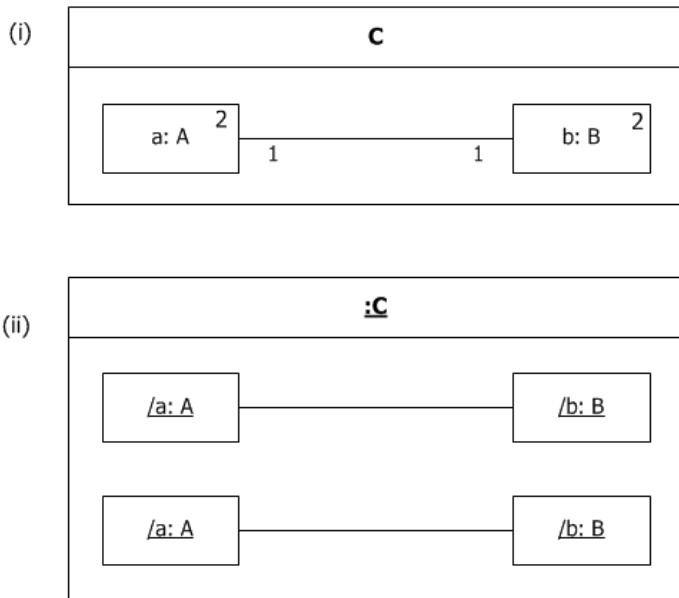
specification of detail that holds only for instances of the *Wheel* and *Engine* classes within the context of the class *Car*, but which will not hold for wheels and engines in general. For example, subfigure (i) specifies that any instance of class *Engine* can be linked to an arbitrary number of instances of class *Wheel*. Subfigure (ii), however, specifies that within the context of class *Car*, the instance playing the role of *e* may only be connected to two instances playing the role of *rear*. In addition, the instances playing the *e* and *rear* roles may only be linked if they are roles of the same instance of class *Car*. In other words, subfigure (ii) asserts additional constraints on the instances of the classes *Wheel* and *Engine*, when they are playing the respective roles within an instance of class *Car*. These constraints are not true for instances of *Wheel* and *Engine* in general. Other wheels and engines may be arbitrarily linked as specified in subfigure (i).

For each instance playing a role in an internal structure, there will initially be as many links as indicated by the lower multiplicity of the opposite ends of Connectors attached to that role. If the multiplicities of the ends match the multiplicities of the roles they are attached to as defined in Figure 11.6 (i), the initial configuration that will be created when an instance of the containing StructuredClassifier is created consists of the set of instances corresponding to the roles (as specified by the multiplicities on the roles) fully connected by links; see the resultant instance shown in Figure 11.6 (ii).



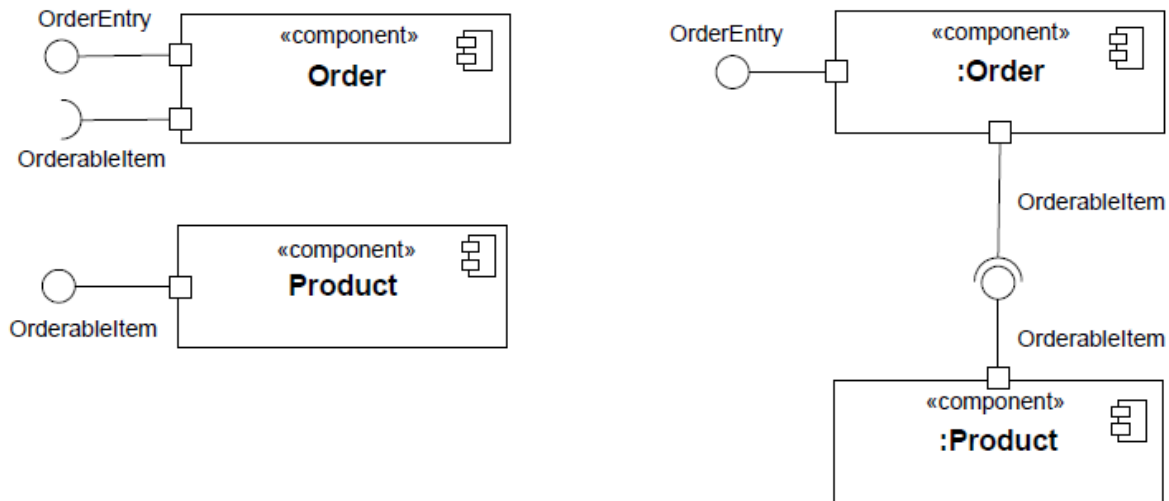
**Figure 11.6 "Star" Connector pattern**

Links will be created for each instance playing the connected roles according to their ordering until the minimum ConnectorEnd multiplicity is reached for both ends of the Connector; see the resultant instance in Figure 11.7 (ii). In this example, only two links are created.



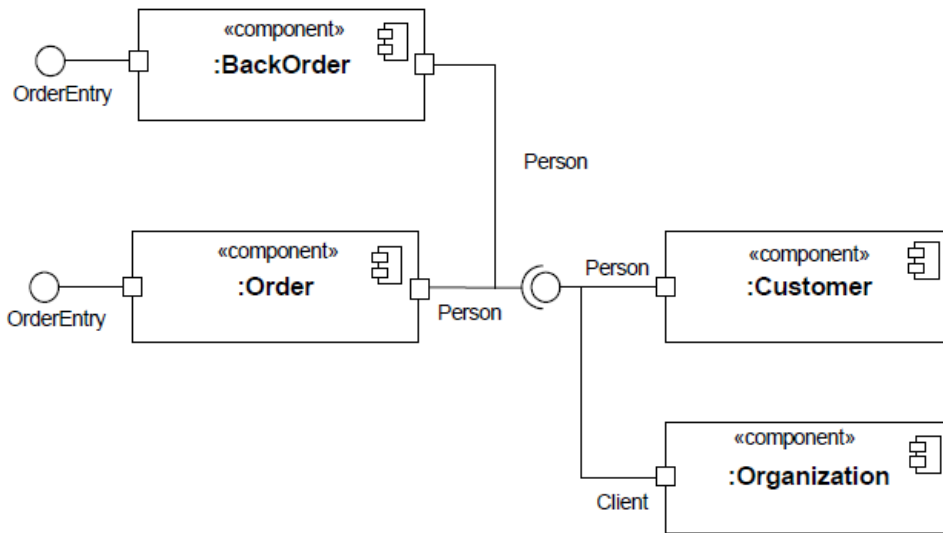
**Figure 11.7 "Array" Connector pattern**

Figure 11.8 shows example notation for parts typed by Components with simple Ports (Ports with only one interface), and the optional ball-and-socket notation to represent an assembly Connector between compatible Ports. The Component definitions are on the left and the corresponding parts on the right.



**Figure 11.8 An assembly Connector maps a simple Port of a Component to a matching simple Port of another Component.**

Figure 11.9 shows “channeled ball-and-socket notation” for a 4-ary Connector. The two simple Ports that require Person have been channeled into a single socket, and the two simple Ports that provide Person (either directly or indirectly) have been channeled into a single ball.



Note: Client interface is a subtype of Person interface

Figure 11.9 An n-ary Connector that assembles four simple Ports using channeled ball-and-socket notation.

## 11.3 Encapsulated Classifiers

### 11.3.1 Summary

EncapsulatedClassifier extends StructuredClassifier with the ability to own Ports, a mechanism for isolating an EncapsulatedClassifier from its environment.

### 11.3.2 Abstract Syntax

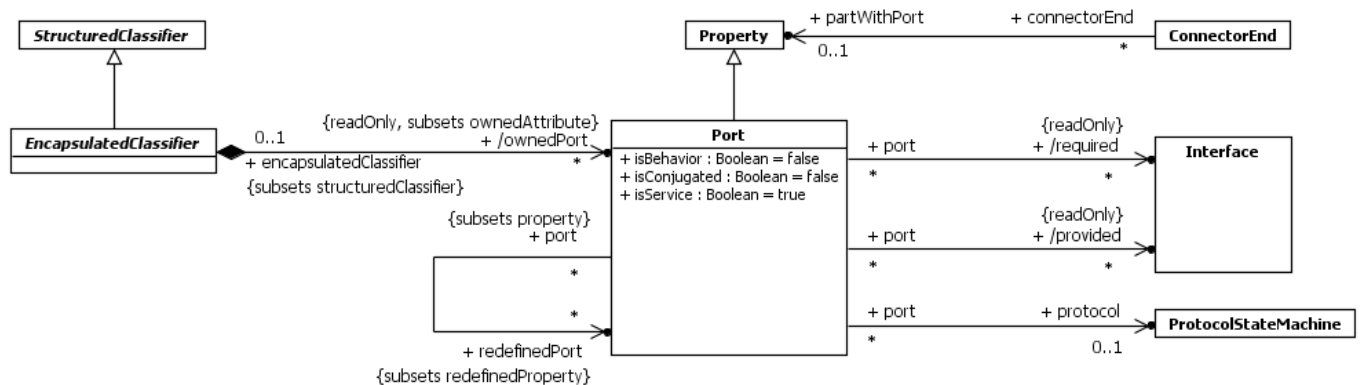


Figure 11.10 Encapsulated Classifiers

### 11.3.3 Semantics

#### Ports

Ports represent interaction points through which an EncapsulatedClassifier communicates with its environment. Multiple Ports can be defined for an EncapsulatedClassifier, enabling different communications to be distinguished based on the Port through which they occur. By decoupling the internals of the EncapsulatedClassifier from its environment, Ports allow an EncapsulatedClassifier to be defined independently of its environment, making it reusable in any environment that conforms to the constraints imposed by its Ports.

A Port is a Property of an EncapsulatedClassifier that specifies a distinct interaction point between that EncapsulatedClassifier and its environment or between the Behavior of the EncapsulatedClassifier and its internal roles. Ports are connected by Connectors through which requests can be made to invoke the BehavioralFeatures of an EncapsulatedClassifier. A Port may specify the services an EncapsulatedClassifier provides (offers) to its environment as well as the services that an EncapsulatedClassifier expects (requires) of its environment.

The property `isService`, when true, indicates that this Port is used to provide the published functionality of an EncapsulatedClassifier. If false, this Port is used to implement the EncapsulatedClassifier but is not part of the essential externally-visible functionality of the EncapsulatedClassifier and can, therefore, be altered or deleted along with the internal implementation of the EncapsulatedClassifier and other properties that are considered part of its implementation.

The phrase *Port on Part* or more generally *Port on Property* signifies the situation where a Property playing a role in a StructuredClassifier is typed by an EncapsulatedClassifier that has Ports. A Connector within the containing StructuredClassifier may be connected to one of these Ports. In such a case, the property `partWithPort` of the applicable ConnectorEnd references the actual Property being connected: in general, there might be many Properties in the structure typed by the same EncapsulatedClassifier, and `partWithPort` is used to signify the right one.

The Interfaces associated with a Port specify the nature of the interactions that may occur over it. The required Interfaces of a Port characterize the requests that may be made from the EncapsulatedClassifier to its environment through this Port. Instances of this EncapsulatedClassifier expect that the Features owned by its required Interfaces will be offered by one or more instances in its environment. The provided Interfaces of a Port characterize requests to the EncapsulatedClassifier that its environment may make through this Port. The owning EncapsulatedClassifier must offer the Features owned by the provided Interfaces.

As a kind of Property, a Port has a type. The provided and required interfaces of the Port are related to its type mediated by the value of `isConjugated` as follows:

- If `isConjugated` is false, `provided` is derived as the union of the sets of Interfaces realized by the type of the Port and its supertypes, or directly from the type of the Port if the Port is typed by an Interface; `required` is derived as the union of the sets of Interfaces used by the type of the Port and its supertypes.
- If `isConjugated` is true, `provided` is derived as the union of the sets of Interfaces used by the type of the Port and its supertypes; `required` is derived as the union of the sets of Interfaces realized by the type of the Port and its supertypes, or directly from the type of the Port if the Port is typed by an Interface.

The Interfaces do not necessarily establish the exact sequences of interactions across the Port. A Port's protocol may reference a ProtocolStateMachine that describes valid sequences of Operation and Reception invocations that may occur at this Port.

When an instance of an EncapsulatedClassifier is created, instances corresponding to each of its Ports are created and held in the slots specified by each Port, in accordance with its type and multiplicity. These instances are referred to as "interaction points" and provide unique references. It is, therefore, possible for an EncapsulatedClassifier instance to differentiate between requests for the invocation of a BehavioralFeature targeted at its different Ports. Similarly, it is possible to direct such requests at a Port, and the requests will be routed as specified by the links corresponding to Connectors attached to this Port.

**NOTE.** In the following, "requests arriving at a Port" shall mean "request occurrences arriving at the interaction point of this instance corresponding to this Port."



If there is a Connector attached to only one side of a Port, any requests arriving at this Port will be lost.

A Port has the ability, by setting the property `isBehavior` to true, to specify that any requests arriving at this Port are handled by the Behavior of the instance of the owning EncapsulatedClassifier, rather than being forwarded to any contained instances, if any. Such a Port is called a *behavior Port*. If there is no Behavior defined for this EncapsulatedClassifier, any communication arriving at a behavior Port is lost.

A *delegation* Connector is a Connector that links a Port to a role within the owning EncapsulatedClassifier. It represents the forwarding of requests (Operation invocations and Signals). A request that arrives at a Port that has a delegation Connector to one or more Properties or Ports on Properties will be passed on to those targets for handling.

Delegation Connectors can be used to model the hierarchical decomposition of behavior, where services provided by an EncapsulatedClassifier may ultimately be realized by one that is nested multiple levels deep within it.

As a ConnectableElement, the *effective provided Interfaces* (see 11.2.3) of a Port are its provided interfaces, and the *effective required Interfaces* are its required Interfaces. However, for a *delegating Port*, i.e., a Port that is at an end of a delegation Connector and is not on a role and that is not a behavior Port, the *effective provided Interfaces* are its *required* interfaces and its *effective required Interfaces* are its *provided* interfaces. Consequently a delegating Port behaves, for connection, as though it had an internal “face” that is the conjugate of its external “face.”

If several Connectors are attached on one side of a Port, then any request arriving at this Port on a link derived from a Connector on the other side of the Port will be forwarded on links corresponding to these Connectors. It is not defined whether these requests will be forwarded on all links, or on only one of those links.

### 11.3.4 Notation

A Port of an EncapsulatedClassifier is shown as a small square symbol. The name of the Port is placed near the square symbol. The Port symbol may be placed either overlapping the boundary of the rectangle symbol denoting that EncapsulatedClassifier or it may be shown inside the rectangle symbol. When the Port is connected to elements visually contained in a compartment of the EncapsulatedClassifier, such as parts or roles in the internal structure compartment, the Port symbol must be placed within or overlapping the boundary of that compartment.

The type of a Port may be shown following the Port name, separated by colon (“:”). When `isConjugated` is true for the Port, the type of the Port is shown with a tilde “~” prepended. A provided Interface may be shown using the *lollipop* notation (see Interface – 10.4) attached to the Port. A required Interface may be shown by the *socket* notation attached to the Port.

A behavior Port is indicated by a Port being connected through a line to a small state symbol drawn inside the symbol representing the containing EncapsulatedClassifier. The small state symbol indicates the Behavior of the containing EncapsulatedClassifier.

The name of a Port may be suppressed. Every depiction of an unnamed Port denotes a different Port from any other Port.

If there are multiple Interfaces associated with a Port, these Interfaces may be listed on the one Interface lollipop, separated by commas.

In the case of a Dependency wired from a simple Port with a required Interface to a simple Port to a provided Interface it is a notational option to show the dependency arrow joining the socket to the lollipop.

### 11.3.5 Examples

Figure 11.11 illustrates the notation for Ports. At the top of the figure is the definition of a class *PowerTrain*, together with an interface *IPowerTrain* that it realizes, and an interface *IFeedback* that it uses.

On the lower left figure, *p* is a Port on the Engine Class which is typed by *PowerTrain*. As a consequence, the provided Interface of Port *p* is *IPowerTrain* and the required Interface is *IFeedback*. The multiplicity of *p* is 1, and `isConjugated` is *false*. On the right

figure, *e* is a Port of the Class *Wheel*, which also has the type *PowerTrain* and *isConjugated* set to *true*, which results in the reversal of the provided and required Interfaces.

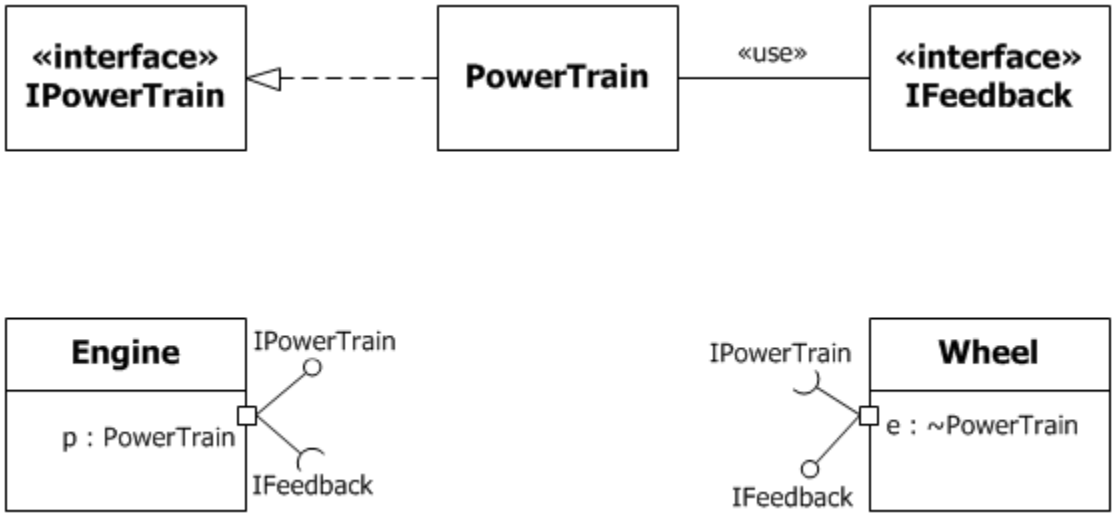


Figure 11.11 Port notation

Figure 11.12 illustrates a *behavior port* *p*, as indicated by its connection to the small state symbol representing the Behavior of the *Engine* Class. Its type is *PowerTrain*, as in the earlier example.

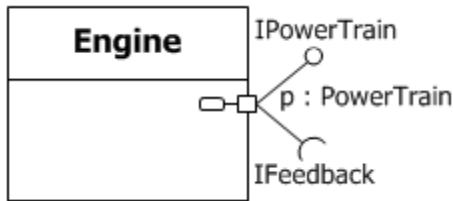


Figure 11.12 Behavior Port notation

Figure 11.13 below shows a Port *OnlineServices* on the *OrderProcess* Class with two provided Interfaces, *OrderEntry* and *Tracking* listed on the same interface lollipop, as well as a required Interface *Payment*.

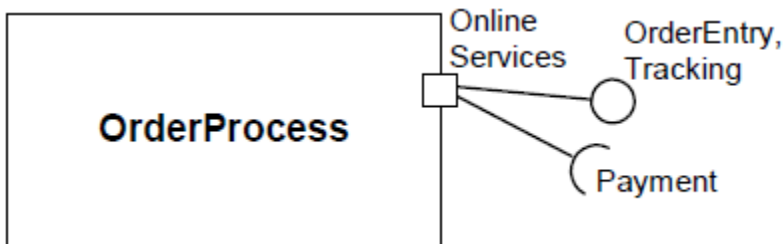
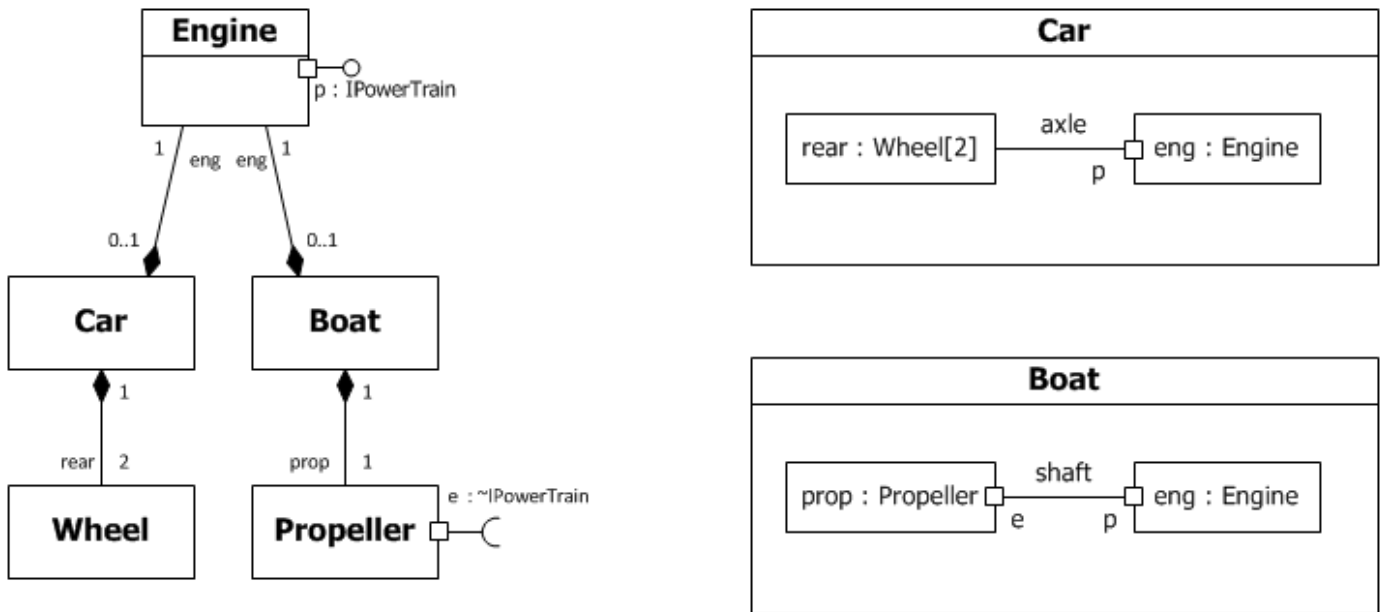


Figure 11.13 Port notation showing multiple provided Interfaces



**Figure 11.14** Port examples

Figure 11.14 shows a Class *Engine* with a Port *p* typed by its provided Interface *IPowerTrain*. This Interface specifies the services that the Engine offers at this Port (i.e., the Operations and Receptions that are accessible by communication arriving at this Port).

Two uses of the *Engine* Class are depicted: Both a Boat and a Car contain a part that is an Engine. The *Car* Class connects Port *p* of the Engine to a pair of Wheels via the *axle*. The *Boat* Class connects Port *p* of the engine to a Propeller via the *shaft*. As long as the interaction between the *Engine* and the part linked to its Port *p* obeys the constraints specified by its Interface, the Engine will function as specified, whether it is in a Car or a Boat. This example also shows that Connectors need not necessarily attach to parts via Ports (as shown in the *Car* Class).

Because the Ports are simple, the depiction of the connector within Boat could have been shown using any of the notational options shown in Figure 11.4.

## 11.4 Classes

### 11.4.1 Summary

Class is the concrete realization of EncapsulatedClassifier and BehavioredClassifier. 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.

### 11.4.2 Abstract Syntax

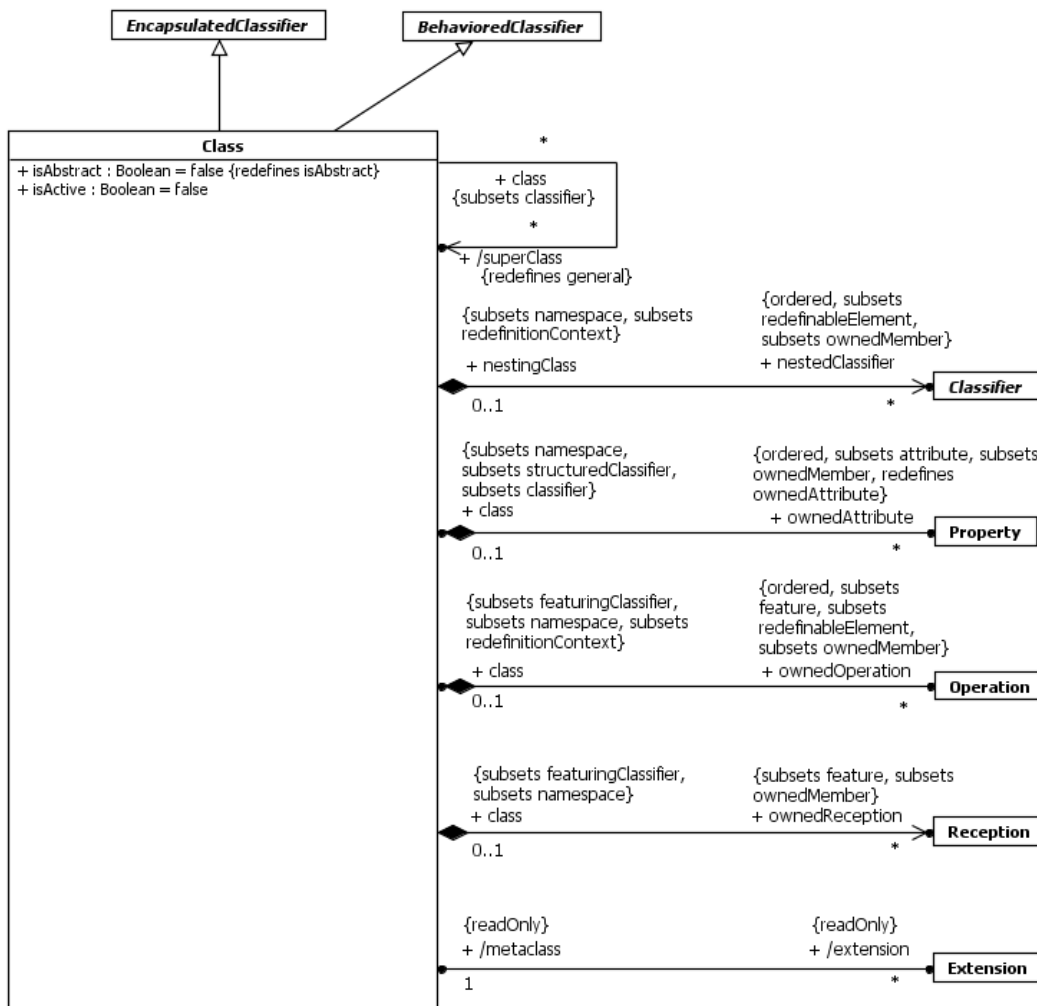


Figure 11.15 Classes

### 11.4.3 Semantics

#### Classes

Class is a kind of EncapsulatedClassifier whose Features are Properties, Operations, Receptions, Ports and Connectors. Attributes of a Class are Properties that are owned by the Class. Some of these attributes may represent the ends of binary Associations.

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 ValueSpecification 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 values for the parameters of the Operation, according to the semantics specified in [9.6.3](#).

A Class cannot access private Features of another Class, or protected Features on another Class that is not its ancestor.

A Class acts as the namespace for various kinds of Classifiers defined within its scope, including Classes. Nested Classifiers are members of the namespace of the containing Class. Classifier nesting is used for reasons of information hiding. Nested Classifiers are used like any other Classifier in the containing Class.

A Class may be designated by setting `isActive` to true as active (i.e., each of its instances is an active object). When `isActive` is false the Class is passive (i.e., each of its instances executes within the context of some other object).

An active object is an object that, as a direct consequence of its creation, commences to execute its classifierBehavior, and does not cease until either the complete Behavior is executed or the object is terminated by some external object. (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 is determined solely by the Behavior of the active object and not by the invoking object. If the classifierBehavior of an active object completes, the object is terminated.

A Class’s Receptions specify which Signals the instances of this Class handle.

An InstanceSpecification may be used to specify the initial value to be created for a Class.

All instances corresponding to parts and ports of a Class are destroyed recursively, when an instance of that Class is deleted.

A Class may act as a metaclass in the definition of Profiles and metamodels. See Profiles in [12.3](#).

#### 11.4.4 Notation

A Class is shown using the Classifier symbol. As Class is the most widely used Classifier, the keyword “Class” need not be shown in guillemets above the name.

A Class has four mandatory compartments: attributes, operations, receptions (see [9.2.4](#)) and internal structure (see 11.2.4). A Class may also have optional compartments as described for Classifiers in general (see [9.2.4](#)).

The operations compartment of a Class contains notation for its ownedOperations using the notation specified in [9.6.4](#). The receptions compartment contains ownedReceptions using the notation specified in [10.3.4](#).

A usage dependency may relate an InstanceSpecification to a constructor for a Class, describing the single value returned by the constructor Operation. The Operation is the client, the created instance the supplier. The InstanceSpecification may reference parameters declared by the Operation. A constructor is an Operation having a single return result parameter of the type of the owning Class, and marked with the standard stereotype «Create». The InstanceSpecification that is the supplier of the usage dependency represents the default value of the single return result parameter of a constructor Operation.

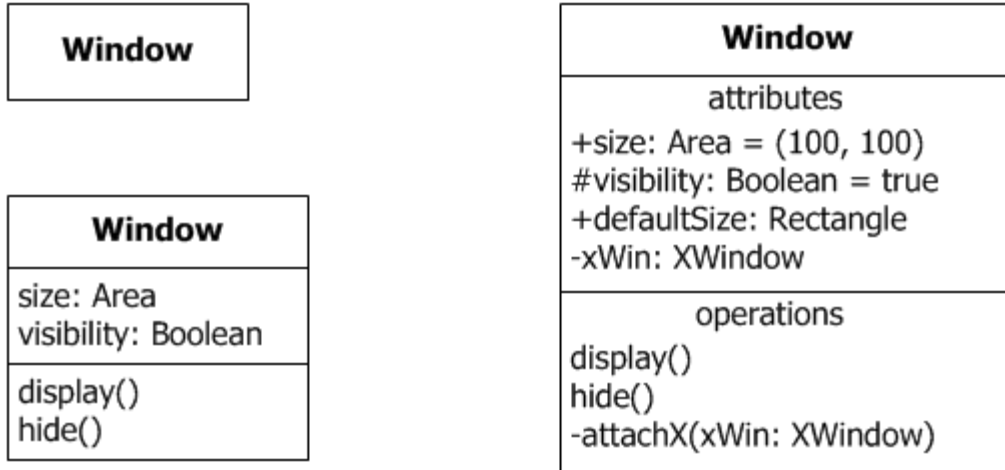
A Class with the Property `isActive = true` can be shown by a Class box with an additional vertical bar on either side.

A Class that represents a metaclass may be extended by the optional stereotype «Metaclass» (see StandardProfile in clause [22](#)) shown above or before its name.

### 11.4.5 Examples

Figure 11.16 shows three ways of displaying the Class Window, according to the options set out for Classifier notation in [9.2.4](#). The top left symbol shows all compartments suppressed. The lower left symbol shows the attributes and operations compartments, each listing the features but suppressing details such as default values, parameters, and visibility markings. The right symbol shows these details, as well as the optional compartment headers.

**NOTE.** The `display()` and `hide()` operations have no visibility specified.



**Figure 11.16** Class notation variants

Figure 11.17 shows the visibility grouping option (see [9.2.4](#)) applied to the attributes and operations compartments in the Class Window.

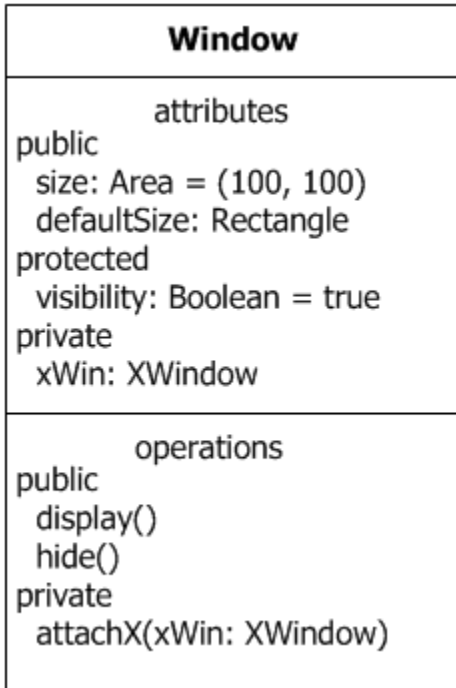


Figure 11.17 Class notation: attributes and Operations grouped according to visibility

Figure 11.18 shows an example of an active class.



Figure 11.18 Active Class

The following example uses two Classes, *Car* and *Wheel*. The *Car* Class has four parts, all of type *Wheel*, representing the four wheels of the car. The front wheels and the rear wheels are linked via Connectors representing the front and rear axle, respectively. Figure 11.19 specifies that whenever an instance of the *Car* Class is created, four instances of the *Wheel* Class are created and held by composition within the car instance. In addition, one link each is created between the front wheel instances and the rear wheel instances.

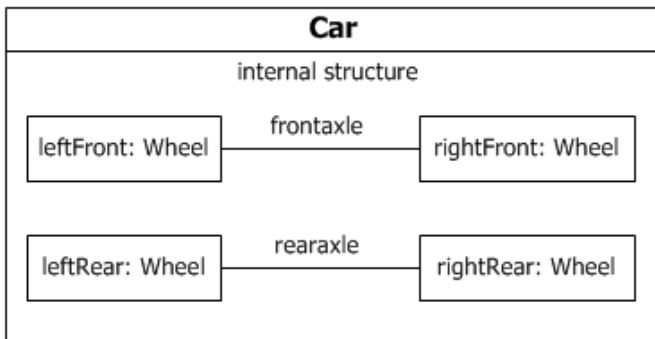


Figure 11.19 Connectors and Parts

Figure 11.20 specifies an equivalent system, but relies on multiplicities to show the replication of the wheel and axle arrangement. This diagram specifies that there will be exactly two instances of the left wheel and exactly two instances of the right wheel, with each matching instance connected by a link deriving from the Connector representing the axle.

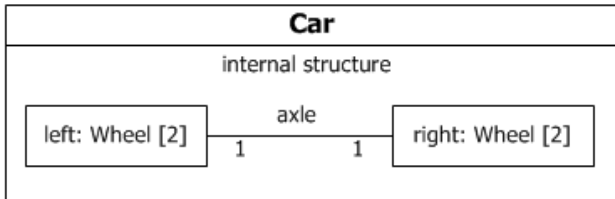


Figure 11.20 Connectors and Parts in a structure diagram using multiplicities

Figure 11.21 shows an InstanceSpecification (see 9.8) for an instance of the Car Class (as specified in Figure 11.19). It describes the internal structure of the Car that it creates and how the four contained instances of Wheel will be initialized. In this case, every instance of Wheel will have the predefined size and use the brand of tire as specified. The left wheel instances are given names, and all wheel instances are shown as playing the respective roles. The types of the wheel instances have been suppressed.

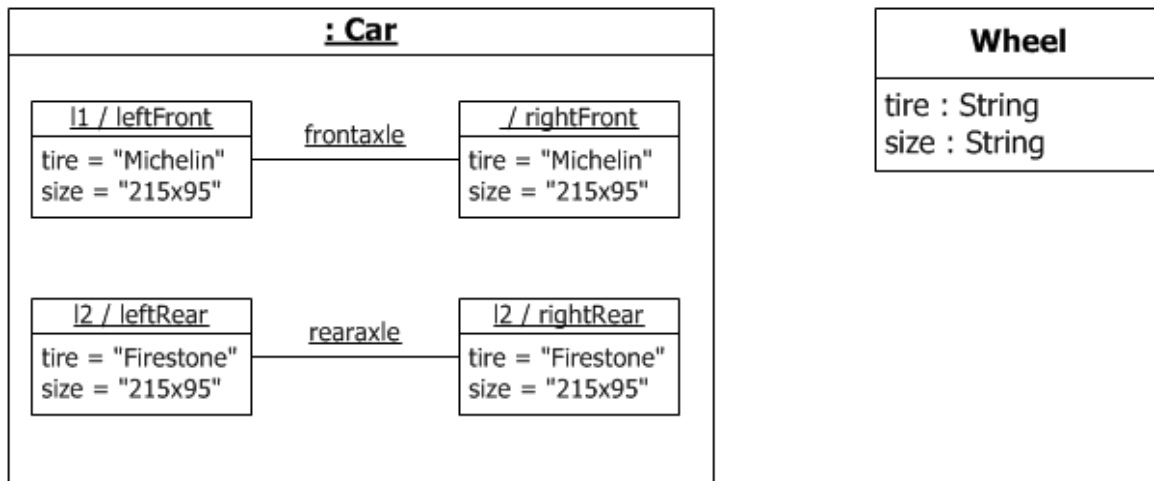


Figure 11.21 An Instance of the Car Class

Figure 11.22 shows a constructor for the Window class, illustrating how the standard stereotype «Create» is applied to the make Operation to mark it as a constructor.

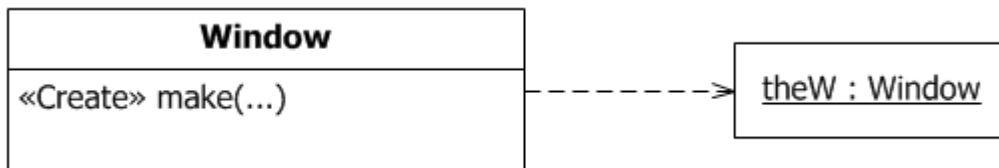


Figure 11.22 InstanceSpecification describes the return value of an Operation

Figure 11.23 shows a constructor for the Car Class. This constructor takes a parameter brand of type String. It describes the internal structure of the Car that it creates and how the four contained instances of Wheel will be initialized. In this case, every instance of Wheel will have the predefined size and use the brand of tire passed as parameter. The left wheel instances are given names, and all wheel instances are shown as playing the parts. The types of the wheel instances have been suppressed.



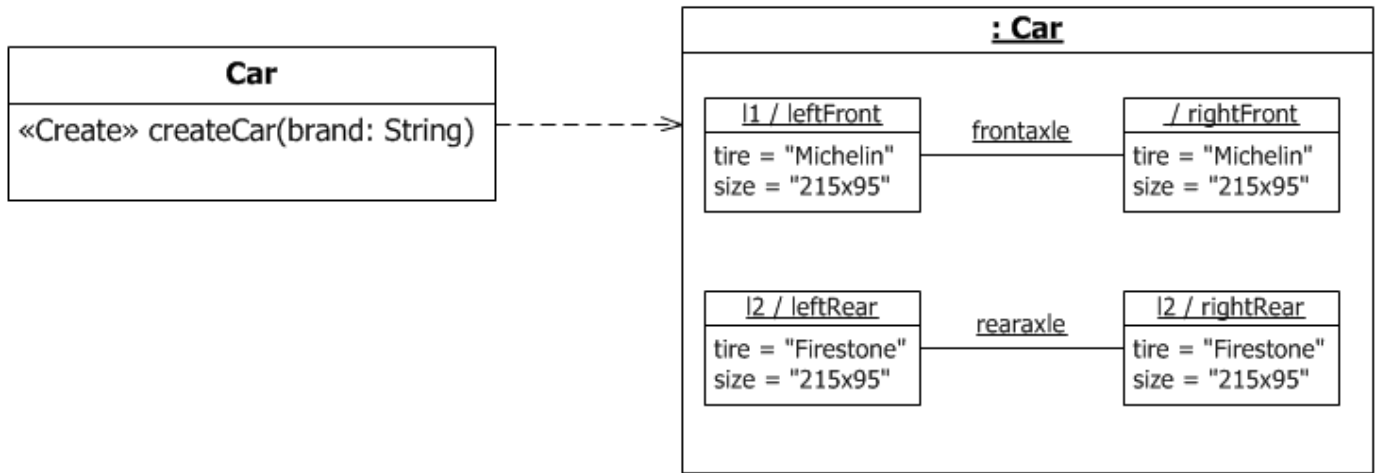


Figure 11.23 A constructor for the Car Class

In Figure 11.24, it is made explicit that the extended Class *Interface* is in fact a metaclass (from a reference metamodel).

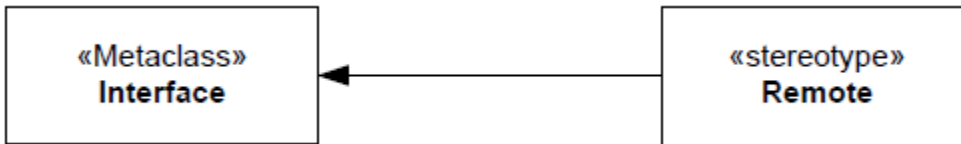


Figure 11.24 Showing that the extended Class is a metaclass

## 11.5 Associations

### 11.5.1 Summary

An Association classifies a set of tuples representing links between typed instances. An AssociationClass is both an Association and a Class.

### 11.5.2 Abstract Syntax

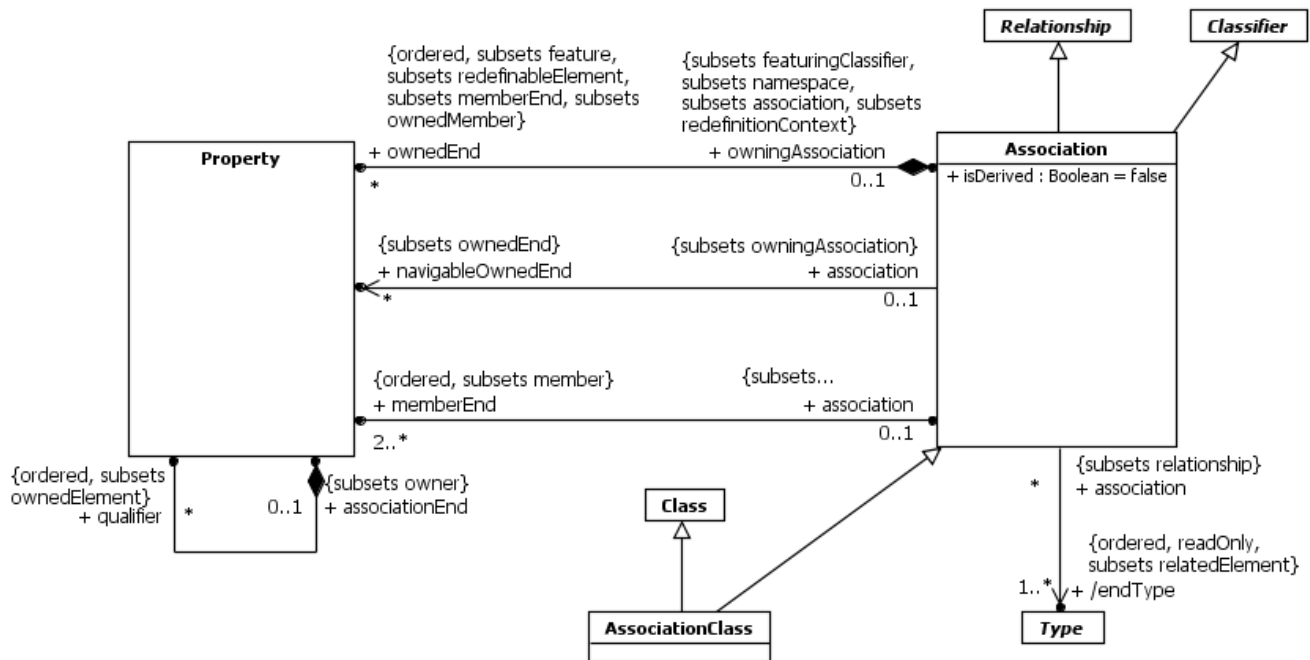


Figure 11.25 Associations

### 11.5.3 Semantics

#### Associations

An Association specifies a semantic relationship that can occur between typed instances. It has at least two memberEnds represented by Properties, each of which has the type of the end. More than one end of the Association may have the same type.

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

Not all links need to be classified by an Association.

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 memberEnds, 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 other end constrains the size of this collection. If the other end is marked as `isOrdered`, this collection will be ordered. If the other end is marked as `isUnique`, this collection is a set; otherwise, it allows duplicate elements.

*Subsetting* represents the familiar set-theoretic concept. It applies to the collections represented by Association ends, not to the Association itself. It means that the subsetting Association end represents a collection that is either equal to or a proper subset of the collection that it is subsetting. Subsetting is a relationship in the domain of extensional semantics.

*Specialization* is, in contrast to subsetting, a relationship in the domain of intentional semantics, which is to say it characterizes the criteria whereby membership in the collection is defined, not by the membership. In the case of Associations, specialization means that a link classified by the specializing association is also classified by the specialized association. Semantically this implies that the collections representing the ends of the specializing association are subsets of the corresponding collections representing the ends of the specialized association; this fact of subsetting may or may not be explicitly declared in a model.

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

A binary Association may represent a composite aggregation (i.e., a whole/part relationship). Composition is represented by the `isComposite` attribute on the part end of the Association being set to true. See the semantics of composition in [9.5.3](#).

An end Property of an Association that is owned by an end Class or that is a `navigableOwnedEnd` of the Association indicates that the Association is navigable from the opposite ends; otherwise, the Association is not navigable from the opposite ends. Navigability means that instances participating in links at runtime (instances of an Association) can be accessed efficiently from instances at the other ends of the Association. The precise mechanism by which such efficient access is achieved is implementation specific. If an end is not navigable, access from the other ends may or may not be possible, and if it is, it might not be efficient.

**NOTE.** Tools operating on UML models are not prevented from navigating Associations from non-navigable ends.

The existence of an association may be derived from other information in the model. The logical relationship between the derivation of an Association and the derivation of its ends is model-specific.

## Association Classes

An AssociationClass is a declaration of an Association that has a set of Features of its own. An AssociationClass is both an Association and a Class, and preserves the static and dynamic semantics of both. An AssociationClass describes a set of objects that each share the same specifications of Features, Constraints, and semantics entailed by the AssociationClass as a kind of Class, and correspond to a unique link instantiating the AssociationClass as a kind of Association.

Both Association and Class are Classifiers and hence have a set of common properties, like being able to have Features, having a name, etc. These properties are multiply inherited from the same construct (Classifier), and are not duplicated. Therefore, an AssociationClass has only one name, and has the set of Features that are defined for Classes and Associations. The constraints defined for Class and Association also are applicable for AssociationClass, which implies for example that the attributes of the AssociationClass, the `memberEnds` of the AssociationClass, and the opposite ends of Associations connected to the AssociationClass must all have distinct names. Moreover, the specialization and refinement rules defined for Class and Association are also applicable to AssociationClass. Redefinition is applicable to an AssociationClass nested in the context of a Classifier just as it is applicable to a nested Class.

An AssociationClass inherits the composite Properties `Class::ownedAttribute` and `Association::ownedEnd`. Values of `ownedAttribute` are Properties that are attributes of the Class, not ends of the AssociationClass owned through `Association::ownedEnd`. Values of `Association::ownedEnd` are the ends of the Association owned by the AssociationClass, not attributes of the AssociationClass. As Association ends, they can be used for navigation between end objects, as in all Associations, depending on whether they are navigable (see Navigability in the semantics of Association).

An instance of an AssociationClass has the characteristics both of a link representing an instantiation of the AssociationClass as a kind of Association, and of an object representing an instantiation of the AssociationClass as a kind of Class.

**NOTE.** When one or more ends of the AssociationClass have `isUnique=false`, it is possible to have several instances associating the same set of instances of the end Classes.

An AssociationClass cannot be a generalization of an Association or a Class.

## 11.5.4 Notation

Any Association may be drawn as a diamond (larger than a terminator on a line) with a solid line for each Association memberEnd 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 Association 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 11.27). This notation is for documentation purposes only and has no general semantic interpretation. It is used to capture some application-specific detail of the relationship between the associated Classifiers.

Generalizations between Associations can be shown using a generalization arrow between the Association symbols.

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

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

- A multiplicity
- A *<prop-modifier>* enclosed in curly braces, where *<prop-modifier>* is defined in Property (see [9.5.4](#)).
- A *<visibility>* symbol (see [9.5.4](#)).

**NOTE.** By default an Association end represents a set, i.e., `isOrdered = false` and `isUnique = true`.

An open 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.

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.

A binary Association may have one end with `aggregation = AggregationKind::shared` or `aggregation = AggregationKind::composite`. When one end has `aggregation = AggregationKind::shared` a hollow diamond is added as a terminal

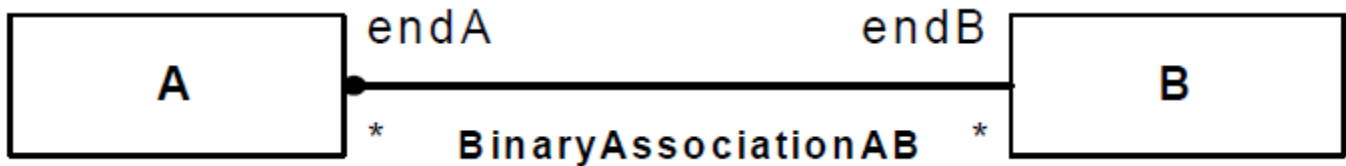
adornment at the end of the Association line opposite the end marked with aggregation = AggregationKind::shared. The diamond shall be noticeably smaller than the diamond notation for Associations. An Association with aggregation = AggregationKind::composite likewise has a diamond at the corresponding end, but differs in having the diamond filled in.

Ownership of Association ends by an associated Classifier may be indicated graphically by a small filled circle, which for brevity we will term a *dot*. The dot is to be drawn integral to the graphic path of the line, at the point where it meets the Classifier, inserted between the end of the line and the side of the node representing the Classifier. The diameter of the dot shall not exceed half the height of the aggregation diamond, and shall be larger than the width of the line. This avoids visual confusion with the filled diamond notation while ensuring that it can be distinguished from the line. The dot shows that the model includes a Property of the type represented by the Classifier touched by the dot. This Property is owned by the Classifier at the other end. In such a case it is normal to suppress the Property from the attributes compartment of the owning Classifier.

The dot may be used in combination with the other graphic line-path notations for Properties of Associations and Association ends. These include aggregation type and navigability.

Explicit end-ownership notation is not mandatory, i.e., a conforming tool may not support it. Where the dot notation is used, it shall be applied consistently throughout each diagram, so that the absence of the dot signifies ownership by the Association. Stated otherwise, when applying this notation to a binary Association in a user model, the dot will be omitted only for ends which are not owned by a Classifier. In this way, in contexts where the notation is used, the absence of the dot on certain ends does not leave the ownership of those ends ambiguous.

The dot is illustrated in Figure 11.26, at the maximum allowed size. The diagram shows endA to be owned by Classifier B, and because the notation must be applied consistently throughout the diagram, this diagram also shows unambiguously that endB is owned by BinaryAssociationAB.



**Figure 11.26** Graphic notation indicating exactly one Association end owned by the Association

Navigability notation was often used in the past according to an informal convention, whereby non-navigable ends were assumed to be owned by the Association whereas navigable ends were assumed to be owned by the Classifier at the opposite end. This convention is now deprecated. Aggregation type, navigability, and end ownership are separate concepts, each with their own explicit notation. Association ends owned by classes are always navigable, while those owned by associations may be navigable or not.

An AssociationClass is shown as a Class symbol attached to the Association path by a dashed line. The Association path and the AssociationClass symbol represent the same underlying model element, which has a single name. The name may be placed on the path, in the Class symbol, or on both, but they must be the same name. Association end names appear in the same position as regular Associations, not in the attribute compartment of the AssociationClass.

Logically, the AssociationClass and the Association are the same semantic entity; however, they are graphically distinct. The AssociationClass symbol can be dragged away from the line, but the dashed line must remain attached to both the path and the Class symbol.

When two association lines cross, a conforming tool may provide the option to show a small semicircular jog to indicate that the lines do not intersect (as in electrical circuit diagrams).

In practice, it is often convenient to suppress some of the arrows and crosses that signify navigability of association ends. A conforming tool may provide various options for showing navigation arrows and crosses. As with dot notation, these options apply at the level of complete diagrams.

- Show all arrows and crosses. Navigation and its absence are made completely explicit.

- Suppress all arrows and crosses. No inference can be drawn about navigation.
- Suppress all crosses. 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, a conforming tool may draw them 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.

### 11.5.5 Examples

Figure 11.27 shows a binary Association from *Player* to *Year* named *PlayedInYear*.

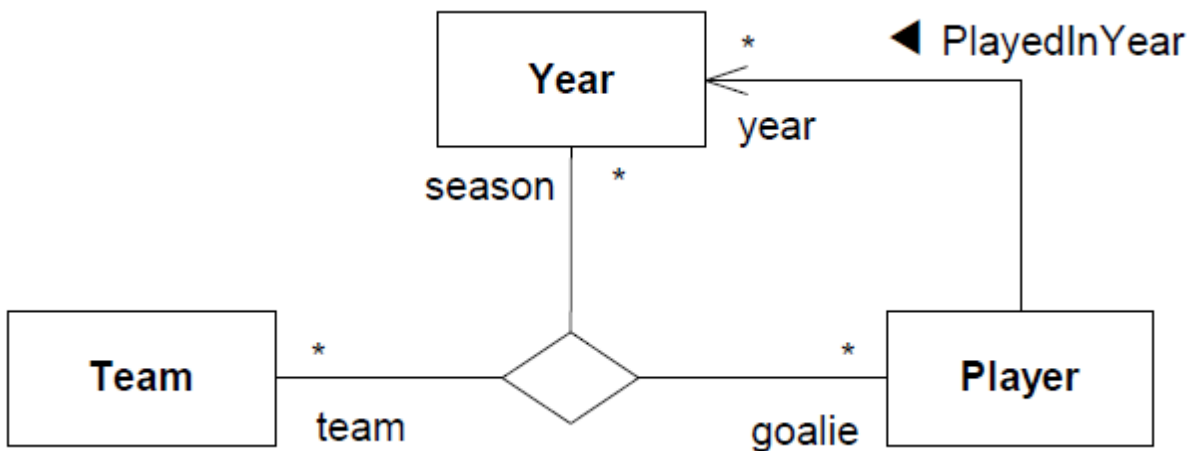


Figure 11.27 Binary and ternary Associations

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

The following example shows Association ends with various adornments.

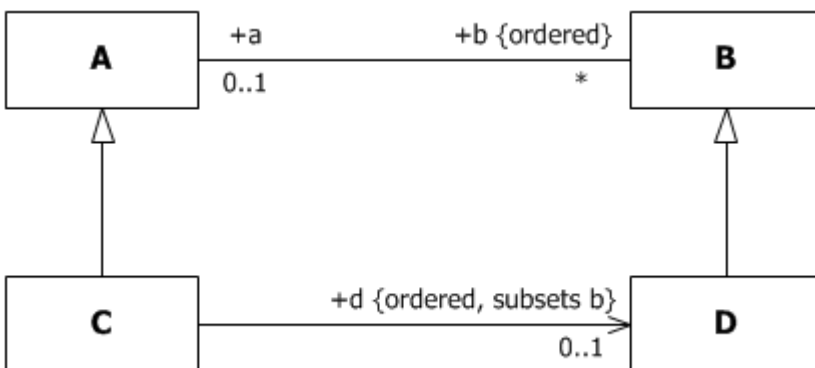


Figure 11.28 Association ends with various adornments

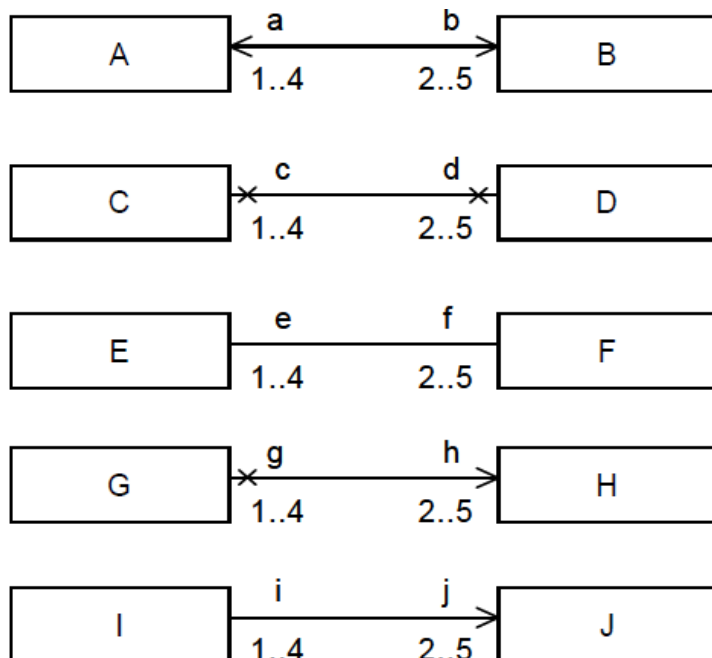
The following adornments are shown on the four Association ends in Figure 11.28.

- Names *a*, *b*, and *d* on three of the ends.

- Public visibility marked on the ends *a*, *b* and *d*.
- Multiplicities 0..1 on *a*, \* on *b*, and 0..1 on *d*.
- Specification of ordering on *b* and *d*.
- 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 ends owned by an association (no dots).



**Figure 11.29** Examples of navigable association-owned ends

In Figure 11.29:

- 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. In a diagram where arrows are only shown for one-way navigable associations, this probably signifies bidirectional 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 non-navigable, in a diagram where arrows are only shown for one-way navigable associations, and crosses are suppressed.

The following examples show some class-owned ends, where class ownership is indicated by the dot. In Figure 11.30:

- In the top pair AB, end *b* is owned by Class A and end *a* is owned by Class B. Because the ends are class-owned, they are navigable.

- In the second pair CD, end d is owned by Class C, and hence is navigable. End c is owned by the Association, and is marked as navigable.
- In the third pair EF, end f is owned by Class E, and hence is navigable. End e is owned by the Association, and is marked as not navigable, in a diagram where arrows are only shown for one-way navigable associations, and crosses are suppressed.
- In the fourth pair GH, end h is owned by Class G and end g is owned by Class H. Because the ends are class-owned, they are navigable. This is in a diagram where arrows are only shown for one-way navigable associations.

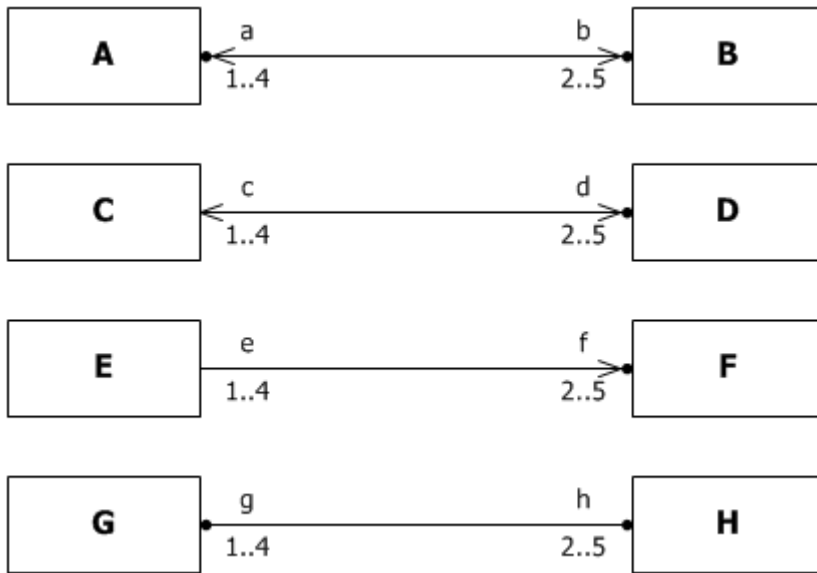


Figure 11.30 Examples of class-owned ends

Figure 11.31 shows that the attribute notation can be used for an Association end owned by a Class, because an Association end owned by a Class is also an attribute. Although it would typically be suppressed on grounds of redundancy, this notation may be used in conjunction with the association notation to make it perfectly clear that the attribute is also an Association end.

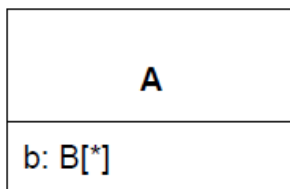


Figure 11.31 Example of attribute notation for navigable end owned by an end Class

Figure 11.32 shows the notation for a derived union. The attribute A::b is derived by being the strict union of all of the attributes that subset it. In this case there is just one of these, C::d. So for an instance of the Class C, d is a subset of b, and b is derived from d.



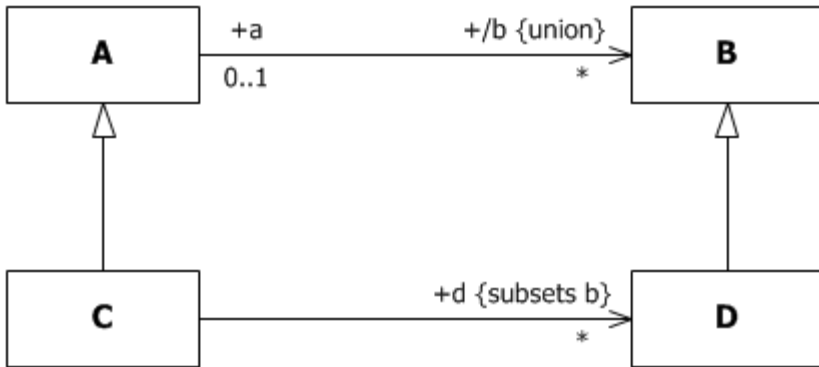


Figure 11.32 Derived supersets (union)

Figure 11.33 shows the black diamond notation for composite aggregation.

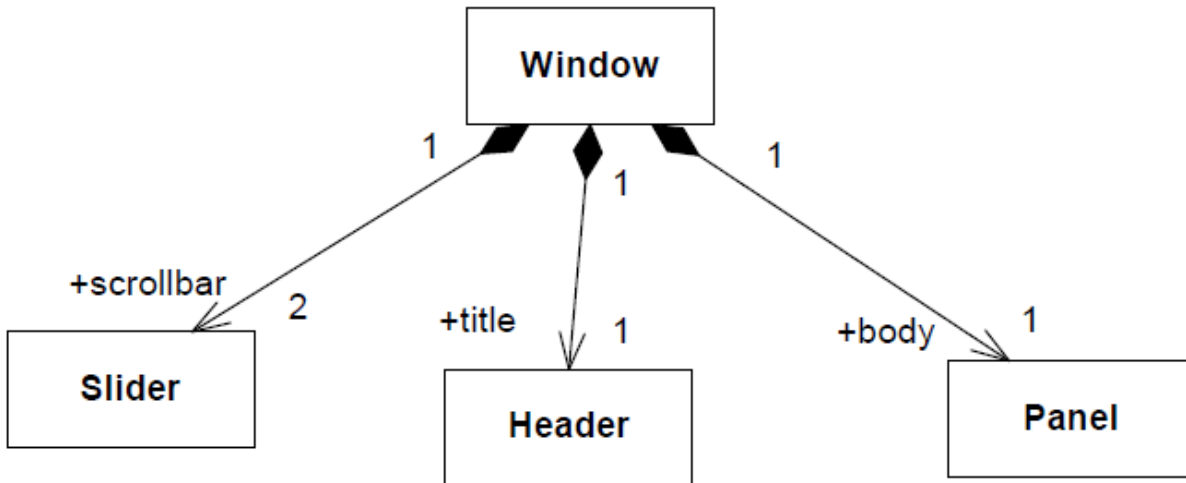
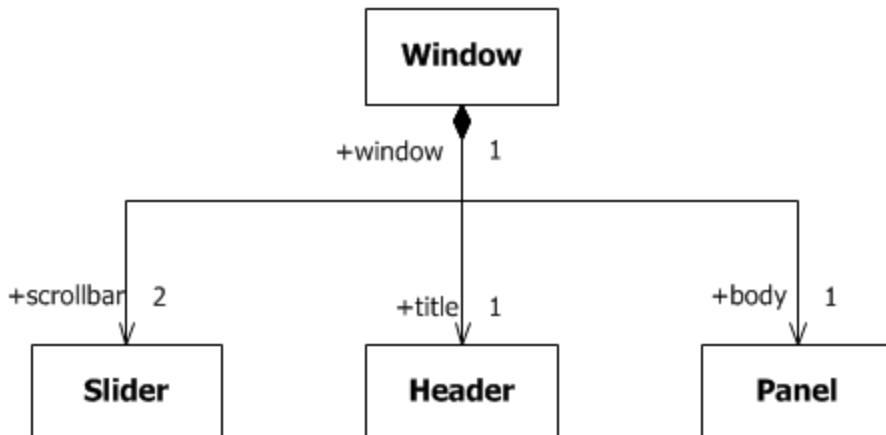


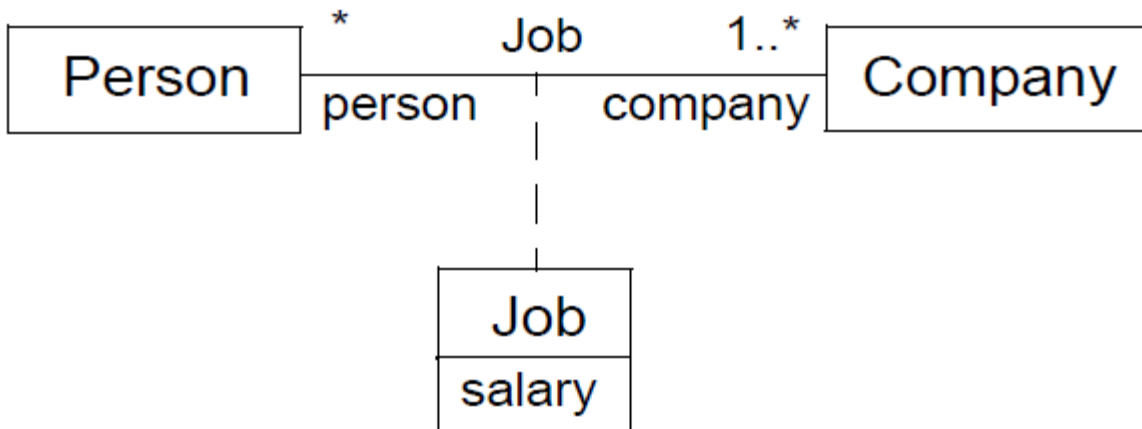
Figure 11.33 Composite aggregation is depicted as a black diamond

Figure 11.34 shows the same model using the notational option of sharing the same source segment between multiple compositions. The multiplicity and name adornments on the shared end apply to all of the compositions.



**Figure 11.34 Composite aggregation sharing a source segment**

Figure 11.35 shows the notation for an AssociationClass. In this example the name of the AssociationClass appears twice, once on the Class rectangle and once on the Association. These are both renderings of the same model element.



**Figure 11.35 Example AssociationClass Job, which is defined between the two Classes Person and Company.**

## 11.6 Components

### 11.6.1 Summary

This sub clause specifies a set of constructs that can be used to define software systems of arbitrary size and complexity. In particular, it specifies a Component as a modular unit with well-defined Interfaces that is replaceable within its environment. The Component concept addresses the area of component-based development and component-based system structuring, where a Component is modeled throughout the development life cycle and successively refined into deployment and run-time.

An important aspect of component-based development is the reuse of previously constructed Components. A Component can always be considered an autonomous unit within a system or subsystem. It has one or more provided and/or required Interfaces (potentially exposed via Ports), and its internals are hidden and inaccessible other than as provided by its Interfaces. Although it may be dependent on other elements in terms of Interfaces that are required, a Component is encapsulated and its Dependencies are designed such that it can be treated as independently as possible. As a result, Components and subsystems can be flexibly reused and replaced by connecting (“wiring”) them together. The aspects of autonomy and reuse also extend to Components at deployment time. The artifacts that implement Component are intended to be capable of being deployed and re-deployed independently, for instance to update an existing system.

The Components package supports the specification of both logical Components (e.g., business components, process components) and physical Components (e.g., EJB components, CORBA components, COM+ and .NET components, WSDL components, etc.), along with the artifacts that implement them and the nodes on which they are deployed and executed. It is anticipated that profiles based around Components will be developed for specific component technologies and associated hardware and software environments.

### 11.6.2 Abstract Syntax

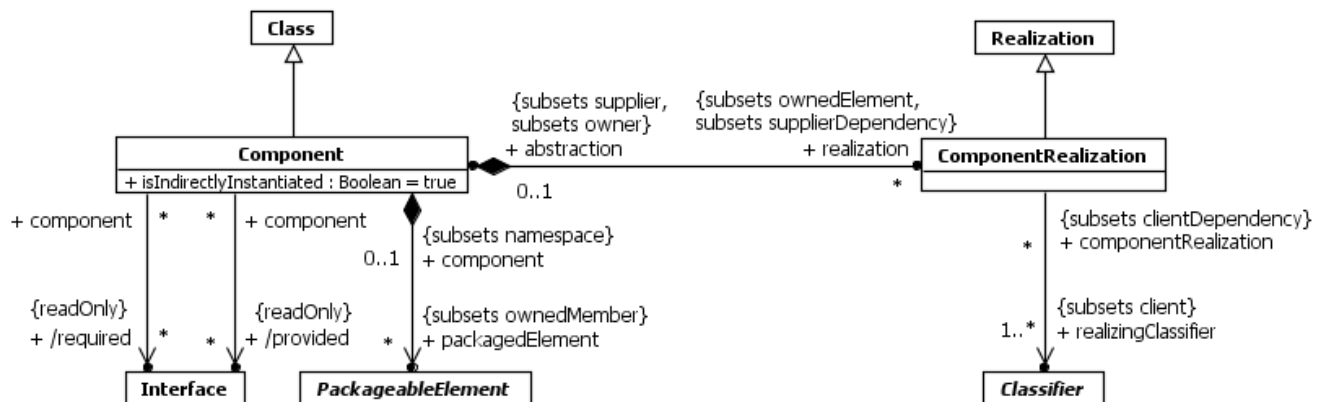


Figure 11.36 Components

### 11.6.3 Semantics

#### Components

A Component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.

A Component is a *self-contained* unit that encapsulates the state and behavior of a number of Classifiers. A Component specifies a formal contract of the services that it provides to its clients and those that it requires from other Components or services in the system in terms of its provided and required Interfaces.

A Component is a *substitutable* unit that can be replaced at design time or run-time by a Component that offers equivalent functionality based on compatibility of its Interfaces. As long as the environment is fully compatible with the provided and required Interfaces of a Component, it will be able to interact with this environment. Similarly, a system can be extended by adding new Component types that add new functionality. Larger pieces of a system's functionality may be assembled by reusing Components as parts in an encompassing Component or assembly of Components, and wiring them together.

A Component is modeled throughout the development life cycle and successively refined into deployment and run-time. A Component may be manifested by one or more Artifacts, and in turn, that Artifact may be deployed to its execution environment. A DeploymentSpecification may define values that parameterize the Component's execution. (See Deployments – Clause 19).

The required and provided Interfaces of a Component allow for the specification of StructuralFeatures such as attributes and Association ends, as well as BehavioralFeatures such as Operations and Receptions. A Component may implement a provided Interface directly, or its realizing Classifiers may do so, or they may be inherited. The required and provided Interfaces may optionally be organized through Ports; these enable the definition of named sets of provided and required Interfaces that are typically (but not always) addressed at run-time.

A Component has an *external view* (or “black-box” view) by means of its publicly visible Properties and Operations. Optionally, a Behavior such as a ProtocolStateMachine may be attached to an Interface, Port, and to the Component itself, to define the external view more precisely by making dynamic constraints in the sequence of Operation calls explicit.

The wiring between Components in a system or other context can be structurally defined by using Dependencies between compatible simple Ports, or between Usages and matching InterfaceRealizations that are represented by sockets and lollipops (see [10.4.4](#)) on Components on Component diagrams. Creating a wiring Dependency between a Usage and a matching InterfaceRealization, or between compatible simple Ports, means that there may be some additional information, such as performance requirements, transport bindings, or other policies that determine that the Interface is realized in a way that is suitable for consumption by the depending Component. Such additional information could be captured in a profile by means of stereotypes.

A Component also has an *internal view* (or “white-box” view) by means of its private Properties and realizing Classifiers. This view shows how the external Behavior is realized internally. Dependencies on the external view provide a convenient overview of what may happen in the internal view; they do not prescribe what must happen. More detailed behavior specifications such as Interactions and Activities may be used to detail the mapping from external to internal behavior.

The execution time semantics for an assembly Connector in a Component are that requests (signals and operation invocations) travel along an instance of a Connector. The execution semantics for multiple Connectors directed to and from different roles, or n-ary Connectors where  $n > 2$ , indicates that the instance that will originate or handle the request will be determined at execution time.

A number of UML standard stereotypes exist that apply to Component. For example, «Subsystem» to model large-scale Components, and «Specification» and «Realization» to model Components with distinct specification and realization definitions, where one specification may have multiple realizations (see the Standard Profiles).

A Component may be realized (or implemented) by a number of Classifiers. In that case, a Component owns a set of ComponentRealizations to these Classifiers.

A component acts like a Package for all model elements that are involved in or related to its definition, which should be either owned or imported explicitly. Typically the Classifiers that realize a Component are owned by it.

The `isDirectlyInstantiated` property specifies the kind of instantiation that applies to a Component. If false, the Component is instantiated as an addressable object. If true, the Component is defined at design-time, but at run-time (or execution-time) an object specified by the Component does not exist, that is, the Component is instantiated indirectly, through the instantiation of its realizing Classifiers or parts.

## 11.6.4 Notation

A Component is shown as a Classifier rectangle with the keyword «component». Optionally, in the right hand corner a Component icon can be displayed. This is a Classifier rectangle with two smaller rectangles protruding from its left hand side. If the icon symbol is shown, the keyword «component» may be hidden.

The attributes, operations and internal structure compartments all have their normal meaning. The internal structure uses the notation defined in StructuredClassifiers (11.2).

The provided and required Interfaces of a Component may be shown by means of ball (lollipop) and socket notation (see [10.4.4](#)), where the lollipops and sockets stick out of the Component rectangle.

For displaying the full signature of a provided or required Interface of a Component, the Interfaces can also be displayed as normal expandable Classifier rectangles. For this option, the Interface rectangles are connected to the Component rectangle by appropriate dependency arrows, as specified in [7.7.4](#) and [10.4.4](#).

A conforming tool may optionally support compartments named “provided interfaces” and “required interfaces” listing the provided and required Interfaces by name. This may be a useful option in scenarios in which a Component has a large number of provided or required Interfaces.

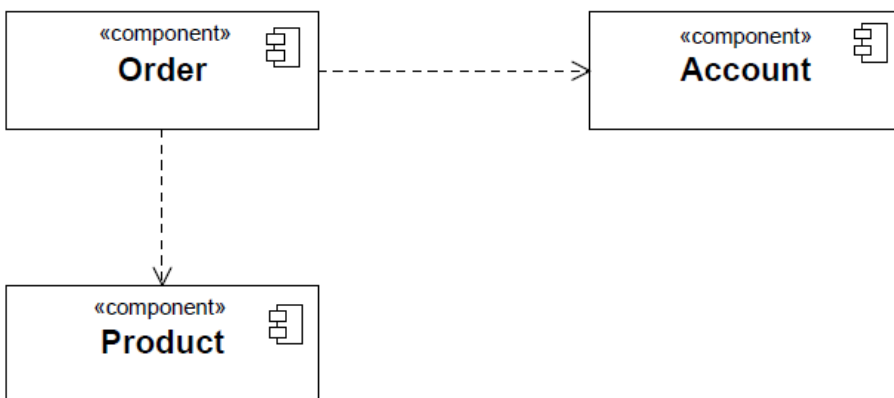
Additional optional compartments “realizations” and “artifacts” may be used to list the realizing Classifiers (Classifiers reached by following the realization property) and manifesting Artifacts (Artifacts that manifest this component – see [19.3](#)).

A ComponentRealization is notated in the same way as a Realization dependency (i.e., as a general dashed line with a hollow triangle as an arrowhead).

The packagedElements of a Component may be displayed in an optional compartment named “packaged elements,” according to the specification for optional compartments for ownedMembers set out in [9.2.4](#).

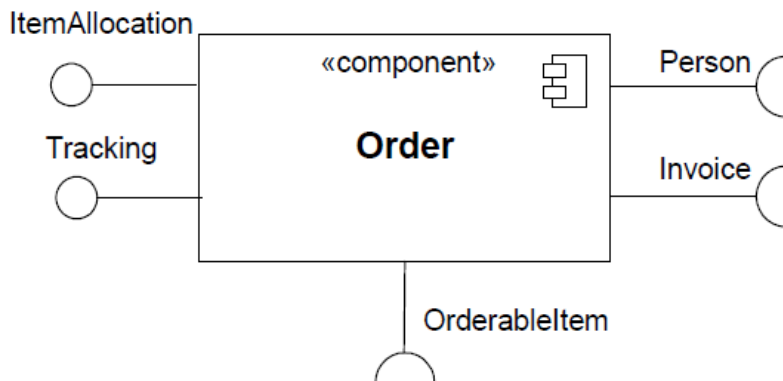
## 11.6.5 Examples

An overview diagram can show Components related by Dependencies, which signify some further unspecified kind of dependency between the components, and by implication a lack of dependency where there are no Dependency arrows.



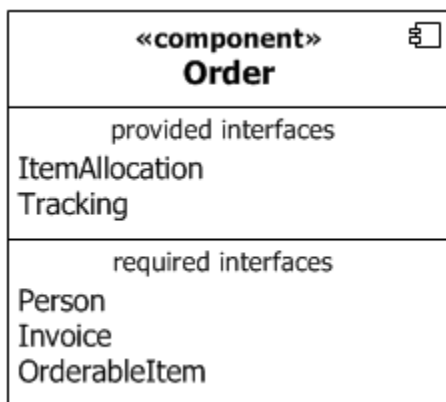
**Figure 11.37 Example of an overview diagram showing Components and their general Dependencies**

Figure 11.38 shows an external (“black-box”) view of a Component by means of interface lollipops and sockets sticking out of the Component rectangle.



**Figure 11.38 A Component with two provided and three required Interfaces**

Figure 11.39 shows provided and required interfaces listed in optional compartments.



**Figure 11.39 Black box notation showing a listing of provided and required interfaces**

Figure 11.40 shows a “white box” view of a Component listing realizing Classifiers and manifesting Artifacts in additional optional compartments.

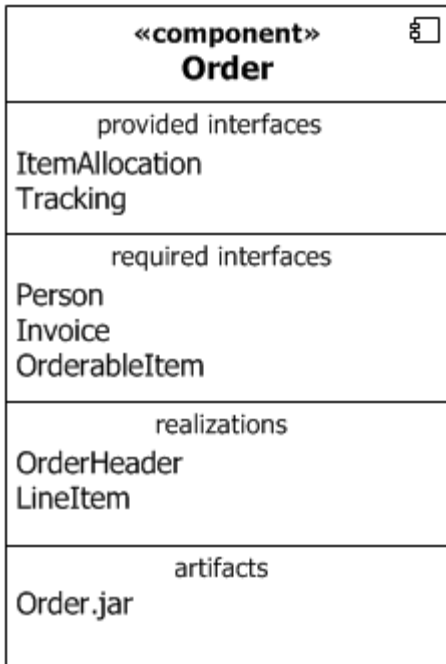


Figure 11.40 Optional “white-box” representation of a Component

Figure 11.41 shows explicit representation of the provided and required Interfaces using Dependency notations, allowing Interface details such as Operations to be displayed.

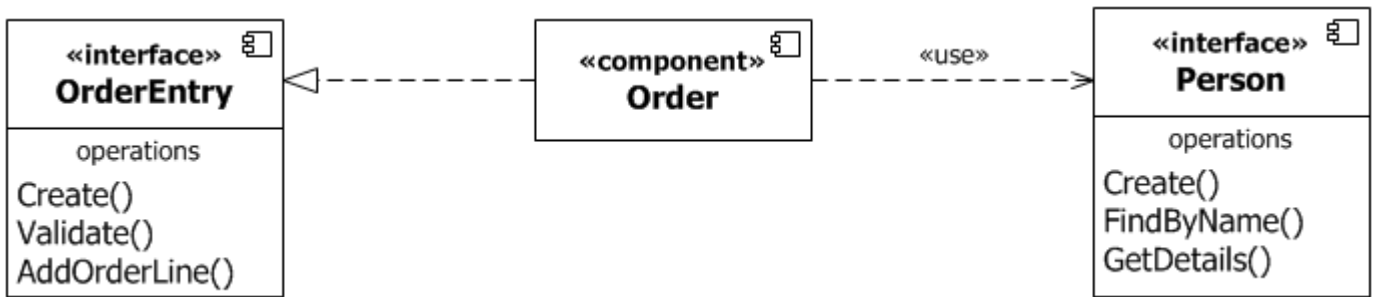
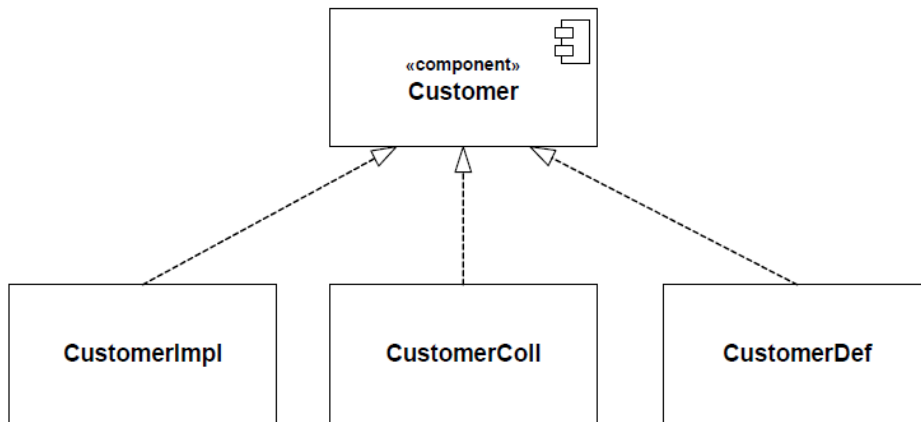


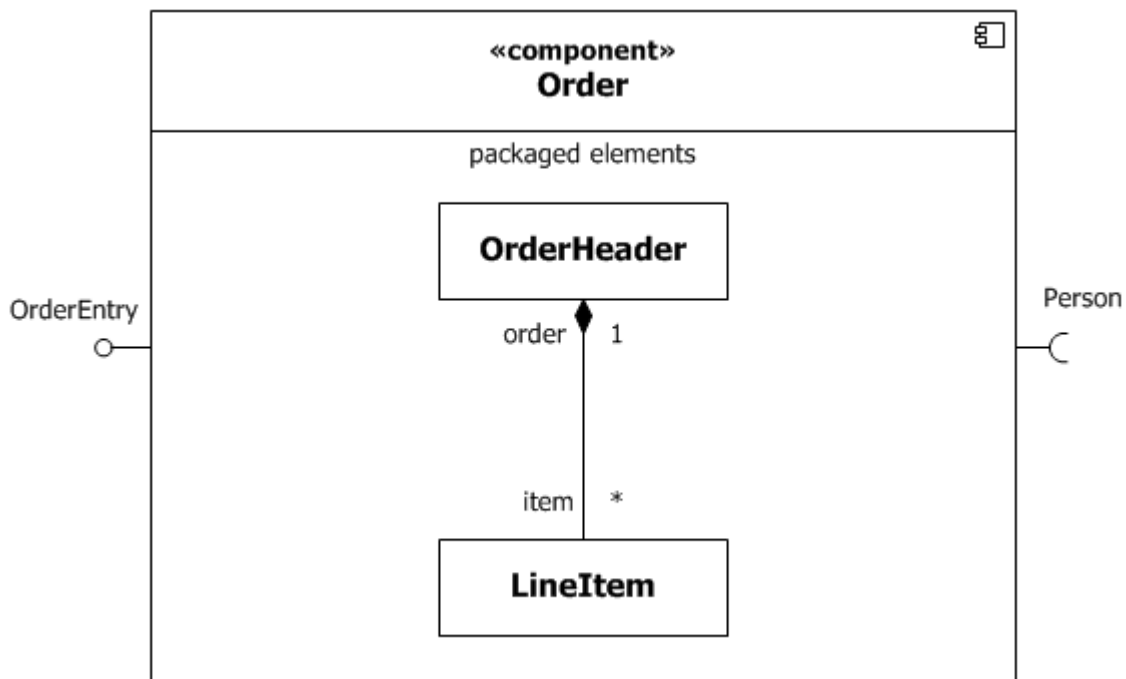
Figure 11.41 Explicit representation of provided and required Interfaces using Dependency notation.

Figure 11.42 shows a set of Classifiers that realize a Component with realization arrows representing the ComponentRealizations.



**Figure 11.42** A representation of the realization of a complex Component

Figure 11.43 shows owned Classes that realize a Component nested within an optional “packaged elements” compartment of the Component shape.



**Figure 11.43** An alternative nested representation of a complex Component

Figure 11.44 shows various ways of wiring Components using Dependencies.

The Dependency on the right of the figure is from the Usage of OrderableItem to the InterfaceRealization of OrderableItem. This also shows that “/OrderableItem” is an Interface that is implemented by a supertype of Product, following the notation specified in [10.4.4](#).

The Dependency between the AccountPayable Ports illustrates the notational option of showing the dependency arrow joining the socket to the lollipop, when a Dependency is wired between simple Ports.



When realizing Classifiers are shown in a packaged elements compartment, a Dependency may be shown from a simple Port to a realizing Classifier to indicate that the Interface provided or required by the Port is dependent in some way upon the Classifier. This is illustrated by the Dependency from AccountPayable to OrderHeader, which indicates that something about the fact that the Component requires AccountPayable is dependent upon OrderHeader.

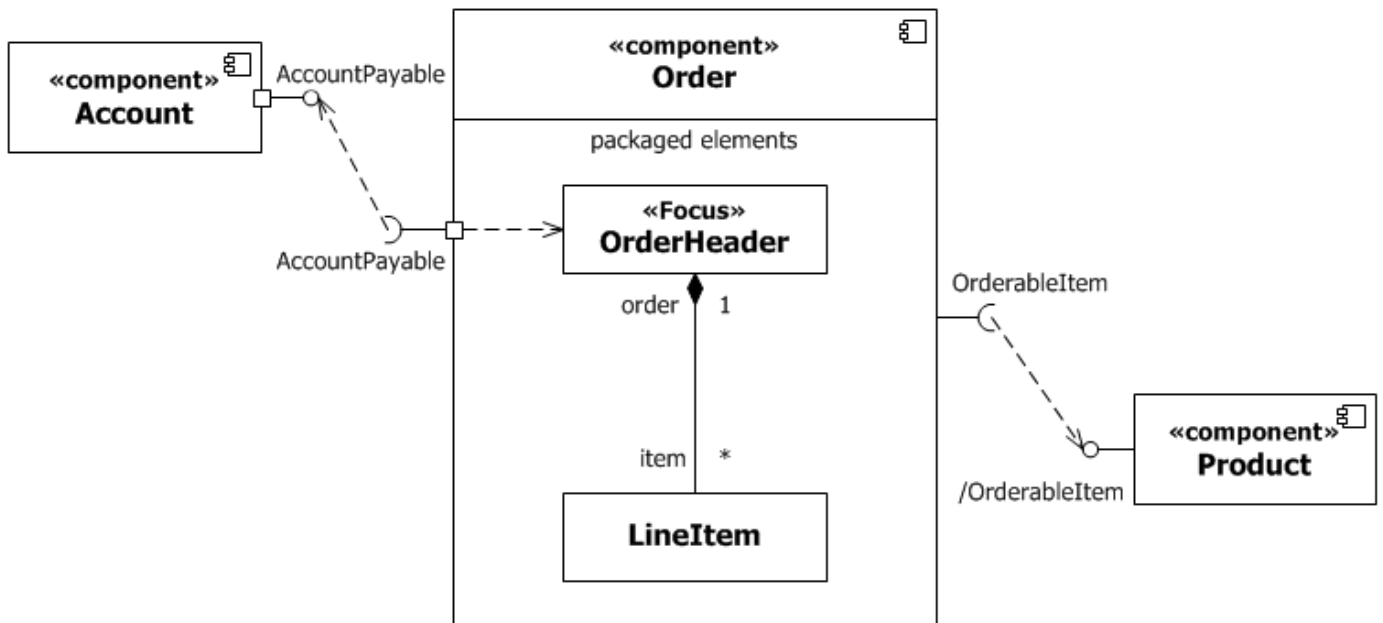


Figure 11.44 Example model of a Component, its provided and required Interfaces, and wiring through Dependencies.

Figure 11.45 shows an internal or white-box view of the internal structure of a Component that contains other Components with simple Ports as parts of its internal assembly. The assembly Connectors use ball-and-socket notation. The delegation connectors use the notational option that the Connector line can end on the ball or socket, rather than the simple port itself.

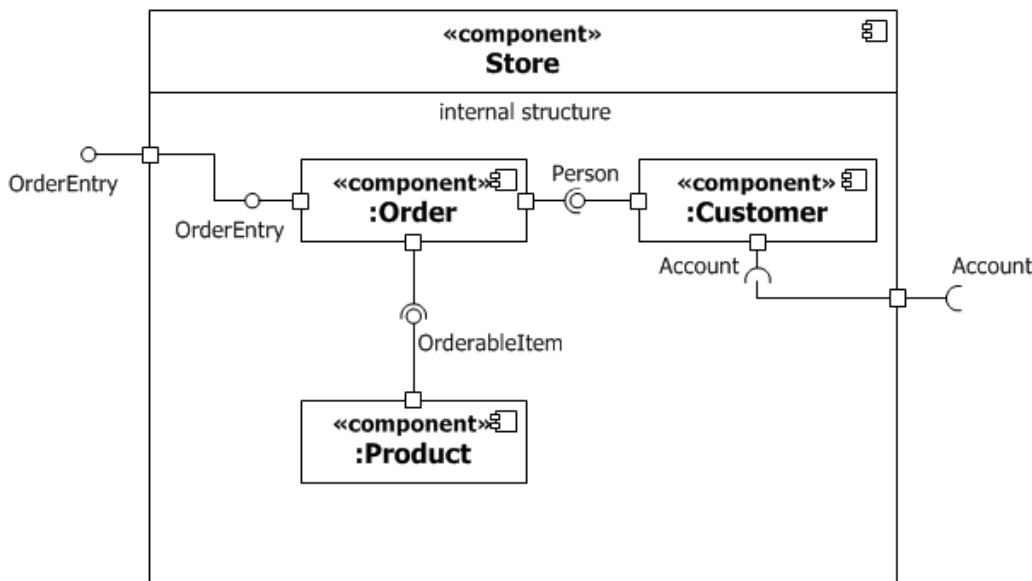


Figure 11.45 Internal structure of a Component

Figure 11.46 shows delegation Connectors from delegating Ports to handling parts; in this example the parts in the internal structure compartment are typed by Classes shown in the optional packaged elements compartment.

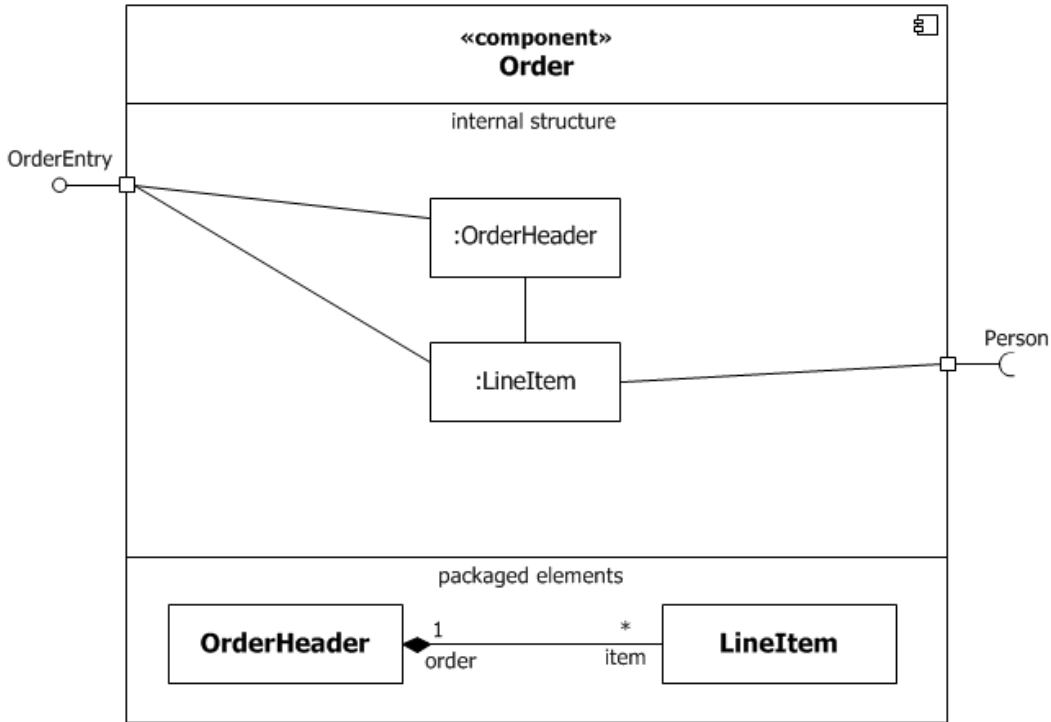


Figure 11.46 Delegation Connectors connect externally provided Interfaces to the parts that realize or require them.

## 11.7 Collaborations

### 11.7.1 Summary

The primary purpose of Collaborations is to explain how a system of communicating elements (roles) collectively accomplish a specific task or set of tasks without necessarily having to incorporate detail that is irrelevant to the explanation. It is intended as a means for capturing standard design patterns.

A CollaborationUse represents the application of the pattern described by a Collaboration to a specific situation involving specific elements playing the roles of the Collaboration.

## 11.7.2 Abstract Syntax

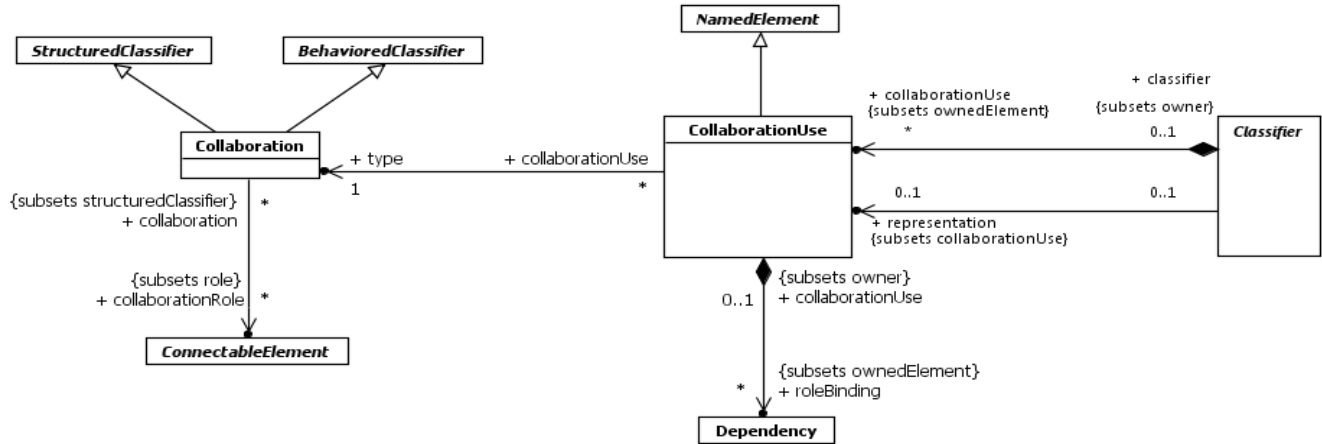


Figure 11.47 Collaborations

## 11.7.3 Semantics

### Collaborations

Collaborations may be used to explain how a collection of cooperating instances achieve a joint task or set of tasks. Therefore, a Collaboration typically incorporates only those aspects that are necessary for its explanation and suppresses everything else. Thus, a given object may be simultaneously playing roles in multiple different Collaborations, but each Collaboration would only represent those aspects of that object that are relevant to its purpose.

A Collaboration defines a set of cooperating participants that are needed for a given task. The roles of a Collaboration will be played by instances when interacting with each other. Their relationships relevant for the given task are shown as Connectors between the roles. Roles of Collaborations define a usage of instances, while the Classifiers typing these roles specify all required Properties of these instances. Thus, a Collaboration specifies what Properties instances must have to be able to participate in the Collaboration. The Connectors between the roles specify what communication paths must exist between the participating instances.

Neither all Features nor all contents of the participating instances nor all links between these instances are always required in a particular Collaboration. Therefore, a Collaboration is often defined in terms of roles typed by Interfaces.

Collaborations may be specialized from other Collaborations. If a role is extended in the specialization, its type in the specialized Collaboration must conform to its type in the general Collaboration. The specialization of the types of the roles does not imply corresponding specialization of the Classifiers that realize those roles. It is sufficient that they conform to the constraints defined by those roles.

A Collaboration is not directly instantiable. Instead, the cooperation defined by the Collaboration comes about as a consequence of the actual cooperation between the instances that play the roles defined in the Collaboration.

### CollaborationUses

A CollaborationUse represents a particular use of a Collaboration to explain the relationships between a set of elements. A CollaborationUse shows how the pattern described by a Collaboration is applied in a given *context* Classifier, by binding specific ConnectableElements from that context to the roles of the Collaboration. There may be multiple CollaborationUses related to a given Collaboration within a Classifier, each bound differently. A given role or Connector may be involved in multiple uses of the same or different Collaborations.

The `roleBindings` are implemented using `Dependencies` owned by the `CollaborationUse`. Each role in the `Collaboration` is bound by a distinct `Dependency` and is its supplier. The client of the `Dependency` is a `ConnectableElement` that relates in some way to the context `Classifier`: it may be a direct role of the context `Classifier`, or an element reachable by some set of references from the context `Classifier`. These `roleBindings` indicate which `ConnectableElement` from the context `Classifier` plays which role in the `Collaboration`.

Connectors in a `Collaboration` typing a `CollaborationUse` must have corresponding Connectors between elements bound in the context `Classifier`, and these corresponding Connectors must have the same or more general type than the `Collaboration` Connectors.

One of the `CollaborationUses` owned by a `Classifier` may be singled out as representing the Behavior of the `Classifier` as a whole. This is called the `Classifier's` representation. The `Collaboration` that is related to the `Classifier` by its representation shows how the instances corresponding to the `StructuralFeatures` of this `Classifier` (e.g., its attributes and parts) interact to generate the overall Behavior of the `Classifier`. The representing `Collaboration` may be used to provide a description of the Behavior of the `Classifier` at a different level of abstraction than is offered by the internal structure of the `Classifier`. The `Properties` of the `Classifier` are mapped to roles in the `Collaboration` by the role bindings of the `CollaborationUse`.

Any Behavior attached to the `Collaboration` applies to the set of roles and Connectors bound within a given `CollaborationUse`. For example, an interaction among parts of a `Collaboration` applies to the `Classifier` parts bound to a single `CollaborationUse`.

If the same `ConnectableElement` is used in both the `Collaboration` and the represented element, no role binding is required.

It is not specified further when client and supplier elements in role bindings are compatible.

#### 11.7.4 Notation

A `Collaboration` is shown as a dashed ellipse shape containing the name of the `Collaboration`. The internal structure of a `Collaboration` as comprised by roles and Connectors may be shown in a compartment within the dashed ellipse shape. This compartment follows the same notational specification as for the internal structure compartment of a normal `Classifier` rectangle.

Alternatively, a composite structure diagram can be used, or a normal `Classifier` rectangle with the keyword «`Collaboration`».

There is no notation defined for a `Collaboration` whose `collaborationRoles` are not `Properties`.

Using an alternative notation for `Properties`, a line may be drawn from the elliptical `Collaboration` shape to rectangles denoting `Classifiers` that are the types of `Properties` of the `Collaboration`. Each line is labeled by the name of the `Property`. In this manner, a diagram can show the definition of a `Collaboration` together with the actual `Classifiers` that type the `collaborationRoles` in that definition

A `CollaborationUse` is shown within an internal structure compartment of the context `Classifier` by a dashed ellipse containing the name of the occurrence, a colon, and the name of the `Collaboration` type. For every `roleBinding`, there is a dashed line from the ellipse to the client element; the dashed line is labeled on the client end with the name of the supplier element. With this notation the Connectors that must exist in the context `Classifier` as a consequence of the bindings may be suppressed.

An optional notation for `CollaborationUse` is as a dashed arrow with the keyword «`occurrence`» pointing from the using `Classifier` to the used `Collaboration`. In conjunction with this the `roleBindings` are shown as normal `Dependency` arrows. With this option any Connectors that must exist in the context `Classifier` as a consequence of the bindings should be shown.

#### 11.7.5 Examples

Figure 11.48 shows the internal structure of the `Collaboration` named `Observer`, with two parts (and hence roles) named `subject` and `observer`, and a `Connector` between them.

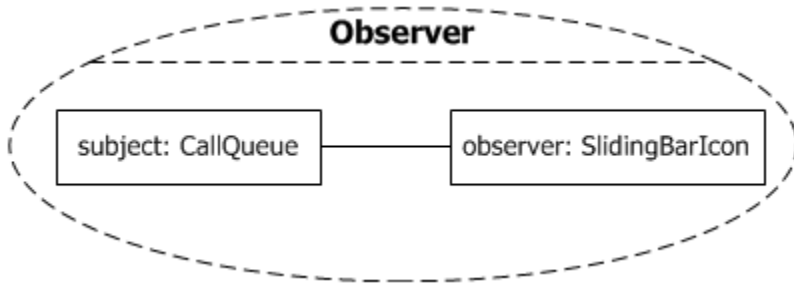


Figure 11.48 The internal structure of the Observer Collaboration

Figure 11.49 shows the alternative notation for definition of the parts of the Observer Collaboration, which allows the details of the Classes CallQueue and SlidingBarIcon to be shown in the same definition. Any instance playing the Subject role must possess the Properties specified by CallQueue, and similarly for the Observer role. The example also shows a Constraint on Observer.

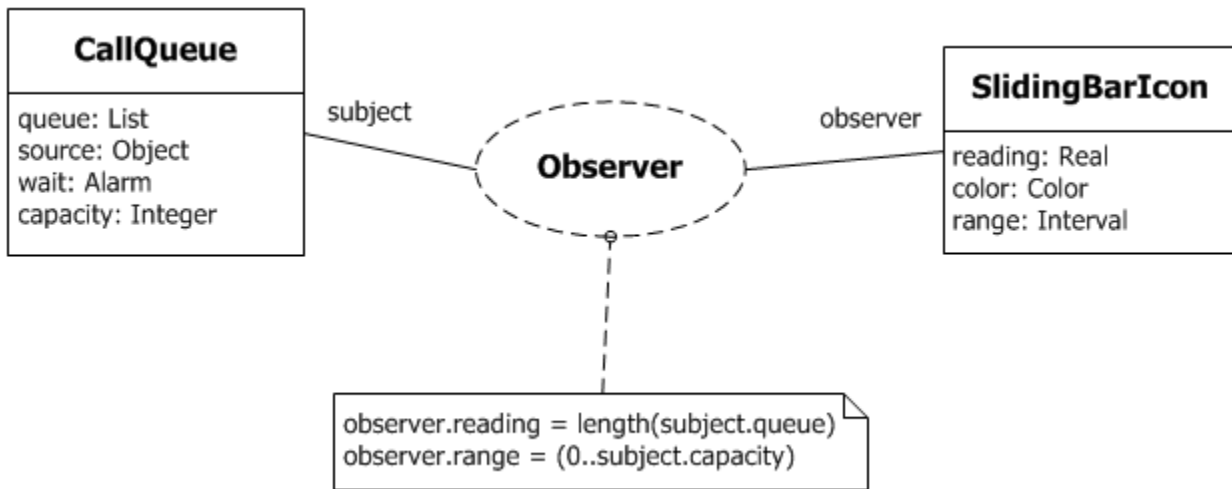


Figure 11.49 Alternative notation for the parts of the Observer Collaboration.

The next example shows the definition of two Collaborations, *Sale* (Figure 11.50) and *BrokeredSale* (Figure 11.51). *Sale* is used twice as part of the definition of *BrokeredSale*. *Sale* is a Collaboration among two collaborationRoles (actually parts), a *seller* and a *buyer*. An interaction, or other Behavior specification, could be attached to *Sale* to specify the steps involved in making a *Sale*.

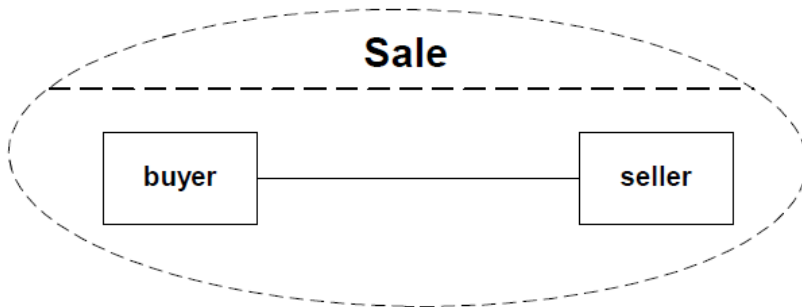


Figure 11.50 The Sale Collaboration

*BrokeredSale* is a Collaboration among three collaborationRoles, a *producer*, a *broker*, and a *consumer*. The specification of *BrokeredSale* shows that it consists of two CollaborationUses of the *Sale* Collaboration, indicated by the dashed ellipses. The occurrence *wholesale* indicates a *Sale* in which the *producer* is the *seller* and the *broker* is the *buyer*. The occurrence *retail* indicates a *Sale* in which the *broker* is the *seller* and the *consumer* is the *buyer*. The Connectors between *sellers* and *buyers* are not shown in the two occurrences; these Connectors must exist in the *BrokeredSale* Collaboration as a consequence of the Connector defined in *Sale*. The *BrokeredSale* Collaboration could itself be used as part of a larger Collaboration.

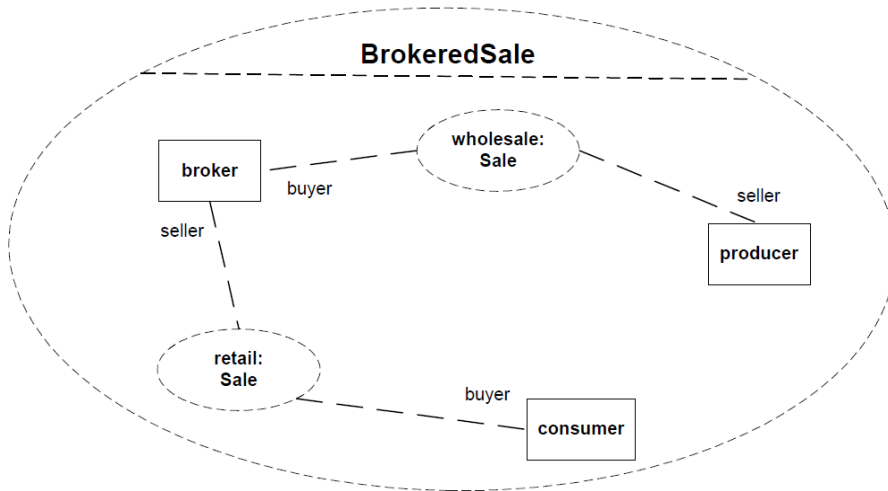


Figure 11.51 The BrokeredSale Collaboration

Figure 11.52 shows part of the *BrokeredSale* Collaboration using the optional «occurrence» notation.

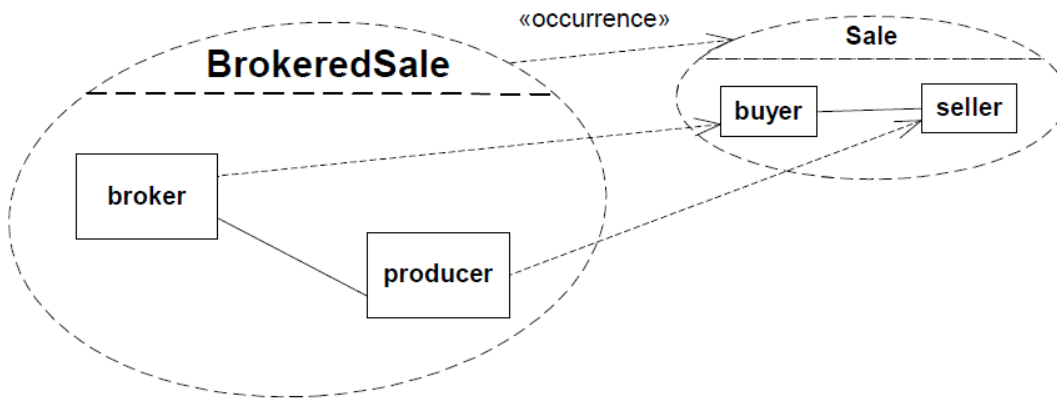


Figure 11.52 A subset of the BrokeredSale Collaboration using «occurrence» and Dependency arrows

## 11.8 Classifier Descriptions

### Association [Class]

#### Description

A link is a tuple of values that refer to typed objects. An Association classifies a set of links, each of which is an instance of the Association. Each value in the link refers to an instance of the type of the corresponding end of the Association.

#### Diagrams

[Structured Classifiers](#), [Associations](#), [Profiles](#), [Nodes](#), [Properties](#), [Link Actions](#)

#### Generalizations

[Relationship](#), [Classifier](#)

#### Specializations

[AssociationClass](#), [Extension](#), [CommunicationPath](#)

#### Attributes

- isDerived : [Boolean](#) [1..1] = false  
Specifies whether the Association is derived from other model elements such as other Associations.

#### Association Ends

- /endType : [Type](#) [1..\*]{ordered, subsets [Relationship::relatedElement](#)} (opposite [A\\_endType\\_association::association](#))  
The Classifiers that are used as types of the ends of the Association.
- memberEnd : [Property](#) [2..\*]{ordered, subsets [Namespace::member](#)} (opposite [Property::association](#))  
Each end represents participation of instances of the Classifier connected to the end in links of the Association.
- navigableOwnedEnd : [Property](#) [0..\*]{subsets [Association::ownedEnd](#)} (opposite [A\\_navigableOwnedEnd\\_association::association](#))  
The navigable ends that are owned by the Association itself.
- ♦ ownedEnd : [Property](#) [0..\*]{ordered, subsets [Classifier::feature](#), subsets [A\\_redefinitionContext\\_redefinableElement::redefinableElement](#), subsets [Association::memberEnd](#), subsets [Namespace::ownedMember](#)} (opposite [Property::owningAssociation](#))  
The ends that are owned by the Association itself.

#### Operations

- endType() : [Type](#) [0..\*]  
endType is derived from the types of the member ends.

```
body: memberEnd->collect(type)
```

## Constraints

- **specialized\_end\_number**  
An Association specializing another Association has the same number of ends as the other Association.

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

- **specialized\_end\_types**  
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 corresponding general end.

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

- **binary\_associations**  
Only binary Associations can be aggregations.

```
inv: memberEnd->exists(aggregation <> AggregationKind::none) implies memberEnd->size() = 2
```

- **association\_ends**  
Ends of Associations with more than two ends must be owned by the Association itself.

```
inv: memberEnd->size() > 2 implies ownedEnd->includesAll(memberEnd)
```

## AssociationClass [Class]

### Description

A model element that has both Association and Class properties. An AssociationClass can be seen as an Association that also has Class properties, or as a Class that also has Association properties. It not only connects a set of Classifiers but also defines a set of Features that belong to the Association itself and not to any of the associated Classifiers.

### Diagrams

[Associations](#)

### Generalizations

[Class, Association](#)

### Constraints

- **cannot\_be\_defined**  
An AssociationClass cannot be defined between itself and something else.

```
inv: self.endType()->excludes(self) and self.endType()->collect(et|et.oclAsType(Classifier).allParents())->flatten()->excludes(self)
```

- **disjoint\_attributes\_ends**  
The owned attributes and owned ends of an AssociationClass are disjoint.



```
inv: ownedAttribute->intersection(ownedEnd)->isEmpty()
```

## Class [Class]

### Description

A Class classifies a set of objects and specifies the features that characterize the structure and behavior of those objects. A Class may have an internal structure and Ports.

### Diagrams

[Classes](#), [Associations](#), [Components](#), [Profiles](#), [Nodes](#), [Behaviors](#), [Properties](#), [Operations](#)

### Generalizations

[BehavedClassifier](#), [EncapsulatedClassifier](#)

### Specializations

[AssociationClass](#), [Component](#), [Behavior](#), [Stereotype](#), [Node](#)

### Attributes

- isAbstract : [Boolean](#) [1..1] = false  
If true, the Class does not provide a complete declaration and cannot be instantiated. An abstract Class is typically used as a target of Associations or Generalizations.
- isActive : [Boolean](#) [1..1] = false  
Determines whether an object specified by this Class is active or not. If true, then the owning Class is referred to as an active Class. If false, then such a Class is referred to as a passive Class.

### Association Ends

- /extension : [Extension](#) [0..\*]{ } (opposite [Extension::metaclass](#))  
This property is used when the Class is acting as a metaclass. It references the Extensions that specify additional properties of the metaclass. The property is derived from the Extensions whose memberEnds are typed by the Class.
- ♦ nestedClassifier : [Classifier](#) [0..\*]{ ordered, subsets [A\\_redefinitionContext\\_redefinableElement::redefinableElement](#), subsets [Namespace::ownedMember](#) } (opposite [A\\_nestedClassifier\\_nestingClass::nestingClass](#))  
The Classifiers that are nested within the Class.
- ♦ ownedAttribute : [Property](#) [0..\*]{ ordered, subsets [Classifier::attribute](#), subsets [Namespace::ownedMember](#), redefines [StructuredClassifier::ownedAttribute](#) } (opposite [Property::class](#))  
The attributes (i.e., the Properties) owned by the Class.
- ♦ ownedOperation : [Operation](#) [0..\*]{ ordered, subsets [Classifier::feature](#), subsets [A\\_redefinitionContext\\_redefinableElement::redefinableElement](#), subsets [Namespace::ownedMember](#) } (opposite [Operation::class](#))  
The Operations owned by the Class.

- ♦ ownedReception : [Reception](#) [0..\*]{subsets [Classifier::feature](#), subsets [Namespace::ownedMember](#)} (opposite [A\\_ownedReception\\_class::class](#))  
The Receptions owned by the Class.
- /superClass : [Class](#) [0..\*]{redefines [Classifier::general](#)} (opposite [A\\_superClass\\_class::class](#))  
The superclasses of a Class, derived from its Generalizations.

## Operations

- extension() : [Extension](#) [0..\*]  
Derivation for Class::/extension : Extension

```
body: Extension.allInstances()->select(ext |
  let endTypes : Sequence(Classifier) = ext.memberEnd->collect(type.oclAsType(Classifier)) in
  endTypes->includes(self) or endTypes.allParents()->includes(self) )
```

- inherit(inhs : [NamedElement](#) [0..\*]) : [NamedElement](#) [0..\*]  
The inherit operation is overridden to exclude redefined Properties.

```
body: let excludedElements : Set(RedefinableElement) =
  inhs->select(inh |
    inh.oclIsKindOf(RedefinableElement) and
    let redinh : RedefinableElement = inh.oclAsType(RedefinableElement) in
    (self.ownedMember->select(oclIsKindOf(RedefinableElement)) -
  >collect(oclAsType(RedefinableElement).redefinedElement))->includes(redinh))->
  collect(oclAsType(RedefinableElement))->asSet()
  in
  inhs - excludedElements
```

- superClass() : [Class](#) [0..\*]  
Derivation for Class::/superClass : Class

```
body: self.general()->select(oclIsKindOf(Class))->collect(oclAsType(Class))->asSet()
```

## Constraints

- passive\_class  
Only an active Class may own Receptions and have a classifierBehavior.

```
inv: not isActive implies (ownedReception->isEmpty() and classifierBehavior = null)
```

## Collaboration [Class]

### Description

A Collaboration describes a structure of collaborating elements (roles), each performing a specialized function, which collectively accomplish some desired functionality.

### Diagrams

[Collaborations](#)

## Generalizations

[StructuredClassifier](#), [BehavioedClassifier](#)

## Association Ends

- collaborationRole : [ConnectableElement](#) [0..\*]{subsets [StructuredClassifier::role](#)} (opposite [A collaborationRole collaboration::collaboration](#))  
References ConnectableElements (possibly owned by other Classifiers) which represent roles that instances may play in this Collaboration.

## CollaborationUse [Class]

### Description

A CollaborationUse is used to specify the application of a pattern specified by a Collaboration to a specific situation.

### Diagrams

[Collaborations](#), [Classifiers](#)

## Generalizations

[NamedElement](#)

## Association Ends

- ♦ roleBinding : [Dependency](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [A roleBinding collaborationUse::collaborationUse](#))  
A mapping between features of the Collaboration and features of the owning Classifier. This mapping indicates which ConnectableElement of the Classifier plays which role(s) in the Collaboration. A ConnectableElement may be bound to multiple roles in the same CollaborationUse (that is, it may play multiple roles).
- type : [Collaboration](#) [1..1] (opposite [A type collaborationUse::collaborationUse](#))  
The Collaboration which is used in this CollaborationUse. The Collaboration defines the cooperation between its roles which are mapped to ConnectableElements relating to the Classifier owning the CollaborationUse.

## Constraints

- client\_elements  
All the client elements of a roleBinding are in one Classifier and all supplier elements of a roleBinding are in one Collaboration.

```
inv: roleBinding->collect(client)->forall(ne1, ne2 |
    ne1.ocIsKindOf(ConnectableElement) and ne2.ocIsKindOf(ConnectableElement) and
    let ce1 : ConnectableElement = ne1.ocAsType(ConnectableElement), ce2 : ConnectableElement =
    ne2.ocAsType(ConnectableElement) in
    ce1.structuredClassifier = ce2.structuredClassifier)
and
roleBinding->collect(supplier)->forall(ne1, ne2 |
    ne1.ocIsKindOf(ConnectableElement) and ne2.ocIsKindOf(ConnectableElement) and
```

```

let ce1 : ConnectableElement = ne1.oclAsType(ConnectableElement), ce2 : ConnectableElement =
ne2.oclAsType(ConnectableElement) in
ce1.collaboration = ce2.collaboration)

```

- **every\_role**  
Every collaborationRole in the Collaboration is bound within the CollaborationUse.

```

inv: type.collaborationRole->forall(role | roleBinding->exists(rb | rb.supplier->includes(role)))

```

- **connectors**  
Connectors in a Collaboration typing a CollaborationUse must have corresponding Connectors between elements bound in the context Classifier, and these corresponding Connectors must have the same or more general type than the Collaboration Connectors.

```

inv: type.ownedConnector->forall(connector |
let rolesConnectedInCollab : Set(ConnectableElement) = connector.end.role->asSet(),
relevantBindings : Set(Dependency) = roleBinding->select(rb | rb.supplier-
>intersection(rolesConnectedInCollab)->notEmpty()),
boundRoles : Set(ConnectableElement) = relevantBindings-
>collect(client.oclAsType(ConnectableElement))->asSet(),
contextClassifier : StructuredClassifier = boundRoles->any(true).structuredClassifier-
>any(true) in
contextClassifier.ownedConnector->exists( correspondingConnector |
correspondingConnector.end.role->forall( role | boundRoles->includes(role) )
and (connector.type->notEmpty() and correspondingConnector.type->notEmpty()) implies
connector.type->forall( conformsTo( correspondingConnector.type) ) )
)

```

## Component [Class]

### Description

A Component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.

### Diagrams

[Components](#)

### Generalizations

[Class](#)

### Attributes

- isIndirectlyInstantiated : [Boolean](#) [1..1] = true  
If true, the Component is defined at design-time, but at run-time (or execution-time) an object specified by the Component does not exist, that is, the Component is instantiated indirectly, through the instantiation of its realizing Classifiers or parts.

### Association Ends

- ♦ packagedElement : [PackageableElement](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [A\\_packageableElement\\_component::component](#))

The set of PackageableElements that a Component owns. In the namespace of a Component, all model elements that are involved in or related to its definition may be owned or imported explicitly. These may include e.g., Classes, Interfaces, Components, Packages, UseCases, Dependencies (e.g., mappings), and Artifacts.

- /provided : [Interface](#) [0..\*]{} (opposite [A\\_provided\\_component::component](#))  
The Interfaces that the Component exposes to its environment. These Interfaces may be Realized by the Component or any of its realizingClassifiers, or they may be the Interfaces that are provided by its public Ports.
- ♦ realization : [ComponentRealization](#) [0..\*]{} (subsets [Element::ownedElement](#), subsets [A\\_supplier\\_supplierDependency::supplierDependency](#)) (opposite [ComponentRealization::abstraction](#))  
The set of Realizations owned by the Component. Realizations reference the Classifiers of which the Component is an abstraction; i.e., that realize its behavior.
- /required : [Interface](#) [0..\*]{} (opposite [A\\_required\\_component::component](#))  
The Interfaces that the Component requires from other Components in its environment in order to be able to offer its full set of provided functionality. These Interfaces may be used by the Component or any of its realizingClassifiers, or they may be the Interfaces that are required by its public Ports.

## Operations

- provided() : [Interface](#) [0..\*]  
Derivation for Component::/provided

```
body: let ris : Set(Interface) = allRealizedInterfaces(),
      realizingClassifiers : Set(Classifier) = self.realization.realizingClassifier-
>union(self.allParents()->collect(realization.realizingClassifier))->asSet(),
      allRealizingClassifiers : Set(Classifier) = realizingClassifiers-
>union(realizingClassifiers.allParents()->asSet(),
      realizingClassifierInterfaces : Set(Interface) = allRealizingClassifiers->iterate(c; rci :
Set(Interface) = Set{} | rci->union(c.allRealizedInterfaces()))),
      ports : Set(Port) = self.ownedPort->union(allParents()->collect(ownedPort))->asSet(),
      providedByPorts : Set(Interface) = ports.provided->asSet()
in ris->union(realizingClassifierInterfaces) ->union(providedByPorts) ->asSet()
```

- required() : [Interface](#) [0..\*]  
Derivation for Component::/required

```
body: let uis : Set(Interface) = allUsedInterfaces(),
      realizingClassifiers : Set(Classifier) = self.realization.realizingClassifier-
>union(self.allParents()->collect(realization.realizingClassifier))->asSet(),
      allRealizingClassifiers : Set(Classifier) = realizingClassifiers-
>union(realizingClassifiers.allParents()->asSet(),
      realizingClassifierInterfaces : Set(Interface) = allRealizingClassifiers->iterate(c; rci :
Set(Interface) = Set{} | rci->union(c.allUsedInterfaces()))),
      ports : Set(Port) = self.ownedPort->union(allParents()->collect(ownedPort))->asSet(),
      usedByPorts : Set(Interface) = ports.required->asSet()
in uis->union(realizingClassifierInterfaces) ->union(usedByPorts) ->asSet()
```

## Constraints

- no\_nested\_classifiers  
A Component cannot nest Classifiers.

```
inv: nestedClassifier->isEmpty()
```

- `no_packaged_elements`  
A Component nested in a Class cannot have any packaged elements.

```
inv: nestingClass <> null implies packagedElement->isEmpty()
```

## ComponentRealization [Class]

### Description

Realization is specialized to (optionally) define the Classifiers that realize the contract offered by a Component in terms of its provided and required Interfaces. The Component forms an abstraction from these various Classifiers.

### Diagrams

[Components](#)

### Generalizations

[Realization](#)

### Association Ends

- abstraction : [Component](#) [0..1]{subsets [Dependency::supplier](#), subsets [Element::owner](#)} (opposite [Component::realization](#))  
The Component that owns this ComponentRealization and which is implemented by its realizing Classifiers.
- realizingClassifier : [Classifier](#) [1..\*]{subsets [Dependency::client](#)} (opposite [A\\_realizingClassifier\\_componentRealization::componentRealization](#))  
The Classifiers that are involved in the implementation of the Component that owns this Realization.

## ConnectableElement [Abstract Class]

### Description

ConnectableElement is an abstract metaclass representing a set of instances that play roles of a StructuredClassifier. ConnectableElements may be joined by attached Connectors and specify configurations of linked instances to be created within an instance of the containing StructuredClassifier.

### Diagrams

[Structured Classifiers](#), [Collaborations](#), [Activities](#), [Lifelines](#), [Features](#), [Properties](#)

### Generalizations

[TypedElement](#), [ParameterableElement](#)

### Specializations

[Variable](#), [Parameter](#), [Property](#)

## Association Ends

- /end : [ConnectorEnd](#) [0..\*]{ordered} (opposite [ConnectorEnd::role](#))  
A set of ConnectorEnds that attach to this ConnectableElement.
- templateParameter : [ConnectableElementTemplateParameter](#) [0..1]{redefines [ParameterableElement::templateParameter](#)} (opposite [ConnectableElementTemplateParameter::parameteredElement](#))  
The ConnectableElementTemplateParameter for this ConnectableElement parameter.

## Operations

- end() : [ConnectorEnd](#) [0..\*]  
Derivation for ConnectableElement::end : ConnectorEnd  
  
body: ConnectorEnd.allInstances()->select(role = self)

## ConnectableElementTemplateParameter [Class]

### Description

A ConnectableElementTemplateParameter exposes a ConnectableElement as a formal parameter for a template.

### Diagrams

[Structured Classifiers](#)

### Generalizations

[TemplateParameter](#)

### Association Ends

- parameteredElement : [ConnectableElement](#) [1..1]{redefines [TemplateParameter::parameteredElement](#)} (opposite [ConnectableElement::templateParameter](#))  
The ConnectableElement for this ConnectableElementTemplateParameter.

## Connector [Class]

### Description

A Connector specifies links that enables communication between two or more instances. In contrast to Associations, which specify links between any instance of the associated Classifiers, Connectors specify links between instances playing the connected parts only.

### Diagrams

[Structured Classifiers](#), [Messages](#), [Information Flows](#)

## Generalizations

### [Feature](#)

## Attributes

- /kind : [ConnectorKind](#) [1..1]  
Indicates the kind of Connector. This is derived: a Connector with one or more ends connected to a Port which is not on a Part and which is not a behavior port is a delegation; otherwise it is an assembly.

## Association Ends

- contract : [Behavior](#) [0..\*] (opposite [A\\_contract\\_connector::connector](#))  
The set of Behaviors that specify the valid interaction patterns across the Connector.
- ♦ end : [ConnectorEnd](#) [2..\*]{ordered, subsets [Element::ownedElement](#)} (opposite [A\\_end\\_connector::connector](#))  
A Connector has at least two ConnectorEnds, each representing the participation of instances of the Classifiers typing the ConnectableElements attached to the end. The set of ConnectorEnds is ordered.
- redefinedConnector : [Connector](#) [0..\*]{subsets [RedefinableElement::redefinedElement](#)} (opposite [A\\_redefinedConnector\\_connector::connector](#))  
A Connector may be redefined when its containing Classifier is specialized. The redefining Connector may have a type that specializes the type of the redefined Connector. The types of the ConnectorEnds of the redefining Connector may specialize the types of the ConnectorEnds of the redefined Connector. The properties of the ConnectorEnds of the redefining Connector may be replaced.
- type : [Association](#) [0..1] (opposite [A\\_type\\_connector::connector](#))  
An optional Association that classifies links corresponding to this Connector.

## Operations

- kind() : [ConnectorKind](#)  
Derivation for Connector::/kind : ConnectorKind

```
body: if end->exists(  
    role.ocIsKindOf(Port)  
    and partWithPort->isEmpty()  
    and not role.ocIsType(Port).isBehavior)  
then ConnectorKind::delegation  
else ConnectorKind::assembly  
endif
```

## Constraints

- types  
The types of the ConnectableElements that the ends of a Connector are attached to must conform to the types of the ends of the Association that types the Connector, if any.

```
inv: type<>null implies  
    let noOfEnds : Integer = end->size() in
```



```
(type.memberEnd->size() = noOfEnds) and Sequence{1..noOfEnds}->forall(i | end->at(i).role.type.conformsTo(type.memberEnd->at(i).type))
```

- roles  
The ConnectableElements attached as roles to each ConnectorEnd owned by a Connector must be roles of the Classifier that owned the Connector, or they must be Ports of such roles.

```
inv: structuredClassifier <> null
and
end->forall( e |
  structuredClassifier.role->includes(e.role)
  or
  e.role.ocIsKindOf(Port) and structuredClassifier.role->includes(e.partWithPort))
```

## ConnectorEnd [Class]

### Description

A ConnectorEnd is an endpoint of a Connector, which attaches the Connector to a ConnectableElement.

### Diagrams

[Encapsulated Classifiers](#), [Structured Classifiers](#)

### Generalizations

[MultiplicityElement](#)

### Association Ends

- /definingEnd : [Property](#) [0..1]{} (opposite [A definingEnd connectorEnd::connectorEnd](#))  
A derived property referencing the corresponding end on the Association which types the Connector owing this ConnectorEnd, if any. It is derived by selecting the end at the same place in the ordering of Association ends as this ConnectorEnd.
- partWithPort : [Property](#) [0..1] (opposite [A partWithPort connectorEnd::connectorEnd](#))  
Indicates the role of the internal structure of a Classifier with the Port to which the ConnectorEnd is attached.
- role : [ConnectableElement](#) [1..1] (opposite [ConnectableElement::end](#))  
The ConnectableElement attached at this ConnectorEnd. When an instance of the containing Classifier is created, a link may (depending on the multiplicities) be created to an instance of the Classifier that types this ConnectableElement.

### Operations

- definingEnd() : [Property](#) [0..1]  
Derivation for ConnectorEnd::/definingEnd : Property

```
body: if connector.type = null
then
  null
else
  let index : Integer = connector.end->indexOf(self) in
  connector.type.memberEnd->at(index)
```

```
endif
```

## Constraints

- **role\_and\_part\_with\_port**  
If a ConnectorEnd references a partWithPort, then the role must be a Port that is defined or inherited by the type of the partWithPort.

```
inv: partWithPort->notEmpty() implies  
    (role.ocIsKindOf(Port) and partWithPort.type.ocIsType(Namespace).member->includes(role))
```

- **part\_with\_port\_empty**  
If a ConnectorEnd is attached to a Port of the containing Classifier, partWithPort will be empty.

```
inv: (role.ocIsKindOf(Port) and role.owner = connector.owner) implies partWithPort->isEmpty()
```

- **multiplicity**  
The multiplicity of the ConnectorEnd may not be more general than the multiplicity of the corresponding end of the Association typing the owning Connector, if any.

```
inv: self.compatibleWith(definingEnd)
```

- **self\_part\_with\_port**  
The Property held in self.partWithPort must not be a Port.

```
inv: partWithPort->notEmpty() implies not partWithPort.ocIsKindOf(Port)
```

## ConnectorKind [Enumeration]

### Description

ConnectorKind is an enumeration that defines whether a Connector is an assembly or a delegation.

### Diagrams

- [Structured Classifiers](#)

### Literals

- **assembly**  
Indicates that the Connector is an assembly Connector.
- **delegation**  
Indicates that the Connector is a delegation Connector.

## EncapsulatedClassifier [Abstract Class]

### Description

An EncapsulatedClassifier may own Ports to specify typed interaction points.

### Diagrams

[Encapsulated Classifiers](#), [Classes](#)

### Generalizations

[StructuredClassifier](#)

### Specializations

[Class](#)

### Association Ends

- ◆ /ownedPort : [Port](#) [0..\*]{subsets [StructuredClassifier::ownedAttribute](#)} (opposite [A\\_ownedPort\\_encapsulatedClassifier::encapsulatedClassifier](#))  
The Ports owned by the EncapsulatedClassifier.

### Operations

- ownedPort() : [Port](#) [0..\*]  
Derivation for EncapsulatedClassifier::/ownedPort : Port

```
body: ownedAttribute->select (oclIsKindOf (Port)) ->collect (oclAsType (Port)) ->asOrderedSet ()
```

## Port [Class]

### Description

A Port is a property of an EncapsulatedClassifier that specifies a distinct interaction point between that EncapsulatedClassifier and its environment or between the (behavior of the) EncapsulatedClassifier and its internal parts. Ports are connected to Properties of the EncapsulatedClassifier by Connectors through which requests can be made to invoke BehavioralFeatures. A Port may specify the services an EncapsulatedClassifier provides (offers) to its environment as well as the services that an EncapsulatedClassifier expects (requires) of its environment. A Port may have an associated ProtocolStateMachine.

### Diagrams

[Encapsulated Classifiers](#), [Events](#), [Invocation Actions](#)

### Generalizations

[Property](#)

## Attributes

- isBehavior : [Boolean](#) [1..1] = false  
Specifies whether requests arriving at this Port are sent to the classifier behavior of this EncapsulatedClassifier. Such a Port is referred to as a behavior Port. Any invocation of a BehavioralFeature targeted at a behavior Port will be handled by the instance of the owning EncapsulatedClassifier itself, rather than by any instances that it may contain.
- isConjugated : [Boolean](#) [1..1] = false  
Specifies the way that the provided and required Interfaces are derived from the Port's Type.
- isService : [Boolean](#) [1..1] = true  
If true, indicates that this Port is used to provide the published functionality of an EncapsulatedClassifier. If false, this Port is used to implement the EncapsulatedClassifier but is not part of the essential externally-visible functionality of the EncapsulatedClassifier and can, therefore, be altered or deleted along with the internal implementation of the EncapsulatedClassifier and other properties that are considered part of its implementation.

## Association Ends

- protocol : [ProtocolStateMachine](#) [0..1] (opposite [A\\_protocol\\_port::port](#))  
An optional ProtocolStateMachine which describes valid interactions at this interaction point.
- /provided : [Interface](#) [0..\*]{ } (opposite [A\\_provided\\_port::port](#))  
The Interfaces specifying the set of Operations and Receptions that the EncapsulatedClassifier offers to its environment via this Port, and which it will handle either directly or by forwarding it to a part of its internal structure. This association is derived according to the value of isConjugated. If isConjugated is false, provided is derived as the union of the sets of Interfaces realized by the type of the port and its supertypes, or directly from the type of the Port if the Port is typed by an Interface. If isConjugated is true, it is derived as the union of the sets of Interfaces used by the type of the Port and its supertypes.
- redefinedPort : [Port](#) [0..\*]{ subsets [Property::redefinedProperty](#) } (opposite [A\\_redefinedPort\\_port::port](#))  
A Port may be redefined when its containing EncapsulatedClassifier is specialized. The redefining Port may have additional Interfaces to those that are associated with the redefined Port or it may replace an Interface by one of its subtypes.
- /required : [Interface](#) [0..\*]{ } (opposite [A\\_required\\_port::port](#))  
The Interfaces specifying the set of Operations and Receptions that the EncapsulatedClassifier expects its environment to handle via this port. This association is derived according to the value of isConjugated. If isConjugated is false, required is derived as the union of the sets of Interfaces used by the type of the Port and its supertypes. If isConjugated is true, it is derived as the union of the sets of Interfaces realized by the type of the Port and its supertypes, or directly from the type of the Port if the Port is typed by an Interface.

## Operations

- provided() : [Interface](#) [0..\*]  
Derivation for Port::/provided

```
body: if isConjugated then basicRequired() else basicProvided() endif
```

- `required() : Interface [0..*]`  
Derivation for `Port::/required`

```
body: if isConjugated then basicProvided() else basicRequired() endif
```

- `basicProvided() : Interface [0..*]`  
The union of the sets of Interfaces realized by the type of the Port and its supertypes, or directly the type of the Port if the Port is typed by an Interface.

```
body: if type.oclIsKindOf(Interface)
then type.oclAsType(Interface)->asSet()
else type.oclAsType(Classifier).allRealizedInterfaces()
endif
```

- `basicRequired() : Interface [0..*]`  
The union of the sets of Interfaces used by the type of the Port and its supertypes.

```
body: type.oclAsType(Classifier).allUsedInterfaces()
```

## Constraints

- `port_aggregation`  
Port.aggregation must be composite.
- `default_value`  
A `defaultValue` for port cannot be specified when the type of the Port is an Interface.

```
inv: aggregation = AggregationKind::composite
```

```
inv: type.oclIsKindOf(Interface) implies defaultValue->isEmpty()
```

- `encapsulated_owner`  
All Ports are owned by an `EncapsulatedClassifier`.

```
inv: owner = encapsulatedClassifier
```

## StructuredClassifier [Abstract Class]

### Description

StructuredClassifiers may contain an internal structure of connected elements each of which plays a role in the overall Behavior modeled by the StructuredClassifier.

### Diagrams

[Encapsulated Classifiers](#), [Structured Classifiers](#), [Collaborations](#)

### Generalizations

[Classifier](#)

## Specializations

[Collaboration](#), [EncapsulatedClassifier](#)

## Association Ends

- ♦ ownedAttribute : [Property](#) [0..\*]{ordered, subsets [Classifier::attribute](#), subsets [StructuredClassifier::role](#), subsets [Namespace::ownedMember](#)} (opposite [A\\_ownedAttribute\\_structuredClassifier::structuredClassifier](#))  
The Properties owned by the StructuredClassifier.
- ♦ ownedConnector : [Connector](#) [0..\*]{subsets [Classifier::feature](#), subsets [A\\_redefinitionContext\\_redefinableElement::redefinableElement](#), subsets [Namespace::ownedMember](#)} (opposite [A\\_ownedConnector\\_structuredClassifier::structuredClassifier](#))  
The connectors owned by the StructuredClassifier.
- /part : [Property](#) [0..\*]{} (opposite [A\\_part\\_structuredClassifier::structuredClassifier](#))  
The Properties specifying instances that the StructuredClassifier owns by composition. This collection is derived, selecting those owned Properties where isComposite is true.
- /role : [ConnectableElement](#) [0..\*]{union, subsets [Namespace::member](#)} (opposite [A\\_role\\_structuredClassifier::structuredClassifier](#))  
The roles that instances may play in this StructuredClassifier.

## Operations

- part() : [Property](#) [0..\*]  
Derivation for StructuredClassifier::part  
  
`body: ownedAttribute->select(isComposite)`

# 11.9 Association Descriptions

## A\_collaborationRole\_collaboration [Association]

### Diagrams

[Collaborations](#)

### Owned Ends

- collaboration : [Collaboration](#) [0..\*]{subsets [A\\_role\\_structuredClassifier::structuredClassifier](#)} (opposite [Collaboration::collaborationRole](#))

## A\_connectableElement\_templateParameter\_parameteredElement [Association]

### Diagrams

[Structured Classifiers](#)

### Member Ends

- [ConnectableElement::templateParameter](#)
- [ConnectableElementTemplateParameter::parameteredElement](#)

## A\_contract\_connector [Association]

### Diagrams

[Structured Classifiers](#)

### Owned Ends

- connector : [Connector](#) [0..\*] (opposite [Connector::contract](#))

## A\_definingEnd\_connectorEnd [Association]

### Diagrams

[Structured Classifiers](#)

### Owned Ends

- connectorEnd : [ConnectorEnd](#) [0..\*] (opposite [ConnectorEnd::definingEnd](#))

## A\_endType\_association [Association]

### Diagrams

[Associations](#)

### Owned Ends

- association : [Association](#) [0..\*]{subsets [A\\_relatedElement\\_relationship::relationship](#)} (opposite [Association::endType](#))

## A\_end\_connector [Association]

### Diagrams

[Structured Classifiers](#)

### Owned Ends

- connector : [Connector](#) [1..1]{subsets [Element::owner](#)} (opposite [Connector::end](#))

## A\_end\_role [Association]

### Diagrams

[Structured Classifiers](#)

### Member Ends

- [ConnectableElement::end](#)
- [ConnectorEnd::role](#)

## A\_extension\_metaclass [Association]

### Diagrams

[Classes](#), [Profiles](#)

### Member Ends

- [Class::extension](#)
- [Extension::metaclass](#)

## A\_memberEnd\_association [Association]

### Diagrams

[Associations](#), [Properties](#)

### Member Ends

- [Association::memberEnd](#)
- [Property::association](#)

## A\_navigableOwnedEnd\_association [Association]

### Diagrams

[Associations](#)



## Owned Ends

- association : [Association](#) [0..1]{subsets [Property::owningAssociation](#)} (opposite [Association::navigableOwnedEnd](#))

## A\_nestedClassifier\_nestingClass [Association]

### Diagrams

[Classes](#)

## Owned Ends

- nestingClass : [Class](#) [0..1]{subsets [NamedElement::namespace](#), subsets [RedefinableElement::redefinitionContext](#)} (opposite [Class::nestedClassifier](#))

## A\_ownedAttribute\_class [Association]

### Diagrams

[Classes](#), [Properties](#)

## Member Ends

- [Class::ownedAttribute](#)
- [Property::class](#)

## A\_ownedAttribute\_structuredClassifier [Association]

### Diagrams

[Structured Classifiers](#)

## Generalizations

[A\\_role\\_structuredClassifier](#)

## Owned Ends

- structuredClassifier : [StructuredClassifier](#) [0..1]{subsets [NamedElement::namespace](#), subsets [A\\_attribute\\_classifier::classifier](#), redefines [A\\_role\\_structuredClassifier::structuredClassifier](#)} (opposite [StructuredClassifier::ownedAttribute](#))

## A\_ownedConnector\_structuredClassifier [Association]

### Diagrams

[Structured Classifiers](#)

### Owned Ends

- structuredClassifier : [StructuredClassifier](#) [0..1]{subsets [Feature::featuringClassifier](#), subsets [NamedElement::namespace](#), subsets [RedefinableElement::redefinitionContext](#)} (opposite [StructuredClassifier::ownedConnector](#))

## A\_ownedEnd\_owningAssociation [Association]

### Diagrams

[Associations](#), [Properties](#)

### Member Ends

- [Association::ownedEnd](#)
- [Property::owningAssociation](#)

## A\_ownedOperation\_class [Association]

### Diagrams

[Classes](#), [Operations](#)

### Member Ends

- [Class::ownedOperation](#)
- [Operation::class](#)

## A\_ownedPort\_encapsulatedClassifier [Association]

### Diagrams

[Encapsulated Classifiers](#)

### Owned Ends

- encapsulatedClassifier : [EncapsulatedClassifier](#) [0..1]{subsets [A\\_ownedAttribute\\_structuredClassifier::structuredClassifier](#)} (opposite [EncapsulatedClassifier::ownedPort](#))

## A\_ownedReception\_class [Association]

### Diagrams

[Classes](#)

### Owned Ends

- class : [Class](#) [0..1]{subsets [Feature::featuringClassifier](#), subsets [NamedElement::namespace](#)} (opposite [Class::ownedReception](#))

## A\_packagedElement\_component [Association]

### Diagrams

[Components](#)

### Owned Ends

- component : [Component](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [Component::packagedElement](#))

## A\_partWithPort\_connectorEnd [Association]

### Diagrams

[Encapsulated Classifiers](#)

### Owned Ends

- connectorEnd : [ConnectorEnd](#) [0..\*] (opposite [ConnectorEnd::partWithPort](#))

## A\_part\_structuredClassifier [Association]

### Diagrams

[Structured Classifiers](#)

### Owned Ends

- structuredClassifier : [StructuredClassifier](#) [0..1] (opposite [StructuredClassifier::part](#))

## A\_protocol\_port [Association]

### Diagrams

[Encapsulated Classifiers](#)

### Owned Ends

- port : [Port](#) [0..\*] (opposite [Port::protocol](#))

## A\_provided\_component [Association]

### Diagrams

[Components](#)

### Owned Ends

- component : [Component](#) [0..\*] (opposite [Component::provided](#))

## A\_provided\_port [Association]

### Diagrams

[Encapsulated Classifiers](#)

### Owned Ends

- port : [Port](#) [0..\*] (opposite [Port::provided](#))

## A\_realization\_abstraction\_component [Association]

### Diagrams

[Components](#)

### Member Ends

- [Component::realization](#)
- [ComponentRealization::abstraction](#)

## A\_realizingClassifier\_componentRealization [Association]

### Diagrams

[Components](#)

### Owned Ends

- componentRealization : [ComponentRealization](#) [0..\*]{subsets [NamedElement::clientDependency](#)} (opposite [ComponentRealization::realizingClassifier](#))

## A\_redefinedConnector\_connector [Association]

### Diagrams

[Structured Classifiers](#)

### Owned Ends

- connector : [Connector](#) [0..\*]{subsets [A\\_redefinedElement\\_redefinableElement::redefinableElement](#)} (opposite [Connector::redefinedConnector](#))

## A\_redefinedPort\_port [Association]

### Diagrams

[Encapsulated Classifiers](#)

### Owned Ends

- port : [Port](#) [0..\*]{subsets [A\\_redefinedProperty\\_property::property](#)} (opposite [Port::redefinedPort](#))

## A\_required\_component [Association]

### Diagrams

[Components](#)

### Owned Ends

- component : [Component](#) [0..\*] (opposite [Component::required](#))

## A\_required\_port [Association]

### Diagrams

[Encapsulated Classifiers](#)

### Owned Ends

- port : [Port](#) [0..\*] (opposite [Port::required](#))

## A\_roleBinding\_collaborationUse [Association]

### Diagrams

[Collaborations](#)

### Owned Ends

- collaborationUse : [CollaborationUse](#) [0..1]{subsets [Element::owner](#)} (opposite [CollaborationUse::roleBinding](#))

## A\_role\_structuredClassifier [Association]

### Diagrams

[Structured Classifiers](#)

### Specializations

[A\\_ownedAttribute\\_structuredClassifier](#)

### Owned Ends

- structuredClassifier : [StructuredClassifier](#) [0..\*]{subsets [A\\_member\\_memberNamespace::memberNamespace](#)} (opposite [StructuredClassifier::role](#))

## A\_superClass\_class [Association]

### Diagrams

[Classes](#)

### Owned Ends

- class : [Class](#) [0..\*]{subsets [A\\_general\\_classifier::classifier](#)} (opposite [Class::superClass](#))

### A\_type\_collaborationUse [Association]

#### Diagrams

[Collaborations](#)

### Owned Ends

- collaborationUse : [CollaborationUse](#) [0..\*] (opposite [CollaborationUse::type](#))

### A\_type\_connector [Association]

#### Diagrams

[Structured Classifiers](#)

### Owned Ends

- connector : [Connector](#) [0..\*] (opposite [Connector::type](#))

# 12 Packages

## 12.1 Summary

Packages provide the main generic structuring and organizing capability of UML. There are specializations for Models and for Profiles which organize extensions to UML.

## 12.2 Packages

### 12.2.1 Summary

This sub clause provides the specification for Packages and Models.

### 12.2.2 Abstract Syntax

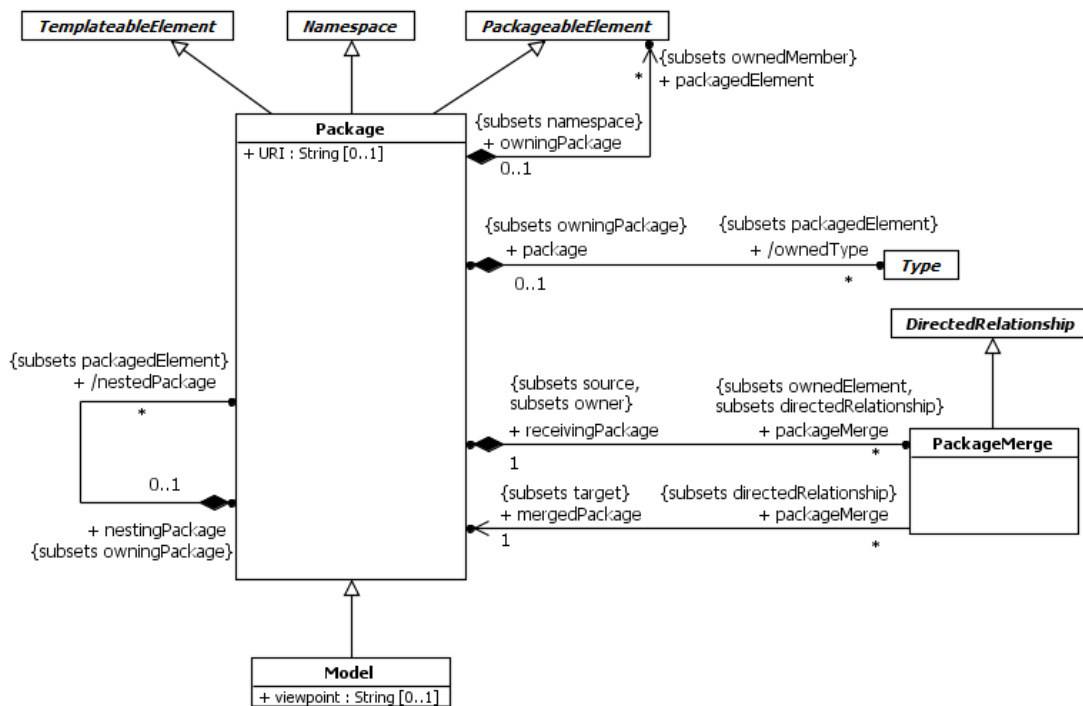


Figure 12.1 Packages

### 12.2.3 Semantics

#### Package

A Package is a namespace for its members, which comprise those elements associated via packagedElement (which are said to be *owned* or *contained*), and those *imported*.

A Package definition can extend the contents of other Packages through the *merging* of the contained elements.



A Package may be defined as a template and bound to other templates: see sub clause [7.3](#), Templates, for further information.

The URI can be specified to provide a unique identifier for a Package. Within UML there is no predetermined usage for this, with the exception of profiles (see sub clause 12.3.3). It may, for example, be used by model management facilities for model identification. The URI should hence be unique and unchanged once assigned. There is no requirement that the URI be dereferenceable (though this is of course permitted).

## PackageMerge

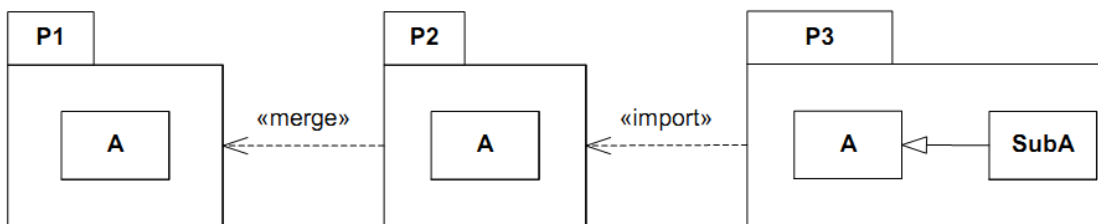
A PackageMerge is a directed relationship between two Packages that indicates that the contents of the target mergedPackage are combined into the source receivingPackage according to a set of rules defined below. It is very similar to Generalization in the sense that the source element conceptually adds the characteristics of the target element to its own characteristics resulting in an element that combines the characteristics of both. Just as a subclass is not normally depicted with its inherited features, a receiving Package is not normally depicted with the merged elements from its mergedPackages. In terms of model semantics, there is no difference between a model with explicit PackageMerges, and a model in which all the merges have been performed. Likewise XMI files containing PackageMerge are semantically equivalent to the same XMI files with the PackageMerges expanded.

Also, as with Generalization, a Package may not merge itself (directly or indirectly).

This capability is designed to be used when elements defined in different Packages have the same name and are intended to represent the same concept. A given base concept may be merged for different purposes, with each purpose defined in a separate receiving Package. By selecting different receiving packages, it is possible to obtain a custom definition of a concept for a specific end.

Thus, any reference to a model element contained in the receiving Package implies a reference to the results of the merge rather than to the increment that is contained in that Package. This is illustrated by the example in Figure 12.2 in which Package P2 defines an increment of Class A originally defined in P1. Package P2 merges the contents of Package P1, which implies the merging of P1::A into increment P2::A. Package P3 defines a subclass of P2::A called SubA. In this case, element A in Package P2 (P2::A) represents the *result* of the merge of P1::A into P2::A and not just the increment P2::A.

**NOTE.** If another package were to import P1, then a reference to A in the importing package would represent P1::A rather than the A resulting from merge.



**Figure 12.2 Illustration of the Meaning of Package Merge**

A PackageMerge can be viewed as an operation (that is itself a set of transformations) whereby the contents of the Package to be merged are combined with the contents of the receiving Package. In cases in which certain elements in the two Packages match (according to defined rules), their contents are (conceptually) merged into a single resulting element according to the formal rules of PackageMerge specified below. This operation is akin to “copying down” the features of superclasses into a subclass: the fully expanded subclass is the equivalent to the resulting package.

To understand the rules of PackageMerge, it is necessary to clearly distinguish between three distinct entities: the mergedPackage (e.g., P1 in Figure 12.2), the receivingPackage (e.g., P2), and the result of the merge transformations (also P2). The receivingPackage also plays the role of resultingPackage. This dual interpretation of the same model element can be confusing, so it is useful to introduce the following terminology that aids understanding:

- *merged package* - the package that is to be merged into the receiving package (this is the package that is the target of the merge arrow in the diagrams).
- *receiving package* - the package that, conceptually, contains the results of the merge (and which is the source of the merge arrow in the diagrams). However, this term is used to refer to the package and its contents *before* the merge transformations have been performed.
- *resulting package* - the package that, conceptually, contains the results of the merge. In the model, this is, of course, the same package as the receiving package, but this particular term is used to refer to the package and its contents *after* the merge has been performed.
- *merged element* - refers to a model element that exists in the merged package.
- *receiving element* - is a model element in the receiving package. If the element has a matching (as defined below) merged element, the two are combined to produce the resulting element (see below). This term is used to refer to the element *before* the merge has been performed.
- *resulting element* - is a model element in the resulting package *after* the merge was performed. For receiving elements that have a matching merged element, this is the combined element *after* the merge was performed. For merged elements that have no matching receiving element, this is the same as the merged element. For receiving elements that have no matching merged element, this is the same as the receiving element.
- *element type* - refers to the type of any kind of TypedElement, such as the type of a Parameter or StructuralFeature.
- *element metatype* - is the MOF type of a model element (e.g., Classifier, Association, Feature).

This terminology is based on a conceptual view of PackageMerge that is represented by the schematic diagram in Figure 12.3 (NB: this is not a UML diagram). The packagedElements (direct and indirect) of Packages A and B are all incorporated into the namespace of Package B'. However, it is important to emphasize that this view is merely a convenience for describing the semantics of PackageMerge and is not reflected in the *stored* model, that is, the *physical* model itself is not transformed in any way by the presence of PackageMerges.

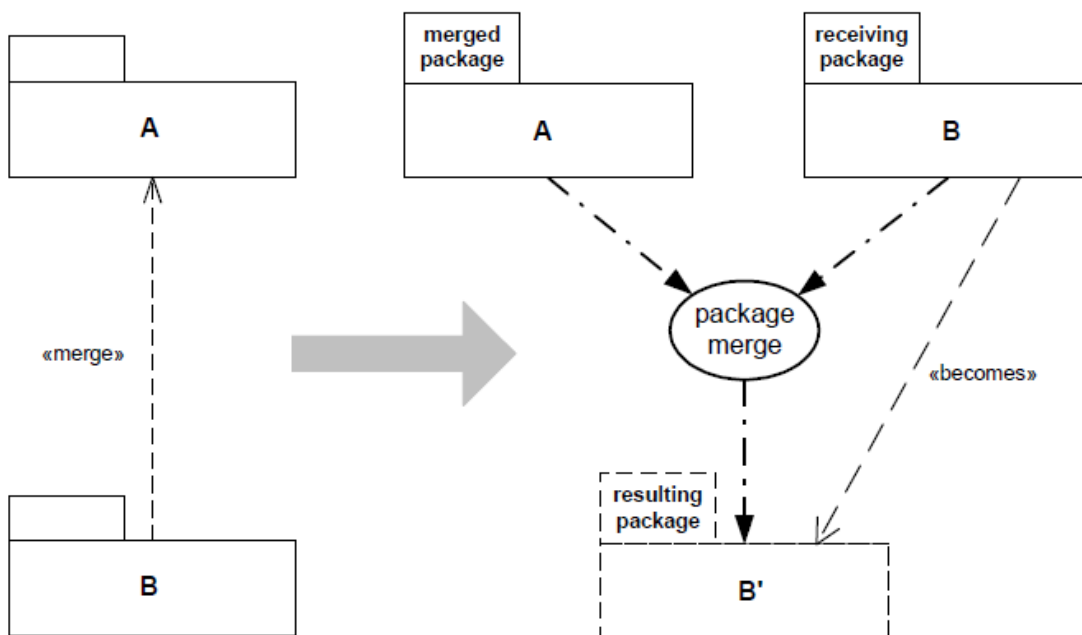


Figure 12.3 Conceptual View of the Package Merge Semantics

The semantics of PackageMerge are defined by a set of constraints and transformations. The constraints specify the preconditions for a valid PackageMerge, while the transformations describe its semantic effects (i.e., postconditions). If any constraints are violated, the PackageMerge is ill-formed and the model that contains it is invalid. Different element metatypes have different semantics, but the general principle is always the same: a resulting element will not be any less capable than it was prior to the merge: meaning, for instance, that the resulting navigability, multiplicity, visibility, etc. of a receiving model element will not be reduced as a result of a PackageMerge. One of the key consequences of this is that model elements in the resulting Package are compatible extensions of the corresponding elements in the (unmerged) receiving package.

In this specification, explicit merge transformations are only defined for certain general element metatypes found mostly in metamodels (Packages, Classes, Associations, Properties, etc.), as the semantics of merging other kinds of element metatypes (e.g., state machines, interactions) are complex and domain specific. Elements of all other kinds of metatypes are transformed according to the *default rule*: they are simply deep copied into the resulting package. (This rule can be superseded for specific metatypes through profiles or other kinds of language extensions.)

### General Package Merge Rules

A merged element and a receiving element *match* if they satisfy the matching rules for their metatype.

#### CONSTRAINTS:

1. There can be no cycles in the «merge» directed graph.
2. A Package cannot merge a Package in which it is contained (via owningPackage – direct or indirect).
3. A Package cannot merge a Package that it contains (via packagedElement – direct or indirect).
4. A merged element whose metatype is not a kind of Package, Class, DataType, Property, Association, Operation, Constraint, Enumeration, or EnumerationLiteral cannot have a receiving element with the same name and metatype unless that receiving element is an exact copy of the merged element (i.e., they are the same).
5. A PackageMerge is valid if and only if all the constraints (in this clause) required to perform the merge are satisfied.
6. Matching typed elements (e.g., Properties, Parameters) must have conforming types. For types that are Classes or Datatypes, a conforming type is either the same type or a common supertype. For all other cases, conformance means that the types must be the same.
7. A receiving element cannot have explicit references to any merged element.
8. Any redefinitions associated with matching RedefinableElements must not be conflicting.

#### TRANSFORMATIONS:

1. (*The default rule*) Merged or receiving elements for which there is no matching element are deep copied into the resulting package.
2. The result of merging two elements with matching names and metatypes that are exact copies of each other is the receiving element.
3. Matching elements are combined according to the transformation rules specific to their metatype and the results included in the resulting Package.
4. All type references to typed elements that end up in the resulting package are transformed into references to the corresponding resulting TypedElements (i.e., not to their respective increments).
5. For all matching elements: if both matching elements have **private** visibility, the resulting element will have **private** visibility; otherwise, the resulting element will have **public** visibility.

6. For all matching Classifier elements: if both matching elements have `isAbstract = true`, the resulting element has `isAbstract = true`; otherwise, the resulting element has `isAbstract = false`.
7. For all matching Classifier elements: if both matching elements has `isFinalSpecialization = true`, the resulting element has `isFinalSpecialization = true`; otherwise, the resulting element has `isFinalSpecialization = false`.
8. For all matching elements: if both matching elements are not derived, the resulting element is also not derived; otherwise, the resulting element is derived.
9. For all matching MultiplicityElements: the lower bound of the resulting element is the lesser of the lower bounds of the matching elements.
10. For all matching MultiplicityElements: the upper bound of the resulting element is the greater of the upper bounds of the matching elements.
11. Any stereotypes applied to a model element in either a merged or receiving element are also applied to the corresponding resulting element.
12. For matching RedefinableElements: different redefinitions of matching RedefinableElements are all applied to the resulting element.
13. For matching RedefinableElements: if both matching elements have `isLeaf = true`, the resulting element also has `isLeaf = true`; otherwise, the resulting element has `isLeaf = false`.

## Package Rules

Elements that are kinds of Package match by name and metatype

CONSTRAINTS:

1. All Classifiers in the merged Package must have a non-empty `qualifiedName` and have `isDistinguishableFrom() = true` in the merged Package.
2. All Classifiers in the receiving Package must have a non-empty `qualifiedName` and have `isDistinguishableFrom() = true` in the receiving Package.

TRANSFORMATIONS:

1. A `nestedPackage` from the merged Package is transformed into a `nestedPackage` with the same name and contents in the resulting Package, unless the receiving Package already contains a `nestedPackage` that matches. In the latter case, the merged `nestedPackage` is recursively merged with the matching receiving `nestedPackage`.
2. An `ElementImport` which is an `elementImport` of the receiving Package is transformed into a corresponding `ElementImport` in the resulting Package. Imported elements are not merged (unless there is also a `PackageMerge` to the Package owning the imported element).

## Class and DataType Rules

Elements that are kinds of Class or DataType match by name and metatype.

TRANSFORMATIONS:

1. All Properties that are `ownedAttributes` of the merged Classifier are merged with the receiving Classifier to produce the resulting Classifier according to the Property transformation rules specified below.
2. `nestedClassifiers` are merged recursively according to the same rules.

## Property Rules

Elements that are kinds of Property match by name and metatype.

### CONSTRAINTS:

1. The value of `isStatic` of matching Properties must be the same.
2. The value of `isUnique` of matching Properties must be the same.
3. Any Constraints associated with matching Properties must not be conflicting.

### TRANSFORMATIONS:

1. For merged Properties that do not have a matching receiving Property, the resulting Property is a Property in the resulting Classifier that is the same as the merged Property.
2. For merged Properties that have a matching receiving Property, the resulting Property is a Property with the same name and characteristics except where these characteristics are different. Where these characteristics are different, the resulting Property characteristics are determined by application of the appropriate transformation rules.
3. For matching Properties: if both Properties have `isReadOnly = true`, the resulting Property also has `isReadOnly = true`; otherwise, the resulting Property has `isReadOnly = false`.
4. For matching Properties: if both Properties have `isOrdered = false`, then the resulting Property also has `isOrdered = false`; otherwise, the resulting Property has `isOrdered = true`.
5. For matching Properties: if neither Property is designated as a subset of some derived union, then the resulting Property will not be designated as a subset; otherwise, the resulting Property will be designated as a subset of that derived union.
6. For matching Properties: different Constraints of matching Properties are all applied to the resulting Property.
7. For matching Properties: if either the merged and/or receiving elements have `isUnique = false`, the resulting element has `isUnique = false`; otherwise, the resulting element has `isUnique = true`.
8. The value of `type` for the resulting Property is transformed to refer to the corresponding type in the resulting Package.

## Association Rules

Elements that are kinds of Association match by name and metatype.

### CONSTRAINTS:

1. These rules only apply to binary Associations. (For merging n-ary associations the *default rule* is used)
2. The receiving association end must have `aggregation = composite` if the matching merged association end has `aggregation = composite`.
3. The receiving association end must be owned by the Association if the matching merged association end is owned by the Association.

### TRANSFORMATIONS:

1. A merge of matching Associations is accomplished by merging the Association classifiers (using the merge rules for Classifiers) and merging their corresponding `ownedEnd` Properties according to the rules for Properties and the following rule for association ends.

2. For matching association ends: if neither association end is in `ownedNavigableEnd`, then the resulting association end is also not in `ownedNavigableEnd`. In all other cases, the resulting association end is in `ownedNavigableEnd`.

### Operation Rules

Elements that are kinds of Operation match by name, Parameter order, and Parameter types, not including any return type.

CONSTRAINTS:

1. Operation Parameters and their types must conform to the same rules for type and multiplicity as were defined for Properties.
2. The receiving Operation must have `isQuery = true` if the matching merged Operation has `isQuery = true`.

TRANSFORMATIONS:

1. For merged Operations that do not have a matching receiving Operation, the resulting Operation is an Operation with the same name and signature in the resulting classifier.
2. For merged Operations that have a matching receiving Operation, the resulting Operation is the outcome of a merge of the matching merged and receiving Operations, with Parameter transformations performed according to the Property transformations defined above.

### Enumeration Rules

Elements that are kinds of EnumerationLiteral match by owning Enumeration and Literal name.

CONSTRAINTS:

1. Matching EnumerationLiterals must be in the same order.

TRANSFORMATIONS:

1. Non-matching EnumerationLiterals from the merged Enumeration are included in the receiving Enumeration.

### Constraint Rules

CONSTRAINTS:

1. Constraints must be mutually non-contradictory.

TRANSFORMATIONS:

1. The Constraints of the merged model elements are all added to the Constraints of the matching receiving model elements.

### Model

A Model is a description of a system, where ‘system’ is meant in the broadest sense and may include not only software and hardware but organizations and processes. It describes the system from a certain *viewpoint* (or vantage point) for a certain category of *stakeholders* (e.g., designers, users, or customers of the system) and at a certain level of abstraction. A Model is complete in the sense that it covers the whole system, although only those aspects relevant to its purpose (i.e., within the given level of abstraction and viewpoint) are represented in the Model.

As a Package, a Model has a set of members that together describe the system being modeled. The organization of these elements varies by the modeling method being used. One approach is one or more composition hierarchies where a top-most

Package/Component represents the boundary of the system. A Model may also contain elements describing relevant parts of the system's environment. The environment is typically modeled by Actors and their Interfaces. As these are external to the system, they reside outside the Package/Component hierarchy. They may be collected in a separate Package, or owned directly by the Model as packagedElements.

Different Models can be defined for the same system, where typically the different Models are complementary and defined from the perspectives (viewpoints) of different system stakeholders. With composition of Models, a container model represents a comprehensive view of the system given by the different views defined by the contained Models.

Models can have Abstraction Dependencies between them: refinement (stereotyped by «refine» from the Standard Profile) or mapping (for example stereotyped by «trace» from the Standard Profile). These are typically represented in more detail by Dependencies between the elements contained in the Models. Relationships between elements in different Models generally no direct impact on the contents of the Models because each Model is meant to be complete. However, they are useful for tracing refinements and for keeping track of cross-references between models.

### 12.2.4 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: collectively this represents a 'folder icon.' 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 Package.

Conformant tools may restrict the use of these notations to packagedElements. Optionally, elements that become available for use in an importing Package through a PackageImport or an ElementImport may have a distinct color or be dimmed to indicate that they are not packagedElements.

- 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 packagedElement may be indicated by preceding the name by a visibility symbol ('+' for **public** and '-' for **private**). Packages may not have **protected** or **package** visibility.

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 members need not necessarily show all its members; it may show a subset of the members according to some criterion.

The *URI* for a Package may be indicated with the text {uri = <uri>} following the Package name.

A PackageMerge is shown using a dashed line with an open arrowhead pointing from the receivingPackage (the source) to the mergedPackage (the target). In addition, the keyword «merge» is shown near the dashed line.

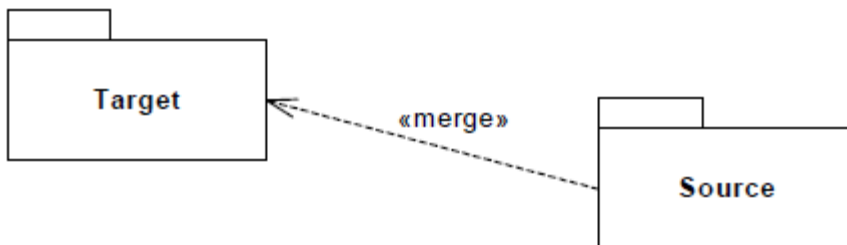


Figure 12.4 Notation for Package Merge

A Model is notated using the ordinary Package symbol (a folder icon) with a small triangle in the upper right corner of the large rectangle.

Optionally, especially if the members of the Model are shown within the large rectangle, the triangle may be drawn to the right of the Model name in the tab.

A Model may also be notated as a Package, using the ordinary Package symbol with the keyword «model» placed above the name of the Model.

### 12.2.5 Examples

There are three alternative representations of the same Package named **Types** in Figure 12.5. 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 rectangle (and also its URI), and the one to the right shows some of the members using the alternative ownership notation.

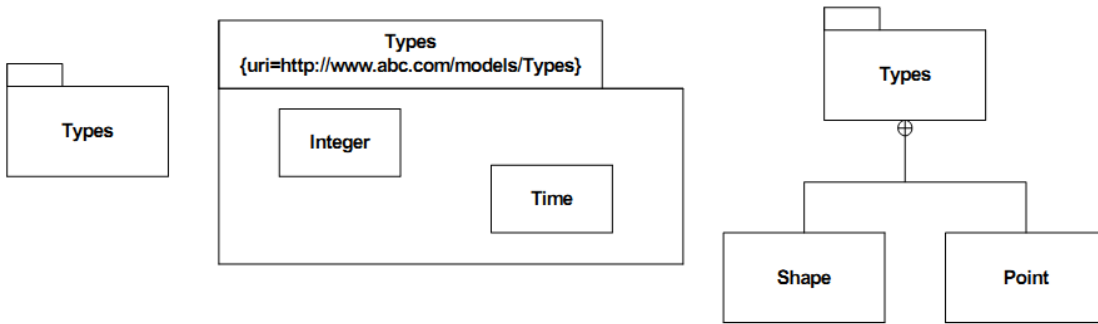
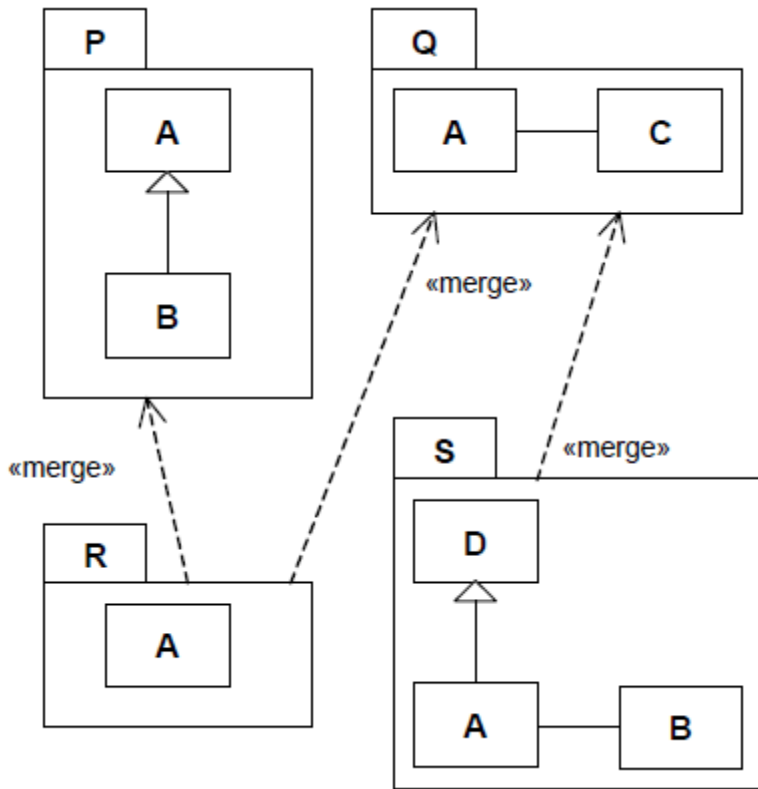


Figure 12.5 Examples of a Package with Members

In Figure 12.6, packages P and Q are being merged by package R, while package S merges only package Q.

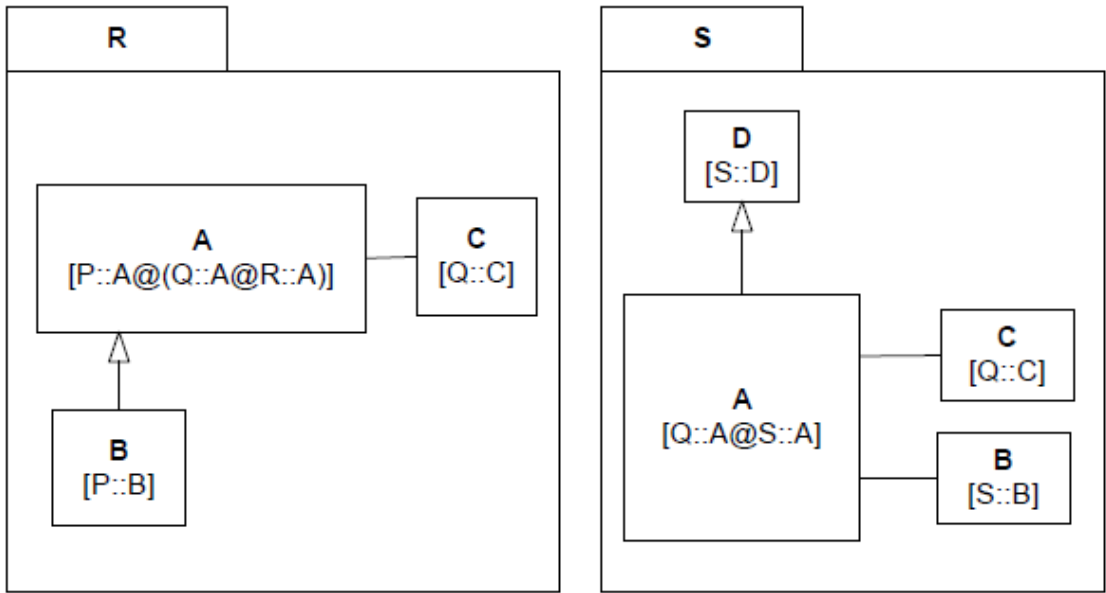




**Figure 12.6 Simple Example of Package Merge**

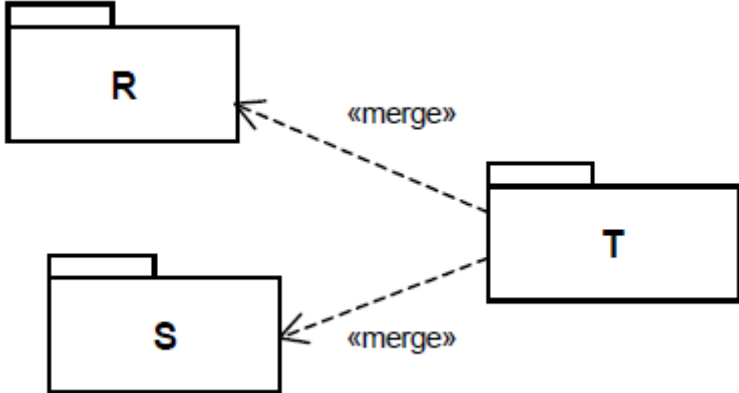
The conceptually resulting packages R and S are shown in Figure 12.7. The expressions in square brackets indicate which individual elements were merged to produce the final result, with the “@” character denoting the conceptual merge ‘transformation’ as an operator, where X@Y signifies the resulting element from the merge transformation applied to matching receiving element X and merged element Y.

**NOTE.** These expressions are not part of the standard notation, but are included here for explanatory purposes.



**Figure 12.7 Simple Example of Transformed Packages Following the Merges**

In Figure 12.8, additional PackageMerges are introduced by having Package T, which has no packagedElements of its own, merge Packages R and S defined previously.



**Figure 12.8 Introducing Additional Package Merges**

In Figure 12.9, the conceptually resulting Package T is depicted. In this Package, the definitions of A, B, C, and D have all been brought together.

**NOTE.** The types of the ends of the Associations that were originally in the packages Q and S have all been updated to refer to the appropriate elements in Package T.

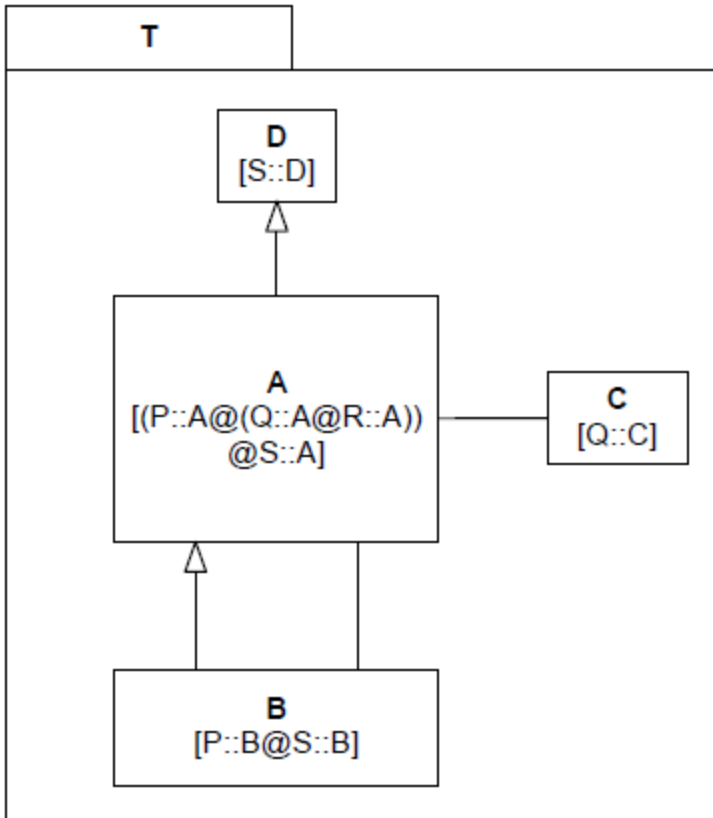


Figure 12.9 Result of the Additional Package Merges

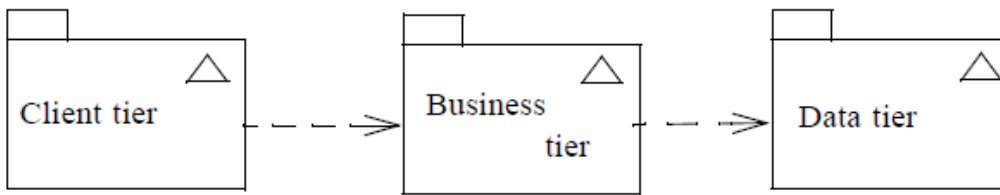


Figure 12.10 Three Models Representing Parts of a System

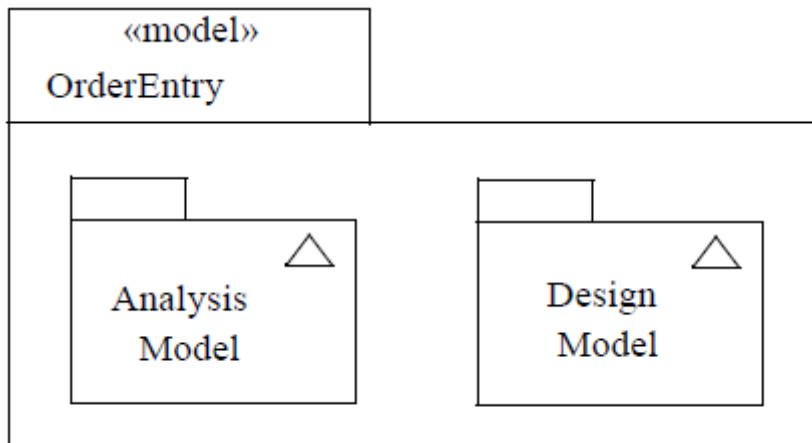


Figure 12.11 Two Views of One System Collected in a Container Model

## 12.3 Profiles

### 12.3.1 Summary

The Profiles clause describes capabilities that allow metaclasses 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 Service Oriented Architecture). The Profiles clause is consistent with the OMG Meta Object Facility (MOF).

#### Positioning Profiles versus Metamodels, MOF and UML

UML is reused at several meta-levels in various OMG specifications that deal with modeling. For example, MOF uses it to provide the ability to model metamodels. This clause deals with use cases comparable to the MOF at the meta-meta-level, which is one level higher than the rest of the superstructure specification. In order to allow this, the reference metamodel must be defined as an instance of UML that corresponds to its definition using MOF. Thus when defining a UML profile, the profile's stereotypes are defined to extend the UML classes in the normative version of the UML metamodel whose XMI serialization is referenced in [Annex E](#).

Profiles are not a first-class extension capability (i.e., it does not allow for creating new metamodels). Rather, the intention of Profiles is to give a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method. Each such adaptation is grouped in a Profile. It is not possible to remove any of the Constraints that apply to 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 this Profiles clause; 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 at the metamodel level: it is possible to add subclasses and associations as necessary.

There are several reasons why you may want to extend UML:

- Give a terminology that is adapted to a particular platform or domain (for example 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 additional semantics to UML or specific metaclasses.
- Add types that do not exist in UML (such as defining a timer, clock, or continuous time).
- Add Constraints that restrict the way UML's constructs are used (such as such as disallowing multiple inheritance).
- 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).

There is no simple answer for when to create a new metamodel, when to create a new profile, and when to create both (one for UML tooling, the other for MOF-based tooling).

### 12.3.2 Abstract Syntax

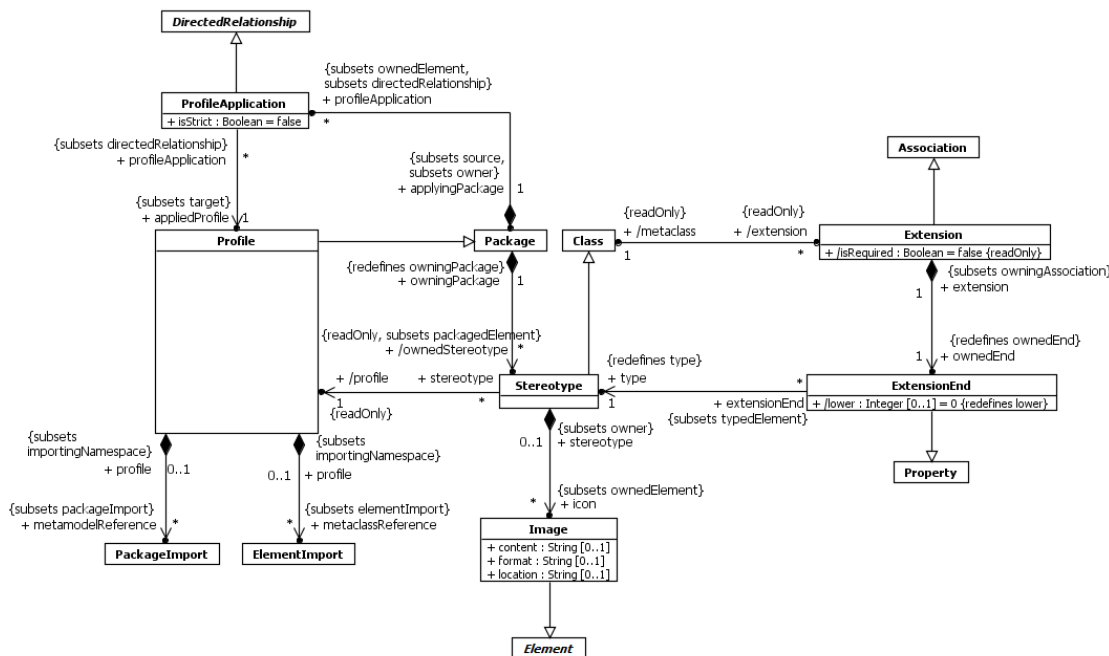


Figure 12.12 Profiles

### 12.3.3 Semantics

#### Profiles

A Profile is a restricted form of metamodel that can be used to extend UML, as described below. The primary extension construct is the Stereotype.

#### Restricting Availability of UML Elements

The metaclassReference **ElementImports** and metamodelReference **PackageImports** may be used to specify the Profile's filtering rules. The filtering rules determine which UML elements are *available* when the Profile is applied and which ones are *hidden*.

**NOTE.** Applying a Profile to a model does not change that model in any way; it merely defines a view of the underlying model.

The effects of a metaclass being hidden (not available) are as follows:

- It is not possible to create new instances of that metaclass (or its subclasses)
- Existing instances of that metaclass (or its subclasses) can no longer be seen in diagrams or selected in lists, including browser panes
- Relationships with existing instances of that metaclass (or its subclasses) can no longer be seen in diagrams or selected in lists, including browser panes

Tools may vary in how they implement the above – for example they may hide the metaclass/instances completely in selection lists or make them grayed out/unselectable .

In order for the filtering rules (described further below) on a Profile to be activated, the Profile must be applied in *Strict* mode: specifically the `isStrict` attribute on the `ProfileApplication` must be set to true; otherwise the filtering rules are ignored for this profile application.

The most common case is when a Profile imports UML itself using a `metamodelReference`. A conformant tool may provide this as built-in behavior when the user creates a Profile. In that case, every UML metaclass is available. Alternatively, specific metaclasses could be referenced through `metaclassReferences` and only those would then be available. A further option is to use one or more `metamodelReferences` to `Package(s)` that contain `ElementImports` for a subset of UML metaclasses. This allows the set to be reusable across many Profiles without having to specify individual `metaclassReferences` each time.

The visibility and alias properties of `ElementImports` are ignored when it is used as a `metaclassReference`.

Where both a `metaclassReference` and a `metamodelReference` are present on a profile, the latter is ignored and only the specific metaclasses are available.

In detail, the following rules are used to determine whether a model element is available after a Profile has been applied in *Strict* mode. Metaclasses and their instances are available if they are:

1. referenced by an explicit `metaclassReference`, or
2. (in the absence of a `metaclassReference`) members (directly or transitively) of a `Package` that is referenced by an explicit `metamodelReference`, or
3. extended by a `Stereotype` which is a member of the applied profile (even if the extended metaclass itself is not available).

All other model elements are *hidden* (not available) when the Profile is applied in *Strict* mode.

This makes invalid the combination of applied profiles that specify non-overlapping (disjoint) sets of available metaclasses.

If a Profile P1 imports another Profile P2, then all `metaclassReference` and `metamodelReference` associations will be combined at the P1 level, and the filtering rules apply to this union. Stereotypes imported from another Profile using `ElementImport` or `PackageImport` are added to the namespace members of the importing profile.`Profile Contents`

A Profile can define `Classes`, `Associations`, `DataTypes`, `PrimitiveTypes` and `Enumerations` as well as `Stereotypes`. More precisely all the constraints of a CMOF-compliant metamodel apply to a UML Profile. These are defined in detail in Section 14.3 of the MOF Core Specification.

Apart from `Stereotypes`, `Types` defined in a Profile can only be used as the type of `Properties` in the Profile, they cannot be used as `Types` in models the Profile is applied to as they apply at the meta-model level, not the model level. It is however possible to define these types in separate `Packages` and import them as needed in both Profiles and model `Packages` in order to use them for both purposes.

Stereotypes can participate only in binary Associations. The opposite class can be another Stereotype, a non-Stereotype Class that is a packagedElement of a Profile (directly or indirectly), or a UML metaclass. For these Associations there must be an ownedAttribute Property typed by the opposite class. Where the opposite class is not a stereotype, the opposite property must be an ownedMember of the Association itself rather than the other class/metaclass. The effect of these rules is that Associations in a Profile are not required to involve a Stereotype but may not own both of their Ends.

The most direct implementation of the Profile capability that a tool can provide is by having a metamodel based implementation, similar to the Profile metamodel. However, this is not a requirement of the current standard, which requires only the support of the specification, and the standard XMI based interchange capacities. The Profile capability has been designed to be implementable by tools that do not have a metamodel-based implementation. Practically any mechanism used to attach new values to model elements can serve as a valid profile implementation. As an example, the UML1.4 profile metamodel could be the basis for implementing a UML2-compliant profile tool.

### Integrating and Extending Profiles

There is a number of ways to create, extend, and integrate Profiles. These are described briefly in this section in order to foster better profile integration and reuse.

The simplest form of Profile integration is to simply apply multiple Profiles to the same Package. This requires no integration between the Profiles at all. Such Profiles might be designed to complement each other, addressing different concerns.

It is also possible for one Profile to reuse all of or parts of another, and to extend other Profiles. Like any other Class, Stereotypes can be defined in Packages or Profiles that can be factored for reuse. These Stereotypes can be directly reused by being referenced or specialized in other Profiles. Normal rules apply as to whether a referenced Stereotype is visible to users of the extending Profile.: a public import is needed to ensure that Stereotypes from other profiles are visible after applying the extending one.

For example, the *Unified Profile for DoDAF and MODAF* (UPDM) Profile could integrate with the SysML Profile to reuse Stereotypes such as Requirement and ViewPoint. UPDM could be designed to use ViewPoint in a manner that is semantically consistent with SysML. However UPDM could extend ViewPoint with additional properties and associations for its purposes. The UPDM specification could note to users that ViewPoint is a stereotype in UPDM that represents a "placeholder" to ViewPoint in SysML. Users could then apply UPDM to a model, and get UPDM's ViewPoint capabilities without any coupling with, or need for SysML. UPDM could then provide another compliance point that merges with the SysML profile resulting in stereotypes Requirement and ViewPoint having the capabilities of both profiles. The SysML::ViewPoint would be merged with the UPDM::ViewPoint allowing the shared semantics to be supported without making any changes to the existing model. Users who want UPDM with SysML would then apply this merged profile.

### MOF-Equivalent Semantics

The following section specifies the semantics of Stereotypes and their instances using MOF. That does not mean that tools need implement Profiles using MOF, but that a non-MOF-based implementation must do whatever is necessary under the covers to ensure it behaves, in all observable ways, as if it were a MOF implementation.

The same mapping to MOF is used to determine how to serialize applied profiles using XMI. A Profile is an instance of the UML2 metamodel, not a CMOF metamodel. Therefore the MOF to XMI mapping rules do not directly apply for instances of a Profile. Figure 12.15 is an example of a mapping between a UML2 Profile and an equivalent CMOF model. This mapping is used as a means to explain and formally specify how Profiles are serialized and exchanged as XMI. Using the following Profile to CMOF mapping rules, the XMI specification can be used to determine how Profiles, and models with Profiles applied, are represented in XMI. In the mapping:

- A Profile maps to a CMOF Package.
- A Stereotype maps to a CMOF class with the same name and properties.
- A Metaclass is already a CMOF class so it maps to itself.
- An Extension maps to an Association as described in the Semantics sub clause of Extension.

- Any other elements in the Profile (e.g., non-Stereotype Classes) are treated as MOF elements.
- An instance of a Stereotype (created when the Stereotype is applied to an Element) maps to an instance of the CMOF class representing the Stereotype. It is associated with the Element to which it applies using a Link which is an instance of the Association to which the Extension is mapped.

For a Profile the *URI* Property (inherited from Package) is used to determine the nsURI to be used to identify instances of the Profile in XMI.

**NOTE.** By default the name attribute of the Profile is used for the nsPrefix in XMI but this can be overridden by the CMOF tag `org.omg.xmi.nsPrefix`.

OMG normative Profiles, such as the UML Standard Profile, follow an OMG normative naming scheme for URIs. For non-standard profiles a recommended convention is:

`nsUri = http://<profileParentQualifiedName>/<version>/<profileName>.xmi`

`nsPrefix = <profileName>`

where:

- `<profileParentQualifiedName>` is the qualified name of the Package containing the Profile (if any) with / (forward slash) substituted for ::, and all other illegal XML QName characters removed.
- `<version>` is a version identifier.

**NOTE.** For OMG normative profiles this is a date in the format YYYYMMnn where nn is a serial number within the month, and represents the version of the Profile XMI not that of the specification which might be re-issued without affecting the XMI.

- `<profileName>` is the name of the Profile.

A Profile can be exchanged just like any model, as an XMI file, and models that have a Profile applied can also be interchanged.

Figure 12.17 shows a Stereotype named Home extending the Interface UML2 metaclass. Figure 12.15 illustrates the MOF correspondence for that example, basically by introducing an Association from the Home MOF class to the Interface MOF class. For illustration purposes, we add a Property “magic:String” to the Home Stereotype.

The first serialization below shows how the model in Figure 12.17 (definition of the Profile extending the UML2 metamodel) can be exchanged.

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmlns:xmi=http://www.omg.org/spec/XMI/YYYYMMnn>
  xmlns:mofext=http://www.omg.org/spec/MOF/YYYYMMnn xmlns:uml=http://www.omg.org/spec/UML/YYYYMMnn
  <uml:Profile xmi:id="id0" xmi:type="uml:Profile" name="HomeExample">
    <metamodelReference xmi:type="uml:PackageImport" xmi:id="id2">
      <importedPackage href="http://www.omg.org/spec/UML/YYYYMMnn/UML.xmi#_0"/>
    </metamodelReference >
    <packagedElement xmi:type="uml:Stereotype" xmi:id="id3" name="Home">
      <ownedAttribute xmi:type="uml:Property" xmi:id="id5" name="base_Interface" association="id6">
        <type href="http://www.omg.org/spec/UML/YYYYMMnn/UML.xmi#Interface"/>
      </ownedAttribute>
    </packagedElement>
    <packagedElement xmi:type="uml:Extension" xmi:id="id6" name="A_Interface_Home" memberEnd="id7 id5">
      <ownedEnd xmi:type="uml:ExtensionEnd" xmi:id="id7" name="extension_Home" type="id3"
      aggregation="composite">
        </ownedEnd>
      </packagedElement>
    </uml:Profile>
```

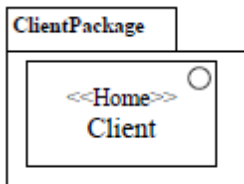


```

<mofext:Tag xmi:type="mofext:Tag" name="org.omg.xmi.nsPrefix" value="HomeExample"/>
<mofext:Tag xmi:type="mofext:Tag" name="org.omg.xmi.nsURI"
value="http://HomeExample/20120501/HomeExample.xmi"/>
</xmi:XMI>

```

Figure 12.13 is an example model that includes an instance of Interface extended by the Home stereotype.



**Figure 12.13 Using the HomeExample Profile to Extend a Model**

Now the XMI below shows how this model extended by the Profile is serialized. A tool importing that XMI file can filter out the elements related to the HomeExample Profile, if the tool does not have this Profile definition.

```

<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmlns:xmi="http://www.omg.org/spec/XMI/YYYYMMnn" xmlns:uml="http://www.omg.org/spec/UML/YYYYMMnn"
xmlns:HomeExample="http://HomeExample/20120501/HomeExample.xmi">
  <uml:Package xmi:type="uml:Package" xmi:id="id1" name="ClientPackage">
    <profileApplication xmi:type="uml:ProfileApplication" xmi:id="id3">
      <appliedProfile href="http://HomeExample/20120501/HomeExample.xmi#id0"/>
    </profileApplication>
    <packagedElement xmi:type="uml:Interface" xmi:id="id2" name="Client"/>
  </uml:Package>
  <!-- applied stereotypes -->
  <HomeExample:Home xmi:id="id4" base_Interface="id2"/>
</xmi:XMI>

```

## Defining Profiles for Non-UML Metamodels

In theory the Profiles capability can be used to define extensions for metamodels other than UML, though this capability has rarely, if at all, been used in practice. It would require any tooling implementing that metamodel to also support some kind of profile application mechanism – that is outside the scope of this specification. The following describes how the Profile definition mechanism may be used in this way.

In addition to UML, a Profile may be related to another MOF-compliant *reference metamodel*. In general 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 (directly or indirectly) of the same reference metamodel. The `metaclassReference ElementImports` and `metamodelReference PackageImports` serve two purposes: (1) they identify the reference metamodel elements that are imported by the profile and (2) they specify the Profile’s filtering rules. The filtering rules determine which elements of the metamodel are *available* when the Profile is applied and which ones are *hidden*.

**NOTE.** Applying a Profile does not change the underlying model in any way; it merely defines a view of the underlying model.

In general, only model elements that are instances of imported reference metaclasses will be visible when the profile is applied. Instances of all other metaclasses will be hidden and further instances may not be created. By default, model elements whose metaclasses are owned by the reference metamodel are visible. This applies transitively to any subpackages of the reference metamodel according to the default rules of package import. If any metaclass is imported using a `metaclassReference ElementImport`, then model elements whose metaclasses are the same as that metaclass are available. However, a `metaclassReference` blocks a `metamodelReference` whenever an element or Package of the referenced metamodel is also referenced by a metaclass reference. In such cases, only the elements that are explicitly referenced by the `metaclassReference` will be visible, while all other elements of the metamodel Package will be hidden.

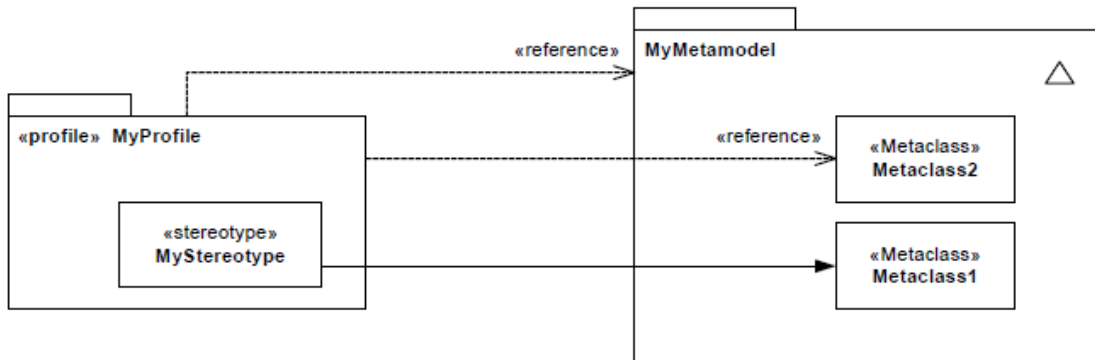
The following rules are used to determine whether a model element is available or hidden after a Profile has been applied. Model elements are *available* if they are instances of metaclasses that are:

1. referenced by an explicit metaclassReference, or
2. contained (directly or transitively) in a Package that is referenced by an explicit metamodelReference; unless there are other elements of subpackages of that Package that are explicitly referenced by a MetaclassReference, or
3. extended by a Stereotype owned by the applied profile (even if the extended metaclass itself is not visible).

All other model elements are hidden (not available) when the Profile is applied.

The most common case is when a Profile just imports an entire metamodel using a metamodelReference. In that case, every element instantiating a metaclass in the metamodel is visible.

In the example in Figure 12.14, MyMetamodel is a metamodel containing two metaclasses: Metaclass1 and Metaclass2. MyProfile is a profile that references MyMetamodel and Metaclass2. However, there is also an explicit metaclass reference to Metaclass2, which overrides the metamodel reference. An application of MyProfile to some model based on MyMetamodel will show instances of Metaclass2 (because it is referenced by an explicit metaclass reference). Also, those instances of Metaclass1 that are extended by an instance of MyStereotype will be visible. However, instances of Metaclass1 that are not extended by MyStereotype remain hidden.



**Figure 12.14 Specification of an Available Metaclass**

If a Profile P1 imports another Profile P2, then all metaclassReference and metamodelReference associations will be combined at the P2 level, and the filtering rules apply to this union.

The filtering rules defined at the Profile level are, in essence, merely a suggestion to modeling tools on what to do when a profile is applied to a model.

The isStrict attribute on a ProfileApplication specifies that the filtering rules have to be applied strictly. If isStrict is true on a ProfileApplication, then no other metaclasses than the accessible one defined by the profile shall be accessible when the Profile is applied on a model. This prohibits the combination of applied profiles that specify different accessible metaclasses.

### ProfileApplication

A ProfileApplication is used to record which Profiles have been applied to a Package.

One or more Profiles that extend UML may be applied at will to a model Package. Applying a Profile means that it is possible to apply the Stereotypes that are defined as part of the Profile. It is possible to apply multiple Profiles to a Package, though this could make the Package invalid if they have conflicting Constraints. Applying a Profile means recursively applying all its nested and imported Profiles. Stereotypes that are public members of a Profile may be applied to applicable model elements in Packages to which the Profile has been applied.

When a Profile is applied, instances of the appropriate Stereotypes must be created for those elements that are instances of metaclasses with ExtensionEnds which have isRequired = **true**. 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 Stereotypes defined in the Profile are deleted. 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.

A Profile which is a packagedElement of another Profile can be applied individually. However, the nested Profile must specify any required metaclass and/or metamodel references if it contains any Stereotypes and may use PackageImport to indicate other Profiles to be co-applied. Metaclass and/or metamodel references are not inherited from a containing Profile.

## Stereotypes

A Stereotype defines an extension for one or more metaclasses, and enables the use of specific terminology or notation in place of, or in addition to, the ones used for the extended metaclasses.

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 metaclasses through association (Extension) rather than generalization/specialization. Similarly, a metaclass may be extended by one or more Stereotypes. Relating an instance “S” of Stereotype to a metaclass “C” from UML using an “Extension” (which is a specific kind of Association) signifies that model elements of type C can be extended by an instance of “S” (see example in Figure 12.22 Defining a Stereotype). At the model level (such as in Figure 12.27) instances of “S” are related to “C” model elements (instances of “C”) by links (occurrences of the Association/Extension from “S” to “C”).

Any metaclass referenced by a metaclassReference or contained in a Package referenced by metamodelReference of the closest Profile directly or indirectly containing a Stereotype can be extended by the Stereotype. For example States, Transitions, Activities, Use Cases, Components, Properties, Dependencies, etc. can all be extended with Stereotypes if the metamodelReference is UML. A Stereotype may be contained in a Package in which case the metaclasses available for extension are those referenced by the closest parent Profile containing the Package.

Just like a Class, a Stereotype may have Properties, which have traditionally been referred to as Tag Definitions. When a Stereotype is applied to a model element, the values of the Properties have traditionally been referred to as *tagged values*. Stereotype specializes Class and its Properties have the same meaning in Stereotypes as they do in Class. Tool vendors may choose to support extensibility that includes owned operations and behaviors, but are not required to do so. Tools must however support Stereotype ownedAttributes.

Its Profile or Package defines the namespace for the Stereotype. When Profiles are applied to a Package, the available Stereotypes for use are defined by the applied Profiles, and these Stereotypes can be displayed using the fully qualified name if needed in order to distinguish Stereotypes with the same name in different Profiles or Packages. PackageImport and ElementImport can be used to allow the use of unqualified names. Stereotypes directly owned by an applied Profile (ownedStereotype) may be used without qualified names.

## Images

The Image class provides the necessary information to display an Image in a diagram. Icons are typically handled through the Image class.

Information such as physical placement or format is provided by the Image class. The Image class provides a generic way of representing images in different formats. Although some predefined values are specified for format for convenience and interoperability, the set of possible formats is open ended. However there is no requirement for a tool to be able to interpret and display any specific format, including those predefined values.

The format property indicates the format of the content, which is how the string content should be interpreted. The following values are reserved: **SVG**, **GIF**, **PNG**, **JPG**, **WMF**, **EMF**, **BMP**. In addition the prefix ‘**MIME:**’ is also reserved: this must be followed

by a valid MIME type as defined by RFC3023. This option can be used as an alternative to express the reserved values above, for example “SVG” could instead be expressed “MIME: image/svg+xml.”

## Extensions

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.

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 (meta)Class, while the latter ties the Extension to a Stereotype that extends the Class.

A required Extension (`isRequired = true`) means that an instance of this Stereotype must be linked to each instance of the extended metaclass in the model to which the containing Profile has been applied (otherwise the model is not well-formed). If the extending Stereotype has subclasses, then at most one instance of the Stereotype or one of its subclasses is required.

A non-required Extension (`isRequired = false`) means that an instance of this Stereotype may 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 stereotyped. However the same stereotype (or its subtypes) can never be applied twice to the same element. An instance of a Stereotype is deleted when either the instance of the extended metaclass is deleted, or when the Profile defining the stereotype is removed from the `appliedProfiles` of the Package.

The equivalence to a MOF construction is shown in Figure 12.15. This figure illustrates the case shown in Figure 12.17, where the Stereotype named Home extends the Interface metaclass. In this figure, Interface is an instance of the UML metaclass (a CMOF Class) and Home is an instance of a Stereotype (also considered a CMOF Class for this purpose). The MOF construct equivalent to an Extension is a composition from the extended metaclass to the extension Stereotype, owned by the extended metaclass. When the Extension is required, then the multiplicity of the property typed by the extension Stereotype is 1.

The name of the Property typed by the extended metaclass is:

*'base\_' extendedMetaclassName*

The name of the Property typed by the extension Stereotype (the ExtensionEnd) is:

*'extension\_' stereotypeName*

Constraints are frequently added to Stereotypes. The above Properties may be used for expressing OCL navigations. For example, the following OCL expression states that a Home Interface shall not have attributes:

```
self.base_Interface.ownedAttributes->isEmpty()
```

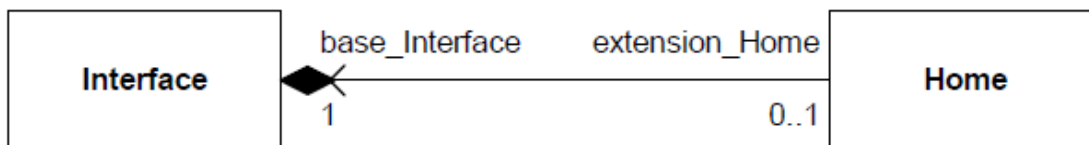


Figure 12.15 MOF Model Equivalent to Extending "Interface" by the "Home" Stereotype

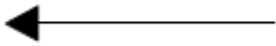
## ExtensionEnd

An ExtensionEnd is used to tie an Extension to a Stereotype when extending a metaclass: it is a `navigableOwnedEnd` of the Extension, avoiding an extra `ownedAttribute` on the extended Class. It is always typed by a Stereotype and must always have `isComposite = true`.

The default multiplicity of an ExtensionEnd is 0..1. It may be 1..1 if the Stereotype is required but the upperBound may never be more than 1.

### 12.3.4 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 they are typically elided and navigability arrows are never shown. If `isRequired = true`, the adornment **{required}** is shown near the ExtensionEnd.



**Figure 12.16 The Notation for an Extension**

It is possible to use the multiplicities 0..1 or 1 on the ExtensionEnd as an alternative to the adornment **{required}**. Due to how `isRequired` is derived, the multiplicity 0..1 corresponds to `isRequired = false`.

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` use the same notation as `Package::elementImport` and `Package::packageImport`, respectively but with the keyword `<reference>`.

ProfileApplications are shown using a dashed arrow with an open arrowhead from the Package to each applied Profile. Either the keyword `<apply>` is shown near the arrow, or the keyword `<strict>` - the latter if `isStrict = true`.

If multiple appliedProfiles have Stereotypes with the same name, it may be necessary to qualify the name of the Stereotype (with the profile name).

A Stereotype uses the same notation as a Class, with the addition that the keyword `<stereotype>` is shown before or above the name of the Class.

When a Stereotype is applied to a model element (an instance of a Stereotype is linked to an instance of a metaclass), the name of the Stereotype is shown within a pair of guillemets above or before the name of the model element, or where the name would appear if the name is omitted or not displayed. For model elements that are not NamedElements but do have a graphical representation, unless specifically stated elsewhere, the stereotypes can be displayed within a pair of guillemets near the upper right corner of the graphical representation. If multiple stereotypes are applied, the names of the applied stereotypes are shown as a comma-separated list within a pair of guillemets. When the extended model element has a keyword, then the stereotype name(s) will be displayed close to the keyword, within the same or separate guillemets (example: `<interface> <Clock>` or `<Clock, interface>`).

Normally a Stereotype's name starts with an upper-case letter, to follow the convention for naming Classes. However Profiles may use different conventions. Matching between the names of Stereotype definitions and applications is case-insensitive, so naming stereotype applications with lower-case letters where the stereotypes are defined using upper-case letters is valid, although stylistically obsolete. For legacy reasons a tool may display stereotype names with the initial letter in lower case even when defined in upper case.

A tool can choose whether it will display Stereotypes or not. In particular, tools can choose not to display *required* stereotypes, but to display only the values of their `ownedAttributes` if any.

The values of the `ownedAttributes` of a Stereotype (or its generalizations) applied to a model element can be shown in one of the following three ways:

1. As part of a comment symbol connected to the graphic node representing the model element.
2. In separate compartments of the graphic node representing that model element.
3. Above the name string within the graphic node or, else, before the name string.

In the case where a compartment or comment symbol is used, the stereotype name may be shown in guillemets before the name string in addition to being included in the compartment or comment.

The values are displayed as name-value pairs:

<namestring> '=' <valuestring>

If a Stereotype Property is multi-valued, then the <valuestring> is displayed as a comma-separated list:

<valuestring> ::= <value> [',' <value>]\*

Certain values have special display rules:

- As an alternative to a name-value pair, when displaying the values of Boolean Properties, tools may use the convention that if the <namestring> is displayed, then the value is **true**; otherwise, the value is **false**.
- If the value is the name of a NamedElement, then, optionally, the qualifiedName of that element can be displayed.

If compartments are used to display Stereotype Property values, then an additional compartment is required for each applied Stereotype whose Property values are to be displayed. Each such compartment is headed by the name of the applied stereotype in guillemets. Such compartments are only applicable to elements for which compartments generally may be used: specifically Classifiers and States.

Within a comment symbol, or, if displayed before or above the model element's name, the Property values from a specific Stereotype are optionally preceded with the name of the applied Stereotype within a pair of guillemets. This is useful if values of more than one applied stereotype should be shown.

When displayed in compartments or in a comment symbol, at most one namestring-valuestring pair can appear on a single line. When displayed above or before a model element's name, the name-value pairs are separated by semicolons and all pairs for a given stereotype are enclosed in braces.

## Icon presentation

It is possible to attach Images to a Stereotype that can be used in lieu of, or in addition to, the normal notation of a model element to which the Stereotype is applied.

When a Stereotype has a value for icon, the referenced Image can be graphically attached to the model elements to which the Stereotype has been applied. Every model element that has a graphical presentation can have an attached icon. When model elements are graphically expressed as:

- Boxes (see Figure 12.23): the box may be replaced by the Image, and the name of the model element appears below the Image. This presentation option can be used only when a model element has one single Stereotype applied and when Properties of the model element (e.g., ownedAttributes, ownedOperations of a Class) are not presented. As another option, the Image may be presented in a reduced size, inside and to the top of the box representing the model element. When several Stereotypes are applied, several Images may be presented within the box.
- Lines: the Image may be placed close to the line.
- Textual notation: the Image may be presented to the left of the textual notation.

Several Images may be referenced by a Stereotype's icon Property. The interpretation of the different attached Images in that case is a semantic variation point. Some tools may use the different Images for different purposes: the icon replacing the box, for the reduced-size icon inside the box, for icons within tree browsers, etc. Alternatively, depending on the Image format, tools may choose to scale one single Image into different sizes for these different purposes.

Some model elements already use an icon for their default presentation. A typical example of this is the Actor model element, which uses the "stickman" icon. When a Stereotype with an icon is applied to such a model element, the Stereotype's icon replaces the default presentation icon within diagrams.

### 12.3.5 Examples

In Figure 12.17, a simple example of using an Extension is shown, where the stereotype Home extends the metaclass Interface.

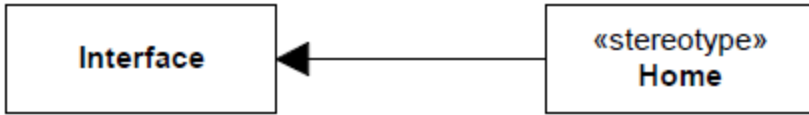


Figure 12.17 Example of Using an Extension

An instance of the stereotype Home can be added to and removed 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 Package.

In Figure 12.18, each instance of metaclass Component in a model to which the Profile has been applied must have applied an instance of the stereotype Bean, as the Extension has `isRequired = true`. (As the stereotype Bean is abstract, this means that each instance of metaclass Component must be stereotyped by an instance of one of its concrete subclasses.) The model is not well-formed unless such a Stereotype is applied. This provides a way to express Extensions that should always be present for all instances of the base metaclass depending on which Profiles are applied.

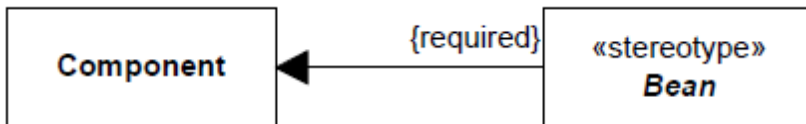


Figure 12.18 Example of a Required Extension

In Figure 12.19, a simple example of an EJB profile is shown.

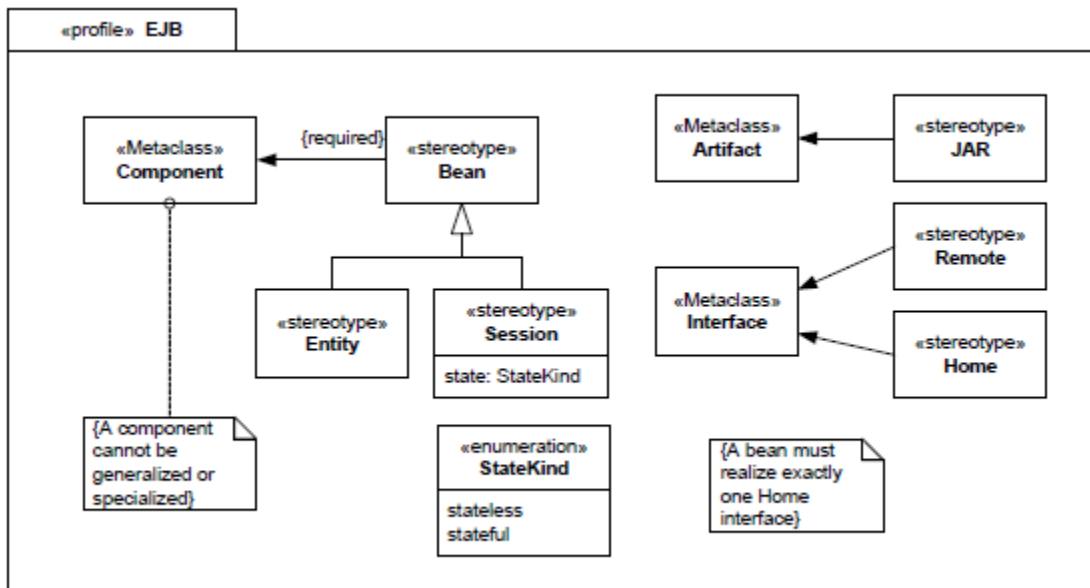
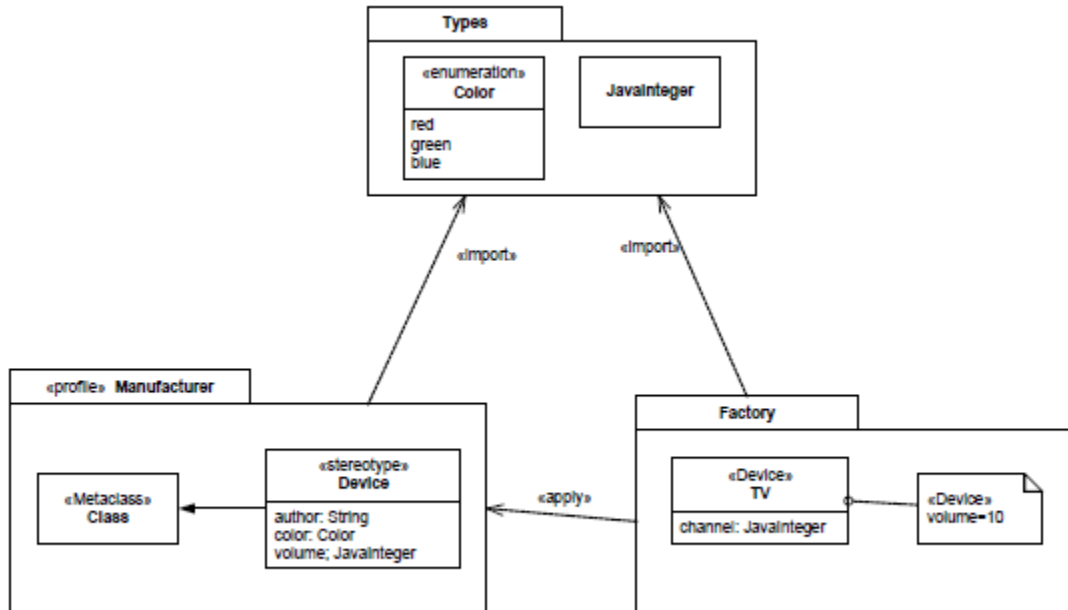


Figure 12.19 Defining a Simple EJB Profile

The Profile defines 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 is applied to a Package, and these Constraints need to be satisfied in order for the model to be well-formed.



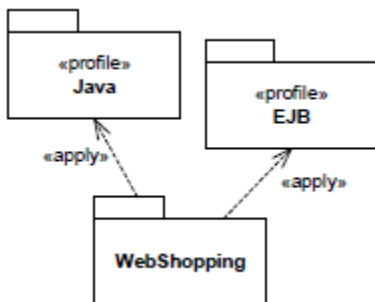
**Figure 12.20 Importing a Package from a Profile**

In Figure 12.20, the Package named **Types** is imported by the Profile named **Manufacturer**. The Enumeration named **Color** and the Class named **JavaInteger** are then used as the type of Properties of the Stereotype named **Device** as well as the standard PrimitiveType **String**.

If the Profile **Manufacturer** is later applied to a Package, then the types from **Types** are not available for use in the Package to which the Profile is applied unless package **Types** is explicitly imported. This means that the class **JavaInteger** can be used as the type of a Stereotype Property (e.g., in **Device**) but not as an ordinary Property (as part of the Class **TV**) unless Package **Factory** also imports Package **Types** (which it does).

**NOTE.** The value of the volume Property is displayed once the Stereotype **Device** has been applied to the Class **TV**.

Given the profiles **Java** and **EJB**, Figure 12.21 shows how these may be applied to the Package **WebShopping**.



**Figure 12.21 Profiles Applied to a Package**

In Figure 12.22, a simple stereotype **Clock** is defined to be applicable at will (dynamically) to instances of the metaclass **Class**.



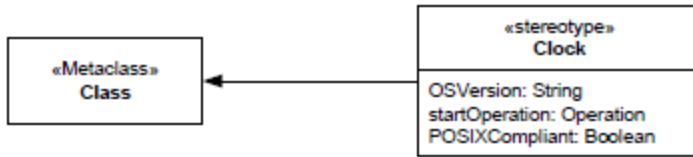


Figure 12.22 Defining a Stereotype

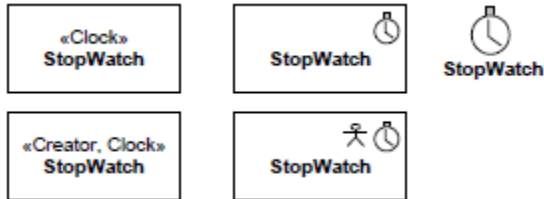


Figure 12.23 Presentation Options for an Extended Class

In Figure 12.24, an instance diagram of the example in Figure 12.22 is shown.

**NOTE.** The ExtensionEnd must be composite, and that the derived isRequired Property in this case is **false**.

Figure 12.24 shows the instances representing the definition of the Stereotype named Clock defined in Figure 12.22. In this definition, the extended metaclass (:Class; “name = Class”) is defined in the UML2 metamodel (reference metamodel). In a UML modeling tool this representation of the UML2 standard metamodel would typically be in a “read only” form, or presented as proxies to the metaclass being extended.

(It is therefore still at the same meta-level as UML, and does not show the instance model of a model extended by the stereotype. An example of this is provided in Figure 12.26 and Figure 12.27.) The Semantics sub clause of the Extension concept explains the MOF equivalent, and how constraints can be attached to stereotypes.

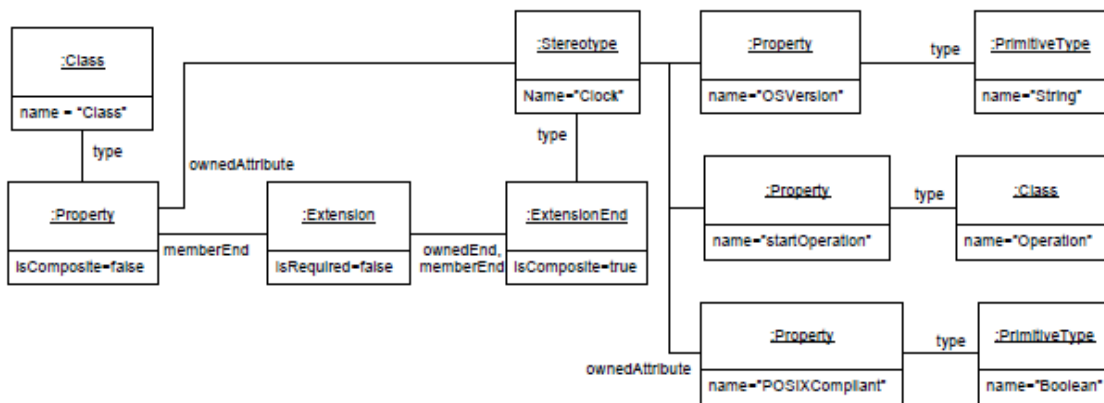


Figure 12.24 An Instance Diagram when Defining a Stereotype

Figure 12.25 shows how the same Stereotype named Clock extends both the metaclass Component and the metaclass Class (though each instance of the Stereotype can extend only one model element). It also shows how different Stereotypes can extend the same metaclass.

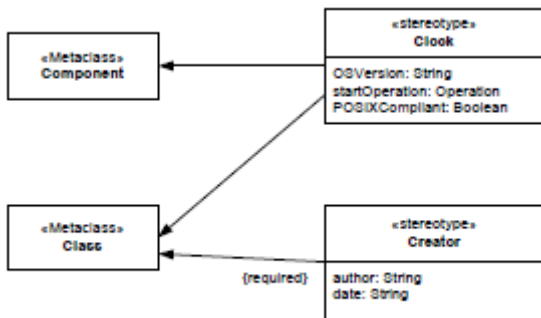


Figure 12.25 Defining Multiple Stereotypes on Multiple Stereotypes

Figure 12.26 shows how the Stereotype Clock, as defined in Figure 12.25, is applied to a Class named StopWatch.

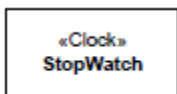


Figure 12.26 Using a Stereotype

Figure 12.27 shows the underlying semantics for when the Stereotype named Clock is applied to a class called StopWatch. The right-hand side uses instance diagram notation to show the MOF-equivalent instances that should be used to understand the behavior and XMI serialization of the UML diagram on the left. The Extension between the Stereotype and the metaclass Class results in a link between the instance of Stereotype Clock and the (user-defined) Class named StopWatch.

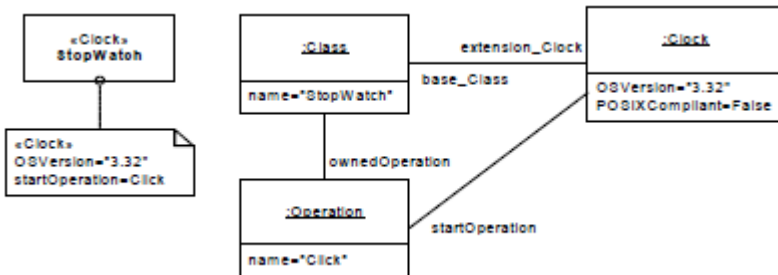


Figure 12.27 Showing Values of Stereotypes and a Simple Instance Specification

Next, two stereotypes, Clock and Creator, are applied to the same model element, as shown in Figure 12.28.

**NOTE.** The Property values of each of the applied Stereotypes are shown in a comment symbol attached to the model element.



Figure 12.28 Using Stereotypes and Showing Values

Finally, two more alternative notational forms are shown in Figure 12.29.

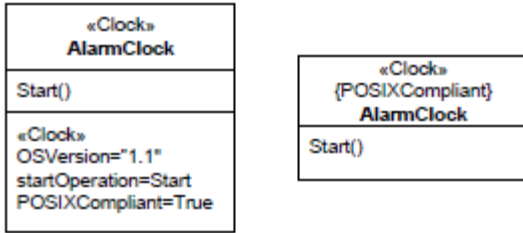


Figure 12.29 Other Notational Forms for Depicting Stereotype Values

## 12.4 Classifier Descriptions

### Extension [Class]

#### Description

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.

#### Diagrams

[Profiles](#), [Classes](#)

#### Generalizations

[Association](#)

#### Attributes

- /isRequired : [Boolean](#) [1..1] = false  
Indicates whether an instance of the extending stereotype must be created when an instance of the extended class is created. The attribute value is derived from the value of the lower property of the ExtensionEnd referenced by Extension::ownedEnd; a lower value of 1 means that isRequired is true, but otherwise it is false. Since the default value of ExtensionEnd::lower is 0, the default value of isRequired is false.

#### Association Ends

- /metaclass : [Class](#) [1..1]{ } (opposite [Class::extension](#))  
References the Class that is extended through an Extension. The property is derived from the type of the memberEnd that is not the ownedEnd.
- ♦ ownedEnd : [ExtensionEnd](#) [1..1]{ redefines [Association::ownedEnd](#) } (opposite [A ownedEnd extension::extension](#))  
References the end of the extension that is typed by a Stereotype.

#### Operations

- isRequired() : [Boolean](#)  
The query isRequired() is true if the owned end has a multiplicity with the lower bound of 1.

```
body: ownedEnd.lowerBound() = 1
```

- metaclass() : [Class](#)  
The query metaclass() returns the metaclass that is being extended (as opposed to the extending stereotype).

```
body: metaclassEnd().type.oclassType(Class)
```

- metaclassEnd() : [Property](#)  
The query metaclassEnd() returns the Property that is typed by a metaclass (as opposed to a stereotype).

```
body: memberEnd->reject(p | ownedEnd->includes(p.oclassType(ExtensionEnd)))->any(true)
```

## Constraints

- `non_owned_end`  
The non-owned end of an Extension is typed by a Class.

```
inv: metaclassEnd()->notEmpty() and metaclassEnd().type.oclIsKindOf(Class)
```

- `is_binary`  
An Extension is binary, i.e., it has only two memberEnds.

```
inv: memberEnd->size() = 2
```

## ExtensionEnd [Class]

### Description

An extension end is used to tie an extension to a stereotype when extending a metaclass. The default multiplicity of an extension end is 0..1.

### Diagrams

[Profiles](#)

### Generalizations

[Property](#)

### Attributes

- `/lower` : [Integer](#) [0..1] = 0  
This redefinition changes the default multiplicity of association ends, since model elements are usually extended by 0 or 1 instance of the extension stereotype.

### Association Ends

- `type` : [Stereotype](#) [1..1]{redefines [TypedElement::type](#)} (opposite [A\\_type\\_extensionEnd::extensionEnd](#))  
References the type of the ExtensionEnd. Note that this association restricts the possible types of an ExtensionEnd to only be Stereotypes.

### Operations

- `lowerBound()` : [Integer](#) [0..1]  
The query `lowerBound()` returns the lower bound of the multiplicity as an Integer. This is a redefinition of the default lower bound, which normally, for `MultiplicityElements`, evaluates to 1 if empty.

```
body: if lowerValue=null then 0 else lowerValue.integerValue() endif
```

## Constraints

- multiplicity  
The multiplicity of ExtensionEnd is 0..1 or 1.

```
inv: (lowerBound() = 0 or lowerBound() = 1) and upperBound() = 1
```

- aggregation  
The aggregation of an ExtensionEnd is composite.

```
inv: self.aggregation = AggregationKind::composite
```

## Image [Class]

### Description

Physical definition of a graphical image.

### Diagrams

[Profiles](#)

### Generalizations

[Element](#)

### Attributes

- content : [String](#) [0..1]  
This contains the serialization of the image according to the format. The value could represent a bitmap, image such as a GIF file, or drawing 'instructions' using a standard such as Scalable Vector Graphic (SVG) (which is XML based).
- format : [String](#) [0..1]  
This indicates the format of the content, which is how the string content should be interpreted. The following values are reserved: SVG, GIF, PNG, JPG, WMF, EMF, BMP. In addition the prefix 'MIME: ' is also reserved. This option can be used as an alternative to express the reserved values above, for example "SVG" could instead be expressed as "MIME: image/svg+xml".
- location : [String](#) [0..1]  
This contains a location that can be used by a tool to locate the image as an alternative to embedding it in the stereotype.

## Model [Class]

### Description

A model captures a view of a physical system. It is an abstraction of the physical system, with a certain purpose. This purpose determines what is to be included in the model and what is irrelevant. Thus the model completely describes those aspects of the physical system that are relevant to the purpose of the model, at the appropriate level of detail.

## Diagrams

[Packages](#)

## Generalizations

[Package](#)

## Attributes

- viewpoint : [String](#) [0..1]  
The name of the viewpoint that is expressed by a model (this name may refer to a profile definition).

## Package [Class]

### Description

A package can have one or more profile applications 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. Package specializes [TemplateableElement](#) and [PackageableElement](#) specializes [ParameterableElement](#) to specify that a package can be used as a template and a [PackageableElement](#) as a template parameter. A package is used to group elements, and provides a namespace for the grouped elements.

### Diagrams

[Packages](#), [Profiles](#), [Namespaces](#)

### Generalizations

[PackageableElement](#), [TemplateableElement](#), [Namespace](#)

### Specializations

[Model](#), [Profile](#)

### Attributes

- URI : [String](#) [0..1]  
Provides an identifier for the package that can be used for many purposes. A URI is the universally unique identification of the package following the IETF URI specification, RFC 2396 <http://www.ietf.org/rfc/rfc2396.txt> and it must comply with those syntax rules.

### Association Ends

- $\diamond$  /nestedPackage : [Package](#) [0..\*]{subsets [Package::packagedElement](#)} (opposite [Package::nestingPackage](#))  
References the packaged elements that are Packages.
- nestingPackage : [Package](#) [0..1]{subsets [A packagedElement owningPackage::owningPackage](#)} (opposite [Package::nestedPackage](#))  
References the Package that owns this Package.

- ♦ /ownedStereotype : [Stereotype](#) [0..\*]{subsets [Package::packagedElement](#)} (opposite [A\\_ownedStereotype\\_owningPackage::owningPackage](#))  
References the Stereotypes that are owned by the Package.
- ♦ /ownedType : [Type](#) [0..\*]{subsets [Package::packagedElement](#)} (opposite [Type::package](#))  
References the packaged elements that are Types.
- ♦ packageMerge : [PackageMerge](#) [0..\*]{subsets [Element::ownedElement](#), subsets [A\\_source\\_directedRelationship::directedRelationship](#)} (opposite [PackageMerge::receivingPackage](#))  
References the PackageMerges that are owned by this Package.
- ♦ packagedElement : [PackageableElement](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [A\\_packagedElement\\_owningPackage::owningPackage](#))  
Specifies the packageable elements that are owned by this Package.
- ♦ profileApplication : [ProfileApplication](#) [0..\*]{subsets [Element::ownedElement](#), subsets [A\\_source\\_directedRelationship::directedRelationship](#)} (opposite [ProfileApplication::applyingPackage](#))  
References the ProfileApplications that indicate which profiles have been applied to the Package.

## Operations

- allApplicableStereotypes() : [Stereotype](#) [0..\*]  
The query allApplicableStereotypes() returns all the directly or indirectly owned stereotypes, including stereotypes contained in sub-profiles.  
  

```
body: let ownedPackages : Bag (Package) = ownedMember->select (oclIsKindOf (Package)) -
>collect (oclAsType (Package)) in
  ownedStereotype->union (ownedPackages.allApplicableStereotypes ()) ->flatten () ->asSet ()
```
- containingProfile() : [Profile](#) [0..1]  
The query containingProfile() returns the closest profile directly or indirectly containing this package (or this package itself, if it is a profile).  
  

```
body: if self.oclIsKindOf (Profile) then
  self.oclAsType (Profile)
else
  self.namespace.oclAsType (Package).containingProfile ()
endif
```
- makesVisible(e1 : [NamedElement](#)) : [Boolean](#)  
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.  
  

```
pre: member->includes (e1)
body: ownedMember->includes (e1) or
(elementImport->select (ei|ei.importedElement = VisibilityKind::public)-
>collect (importedElement.oclAsType (NamedElement)) ->includes (e1)) or
(packageImport->select (visibility = VisibilityKind::public)->collect (importedPackage.member-
>includes (e1)) ->notEmpty ())
```
- mustBeOwned() : [Boolean](#)  
The query mustBeOwned() indicates whether elements of this type must have an owner.



body: false

- nestedPackage() : [Package](#) [0..\*]  
Derivation for Package::/nestedPackage

body: packagedElement->select(oclIsKindOf(Package))->collect(oclAsType(Package))->asSet()

- ownedStereotype() : [Stereotype](#) [0..\*]  
Derivation for Package::/ownedStereotype

body: packagedElement->select(oclIsKindOf(Stereotype))->collect(oclAsType(Stereotype))->asSet()

- ownedType() : [Type](#) [0..\*]  
Derivation for Package::/ownedType

body: packagedElement->select(oclIsKindOf(Type))->collect(oclAsType(Type))->asSet()

- visibleMembers() : [PackageableElement](#) [0..\*]  
The query visibleMembers() defines which members of a Package can be accessed outside it.

body: member->select(m | m.oclIsKindOf(PackageableElement) and self.makesVisible(m))->collect(oclAsType(PackageableElement))->asSet()

## Constraints

- elements\_public\_or\_private  
If an element that is owned by a package has visibility, it is public or private.

inv: packagedElement->forAll(e | e.visibility<> null implies e.visibility = VisibilityKind::public or e.visibility = VisibilityKind::private)

## PackageMerge [Class]

### Description

A package merge defines how the contents of one package are extended by the contents of another package.

### Diagrams

[Packages](#)

### Generalizations

[DirectedRelationship](#)

### Association Ends

- mergedPackage : [Package](#) [1..1]{subsets [DirectedRelationship::target](#)} (opposite [A mergedPackage packageMerge::packageMerge](#))  
References the Package that is to be merged with the receiving package of the PackageMerge.

- receivingPackage : [Package](#) [1..1]{subsets [DirectedRelationship::source](#), subsets [Element::owner](#)} (opposite [Package::packageMerge](#))  
References the Package that is being extended with the contents of the merged package of the PackageMerge.

## Profile [Class]

### Description

A profile defines limited extensions to a reference metamodel with the purpose of adapting the metamodel to a specific platform or domain.

### Diagrams

[Profiles](#)

### Generalizations

[Package](#)

### Association Ends

- ♦ metaclassReference : [ElementImport](#) [0..\*]{subsets [Namespace::elementImport](#)} (opposite [A\\_metaclassReference\\_profile::profile](#))  
References a metaclass that may be extended.
- ♦ metamodelReference : [PackageImport](#) [0..\*]{subsets [Namespace::packageImport](#)} (opposite [A\\_metamodelReference\\_profile::profile](#))  
References a package containing (directly or indirectly) metaclasses that may be extended.

### Constraints

- metaclass\_reference\_not\_specialized  
An element imported as a metaclassReference is not specialized or generalized in a Profile.

```
inv: metaclassReference.importedElement->
  select(c | c.ocIsKindOf(Classifier) and
         (c.ocAsType(Classifier).allParents()->collect(namespace)->includes(self)))->isEmpty()
and
packagedElement->
  select(ocIsKindOf(Classifier))->collect(ocAsType(Classifier).allParents())->
  intersection(metaclassReference.importedElement->select(ocIsKindOf(Classifier))->
  >collect(ocAsType(Classifier)))->isEmpty()
```

- references\_same\_metamodel  
All elements imported either as metaclassReferences or through metamodelReferences are members of the same base reference metamodel.

```
inv: metamodelReference.importedPackage.elementImport.importedElement.allOwningPackages()->
  union(metaclassReference.importedElement.allOwningPackages() )->notEmpty()
```

## ProfileApplication [Class]

### Description

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

### Diagrams

[Profiles](#)

### Generalizations

[DirectedRelationship](#)

### Attributes

- isStrict : [Boolean](#) [1..1] = false  
Specifies that the Profile filtering rules for the metaclasses of the referenced metamodel shall be strictly applied.

### Association Ends

- appliedProfile : [Profile](#) [1..1]{subsets [DirectedRelationship::target](#)} (opposite [A\\_appliedProfile\\_profileApplication::profileApplication](#))  
References the Profiles that are applied to a Package through this ProfileApplication.
- applyingPackage : [Package](#) [1..1]{subsets [DirectedRelationship::source](#), subsets [Element::owner](#)} (opposite [Package::profileApplication](#))  
The package that owns the profile application.

## Stereotype [Class]

### Description

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

### Diagrams

[Profiles](#)

### Generalizations

[Class](#)

### Association Ends

- ♦ icon : [Image](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [A\\_icon\\_stereotype::stereotype](#))  
Stereotype can change the graphical appearance of the extended model element by using attached icons. When this association is not null, it references the location of the icon content to be displayed within diagrams presenting the

extended model elements.

- /profile : [Profile](#) [1..1]{ } (opposite [A\\_profile\\_stereotype::stereotype](#))  
The profile that directly or indirectly contains this stereotype.

## Operations

- containingProfile() : [Profile](#)  
The query containingProfile returns the closest profile directly or indirectly containing this stereotype.

```
body: self.namespace.oclAsType(Package).containingProfile()
```

- profile() : [Profile](#)  
A stereotype must be contained, directly or indirectly, in a profile.

```
body: self.containingProfile()
```

## Constraints

- binaryAssociationsOnly  
Stereotypes may only participate in binary associations.

```
inv: ownedAttribute.association->forall(memberEnd->size()=2)
```

- generalize  
A Stereotype may only generalize or specialize another Stereotype.

```
inv: allParents()->forall(oclIsKindOf(Stereotype))  
and Classifier.allInstances()->forall(c | c.allParents()->exists(oclIsKindOf(Stereotype)) implies  
c.oclIsKindOf(Stereotype))
```

- name\_not\_clash  
Stereotype names should not clash with keyword names for the extended model element.

Cannot be expressed in OCL

- associationEndOwnership  
Where a stereotype's property is an association end for an association other than a kind of extension, and the other end is not a stereotype, the other end must be owned by the association itself.

```
inv: ownedAttribute  
->select(association->notEmpty() and not association.oclIsKindOf(Extension) and not  
type.oclIsKindOf(Stereotype))  
->forall(opposite.owner = association)
```

## 12.5 Association Descriptions

### A\_appliedProfile\_profileApplication [Association]

#### Diagrams

[Profiles](#)

#### Owned Ends

- profileApplication : [ProfileApplication](#) [0..\*]{subsets [A\\_target\\_directedRelationship::directedRelationship](#)} (opposite [ProfileApplication::appliedProfile](#))

### A\_icon\_stereotype [Association]

#### Diagrams

[Profiles](#)

#### Owned Ends

- stereotype : [Stereotype](#) [0..1]{subsets [Element::owner](#)} (opposite [Stereotype::icon](#))

### A\_mergedPackage\_packageMerge [Association]

#### Diagrams

[Packages](#)

#### Owned Ends

- packageMerge : [PackageMerge](#) [0..\*]{subsets [A\\_target\\_directedRelationship::directedRelationship](#)} (opposite [PackageMerge::mergedPackage](#))

### A\_metaclassReference\_profile [Association]

#### Diagrams

[Profiles](#)

## Owned Ends

- profile : [Profile](#) [0..1]{subsets [ElementImport::importingNamespace](#)} (opposite [Profile::metaclassReference](#))

## A\_metamodelReference\_profile [Association]

### Diagrams

[Profiles](#)

## Owned Ends

- profile : [Profile](#) [0..1]{subsets [PackageImport::importingNamespace](#)} (opposite [Profile::metamodelReference](#))

## A\_nestedPackage\_nestingPackage [Association]

### Diagrams

[Packages](#)

## Member Ends

- [Package::nestedPackage](#)
- [Package::nestingPackage](#)

## A\_ownedEnd\_extension [Association]

### Diagrams

[Profiles](#)

## Owned Ends

- extension : [Extension](#) [1..1]{subsets [Property::owningAssociation](#)} (opposite [Extension::ownedEnd](#))

## A\_ownedStereotype\_owningPackage [Association]

### Diagrams

[Profiles](#)

## Generalizations

[A\\_packagedElement\\_owningPackage](#)

## Owned Ends

- owningPackage : [Package](#) [1..1]{redefines [A\\_packageElement\\_owningPackage::owningPackage](#)} (opposite [Package::ownedStereotype](#))

## A\_ownedType\_package [Association]

### Diagrams

[Packages](#)

### Member Ends

- [Package::ownedType](#)
- [Type::package](#)

## A\_packageMerge\_receivingPackage [Association]

### Diagrams

[Packages](#)

### Member Ends

- [Package::packageMerge](#)
- [PackageMerge::receivingPackage](#)

## A\_packagedElement\_owningPackage [Association]

### Diagrams

[Packages](#)

### Specializations

[A\\_ownedStereotype\\_owningPackage](#)

### Owned Ends

- owningPackage : [Package](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [Package::packagedElement](#))

## A\_profileApplication\_applyingPackage [Association]

### Diagrams

[Profiles](#)

## Member Ends

- [Package::profileApplication](#)
- [ProfileApplication::applyingPackage](#)

## A\_profile\_stereotype [Association]

### Diagrams

[Profiles](#)

### Owned Ends

- stereotype : [Stereotype](#) [0..\*] (opposite [Stereotype::profile](#))

## A\_type\_extensionEnd [Association]

### Diagrams

[Profiles](#)

### Owned Ends

- extensionEnd : [ExtensionEnd](#) [0..\*]{subsets [A\\_type\\_typedElement::typedElement](#)} (opposite [ExtensionEnd::type](#))



## 13 Common Behavior

### 13.1 Summary

This clause specifies the core concepts underlying all behavioral modeling in UML. Structural models of Classifiers in UML define the allowable instances that may exist at any point in time, what values their StructuralFeatures may have and how those instances may be related to each other. Behavioral modeling, on the other hand, models how these instances may change over time.

UML provides Behavior, Event, and Trigger constructs to model the corresponding fundamental concepts of behavioral modeling.

*Behavior* is the basic concept for modeling dynamic change. Behavior may be *executed*, either by direct invocation or through the creation of an *active object* that hosts the behavior. Behavior may also be *emergent*, resulting from the interaction of one or more participant objects that are themselves carrying out their own individual behaviors.

Dynamic behavior results in *events* of interest that occur at specific points in time. Such events may be implicit, occurring on the change of some value or the passage of some interval of time. They may also be explicit, occurring when an operation is called or an asynchronous *signal* is received.

The occurrence of an event may then *trigger* new behavior, or change the course of already executing behavior. Explicit events thus provide the basic mechanism for communication between behaviors, in which an action carried out in one behavior, such as calling an operation or sending a signal, can trigger a response in another behavior.

The remainder of this clause further details the fundamental UML modeling mechanisms of Behaviors, Events and Triggers. These mechanisms then provide the framework for the specification in the following clauses of various complete UML behavioral modeling constructs.

### 13.2 Behaviors

#### 13.2.1 Summary

This sub clause introduces the framework for modeling behavior in UML. The concrete subtypes of Behavior, described in subsequent clauses, then provide different mechanisms to specify behaviors.

A variety of behavioral specification mechanisms are supported by UML, including:

- StateMachines that model finite automata (see Clause [14](#))
- Activities defined using Petri-net-like graphs (see Clause [15](#))
- Interactions that model partially-ordered sequences of event occurrences (see Clause [17](#)).

These behavioral specification mechanisms differ in their expressive power and domain of applicability. This means that not all behaviors can be described by each of the mechanisms. Nevertheless, many behaviors can be described by one or more of the mechanisms, in which case the choice of mechanism is one of convenience, or, alternatively, multiple mechanisms can be used to provide different models of the same behavior.

## 13.2.2 Abstract Syntax

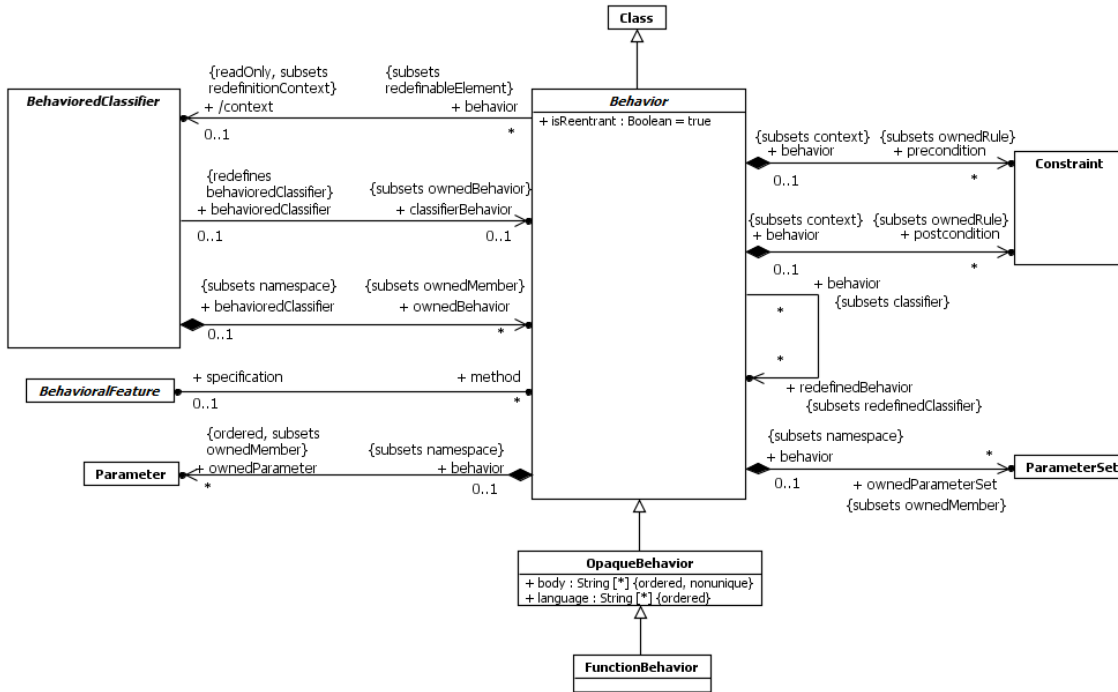


Figure 13.1 Behaviors

## 13.2.3 Semantics

### Behaviors

A Behavior is a specification of events that may occur dynamically over time (see also sub clause 13.3 on the explicit modeling of Events in UML). This specification may be prescriptive of specifically what events may occur in what situations, descriptive of emergent behavior or illustrative of possible sequences of event occurrences. Every Behavior defines at least one event, the event of its invocation. A Behavior may be invoked directly, via a BehavioralFeature that it implements as a method or as the classifierBehavior of a BehavedClassifier.

On each invocation, the subsequent actual sequence of event occurrences due to the invocation, consistent with the specification of the Behavior, is called an *execution trace* for the Behavior. An execution trace always begins with the invocation of the Behavior and may continue indefinitely (if the Behavior does not terminate), or it may end in the occurrence of a *completion* event for the Behavior. Event occurrences during an execution trace include both occurrences caused by the Behavior, such as attribute value changes, creation and destruction of objects and invocation of other Behaviors, and occurrences that trigger responses within the Behavior, such as the changing of a monitored value or the receipt of a Signal instance.

Behaviors in UML are kinds of Classes, which means that they may be instantiated as objects. An object that is an instance of a Behavior is known as a behavior *execution*. Invoking the Behavior corresponds to instantiating the Behavior, and there is a specific execution trace corresponding to each Behavior execution.

Since a Behavior is a Class, it may be specialized and may also itself own StructuralFeatures and BehavioralFeatures. These features may be referenced in the specification of the Behavior. An execution of the Behavior may then access these features, such as reading and modifying attributes of the Behavior. Public features of a Behavior may also be referenced from outside of the Behavior, as usual for the features of any Class.

A Behavior may be invoked many times. A *reentrant* Behavior (i.e., one with its `isReentrant` property equal to `true`) may be invoked again before a previous invocation has completed (this is the default). On the other hand, a *non-reentrant* Behavior (i.e., one with its `isReentrant` property equal to `false`) shall not be invoked again if a previous invocation has not completed. A reentrant Behavior may have many ongoing executions at any one time, but a non-reentrant Behavior shall have at most one uncompleted execution at any time. If an invoking Behavior attempts to invoke a non-reentrant Behavior that already has an uncompleted execution, then the invoker shall block until the existing execution completes (or indefinitely, if the execution never completes).

A Behavior may be invoked *synchronously* or *asynchronously*. Synchronous invocation means that an invoking Behavior retains a reference to the invoked Behavior execution and waits for the execution to complete. Asynchronous invocation, on the other hand, means that the invoked Behavior execution proceeds concurrently with the invoking Behavior.

The preconditions for a Behavior define conditions that shall be true when the Behavior is invoked. These preconditions may be assumed in the detailed specification of the Behavior. The semantics of an invocation of a Behavior when a precondition is not satisfied are intentionally undefined.

The postconditions for a Behavior define conditions that will be true when the invocation of the Behavior completes successfully, assuming the preconditions were satisfied. These postconditions shall be satisfied in the detailed specification of the Behavior.

## Behavior Parameters

A Behavior may have Parameters (see sub clause [9.4](#)) that provide the ability to pass values into and out of Behavior executions.

When a Behavior is invoked, *argument* values may be provided corresponding to Parameters with direction “in” or “inout”, as constrained by the multiplicity of those Parameters. If such an input Parameter has a `defaultValue`, and no explicit argument value is given for it, then the `defaultValue` is evaluated to provide argument values for the Parameter. Argument values are available to affect the course of the invoked Behavior execution.

When a Behavior execution completes, it may produce *result* values corresponding to Parameters with direction “inout,” “out,” and “return,” as constrained by the multiplicity of those Parameters. If such an output Parameter has a `defaultValue`, and no explicit result value is given for it, then the `defaultValue` is evaluated to provide result values for the Parameter. If the Behavior was invoked synchronously, then result values are returned to the invoker. However, if the Behavior was invoked asynchronously, then any result values are lost when the Behavior execution completes.

Parameters may also be marked as *streaming* (i.e., have the `isStreaming` property be `true`). Such Parameters allow values to be passed into and out of a Behavior execution any time during its course, rather than just on invocation and completion.

If an input Parameter is streaming, then argument values may be provided for the Parameter during the course of a Behavior execution rather than just at invocation. One or more argument values may be *posted* to a streaming input Parameter at or any time after the invocation of a Behavior and before its completion. These argument values are then available to affect the further course of the Behavior execution from that time forward.

If an output Parameter is streaming, then a Behavior execution may provide result values for the Parameter during its course rather than just at completion. One or more result values may be *posted* to a streaming output Parameter any time after the invocation of a Behavior up to or at its completion. These result values are then available to affect the further course of the execution of the *invoking* Behavior from that time forward.

**NOTE.** In order for an invoker to be able to obtain posted values from streaming output Parameters, the invoked Behavior has to be invoked synchronously, even though streamed outputs could potentially trigger asynchronous responses in the invoker (see, for example, the semantics of `CallBehaviorActions` for Behaviors with streaming Parameters in sub clause [16.3](#)).

A reentrant Behavior shall not have streaming Parameters, because there are potentially multiple executions of the Behavior going at the same time, and it would be ambiguous which execution would be receiving or producing streamed values.

A Behavior may have one or more output Parameters marked as `isException=true`. In this case, when an execution of the Behavior completes, either none of these Parameters shall have values or exactly one shall have a value and no other parameters (exception or otherwise) shall have any values.

**NOTE.** Returning a value in an exception Parameter is *not* considered to be “raising an exception” in the sense described in sub clause [15.5.3](#).

A Behavior with input ParameterSets can only accept inputs from Parameters in one of the sets per execution. A Behavior with output ParameterSets can only post outputs to the Parameters in one of the sets per execution. The semantics of conditions of input and output ParameterSets are the same as Behavior preconditions and postconditions, respectively, but apply only to the set of Parameters specified.

## Opaque and Function Behaviors

An OpaqueBehavior is a Behavior whose specification is given in a textual language other than UML.

An OpaqueBehavior has a body that consists of a sequence of text Strings representing alternative means of specifying the required behavior. A corresponding sequence of language Strings may be used to specify the languages in which each of the body Strings is to be interpreted. Languages are matched to body Strings by order. The UML specification does not define how body Strings are interpreted relative to any language, though other specifications may define specific language Strings to be used to indicate interpretation with respect to those specifications (e.g., “OCL” for expressions to be interpreted according to the OCL specification).

**NOTE.** It is not required to specify the languages. If they are unspecified, then the interpretation of any body Strings shall be determined implicitly from the form of the bodies or the context of use of the OpaqueBehavior.

If an OpaqueBehavior has more than one body String, then any one of the bodies can be used to determine the behavior of the OpaqueBehavior. The UML specification does not determine how this choice is made.

A FunctionBehavior is an OpaqueBehavior that does not access or modify any objects or other external data. During the execution of a FunctionBehavior, no communication or interaction with anything external to the FunctionBehavior is allowed. The amount of time to compute its results is undefined. A FunctionBehavior may raise exceptions for certain input values, in which case the computation is abandoned.

FunctionBehaviors thus represent functions that transform a set of input argument values (given by the input Parameters of the FunctionBehavior) to a set of output result values (given by the output Parameters of the FunctionBehavior). The execution of a FunctionBehavior depends only on the argument values and has no other effect than to compute result values. Examples of functions that might be modeled as FunctionBehaviors include primitive arithmetic, Boolean, and String functions.

## Behavored Classifiers

A BehavoredClassifier is a Classifier that may have ownedBehaviors, at most one of which may be considered to specify the behavior of the BehavoredClassifier itself. Conversely, a Behavior that is the ownedBehavior of a BehavoredClassifier has that BehavoredClassifier as its context. The specification of such a Behavior may reference features of the context BehavoredClassifier as well as any other elements visible to the context BehavoredClassifier.

A Behavior that is not directly an ownedBehavior of a BehavoredClassifier may nevertheless still have a context. To determine the context of a Behavior that is not directly an ownedBehavior, find the first BehavoredClassifier reached by following the chain of ownership relationships from the Behavior, if any. If there is such a BehavoredClassifier, then it is the context, unless it is itself a Behavior with a non-empty context, in which case this is also the context for the original Behavior. For example, the context of an entry Behavior (see sub clause [14.2](#)) in a StateMachine owned by a BehavoredClassifier is the classifier that owns the StateMachine, not the StateMachine.

A Behavior that is owned directly by a Class as a nestedClassifier (see sub clause [11.4](#)), rather than as an ownedBehavior, does *not* have the Class as its context. The nestedClassifiers of a Class are simply nested in the Class considered as a Namespace. As a nestedClassifier, a Behavior has visibility of elements defined within the owning Class and other elements visible to that Class, and it may itself be visible outside the Class, depending on its declared visibility. But its semantics as a “stand-alone” Behavior are not otherwise affected by being nested in the Class.

If a Behavior has a context, then an execution of the Behavior always has an associated *context object* that is an instance of the context BehavedClassifier (as long as that BehavedClassifier is instantiable). A Behavior without a context BehavedClassifier may still be invoked as a “stand-alone” Behavior. In this case, the Behavior execution serves as its own context object. The Behavior execution also serves as its own context object in the case that the context BehavedClassifier is not instantiable, that is, if it is a Component with `isIndirectlyInstantiated=true` (see sub clause [11.6](#)) or a Collaboration (see sub clause [11.7](#)). Thus, a Behavior execution always has a context object, whether or not the Behavior has an explicit, instantiable context BehavedClassifier.

A BehavedClassifier may have a distinguished ownedBehavior called its classifierBehavior. A classifierBehavior describes the behavior an instance of the owning Classifier may undergo in the course of its lifetime. The classifierBehavior of a BehavedClassifier is considered to be invoked when an instance of the owning BehavedClassifier is created and the resulting execution has the new instance as its context object. The execution is terminated if the instance is destroyed.

A classifierBehavior is always a definition of behavior, not an illustration. Its precise semantics depends on the kind of BehavedClassifier that owns it. For example, the classifierBehavior of a Collaboration (see sub clause [11.7](#)) represents emergent behavior of all the parts, whereas the classifierBehavior of a Class (see sub clause [11.4](#)) is just the behavior of instances of the Class separated from the behaviors of any of its parts. However, a passive Class (with `isActive=false`) shall not have a classifierBehavior.

## Behavioral Features and Methods

There are two kinds of BehavioralFeatures: Operations (see sub clause [9.6](#)) and Receptions (see sub clause [10.3](#)). Of the different kinds of BehavedClassifiers in UML, only Classes may have BehavioralFeatures and only active Classes may have Receptions (see sub clause [11.4](#)). Calling an Operation on or sending a Signal instance to an object of a Class is a *request* for the object to carry out an identified BehavioralFeature. An Operation call identifies a specific operation to be invoked. The receipt of an instance of a Signal, on the other hand, is considered to be a request for any Reception of the receiving object that references that Signal or any direct or indirect generalization of it.

A BehavioralFeature of a Class may be implemented by one or more method Behaviors. Such a BehavioralFeature specifies that instances of the owning Class will respond to a request for the BehavioralFeature by invoking one of the feature’s implementing methods. A Behavior shall be the method for no more than one BehavioralFeature, called its specification. The specification of a Behavior shall be an owned or inherited member of the Class of which the Behavior is an ownedBehavior. It is possible to have more than one method associated with a single BehavioralFeature, but there shall be at most one Behavior for a particular pairing of a Class (as owner of the Behavior) and a BehavioralFeature (as the specification for the Behavior). This means that a single BehavioralFeature may have methods both in its owning Class and any direct or indirect subclass of that Class, but with no more than one method per Class.

The receiving object becomes the context object for the execution of any invoked methods.

**NOTE.** Methods of a Reception are always invoked asynchronously, while the methods of an Operation may be invoked either synchronously or asynchronously, depending on how the Operation is called.

The method resolution process shall be based on the BehavioralFeature being requested, the object receiving the request and any data values associated with the request (i.e., Operation input parameter values or Signal attribute values). However, the UML specification does not mandate that a conforming UML tool support any particular resolution process. In general, the resolution process may be complicated, to include such mechanisms as before-after methods, delegation, etc. In some of these variations, multiple Behaviors may be executed as a result of a single call. If no methods are identified by the resolution process, then it is undefined what happens.

The following is a simple object-oriented resolution process for a CallEvent that always results in at most one method being identified:

If the Class of the receiving object owns a method for the Operation identified in the CallEvent, then that method is the result of the resolution. Otherwise, the superclass of the Class of the receiving object is examined for a method for the Operation, and so on up the generalization hierarchy until a method is found or the root of the hierarchy is reached. If a

Class has multiple superclasses, then all of them are examined for a method. If no method is found, or a method is found in more than one ancestor Class along different paths, then the model is ill-formed for this resolution process and it results in no method.

A method of an Operation shall have Parameters corresponding to the Parameters of the Operation. Similarly, a method of a Reception shall have Parameters corresponding to the attributes of the Signal referenced by the Reception, which are considered as effective “in” Parameters of the Reception. The data values associated with a request – input Operation parameter values or Signal attribute values – are then passed to a method invoked due to the request via the method parameters. For a synchronous Operation call, output Parameter values from the method execution are also passed back to the Operation caller via the corresponding Operation output Parameters.

However, no one approach is defined for matching the Parameters of the method to the Parameters of the BehavioralFeature. Possible approaches include exact match (i.e., the type of the corresponding Parameters, order, must be the same), co-variant match (the type of a Parameter of the method may be a subtype of the type of the Parameter of the BehavioralFeature), contra-variant match (the type of a Parameter of the method may be a supertype of the type of the Parameter of the BehavioralFeature), or a combination thereof.

### 13.2.4 Notation

The notation for various subclasses of Behavior are defined in subsequent clauses.

The notation for Signals and Receptions is covered under Simple Classifiers in sub clause [10.3.4](#).

The notation for active Classes is covered under Structured Classifiers in sub clause [11.4.4](#).

### 13.2.5 Examples

None.

## 13.3 Events

### 13.3.1 Summary

An Event is a something that may occur at a specific instant in time. One Event may have many occurrences, which may happen at different times. In this sense, an Event can be considered a classification of its occurrences, though Events are not actually Classifiers in UML.

Of particular importance are Events that trigger a response within a Behavior. Such Events that may be explicitly modeled within UML include TimeEvents that occur at a specified time or after a duration, ChangeEvents that occur when a specified Boolean value becomes true and MessageEvents that occur on the receipt of a *message*, which is a communication from one Behavior to another requesting an Operation call or Signal reception.

### 13.3.2 Abstract Syntax

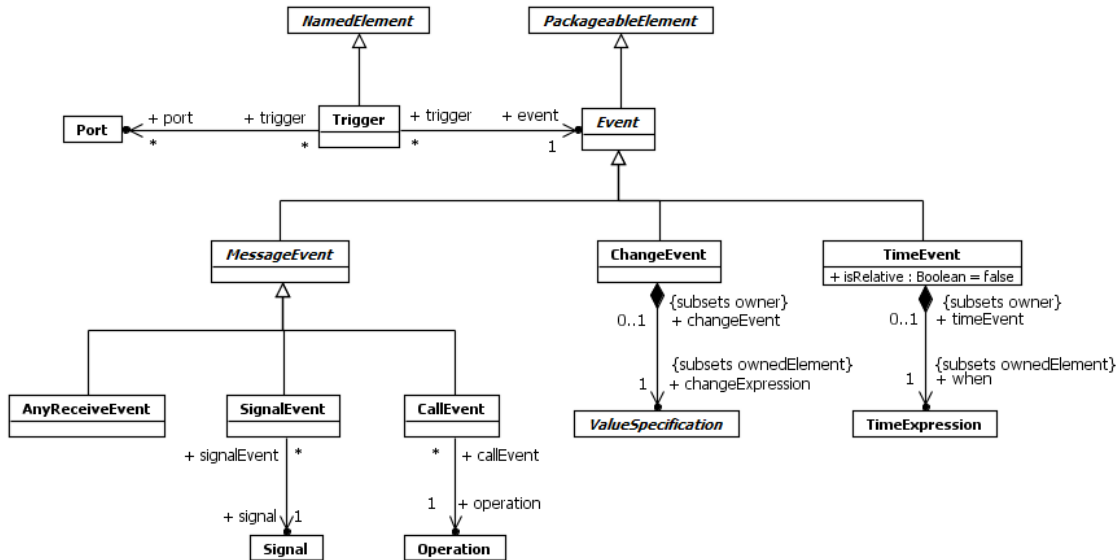


Figure 13.2 Events

### 13.3.3 Semantics

#### Event Dispatching

An Event is the specification of some occurrence that may potentially trigger behavioral effects. A Trigger specifies a specific point in the definition of a Behavior at which an Event occurrence may have such an effect. Event is a PackageableElement, allowing Events to be modeled independently of their use. A Trigger, however, always appears as a part of some larger behavioral specification (e.g., on a StateMachine Transition or in an AcceptEventAction). A single Event may be used in several different Triggers.

As discussed in sub clause 13.2.3, a Behavior execution always has an associated context object (which may be the execution itself). A context object mediates the handling of Event occurrences for all of its associated Behavior executions. When an Event occurrence is recognized by a context object, it may have an immediate effect or it may be saved for later triggered effect. An immediate effect is manifested by direct invocation of a Behavior as determined by the Event, such as the invocation of the method of a BehavioralFeature (see sub clause 13.2.3). A triggered effect is manifested by the storage of the occurrence in the *event pool* of the object and the later consumption of the occurrence by an ongoing Behavior execution that reaches a Trigger that matches the Event corresponding to the occurrence in the pool.

In general, when a Behavior execution comes to a *wait point* where it needs a Trigger to continue, the event pool of its context object is examined for an event that satisfies the outstanding Trigger (or Triggers). If the pool contains an event occurrence that satisfies one of the Triggers, the occurrence is removed from the pool and *dispatched* to the Behavior, which continues its execution as specified. Any data associated with the Event occurrence are made available to the triggered Behavior during its further execution.

**NOTE.** All Behaviors with the same context object share the event pool of that object, but any Event occurrence in the pool can be consumed by only one Behavior.

There is no requirement for a specific order in which Event occurrences in an event pool are examined or dispatched. If an event pool contains an occurrence that satisfies no Triggers at a wait point, then the general semantics of BehavioredClassifiers do not

specify what happens to it. (However, see the specific semantics for the dispatching and deferring of event occurrences for StateMachines in sub clause [14.2](#).)

## Message Events

A message is a communication in which a sender makes a request for either an Operation call or Signal reception by a receiver. This communication involves two events: the event of sending the message and the event of receiving the message. Sending events, however, are not modeled as explicit model elements in UML, though they are implicit in the execution of InvocationActions (see sub clause [16.3](#)) and occurrences of such events can be modeled in Interactions (see sub clause [17.5](#)). A MessageEvent, on the other hand, is an explicit model of the receipt of a message, in order to be able to specify a Trigger that responds to occurrences of that event.

A message contains:

- Data associated with the request being made (arguments for Operation parameters or values for Signal attributes).
- Information about the nature of the request (i.e., the BehavioralFeature invoked).
- For a synchronous invocation, sufficient information to enable the return of a reply from the invoked Behavior.

While each message is targeted at exactly one receiver object and caused by exactly one sending object, an occurrence of a sending event may result in a number of messages being generated (as in SignalBroadcastAction, see sub clause [16.3](#)). The receiver of a message may be the same as the sender, it may be local (i.e., an object held in a slot of the currently executing Behavior or its context object) or it may be remote. The manner of transmitting the message, the amount of time required to transmit it, the order in which the transmissions reach their receiver object and the path for reaching the receiver object are undefined.

The receipt of a message is manifested as a MessageEvent occurrence. A CallEvent is a MessageEvent for messages requesting that a specific Operation be called. A SignalEvent is a MessageEvent for messages requesting the reception of an instance of a specific Signal. An AnyReceiveEvent is a MessageEvent for any message that is not explicitly handled by any other related Trigger.

In the case of a CallEvent for an Operation or a SignalEvent for a Signal that matches a Reception on the receiver, if the Operation or Reception has one or more methods, then the method resolution process described for Behavioral Features and Methods in sub clause 13.2.3 shall be carried out to determine a method to be used to handle a MessageEvent occurrence. If a method is so identified, it is invoked to respond to the message request. Otherwise, the MessageEvent occurrence is saved in the event pool of the receiving object. When a MessageEvent occurrence is dispatched from the event pool and matches a Trigger defined in the Behavior specification for the receiver, it causes the execution of a response within the Behavior.

A Trigger for an AnyReceiveEvent may be triggered by the receipt of any message (Signal send or Operation call). However, if there is a relevant SignalEvent or CallEvent Trigger that specifically matches the message, then the AnyReceiveEvent Trigger is *not* triggered by the message. Which other Triggers are related to an AnyReceiveEvent Trigger depends on the context of the Trigger (in particular, see sub clause [14.2](#) on Transitions and sub clause [16.10](#) on AcceptEventActions). An AnyReceiveEvent may also be triggered by the receipt of a message containing an object other than a SignalInstance, as may be sent by a SendObjectAction (see sub clause [16.3.3](#)).

A Trigger may also specify one or more ports, in which case the event of the Trigger shall be a MessageEvent. In this case the Trigger only matches event occurrences for messages received through one of the specified Ports (see also sub clause [11.3](#) on EncapsulatedClassifiers and Ports).

## Change Events

A ChangeEvent occurs when a Boolean changeExpression becomes true. For example, this could be as a result of a change in the value of some Attribute or a change in the value referenced by a link corresponding to an Association. A ChangeEvent occurs implicitly and is not the result of any explicit action.



An occurrence is considered to be generated any time the value of the `changeExpression` changes from false to true. However, it is not defined specifically when a `changeExpression` is evaluated or whether a `ChangeEvent` occurrence remains available for detection even if the associated `changeExpression` value changes back to false before the occurrence is consumed.

## Time Events

A `TimeEvent` specifies an instant in time at which it occurs. The instant is specified as a `TimeExpression` (see sub clause 8.4). The expression can be absolute or relative to some other instant. Relative `TimeEvents` shall always be used in the context of a `Trigger`, and the starting point is the time at which the `Trigger` becomes active.

### 13.3.4 Notation

There is no notation for Events outside of the context of their use in Triggers. A `Trigger` is denoted textually based on the kind of Event it is for:

```
<trigger> ::= <call-event> | <signal-event> | <any-receive-event> | <time-event> | <change-event>
```

where:

- A `CallEvent` is denoted by the name of the triggering Operation, optionally followed by an assignment specification:

```
<call-event> ::= <name> ['(' [<assignment-specification>] ')']
<assignment-specification> ::= <assigned-name> [',' <assigned-name>]*
```

where:

`<assigned-name>` is an implicit assignment of the argument value for the corresponding Parameter of the Operation to a Property or Variable of the context object for the triggered Behavior.

`<assignment-specification>` is optional and may be omitted even if the Operation has Parameters. No standard mapping is defined from an assignment specification to the UML abstract syntax. A conforming tool is not required to support this notation. If it does, it may provide a mapping to standard UML abstract syntax, e.g., by implicitly inserting Actions to carry out the behavior implied by the notation.

- A `SignalEvent` is denoted by name of the triggering Signal, optionally followed by an assignment specification:

```
<signal-event> ::= <name> ['(' [<assignment-specification>] ')']
<assignment-specification> ::= <attr-name> [',' <attr-name>]*
```

where `<assignment-specification>` is defined as for `CallEvent` above.

- Any `AnyReceiveEvent` is denoted by the keyword “all”:

```
<any-receive-event> ::= 'all'
```

- A `ChangeEvent` is denoted by the keyword “when” followed by a `Boolean ValueSpecification`:

```
<change-event> ::= 'when' <value-specification>
```

See Clause 8 for the notation for various kinds of `ValueSpecifications`.

- A relative `TimeEvent` is denoted with the keyword “after” followed by a `TimeExpression`, such as “after 5 seconds.” An absolute `TimeEvent` is specified with the keyword “at” followed by a `TimeExpression`, such as “at Jan. 1, 2000, Noon”.

```
<time-event> ::= <relative-time-event> | <absolute-time-event>
<relative-time-event> ::= 'after' <time-expression>
<absolute-time-event> ::= 'at' <time-expression>
```

See also sub clause [8.4.4](#) on the notation for TimeExpressions.

### **13.3.5 Examples**

None.

## 13.4 Classifier Descriptions

### AnyReceiveEvent [Class]

#### Description

A trigger for an AnyReceiveEvent is triggered by the receipt of any message that is not explicitly handled by any related trigger.

#### Diagrams

[Events](#)

#### Generalizations

[MessageEvent](#)

### Behavior [Abstract Class]

#### Description

Behavior is a specification of how its context BehavioredClassifier changes state over time. This specification may be either a definition of possible behavior execution or emergent behavior, or a selective illustration of an interesting subset of possible executions. The latter form is typically used for capturing examples, such as a trace of a particular execution.

#### Diagrams

[Behaviors](#), [Object Nodes](#), [Activities](#), [Control Nodes](#), [Expressions](#), [Structured Classifiers](#), [Behavior State Machines](#), [Interfaces](#), [Interactions](#), [Occurrences](#), [Features](#), [Invocation Actions](#), [Other Actions](#)

#### Generalizations

[Class](#)

#### Specializations

[OpaqueBehavior](#), [Activity](#), [StateMachine](#), [Interaction](#)

#### Attributes

- isReentrant : [Boolean](#) [1..1] = true  
Tells whether the Behavior can be invoked while it is still executing from a previous invocation.

#### Association Ends

- /context : [BehavioredClassifier](#) [0..1]{subsets [RedefinableElement::redefinitionContext](#)} (opposite [A\\_context\\_behavior::behavior](#))  
The BehavioredClassifier that is the context for the execution of the Behavior. A Behavior that is directly owned as a nestedClassifier does not have a context. Otherwise, to determine the context of a Behavior, find the first BehavioredClassifier reached by following the chain of owner relationships from the Behavior, if any. If there is such a BehavioredClassifier, then it is the context, unless it is itself a Behavior with a non-empty context, in which case that is also the context for the original Behavior. For example, following this algorithm, the context of an entry Behavior in a

StateMachine is the BehavioredClassifier that owns the StateMachine. The features of the context BehavioredClassifier as well as the Elements visible to the context Classifier are visible to the Behavior.

- ♦ ownedParameter : [Parameter](#) [0..\*]{ordered, subsets [Namespace::ownedMember](#)} (opposite [A\\_ownedParameter\\_behavior::behavior](#))  
References a list of Parameters to the Behavior which describes the order and type of arguments that can be given when the Behavior is invoked and of the values which will be returned when the Behavior completes its execution.
- ♦ ownedParameterSet : [ParameterSet](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [A\\_ownedParameterSet\\_behavior::behavior](#))  
The ParameterSets owned by this Behavior.
- ♦ postcondition : [Constraint](#) [0..\*]{subsets [Namespace::ownedRule](#)} (opposite [A\\_postcondition\\_behavior::behavior](#))  
An optional set of Constraints specifying what is fulfilled after the execution of the Behavior is completed, if its precondition was fulfilled before its invocation.
- ♦ precondition : [Constraint](#) [0..\*]{subsets [Namespace::ownedRule](#)} (opposite [A\\_precondition\\_behavior::behavior](#))  
An optional set of Constraints specifying what must be fulfilled before the Behavior is invoked.
- specification : [BehavioralFeature](#) [0..1] (opposite [BehavioralFeature::method](#))  
Designates a BehavioralFeature that the Behavior implements. The BehavioralFeature must be owned by the BehavioredClassifier that owns the Behavior or be inherited by it. The Parameters of the BehavioralFeature and the implementing Behavior must match. A Behavior does not need to have a specification, in which case it either is the classifierBehavior of a BehavioredClassifier or it can only be invoked by another Behavior of the Classifier.
- redefinedBehavior : [Behavior](#) [0..\*]{subsets [Classifier::redefinedClassifier](#)} (opposite [A\\_redefinedBehavior\\_behavior::behavior](#))  
References the Behavior that this Behavior redefines. A subtype of Behavior may redefine any other subtype of Behavior. If the Behavior implements a BehavioralFeature, it replaces the redefined Behavior. If the Behavior is a classifierBehavior, it extends the redefined Behavior.

## Operations

- context() : [BehavioredClassifier](#) [0..1]  
A Behavior that is directly owned as a nestedClassifier does not have a context. Otherwise, to determine the context of a Behavior, find the first BehavioredClassifier reached by following the chain of owner relationships from the Behavior, if any. If there is such a BehavioredClassifier, then it is the context, unless it is itself a Behavior with a non-empty context, in which case that is also the context for the original Behavior.

```
body: if nestingClass <> null then
    null
else
    let b:BehavioredClassifier = self.behavioredClassifier(self.owner) in
    if b.ooclIsKindOf(Behavior) and b.ooclAsType(Behavior)._'context' <> null then
        b.ooclAsType(Behavior)._'context'
    else
        b
    endif
endif
```

- behavioredClassifier(from : [Element](#)) : [BehavioredClassifier](#) [0..1]  
The first BehavioredClassifier reached by following the chain of owner relationships from the Behavior, if any.

```

body: if from.oclIsKindOf(BehavioredClassifier) then
    from.oclAsType(BehavioredClassifier)
else if from.owner = null then
    null
else
    self.behavioredClassifier(from.owner)
endif
endif

```

- `inputParameters() : Parameter [0..*]`  
The in and inout ownedParameters of the Behavior.

```

body: ownedParameter->select(direction=ParameterDirectionKind::_'in' or
direction=ParameterDirectionKind::inout)

```

- `outputParameters() : Parameter [0..*]`  
The out, inout and return ownedParameters.

```

body: ownedParameter->select(direction=ParameterDirectionKind::out or
direction=ParameterDirectionKind::inout or direction=ParameterDirectionKind::return)

```

## Constraints

- `most_one_behavior`  
There may be at most one Behavior for a given pairing of BehavioredClassifier (as owner of the Behavior) and BehavioralFeature (as specification of the Behavior).

```

inv: specification <> null implies _('context'.ownedBehavior-
>select(specification=self.specification)->size() = 1

```

- `parameters_match`  
If a Behavior has a specification BehavioralFeature, then it must have the same number of ownedParameters as its specification. The Behavior Parameters must also "match" the BehavioralParameter Parameters, but the exact requirements for this matching are not formalized.

```

inv: specification <> null implies ownedParameter->size() = specification.ownedParameter->size()

```

- `feature_of_context_classifier`  
The specification BehavioralFeature must be a feature (possibly inherited) of the context BehavioredClassifier of the Behavior.

```

inv: _('context'.feature->includes(specification)

```

## CallEvent [Class]

### Description

A CallEvent models the receipt by an object of a message invoking a call of an Operation.

### Diagrams

[Events](#)

## Generalizations

[MessageEvent](#)

## Association Ends

- operation : [Operation](#) [1..1] (opposite [A\\_operation\\_callEvent::callEvent](#))  
Designates the Operation whose invocation raised the CalEvent.

## ChangeEvent [Class]

### Description

A ChangeEvent models a change in the system configuration that makes a condition true.

### Diagrams

[Events](#)

## Generalizations

[Event](#)

## Association Ends

- ♦ changeExpression : [ValueSpecification](#) [1..1]{subsets [Element::ownedElement](#)} (opposite [A\\_changeExpression\\_changeEvent::changeEvent](#))  
A Boolean-valued ValueSpecification that will result in a ChangeEvent whenever its value changes from false to true.

## Event [Abstract Class]

### Description

An Event is the specification of some occurrence that may potentially trigger effects by an object.

### Diagrams

[Events](#)

## Generalizations

[PackageableElement](#)

## Specializations

[ChangeEvent](#), [MessageEvent](#), [TimeEvent](#)

## FunctionBehavior [Class]

### Description

A FunctionBehavior is an OpaqueBehavior that does not access or modify any objects or other external data.

### Diagrams

[Behaviors](#)

### Generalizations

[OpaqueBehavior](#)

### Operations

- `hasAllDataTypeAttributes(d : DataType) : Boolean`  
The `hasAllDataTypeAttributes` query tests whether the types of the attributes of the given `DataType` are all `DataTypes`, and similarly for all those `DataTypes`.

```
body: d.ownedAttribute->forall(a |
    a.type.ocIsKindOf(DataType) and
    hasAllDataTypeAttributes(a.type.ocAsType(DataType))
```

### Constraints

- `one_output_parameter`  
A FunctionBehavior has at least one output Parameter.
- `types_of_parameters`  
The types of the ownedParameters are all `DataTypes`, which may not nest anything but other `DataTypes`.

```
inv: self.ownedParameter->
    select(p | p.direction = ParameterDirectionKind::out or p.direction= ParameterDirectionKind::inout
    or p.direction= ParameterDirectionKind::return)->size() >= 1
```

```
inv: ownedParameter->forall(p | p.type <> null and
    p.type.ocIsTypeOf(DataType) and hasAllDataTypeAttributes(p.type.ocAsType(DataType)))
```

## MessageEvent [Abstract Class]

### Description

A MessageEvent specifies the receipt by an object of either an Operation call or a Signal instance.

### Diagrams

[Events](#)

### Generalizations

[Event](#)

## Specializations

[AnyReceiveEvent](#), [CallEvent](#), [SignalEvent](#)

## OpaqueBehavior [Class]

### Description

An OpaqueBehavior is a Behavior whose specification is given in a textual language other than UML.

### Diagrams

[Behaviors](#)

### Generalizations

[Behavior](#)

### Specializations

[FunctionBehavior](#)

### Attributes

- body : [String](#) [0..\*]  
Specifies the behavior in one or more languages.
- language : [String](#) [0..\*]  
Languages the body strings use in the same order as the body strings.

## SignalEvent [Class]

### Description

A SignalEvent represents the receipt of an asynchronous Signal instance.

### Diagrams

[Events](#)

### Generalizations

[MessageEvent](#)

### Association Ends

- signal : [Signal](#) [1..1] (opposite [A\\_signal\\_signalEvent::signalEvent](#))  
The specific Signal that is associated with this SignalEvent.



## TimeEvent [Class]

### Description

A TimeEvent is an Event that occurs at a specific point in time.

### Diagrams

[Events](#)

### Generalizations

[Event](#)

### Attributes

- isRelative : [Boolean](#) [1..1] = false  
Specifies whether the TimeEvent is specified as an absolute or relative time.

### Association Ends

- ♦ when : [TimeExpression](#) [1..1]{ subsets [Element::ownedElement](#) } (opposite [A\\_when\\_timeEvent::timeEvent](#))  
Specifies the time of the TimeEvent.

### Constraints

- when\_non\_negative  
The ValueSpecification when must return a non-negative Integer.

```
inv: when.integerValue() >= 0
```

## Trigger [Class]

### Description

A Trigger specifies a specific point at which an Event occurrence may trigger an effect in a Behavior. A Trigger may be qualified by the Port on which the Event occurred.

### Diagrams

[Events](#), [Behavior State Machines](#), [Accept Event Actions](#)

### Generalizations

[NamedElement](#)

## Association Ends

- event : [Event](#) [1..1] (opposite [A\\_event\\_trigger::trigger](#))  
The Event that detected by the Trigger.
- port : [Port](#) [0..\*] (opposite [A\\_port\\_trigger::trigger](#))  
A optional Port of through which the given effect is detected.

## Constraints

- trigger\_with\_ports  
If a Trigger specifies one or more ports, the event of the Trigger must be a MessageEvent.

```
inv: port->notEmpty() implies event.ocIsKindOf(MessageEvent)
```

## 13.5 Association Descriptions

### A\_changeExpression\_changeEvent [Association]

#### Diagrams

[Events](#)

#### Owned Ends

- changeEvent : [ChangeEvent](#) [0..1]{subsets [Element::owner](#)} (opposite [ChangeEvent::changeExpression](#))

### A\_context\_behavior [Association]

#### Diagrams

[Behaviors](#)

#### Owned Ends

- behavior : [Behavior](#) [0..\*]{subsets [A\\_redefinitionContext\\_redefinableElement::redefinableElement](#)} (opposite [Behavior::context](#))

### A\_event\_trigger [Association]

#### Diagrams

[Events](#)

### Owned Ends

- trigger : [Trigger](#) [0..\*] (opposite [Trigger::event](#))

### A\_operation\_callEvent [Association]

#### Diagrams

[Events](#)

### Owned Ends

- callEvent : [CallEvent](#) [0..\*] (opposite [CallEvent::operation](#))

### A\_ownedParameterSet\_behavior [Association]

#### Diagrams

[Behaviors](#)

### Owned Ends

- behavior : [Behavior](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [Behavior::ownedParameterSet](#))

### A\_ownedParameter\_behavior [Association]

#### Diagrams

[Behaviors](#)

### Owned Ends

- behavior : [Behavior](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [Behavior::ownedParameter](#))

### A\_port\_trigger [Association]

#### Diagrams

[Events](#)

## Owned Ends

- trigger : [Trigger](#) [0..\*] (opposite [Trigger::port](#))

## A\_postcondition\_behavior [Association]

### Diagrams

[Behaviors](#)

## Owned Ends

- behavior : [Behavior](#) [0..1]{subsets [Constraint::context](#)} (opposite [Behavior::postcondition](#))

## A\_precondition\_behavior [Association]

### Diagrams

[Behaviors](#)

## Owned Ends

- behavior : [Behavior](#) [0..1]{subsets [Constraint::context](#)} (opposite [Behavior::precondition](#))

## A\_redefinedBehavior\_behavior [Association]

### Diagrams

[Behaviors](#)

## Owned Ends

- behavior : [Behavior](#) [0..\*]{subsets [A\\_redefinedClassifier\\_classifier::classifier](#)} (opposite [Behavior::redefinedBehavior](#))

## A\_signal\_signalEvent [Association]

### Diagrams

[Events](#)

### Owned Ends

- signalEvent : [SignalEvent](#) [0..\*] (opposite [SignalEvent::signal](#))

### A\_when\_timeEvent [Association]

#### Diagrams

[Events](#)

### Owned Ends

- timeEvent : [TimeEvent](#) [0..1]{subsets [Element::owner](#)} (opposite [TimeEvent::when](#))

## 14 StateMachines

### 14.1 Summary

The StateMachines package defines a set of concepts that can be used for modeling discrete event-driven Behaviors using a finite state-machine formalism. In addition to expressing the Behavior of parts of a system (e.g., the Behavior of Classifier instances), state machines can also be used to express the valid interaction sequences, called *protocols*, for parts of a system. These two kinds of StateMachines are referred to as *behavior state machines* and *protocol state machines* respectively.

The specific form of finite state automata used in UML is based on an object-oriented variant of David Harel's *statecharts* formalism.

### 14.2 Behavior StateMachines

#### 14.2.1 Summary

Behavior StateMachines can be used to specify any of the following:

- The classifierBehavior of an active Class.
- An ownedBehavior of a BehavioredClassifier that is not the classifierBehavior of that BehavioredClassifier.
- A stand-alone Behavior, that is, one that does not have a corresponding BehavioredClassifier.
- A method corresponding to a BehavioralFeature (i.e., an Operation or a Reception).

## 14.2.2 Abstract Syntax

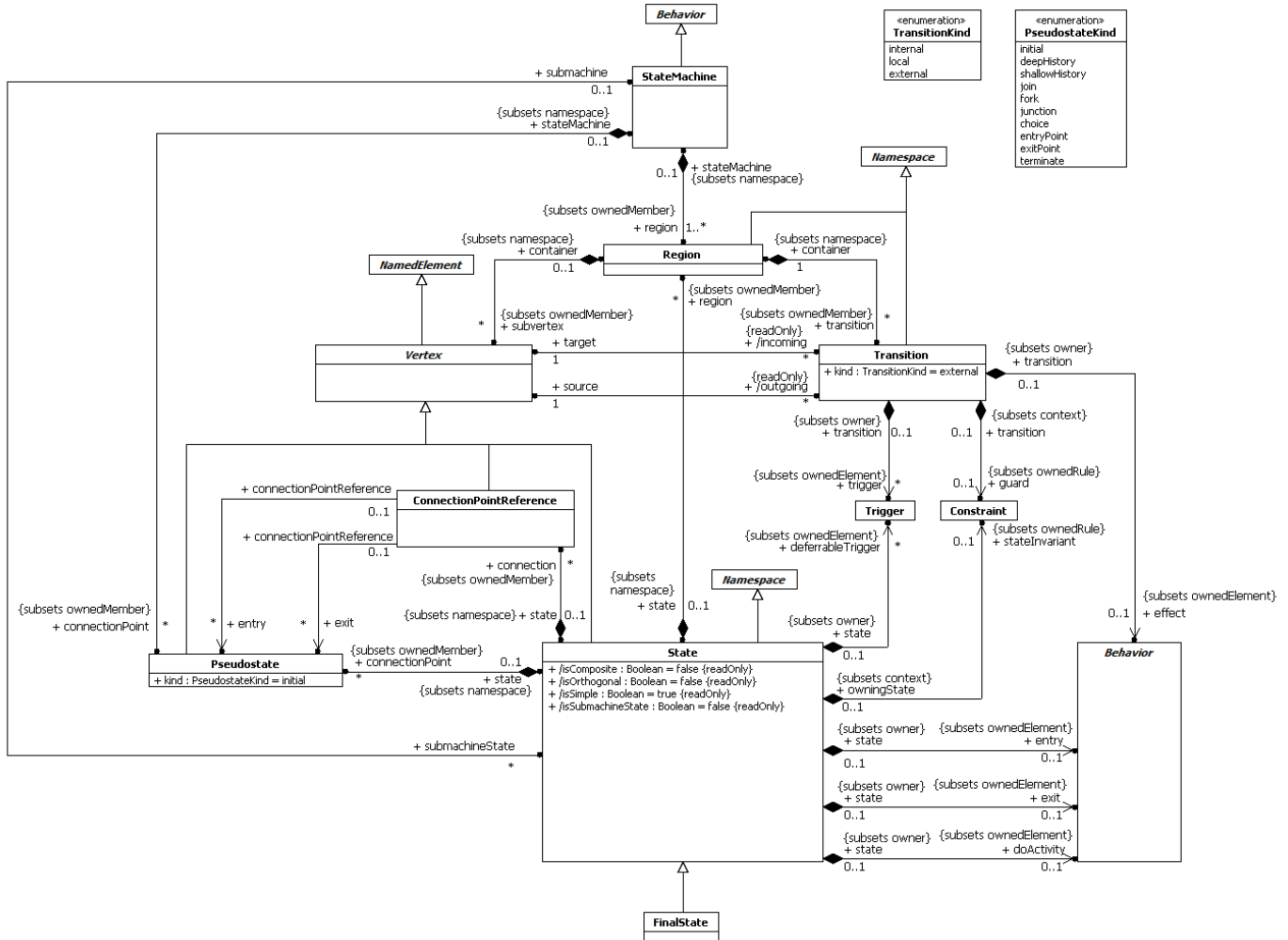


Figure 14.1 Behavior StateMachines

## 14.2.3 Semantics

### StateMachine

A *behavior* StateMachine comprises one or more Regions, each Region containing a graph (possibly hierarchical) comprising a set of Vertices interconnected by arcs representing Transitions. State machine execution is triggered by appropriate Event occurrences. A particular execution of a StateMachine is represented by a set of valid path traversals through one or more Region graphs, triggered by the dispatching of an Event occurrence that *match* active Triggers in these graphs. The rules for matching Triggers are described below. In the course of such a traversal, a StateMachine instance may execute a potentially complex sequence of Behaviors associated with the particular elements of the graphs that are being traversed (transition effects, state entry and state exit Behaviors, etc.)

If the StateMachine has a kind of BehavedClassifier context, then that Classifier defines which Signal and CallEvent triggers are applicable to that StateMachine, and which Features are available to the Behaviors owned by the StateMachine. Signal

Triggers and CallEvent Triggers for the StateMachine are defined according to the Receptions and Operations of this Classifier respectively. These Features may be used to define message event Triggers of the StateMachine.

If the StateMachine has no BehavedClassifier context (i.e., it is a stand-alone Behavior), then its Triggers do not need to be tied to any Receptions or Operations of some Classifier. For example, such a StateMachine might be defined as a Template with its Triggers defined as TemplateParameters. Such a StateMachine can then be reused with different context Classifiers by binding appropriate CallEvent or SignalEvent Triggers to these TemplateParameters.

In situations where a StateMachine specifies the method of a BehavioralFeature (Operation or Reception), the Parameters of the StateMachine shall match the Parameters of the BehavioralFeature (see sub clause [13.2.3](#)). This is the means by which the StateMachine execution accesses the Parameters of the BehavioralFeature. Otherwise, the method by which an executing StateMachine instance accesses the dispatched Event occurrence and its associated data is not defined (see Clause [13](#)).

By definition, invocations of StateMachine executions result in triggered effects (see sub clause [13.3.3](#)) and, hence, there is an associated event pool with such an execution. The event pool for a StateMachine execution belongs to either its context Classifier object or, if the StateMachine defines a method of a BehavioralFeature, to the instance of the Classifier owning the BehavioralFeature.

Due to its event-driven nature, a StateMachine execution is either *in transit* or *in state*, alternating between the two. It is in transit when an event is dispatched that matches at least one of its associated Triggers. While in transit, it may execute a number of Behaviors associated with the paths it is taking.

**NOTE.** A StateMachine execution may be executing Behaviors even when it has settled in a stable state configuration, in cases where there are doActivity Behaviors associated with its active state configuration.

## Regions

A Region denotes a behavior fragment that may execute concurrently with its orthogonal Regions. Two or more Regions are orthogonal to each other if they are either owned by the same State or, at the topmost level, by the same StateMachine. A Region becomes *active* (i.e., it begins executing) either when its owning State is entered or, if it is directly owned by a StateMachine (i.e., it is a top level Region), when its owning StateMachine starts executing. Each Region owns a set of Vertices and Transitions, which determine the behavioral flow within that Region. It may have its own **initial** Pseudostate as well as its own FinalState.

A *default activation* of a Region occurs if the Region is entered implicitly, that is, it is not entered through an incoming Transition that terminates on one of its component Vertices (e.g., a State or a **history** Pseudostate), but either

- through a (**local** or **external**) Transition that terminates on the containing State or,
- in case of a top level Region, when the StateMachine starts executing.

*Default activation* means that execution starts with the Transition originating from the **initial** Pseudostate of the Region, if one is defined. No one approach is defined for the case when there is no **initial** Pseudostate exists within the Region. One possible approach is to deem the model ill defined. An alternative is that the Region remains inactive, although the State that contains it is active. In other words, the containing composite State is treated as a simple (leaf) State.

Conversely, an *explicit activation* occurs when a Region is entered by a Transition terminating on one of the Region's contained Vertices. When one Region of an orthogonal State is activated explicitly, this will result in the default activation of all of its orthogonal Regions, unless those Regions are also entered explicitly (multiple orthogonal Regions can be entered explicitly in parallel through Transitions originating from the same **fork** Pseudostate).

## Vertices

Vertex is an abstract class that captures the common characteristics for a variety of different concrete kinds of nodes in the StateMachine graph (States, Pseudostates, or ConnectionPointReferences). With certain exceptions described below, a Vertex can be the source and/or target of any number of Transitions. The semantics of a Vertex depend on the concrete kind of node it represents. In general, Pseudostates and ConnectionPointReferences are *transitive*, in the sense that a compound transition



execution simply passes through them, arriving on an incoming Transition and leaving on an outgoing Transition without pause. State and FinalState, however, represent *stable* Vertices, such that, when a StateMachine execution enters them it remains in them until either some Event occurs that triggers a transition that moves it to a different State or the StateMachine is terminated.

The semantics of individual types of Vertices are described below.

## States

A State models a situation in the execution of a StateMachine Behavior during which some invariant condition holds. In most cases this condition is not explicitly defined, but is implied, usually through the name associated with the State. For example, in Figure 14.34, which models the behavior of a telephone unit, the states “Idle” and “Active” represent situations where the telephone is and is not being used, respectively. This example also illustrates the fact that a State need not necessarily represent a fully static situation, as there is clearly some detailed activity occurring in the context of the “Active” state. However, throughout all that activity the telephone remains in use (i.e., “active”).

### *Kinds of States*

The following kinds of States are distinguished:

- *simple* State (isSimple = **true**)
- *composite* State (isComposite = **true**)
- *submachine* State (isSubmachineState = **true**)

A simple State has no internal Vertices or Transitions. A composite State contains at least one Region, whereas a submachine State refers to an entire StateMachine, which is, conceptually, deemed to be “nested” within the State. A composite State can be either a simple composite State with exactly one Region or an orthogonal State with multiple Regions (isOrthogonal = **true**). For example, in Figure 14.8, State “CourseAttempt” is an example of a composite State with a single Region, whereas State “Studying” is a composite State that contains three Regions.

Any State enclosed within a Region of a composite State is called a *substate* of that composite State. It is called a *direct substate* when it is not contained in any other State; otherwise, it is referred to as an *indirect substate*.

### *State configurations*

In general, a StateMachine can have multiple Regions, each of which may contain States of its own, some of which may be composites with their own multiple Regions, etc. Consequently, a particular “state” of an executing StateMachine instance is represented by one or more hierarchies of States, starting with the topmost Regions of the StateMachine and down through the composition hierarchy to the simple, or *leaf*, States. Similarly, we can talk about such a hierarchy of substates within a composite State. This complex hierarchy of States is referred to as a *state configuration* (of a State or a StateMachine) For example, one valid state configuration for an execution of the StateMachine depicted in Figure 14.8 is: <CourseAttempt - Studying – (Studying::Lab2, Studying::TermProject, Studying::FinalTest)>. An executing StateMachine instance can only be in exactly one state configuration at a time, which is referred to as its *active state configuration*. StateMachine execution is represented by transitions from one active state configuration to another in response to Event occurrences that match the Triggers of the StateMachine.

A State is said to be active if it is part of the active state configuration.

A state configuration is said to be *stable* when:

- no further Transitions from that state configuration are enabled and
- all the entry Behaviors of that configuration, if present, have completed (but not necessarily the doActivity Behaviors of that configuration, which, if defined, may continue executing).

### *State entry, exit, and doActivity Behaviors*

A State may have an associated entry Behavior. This Behavior, if defined, is executed whenever the State is entered through an external Transition. In addition, a State may also have an associated exit Behavior, which, if defined, is executed whenever the State is exited.

A State may also have an associated doActivity Behavior. This Behavior commences execution when the State is entered (but only after the State entry Behavior has completed) and executes concurrently with any other Behaviors that may be associated with the State, until:

- it completes (in which case a completion event is generated) or
- the State is exited, in which case execution of the doActivity Behavior is aborted.

### *State history*

The concept of State *history* was introduced by David Harel in the original statechart formalism. It is a convenience concept associated with composite States whereby a State keeps track of the state configuration it was in when it was last exited. This allows easy return to that same state configuration, if desired, the next time the State becomes active (e.g., after returning from handling an interrupt), or if there is a local Transition within the composite state that returns to its history. This is achieved simply by terminating a Transition on the desired type of history Pseudostate inside the State. The advantage provided by this facility is that it eliminates the need for users to explicitly keep track of history in cases where this type of behavior is desired, which can result in significantly simpler state machine models.

Two types of history Pseudostates are provided. *Deep history* (**deepHistory**) represents the *full* state configuration of the most recent visit to the containing State. The effect is the same as if the Transition terminating on the **deepHistory** Pseudostate had, instead, terminated on the innermost State of the preserved state configuration, including execution of all entry Behaviors encountered along the way. *Shallow history* (**shallowHistory**) represents a return to only the *topmost substate* of the most recent state configuration, which is entered using the default entry rule.

In cases where a Transition terminates on a history Pseudostate when the State has not been entered before (i.e., no prior history) or it had reached its FinalState, there is an option to force a transition to a specific substate, using the *default history mechanism*. This is a Transition that originates in the history Pseudostate and terminates on a specific Vertex (the *default history state*) of the State containing the history Pseudostate. This Transition is only taken if execution leads to the history Pseudostate and the State had never been active before. Otherwise, the appropriate history entry into the State is executed (see above). If no default history Transition is defined, then standard default entry of the State is performed as explained below.

### *Deferred Events*

A State may specify a set of Event types that may be deferred in that State. This means that Event occurrences of those types will not be dispatched as long as that State remains active. Instead, these Event occurrences remain in the event pool until:

- a state configuration is reached where these Event types are no longer deferred or,
- if a deferred Event type is used explicitly in a Trigger of a Transition whose source is the deferring State (i.e., a kind of override option).

An Event may be deferred by a composite State or submachine States, in which case it remains deferred as long as the composite State remains in the active configuration.

### *Entering a State*

The semantics of entering a State depend on the type of State and the manner in which it is entered. However, in all cases, the entry Behavior of the State is executed (if defined) upon entry, but only after any effect Behavior associated with the incoming Transition is completed. Also, if a doActivity Behavior is defined for the State, this Behavior commences execution immediately after the entry Behavior is executed. It executes concurrently with any subsequent Behaviors associated with entering the State, such as the entry Behaviors of substates entered as part of the same compound transition.

The above description fully covers the case of simple States. For composite States with a single Region the following alternatives exist:

- *Default entry*: This situation occurs when the composite State is the direct target of a Transition (graphically, this is indicated by an incoming Transition that terminates on the outside edge of the composite State). After executing the entry Behavior and forking a possible doActivity Behavior execution, if an **initial** Pseudostate is defined, State entry continues from that Vertex via its outgoing Transition (known as the default Transition of the State). If no **initial** Pseudostate is defined, there is no single approach defined. One alternative is to treat such a model as ill formed. A second alternative is to treat the composite State as a simple State, terminating the traversal on that State despite its internal parts.
- *Explicit entry*: If the incoming Transition or its continuations terminate on a directly contained substate of the composite State, then that substate becomes active and its entry Behavior is executed after the execution of the entry Behavior of the containing composite State. This rule applies recursively if the Transition terminates on an indirect (deeply nested) substate.
- *Shallow history entry*: If the incoming Transition terminates on a **shallowHistory** Pseudostate of the composite State, the active substate becomes the substate that was most recently active prior to this entry, unless:
  - the most recently active substate is the FinalState, or
  - this is the first entry into this State.
  - In the latter two cases, if a default shallow history Transition is defined originating from the **shallowHistory** Pseudostate, it will be taken. Otherwise, default State entry is applied.
- *Deep history entry*: The rule for this case is the same as for shallow history except that the target Pseudostate is of type **deepHistory** and the rule is applied recursively to all levels in the active state configuration below this one.
- *Entry point entry*: If a Transition enters a composite State through an **entryPoint** Pseudostate, then the effect Behavior associated with the outgoing Transition originating from the entry point and penetrating into the State (but after the entry Behavior of the composite State has been executed).

If the composite State is also an orthogonal State with multiple Regions, each of its Regions is also entered, either by default or explicitly. If the Transition terminates on the edge of the composite State (i.e., without entering the State), then all the Regions are entered using the default entry rule above. If the Transition explicitly enters one or more Regions (in case of a fork), these Regions are entered explicitly and the others by default.

#### *Exiting a State*

When exiting a State, regardless of whether it is simple or composite, the final step involved in the exit, *after all other Behaviors associated with the exit are completed*, is the execution of the exit Behavior of that State. If the State has a doActivity Behavior that is still executing when the State is exited, that Behavior is aborted *before* the exit Behavior commences execution.

When exiting from a composite State, exit commences with the innermost State in the active state configuration. This means that exit Behaviors are executed in sequence starting with the innermost active State. If the exit occurs through an **exitPoint** Pseudostate, then the exit Behavior of the State is executed after the effect Behavior of the Transition terminating on the exit point.

When exiting from an orthogonal State, each of its Regions is exited. After that, the exit Behavior of the State is executed.

#### *Encapsulated composite States*

In some modeling situations, it is useful to encapsulate a composite State, by not allowing Transitions to penetrate directly into the State to terminate on one of its internal Vertices. (One common use case for this is when the internals of a State in an abstract Classifier are intended to be specified differently in different subtype refinements of the abstract Classifier.) Despite the encapsulation, it is often necessary to bind the internal elements of the composite State with incoming and outgoing Transitions. This is done by means of entry and exit points, which are realized via the **entryPoint** and **exitPoint** Pseudostates.

*Entry points* represent termination points (sources) for incoming Transitions and origination points (targets) for Transitions that terminate on some internal Vertex of the composite State. In effect, the latter is a continuation of the external incoming Transition, with the proviso that the execution of the entry Behavior of the composite State (if defined) occurs between the effect Behavior of the incoming Transition and the effect Behavior of the outgoing Transition. If there is no outgoing Transition inside the composite State, then the incoming Transition simply performs a default State entry.

Exit points are the inverse of entry points. That is, Transitions originating from a Vertex within the composite State can terminate on the exit point. In a well-formed model, such a Transition should have a corresponding external Transition outgoing from the same exit point, representing a continuation of the terminating Transition. If the composite State has an exit Behavior defined, it is executed after any effect Behavior of the incoming inside Transition and before any effect Behavior of the outgoing external Transition.

### *Submachine States and submachines*

Submachines are a means by which a single StateMachine specification can be reused multiple times. They are similar to encapsulated composite States in that they need to bind incoming and outgoing Transitions to their internal Vertices. However, whereas encapsulated composite States and their internals are contained within the StateMachine in which they are defined, submachines are, like programming language macros, distinct Behavior specifications, which may be defined in a different context than the one where they are used (invoked). Consequently, they require a more complex binding. This is achieved through the concept of *submachine State* (i.e., States with `isSubmachineState = true`), which represent references to corresponding submachine StateMachines. The concept of `ConnectionPointReference` is provided to support binding between the submachine State and the referenced StateMachine. A `ConnectionPointReference` represents a point on the submachine State at which a Transition either terminates or originates. That is, they serve as targets for incoming Transitions to submachine States, as well as sources for outgoing Transitions from submachine States. Each `ConnectionPointReference` is matched by a corresponding entry or exit point in the referenced submachine StateMachine. This provides the necessary binding mechanism between the submachine invocation and its specification.

A submachine State implies a macro-like insertion of the specification of the corresponding submachine StateMachine. It is, therefore, semantically equivalent to a composite State. The Regions of the submachine StateMachine are the Regions of the composite State. The entry, exit, and effect Behaviors and internal Transitions are defined as contained in the submachine State.

**NOTE.** Each submachine State represents a distinct instantiation of a submachine, even when two or more submachine States reference the same submachine.

A submachine StateMachine can be entered via its default (**initial**) Pseudostate or via any of its entry points (i.e., it may imply entering a non-orthogonal or an orthogonal composite State with Regions). Entering via the **initial** Pseudostate has the same meaning as for ordinary composite States. An entry point is equivalent to a **junction** Pseudostate (**fork** in cases where the composite State is orthogonal): Entering via an entry point implies that the entry Behavior of the composite state is executed, followed by the Transition from the entry point to the target Vertex within the composite State. Any guards associated with these entry point Transitions must evaluate to **true** in order for the specification to be well formed.

Similarly, a submachine StateMachine can be exited as a result of:

- reaching its `FinalState`,
- triggering of a group Transition originating from a submachine State, or
- via any of its exit points.

Exiting via a `FinalState` or by a group Transition has the same meaning as for ordinary composite States.

### **ConnectionPointReference**

As noted above, a connection point reference represents a usage (as part of a submachine State) of an entry/exit point defined in the StateMachine referenced by the submachine State. Connection point references of a submachine State can be used as

sources/targets of Transitions. They represent entries into or exits out of the submachine StateMachine referenced by the submachine State.

Connection point references are sources/targets of Transitions implying exits out of/entries into the submachine StateMachine referenced by a submachine State.

An entry point connection point reference as the target of a Transition implies that the target of the Transition is the **entryPoint** Pseudostate as defined in the submachine of the submachine State. As a result, the Regions of the submachine StateMachine are entered through the corresponding **entryPoint** Pseudostates.

An exit point connection point reference as the source of a Transition implies that the source of the Transition is the exit point Pseudostate as defined in the submachine of the submachine State that has the exit point connection point defined. When a Region of the submachine StateMachine reaches the corresponding exit point, the submachine state is exited via this exit point.

### FinalState

FinalState is a special kind of State signifying that the enclosing Region has completed. Thus, a Transition to a FinalState represents the completion of the behaviors of the Region containing the FinalState.

### Pseudostate and PseudostateKind

A Pseudostate is an abstraction that encompasses different types of transient Vertices in the StateMachine graph. Pseudostates are generally used to chain multiple Transitions into more complex *compound transitions* (see below). For example, by combining a Transition entering a **fork** Pseudostate with a set of Transitions exiting that Pseudostate, we get a compound Transition that can enter a set of orthogonal Regions.

The specific semantics of a Pseudostate depend on the kind of Pseudostate, which is defined by its kind attribute of type PseudostateKind. The following describes the different kinds and their semantics:

- **initial** - An **initial** Pseudostate represents a starting point for a Region; that is, it is the point from which execution of its contained behavior commences when the Region is entered via default activation. It is the source for at most one Transition, which may have an associated effect Behavior, but not an associated trigger or guard. There can be at most one **initial** Vertex in a Region.
- **deepHistory** – This type of Pseudostate is a kind of variable that represents the most recent active state configuration of its owning State. As explained above, a Transition terminating on this Pseudostate implies restoring the State to that same state configuration, but with all the semantics of entering a State (see the Subclause describing State entry). The entry Behaviors of all States in the restored state configuration are performed in the appropriate order starting with the outermost State. A **deepHistory** Pseudostates can only be defined for composite States and, at most one such Pseudostate can be contained in a composite State.
- **shallowHistory** – As explained above, this type of Pseudostate is a kind of variable that represents the most recent active substate of its containing State, but not the substates of that substate. A Transition terminating on this Pseudostate implies restoring the composite State to that substate with all the semantics of entering a State. A single outgoing Transition from this Pseudostate may be defined terminating on a substate of the composite State. This substate is the *default shallow history state* of the composite State. A **shallowHistory** Pseudostates can only be defined for composite States and, at most one such Pseudostate can be included in a composite State.
- **join** – This type of Pseudostate serves as a common target Vertex for two or more Transitions originating from Vertices in different orthogonal Regions. Transitions terminating on a **join** Pseudostate cannot have a guard or a trigger. Similar to junction points in Petri nets, **join** Pseudostates perform a synchronization function, whereby all incoming Transitions have to complete before execution can continue through an outgoing Transition.
- **fork** – **fork** Pseudostates serve to split an incoming Transition into two or more Transitions terminating on Vertices in orthogonal Regions of a composite State. The Transitions outgoing from a **fork** Pseudostate cannot have a guard or a trigger.

- **junction** – This type of Pseudostate is used to connect multiple Transitions into compound paths between States. For example, a **junction** Pseudostate can be used to merge multiple incoming Transitions into a single outgoing Transition representing a shared continuation path. Or, it can be used to split an incoming Transition into multiple outgoing Transition segments with different guard Constraints.

**NOTE.** Such guard Constraints are evaluated *before* any compound transition containing this Pseudostate is executed, which is why this is referred to as a *static* conditional branch.

It may happen that, for a particular compound transition, the configuration of Transition paths and guard values is such that the compound transition is prevented from reaching a valid state configuration. In those cases, the entire compound transition is disabled even though its Triggers are enabled. (As a way of avoiding this situation in some cases, it is possible to associate a predefined guard denoted as “else” with at most one outgoing Transition. This Transition is enabled if all the guards attached to the other Transitions evaluate to false). If more than one guard evaluates to **true**, one of these is chosen. The algorithm for making this selection is not defined.

- **choice** – This type of Pseudostate is similar to a **junction** Pseudostate (see above) and serves similar purposes, with the difference that the guard Constraints on all outgoing Transitions are evaluated *dynamically*, when the compound transition traversal reaches this Pseudostate. Consequently, **choice** is used to realize a dynamic conditional branch. It allows splitting of compound transitions into multiple alternative paths such that the decision on which path to take may depend on the results of Behavior executions performed in the same compound transition prior to reaching the choice point. If more than one guard evaluates to **true**, one of the corresponding Transitions is selected. The algorithm for making this selection is not defined. If none of the guards evaluates to **true**, then the model is considered ill formed. To avoid this, it is recommended to define one outgoing Transition with the predefined “else” guard for every **choice** Pseudostate.
- **entryPoint** – An **entryPoint** Pseudostate represents an entry point for a StateMachine or a composite State that provides encapsulation of the insides of the State or StateMachine. In each Region of the StateMachine or composite State owning the entryPoint, there is at most a single Transition from the entry point to a Vertex within that Region.  
**NOTE.** If the owning State has an associated entry Behavior, this Behavior is executed before any behavior associated with the outgoing Transition. If multiple Regions are involved, the entry point acts as a **fork** Pseudostate.
- **exitPoint** – An **exitPoint** Pseudostate is an exit point of a StateMachine or composite State that provides encapsulation of the insides of the State or StateMachine. Transitions terminating on an exit point within any Region of the composite State or a StateMachine referenced by a submachine State implies exiting of this composite State or submachine State (with execution of its associated exit Behavior). If multiple Transitions from orthogonal Regions within the State terminate on this Pseudostate, then it acts like a **join** Pseudostate.
- **terminate** – Entering a **terminate** Pseudostate implies that the execution of the StateMachine is terminated immediately. The StateMachine does not exit any States nor does it perform any exit Behaviors. Any executing doActivity Behaviors are automatically aborted. Entering a **terminate** Pseudostate is equivalent to invoking a DestroyObjectAction.

## Transitions

A Transition is a single directed arc originating from a single source Vertex and terminating on a single target Vertex (the source and target may be the same Vertex), which specifies a valid fragment of a StateMachine Behavior. It may have an associated effect Behavior, which is executed when the Transition is traversed (executed).

**NOTE.** The duration of a Transition traversal is undefined, allowing for different semantic interpretations, including both “zero” and non-“zero” time.

Transitions are executed as part of a more complex *compound transition* that takes a StateMachine execution from one stable state configuration to another. The semantics of compound transitions are described below.

In the course of execution, a Transition instance is said to be:

- *reached*, when execution of its StateMachine execution has reached its source Vertex (i.e., its source State is in the active state configuration);
- *traversed*, when it is being executed (along with any associated effect Behavior); and
- *completed*, after it has reached its target Vertex.

A Transition may own a set of Triggers, each of which specifies an Event whose occurrence, when dispatched, may trigger traversal of the Transition. A Transition trigger is said to be *enabled* if the dispatched Event occurrence matches its Event type.

#### *Transition kinds relative to source*

The semantics of a Transition depend on its relationship to its source Vertex. Three different possibilities are defined, depending on the value of the Transition's kind attribute:

- kind = **external** means that the Transition exits its source Vertex. If the Vertex is a State, then executing this Transition will result in the execution of any associated exit Behavior of that State.
- kind = **local** is the opposite of external, meaning that the Transition does not exit its containing State (and, hence, the exit Behavior of the containing State will not be executed). However, for local Transitions the target Vertex must be different from its source Vertex. A local Transition can only exist within a composite State.
- kind = **internal** is a special case of a local Transition that is a self-transition (i.e., with the same source and target States), such that the State is never exited (and, thus, not re-entered), which means that no exit or entry Behaviors are executed when this Transition is executed. This kind of Transition can only be defined if the source Vertex is a State.

#### *High-level (group) Transitions*

Transitions whose source Vertex is a composite States are called *high-level* or *group* Transitions. If they are external, group Transitions result in the exiting of all substates of the composite State, executing any defined exit Behaviors starting with the innermost States in the active state configuration. In case of local Transitions, the exit Behaviors of the source State and the entry Behaviors of the target State will be executed, but not those of the containing State.

#### *Completion Transitions and completion events*

A special kind of Transition is a completion Transition, which has an implicit trigger. The event that enables this trigger is called a *completion event* and it signifies that all Behaviors associated with the source State of the completion Transition have completed execution. In case of simple States, a completion event is generated when the associated entry and doActivity Behaviors have completed executing. If no such Behaviors are defined, the completion event is generated upon entry into the State. For composite or submachine States, a completion event is generated under the following circumstances:

- All internal activities (e.g., entry and doActivity Behaviors) have completed execution, and
- if the State is a composite State, all its orthogonal Regions have reached a FinalState, or
- if the State is a submachine State, the submachine StateMachine execution has reached a FinalState.

Completion events have dispatching priority. That is, they are dispatched ahead of any pending Event occurrences in the event pool. Completion of all top level Regions in a StateMachine corresponds to a completion of the Behavior of the StateMachine and results in its termination.

#### *Transition guards*

A Transition may have an associated guard Constraint. Transitions that have a guard which evaluates to false are disabled. Guards are evaluated before the compound transition that contains them is enabled, *unless they are on Transitions that originate from a choice Pseudostate*. In the latter case, the guards are evaluated when the choice point is reached. A Transition that does not have an associated guard is treated as if it has a guard that is always **true**.

**NOTE.** A completion Transition may also have a guard.

A guard constraint may involve tests of orthogonal States of the current StateMachine, or explicitly designated States of some reachable object (for example, “in State1” or “not in State2”). State names may be fully qualified by the nested States and Regions that contain them, yielding pathnames of the form “RegionA::State1::Region1::State2::State3”. This may be used in case the same State name occurs in different composite State Regions.

#### *Compound transitions*

As noted earlier, when an Event occurrence triggers an enabled Transition or a StateMachine execution is created, this can initiate traversal of a set of connected and nested Transitions and Vertices until a stable state configuration is reached. In the general case, a trace of this traversal, known as a *compound transition*, can be represented by an acyclical directed graph. The root (source) of this graph can be one of the following:

- A Transition with one or more Triggers defined.
- A completion Transition.
- A set of Transitions (including, possibly, completion Transitions) originating from different orthogonal Regions that converge on a common **join** Pseudostate.
- A Transition originating from an **initial** Pseudostate of the topmost Region (i.e., a Region owned by the StateMachine); this variant applies only to cases when the StateMachine instance is created.

Branching in a compound transition execution occurs whenever an executing Transition performs a default entry into a State with multiple orthogonal Regions, with a separate branch created for each Region, or when a **fork** Pseudostate is encountered. The overall behavior that results from the execution of a compound transition is a partially ordered set of executions of Behaviors associated with the traversed elements, determined by the order in which the elements (Vertices and Transitions) are encountered. For example, if a Transition entering a compound State terminates on a substate of that State, then the effect Behavior of the Transition would be executed before the execution of the entry Behavior of the compound State, followed by the entry Behavior of the substate. If a **fork** Pseudostate is encountered in the traversal, then the effect Behaviors of the individual outgoing branches are, at least conceptually, executed concurrently with each other.

If a **choice** or **join** point is reached with multiple outgoing Transitions with guards, a Transition whose guard evaluates to **true** will be taken. If more than one guard evaluates to **true**, one of these Transitions is chosen for continuing the traversal. The algorithm for making this selection is undefined. In case of Transitions originating from a **choice** Pseudostate, if no guards evaluate to **true** when the Pseudostate is reached, the model is ill formed.

#### *Transition ownership*

The owner of a Transition is not explicitly constrained, though the Region in which it is contained must be owned directly or indirectly by the owning StateMachine. A suggested owner of a Transition is the LCA Least Common Ancestor of the source and target Vertices (see below).

## **Event Processing for StateMachines**

#### *The run-to-completion paradigm*

The processing of Event occurrences by a StateMachine execution conforms to the general semantics defined in Clause 13. Upon creation, a StateMachine will perform its initialization during which it executes an initial compound transition prompted by the creation, after which it enters a *wait point*. In case of StateMachine Behaviors, a wait point is represented by a stable state configuration. It remains thus until an Event stored in its event pool is dispatched. This Event is evaluated and, if it matches a valid Trigger of the StateMachine and there is at least one enabled Transition that can be triggered by that Event occurrence, a single StateMachine *step* is executed. A step involves executing a compound transition and terminating on a stable state configuration (i.e., the next wait point). This cycle then repeats until either the StateMachine completes its Behavior or until it is asynchronously terminated by some external agent.



StateMachines can respond to any of the Event types described in Clause [13](#) as well as to completion events (see above).

**NOTE.** As explained above, completion events have priority and will be dispatched ahead of any pending Event occurrences in the event pool.

Event occurrences are detected, dispatched, and processed by the StateMachine execution, *one at a time*.

**NOTE.** The order of event dispatching is left undefined, allowing for varied scheduling algorithms.

This cycle is referred to as the *run-to-completion paradigm*, and the corresponding StateMachine step is called a *run-to-completion step*. Run-to-completion means that, in the absence of exceptions or asynchronous destruction of the context Classifier object or the StateMachine execution, a pending Event occurrence is dispatched only *after the processing of the previous occurrence is completed* and a stable state configuration has been reached. That is, an Event occurrence will never be dispatched while the StateMachine execution is busy processing the previous one. This behavioral paradigm was chosen to avoid complications arising from concurrency conflicts that may arise when a StateMachine tries to respond to multiple concurrent or overlapping events.

When an Event occurrence is detected and dispatched, it may result in one or more Transitions being *enabled* for firing. If no Transition is enabled and the corresponding Event type is not in any of the *deferrableTriggers* lists of the active state configuration, the dispatched Event occurrence is discarded and the run-to-completion step is completed trivially.

Due to the presence of orthogonal Regions, it is possible that multiple Transitions (in different Regions) can be triggered by the same Event occurrence. The order in which these Transitions are executed is left undefined. Each orthogonal Region in the active state configuration that does not contain nested orthogonal Regions (i.e., a “bottom-level” Region) can fire at most one Transition as a result of the current Event occurrence. When all orthogonal Regions have finished executing the Transition, the current Event occurrence is fully consumed, and the run-to-completion step is completed.

As mentioned above, it is possible for multiple mutually exclusive Transitions in a given Region to be enabled for firing by the same Event occurrence. In those cases, only one is selected and executed. Which of the enabled Transitions is chosen is determined by the Transition selection algorithm described below.

During a Transition, a number of actions Behaviors may be executed. If such a Behavior includes a synchronous invocation call on another object executing a StateMachine, then the Transition step is not completed until the invoked object method completes its run-to-completion step.

Run-to-completion may be implemented in various ways. For active Classes, it may be realized by an event-loop running in its own thread, and that reads event occurrences from a pool. For passive Classes it may be implemented using a monitor .

**IMPLEMENTATION NOTE.** Run-to-completion is often mistakenly interpreted as implying that an executing StateMachine cannot be interrupted, which, of course would lead to priority inversion issues in some time-sensitive systems. However, this is not the case; in a given implementation a thread executing a StateMachine step can be suspended, allowing higher-priority threads to run, and, once it is allocated processor time again by the underlying thread scheduler, it can safely resume its execution and complete its event processing.

#### *Enabled Transitions*

A Transition is enabled if and only if:

- All of its source States are in the active state configuration.
- At least one of the triggers of the Transition has an Event that is matched by the Event type of the dispatched Event occurrence. In case of Signal Events, any occurrence of the same or compatible type as specified in the Trigger will match. If one of the Triggers is for an AnyReceiveEvent, then either a Signal or CallEvent satisfies this Trigger, provided that there is no other Signal or CallEvent Trigger for the same Transition or any other Transition having the same source Vertex as the Transition with the AnyReceiveEvent trigger (see also [13.3.1](#)).

- If there exists at least one full path from the source state configuration to either the target state configuration or to a dynamic **choice** Pseudostate in which all guard conditions are **true** (Transitions without guards are treated as if their guards are always **true**).

As more than one Transition may be enabled by the same Event occurrence, being enabled is a necessary but not sufficient condition for the firing of a Transition.

#### *Conflicting Transitions*

It is possible for more than one Transition to be enabled within a StateMachine. If that happens, then such Transitions may be *in conflict* with each other. For example, consider the case of two Transitions originating from the same State, triggered by the same event, but with different guards. If that event occurs and both guard conditions are **true**, then at most one of those Transition can fire in a given run-to-completion step.

Two Transitions are said to conflict if they both exit the same State, or, more precisely, that the intersection of the set of States they exit is non-empty. Only Transitions that occur in mutually orthogonal Regions may be fired simultaneously. This constraint guarantees that the new active state configuration resulting from executing the set of Transitions is well formed.

An internal Transition in a State conflicts only with Transitions that cause an exit from that State.

#### *Firing priorities*

In situations where there are conflicting Transitions, the selection of which Transitions will fire is based in part on an implicit priority. These priorities resolve some but not all Transition conflicts, as they only define a partial ordering. The priorities of conflicting Transitions are based on their relative position in the state hierarchy. By definition, a Transition originating from a substate has higher priority than a conflicting Transition originating from any of its containing States.

The priority of a Transition is defined based on its source State. The priority of Transitions chained in a compound transition is based on the priority of the Transition with the most deeply nested source State.

In general, if t1 is a Transition whose source State is s1, and t2 has source s2, then:

- If s1 is a direct or indirectly nested substate of s2, then t1 has higher priority than t2.
- If s1 and s2 are not in the same state configuration, then there is no priority difference between t1 and t2.

#### *Transition selection algorithm*

The set of Transitions that will fire is a maximal set of Transitions that satisfies the following conditions:

- All Transitions in the set are enabled.
- There are no conflicting Transitions within the set.
- There is no Transition outside the set that has higher priority than a Transition in the set (that is, enabled Transitions with highest priorities are in the set while conflicting Transitions with lower priorities are left out).

This can be implemented by a greedy selection algorithm, with a straightforward traversal of the active state configuration. States in the active state configuration are traversed starting with the innermost nested simple States and working outwards. For each State at a given level, all originating Transitions are evaluated to determine if they are enabled. This traversal guarantees that the priority principle is not violated. The only non-trivial issue is resolving Transition conflicts across orthogonal States on all levels. This is resolved by terminating the search in each orthogonal State once a Transition inside any one of its components is fired.

#### *Transition execution sequence*

Every Transition, except for internal and local Transitions, causes exiting of a source State, and entering of the target State. These two States, which may be composite, are designated as the *main source* and the *main target* of a Transition respectively.

The *least common ancestor* (LCA) of a compound transition is the innermost Region that contains (directly or indirectly) both the source and target States of the compound transition.

The main source is a direct substate of the Region that contains the source States, and the main target is the substate of the Region that contains the target States.

**NOTE.** A Transition from one Region to another in the same immediate enclosing composite State is not allowed.

Once a Transition is enabled and is selected to fire, the following steps are carried out in order:

- The main source State is exited.
- Behaviors are executed in sequence following their linear order along the segments of the Transition: the closer the Behavior to the source State, the earlier it is executed.
- If a choice point is encountered, the guards following that choice point are evaluated dynamically and a path whose guards are **true** is selected.
- The main target State is entered.

## 14.2.4 Notation

### StateMachine Diagrams

StateMachine diagrams specify StateMachines. This Clause outlines the graphic elements that may be shown in StateMachine diagrams, and provides cross references where detailed information about the semantics and concrete notation for each element can be found. It also furnishes examples that illustrate how the graphic elements can be assembled into diagrams.

A StateMachine diagram is a graph that represents a StateMachine. States and various other types of Vertices in the StateMachine graph are rendered by appropriate State and Pseudostate symbols, while Transitions are generally rendered by directed arcs that connect them, or by control symbols representing the actions of the Behavior on the Transition.

### StateMachine

When depicting StateMachine redefinition in a class diagram, the default rectangle notation for Classifier can be used, with the keyword «StateMachine» inside the name compartment above or before the name of the StateMachine.

The association between a StateMachine and its **context** Classifier or BehavioralFeatures does not have a special graphical representation.

### Region

A composite State or StateMachine with Regions is shown by tiling the graph Region of the State/StateMachine using dashed lines to divide it into Regions (Figure 14.2). Each Region may have an optional name and contains the nested disjoint States and the Transitions between these. The text compartments of the entire State are separated from the orthogonal Regions by a solid line.

A composite State or StateMachine with just one Region is shown by showing a nested state diagram within the graph Region.

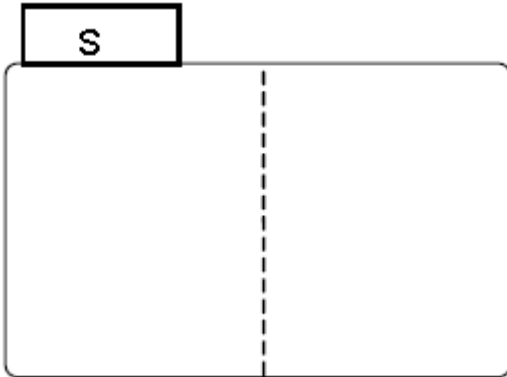


Figure 14.2 Notation for a composite State with Regions

### State

State is shown as a rectangle with rounded corners, with the State name shown within (Figure 14.3).

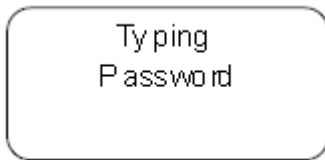


Figure 14.3 State notation

Optionally, it may have an attached name tab (Figure 14.4). The name tab is a rectangle, usually resting on the outside of the top side of a State and it contains the name of that State. It is normally used to keep the name of a composite State that has orthogonal Regions, but may be used in other cases as well.

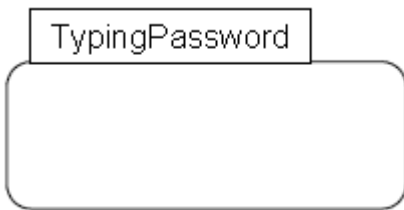
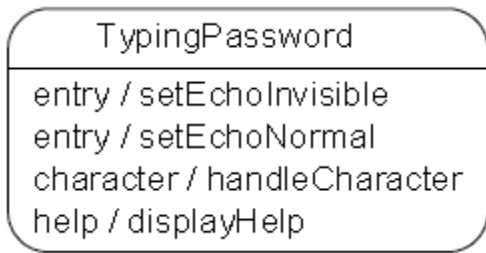


Figure 14.4 State with a name tab

A State may be subdivided into multiple compartments separated from each other by a horizontal line (Figure 14.5).



**Figure 14.5 State with compartments**

The compartments of a State are:

- name compartment
- internal Behaviors compartment
- internal Transitions compartment.

A composite State also has a:

- decomposition compartment.

Each of these compartments is described below.

- Name compartment

This compartment holds the (optional) name of the State, as a string. States without names are anonymous and are all distinct. It is undesirable to show the same named State twice in the same diagram, as confusion may ensue, unless control icons are used to show a Transition-oriented view of the StateMachine. Name compartments should not be used if a name tab is used and vice versa.

In case of a submachine State, the name of the referenced StateMachine is shown as a string following ‘:’ after the name of the State.

- Internal activities Behaviors compartment

This compartment holds a list of internal Behaviors associated with a State. Each entry has the following format:

*<behavior-type-label> [ '/' <behavior-expression> ]*

The *<behavior-type-label>* identifies the circumstances under which the Behavior specified by the *<behavior-expression>* will be invoked and can be one of the following:

- **entry** — This label identifies a Behavior, specified by the corresponding expression, which is performed upon entry to the State (entry Behavior).
- **exit** — This label identifies a Behavior, specified by the corresponding expression, that is performed upon exit from the State (exit Behavior).
- **do** — This label identifies an ongoing Behavior (doActivity Behavior) that is performed as long as the modeled element is in the State or until the computation specified by the expression is completed.

The optional *<behavior-expression>* is an expression in some textual surface language, which may be either a vendor-specific or some standard language (see sub clause [16.1](#)).

- Internal Transition compartment

This compartment contains a list of internal Transitions, where each item has the following syntax:

$\{ \langle trigger \rangle \}^* [ [ ' \langle guard \rangle ' ] ] [ / \langle behavior-expression \rangle ]$

Where  $\langle trigger \rangle$  is the notation for Triggers (see sub clause 13.3.4),  $\langle guard \rangle$  is a Boolean expression for a guard, and the optional  $\langle behavior-expression \rangle$  is the specification of the effect Behavior to be executed if the Event occurrence matches the trigger and guard of the internal Transition. It is an expression written in some textual surface language, which may be either a vendor-specific or some standard language (see sub clause 16.1).

Alternatively, in place of a textual behavior expression the various Behaviors associated with a State or internal Transition can be expressed using the appropriate graphical representation (e.g., an activity diagram).

### Composite State

- decomposition compartment

This compartment shows its composition structure in terms of Regions, States, and Transition. In addition to the (optional) name and internal Transition compartments, the State may have an additional compartment that contains a nested diagram. For convenience and appearance, the text compartments may be shrunk horizontally within the graphic Region.

In some cases, it is convenient to hide the decomposition of a composite State. For example, there may be a large number of States nested inside a composite State and they may simply not fit in the graphical space available for the diagram. In that case, the composite State may be represented by a simple State graphic with a special “composite” icon, usually in the lower right-hand corner (see Figure 14.7). This icon, consisting of two horizontally placed and connected States, is an optional visual cue that the State has a decomposition that is not shown in this particular diagram. Instead, the contents of the composite State are shown in a separate diagram.

**NOTE.** The “hiding” here is purely a matter of graphical convenience and has no semantic significance in terms of access restrictions.

A composite State may have one or more entry and exit points on its outside border or in close proximity of that border (inside or outside).

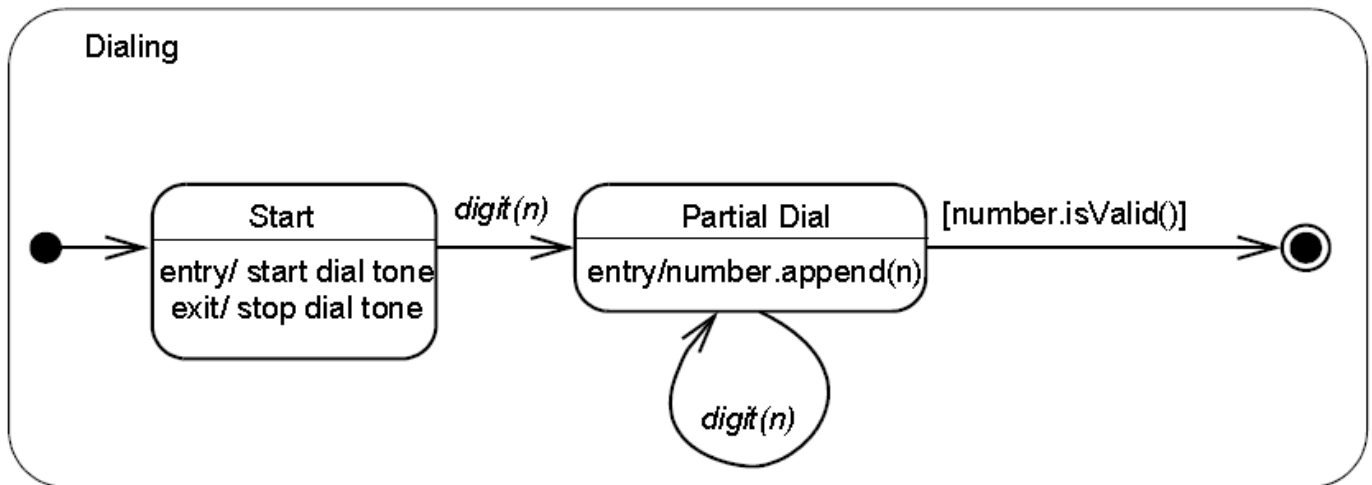


Figure 14.6 Composite State with two States

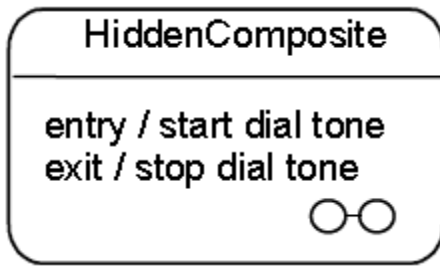


Figure 14.7 Composite State with a hidden decomposition indicator icon

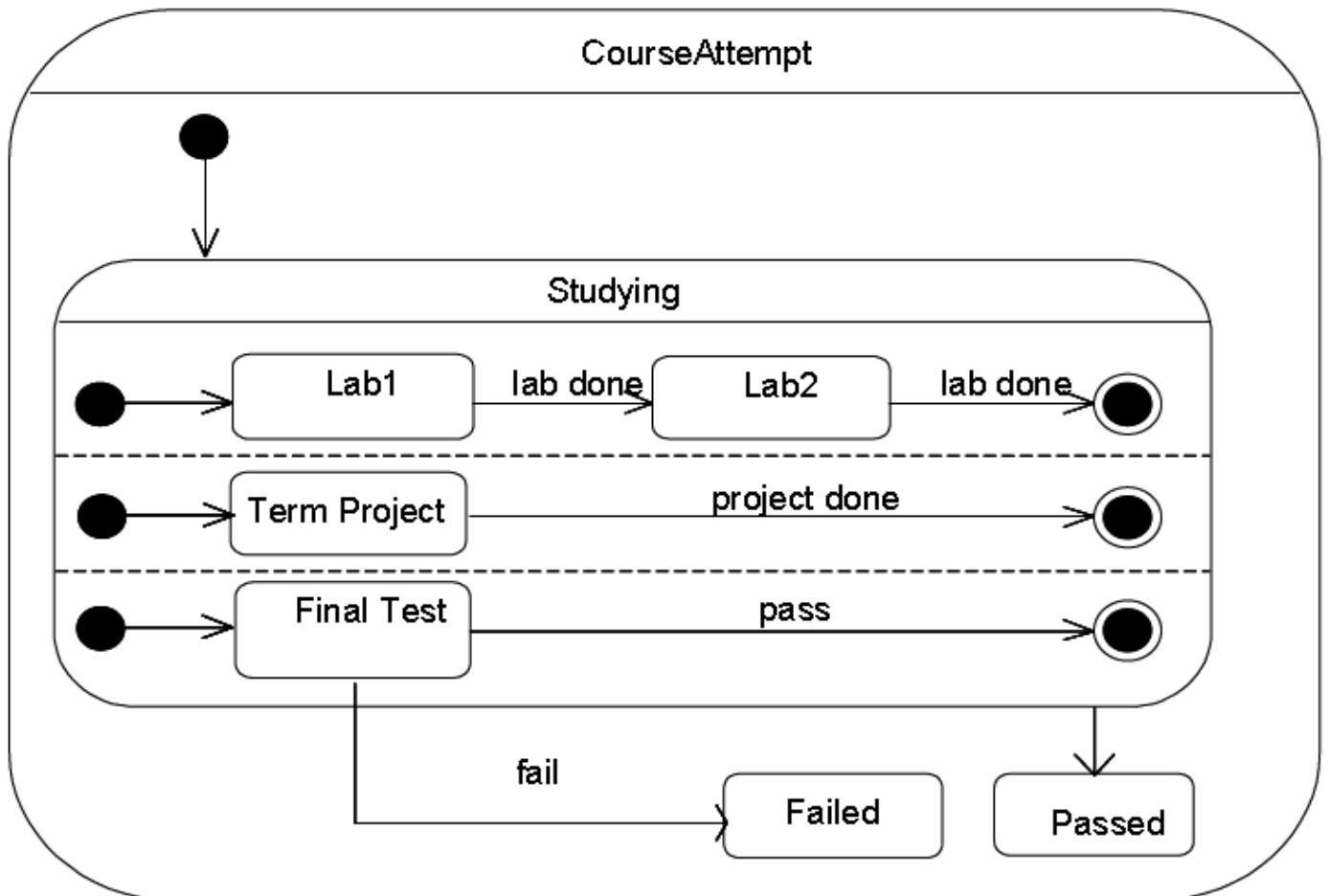


Figure 14.8 Composite State with Regions

*Submachine State*

The submachine State is depicted as a normal State where the string in the name compartment has the following syntax:

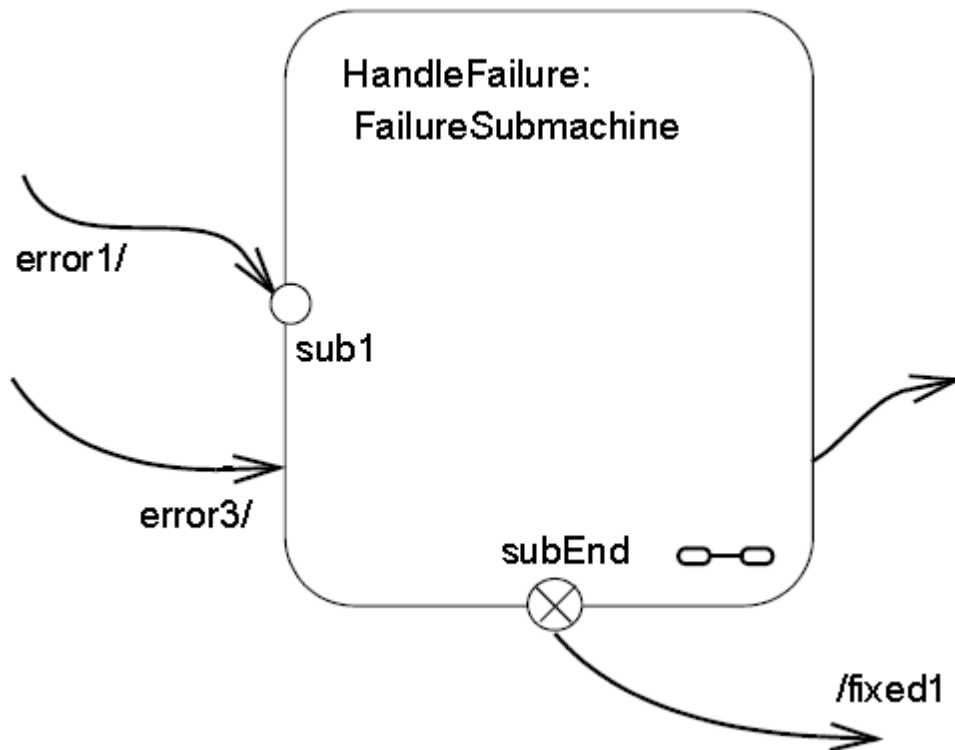
*<state-name> ':' <name-of-referenced-StateMachine>*

The submachine State symbol may contain the references to one or more entry points and to one or more exit points. The notation for these connection point references comprises entry/exit Pseudostates on the border of the submachine State. The names are the names of the corresponding entry/exit points defined within the referenced StateMachine (see ConnectionPointReference).

If the submachine StateMachine is entered through its default **initial** Pseudostate or if it is exited as a result of the completion of the submachine, it is not necessary to use the entry/exit point notation. Similarly, an exit point is not required if the exit occurs through an explicit group Transition that originates from the boundary of the submachine State (implying that it applies to all the substates of the submachine).

Submachine States invoking the same submachine may occur multiple times in the same state diagram with the entry and exit points being part of different Transitions.

The diagram in Figure 14.9 shows a fragment from a StateMachine diagram in which a submachine State (the FailureSubmachine) is referenced. The actual submachine StateMachine is defined in some enclosing or imported namespace.



**Figure 14.9 Submachine State example**

In the above example, the Transition triggered by Event “error1” will terminate on entry point “sub1” of the FailureSubmachine StateMachine. The “error3” Transition implies taking the default Transition of the FailureSubmachine.

The Transition originating from the “subEnd” exit point of the submachine will execute the “fixed1” Behavior in addition to what is executed within the HandleFailure StateMachine. This Transition must have been triggered within the HandleFailure StateMachine. Finally, the Transition originating from the edge of the submachine State is taken as a result of the completion event generated when the FailureSubmachine reaches its FinalState.

**NOTE.** The same notation would apply to composite States with the exception that there would be no reference to a StateMachine in the State name.

Figure 14.10 is an example of a StateMachine defined with two exit points. Entry and exit points may be shown on the frame or within the state graph. Figure 14.10 is an example of a StateMachine defined with an exit point shown within the state graph. Figure 14.11 shows the same StateMachine using a notation shown on the frame of the StateMachine.



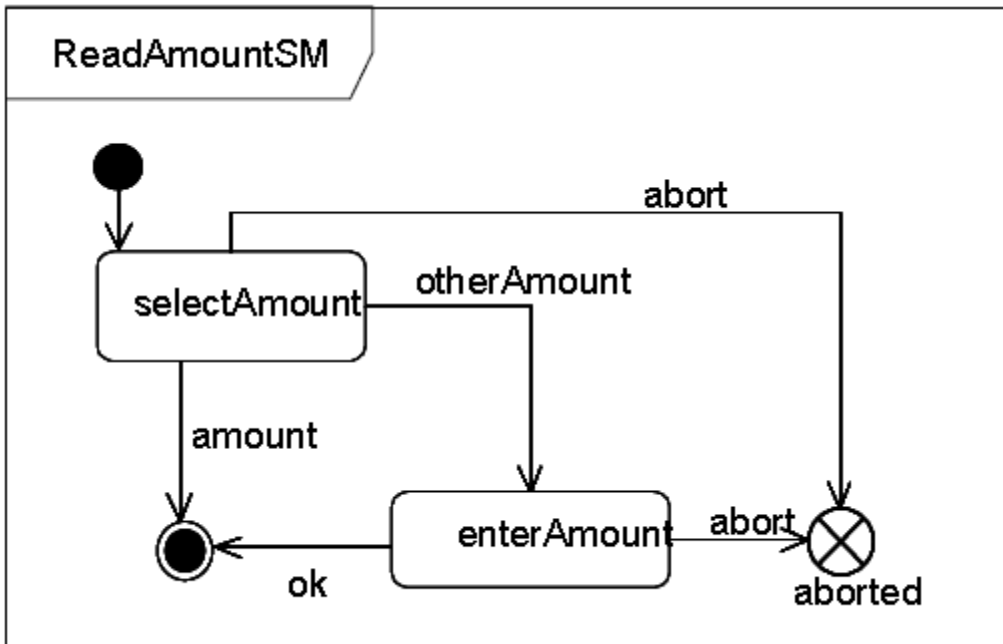


Figure 14.10 StateMachine with an exit point as part of the StateMachine graph

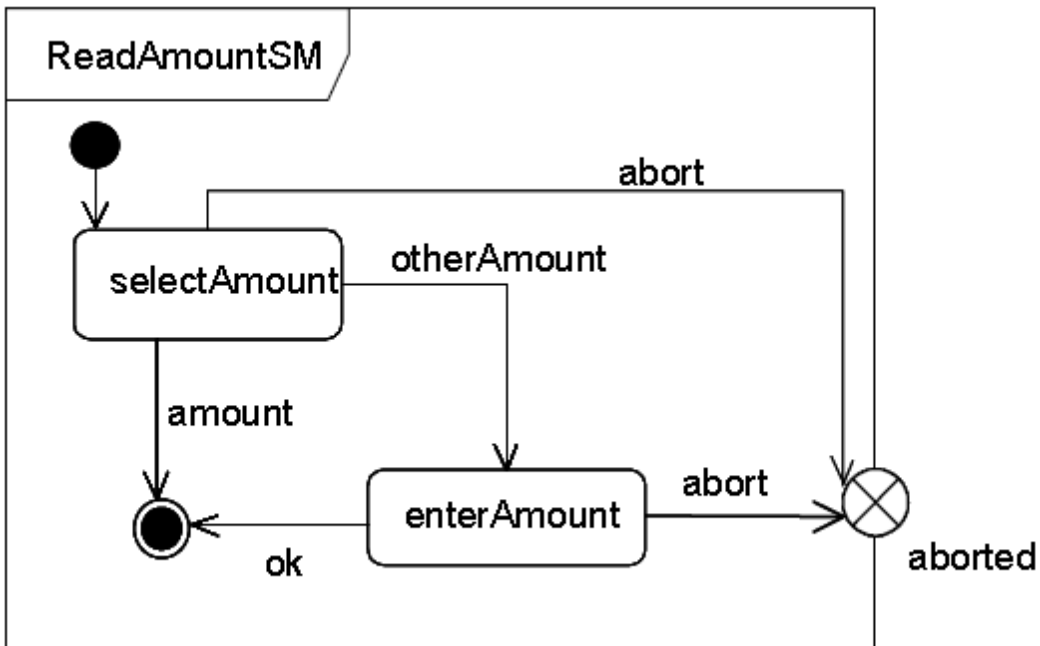


Figure 14.11 StateMachine with an exit point on the border

In Figure 14.12 the StateMachine shown in Figure 14.11 is referenced in a submachine State, and the presentation option with the exit points on the State symbol is shown.

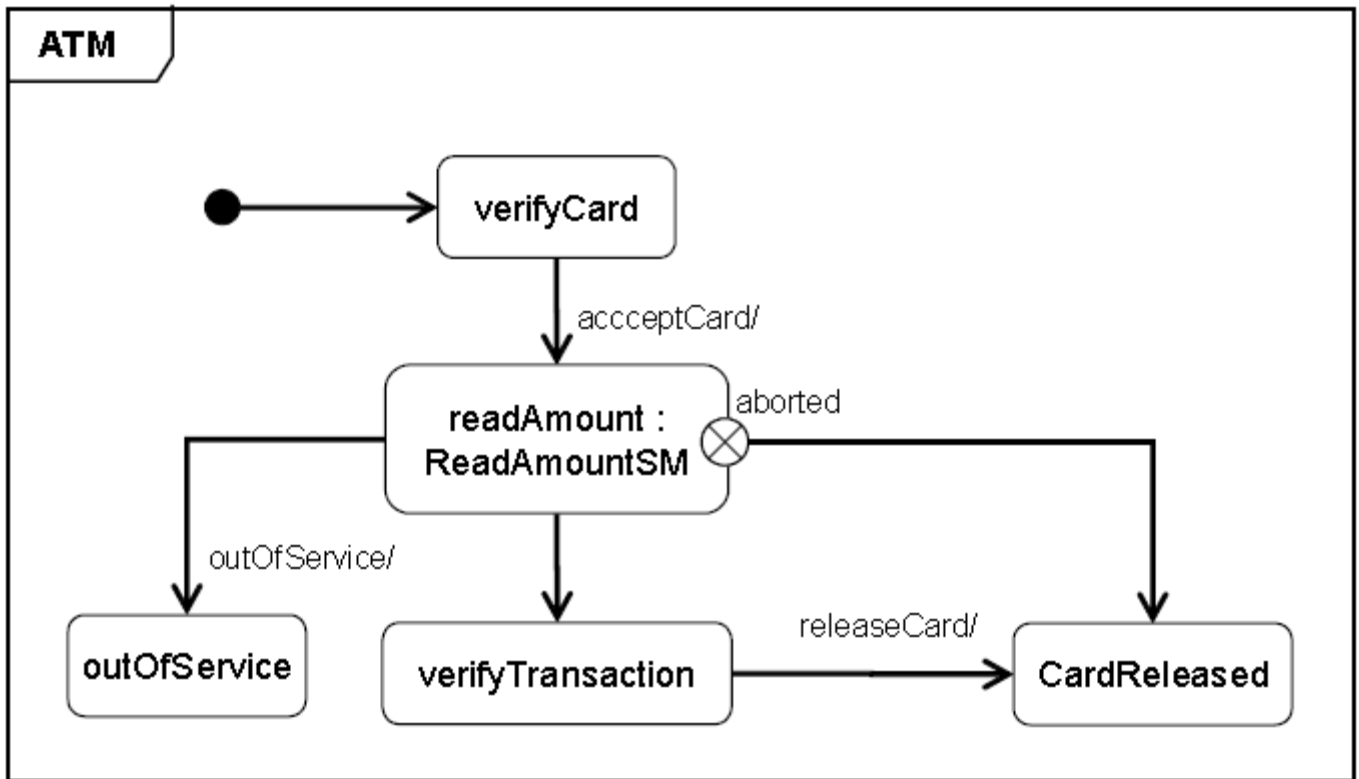


Figure 14.12 Submachine State that uses an exit point

An example of the notation for entry and exit points for composite States is shown in Figure 14.21.

*State list notation*

State lists provide a graphical shortcut for certain situations that sometimes occur in practice.

**NOTE.** These are purely notational forms with no corresponding abstract syntax representation. Consequently, there is no normative way by which such graphical representations can be exchanged between conformant tools, although vendors might provide implementation-specific support for such an interchange.

Multiple effect-free Transitions with the same Trigger values originating on different States but all either (a) targeting a common **junction** Vertex with a single outgoing Transition or (b) terminating on the same target State, may be represented by a Single Transition-like arc originating from a State-like graphic element, labeled with a list of the names of the originating States. This arc terminates on the joint target State. For example, transition e in Figure 14.13, is equivalent to two distinct Transitions, one from State “S1” and the second from State “S2”, with both terminating on State “S3”.

Similarly, two or more effect-free Transitions originating from different States but with the same Triggers and all terminating on a history Vertex can be rendered by a Transition-like directed arc originating and terminating on the same state list symbol (e.g., Figure 14.13 and Figure 14.14).

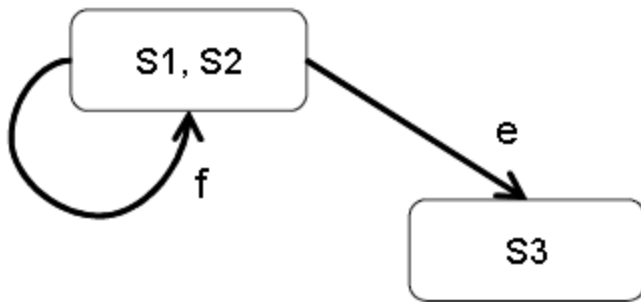


Figure 14.13 State list notation option



Figure 14.14 State lists

### FinalState

A FinalState is shown as a circle surrounding a small solid filled circle (see Figure 14.15). The corresponding completion Transition on the enclosing State has as notation an unlabeled Transition.



Figure 14.15 FinalState notation

Figure 14.6 has an example of a FinalState (the right-most of the States within the composite State).

### Pseudostate

An **initial** Pseudostate is shown as a small solid filled circle (see Figure 14.16). In a Region of a ClassifierBehavior StateMachine, the Transition from an **initial** Pseudostate may be labeled with the Event type of the occurrence that creates the object; otherwise, it must be unlabeled. If it is unlabeled, it represents any Transition from the enclosing State.



Figure 14.16 initial Pseudostate

A **shallowHistory** Pseudostate is indicated by a small circle containing an 'H' (see Figure 14.17). It applies to the State Region that directly encloses it.



Figure 14.17 shallowHistory Pseudostate

A **deepHistory** Pseudostate is indicated by a small circle containing an 'H\*' (see Figure 14.18). It applies to the State Region that directly encloses it.



**Figure 14.18 deepHistory Pseudostate**

An entry point is shown as a small circle on the border of the StateMachine diagram or composite State, with the name associated with it (see Figure 14.19).



**Figure 14.19 entryPoint Pseudostate**

Optionally it may be placed both within the StateMachine diagram and outside the border of the StateMachine diagram or composite State.

An exit point is shown as a small circle with a cross on the border of the StateMachine diagram or composite State, with the name associated with it (see Figure 14.20).



**Figure 14.20 exitPoint Pseudostate**

Optionally, and exit point symbol may be placed both within the StateMachine diagram or composite State and outside the border of the StateMachine diagram or composite State. Figure 14.21 illustrates the notation for depicting entry and exit points of composite States.

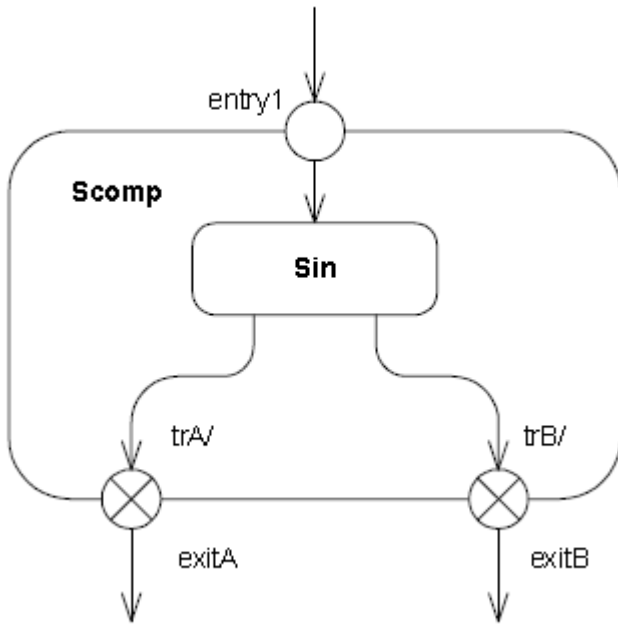


Figure 14.21 entryPoint and exitPoints on a composite State

Alternatively, the “bracket” notation shown in Figure 14.28 and Figure 14.29 can also be used for the transition-oriented notation. A **junction** is represented by a small filled circle (see Figure 14.22).

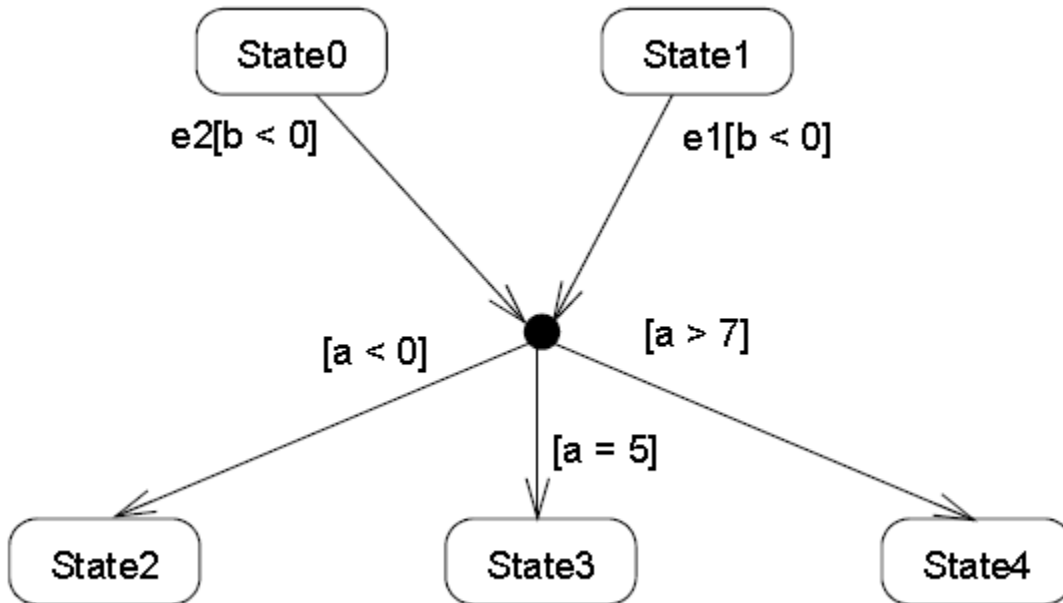


Figure 14.22 junction Pseudostate with incoming and outgoing Transitions

A **choice** Pseudostate is shown as a diamond-shaped symbol as exemplified shown by the left-hand diagram in Figure 14.23.

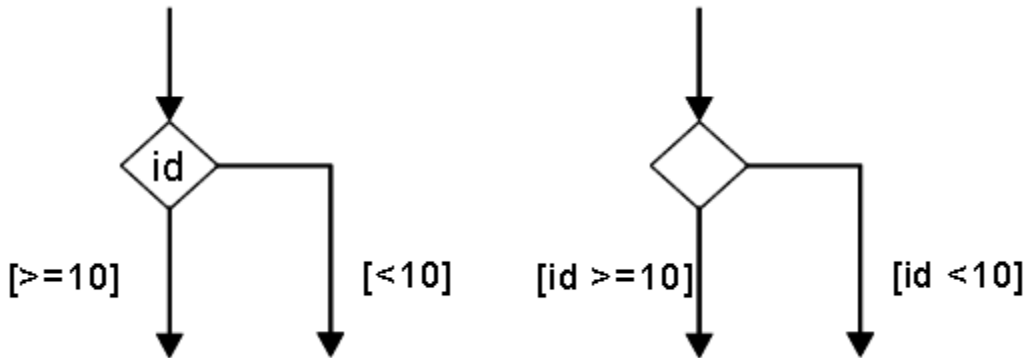


Figure 14.23 choice Pseudostates

**NOTE.** In cases when all guards associated with triggers of Transitions leaving a **choice** Pseudostate are binary expressions that share a common left operand, then the notation for **choice** Pseudostate may be simplified. The left operand may be placed inside the diamond-shaped symbol and the rest of the Guard expressions placed on the outgoing Transitions. This is illustrated by the right-hand diagram in Figure 14.23.

A **terminate** Pseudostate is shown as a cross, see Figure 14.24.



Figure 14.24 terminate Pseudostate

The notation for a **fork** and **join** is a short heavy bar (Figure 14.25). The bar may have one or more arrows from source States to the bar (when representing a joint). The bar may have one or more arrows from the bar to States (when representing a fork). A Transition string may be shown near the bar.

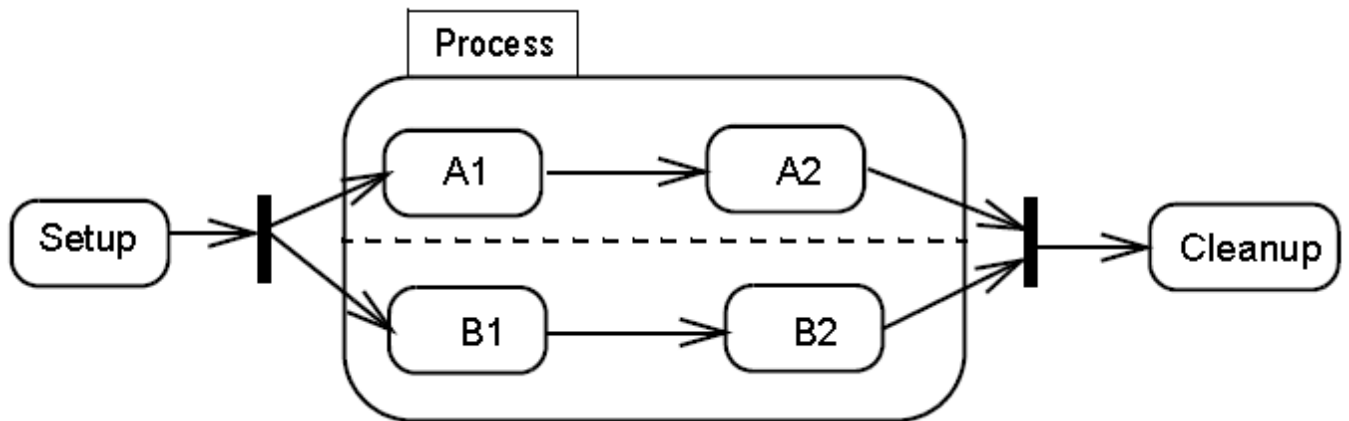
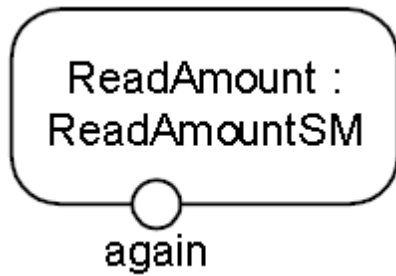


Figure 14.25 fork and join Pseudostates

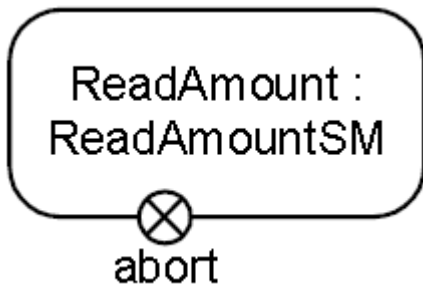
### ConnectionPointReference

A connection point reference to an entry point has the same notation as an entry Pseudostate. The circle is placed on the border of the State symbol of a submachine State.



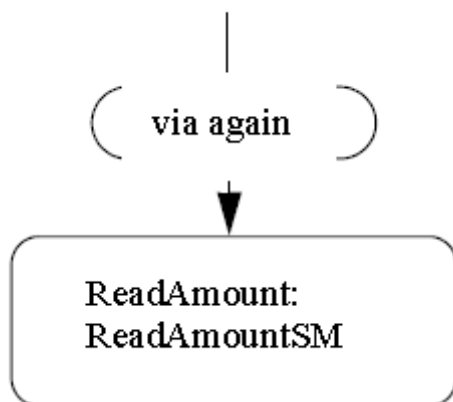
**Figure 14.26 Entry point ConnectionPointReference notation**

A connection point reference to an exit point has the same notation as an exit Pseudostate. The encircled cross is placed on the border of the State symbol of a submachine State.



**Figure 14.27 Exit point ConnectionPointReference notation**

Alternatively, a connection point reference to an entry point can also be visualized using a “bracketed space” symbol as shown in Figure 14.28. The text inside the symbol shall contain the keyword ‘via’ followed by the name of the connection point. This notation may only be used if the Transition ending with the connection point is defined using the graphical Transition notation, such as the one shown in Figure 14.30.



**Figure 14.28 Alternative entry point ConnectionPointReference notation**

A connection point reference to an exit point can also be visualized using a “bracketed space” symbol as shown in Figure 14.29. The text inside the symbol shall contain the keyword ‘via’ followed by the name of the connection point. This notation may only be used if the Transition associated with the connection point is defined using the graphical Transition notation such as the one shown in Figure 14.30.

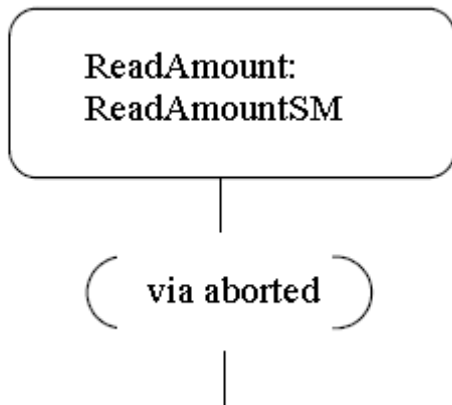


Figure 14.29 Alternative exit point `ConnectionPointReference` notation

### Transition

The default *textual* notation for a Transition is defined by the following BNF expression:

$$[<trigger> [',' <trigger>]* [ '[' <guard> ']' ] [ '/' <behavior-expression> ]]$$

Where *<trigger>* is the standard notation for Triggers (see sub clause 13.3.4), *<guard>* is a Boolean expression for a guard, and the optional *<behavior-expression>* is an expression specifying the effect Behavior written in some vendor-specific or standard textual surface language (see sub clause 16.1). The trigger may be any of the standard trigger types. SignalEvent triggers and CallEvent triggers are not distinguishable by syntax and must be discriminated by their declaration elsewhere.

As an alternative, in cases where the effect Behavior can be described as a control-flow based sequence of Actions, there is a graphical representation for Transitions and compound transitions which is similar to the notation used for Actions (see Clause 16).

This notation is in the form of a directed graph, which consists of one or more graphical symbols interconnected by directed arcs that represent control flow (see Figure 14.30). In all cases except for the Transition originating from the **initial** Pseudostate, the starting symbol, which has the form of the standard simple State notation, represents the source State of the Transition. If this Transition has a Signal-based Trigger, then the source state symbol is connected by an arc pointing to a special *Signal receipt* symbol described below. If there are multiple Triggers for the Transition, they are all listed in the same symbol as explained below.

If the Transition originates from the **initial** Pseudostate, the starting symbol is the *initial symbol*, which is the same as used for the **initial** Pseudostate: a filled black circle. In that case, there is no Signal receipt symbol immediately following the starting symbol.

Except for end symbols that terminate the paths, any of the following symbols can appear in the chain as appropriate

- an action symbol
- a choice point symbol
- a Signal send symbol
- a merge symbol

The terminating symbol in these directed graphs is always either a State-like symbol representing the target State of the transition or a final state symbol (which is the same as the symbol for a FinalState).

As this representation only applies for control-flow Behaviors, there is no equivalent to Data nodes.



### *Action symbols*

Each action symbol is represented by a rectangle with an optional textual specification of the action. It maps either to an `OpaqueAction` or to a `SequenceNode` containing one or more `Actions` executed in sequence (see sub clause [16.11.3](#)) and which are part of the `Activity` specifying the effect Behavior of the appropriate `Transition` in the compound transition.

### *Signal receipt symbol*

The Signal receipt symbol is shown as a five-pointed polygon that looks like a rectangle with a triangular notch in one of its sides (either one). It maps to the trigger of the `Transition` and does *not* map to an `Action` of the `Activity` that specifies the effect Behavior. The names of the `Signals` of the `Trigger` as well as any guard are contained within the symbol as follows:

$$\langle trigger \rangle [ ; ' \langle trigger \rangle ] * [ [ ' \langle guard \rangle ' ] ]$$

Where  $\langle trigger \rangle$  is specified as described in sub clause [13.3.4](#) with the restriction that only `Signal` and change `Event` types are allowed. The trigger symbol is always first in the path of symbols and a compound transition can only have at most one such symbol.

### *Signal send symbol*

This represents the special action of sending a signal and maps directly to a `SendSignalAction` that is part of the `Activity` that describes the effect Behavior of the corresponding `Transition`. The notation corresponds to the notation for the `SendSignalAction` (see Subclause [16.3.4](#)).

### *Choice point symbol*

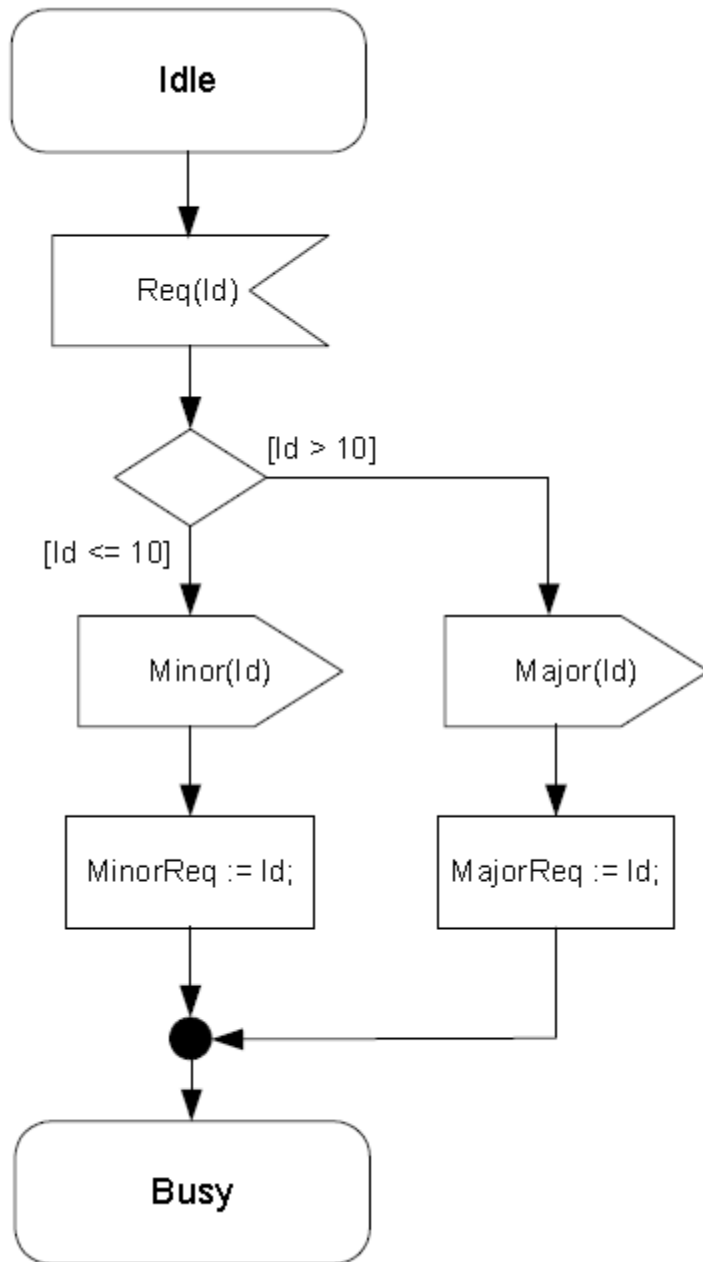
This symbol maps directly to a **choice** `Pseudostate` and uses the same notation.

**NOTE.** It is not part of any `Activity`.

### *Merge symbol*

A merge symbol is used to join multiple control-flow arcs and maps directly to a **junction** `Pseudostate` and uses the same notation. It is not part of any `Activity`.

Figure 14.30 shows a compound transition consisting of three connected `Transitions`: one from the `Idle State` to the choice symbol, one for each of the branches of the choice through the junction symbol, and one from the **junction** `Pseudostate` to the `Busy State`.



**Figure 14.30 Symbols for Signal reception, Sending, and Actions on a Transition**

*Deferred triggers*

A deferrable trigger is shown by listing it within the State followed by a slash and the special name “**defer**” (Figure 14.31).

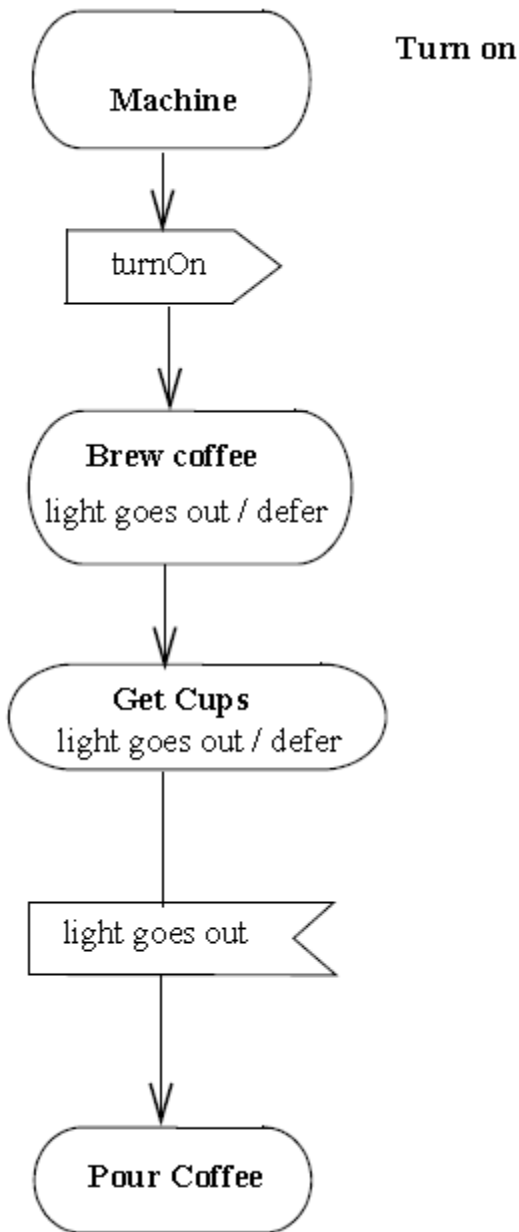


Figure 14.31 Deferred Trigger notation

### TransitionKind

- Transitions of kind **local** will be on the inside of the frame of the composite State, leaving the border of the composite State, or one of its entry points, and end at a Vertex inside the composite State. In the case of a **local** self-transition, the target may be the source State itself, or an exit point on the source State. Alternatively, a Transition of kind **local** can be shown as a Transition leaving a State symbol containing the text “\*.” The Transition is then considered to belong to the enclosing composite State. All of the Transitions in Figure 14.32 are **local**.
- Transitions of kind **external** can target any Vertex contained within or external to the source Vertex. The part of the **external** Transition closest to the source must be drawn outside of the source Vertex border. In the case of an **external** self

Transition where the source is a State or exit point on the State, it may target the State itself or an entry point on the State and it will be drawn completely outside of the State border. All of the Transitions in Figure 14.33 are **external**.

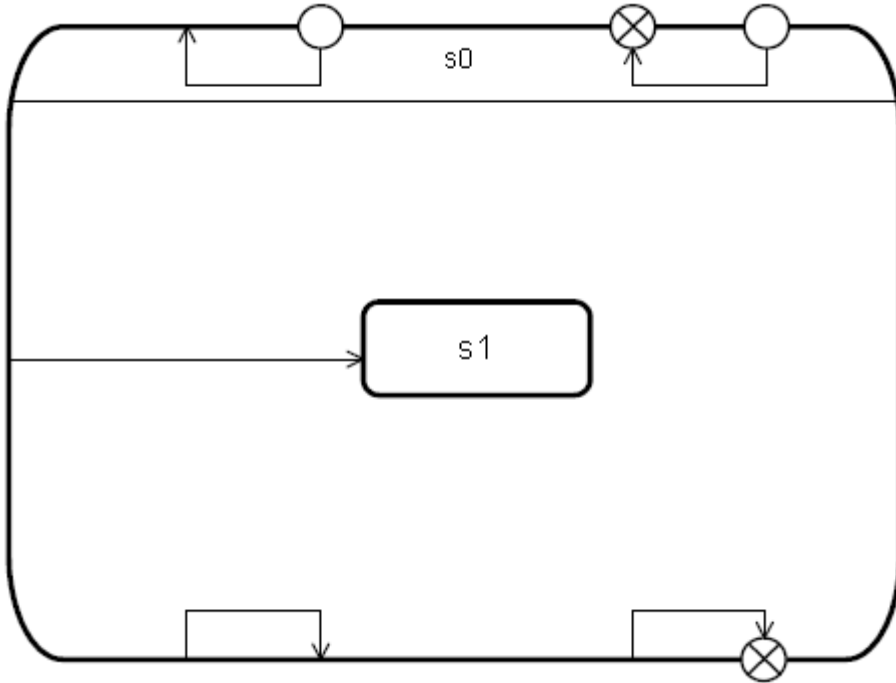


Figure 14.32 Local Transitions

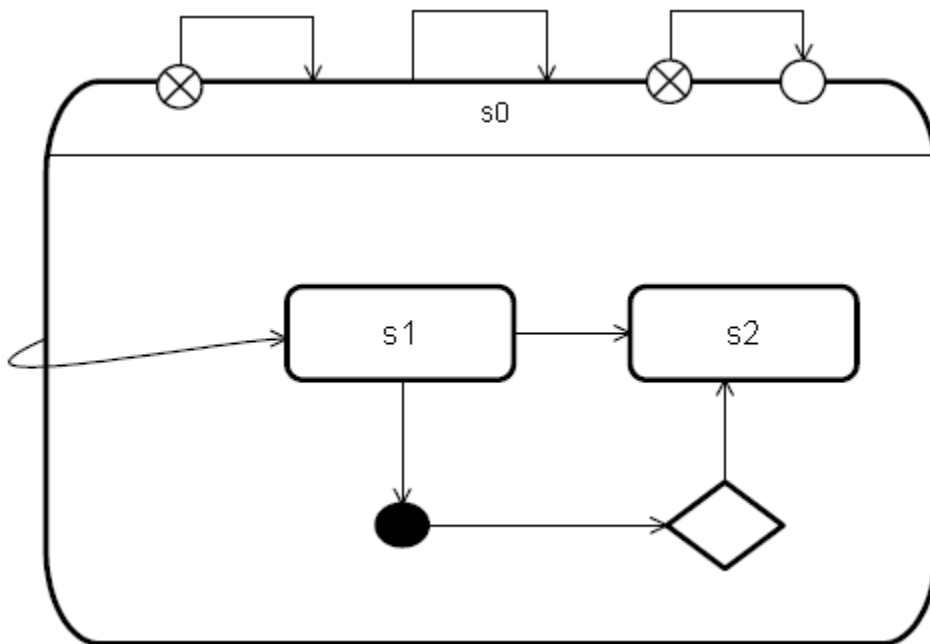


Figure 14.33 External Transitions



## 14.3.2 Abstract Syntax

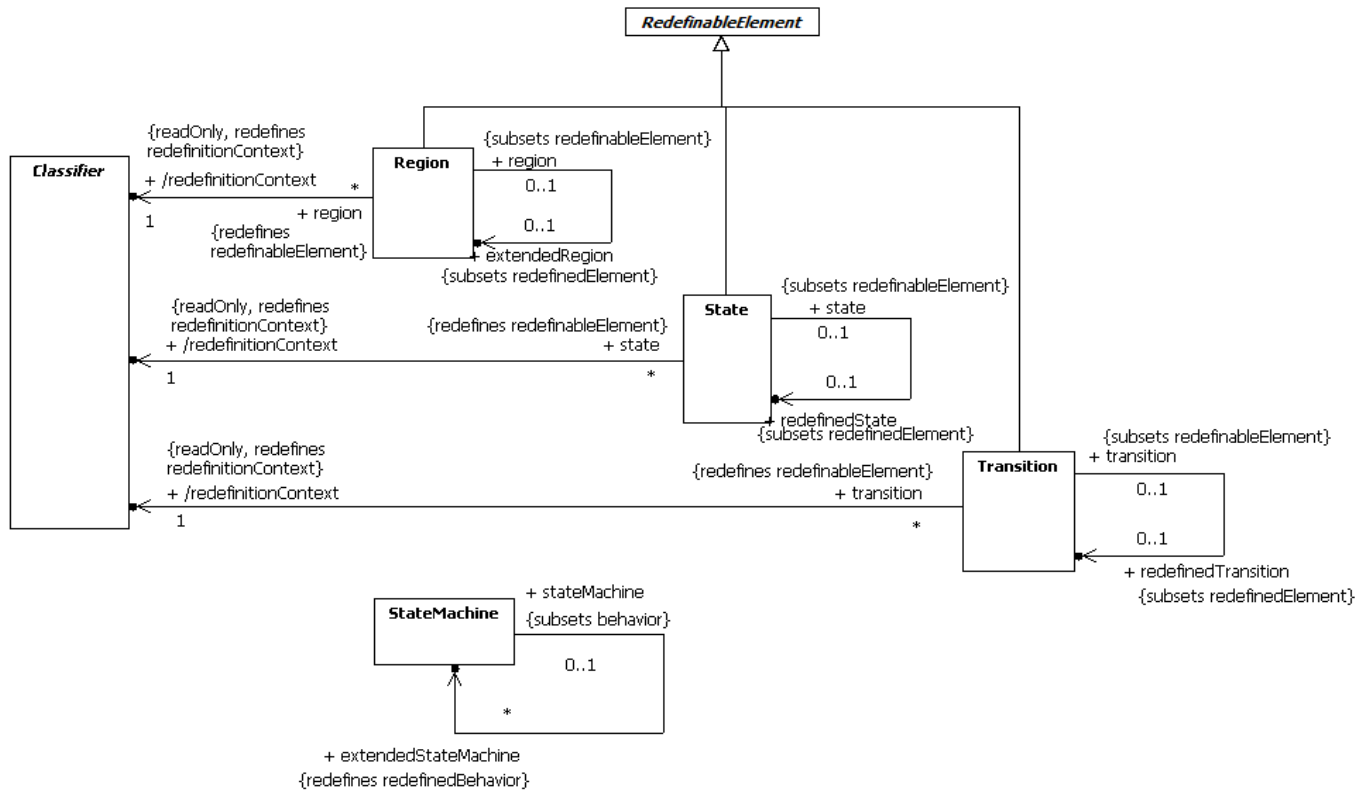


Figure 14.35 StateMachine redefinition

## 14.3.3 Semantics

### StateMachine Extension

A StateMachine is generalizable. A *specialized* StateMachine is an extension of the *general* StateMachine, such that

- new Regions, Vertices, and Transitions may be added;
- Regions and States may be redefined (e.g., simple States can be redefined as composite States while composite States can be redefined by adding States and Transitions); and
- Transitions can be redefined.

This can be done as part of Classifier specialization; that is, StateMachine behaviors and classifierBehaviors owned by a general Classifier can be specialized as can StateMachines that specify the methods of Behavioral features of a general Classifier.

A specialized StateMachine will have *all* the elements of the general StateMachine, and it may include additional elements. Regions may be added. Inherited Regions may be redefined by extension: States and Vertices are inherited, and States and Transitions of the Regions of the StateMachine may be redefined.

### *State redefinition*

A simple State may be redefined (extended) to become a composite State by one or more Regions. A composite State can be redefined (extended) by:

- adding new Regions,
- adding Vertices and Transitions to inherited Regions,
- adding entry/exit/doActivity Behaviors, if the general State does not have any,
- redefining States and Transitions.

The redefinition of a State applies to the whole StateMachine.

A submachine State may also be redefined. The submachine StateMachine may be replaced by another submachine StateMachine, provided that it has the same entry/exit points as the redefined submachine StateMachine. However, it may have additional entry/exit points.

In case of multiple general Classifiers, extension implies that the extension StateMachine gets orthogonal Regions for each of the StateMachines of the general Classifiers in addition to a distinct new Region.

### *Transition redefinition*

A Transition of an extended StateMachine may in the StateMachine extension be redefined. Transitions can have their effect and target State replaced, while the source State and trigger are preserved.

## **14.3.4 Notation**

A StateMachine that is an extension of the StateMachine in a general Classifier will have the keyword «extended» associated with the name of the StateMachine (e.g., see Figure 14.37 and Figure 14.38). Similarly, to indicate that an inherited Region is extended or that a State is extended, the keyword «extended» is added to the name of the element. Inherited elements in a StateMachine, Region, or State are drawn either with dashed lines or light-toned lines (e.g., Figure 14.37).

### **Examples**

As an example of StateMachine specialization, the States “VerifyCard,” “OutOfService,” and “VerifyTransaction” in the ATM StateMachine in Figure 14.36 have been designated as **{final}**, which means that they cannot be redefined in specializations of ATM. All other States can be redefined. The (verifyTransaction-releaseCard) Transition has also been specified as **{final}**, meaning that its effect Behavior and the target State cannot be redefined.

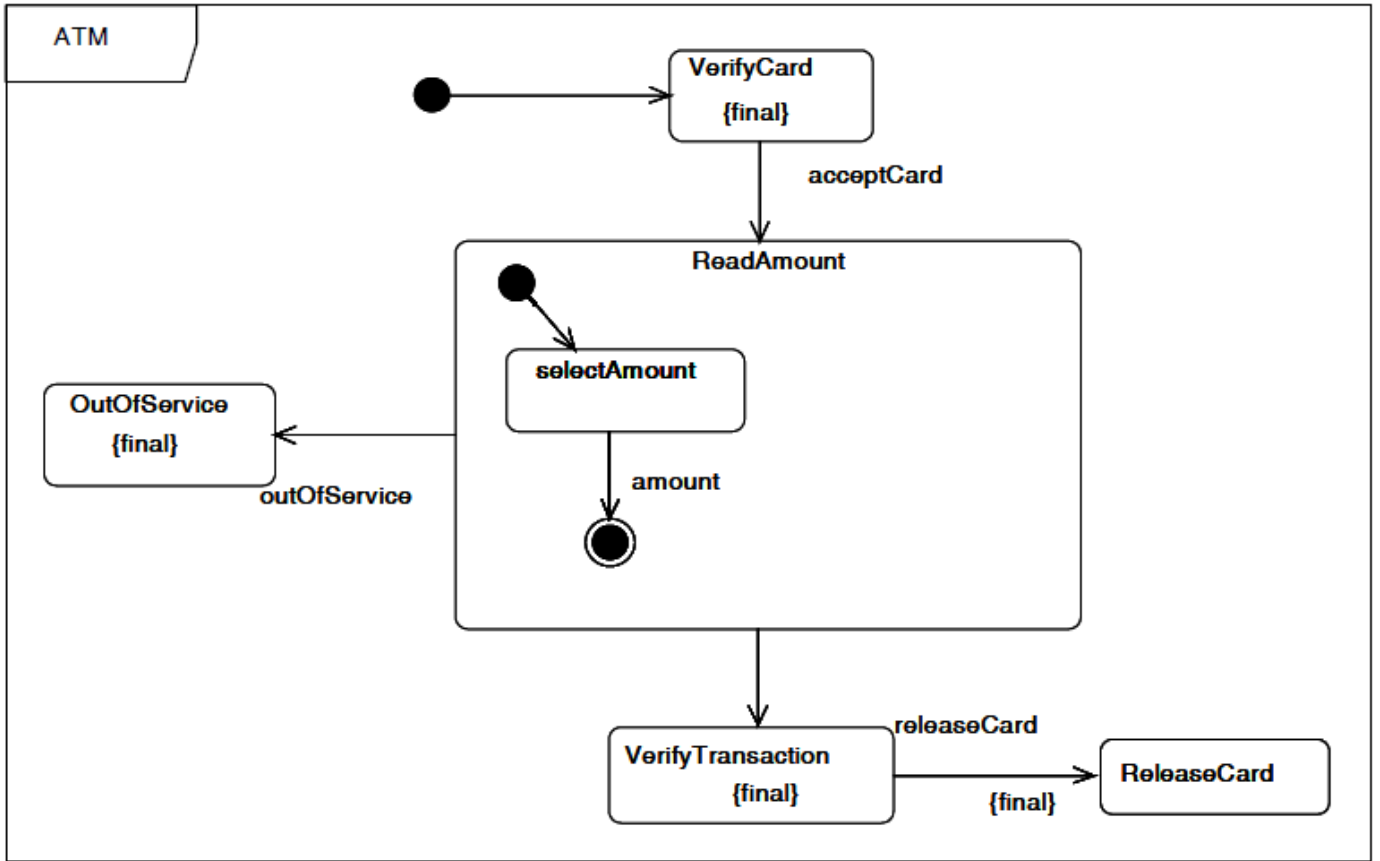
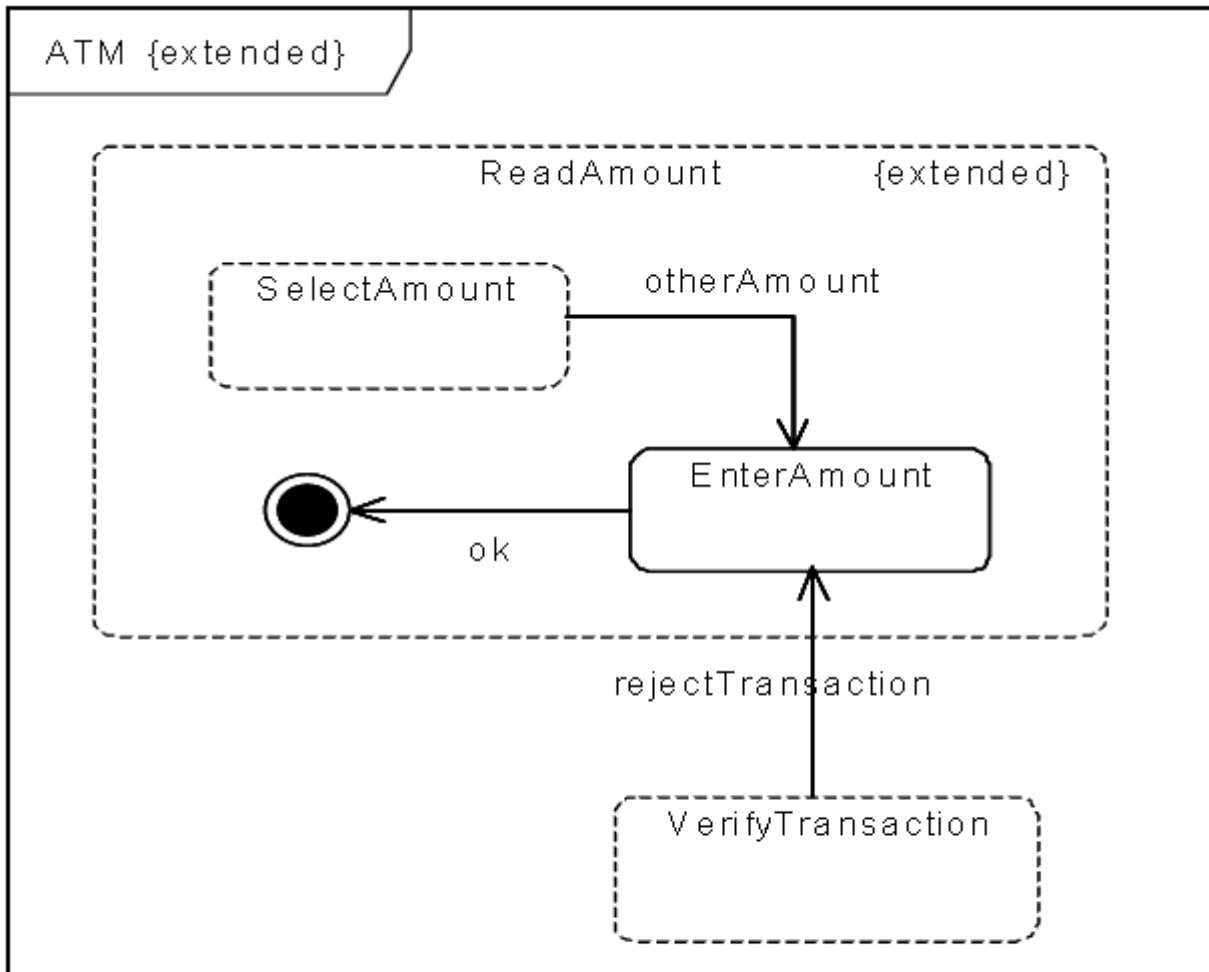


Figure 14.36 A general StateMachine



In Figure 14.37 a specialized ATM (which is the StateMachine of a Class that is a specialization of the Class with the ATM StateMachine of Figure 14.36) is defined by extending the composite State by adding a State and a Transition, so that users can enter the desired amount. In addition, a Transition is added from an inherited State to the newly introduced State.



**Figure 14.37 An extended StateMachine**

Figure 14.38 shows an example of adding Transitions to a specialized StateMachine.

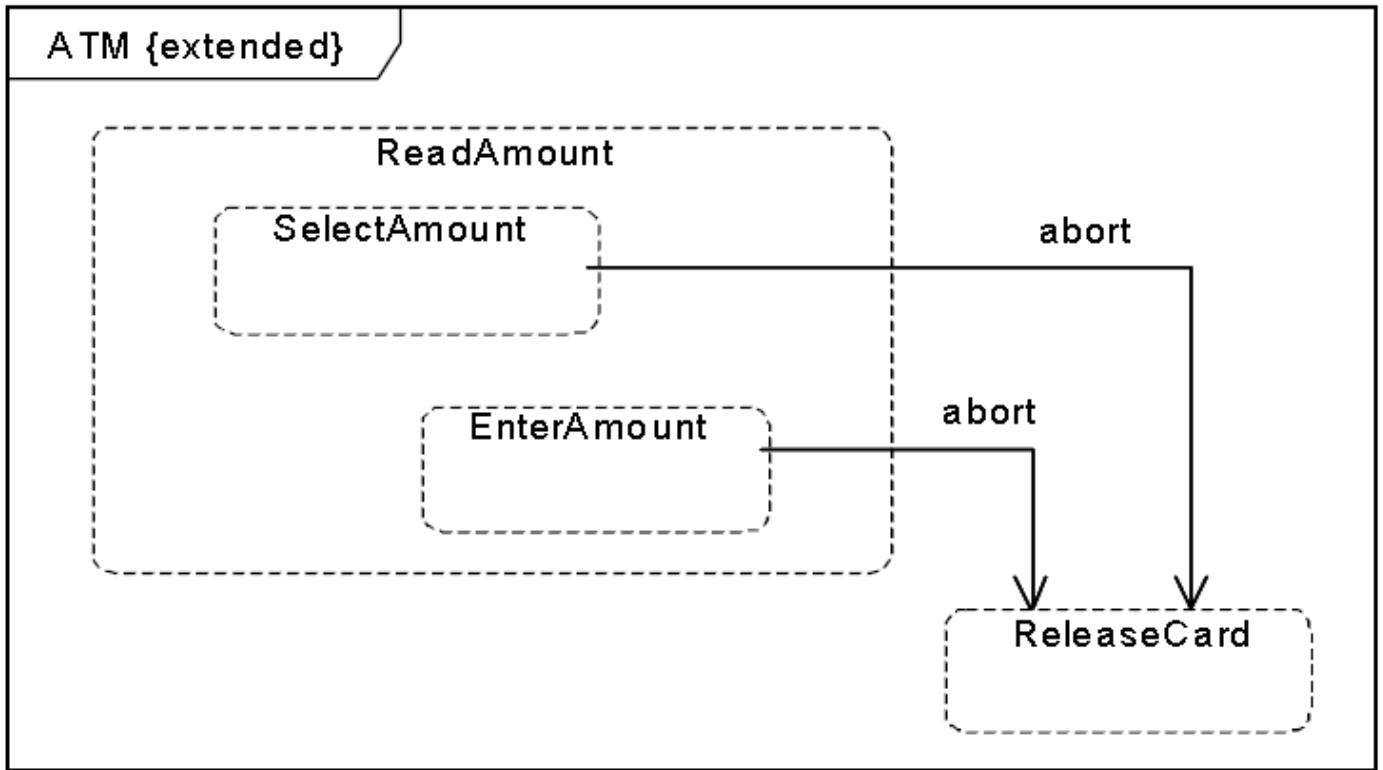


Figure 14.38 Adding Transitions

## 14.4 ProtocolStateMachines

### 14.4.1 Summary

ProtocolStateMachines are used to express usage protocols. ProtocolStateMachines express the legal sequences of Event occurrences to which the Behaviors of an associated BehavedClassifier must conform. The StateMachine notation is a convenient way to define the order of invocations of the behavioral features of a Classifier. ProtocolStateMachines can be associated with Classifiers, Interfaces, and Ports.

## 14.4.2 Abstract Syntax

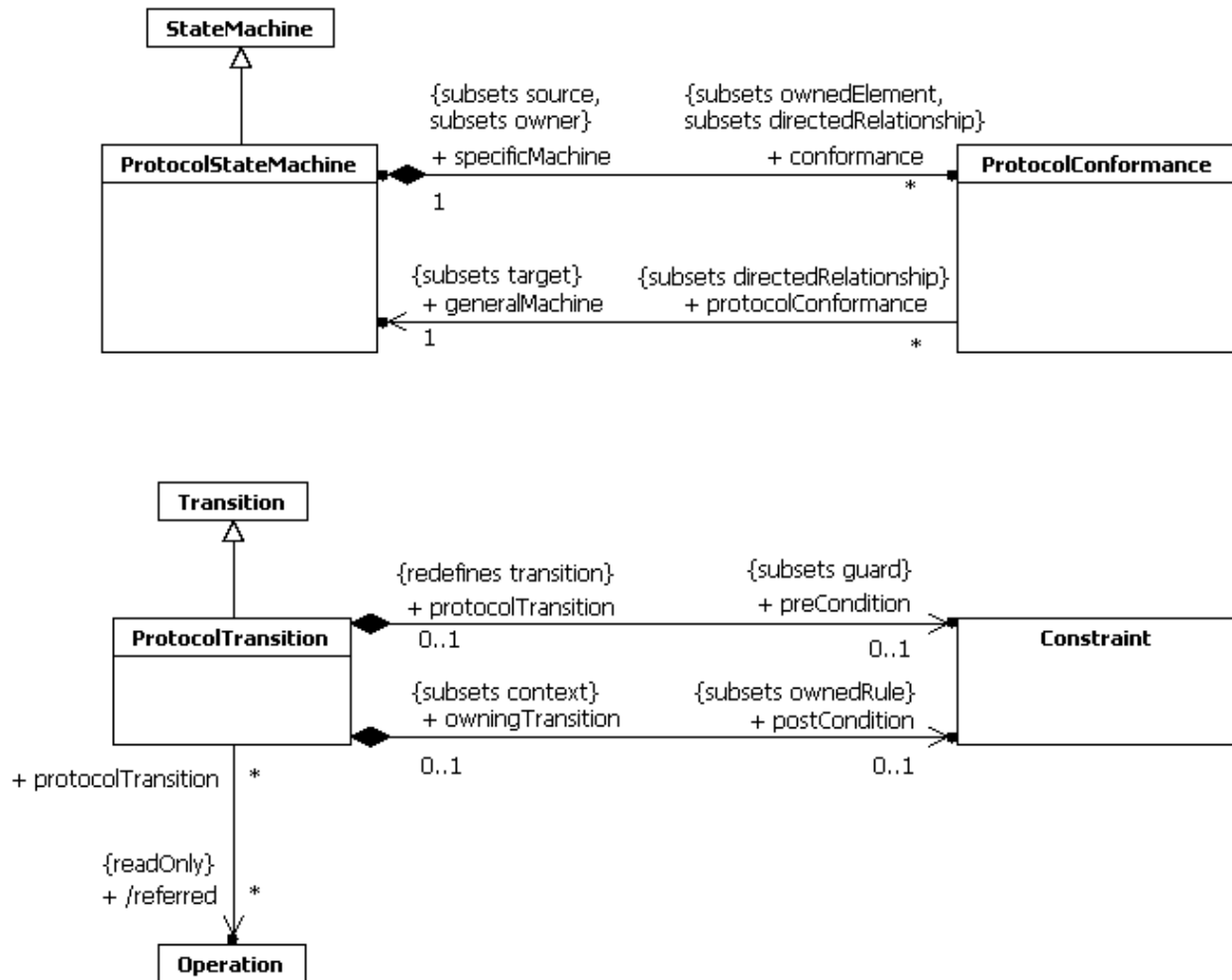


Figure 14.39 ProtocolStateMachines

## 14.4.3 Semantics

### ProtocolStateMachine

A ProtocolStateMachine is always defined in the context of a Classifier. It specifies which BehavioralFeatures of that Classifier can be invoked in a given protocol state and under what conditions, thereby specifying allowed invocation sequences. In this manner, a specification of the lifecycle of an instance of the Classifier is defined from an external perspective.

ProtocolStateMachines help define the order in which BehavioralFeatures of a Classifier are invoked by specifying:

- the behavioral context (i.e., which states and pre-conditions) in which they can be validly invoked,
- the valid orderings of invocations,
- the expected outcomes (post-conditions) of invocations.

ProtocolStateMachine present an external view of the owning Classifier as perceived by its collaborators. This extends beyond what can be captured via pre- and post-conditions on individual BehavioralFeatures, as ProtocolStateMachines also specify the valid orderings of invocations of the different features. This is achieved by a state machine specification in which the transition triggers are feature invocations and the guards of the transitions (ProtocolTransitions) specify the pre-condition that must apply for the invocation to be valid. The states (ProtocolStates) of this state machine, being a consequence of past invocation sequences, capture the state of the protocol and are also a form of pre-condition.

**NOTE.** Because ProtocolStateMachines provide a “black box” view of the behavior of a Classifier, their States may not necessarily correspond to the States of internal behavioral StateMachines.

ProtocolStateMachine interpretation can vary from:

1. *Declarative* ProtocolStateMachines, which specify the legal Transitions for BehavioralFeature invocations. The effects of a BehavioralFeature invocation is not specified. This type of specification only provides a contract for the user of the context Classifier.
2. *Executable* (run time) ProtocolStateMachines, which specify all Event occurrences that an object may receive and handle, together with the Transitions that these trigger. In this case, the legal Transitions for BehavioralFeature invocations must match exactly the triggered Transitions or a run-time exception occurs. The invocation results in the execution of the method associated with the invoked BehavioralFeatures.

The specifications for both interpretations is the same, the only difference being the direct dynamic implication that the latter interpretation provides.

The more sophisticated forms of modeling encountered in behavioral StateMachines such as compound Transitions, submachine StateMachines, composite States, and concurrent orthogonal Regions, can also be used for ProtocolStateMachines. For example, concurrent Regions make it possible to express protocols where an instance can have several active States simultaneously. Submachine StateMachines and compound transitions can be used for factorizing complex ProtocolStateMachines.

A Classifier may have several ProtocolStateMachines. This can be used, for example, when a Classifier has multiple parents, each having its own ProtocolStateMachine, and the protocols are orthogonal. An alternative to this is to simply have one ProtocolStateMachine, with distinct StateMachines in concurrent Regions.

#### *State in ProtocolStateMachines*

The States of ProtocolStateMachines are exposed to the users of their context Classifiers. A protocol State represents an exposed stable situation of its context Classifier: When an instance of the Classifier classifier is not processing any BehavioralFeature invocation, users of this instance can always know its state configuration.

The States of a ProtocolStateMachine cannot have defined entry, exit, or doActivity Behaviors.

#### **ProtocolTransition**

A ProtocolTransition specifies a legal Transition for an invocation of a BehavioralFeature of the context Classifier. ProtocolTransitions have the following features:

- a pre-condition (preCondition), which specializes the guard attribute of Transition,
- a trigger,
- a post-condition (postCondition).

The protocol Transition specifies that (a) the associated (referred) feature can be invoked on an instance of the context Classifier, if it is in the origin State and the guard condition holds, and that (b) upon completion of the Transition, the instance will be in the target State in which the post-condition will hold.

ProtocolTransitions do not have an associated effect Behavior. The consequence of a ProtocolTransition executed as a result of a BehavioralFeature invocation is implicit: it is the execution of the method corresponding to the invoked BehavioralFeature. In case of other types of Triggers, the consequences are unspecified except that a Transition will lead to another State under a specific post-condition, regardless of any Behaviors associated with this Transition.

*Unexpected trigger reception*

The interpretation of the reception of an Event occurrence that does not match a valid trigger for the current State, state invariant, or pre-condition is not defined (e.g., it can be ignored, rejected, or deferred; an exception can be raised; or the application can stop on an error). It corresponds semantically to a pre-condition violation, for which no predefined Behavior is defined in UML.

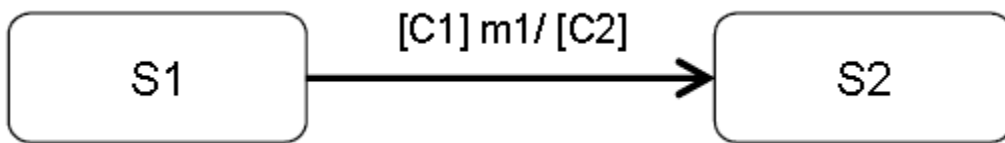
*Unexpected Behavior*

The interpretation of an unexpected Behavior, that is an unexpected result of a Transition (wrong FinalState or FinalState invariant, or post-condition) is also not defined. However, this should be interpreted as an error of the implementation of the ProtocolStateMachine.

*Equivalences to pre- and post-conditions of operations*

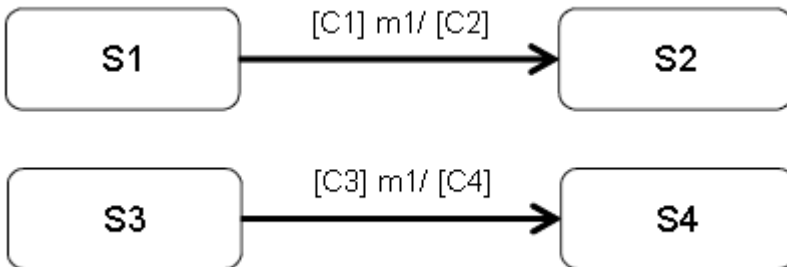
A protocol Transition can be semantically interpreted in terms of pre- and post-conditions on the associated operation. For example, the Transition in Figure 14.40 can be interpreted in the following way:

1. The operation “m1” can be called on an instance when it is in the ProtocolState “S1” under the condition “C1.”
2. When “m1” is called in the ProtocolState “S1” under the condition “C1,” then the ProtocolState “S2” must be reached under the condition “C2.”



**Figure 14.40** An example of a ProtocolTransition associated with the operation "m1"

Operations referred by several Transitions



**Figure 14.41** Example of several ProtocolTransitions associated with the same operation (m1)

In a ProtocolStateMachine, several Transitions can refer to the same operation as illustrated in Figure 14.41. In that case, all pre- and post-conditions will be combined in the operation pre-condition as shown below.

```

Operation m1()
Pre:   in state S1 and condition C1
      or
      in state S3 and condition C3
Post:  if the initial condition was "in state S1 and condition C1"
    
```

```
        then in S2 and C2
else
if the initial condition was "in state S3 and condition C3"
    then in S4 and C4
```

A `ProtocolStateMachine` specifies all the legal `ProtocolTransition` for each `BehavioralFeature` referred by its `Transitions`.

#### *Unreferred Operations*

If a `BehavioralFeature` is not referred by any `ProtocolTransition`, then the operation can be called for any `State` of the `ProtocolStateMachine`, and will not change the current `State` or pre- and post-conditions.

#### *Using other types of Events in ProtocolStateMachines*

Apart from invocations of `BehavioralFeatures`, other `Events` may be used for expressing the behavior of `ProtocolStateMachines`. A `Trigger` that is not a `BehavioralFeature` invocation can be specified for a protocol `Transition`. In that case, this specification is a requirement for the environment external to the `ProtocolStateMachine`. That is, it is legal to send an `Event` occurrence of this type to an instance of the context `Classifier` only under the conditions specified by the `ProtocolStateMachine`. The precise semantic interpretation of this is not defined.

### **ProtocolConformance**

`ProtocolStateMachines` can be refined into more specific `ProtocolStateMachines`. Protocol conformance declares that the specific `ProtocolStateMachine` specifies a protocol that conforms to that specified by the general `ProtocolStateMachine`.

A `ProtocolStateMachine` is owned by a `Classifier`. The `Classifiers` owning a general `StateMachine` and an associated specific `StateMachine` are generally also connected by a `Generalization` or a `Realization`.

Protocol conformance represents a declaration that every rule and constraint specified for the general `ProtocolStateMachine` (state invariants, pre- and post-conditions for the operations referred by the `ProtocolStateMachine`) apply to the specific `ProtocolStateMachine`.

## **14.4.4 Notation**

### **ProtocolStateMachine**

The notation for `ProtocolStateMachine` is very similar to the one for behavioral `StateMachines`. The keyword `{protocol}` placed close to the name of the `StateMachine` differentiates graphically `ProtocolStateMachine` diagrams.

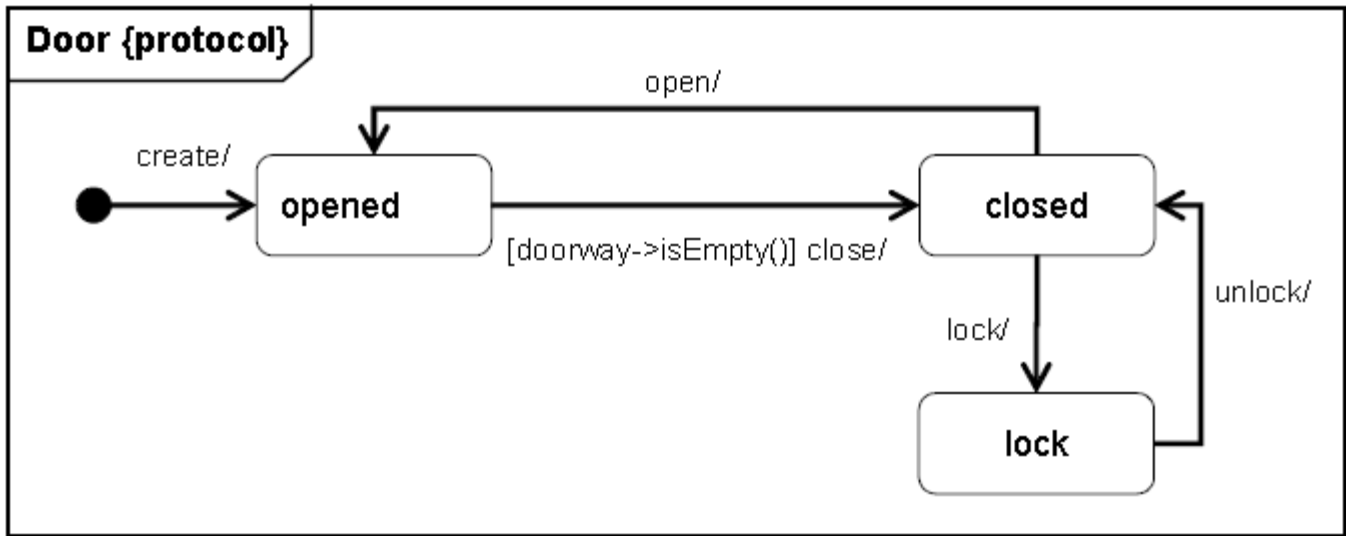


Figure 14.42 ProtocolStateMachine example

The textual expression of an invariant associated with a State in a ProtocolStateMachine is represented by placing it after or under the name of the State, enclosed in square brackets (Figure 14.43).

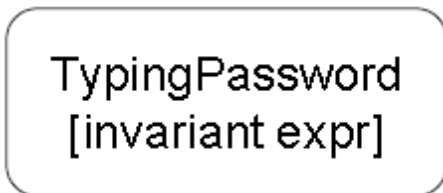


Figure 14.43 Notation for a State with an invariant

### ProtocolTransition

The usual StateMachine notation applies. The difference is that no effect Behaviors are specified for ProtocolTransitions, and that post-conditions can exist. Post-conditions have the same syntax as guard conditions, but appear at the end of the Transition syntax.

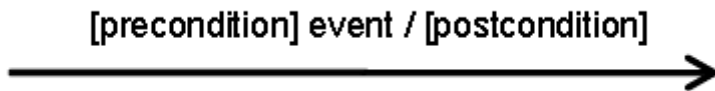


Figure 14.44 ProtocolTransition notation

## 14.5 Classifier Descriptions

### ConnectionPointReference [Class]

#### Description

A ConnectionPointReference represents a usage (as part of a submachine State) of an entry/exit point Pseudostate defined in the StateMachine referenced by the submachine State.

#### Diagrams

[Behavior State Machines](#)

#### Generalizations

[Vertex](#)

#### Association Ends

- entry : [Pseudostate](#) [0..\*] (opposite [A\\_entry\\_connectionPointReference::connectionPointReference](#))  
The entryPoint Pseudostates corresponding to this connection point.
- exit : [Pseudostate](#) [0..\*] (opposite [A\\_exit\\_connectionPointReference::connectionPointReference](#))  
The exitPoints kind Pseudostates corresponding to this connection point.
- state : [State](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [State::connection](#))  
The State in which the ConnectionPointReference is defined.

#### Constraints

- exit\_pseudostates  
The exit Pseudostates must be Pseudostates with kind exitPoint.  

```
inv: exit->forall(kind = PseudostateKind::exitPoint)
```
- entry\_pseudostates  
The entry Pseudostates must be Pseudostates with kind entryPoint.  

```
inv: entry->forall(kind = PseudostateKind::entryPoint)
```

### FinalState [Class]

#### Description

A special kind of State, which, when entered, signifies that the enclosing Region has completed. If the enclosing Region is directly contained in a StateMachine and all other Regions in that StateMachine also are completed, then it means that the entire StateMachine behavior is completed.



## Diagrams

[Behavior State Machines](#)

## Generalizations

[State](#)

## Constraints

- **no\_exit\_behavior**  
A FinalState has no exit Behavior.  
  
`inv: exit->isEmpty()`
- **no\_outgoing\_transitions**  
A FinalState cannot have any outgoing Transitions.  
  
`inv: outgoing->size() = 0`
- **no\_regions**  
A FinalState cannot have Regions.  
  
`inv: region->size() = 0`
- **cannot\_reference\_submachine**  
A FinalState cannot reference a submachine.  
  
`inv: submachine->isEmpty()`
- **no\_entry\_behavior**  
A FinalState has no entry Behavior.  
  
`inv: entry->isEmpty()`
- **no\_state\_behavior**  
A FinalState has no state (doActivity) Behavior.  
  
`inv: doActivity->isEmpty()`

## ProtocolConformance [Class]

### Description

A ProtocolStateMachine can be redefined into a more specific ProtocolStateMachine or into behavioral StateMachine. ProtocolConformance declares that the specific ProtocolStateMachine specifies a protocol that conforms to the general ProtocolStateMachine or that the specific behavioral StateMachine abides by the protocol of the general ProtocolStateMachine.

### Diagrams

[Protocol State Machines](#)

## Generalizations

[DirectedRelationship](#)

## Association Ends

- generalMachine : [ProtocolStateMachine](#) [1..1]{subsets [DirectedRelationship::target](#)} (opposite [A\\_generalMachine\\_protocolConformance::protocolConformance](#))  
Specifies the ProtocolStateMachine to which the specific ProtocolStateMachine conforms.
- specificMachine : [ProtocolStateMachine](#) [1..1]{subsets [DirectedRelationship::source](#), subsets [Element::owner](#)} (opposite [ProtocolStateMachine::conformance](#))  
Specifies the ProtocolStateMachine which conforms to the general ProtocolStateMachine.

## ProtocolStateMachine [Class]

### Description

A ProtocolStateMachine is always defined in the context of a Classifier. It specifies which BehavioralFeatures of the Classifier can be called in which State and under which conditions, thus specifying the allowed invocation sequences on the Classifier's BehavioralFeatures. A ProtocolStateMachine specifies the possible and permitted Transitions on the instances of its context Classifier, together with the BehavioralFeatures that carry the Transitions. In this manner, an instance lifecycle can be specified for a Classifier, by defining the order in which the BehavioralFeatures can be activated and the States through which an instance progresses during its existence.

### Diagrams

[Protocol State Machines](#), [Encapsulated Classifiers](#), [Interfaces](#)

## Generalizations

[StateMachine](#)

## Association Ends

- ♦ conformance : [ProtocolConformance](#) [0..\*]{subsets [Element::ownedElement](#), subsets [A\\_source\\_directedRelationship::directedRelationship](#)} (opposite [ProtocolConformance::specificMachine](#))  
Conformance between ProtocolStateMachine

## Constraints

- classifier\_context  
A ProtocolStateMachine must only have a Classifier context, not a BehavioralFeature context.

```
inv: _('context' <> null and specification = null
```

- deep\_or\_shallow\_history  
ProtocolStateMachines cannot have deep or shallow history Pseudostates.

```
inv: region->forall (r | r.subvertex->forall (v | v.ocIsKindOf(Pseudostate) implies
```

```
((v.oclasType(Pseudostate).kind <> PseudostateKind::deepHistory) and (v.oclasType(Pseudostate).kind <> PseudostateKind::shallowHistory)))
```

- **entry\_exit\_do**  
The states of a ProtocolStateMachine cannot have entry, exit, or do activity Behaviors.

```
inv: region->forall(r | r.subvertex->forall(v | v.oclisKindOf(State) implies (v.oclasType(State).entry->isEmpty() and v.oclasType(State).exit->isEmpty() and v.oclasType(State).doActivity->isEmpty()))))
```

- **protocol\_transitions**  
All Transitions of a ProtocolStateMachine must be ProtocolTransitions.

```
inv: region->forall(r | r.transition->forall(t | t.oclisTypeOf(ProtocolTransition)))
```

## ProtocolTransition [Class]

### Description

A ProtocolTransition specifies a legal Transition for an Operation. Transitions of ProtocolStateMachines have the following information: a pre-condition (guard), a Trigger, and a post-condition. Every ProtocolTransition is associated with at most one BehavioralFeature belonging to the context Classifier of the ProtocolStateMachine.

### Diagrams

[Protocol State Machines](#)

### Generalizations

[Transition](#)

### Association Ends

- ♦ postCondition : [Constraint](#) [0..1]{subsets [Namespace::ownedRule](#)} (opposite [A\\_postCondition\\_owningTransition::owningTransition](#))  
Specifies the post condition of the Transition which is the Condition that should be obtained once the Transition is triggered. This post condition is part of the post condition of the Operation connected to the Transition.
- ♦ preCondition : [Constraint](#) [0..1]{subsets [Transition::guard](#)} (opposite [A\\_preCondition\\_protocolTransition::protocolTransition](#))  
Specifies the precondition of the Transition. It specifies the Condition that should be verified before triggering the Transition. This guard condition added to the source State will be evaluated as part of the precondition of the Operation referred by the Transition if any.
- /referred : [Operation](#) [0..\*]{ } (opposite [A\\_referred\\_protocolTransition::protocolTransition](#))  
This association refers to the associated Operation. It is derived from the Operation of the CallEvent Trigger when applicable.

## Operations

- referred() : [Operation](#) [0..\*]  
Derivation for ProtocolTransition::/referred

```
body: trigger->collect(event)->select(oclIsKindOf(CallEvent))-  
>collect(oclAsType(CallEvent).operation)->asSet()
```

## Constraints

- refers\_to\_operation  
If a ProtocolTransition refers to an Operation (i.e., has a CallEvent trigger corresponding to an Operation), then that Operation should apply to the context Classifier of the StateMachine of the ProtocolTransition.

```
inv: if (referred()->notEmpty() and containingStateMachine()._context->notEmpty()) then  
    containingStateMachine()._context.oclAsType(BehavioredClassifier).allFeatures()-  
>includesAll(referred())  
else true endif
```

- associated\_actions  
A ProtocolTransition never has associated Behaviors.

```
inv: effect = null
```

- belongs\_to\_psm  
A ProtocolTransition always belongs to a ProtocolStateMachine.

```
inv: container.belongsToPSM()
```

## Pseudostate [Class]

### Description

A Pseudostate is an abstraction that encompasses different types of transient Vertices in the StateMachine graph. A StateMachine instance never comes to rest in a Pseudostate, instead, it will exit and enter the Pseudostate within a single run-to-completion step.

### Diagrams

[Behavior State Machines](#)

### Generalizations

[Vertex](#)

### Attributes

- kind : [PseudostateKind](#) [1..1] = initial  
Determines the precise type of the Pseudostate and can be one of: entryPoint, exitPoint, initial, deepHistory, shallowHistory, join, fork, junction, terminate or choice.

## Association Ends

- state : [State](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [State::connectionPoint](#))  
The State that owns this Pseudostate and in which it appears.
- stateMachine : [StateMachine](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [StateMachine::connectionPoint](#))  
The StateMachine in which this Pseudostate is defined. This only applies to Pseudostates of the kind `entryPoint` or `exitPoint`.

## Constraints

- `transitions_outgoing`

All transitions outgoing a fork vertex must target states in different regions of an orthogonal state.

```
inv: (kind = PseudostateKind::fork) implies
-- for any pair of outgoing transitions there exists an orthogonal state which contains the targets
of these transitions
-- such that these targets belong to different regions of that orthogonal state

outgoing->forall(t1:Transition, t2:Transition | let contState:State =
containingStateMachine().LCASState(t1.target, t2.target) in
((contState <> null) and (contState.region
->exists(r1:Region, r2: Region | (r1 <> r2) and t1.target.isContainedInRegion(r1) and
t2.target.isContainedInRegion(r2))))))
```

- `choice_vertex`

In a complete statemachine, a choice Vertex must have at least one incoming and one outgoing Transition.

```
inv: (kind = PseudostateKind::choice) implies (incoming->size() >= 1 and outgoing->size() >= 1)
```

- `outgoing_from_initial`

The outgoing Transition from an initial vertex may have a behavior, but not a trigger or a guard.

```
inv: (kind = PseudostateKind::initial) implies (outgoing.guard = null and outgoing.trigger-
>isEmpty())
```

- `join_vertex`

In a complete StateMachine, a join Vertex must have at least two incoming Transitions and exactly one outgoing Transition.

```
inv: (kind = PseudostateKind::join) implies (outgoing->size() = 1 and incoming->size() >= 2)
```

- `junction_vertex`

In a complete StateMachine, a junction Vertex must have at least one incoming and one outgoing Transition.

```
inv: (kind = PseudostateKind::junction) implies (incoming->size() >= 1 and outgoing->size() >= 1)
```

- `history_vertices`

History Vertices can have at most one outgoing Transition.

```
inv: ((kind = PseudostateKind::deepHistory) or (kind = PseudostateKind::shallowHistory)) implies
(outgoing->size() <= 1)
```

- **initial\_vertex**  
An initial Vertex can have at most one outgoing Transition.

```
inv: (kind = PseudostateKind::initial) implies (outgoing->size() <= 1)
```

- **fork\_vertex**  
In a complete StateMachine, a fork Vertex must have at least two outgoing Transitions and exactly one incoming Transition.

```
inv: (kind = PseudostateKind::fork) implies (incoming->size() = 1 and outgoing->size() >= 2)
```

- **transitions\_incoming**  
All Transitions incoming a join Vertex must originate in different Regions of an orthogonal State.

```
inv: (kind = PseudostateKind::join) implies
```

```
-- for any pair of incoming transitions there exists an orthogonal state which contains the source
vertices of these transitions
-- such that these source vertices belong to different regions of that orthogonal state
```

```
incoming->forall(t1:Transition, t2:Transition | let contState:State =
containingStateMachine().LCASState(t1.source, t2.source) in
((contState <> null) and (contState.region
->exists(r1:Region, r2: Region | (r1 <> r2) and t1.source.isContainedInRegion(r1) and
t2.source.isContainedInRegion(r2))))))
```

## PseudostateKind [Enumeration]

### Description

PseudostateKind is an Enumeration type that is used to differentiate various kinds of Pseudostates.

### Diagrams

- [Behavior State Machines](#)

### Literals

- **initial**  
An initial Pseudostate represents a starting point for a Region; that is, it is the point from which execution of its contained behavior commences when the Region is entered via default activation. It is the source for at most one Transition, which may have an associated effect Behavior, but not an associated trigger or guard. There can be at most one initial Vertex in a Region.
- **deepHistory**  
This type of Pseudostate is a kind of variable that represents the most recent active state configuration of its owning State. As explained above, a Transition terminating on this Pseudostate implies restoring the State to that same state configuration, but with all the semantics of entering a State (see the clause describing State entry). The entry Behaviors of all States in the restored state configuration are performed in the appropriate order starting with the outermost State. A deepHistory Pseudostates can only be defined for composite States and, at most one such Pseudostate can be contained

in a composite State.

- **shallowHistory**  
As explained for the deepHistory Pseudostate, this type of Pseudostate is a kind of variable that represents the most recent active substate of its containing State, but not the substates of that substate. A Transition terminating on this Pseudostate implies restoring the composite State to that substate with all the semantics of entering a State. A single outgoing Transition from this Pseudostate may be defined terminating on a substate of the composite State. This substate is the default shallow history state of the composite State. A shallowHistory Pseudostates can only be defined for composite States and, at most one such Pseudostate can be included in a composite State.
- **join**  
This type of Pseudostate serves as a common target Vertex for two or more Transitions originating from Vertices in different orthogonal Regions. Transitions terminating on a join Pseudostate cannot have a guard or a trigger. Similar to junction points in Petri nets, join Pseudostates perform a synchronization function, whereby all incoming Transitions have to complete before execution can continue through an outgoing Transition.
- **fork**  
fork Pseudostates serve to split an incoming Transition into two or more Transitions terminating on Vertices in orthogonal Regions of a composite State. The Transitions outgoing from a fork Pseudostate cannot have a guard or a trigger.
- **junction**  
This type of Pseudostate is used to connect multiple Transitions into compound paths between States. For example, a junction Pseudostate can be used to merge multiple incoming Transitions into a single outgoing Transition representing a shared continuation path. Or, it can be used to split an incoming Transition into multiple outgoing Transition segments with different guard Constraints. Note that such guard Constraints are evaluated before any compound transition containing this Pseudostate is executed, which is why this is referred to as a static conditional branch. It may happen that, for a particular compound transition, the configuration of Transition paths and guard values is such that the compound transition is prevented from reaching a valid state configuration. In those cases, the entire compound transition is disabled even though its Triggers are enabled. (As a way of avoiding this situation in some cases, it is possible to associate a predefined guard denoted as “else” with at most one outgoing Transition. This Transition is enabled if all the guards attached to the other Transitions evaluate to false). If more than one guard evaluates to true, one of these is chosen. The algorithm for making this selection is not defined.
- **choice**  
This type of Pseudostate is similar to a junction Pseudostate (see above) and serves similar purposes, with the difference that the guard Constraints on all outgoing Transitions are evaluated dynamically, when the compound transition traversal reaches this Pseudostate. Consequently, choice is used to realize a dynamic conditional branch. It allows splitting of compound transitions into multiple alternative paths such that the decision on which path to take may depend on the results of Behavior executions performed in the same compound transition prior to reaching the choice point. If more than one guard evaluates to true, one of the corresponding Transitions is selected. The algorithm for making this selection is not defined. If none of the guards evaluates to true, then the model is considered ill formed. (To avoid this, it is recommended to define one outgoing Transition with the predefined “else” guard for every choice Pseudostate.)
- **entryPoint**  
An entryPoint Pseudostate represents an entry point for a StateMachine or a composite State that provides encapsulation of the insides of the State or StateMachine. In each Region of the StateMachine or composite State owning the entryPoint, there is at most a single Transition from the entry point to a Vertex within that Region. Note that, if the owning State has an associated entry Behavior, this Behavior is executed before any behavior associated with the outgoing Transition. If multiple Regions are involved, the entry point acts as a fork Pseudostate.

- **exitPoint**  
An exitPoint Pseudostate is an exit point of a StateMachine or composite State that provides encapsulation of the insides of the State or StateMachine. Transitions terminating on an exit point within any Region of the composite State or a StateMachine referenced by a submachine State implies exiting of this composite State or submachine State (with execution of its associated exit Behavior). If multiple Transitions from orthogonal Regions within the State terminate on this Pseudostate, then it acts like a join Psuedostate. composite state.
- **terminate**  
Entering a terminate Pseudostate implies that the execution of the StateMachine is terminated immediately. The StateMachine does not exit any States nor does it perform any exit Behaviors. Any executing doActivity Behaviors are automatically aborted. Entering a terminate Pseudostate is equivalent to invoking a DestroyObjectAction.

## Region [Class]

### Description

A Region is a top-level part of a StateMachine or a composite State, that serves as a container for the Vertices and Transitions of the StateMachine. A StateMachine or composite State may contain multiple Regions representing behaviors that may occur in parallel.

### Diagrams

[Behavior State Machines](#), [State Machine Redefinition](#)

### Generalizations

[Namespace](#), [RedefinableElement](#)

### Association Ends

- **extendedRegion** : [Region](#) [0..1]{subsets [RedefinableElement::redefinedElement](#)} (opposite [A\\_extendedRegion\\_region::region](#))  
The region of which this region is an extension.
- **/redefinitionContext** : [Classifier](#) [1..1]{redefines [RedefinableElement::redefinitionContext](#)} (opposite [A\\_redefinitionContext\\_region::region](#))  
References the Classifier in which context this element may be redefined.
- **state** : [State](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [State::region](#))  
The State that owns the Region. If a Region is owned by a State, then it cannot also be owned by a StateMachine.
- **stateMachine** : [StateMachine](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [StateMachine::region](#))  
The StateMachine that owns the Region. If a Region is owned by a StateMachine, then it cannot also be owned by a State.
- **♦ subvertex** : [Vertex](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [Vertex::container](#))  
The set of Vertices that are owned by this Region.
- **♦ transition** : [Transition](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [Transition::container](#))  
The set of Transitions owned by the Region.



## Operations

- `belongsToPSM() : Boolean`  
The operation `belongsToPSM ()` checks if the Region belongs to a ProtocolStateMachine.

```
body: if stateMachine <> null
then
    stateMachine.oclIsKindOf(ProtocolStateMachine)
else
    state <> null implies state.container.belongsToPSM()
endif
```

- `containingStateMachine() : StateMachine`  
The operation `containingStateMachine()` returns the StateMachine in which this Region is defined.

```
body: if stateMachine = null
then
    state.containingStateMachine()
else
    stateMachine
endif
```

- `isConsistentWith(redefinee : RedefinableElement) : Boolean`  
The query `isConsistentWith()` specifies that a redefining Region is consistent with a redefined Region provided that the redefining Region is an extension of the Redefined region, i.e., its Vertices and Transitions conform to one of the following: (1) they are equal to corresponding elements of the redefined Region or, (2) they consistently redefine a State or Transition of the redefined region, or (3) they add new States or Transitions.

```
pre: redefinee.isRedefinitionContextValid(self)
body: -- the following is merely a default body; it is expected that the specific form of this
constraint will be specified by profiles
true
```

- `isRedefinitionContextValid(redefined : Region) : Boolean`  
The query `isRedefinitionContextValid()` specifies whether the redefinition contexts of a Region are properly related to the redefinition contexts of the specified Region to allow this element to redefine the other. The containing StateMachine or State of a redefining Region must Redefine the containing StateMachine or State of the redefined Region.

```
body: if stateMachine->isEmpty() then
    -- the Region is owned by a State
    (state.redefinedState->notEmpty() and
    state.redefinedState.region->includes(redefined))
else -- the region is owned by a StateMachine
    (stateMachine.extendedStateMachine->notEmpty() and
    stateMachine.extendedStateMachine->exists(sm : StateMachine | sm.region->includes(redefined)))
endif
```

- `redefinitionContext() : Classifier`  
The redefinition context of a Region is the nearest containing StateMachine.

```
body: let sm : StateMachine = containingStateMachine() in
if sm._'context' = null or sm.general->notEmpty() then
    sm
else
    sm._'context'
endif
```

## Constraints

- **deep\_history\_vertex**  
A Region can have at most one deep history Vertex.

```
inv: self.subvertex->select (oclIsKindOf(Pseudostate)) ->collect (oclAsType(Pseudostate)) ->
    select(kind = PseudostateKind::deepHistory)->size() <= 1
```

- **shallow\_history\_vertex**  
A Region can have at most one shallow history Vertex.

```
inv: subvertex->select (oclIsKindOf(Pseudostate)) ->collect (oclAsType(Pseudostate)) ->
    select(kind = PseudostateKind::shallowHistory)->size() <= 1
```

- **owned**  
If a Region is owned by a StateMachine, then it cannot also be owned by a State and vice versa.

```
inv: (stateMachine <> null implies state = null) and (state <> null implies stateMachine = null)
```

- **initial\_vertex**  
A Region can have at most one initial Vertex.

```
inv: self.subvertex->select (oclIsKindOf(Pseudostate)) ->collect (oclAsType(Pseudostate)) ->
    select(kind = PseudostateKind::initial)->size() <= 1
```

## State [Class]

### Description

A State models a situation during which some (usually implicit) invariant condition holds. For behavior StateMachines, States are internal elements hidden from view of external parties. However, the states of ProtocolStateMachines are intended to be exposed to the users of their context Classifiers. A protocol State represents an exposed stable situation of its context classifier: when an instance of the Classifier is not processing any BehavioralFeature, external parties interacting with this instance can always know its State configuration.

### Diagrams

[Behavior State Machines](#), [State Machine Redefinition](#), [Object Nodes](#)

### Generalizations

[RedefinableElement](#), [Namespace](#), [Vertex](#)

### Specializations

[FinalState](#)

### Attributes

- `/isComposite` : [Boolean](#) [1..1] = false  
A state with `isComposite=true` is said to be a composite State. A composite State is a State that contains at least one

Region.

- /isOrthogonal : [Boolean](#) [1..1] = false  
A State with isOrthogonal=true is said to be an orthogonal composite State An orthogonal composite State contains two or more Regions.
- /isSimple : [Boolean](#) [1..1] = true  
A State with isSimple=true is said to be a simple State A simple State does not have any Regions and it does not refer to any submachine StateMachine.
- /isSubmachineState : [Boolean](#) [1..1] = false  
A State with isSubmachineState=true is said to be a submachine State Such a State refers to another StateMachine(submachine).

## Association Ends

- ♦ connection : [ConnectionPointReference](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [ConnectionPointReference::state](#))  
The entry and exit connection points used in conjunction with this (submachine) State, i.e., as targets and sources, respectively, in the Region with the submachine State. A connection point reference references the corresponding definition of a connection point Pseudostate in the StateMachine referenced by the submachine State.
- ♦ connectionPoint : [Pseudostate](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [Pseudostate::state](#))  
The entry and exit Pseudostates of a composite State. These can only be entry or exit Pseudostates, and they must have different names. They can only be defined for composite States.
- ♦ deferrableTrigger : [Trigger](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [A\\_deferrableTrigger\\_state::state](#))  
A list of Triggers that are candidates to be retained by the StateMachine if they trigger no Transitions out of the State (not consumed). A deferred Trigger is retained until the StateMachine reaches a State configuration where it is no longer deferred.
- ♦ doActivity : [Behavior](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_doActivity\\_state::state](#))  
An optional Behavior that is executed while being in the State. The execution starts when this State is entered, and ceases either by itself when done, or when the State is exited, whichever comes first.
- ♦ entry : [Behavior](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_entry\\_state::state](#))  
An optional Behavior that is executed whenever this State is entered regardless of the Transition taken to reach the State. If defined, entry Behaviors are always executed to completion prior to any internal Behavior or Transitions performed within the State.
- ♦ exit : [Behavior](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_exit\\_state::state](#))  
An optional Behavior that is executed whenever this State is exited regardless of which Transition was taken out of the State. If defined, exit Behaviors are always executed to completion only after all internal and transition Behaviors have completed execution.
- redefinedState : [State](#) [0..1]{subsets [RedefinableElement::redefinedElement](#)} (opposite [A\\_redefinedState\\_state::state](#))  
The State of which this State is a redefinition.
- /redefinitionContext : [Classifier](#) [1..1]{redefines [RedefinableElement::redefinitionContext](#)} (opposite [A\\_redefinitionContext\\_state::state](#))

References the Classifier in which context this element may be redefined.

- ♦ region : [Region](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [Region::state](#))  
The Regions owned directly by the State.
- ♦ stateInvariant : [Constraint](#) [0..1]{subsets [Namespace::ownedRule](#)} (opposite [A stateInvariant owningState::owningState](#))  
Specifies conditions that are always true when this State is the current State. In ProtocolStateMachines state invariants are additional conditions to the preconditions of the outgoing Transitions, and to the postcondition of the incoming Transitions.
- submachine : [StateMachine](#) [0..1] (opposite [StateMachine::submachineState](#))  
The StateMachine that is to be inserted in place of the (submachine) State.

## Operations

- containingStateMachine() : [StateMachine](#)  
The query containingStateMachine() returns the StateMachine that contains the State either directly or transitively.

```
body: container.containingStateMachine()
```

- isComposite() : [Boolean](#)  
A composite State is a State with at least one Region.

```
body: region->notEmpty()
```

- isConsistentWith(redefinee : [RedefinableElement](#)) : [Boolean](#)  
The query isConsistentWith() specifies that a redefining State is consistent with a redefined State provided that the redefining State is an extension of the redefined State A simple State can be redefined (extended) to become a composite State (by adding one or more Regions) and a composite State can be redefined (extended) by adding Regions and by adding Vertices, States, and Transitions to inherited Regions. All States may add or replace entry, exit, and 'doActivity' Behaviors.

```
pre: redefinee.isRedefinitionContextValid(self)
body: -- the following is merely a default body; it is expected that the specific form of this
constraint will be specified by profiles
true
```

- isOrthogonal() : [Boolean](#)  
An orthogonal State is a composite state with at least 2 regions.

```
body: region->size () > 1
```

- isRedefinitionContextValid(redefined : [State](#)) : [Boolean](#)  
The query isRedefinitionContextValid() specifies whether the redefinition contexts of a State are properly related to the redefinition contexts of the specified State to allow this element to redefine the other. This means that the containing Region of a redefining State must redefine the containing Region of the redefined State.

```
body: container.redefinedElement.oclassType(Region)->exists(r:Region | r.subvertex-
>includes(redefined))
```

- `isSimple() : Boolean`  
A simple State is a State without any regions.

```
body: region->isEmpty()
```

- `isSubmachineState() : Boolean`  
Only submachine State references another StateMachine.

```
body: submachine <> null
```

- `redefinitionContext() : Classifier`  
The redefinition context of a State is the nearest containing StateMachine.

```
body: let sm : StateMachine = containingStateMachine() in
if sm._'context' = null or sm.general->notEmpty() then
  sm
else
  sm._'context'
endif
```

## Constraints

- `entry_or_exit`  
Only entry or exit Pseudostates can serve as connection points.

```
inv: connectionPoint->forAll(kind = PseudostateKind::entryPoint or kind = PseudostateKind::exitPoint)
```

- `submachine_states`  
Only submachine States can have connection point references.

```
inv: isSubmachineState implies connection->notEmpty( )
```

- `composite_states`  
Only composite States can have entry or exit Pseudostates defined.

```
inv: connectionPoint->notEmpty() implies isComposite
```

- `destinations_or_sources_of_transitions`  
The connection point references used as destinations/sources of Transitions associated with a submachine State must be defined as entry/exit points in the submachine StateMachine.

```
inv: self.isSubmachineState implies (self.connection->forAll (cp |
  cp.entry->forAll (ps | ps.stateMachine = self.submachine) and
  cp.exit->forAll (ps | ps.stateMachine = self.submachine)))
```

- `submachine_or_regions`  
A State is not allowed to have both a submachine and Regions.

```
inv: isComposite implies not isSubmachineState
```

## StateMachine [Class]

### Description

StateMachines can be used to express event-driven behaviors of parts of a system. Behavior is modeled as a traversal of a graph of Vertices interconnected by one or more joined Transition arcs that are triggered by the dispatching of successive Event occurrences. During this traversal, the StateMachine may execute a sequence of Behaviors associated with various elements of the StateMachine.

### Diagrams

[Behavior State Machines](#), [State Machine Redefinition](#), [Protocol State Machines](#)

### Generalizations

[Behavior](#)

### Specializations

[ProtocolStateMachine](#)

### Association Ends

- ◆ connectionPoint : [Pseudostate](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [Pseudostate::stateMachine](#))  
The connection points defined for this StateMachine. They represent the interface of the StateMachine when used as part of submachine State
- extendedStateMachine : [StateMachine](#) [0..\*]{redefines [Behavior::redefinedBehavior](#)} (opposite [A\\_extendedStateMachine\\_stateMachine::stateMachine](#))  
The StateMachines of which this is an extension.
- ◆ region : [Region](#) [1..\*]{subsets [Namespace::ownedMember](#)} (opposite [Region::stateMachine](#))  
The Regions owned directly by the StateMachine.
- submachineState : [State](#) [0..\*] (opposite [State::submachine](#))  
References the submachine(s) in case of a submachine State. Multiple machines are referenced in case of a concurrent State.

### Operations

- LCA(s1 : [Vertex](#), s2 : [Vertex](#)) : [Region](#)  
The operation LCA(s1,s2) returns the Region that is the least common ancestor of Vertices s1 and s2, based on the StateMachine containment hierarchy.

```
body: if ancestor(s1, s2) then
    s2.container
else
    if ancestor(s2, s1) then
        s1.container
    else
        LCA(s1.container.state, s2.container.state)
    endif
endif
```

- `ancestor(s1 : Vertex, s2 : Vertex) : Boolean`
- The query `ancestor(s1, s2)` checks whether `Vertex s2` is an ancestor of `Vertex s1`.

```
body: if (s2 = s1) then
  true
else
  if s1.container.stateMachine->notEmpty() then
    true
  else
    if s2.container.stateMachine->notEmpty() then
      false
    else
      ancestor(s1, s2.container.state)
    endif
  endif
endif
```

- `isConsistentWith(redefinee : RedefinableElement) : Boolean`
- The query `isConsistentWith()` specifies that a redefining `StateMachine` is consistent with a redefined `StateMachine` provided that the redefining `StateMachine` is an extension of the redefined `StateMachine` : `Regions` are inherited and `Regions` can be added, inherited `Regions` can be redefined. In case of multiple redefining `StateMachine`, extension implies that the redefining `StateMachine` gets orthogonal `Regions` for each of the redefined `StateMachine`.

body: -- the following is merely a default body; it is expected that the specific form of this constraint will be specified by profiles  
true

- `isRedefinitionContextValid(redefined : StateMachine) : Boolean`
- The query `isRedefinitionContextValid()` specifies whether the redefinition context of a `StateMachine` is properly related to the redefinition contexts of the specified `StateMachine` to allow this element to redefine the other. The context Classifier of a redefining `StateMachine` must redefine the context Classifier of the redefined `StateMachine`.

```
body: self._'context'().oclAsType(BehavioredClassifier).redefinedClassifier->includes(redefined.oclAsType(Behavior)._'context'())
```

- `LCAState(v1 : Vertex, v2 : Vertex) : State`
- This utility function is like the `LCA`, except that it returns the nearest composite `State` that contains both input `Vertices`.

```
body: if v2.oclIsTypeOf(State) and ancestor(v1, v2) then
  v2.oclAsType(State)
else if v1.oclIsTypeOf(State) and ancestor(v2, v1) then
  v1.oclAsType(State)
else if (v1.container.state->isEmpty() or v2.container.state->isEmpty()) then
  null.oclAsType(State)
else LCAState(v1.container.state, v2.container.state)
endif endif endif
```

## Constraints

- `connection_points`  
The connection points of a `StateMachine` are Pseudostates of kind entry point or exit point.

```
inv: connectionPoint->forall (kind = PseudostateKind::entryPoint or kind = PseudostateKind::exitPoint)
```

- classifier\_context  
The Classifier context of a StateMachine cannot be an Interface.

```
inv: _('context' <> null implies not _('context'.oclIsKindOf(Interface))
```

- method  
A StateMachine as the method for a BehavioralFeature cannot have entry/exit connection points.

```
inv: specification <> null implies connectionPoint->isEmpty()
```

- context\_classifier  
The context Classifier of the method StateMachine of a BehavioralFeature must be the Classifier that owns the BehavioralFeature.

```
inv: specification <> null implies ( _('context' <> null and specification.featuringClassifier->exists(c | c = _('context'))
```

## Transition [Class]

### Description

A Transition represents an arc between exactly one source Vertex and exactly one Target vertex (the source and targets may be the same Vertex). It may form part of a compound transition, which takes the StateMachine from one steady State configuration to another, representing the full response of the StateMachine to an occurrence of an Event that triggered it.

### Diagrams

[Behavior State Machines](#), [State Machine Redefinition](#), [Protocol State Machines](#)

### Generalizations

[Namespace](#), [RedefinableElement](#)

### Specializations

[ProtocolTransition](#)

### Attributes

- kind : [TransitionKind](#) [1..1] = external  
Indicates the precise type of the Transition.

### Association Ends

- container : [Region](#) [1..1]{subsets [NamedElement::namespace](#)} (opposite [Region::transition](#))  
Designates the Region that owns this Transition.
- ♦ effect : [Behavior](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_effect transition::transition](#))  
Specifies an optional behavior to be performed when the Transition fires.



- ♦ guard : [Constraint](#) [0..1]{subsets [Namespace::ownedRule](#)} (opposite [A\\_guard\\_transition::transition](#))  
A guard is a Constraint that provides a fine-grained control over the firing of the Transition. The guard is evaluated when an Event occurrence is dispatched by the StateMachine. If the guard is true at that time, the Transition may be enabled, otherwise, it is disabled. Guards should be pure expressions without side effects. Guard expressions with side effects are ill formed.
- redefinedTransition : [Transition](#) [0..1]{subsets [RedefinableElement::redefinedElement](#)} (opposite [A\\_redefinedTransition\\_transition::transition](#))  
The Transition that is redefined by this Transition.
- /redefinitionContext : [Classifier](#) [1..1]{redefines [RedefinableElement::redefinitionContext](#)} (opposite [A\\_redefinitionContext\\_transition::transition](#))  
References the Classifier in which context this element may be redefined.
- source : [Vertex](#) [1..1] (opposite [Vertex::outgoing](#))  
Designates the originating Vertex (State or Pseudostate) of the Transition.
- target : [Vertex](#) [1..1] (opposite [Vertex::incoming](#))  
Designates the target Vertex that is reached when the Transition is taken.
- ♦ trigger : [Trigger](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [A\\_trigger\\_transition::transition](#))  
Specifies the Triggers that may fire the transition.

## Operations

- containingStateMachine() : [StateMachine](#)  
The query containingStateMachine() returns the StateMachine that contains the Transition either directly or transitively.

```
body: container.containingStateMachine()
```

- isConsistentWith(redefinee : [RedefinableElement](#)) : [Boolean](#)  
The query isConsistentWith() specifies that a redefining Transition is consistent with a redefined Transition provided that the redefining Transition has the following relation to the redefined Transition: A redefining Transition redefines all properties of the corresponding redefined Transition except the source State and the Trigger.

```
pre: redefinee.isRedefinitionContextValid(self)
body: -- the following is merely a default body; it is expected that the specific form of this
constraint will be specified by profiles
true
```

- redefinitionContext() : [Classifier](#)  
The redefinition context of a Transition is the nearest containing StateMachine.

```
body: let sm : StateMachine = containingStateMachine() in
if sm._'context' = null or sm.general->notEmpty() then
  sm
else
  sm._'context'
endif
```

## Constraints

- **state\_is\_external**

A Transition with kind external can source any Vertex except entry points.

```
inv: (kind = TransitionKind::external) implies
    not (source.ocIsKindOf(Pseudostate) and source.ocAsType(Pseudostate).kind =
PseudostateKind::entryPoint)
```

- **join\_segment\_guards**

A join segment must not have Guards or Triggers.

```
inv: (target.ocIsKindOf(Pseudostate) and target.ocAsType(Pseudostate).kind = PseudostateKind::join)
implies (guard = null and trigger->isEmpty())
```

- **state\_is\_internal**

A Transition with kind internal must have a State as its source, and its source and target must be equal.

```
inv: (kind = TransitionKind::internal) implies
    (source.ocIsKindOf (State) and source = target)
```

- **outgoing\_pseudostates**

Transitions outgoing Pseudostates may not have a Trigger.

```
inv: source.ocIsKindOf(Pseudostate) and (source.ocAsType(Pseudostate).kind <>
PseudostateKind::initial) implies trigger->isEmpty()
```

- **join\_segment\_state**

A join segment must always originate from a State.

```
inv: (target.ocIsKindOf(Pseudostate) and target.ocAsType(Pseudostate).kind = PseudostateKind::join)
implies (source.ocIsKindOf(State))
```

- **fork\_segment\_state**

A fork segment must always target a State.

```
inv: (source.ocIsKindOf(Pseudostate) and source.ocAsType(Pseudostate).kind =
PseudostateKind::Fork) implies (target.ocIsKindOf(State))
```

- **state\_is\_local**

A Transition with kind local must have a composite State or an entry point as its source.

```
inv: (kind = TransitionKind::local) implies
    ((source.ocIsKindOf (State) and source.ocAsType(State).isComposite) or
    (source.ocIsKindOf (Pseudostate) and source.ocAsType(Pseudostate).kind =
PseudostateKind::entryPoint))
```

- **initial\_transition**

An initial Transition at the topmost level Region of a StateMachine that has no Trigger.

```
inv: (source.ocIsKindOf(Pseudostate) and container.stateMachine->notEmpty()) implies
    trigger->isEmpty()
```

- `fork_segment_guards`  
A fork segment must not have Guards or Triggers.

```
inv: (source.oclIsKindOf(Pseudostate) and source.oclAsType(Pseudostate).kind = PseudostateKind::fork)
implies (guard = null and trigger->isEmpty())
```

## TransitionKind [Enumeration]

### Description

TransitionKind is an Enumeration type used to differentiate the various kinds of Transitions.

### Diagrams

- [Behavior State Machines](#)

### Literals

- `internal`  
Implies that the Transition, if triggered, occurs without exiting or entering the source State (i.e., it does not cause a state change). This means that the entry or exit condition of the source State will not be invoked. An internal Transition can be taken even if the SateMachine is in one or more Regions nested within the associated State.
- `local`  
Implies that the Transition, if triggered, will not exit the composite (source) State, but it will exit and re-enter any state within the composite State that is in the current state configuration.
- `external`  
Implies that the Transition, if triggered, will exit the composite (source) State.

## Vertex [Abstract Class]

### Description

A Vertex is an abstraction of a node in a StateMachine graph. It can be the source or destination of any number of Transitions.

### Diagrams

[Behavior State Machines](#)

### Generalizations

[NamedElement](#)

### Specializations

[ConnectionPointReference](#), [Pseudostate](#), [State](#)

## Association Ends

- container : [Region](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [Region::subvertex](#))  
The Region that contains this Vertex.
- /incoming : [Transition](#) [0..\*]{} (opposite [Transition::target](#))  
Specifies the Transitions entering this Vertex.
- /outgoing : [Transition](#) [0..\*]{} (opposite [Transition::source](#))  
Specifies the Transitions departing from this Vertex.

## Operations

- containingStateMachine() : [StateMachine](#)  
The operation containingStateMachine() returns the StateMachine in which this Vertex is defined.

```
body: if container <> null
then
-- the container is a region
  container.containingStateMachine()
else
  if (self.oclIsKindOf(Pseudostate)) and ((self.oclAsType(Pseudostate).kind =
PseudostateKind::entryPoint) or (self.oclAsType(Pseudostate).kind = PseudostateKind::exitPoint)) then
    self.oclAsType(Pseudostate).stateMachine
  else
    if (self.oclIsKindOf(ConnectionPointReference)) then
      self.oclAsType(ConnectionPointReference).state.containingStateMachine() -- no other valid
cases possible
    else
      null
    endif
  endif
endif
endif
```

- incoming() : [Transition](#) [0..\*]  
Derivation for Vertex::/incoming.

```
body: Transition.allInstances()->select(target=self)
```

- outgoing() : [Transition](#) [0..\*]  
Derivation for Vertex::/outgoing

```
body: Transition.allInstances()->select(source=self)
```

- isContainedInState(s : [State](#)) : [Boolean](#)  
This utility operation returns true if the Vertex is contained in the State s (input argument).

```
body: if not s.isComposite() or container->isEmpty() then
  false
else
  if container.state = s then
    true
  else
    container.state.isContainedInState(s)
  endif
endif
```

```
endif
endif
```

- `isContainedInRegion(r : Region) : Boolean`  
This utility query returns true if the Vertex is contained in the Region r (input argument).

```
body: if (container = r) then
  true
else
  if (r.state->isEmpty()) then
    false
  else
    container.state.isContainedInRegion(r)
  endif
endif
endif
```

## 14.6 Association Descriptions

### A\_conformance\_specificMachine [Association]

#### Diagrams

[Protocol State Machines](#)

#### Member Ends

- [ProtocolStateMachine::conformance](#)
- [ProtocolConformance::specificMachine](#)

### A\_connectionPoint\_state [Association]

#### Diagrams

[Behavior State Machines](#)

#### Member Ends

- [State::connectionPoint](#)
- [Pseudostate::state](#)

### A\_connectionPoint\_stateMachine [Association]

#### Diagrams

[Behavior State Machines](#)

#### Member Ends

- [StateMachine::connectionPoint](#)
- [Pseudostate::stateMachine](#)

## A\_connection\_state [Association]

### Diagrams

[Behavior State Machines](#)

### Member Ends

- [State::connection](#)
- [ConnectionPointReference::state](#)

## A\_deferrableTrigger\_state [Association]

### Diagrams

[Behavior State Machines](#)

### Owned Ends

- state : [State](#) [0..1]{subsets [Element::owner](#)} (opposite [State::deferrableTrigger](#))

## A\_doActivity\_state [Association]

### Diagrams

[Behavior State Machines](#)

### Owned Ends

- state : [State](#) [0..1]{subsets [Element::owner](#)} (opposite [State::doActivity](#))

## A\_effect\_transition [Association]

### Diagrams

[Behavior State Machines](#)

### Owned Ends

- transition : [Transition](#) [0..1]{subsets [Element::owner](#)} (opposite [Transition::effect](#))

## A\_entry\_connectionPointReference [Association]

### Diagrams

[Behavior State Machines](#)

### Owned Ends

- connectionPointReference : [ConnectionPointReference](#) [0..1] (opposite [ConnectionPointReference::entry](#))

## A\_entry\_state [Association]

### Diagrams

[Behavior State Machines](#)

### Owned Ends

- state : [State](#) [0..1]{subsets [Element::owner](#)} (opposite [State::entry](#))

## A\_exit\_connectionPointReference [Association]

### Diagrams

[Behavior State Machines](#)

### Owned Ends

- connectionPointReference : [ConnectionPointReference](#) [0..1] (opposite [ConnectionPointReference::exit](#))

## A\_exit\_state [Association]

### Diagrams

[Behavior State Machines](#)

### Owned Ends

- state : [State](#) [0..1]{subsets [Element::owner](#)} (opposite [State::exit](#))

## A\_extendedRegion\_region [Association]

### Diagrams

[State Machine Redefinition](#)

### Owned Ends

- region : [Region](#) [0..1]{subsets [A\\_redefinedElement\\_redefinableElement::redefinableElement](#)} (opposite [Region::extendedRegion](#))

## A\_extendedStateMachine\_stateMachine [Association]

### Diagrams

[State Machine Redefinition](#)

### Owned Ends

- stateMachine : [StateMachine](#) [0..1]{subsets [A\\_redefinedBehavior\\_behavior::behavior](#)} (opposite [StateMachine::extendedStateMachine](#))

## A\_generalMachine\_protocolConformance [Association]

### Diagrams

[Protocol State Machines](#)

### Owned Ends

- protocolConformance : [ProtocolConformance](#) [0..\*]{subsets [A\\_target\\_directedRelationship::directedRelationship](#)} (opposite [ProtocolConformance::generalMachine](#))

## A\_guard\_transition [Association]

### Diagrams

[Behavior State Machines](#)

### Specializations

[A\\_preCondition\\_protocolTransition](#)



## Owned Ends

- transition : [Transition](#) [0..1]{subsets [Constraint::context](#)} (opposite [Transition::guard](#))

## A\_incoming\_target\_vertex [Association]

### Diagrams

[Behavior State Machines](#)

### Member Ends

- [Vertex::incoming](#)
- [Transition::target](#)

## A\_outgoing\_source\_vertex [Association]

### Diagrams

[Behavior State Machines](#)

### Member Ends

- [Vertex::outgoing](#)
- [Transition::source](#)

## A\_postCondition\_owningTransition [Association]

### Diagrams

[Protocol State Machines](#)

### Owned Ends

- owningTransition : [ProtocolTransition](#) [0..1]{subsets [Constraint::context](#)} (opposite [ProtocolTransition::postCondition](#))

## A\_preCondition\_protocolTransition [Association]

### Diagrams

[Protocol State Machines](#)

### Generalizations

[A\\_guard\\_transition](#)

## Owned Ends

- protocolTransition : [ProtocolTransition](#) [0..1]{redefines [A\\_guard\\_transition::transition](#)} (opposite [ProtocolTransition::preCondition](#))

## A\_redefinedState\_state [Association]

### Diagrams

[State Machine Redefinition](#)

## Owned Ends

- state : [State](#) [0..1]{subsets [A\\_redefinedElement\\_redefinableElement::redefinableElement](#)} (opposite [State::redefinedState](#))

## A\_redefinedTransition\_transition [Association]

### Diagrams

[State Machine Redefinition](#)

## Owned Ends

- transition : [Transition](#) [0..1]{subsets [A\\_redefinedElement\\_redefinableElement::redefinableElement](#)} (opposite [Transition::redefinedTransition](#))

## A\_redefinitionContext\_region [Association]

### Diagrams

[State Machine Redefinition](#)

## Generalizations

[A\\_redefinitionContext\\_redefinableElement](#)

## Owned Ends

- region : [Region](#) [0..\*]{redefines [A\\_redefinitionContext\\_redefinableElement::redefinableElement](#)} (opposite [Region::redefinitionContext](#))

## A\_redefinitionContext\_state [Association]

### Diagrams

[State Machine Redefinition](#)

### Generalizations

[A\\_redefinitionContext\\_redefinableElement](#)

### Owned Ends

- state : [State](#) [0..\*]{redefines [A\\_redefinitionContext\\_redefinableElement::redefinableElement](#)} (opposite [State::redefinitionContext](#))

## A\_redefinitionContext\_transition [Association]

### Diagrams

[State Machine Redefinition](#)

### Generalizations

[A\\_redefinitionContext\\_redefinableElement](#)

### Owned Ends

- transition : [Transition](#) [0..\*]{redefines [A\\_redefinitionContext\\_redefinableElement::redefinableElement](#)} (opposite [Transition::redefinitionContext](#))

## A\_referred\_protocolTransition [Association]

### Diagrams

[Protocol State Machines](#)

### Owned Ends

- protocolTransition : [ProtocolTransition](#) [0..\*] (opposite [ProtocolTransition::referred](#))

## A\_region\_state [Association]

### Diagrams

[Behavior State Machines](#)

### Member Ends

- [State::region](#)
- [Region::state](#)

### A\_region\_stateMachine [Association]

#### Diagrams

[Behavior State Machines](#)

### Member Ends

- [StateMachine::region](#)
- [Region::stateMachine](#)

### A\_stateInvariant\_owningState [Association]

#### Diagrams

[Behavior State Machines](#)

### Owned Ends

- owningState : [State](#) [0..1]{subsets [Constraint::context](#)} (opposite [State::stateInvariant](#))

### A\_submachineState\_submachine [Association]

#### Diagrams

[Behavior State Machines](#)

### Member Ends

- [StateMachine::submachineState](#)
- [State::submachine](#)

### A\_subvertex\_container [Association]

#### Diagrams

[Behavior State Machines](#)

### Member Ends

- [Region::subvertex](#)
- [Vertex::container](#)

## A\_transition\_container [Association]

### Diagrams

[Behavior State Machines](#)

### Member Ends

- [Region::transition](#)
- [Transition::container](#)

## A\_trigger\_transition [Association]

### Diagrams

[Behavior State Machines](#)

### Owned Ends

- transition : [Transition](#) [0..1]{subsets [Element::owner](#)} (opposite [Transition::trigger](#))

# 15 Activities

## 15.1 Summary

An Activity is a kind of Behavior (see sub clause [13.2](#)) that is specified as a graph of *nodes* interconnected by *edges*. A subset of the nodes are *executable nodes* that embody lower-level steps in the overall Activity. *Object nodes* hold data that is input to and output from executable nodes, and moves across *object flow* edges. *Control nodes* specify sequencing of executable nodes via *control flow* edges. Activities are essentially what are commonly called “control and data flow” models. Such models of computation are inherently concurrent, as any sequencing of activity node execution is modeled explicitly by activity edges, and no ordering is mandated for any computation not explicitly sequenced.

Activities may describe procedural computation, forming hierarchies of Activities invoking other Activities, or, in an object-oriented model, they may be invoked indirectly as methods bound to Operations that are directly invoked. Activities may be applied to organizational modeling for business process engineering and workflow. In this context, events often originate from inside the system, such as the finishing of a task, but also from outside the system, such as a customer call. Activities can also be used for information system modeling to specify system level processes.

The remainder of this clause describes how activity models are structured and various kinds of object and control nodes. The only kind of executable nodes in UML are *Actions*, which are fully described in Clause [16](#). Actions are required for any significant capabilities of Activities. Actions invoke other Behaviors and Operations (see above), access and modify objects, as well as link them together, and perform more advanced coordination of other Actions (Structured Actions). They are central to the “data flow” aspects of Activities, introducing a specialized form of object node (Pins) for object flows to get and provide data to Actions. Most of the examples of Activity data flow appear in Clause [16](#). The concrete syntax for Actions is a subset of the concrete syntax for Activities (Action notation only appears in Activity diagrams), and some concrete syntax for Actions is specified in this clause. This clause uses executable nodes to provide some independence from Actions but must still be read in conjunction with Clause [16](#).

## 15.2 Activities

### 15.2.1 Summary

An Activity is a Behavior specified as sequencing of subordinate units, using a control and data flow model. Subordinate behaviors coordinated by these models may be initiated because other behaviors in the model finish executing, because objects and data become available or because events occur externally to the flow. The flow of execution is modeled as ActivityNodes connected by ActivityEdges. An ExecutableNode can be the execution of a subordinate behavior, such as an arithmetic computation, a call to an operation, or manipulation of object contents (see Clause [16](#) on Actions for details). ActivityNodes also include flow-of-control constructs, such as synchronization, decision, and concurrency control.

This subclause describes the basic structure and flow semantics of an activity model as a graph of nodes and edges. Subsequent subclauses then describe the various kinds of ActivityNodes that an Activity may contain and how those nodes may be grouped within the Activity.

## 15.2.2 Abstract Syntax

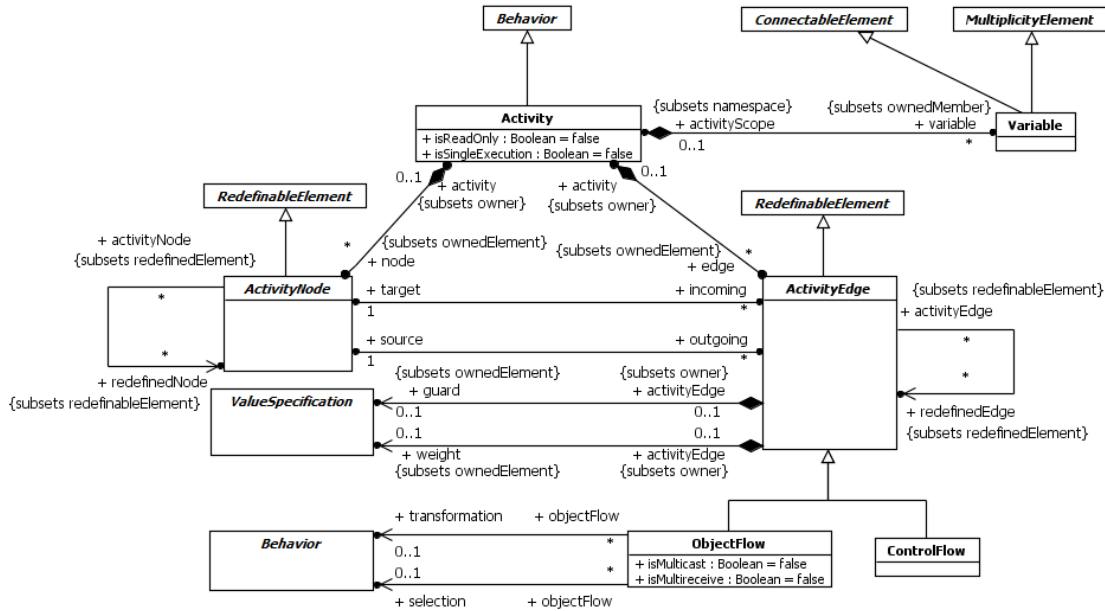


Figure 15.1 Activities

## 15.2.3 Semantics

### Activities

The execution of one **ActivityNode** within an **Activity** may affect, and may be affected by, the execution of other **ActivityNodes** in the **Activity**. Such edges are represented by **ActivityEdges** that interconnect the **ActivityNodes**. The effect of one **ActivityNode** on another is specified by the *flow of tokens* over the **ActivityEdges** between the **ActivityNodes**.

*Tokens* are not explicitly modeled in an **Activity**, but are used for describing the execution of an **Activity**. An *object token* is a container for a value that flows over **ObjectFlow** edges (some object tokens can flow over **ControlFlow** edges, as specified by the modeler, see `isControlType` for **ObjectNodes** in sub clause 15.4). An object token with no value in it is called a *null* token. A *control token* affects execution of **ActivityNodes**, but does not carry any data, and flows only over **ControlFlow** edges. Each token is distinct from any other, even if it contains the same value as another.

**ActivityEdges** are directed, with tokens flowing from the source **ActivityNode** to the target **ActivityNode**. However, tokens *offered* to an **ActivityEdge** by the source **ActivityNode** may not immediately flow along the edge. Instead, the tokens only move when the offer is *accepted* by the **ActivityEdge**, which requires at least the target **ActivityNode** to accept them also, which in turn might depend on acceptance of cascading offers of the same tokens to edges and nodes further downstream of the target. As described below, object tokens shall only be accepted by **ObjectNodes** while control tokens shall only be accepted by **ExecutableNodes** (with a modeler-specified exception for some object tokens accepted by **ExecutableNodes**, see `isControlType` for **ObjectNodes** in sub clause 15.4). **ControlNodes** are used to control the routing of offers through a network of **ActivityEdges**, controlling the flow of accepted tokens.

**ActivityNodes** and **ActivityEdges** may be named, however, the nodes and edges of an **Activity** are not required to have unique names within that **Activity**. This allows, for example, similar nodes within an **Activity** (such as multiple invocations of other **Behavior**) to be given the same name. Even though an **Activity** is a **Namespace** (a **Behavior** is a **Class**, which is a **Classifier**, which is a **Namespace**), and the members of a **Namespace** are required to be distinguishable (see sub clause 7.4), this constraint

does not affect the naming of Activity nodes and edges because the nodes and edges of an Activity are ownedElements but *not* ownedMembers of the Activity.

**NOTE.** Activities are Classes (see sub clause [13.2](#)) and may support Properties, such as how long the process has been executing or how much it costs; Associations specifying links to objects, such as the performer of the execution, who to report completion to, or resources being used; Operations for managing execution of their instances, such as starting, stopping, aborting, and so on; and StateMachines specifying states of execution such as started, suspended, and so on. Profiles may include class libraries with standard Classes that are used as root classes for activities in the user model and vendors may define their own libraries, or support user defined features on Activity Classes.

## Activity Nodes

ActivityNodes are used to model the individual steps in the behavior specified by an Activity.

An ActivityNode is enabled to begin execution when specified conditions are satisfied on the tokens offered to it on incoming ActivityEdges; the conditions depend on the kind of node. When an ActivityNode begins execution, tokens are accepted from some or all of its incoming ActivityEdges and a token is *placed on* the node. When a node completes execution, a token is removed from the node and tokens are offered to some or all of its outgoing ActivityEdges. The actual effect of the node execution depends on the kind of the node, as detailed in subsequent subclauses.

All restrictions on the relative execution order of two or more ActivityNodes are explicitly constrained by ActivityEdge relationships. If two ActivityNodes are not ordered by ActivityEdge relationships (directly or indirectly, e.g., by being separately contained in ordered StructuredActivityNodes; see sub clause [16.11](#)), they may execute concurrently.

**NOTE.** As used here, *concurrent* execution simply means that there is no required order in which the nodes must be executed; a conforming execution of the Activity may execute the nodes sequentially in either order or may execute them in parallel.

As an ActivityNode may be the source for multiple ActivityEdges, the same token can be offered to multiple targets. However, the same token can only be accepted at one target at a time (unless it is copied, whereupon it is not the same token, see ForkNodes in sub clause 15.3 and ExecutableNodes in sub clause 15.5). If a token is offered to multiple ActivityNodes at the same time, it shall be accepted by at most one of them, but exactly which one is not completely determined by the Activity flow semantics. This means that an Activity model in which non-determinacy occurs may be subject to timing issues and race conditions. It is the responsibility of the modeler to avoid such conditions in the construction of the Activity model, if they are not desired.

There are three kinds of ActivityNodes:

1. ControlNodes act as “traffic switches” managing the flow of tokens across ActivityEdges. Tokens cannot “rest” at ControlNodes (with exceptions for InitialNodes and ForkNodes, see sub clause 15.3).
2. ObjectNodes hold object tokens accepted from incoming ObjectFlows and may subsequently offer them to outgoing ObjectFlows (with a modeler-specified exception for ControlFlows, see isControlType for ObjectNodes in sub clause 15.4).
3. ExecutableNodes actually carry out the desired behavior of an Activity. If an ExecutableNode has incoming ControlFlows, then there must be tokens offered on all these flows that it accepts before beginning execution. While executing, an ExecutableNode is considered to hold a single control token indicating it is executing. When it completes execution, it offers control tokens on all outgoing ControlFlows. All incoming and outgoing ActivityEdges of an ExecutableNode must be ControlFlows. (Actions, which are the only kind of ExecutableNode, use special attached ObjectNodes called *Pins* to accept input and produce output object tokens; see Clause [16](#).)

Each of these kinds of ActivityNodes are described further in subsequent subclauses.

## Activity Edges

An ActivityEdge is a directed connection between two ActivityNodes along which tokens may flow, from the source ActivityNode to the target ActivityNode.



Tokens are *offered* to an ActivityEdge by the source ActivityNode of the edge. Offers propagate through ActivityEdges and ControlNodes, according to the rules associated with ActivityEdges (see below) and each kind of ControlNode (see sub clause 15.3) until they reach an ObjectNode (for object tokens) or an ExecutableNode (for control tokens and some object tokens as specified by modelers, see ObjectNodes in sub clause 15.4). Each kind of ObjectNode (see sub clause 15.4) and ExecutableNode (see sub clause 15.5 and Clause 16 on Actions) has rules for when offered tokens may be *accepted*. If an ObjectNode or ExecutableNode accepts an offered token, then that token flows from its original offering ActivityNode to the accepting ActivityNode. As described above, there may be contention between multiple nodes to which a token is offered – the concept of offers defines the semantics for managing such contention.

An ActivityEdge may have a guard, which is a ValueSpecification that is evaluated for each token offered to the edge. An offer shall only pass along an ActivityEdge if the guard for the edge evaluates to true for the offered token. An ActivityEdge without a guard is equivalent to one with a guard that evaluates to true for every token. (Guards are commonly used with DecisionNodes, as described in sub clause 15.3, but they are allowed on any ActivityEdge.)

Any number of tokens can pass along an ActivityEdge, in groups at one time, or individually at different times. The weight property dictates the minimum number of tokens that must traverse the edge at the same time. It is a ValueSpecification that is evaluated every time a new token is offered by the source ActivityNode. It must evaluate to a positive LiteralUnlimitedNatural and may be a constant. Once the minimum number of tokens are offered, all the tokens offered by the source are offered to the target all at once. The minimum number of tokens must then be accepted before any tokens shall traverse the edge. If the ActivityEdge has a guard, the guard must evaluate to true for each token offered to the edge that counts towards the minimum. If the guard fails for any of the tokens, and this reduces the number of tokens that can be offered to the target to less than the weight, then all the tokens fail to be offered. An unlimited weight means that all the tokens offered by the source must be accepted before any of them shall traverse the edge. (This can be combined with a JoinNode to take all of the tokens at the source when certain conditions hold; see examples in Figure 15.21 and Figure 15.59). If a weight is not specified for an ActivityEdge, this is equivalent to specifying a weight of 1.

**NOTE.** A weaker but simpler alternative to weight is to group information into larger objects so that a single token carries all necessary data.

There are two kinds of ActivityEdges:

1. A ControlFlow is an ActivityEdge that only passes control tokens (and some object tokens as specified by modelers, see isControlType for ObjectNodes in sub clause 15.4). ControlFlows are used to explicitly sequence execution of ActivityNodes, as the target ActivityNode cannot receive a control token and start execution until the source ActivityNode completes execution and produces the token.
2. An ObjectFlow is an ActivityEdge that can have object tokens passing along it. ObjectFlows model the flow of values between ObjectNodes. Tokens are offered to the target ActivityNode in the same order as they are offered from the source. If multiple tokens are offered at the same time, then the tokens are offered in the same order as if they had been offered one at a time from the source. If the source is an ObjectNode with an ordering specified, then tokens from the source are offered to the ObjectFlow in that order and, consequently, are offered from the ObjectFlow to the target in the same order. (See also sub clause 15.4 on the offering of tokens from an ObjectNode.)

Unlike ControlFlows, ObjectFlows also provide additional support for multicast/receive, token selection from ObjectNodes and transformation of tokens, as described below.

## Object Flows

Object tokens pass over ObjectFlows, carrying data through an Activity via their values, or carrying no data (*null tokens*). A null token can still be passed along an ObjectFlow and used like any other token. For example, an Action can output a null token to explicitly indicate that it did not produce an optional value, and a downstream DecisionNode (see sub clause 15.3) can test for this and branch accordingly.

An ObjectFlow may have a transformation Behavior that has a single input Parameter and a single output Parameter. If a transformation Behavior is specified, then the Behavior is invoked for each object token offered to the ObjectFlow, with the value

in the token passed to the Behavior as input (for a null token, the behavior is invoked but no value is passed). The output of the Behavior is put in an object token that is offered to the target ActivityNode instead of the original object token. If the output parameter of the Behavior has a multiplicity upper bound greater than 1, and the Behavior produces multiple values, then each value is put in a separate object token, all of which are passed to the target ActivityNode (if the output Parameter is ordered, this ordering is preserved in the sequencing of the tokens). If the output Parameter has a multiplicity lower bound of 0 and the Behavior produces no value, then a null token is offered to the target ActivityNode.

An ObjectFlow may have a selection Behavior that has a single input Parameter and a single output Parameter. The input Parameter of the Behavior must be unordered, nonunique and have a multiplicity of 0..\* (a “bag”), and the output Parameter must have a multiplicity upper bound of 1. If a selection Behavior is specified, then it is used to offer a token from a source ObjectNode to the ObjectFlow, rather than using the ObjectNode’s ordering. Whenever a new token is offered to the ObjectFlow, or an offer is withdrawn, the selection Behavior is invoked with the values from all the object tokens currently being offered to the ObjectFlow passed to the Behavior input Parameter. The selection Behavior should then select one of the input values and produce it as output. This output value is put in an object token and passed to the target ActivityNode. If the selection Behavior does not produce an output, then a null token is passed to the target ActivityNode.

If an ObjectFlow has both a transformation and a selection Behavior, then the transformation Behavior is invoked first when a new token is offered to the ObjectFlow and the resulting value is used in the invocation of the selection behavior.

Because a transformation or selection Behavior is used while offering tokens to the target node, it may be run many times on the same token before the token is accepted by the target node. This means the Behavior cannot have side effects. It shall not modify objects, but transformations may for example, navigate from one object to another, get an attribute value from an object, or replace a data value with another.

Multicasting and multireceiving are used in conjunction with ActivityPartitions (see sub clause 15.6) to model flows between Behaviors that are the responsibility of objects determined by a publish and subscribe facility. However, the particular publish/subscribe semantics used are not specified in this standard. (To support execution, a model must, therefore, be refined to specify the particular publish/subscribe facility employed.) This is illustrated in Figure 15.7 in sub clause 15.2.5.

## Variables

ObjectFlows provide the primary means for moving data within an Activity. Variables provide an alternate means for passing data indirectly.

During the execution of an Activity, each of the Variables of the Activity may hold one or more values. There are Actions to write values to Variables and to subsequently read values from those Variables (as described in sub clause 16.9). The Variables of an Activity are ownedMembers of the Activity considered as a Namespace, but they are local to the Activity and are not visible outside it.

The use of a Variable effectively provides indirect data flow paths from the point at which a value is written to the Variable to all the points at which the value is read from the Variable. Because there is no predefined relationship between the Actions within an Activity that read from and write to Variables, these actions must be sequenced by control flows to prevent race conditions that may occur between Actions that read or write the same Variable.

A Variable is a kind of ConnectableElement (see sub clause 11.2) and, as such, is a TypedElement (see sub clause 7.5). Any values held by a Variable must conform to the Type of the Variable.

A Variable is also a MultiplicityElement (see sub clause 7.5). If the upper bound on a Variable is 1, then that Variable may only hold a single value. If the upper bound on a Variable is greater than 1, then it may hold multiple values up to the maximum number given by the upper bound (or an unbounded number, if the upper bound is “\*”). If the lower bound on a Variable is anything other than 0, then the Variable should nominally always hold at least as many values as given by the lower bound. However, as the only way to write values into a Variable is through Actions within the Activity, it is not always possible to enforce such a multiplicity lower bound. (For further discussion of the semantics of Variable multiplicity, see the description of the Actions used to read from and write to Variables, in sub clause 16.9.)

**NOTE.** Variables are introduced to simplify translation of common programming languages into activity models for those applications that do not require object flow information to be readily accessible. However, source programs that set variables only once can be easily translated to use object flows from the action that determines the values to the actions that use them. Source programs that set variables more than once can be translated to object flows by introducing a local object containing properties for the variables, or one object per variable combined with data store nodes.

## Activity Execution

An Activity may have precondition and postcondition Constraints, as inherited from Behavior (see sub clause [13.2](#)). These apply globally to all invocations of the Activity. (Actions within an Activity may also have local pre- and postconditions, see sub clause [16.2](#).)

As a Behavior, an Activity may have Parameters (see sub clause [13.2](#)). For each such Parameter, the Activity has a corresponding ActivityParameterNode (two in the case of an inout Parameter, one for input and one for output). An ActivityParameterNode is an ObjectNode that makes Parameter values accessible within the Activity. (See sub clause [15.4](#) for a full discussion of ActivityParameterNodes.)

When an Activity is invoked, any values passed to its input Parameters are put in object tokens and placed on the corresponding input ActivityParameterNodes for the Activity (if an input parameter has no value, a null token is placed on the corresponding ActivityParameterNode). These ActivityParameterNodes then offer their tokens to outgoing ActivityEdges.

When an Activity is first invoked, none of its nodes other than input ActivityParameterNodes will initially hold any tokens. However, nodes that do not have incoming edges and require no input data to execute are immediately enabled. A single control token is placed on each enabled node and they begin executing concurrently. Such nodes include ExecutableNodes (see sub clause [15.5](#)) with no incoming ControlFlows and no mandatory input data and InitialNodes (see sub clause [15.3](#)).

On each subsequent invocation of the Activity, the `isSingleExecution` property indicates whether the same execution of the Activity handles tokens for all invocations, or a separate execution of the Activity is created for each invocation. For example, an Activity that models a manufacturing plant might have a parameter for an order to fill. Each time the activity is invoked, a new order enters the flow. As there is only one plant, one execution of the Activity handles all orders. This applies even if the Activity is an Operation method (see sub clause [13.2](#)), for example, on each order.

If a single execution of the Activity is used for all invocations, the modeler must consider the interactions between the multiple streams of tokens moving through the ActivityNodes and ActivityEdges. Tokens may reach bottlenecks waiting for other tokens ahead of them to move downstream, they may overtake each other due to variations in the execution time of invoked behaviors, and most importantly, may abort each other with constructs such as ActivityFinalNodes (see sub clause [15.3](#)).

If a separate execution of the Activity is used for each invocation (this is the default), tokens from the various invocations do not interact. For example, an Activity that is a classifierBehavior is invoked when the Classifier is instantiated (see sub clause [13.2](#)), and the modeler will usually want a separate execution of the Activity for each instance of the classifier. The same is true for modeling methods in common programming languages, which have separate stack frames for each method call.

However, if an Activity has streaming Parameters (see sub clause [13.2](#)), then additional tokens may flow into and out of the Activity (via the corresponding ActivityParameterNodes) even during the course of a single execution. This may result in the same sorts of token interaction issues that result from using a single execution.

The execution of an Activity with no streaming Parameters completes when it has no nodes executing and no nodes enabled for execution, or when it is explicitly terminated using an ActivityFinalNode (see sub clause [15.3](#)). The execution of an Activity with streaming input Parameters shall not terminate until the cumulative number of values posted to each of those input Parameters (by the invoker of the Activity) is at least equal to the Parameter multiplicity lower bound. The execution of an Activity with streaming output Parameters shall not terminate until the cumulative number of values posted to each of those output Parameters (by the Activity itself) is at least equal to the Parameter multiplicity lower bound.

When the execution of an Activity completes, all ActivityParameterNodes corresponding to non-streaming output Parameters shall hold at least as many non-null object tokens as given by the corresponding Parameter multiplicity lower bound. The values

associated with the object tokens of each output ActivityParameterNode are then passed out of the Activity on the corresponding output Parameter and made available to the invoker of the Activity.

An output Parameter may also be identified as an *exception* Parameter by having `isException=true` (see sub clause 9.4). An output posted to an exception Parameter precludes outputs from being posted to other output Parameters of a Behavior. If an object token arrives at an output ActivityParameterNode associated with an exception Parameter, then the execution of the Activity is immediately terminated. The value on the token is then passed to the exception Parameter as usual, but any tokens on other output ActivityParameterNodes associated with non-streaming Parameters are lost and their values are not passed to the associated Parameters. Values posted to streaming output Parameters before the termination of the Activity are not affected.

Use exception Parameters on Activities only if it is desired to abort all flows in the Activity. For example, if the same execution of an activity is being used for all its invocations (i.e., `isSingleExecution=true`), then multiple streams of tokens will be flowing through the same Activity. In this case, it is probably not desired to abort all flows just because one reaches an exception output. Arrange for separate invocations of the Activity to use separate executions of the Activity (i.e., `isSingleExecution=false`) when employing exception Parameters, so flows from separate executions will not affect each other.

## Activity Generalization

An Activity is a Classifier and, as such, may participate in Generalization relationships. A specialized Activity inherits the nodes and edges of its general Activities. ActivityNodes and ActivityEdges are RedefinableElements (see sub clause 9.2) that may be redefined in a specialized Activity.

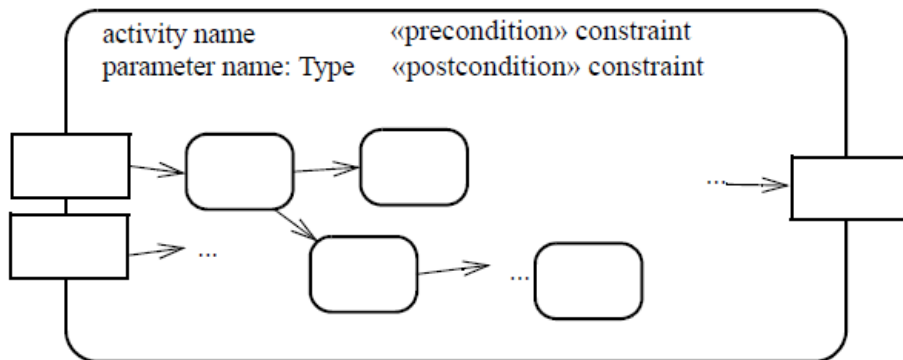
An ActivityNode in a specialized Activity that redefines an ActivityNode from a general Activity is considered to replace the redefined ActivityNode for any inherited ActivityEdges that had the redefined ActivityNode as a source or target. Similarly, an ActivityEdge that redefines an ActivityEdge from a general Activity is considered to replace the redefined ActivityEdge for any inherited ActivityNode that had the redefined ActivityEdge as an incoming or outgoing edge. If the redefined ActivityEdge is an incoming or outgoing edge for any ActivityNode that is not inherited but is itself redefined, then the ActivityEdge is replaced for the redefining ActivityNode.

The effective sets of nodes and edges used in executing a specialized Activity consists of the unions of the inherited nodes and edges (which do not include redefined nodes and edges) and any additional nodes and edges defined in the specialized Activity (including any redefining nodes and edges). The execution of the specialized Activity then proceeds as usual, but using a graph of nodes and edges constructed from the union sets.

## 15.2.4 Notation

This subclause specifies a graphical notation for Activities. This notation is optional in that a conforming tool may use a textual concrete syntax instead. However, the notation given in this and subsequent notation subclauses within this clause is the only graphical notation for Activities conformant with this specification.

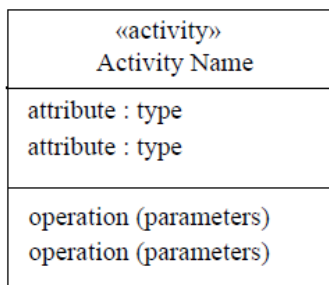
The notation for an Activity is a combination of the notations of the ActivityNodes and ActivityEdges it contains, plus a border and name displayed in the upper left corner. ActivityParameterNodes are displayed on the border (see also the notation for ActivityParameterNode in sub clause 15.4). Pre- and post-condition constraints, inherited from Behavior, are shown as textual expressions with the keywords «precondition» and «postcondition», respectively. The keyword «singleExecution» is used for Activities `isSingleExecution=true`.



**Figure 15.2 Activity notation**

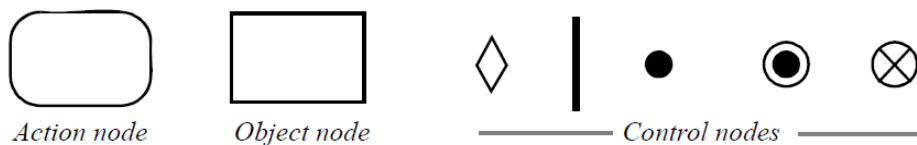
The round-cornered border of Figure 15.2 may be replaced with the frame notation described in Annex A. ActivityParameterNodes are displayed on the frame. The round-cornered border or frame may also be omitted completely, in which case ActivityParameterNodes may appear anywhere on the diagram.

The notation for Classes can be used for diagramming the features of an Activity as shown in Figure 15.3, with the keyword «activity». They can be shown in class diagrams with associations.



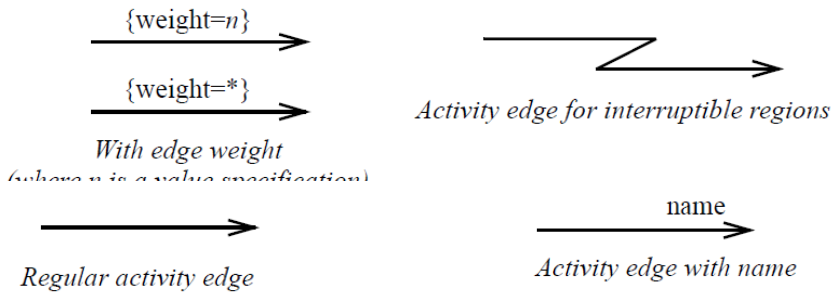
**Figure 15.3 Activity class notation**

The notations for ActivityNodes are illustrated below. This notation is discussed in more detail in the following subclauses (and in Clause 16 for Actions).



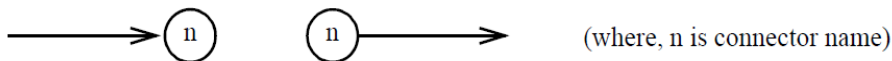
**Figure 15.4 ActivityNode notation**

An ActivityEdge (whether a ControlFlow or ObjectFlow) is notated by an open arrowhead line connecting two ActivityNodes. If the edge has a name, it is notated near the arrow. Guards are shown as text in square brackets near tail of the line.



**Figure 15.5 ActivityEdge notation**

An ActivityEdge may also be notated using a connector, which is a small circle with the name of the edge in it. This is purely notational. It does not affect the underlying model. The circles and lines involved map to a single ActivityEdge in the model. Every connector with a given label must be paired with exactly one other with the same label on the same Activity diagram. One connector must have exactly one incoming edge and the other exactly one outgoing edge, each with the same type of flow, object or control.



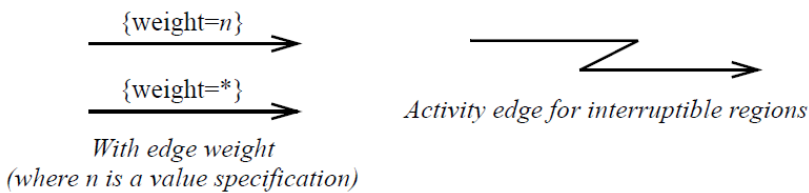
**Figure 15.6 ActivityEdge connector notation**

The weight of an ActivityEdge may be shown in curly braces using the notation:

*weight-annotation* ::= '{ 'weight' '=' value-specification '}'

The weight is a value specification, which may be a constant, that evaluates to a non-zero unlimited natural value. An unlimited weight is notated as “\*”. (See also the notation for ValueSpecifications in Clause 8.)

An interruptingEdge of an InterruptibleRegion can be notated with a lightning-bolt (see also the alternative notation for interruptingEdges in sub clause 15.6).



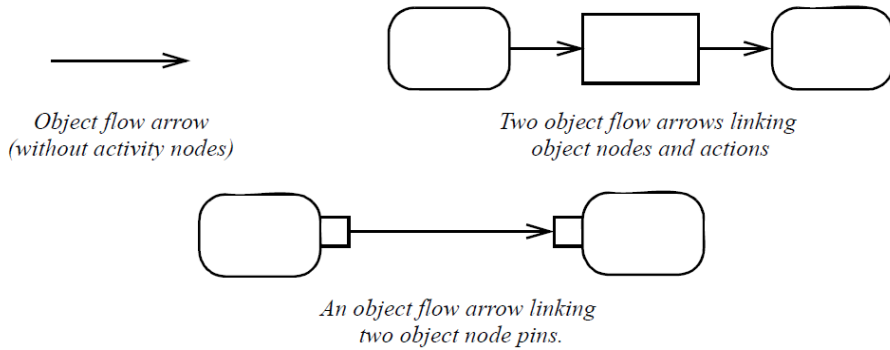
**Figure 15.7 ActivityEdge notation**

A control flow is notated by an arrowed line connecting two actions.



**Figure 15.8 ControlFlow notation**

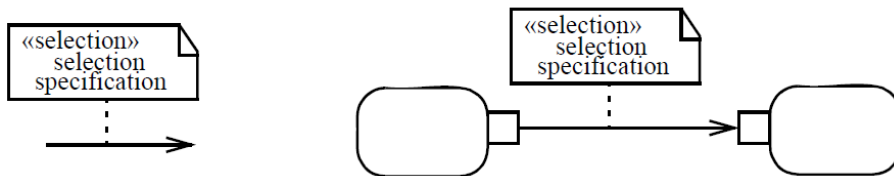
An object flow is notated by an arrowed line. In Figure 15.9, upper right, the two object flow arrows denote a single object flow edge between two pins in the underlying model, as shown in the lower middle of the figure. (See other Pin notations in sub clause 16.2. The specific notational variant used shall be preserved when the diagram is interchanged, see Annex B.)



**Figure 15.9 ObjectFlow notations**

A selection Behavior is specified with the keyword «selection» placed in a note symbol and attached to the appropriate ObjectFlow symbol as illustrated in the figure below. A transformation Behavior is similarly specified using the keyword «transformation».

The body of the note symbol may either contain a textual representation of the Behavior (e.g., the body of an OpaqueBehavior) or the name of a Behavior that is not represented textually.



**Figure 15.10 Specifying selection behavior on an ObjectFlow**

To reduce clutter in complex diagrams, Pins may be elided. The names of the Actions can suggest their Pins. Tools may support hyperlinking from the ObjectFlow lines to show the data flowing along them, and show a small square above the line to indicate that Pins are elided, as illustrated in the figure below. Any adornments that would normally be near the Pin, like effect, can be displayed at the ends of the flow lines.

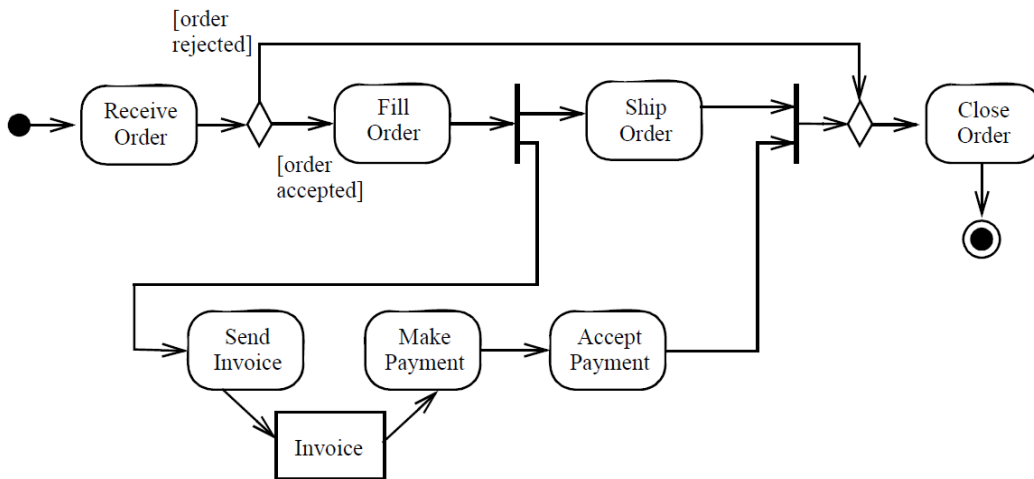


**Figure 15.11 Eliding objects flowing on the edge**

Multicast and multireceive are specified by annotating an ObjectFlow with «multicast» or «multireceive», respectively, see sub clause 15.2.5 for examples.

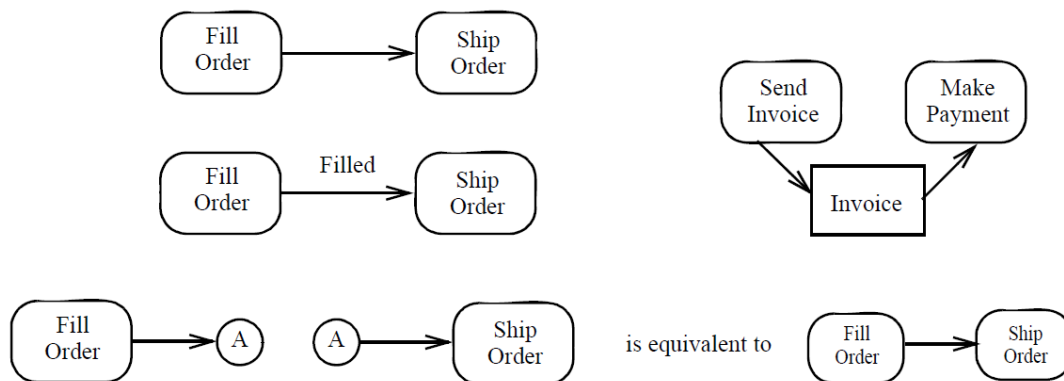
## 15.2.5 Examples

Figure 15.12 illustrates the following kinds of ActivityNodes: ExecutableNodes (e.g., Receive Order, Fill Order), ObjectNodes (Invoice), and ControlNodes (the InitialNode before Receive Order, the DecisionNode after Receive Order, and the ForkNode and JoinNode around Ship Order, the MergeNode before Close Order and the ActivityFinalNode after Close Order).



**Figure 15.12** Activity node example (where the arrowed lines are the only non-activity node symbols)

In Figure 15.13, the arrowed line connecting Fill Order to Ship Order is a ControlFlow edge. This means that when the Fill Order behavior is completed, control is passed to the Ship Order. Below it, the same ControlFlow is shown with an edge name. The one at the bottom left employs connectors, instead of a continuous line. On the upper right, the arrowed lines starting from Send Invoice and ending at Make Payment (with the Invoice object node in between) are ObjectFlow edges (at least in notation, see discussion of Figure 15.14). This indicates that the flow of Invoice objects goes from Send Invoice to Make Payment.



**Figure 15.13** ActivityEdge examples

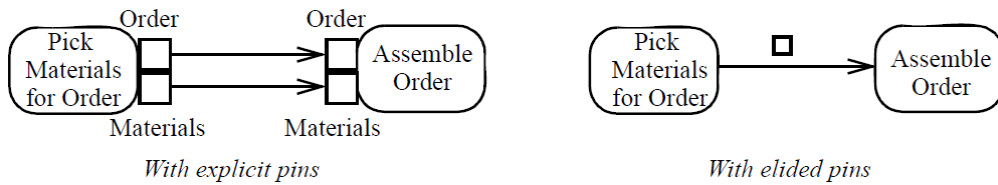
Both examples in Figure 15.14 indicate that order objects flow from Fill Order to Ship Order. The example on the left has two arrowed lines, one from Fill Order and the other to Ship Order. The example on the right has one arrowed line starting from a Fill Order OutputPin (an ObjectNode) and ends at a Ship Order InputPin. The underlying model of these examples is the same, with one object flow in the model shown as two arrows on the left, assuming the Order rectangle on the left does not represent a CentralBufferNode (see sub clause 15.4).



**Figure 15.14** ObjectFlow example

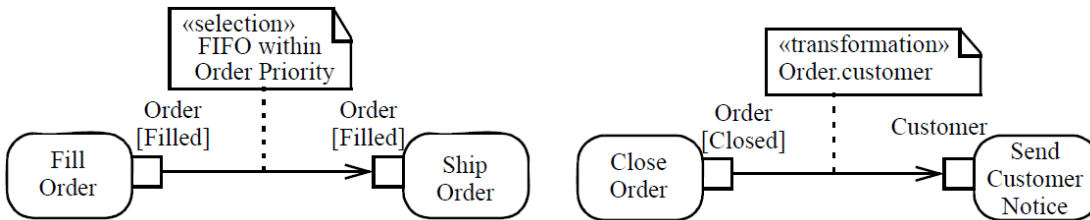
The example on the left in Figure 15.15 shows that the Pick Materials activity provides an order along with its associated materials for assembly. On the right, the ObjectFlow has been simplified through eliding the ObjectFlow details.





**Figure 15.15** Eliding objects flowing on the edge

Figure 15.16 illustrates examples of selection and transformation Behaviors. The example on the left indicates that the orders are to be shipped based on order priority—and those with the same priority should be filled on a first-in/first-out (FIFO) basis. The example on the right indicates that the result of Close Order produces closed order objects, but Send Customer Notice requires a customer object. The transformation specifies the invocation of a query operation that takes an Order and produces the associated customer object.

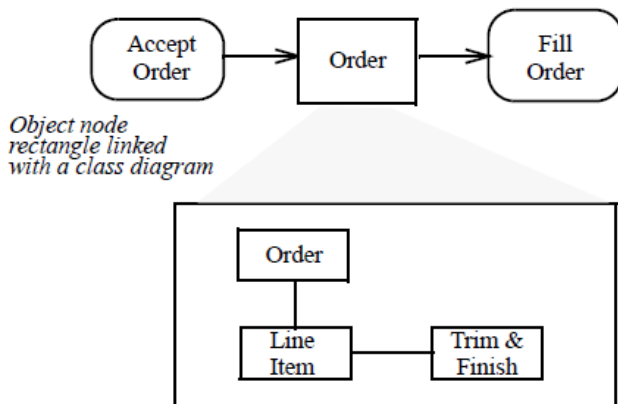


**Figure 15.16** Specifying selection and transformation Behaviors on an ObjectFlow

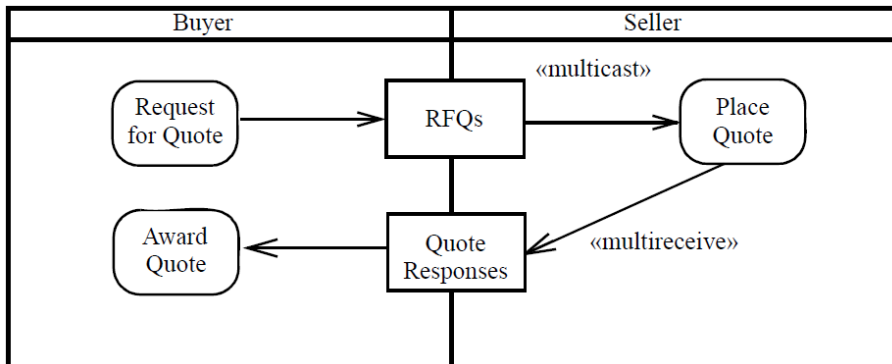
Figure 15.18, the Requests for Quote (RFQs) are sent to multiple specific sellers (i.e., is multicast) for a quote response by each of the sellers. Some number of sellers then respond by returning their quote response. As multiple responses can be received, the edge is labeled for the multiple receive option. Publish/subscribe and other brokered mechanisms can be handled using the multicast and multireceive mechanisms.

**NOTE.** The swimlanes are an important feature for indicating senders and responders.

In Figure 15.17, the object node rectangle Order is linked to a class diagram that further defines the node. The class diagram shows that filling an order requires order, line item, and the customer’s trim-and-finish requirements. An Order token is the object flowing between the Accept and Fill activities, but linked to other objects. The activity without the class diagram provides a simplified view of the process. The link to an associated class diagram is used to show more detail.

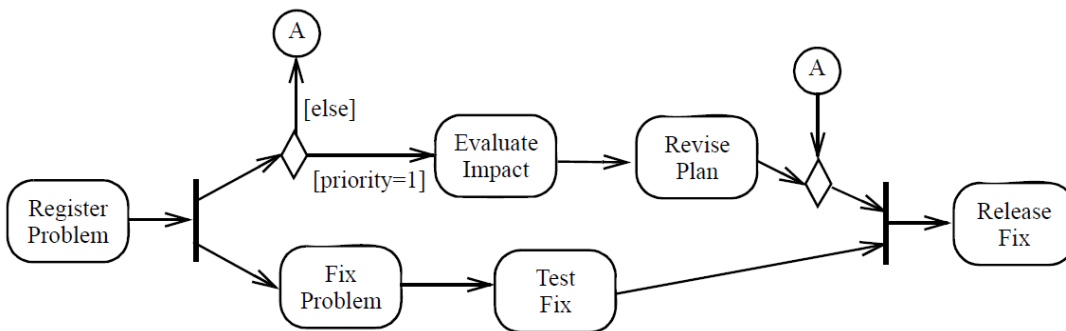


**Figure 15.17** Linking a class diagram to an object node

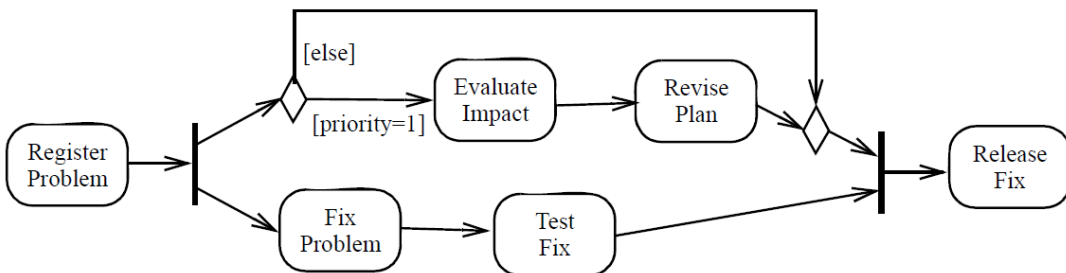


**Figure 15.18** Specifying multicast and multireceive on the edge

In Figure 15.19, a connector is used to avoid drawing a long edge around one tine of the fork. If a problem is not a priority, the token going to the connector is sent to the merge instead of Evaluate Impact. The merge receives its token from Register Problem, via the decision, instead of from Revise Plan for priority one problems. This is equivalent to the activity shown in Figure 15.20, which has the same abstract syntax representation as Figure 15.19.

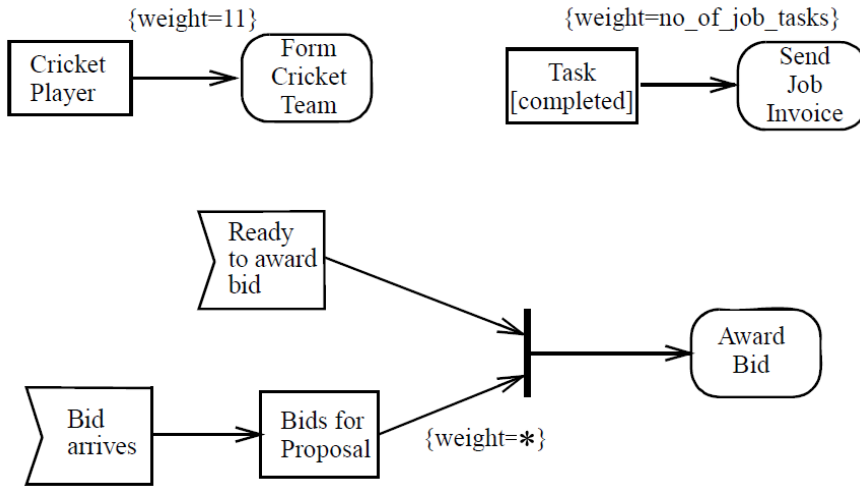


**Figure 15.19** ActivityEdge connector example



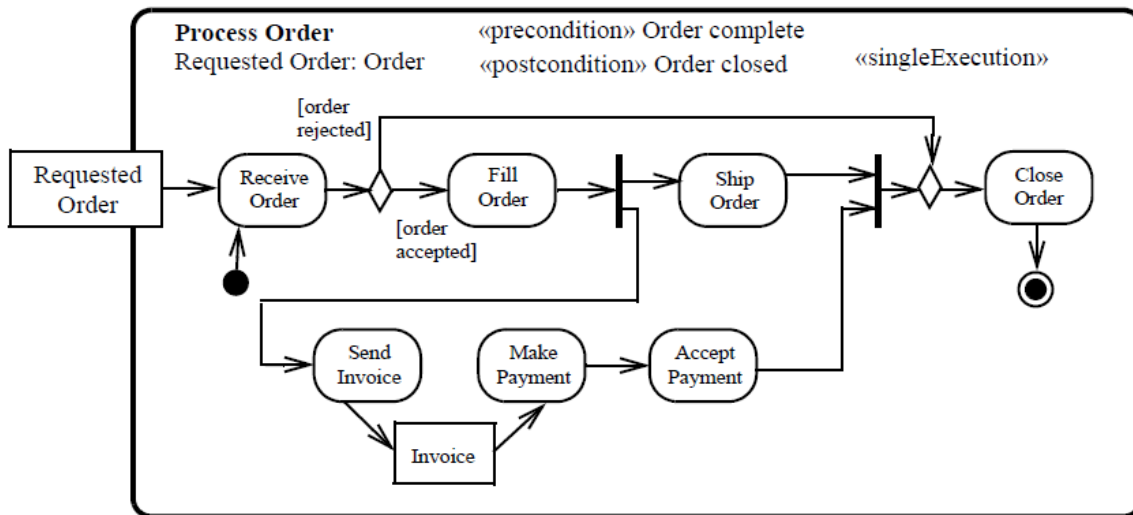
**Figure 15.20** Equivalent model

Figure 15.21 illustrates three examples of using the weight property. The Cricket example uses a constant weight to indicate that a cricket team cannot be formed until eleven players are present. The Task example uses a non-constant weight to indicate that an invoice for a particular job can only be sent when all of its tasks have been completed. The bottom example depicts an Activity for placing bids for a proposal, where many such bids can be placed. When the bidding period is over, the Ready to award bid event arrives, then Award Bid receives all the bids at once, and chooses one for the award.



**Figure 15.21 ActivityEdge weight examples**

The definition of Process Order in Figure 15.22 uses the border notation to indicate that it is an Activity with an ActivityParameterNode corresponding to a single input Parameter. It has pre- and postconditions on the requested order. All invocations of it use the same execution.



**Figure 15.22 Example of an activity with input parameter**

The diagram in Figure 15.23 is a standard part selection workflow within an airline design process. The Standards Engineer ensures that the substeps in Provide Required Part are performed in the order specified and under the conditions specified, but doesn't necessarily perform the steps. Some of the substeps are performed by the Design Engineer even though the Standards Engineer is managing the process. The Expert Part Search behavior can result in a part found or not. When a part is not found, the Assign Standards Engineer behavior is invoked. Lastly, Specify Part Mod Workflow produces values that are instances of Activities representing work to be done. These are passed to subsequent actions for scheduling and execution (i.e., Schedule Part Mod Workflow, Execute Part Mod Workflow and Research Production Possibility). As Activities are Classes, instances of them can be passed in object tokens and then later be executed. This is an example of runtime Activity instantiation and execution.

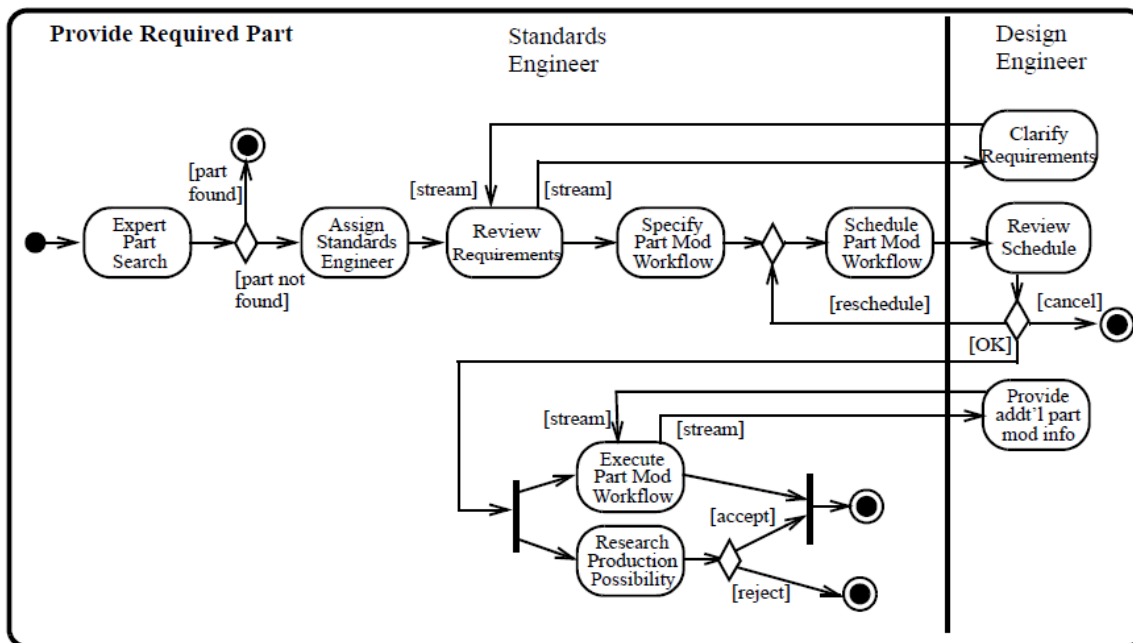
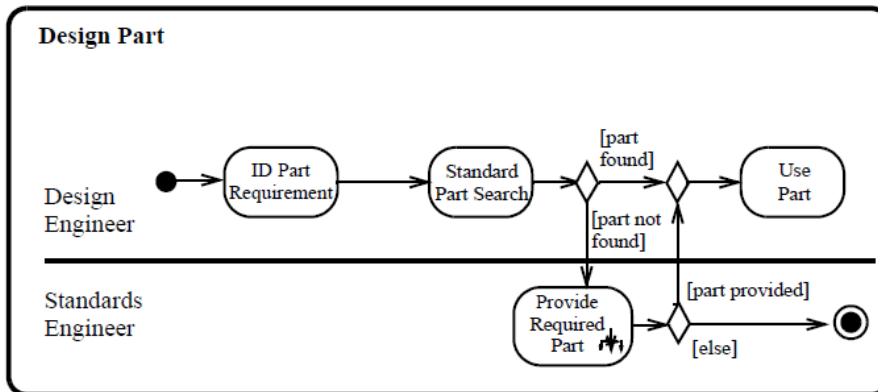


Figure 15.23 Part selection workflow example

Figure 15.24 shows an example activity for a process to resolve a trouble ticket.

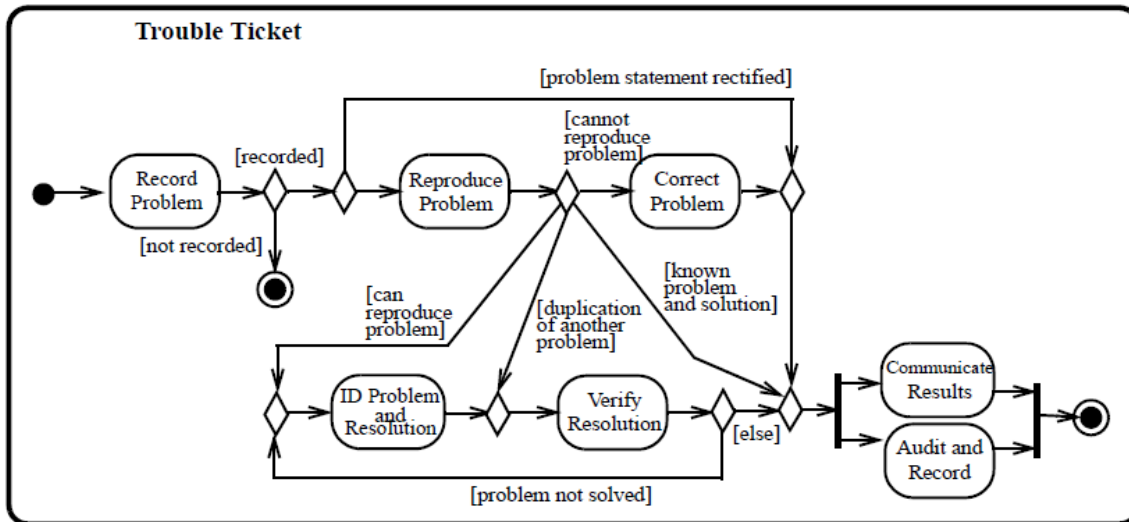


Figure 15.24 Trouble ticket workflow example

«activity» Fill Order
costSoFar : USD timeToComplete : Integer
suspend () resume ()

Figure 15.25 is an example of using class notation to show the class features of an activity.

Figure 15.25 Activity with attributes and operations

## 15.3 Control Nodes

### 15.3.1 Summary

A ControlNode is a kind of ActivityNode (see sub clause 15.2.2) used to manage the flow of tokens between other nodes in an Activity. This subclause describes the various concrete kinds of ControlNodes, including InitialNodes, FinalNodes, ForkNodes, JoinNodes, MergeNodes, and DecisionNodes.

## 15.3.2 Abstract Syntax

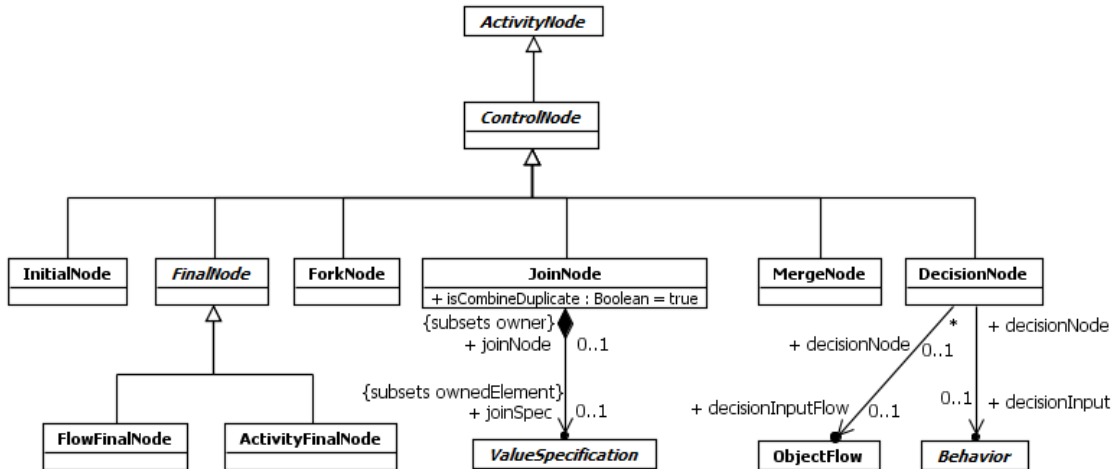


Figure 15.26 Control Nodes

## 15.3.3 Semantics

### Initial Node

An InitialNode is a ControlNode that acts as a starting point for executing an Activity. An Activity may have more than one InitialNode. If an Activity has more than one InitialNode, then invoking the Activity starts multiple concurrent control flows, one for each InitialNode. (Additional concurrent flows may begin at input ActivityParameterNodes and enabled ExecutableNodes; see Subclauses 15.4.3 and 15.5.3.)

An InitialNode shall not have any incoming ActivityEdges, which means the InitialNodes owned by an Activity will always be enabled when the Activity begins execution and a single control token is placed on each such InitialNode when Activity execution starts. The outgoing ActivityEdges of an InitialNode must all be ControlFlows. The control token placed on an InitialNode is offered concurrently on all outgoing ControlFlows.

InitialNodes are an exception to the rule that ControlNodes cannot “hold” tokens, but only manage their flow. If the token offered by an InitialNode is not immediately accepted, or is otherwise blocked from moving downstream (for example by an ActivityEdge guard), then it remains on the InitialNode. (This is semantically equivalent to interposing a CentralBufferNode between the InitialNode and its outgoing edges; see sub clause 15.4.3 on the semantics of CentralBufferNodes.)

### Final Nodes

A FinalNode is a ControlNode at which a flow in an Activity stops. A FinalNode shall not have outgoing ActivityEdges. A FinalNode accepts all tokens offered to it on its incoming ActivityEdges.

There are two kinds of FinalNode:

1. A FlowFinalNode is a FinalNode that terminates a flow. All tokens accepted by a FlowFinalNode are destroyed. This has no effect on other flows in the Activity.
2. An ActivityFinalNode is a FinalNode that stops all flows in an Activity (or StructuredActivityNode, see sub clause 16.11). A token reaching an ActivityFinalNode owned by an Activity terminates the execution of that Activity. If an Activity owns more than one ActivityFinalNode, then the first one to accept a token (if any) terminates the execution of the Activity, including the execution of any other ActivityFinalNodes. The termination of Activity execution shall destroy all tokens held

in any ObjectNodes other than output ActivityParameterNodes and shall terminate the execution of any behaviors synchronously called from the Activity. However, the execution of Behaviors invoked asynchronously from the Activity shall not be affected. Once the execution of the Activity has terminated, the invocation of the Activity completes as described in sub clause 15.2.3.

**NOTE.** If it is not desired to abort all flows in an Activity, use a FlowFinalNode, not an ActivityFinalNode. For example, if the same execution of an Activity is being used for all its invocations (isSingleExecution=true), then multiple flows of tokens will be flowing through that one execution. In this case, it is probably not desired to abort all flows just because one reaches a FinalNode. Using a FlowFinalNode will simply consume the tokens reaching it without aborting other flows. Alternatively, arrange for separate invocations of the Activity to use separate executions (isSingleExecution=false), so tokens from separate invocations will not affect each other.

### Fork Nodes

A ForkNode is a ControlNode that splits a flow into multiple concurrent flows. A ForkNode shall have exactly one incoming ActivityEdge, though it may have multiple outgoing ActivityEdges. If the incoming edge is a ControlFlow, then all outgoing edges shall be ControlFlows and, if the incoming edge is an ObjectFlow, then all outgoing edges shall be ObjectFlows.

Tokens offered to a ForkNode are offered to all outgoing ActivityEdges of the node. If at least one of these offers is accepted, the offered tokens are removed from their original source and the acceptor receives a copy of the tokens. Any other offer that was not accepted on an outgoing edge due to the failure of the target to accept it remains pending from that edge and may be accepted by the target at a later time. These edges effectively accept separate copies of the offered tokens, and offers made to the edges shall stand to their targets in the order in which they were accepted by the edge (first in, first out). This is an exception to the rule that ActivityEdges cannot “hold” tokens if they are blocked from moving downstream. The ActivityEdges going out of ForkNodes continue to hold the tokens they accept until all pending offers have been accepted by their targets.

**NOTE.** Any outgoing ActivityEdges that fail to accept an offer due to the failure of their guard, rather than their target, shall not receive copies of those tokens.

**NOTE.** If guards are used on ActivityEdges outgoing from a ForkNode, the modeler should ensure that no downstream JoinNodes depend on the arrival of tokens passing through the guarded edge. If that cannot be avoided, then a DecisionNode should be introduced between the ForkNode and the edge with the guard, such that tokens may be shunted to the downstream JoinNode if the guard fails. (See also the example in Figure 15.20.)

### Join Nodes

A JoinNode is a ControlNode that synchronizes multiple flows. A JoinNode shall have exactly one outgoing ActivityEdge but may have multiple incoming ActivityEdges. If any of the incoming edges of a JoinNode are ObjectFlows, the outgoing edge shall be an ObjectFlow. Otherwise the outgoing edge shall be a ControlFlow.

Join nodes may have a joinSpec, which is a ValueSpecification that determines the condition under which the join will emit a token. If a JoinNode has a joinSpec, then this ValueSpecification is evaluated whenever a new token is offered to the JoinNode on any incoming ActivityEdge. This evaluation shall not be interrupted by any new tokens offered during the evaluation, nor shall concurrent evaluations be started when new tokens are offered during an evaluation. The ValueSpecification shall evaluate to a Boolean value.

If the joinSpec ValueSpecification is given by a textual expression, then the names of the incoming edges may be used to denote a Boolean value indicating the presence (true) or absence (false) of an offer from a ControlFlow or to denote the value associated with an object token offered from an ObjectFlow (if any). Alternatively, the joinSpec may consist of an Expression with the name of a single Boolean operator and no operands specified. In this case, the value of the joinSpec shall be given by applying the given operator to Boolean values indicating the presence (true) or absence (false) of offers on each incoming edge (with the ordering of the operands not specified).

If a JoinNode does not have a joinSpec, then this is equivalent to a joinSpec Expression with the Boolean operator “and.” That is, the implicit default joinSpec condition is that there is at least one token offered on each incoming ActivityEdge.

If the (implicit or explicit) `joinSpec` of a `JoinNode` evaluates to true, then tokens are offered on the outgoing `ActivityEdge` of the `JoinNode` according to the following rules:

1. If all the tokens offered on the incoming edges are control tokens, then one control token is offered on the outgoing edge.
2. If some of the tokens offered on the incoming edges are control tokens and others are object tokens, then only the object tokens are offered on the outgoing edge. Tokens are offered on the outgoing edge in the same order they were offered to the join. If `isCombinedDuplicate` is true for the `JoinNode`, then before object tokens are offered to the outgoing edge, those containing objects with the same identity are combined into one token.

The above rules apply to all tokens offered to the `JoinNode`, including multiple tokens offered from the same incoming edge.

If any tokens are offered to the outgoing `ActivityEdge` of a `JoinNode`, they shall be accepted by the target or rejected for traversal over the edge (e.g., due to a failed guard) before any more tokens are offered to the outgoing edge. If tokens are rejected for traversal, they shall no longer be offered to the outgoing edge. A conforming implementation may omit unnecessary `joinSpec` evaluations if the `JoinNode` is blocked from offering tokens on its outgoing edge.

## Merge Nodes

A `MergeNode` is a control node that brings together multiple flows without synchronization. A `MergeNode` shall have exactly one outgoing `ActivityEdge` but may have multiple incoming `ActivityEdges`. If the outgoing edge of a `MergeNode` is a `ControlFlow`, then all incoming edges must be `ControlFlows`, and, if the outgoing edge is an `ObjectFlow`, then all incoming edges must be `ObjectFlows`.

All tokens offered on the incoming edges of a `MergeNode` are offered to the outgoing edge. There is no synchronization of flows or joining of tokens.

## Decision Nodes

A `DecisionNode` is a `ControlNode` that chooses between outgoing flows. A `DecisionNode` shall have at least one and at most two incoming `ActivityEdges`, and at least one outgoing `ActivityEdge`. If it has two incoming edges, then one shall be identified as the `decisionInputFlow`, the other being called the *primary incoming edge*. If the `DecisionNode` has only one incoming edge, then it is the primary incoming edge. If the primary incoming edge of a `DecisionNode` is a `ControlFlow`, then all outgoing edges shall be `ControlFlows` and, if the primary incoming edge is an `ObjectFlow`, then all outgoing edges shall be `ObjectFlows`.

A `DecisionNode` accepts tokens on its primary incoming edge and offers them to all its outgoing edges. However, each token offered on the primary incoming edge shall traverse at most one outgoing edge. Tokens are not duplicated.

If any of the outgoing edges of a `DecisionNode` have guards, then these are evaluated for each incoming token. The order in which guards are evaluated is not defined and may be evaluated concurrently. If the primary incoming edge of a `DecisionNode` is an `ObjectFlow`, and the `DecisionNode` does not have a `decisionInputBehavior` or `decisionInputFlow`, then the value contained in an incoming object token may be used in the evaluation of the guards on outgoing `ObjectFlows`.

If a `DecisionNode` has a `decisionInputFlow`, then a token must be offered on both the primary incoming edge and the `decisionInputFlow` before the token from the primary incoming edge is offered to the outgoing edges. If the `DecisionNode` does not have a `decisionInputBehavior`, then the value contained in the object token on the `decisionInputFlow` is made available to the guards on each outgoing edge, regardless of whether the primary incoming flow is a `ControlFlow` or an `ObjectFlow`.

If a `DecisionNode` has a `decisionInputBehavior`, then this must be a `Behavior` with a return `Parameter` and no other output `Parameters`. This `Behavior` is invoked for each incoming (control or object) token, and the result returned from the `Behavior` is available in the evaluation of the guards on outgoing edges. A `decisionInputBehavior` shall not have side effects. It shall not modify objects, but it may, for example, navigate from one object to another to get an attribute value from an object.

If the primary incoming edge of a `DecisionNode` is a `ControlFlow`, and the `DecisionNode` has a `decisionInputBehavior` but not a `decisionInputFlow`, then the `decisionInputBehavior` shall have no input `Parameters`. However, if the `DecisionNode` has both a `decisionInputBehavior` and a `decisionInputFlow`, then the `decisionInputBehavior` shall have a single in `Parameter`, and the value contained in the object token offered on the `decisionInputFlow` shall be passed via this `Parameter` when the `Behavior` is invoked.



If the primary incoming edge of a `DecisionNode` is an `ObjectFlow`, and the `DecisionNode` has a `decisionInputBehavior` but not a `decisionInputFlow`, then the `decisionInputBehavior` shall have an `in` `Parameter` and the value contained in an object token offered on the primary incoming edge is passed via this `Parameter` when the `Behavior` is invoked for the token. However, if the `DecisionNode` has both a `decisionInputFlow` and a `decisionInputBehavior`, then the `decisionInputBehavior` shall have two `in` `Parameters`, with the value contained in the object token offered on the primary incoming edge passed via the first `Parameter` and the value contained in the object token offered on the `decisionInputFlow` passed via the second `Parameter` when the `Behavior` is invoked.

A token offered on the primary incoming edge of a `DecisionNode` shall not traverse any outgoing edge for which the guard evaluates to false. If there are multiple outgoing edges that either have no guard or that have a guard that evaluates to true, then the incoming token shall traverse at most one of these edges. If exactly one target of an unblocked outgoing edge accepts the token, then the token traverses the corresponding edge and all other offers are withdrawn. If multiple targets accept the token simultaneously, then the token traverses only one of the edges corresponding to the accepting targets, but which one is not determined by this specification.

In order to avoid non-deterministic behavior, the modeler should arrange that at most one guard evaluate to true for each incoming token. If it can be ensured that only one guard will evaluate to true, a conforming implementation is not required to evaluate the guards on all outgoing edges once one has been found to evaluate to true.

For use only with `DecisionNodes`, a predefined guard “else” (represented as an `Expression` with “else” as its operator and no operands) may be used for at most one outgoing edge. This guard evaluates to true only if the token is not accepted by any other outgoing edge from the `DecisionNode`.

### 15.3.4 Notation

#### Initial and Final Nodes

`InitialNodes` are notated as a solid circle, as shown in Figure 15.27.



Figure 15.27 `InitialNode` notation

`ActivityFinalNodes` are notated as a solid circle within a hollow circle, as shown in Figure 15.28. This can be thought of as a goal notated as “bull’s eye,” or target. `FlowFinalNodes` are notated as a circle with an “X” cross inside it.



*Activity final*

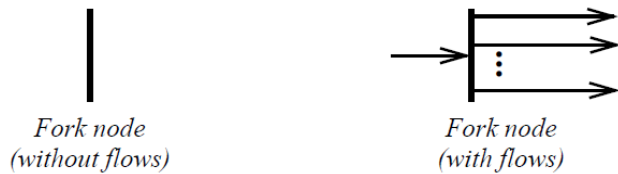


*Flow final*

Figure 15.28 `FinalNode` notation

#### Fork and Join Nodes

The notation for both `ForkNodes` and `JoinNodes` is simply a line segment, as illustrated on the left side of Figure 15.29 (not necessarily in that orientation). When used, however, a `ForkNode` must have a single incoming `ActivityEdge` and usually has two or more outgoing `ActivityEdges`, while a `JoinNode` usually has two or more incoming `ActivityEdges` and must have a single outgoing `ActivityEdge`.

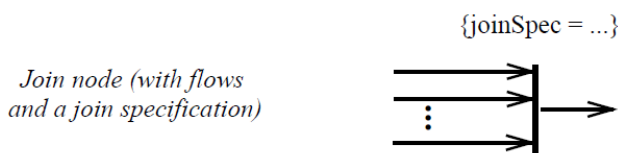


**Figure 15.29 ForkNode and JoinNode notation**

A joinSpec on a JoinNode is shown in an annotation near the JoinNode symbol (see Figure 15.30):

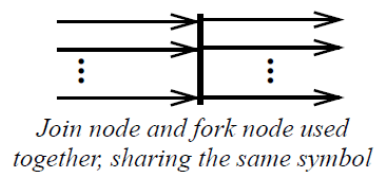
*join-spec-annotation ::= '{ 'joinSpec' '=' value-specification '}'*

See also Clause 8 on the notation for ValueSpecifications.



**Figure 15.30 joinSpec notation**

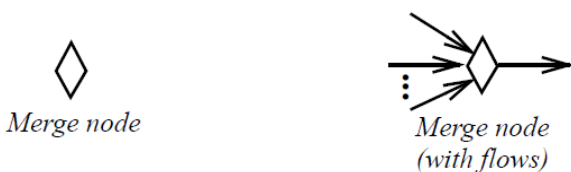
The functionality of a JoinNode and a ForkNode can be combined by using the same node symbol, as illustrated in Figure 15.31. This notation maps to a model containing a JoinNode with all the incoming ActivityEdges shown in the diagram and one outgoing ActivityEdge to a ForkNode that has all the outgoing ActivityEdges shown in the diagram.



**Figure 15.31 Combined JoinNode/ ForkNode notation**

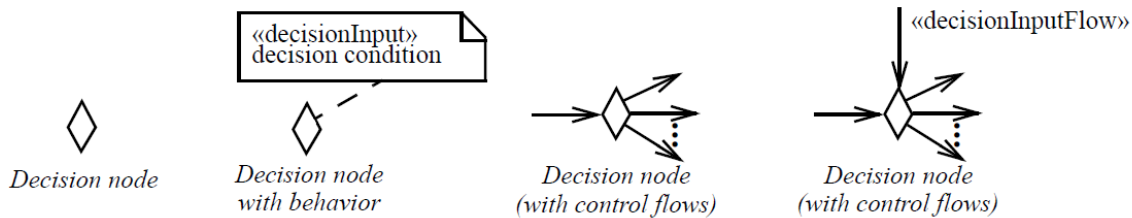
### Merge Nodes and Decision Nodes

The notation for both MergeNodes and DecisionNodes is a diamond-shaped symbol, as shown on the left side of Figure 15.32. When used, however, a MergeNode must have two or more incoming ActivityEdges and a single outgoing ActivityEdge, while a DecisionNode must have a single incoming ActivityEdge (other than a possible decisionInputFlow) and multiple outgoing ActivityEdges.



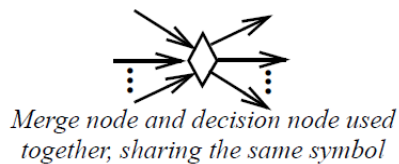
**Figure 15.32 MergeNode notation**

A decisionInputBehavior on a DecisionNode is notated in a note symbol attached to the DecisionNode symbol, with the keyword «decisionInput», as shown in Figure 15.33. A decisionInputFlow is identified by the keyword «decisionInputFlow» annotating that flow.



**Figure 15.33 DecisionNode notation**

The functionality of a MergeNode and a DecisionNode can be combined by using the same node symbol, as shown in Figure 15.34. At most one of the incoming flows may be annotated as a decisionInputFlow. This notation maps to a model containing a MergeNode with all the incoming edges shown in the diagram and one outgoing edge to a DecisionNode that has all the outgoing edges shown in the diagram.

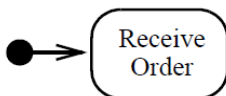


**Figure 15.34 Combined MergeNode/DecisionNode notation**

## 15.3.5 Examples

### Initial Nodes

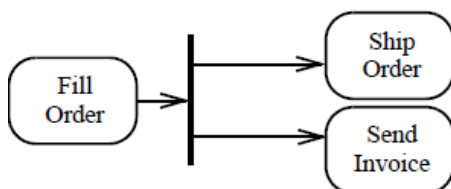
In Figure 15.35, the InitialNode passes control to the Receive Order ExecutableNode at the start of the execution of an Activity.



**Figure 15.35 InitialNode example**

### Fork and Join Nodes

In Figure 15.36, the ForkNode passes control to both the Ship Order and Send Invoice nodes when Fill Order is completed.



**Figure 15.36 ForkNode example**

In Figure 15.37, a JoinNode is used to synchronize the processing of the Ship Order and Send Invoice nodes. Here, when both have been completed, control is passed to Close Order.

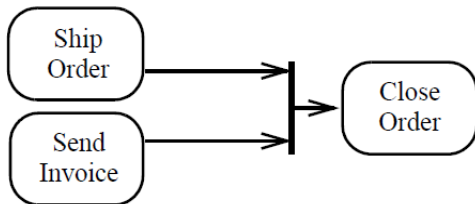


Figure 15.37 JoinNode example

Figure 15.38 illustrates how a joinSpec can be used to ensure that both a drink is selected and the correct amount of money has been inserted before the drink is dispensed. Names of the incoming edges are used in the join specification to refer to whether tokens are available on the edges.

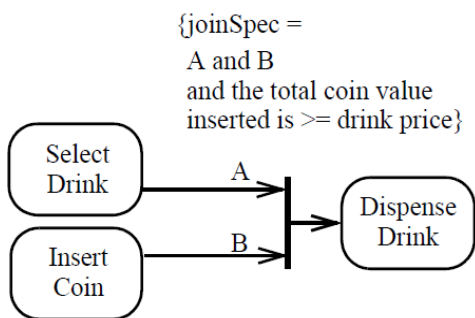


Figure 15.38 joinSpec example

### Merge and Decision Nodes

In Figure 15.39, either one or both of the nodes Buy Item or Make Item could have been executed. As *each* completes, control is passed to Ship Item. That is, if only one of Buy Item or Make Item completes, then Ship Item is executed only once; if both complete, Ship Item is executed twice.

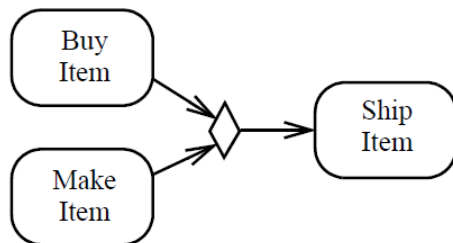


Figure 15.39 MergeNode example

Figure 15.40 contains a DecisionNode that follows the Received Order node. The branching is based on whether order was rejected or accepted. An order accepted condition results in passing control to Fill Order and an order rejected condition results in passing control to Close Order.

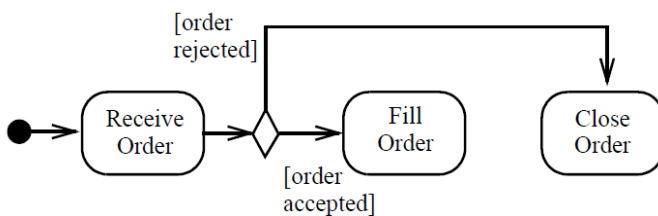


Figure 15.40 DecisionNode example

Figure 15.41 illustrates an order process example. Here, an order item is pulled from stock and prepared for delivery. As the item has been removed from inventory, the reorder level should also be checked; and if the actual level falls below a pre-specified reorder point, more of the same type of item should be reordered.

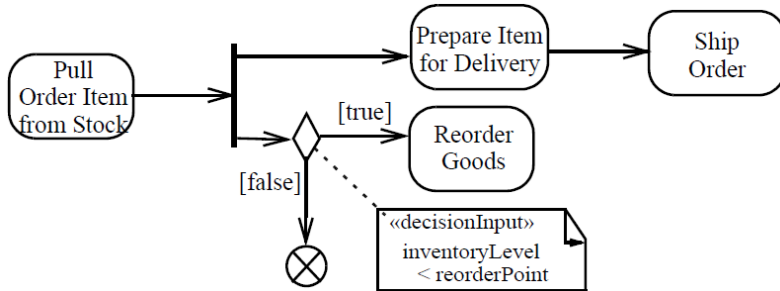


Figure 15.41 DecisionNode example with decisionInput

### Final Nodes

Figure 15.42 depicts that, when the Close Order node is completed, the Activity is terminated. This is indicated by passing control to an Activity FinalNode.

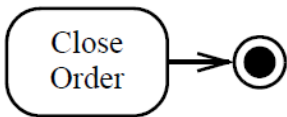


Figure 15.42 ActivityFinalNode example

Figure 15.43 is based on an example for an employee expense reimbursement process, illustrating two concurrent flows racing to complete. The first one to reach the ActivityFinalNode aborts the other. The two flows appear in the same Activity so they can share data, such as who to notify in the case of no action. A ForkNode is used to split the initial flow from the InitialNode and begin the two concurrent flows.

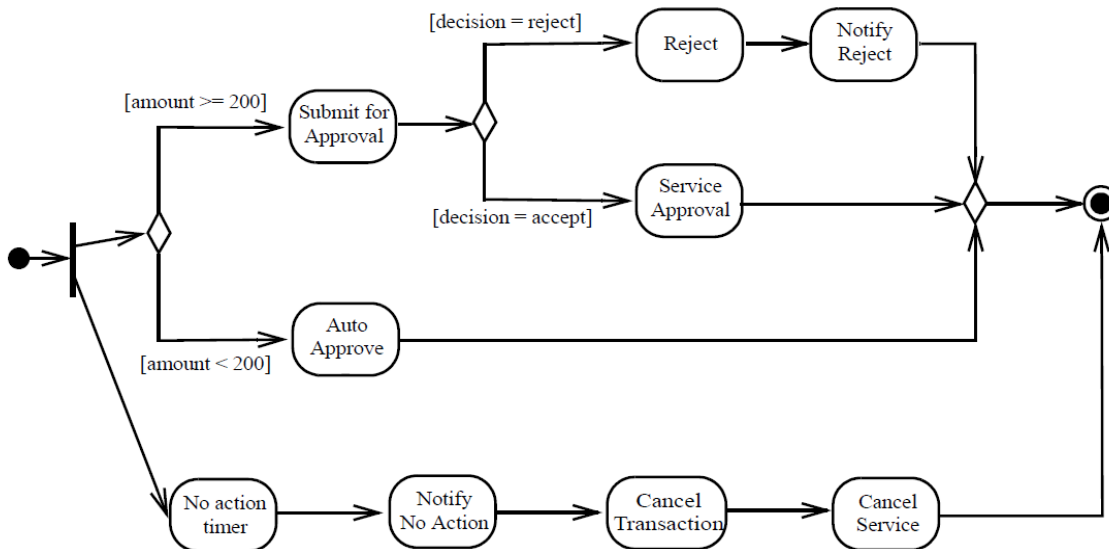
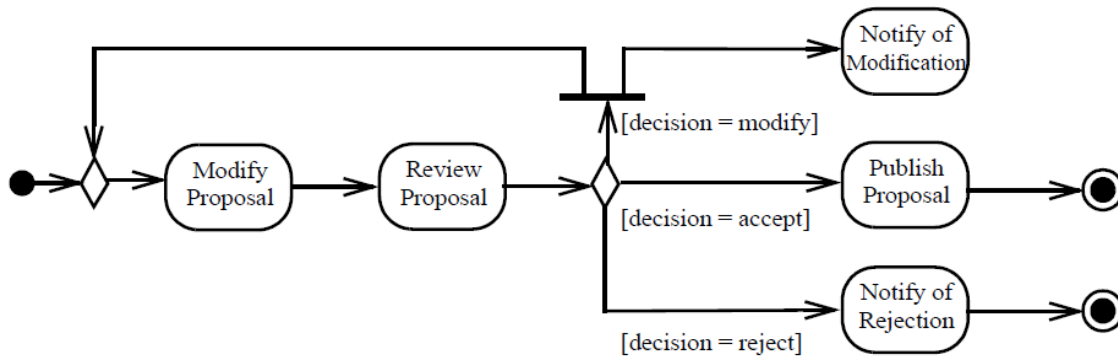


Figure 15.43 ActivityFinalNode example

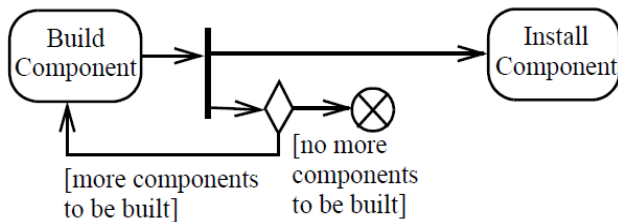
In Figure 15.44, two ways to reach an ActivityFinalNode exist; but it is the result of exclusive “or” branching, not a “race” situation like the example in Figure 15.43. This example uses two Activity FinalNodes, which has the same semantics as using one with two incoming edges.

**NOTE.** Execution of the Notify of Modification node must not take longer than Publish Proposal or Notify of Rejection, or an ActivityFinalNode might kill it.



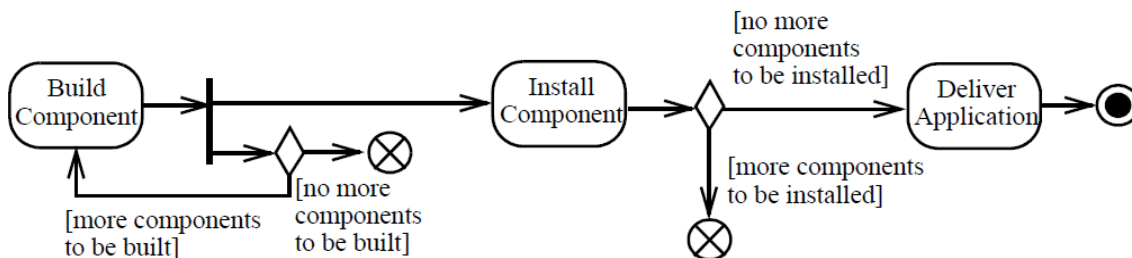
**Figure 15.44 ActivityFinalNode example**

In Figure 15.45, it is assumed that many components can be built and installed. Here, the Build Component node executes iteratively for each component. When the last component is built, the end of the building iteration is indicated with a FlowFinalNode. However, even though all component building has come to an end, other nodes are still executing (such as Install Component).



**Figure 15.45 FlowFinalNode example**

Figure 15.46 illustrates both kinds of FinalNode: FlowFinalNode and ActivityFinalNode. This example extends the model shown in Figure 15.45. In the extended model, when the last component has been installed, the application is delivered. When Deliver Application has completed, control is passed to an ActivityFinalNode—indicating that all processing in the Activity is terminated.



**Figure 15.46 FlowFinalNode and ActivityFinalNode example**

## Various Control Nodes

Figure 15.47 contains examples of various kinds of ControlNodes. An InitialNode is depicted in the upper left as triggering the Receive Order node. A DecisionNode after Received Order illustrates branching based on order rejected or order accepted conditions. Fill Order is followed by a ForkNode that passes control both to Send Invoice and Ship Order. The JoinNode indicates that control will be passed to the MergeNode when both Ship Order and Accept Payment are completed. As a MergeNode will just pass the token along, the Close Order node will be executed. (Control is also passed to Close Order whenever an order is rejected.) When Close Order is completed, control passes to an ActivityFinalNode.

The JoinNode indicates that control will be passed to the MergeNode when both Ship Order and Accept Payment are completed. As a MergeNode will just pass the token along, the Close Order node will be executed. (Control is also passed to Close Order whenever an order is rejected.) When Close Order is completed, control passes to an ActivityFinalNode.

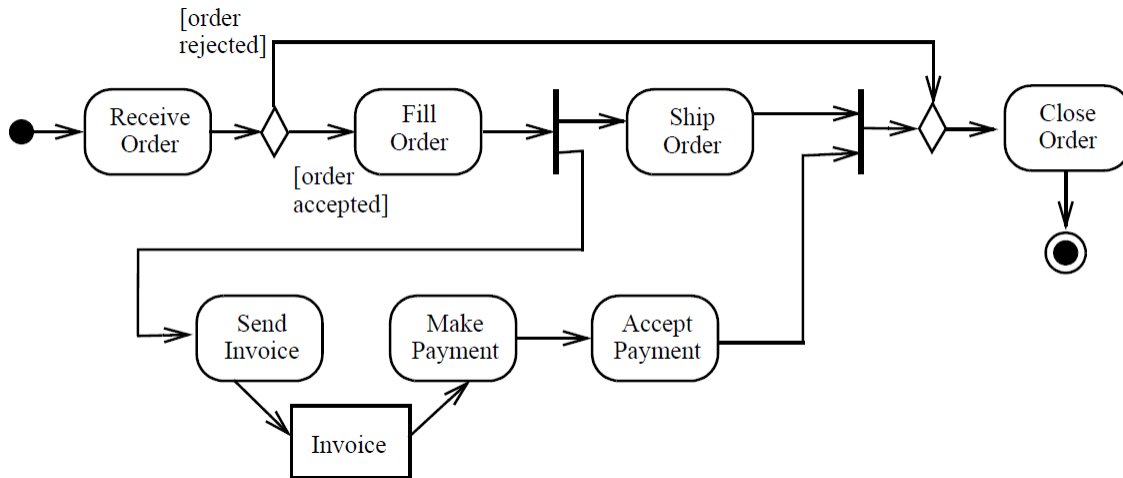


Figure 15.47 ControlNode examples (with accompanying actions and control flows)

## 15.4 Object Nodes

### 15.4.1 Summary

An ObjectNode is a kind of ActivityNode (see sub clause 15.2.2) used to hold value-containing object tokens during the course of the execution of an Activity. This subclause describes ObjectNodes in general, as well as three concrete kinds of ObjectNodes: ActivityParameterNodes, CentralBufferNodes and DataStoreNodes. A fourth kind of ObjectNode, Pins, are always associated with Actions and are described in Clause 16 on Actions (see also sub clause 15.2).

## 15.4.2 Abstract Syntax

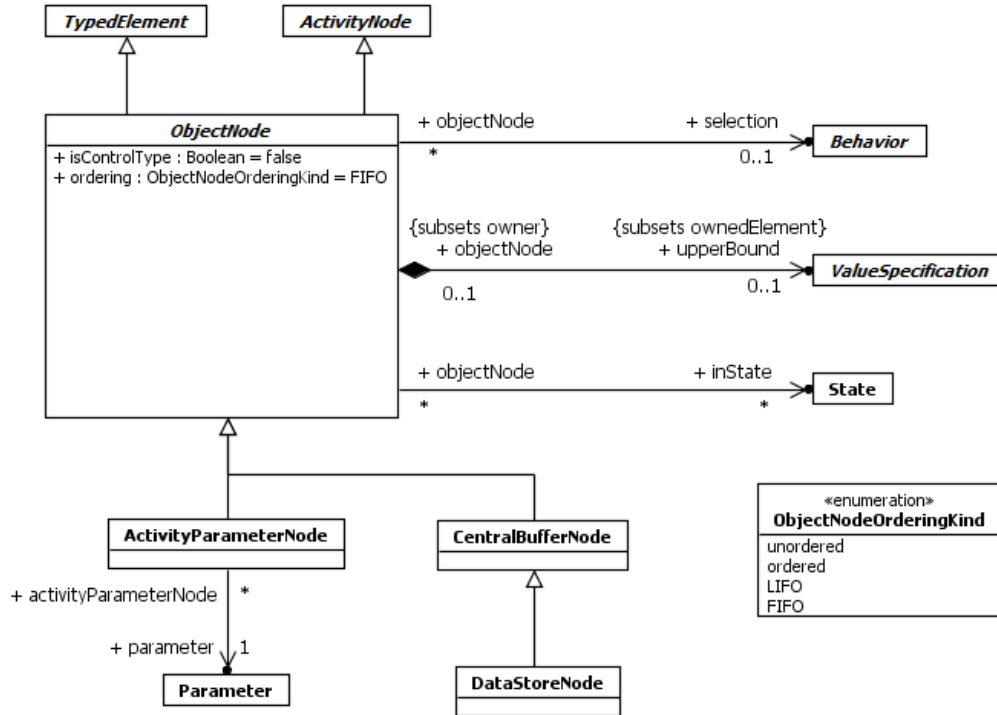


Figure 15.48 Object Nodes

## 15.4.3 Semantics

### Object Nodes

An ObjectNode holds object tokens during the course of the execution of an Activity. Except in the case of an input ActivityParameterNode (as discussed further below), the tokens held by an ObjectNode arrive from incoming ActivityEdges. Except in the case of an output ActivityParameterNode, tokens held by an ObjectNode may leave the node on outgoing ActivityEdges. A token may traverse only one of the outgoing edges. (See the discussion of Object Flows in sub clause 15.2.3, and below, for additional rules regarding when tokens may traverse the edges incoming to and outgoing from an ObjectNode.)

An ObjectNode may contain multiple object tokens with the same value. Such tokens are not normally combined (but see the special semantics for DataStoreNodes below).

ObjectNodes are TypedElements (see sub clause 7.3). If an ObjectNode has a type specified, then any object tokens held by the ObjectNode shall have values that conform to the type of the ObjectNode. If no type is specified, then the values may be of any type. Null tokens (object tokens without a value) satisfy the type of all object nodes.

ObjectNodes may also specify an inState set of States. If such a set is specified, then any object token held by the ObjectNode shall have a value with a type that has or inherits a StateMachine as its classifierBehavior that has all of the states in the inState set, and whose instance for the given value shall be in a state configuration containing all of the States specified in the inState set (see Clause 14 on the semantics of StateMachines).

An ObjectNode may not contain more tokens than specified by its upperBound, if any. If an ObjectNode has an upperBound, then this ValueSpecification shall evaluate to an UnlimitedNatural value. The upperBound is evaluated each time a token is offered to or removed from the ObjectNode. If the number of tokens already held by the ObjectNode is greater than or equal to the evaluated



upperBound, then the ObjectNode shall not accept any further tokens until some of the ones it is holding are removed. If the removal of one or more tokens brings the number of tokens held below the evaluated upperBound, then the ObjectNode may accept any pending offers up to the limit of the upperBound. If the upperBound evaluates to \*, then there is no limit on the number of tokens the ObjectNode may hold.

The ordering of an ObjectNode specifies the order in which tokens held by the node are offered to its outgoing ActivityEdges. This property has one of the following values:

- *unordered* – The order in which tokens held by the ObjectNode are offered to outgoing edges is not defined.
- *FIFO* – Tokens held by the ObjectNode are offered to outgoing edges in the order in which they were accepted by the ObjectNode (i.e., “First In First Out”). Tokens do not overtake each other as they pass through the node.
- *LIFO* – Tokens held by the ObjectNode are offered to outgoing edges in the reverse order to which they were accepted by the ObjectNode (i.e., “Last In First Out”).
- *ordered* – The ordering is modeler-defined using a selection Behavior (see below).

An ObjectNode shall have a selection Behavior if and only if ordering=ordered. A selection Behavior shall have one input Parameter and one output Parameter. The input Parameter must have multiplicity 0..\*, not ordered, and non-unique (i.e., a “Bag”), while the output Parameter must have multiplicity 1..1. If the ObjectNode is untyped, then the Parameters shall also be untyped. Otherwise, the input Parameter shall have either the same type as or a supertype of the ObjectNode, while the output Parameter shall have either the same type or a subtype of the ObjectNode.

The selection Behavior of an ObjectNode is executed whenever a token is to be offered to the outgoing edges of the node. The values contained in all the object tokens held by the Object Node are passed as input to the Behavior invocation. The Behavior should choose one of these values and return it. An object token containing this value is then offered to the outgoing edges of the ObjectNode.

The selection Behavior of an object node is overridden by any selection Behaviors on its outgoing ObjectFlows (see the discussion of ObjectFlow selection Behaviors in sub clause 15.2.3).

Note that tokens overtaking each other due to ordering is independent of the case where each invocation of an Activity is handled by a separate execution of the Activity (i.e., isSingleExecution=false). In this case, the tokens in separate executions have no interaction with each other, because they flow through separate executions of the Activity (see the discussion of Activity Execution in sub clause 15.2.3), but the separate executions can have multiple tokens (due to forks or results of executable nodes) that might overtake each other due to ordering.

If isControlType=true for an ObjectNode, ControlFlows may be incoming to and outgoing from the ObjectNode, objects tokens can come into or go out of the ObjectNode along ControlFlows, and these tokens can flow along ControlFlows reached downstream of the ObjectNode. The values on such object tokens may be used to affect the control of ExecutableNodes that are the targets of such ControlFlows, though the specific meaning of such values is not defined in this specification (see sub clause 15.5).

### Activity Parameter Nodes

As a kind of Behavior, an Activity may have Parameters (see sub clause [13.2](#)). When the Activity is invoked, values may be passed into the Activity execution on input Parameters (i.e., those with direction in or inout) and values may be passed out of the Activity execution on output Parameters (i.e., those with direction inout, out or return).

Within an Activity, inputs to and outputs from an Activity are handled using ActivityParameterNodes. Each ActivityParameterNode is associated with one Parameter of the Activity that owns the node. The type of an ActivityParameterNode shall be the same as the type of its associated Parameter.

An ActivityParameterNode shall have either all incoming or all outgoing ActivityEdges. An ActivityParameterNode with outgoing edges is an *input* ActivityParameterNode, while an ActivityParameterNode with incoming edges is an *output*

ActivityParameterNode. (Note that whether an ActivityParameterNode is for input or output is not determined until at least one ActivityEdge is connected to it.)

An Activity shall have one ActivityParameterNode corresponding to each in, out, or return Parameter and two ActivityParameterNodes for each inout Parameter. An in Parameter shall not be associated with an output ActivityParameterNode and an out or return Parameter shall not be associated with an input ActivityParameterNode (though either may be associated with an ActivityParameterNode that does not have any edges connected). An inout Parameter shall be associated with at most one input ActivityParameterNode and at most one output ActivityParameterNode.

If an input ActivityParameterNode is associated with a non-streaming Parameter, then, when the containing Activity is invoked, any values passed on that Parameter are wrapped in object tokens and placed on the ActivityParameterNode at the start of the Activity execution. If the Parameter is ordered, the tokens are placed on the ActivityParameterNode in the order of the values in the Parameter; otherwise the order is undefined. The ActivityParameterNode then offers the tokens to all its outgoing edges.

During the course of the execution of an Activity, object tokens may flow into the output ActivityParameterNodes of the Activity. An output ActivityParameterNode accepts all tokens offered to it, which are then placed onto the node. If an output ActivityParameterNode is associated with a non-streaming Parameter, then, when the execution of the containing Activity completes, the values contained in the object tokens held by the ActivityParameterNode are passed out of the execution on the Parameter. If the Parameter is ordered, then the values are ordered corresponding to the ordering of the tokens for the ActivityParameterNode.

If an input ActivityParameterNode is associated with a streaming Parameter, then, whenever a new value is posted to the Parameter, that value is wrapped in an object token, placed on the ActivityParameterNode and offered to all outgoing edges. If an output ActivityParameterNode is associated with a streaming Parameter, then, whenever a new object token is accepted by the ActivityParameterNode, the token is immediately removed from the ActivityParameterNode and the value it contains is immediately posted to the Parameter. (See also the description of the semantics of streaming Parameters in sub clause [13.2](#).)

### Central Buffer Nodes

A CentralBufferNode acts as a buffer between incoming ObjectFlows and outgoing ObjectFlows. It accepts all object tokens offered to it on all incoming flows, which are then held by the node. Held object tokens are offered to outgoing flows according to the general ordering rules for ObjectNodes. When an offer for a token is accepted by a downstream object node, that token is removed from the CentralBufferNode and moved to the accepting object node, as for any object node.

### Data Store Nodes

A DataStoreNode is a CentralBufferNode that holds its object tokens persistently while its activity is executing.

When an offer for an object token held by a DataStoreNode is accepted by a downstream object node, the offered token is removed from the DataStoreNode, per the usual CentralBufferNode semantics. However, a copy is made of the removed object token, with the same value, and this is immediately placed back onto the DataStoreNode. Thus, the values held by a DataStoreNode appear to persist for the duration of each execution of its containing activity, even as tokens move downstream from the node.

When a DataStoreNode accepts an object token, if that token contains an object with the same identity as an object contained in a token already held by the node, then the duplicate object token shall not be placed on the DataStoreNode. Unlike a regular CentralBufferNode, a DataStoreNode contains objects uniquely.

The selection and transformation Behaviors on outgoing ObjectFlows can be used to get information out of a DataStoreNode as if a query were being performed. For example, the selection Behavior can identify an object to retrieve and the transformation Behavior can get the value of an attribute on that object. Also, while tokens are always offered on the outgoing flows of a DataStoreNode, they can only actually flow when they are accepted by a downstream object node. This can be used to model pull semantics for data flow (for instance, see the example in Figure 15.59).

## 15.4.4 Notation

### Object Nodes

ObjectNodes are notated as rectangles, as shown in Figure 15.49. A name labeling the node is placed inside the symbol, where the name indicates the type of the ObjectNode, or the name and type of the node in the format “name:type.” ObjectNodes whose instances are sets of the “name” type are labeled as such. An ObjectNode with a Signal as its type is shown with the symbol on the right of Figure 15.49.

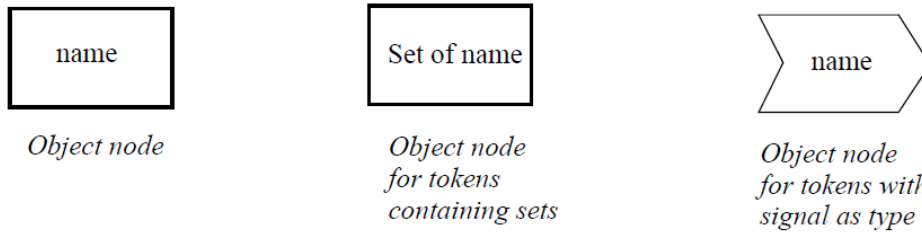


Figure 15.49 ObjectNode notations

If an ObjectNode has an inState set of States, the names of the States in this set are written as a comma-separated list within brackets below the name of the ObjectNode, as shown on the left in Figure 15.50. Values for upperBound, ordering and isControlType are notated by placing an annotation with the following form beneath the ObjectNode symbol (as shown in Figure 15.50):

*object-node-annotation* ::= '{ 'object-node-property' ( ',' object-node-property ) \* '}'

*object-node-property* ::= 'upperBound' '=' value-specification /  
'ordering' '=' object-node-ordering-kind /  
'controlType'

*object-node-ordering-kind* ::= 'unordered' | 'ordered' | 'FIFO' | 'LIFO'

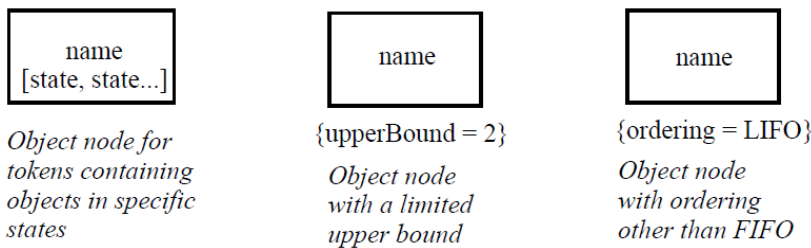


Figure 15.50 ObjectNode annotations

The selection Behavior of an ObjectNode is specified within a note symbol with the keyword «selection», attached to the ObjectNode symbol as illustrated in Figure 15.51.

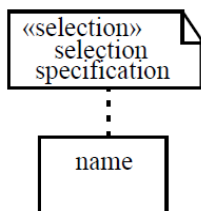


Figure 15.51 Specifying selection behavior on an ObjectNode

## Activity Parameter Nodes

An ActivityParameterNode is notated as an ObjectNode, except that the full textual specification of the associated Parameter (see sub clause 9.4) may be used to label the ActivityParameterNode instead of the normal name/type label. If the containing Activity of the ActivityParameterNode is drawn with a border or frame, then the ActivityParameterNode symbol is drawn overlapping the border or frame (see Figure 15.2). If the Activity is drawn without a border or frame, then the ActivityParameterNode symbol may be placed anywhere on the diagram (though it is clearer if it is placed on the edge of the diagram in a similar location as it would be if a border or frame were present).

Figure 15.52 shows the annotations for an ActivityParameterNode associated with a streaming Parameter or an exception Parameter. If the ActivityParameterNode is associated with a streaming Parameter, then the annotation “{stream}” is placed close to the node symbol. If the ActivityParameterNode is associated with an exception Parameter, then the node symbol is annotated with a small triangle. (See also sub clause 13.2 on the semantics of streaming and exception Parameters.)

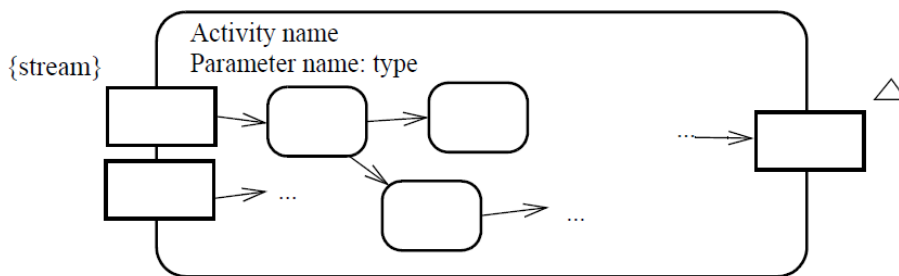


Figure 15.52 Notation for stream and exception parameters

The presentation option at the top of the Activity diagram below may be used as notation for a model corresponding to the notation at the bottom of the diagram.

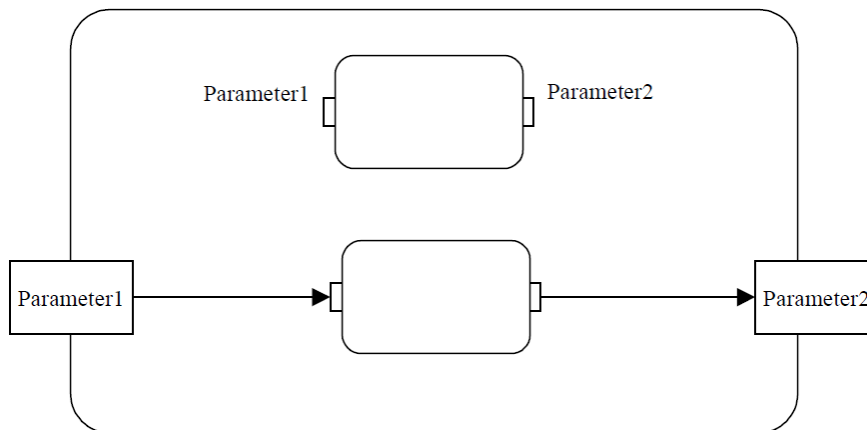


Figure 15.53 Presentation option for flows between pins and parameter nodes

## Central Buffer and Data Store Nodes

A CentralBufferNode symbol may optionally include the keyword «centralBuffer», as shown in Figure 15.54.

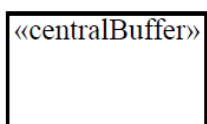


Figure 15.54 Optional CentralBufferNode notation

A DataStoreNode is notated as an ObjectNode with the keyword «datastore», as shown in Figure 15.55.

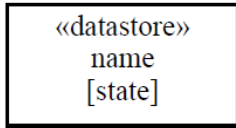


Figure 15.55 DataStoreNode notation

## 15.4.5 Examples

### Activity Parameter Nodes

In Figure 15.56, production materials are fed into printed circuit board. At the end of the Activity, computers are quality checked.

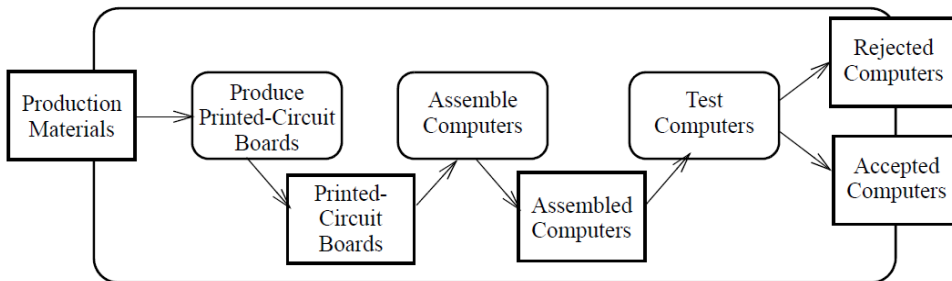


Figure 15.56 Example ActivityParameterNodes

In Figure 15.57, production materials are streaming in to feed the ongoing printed circuit board fabrication. At the end of the Activity, computers are quality checked. Computers that do not pass the test are exceptions.

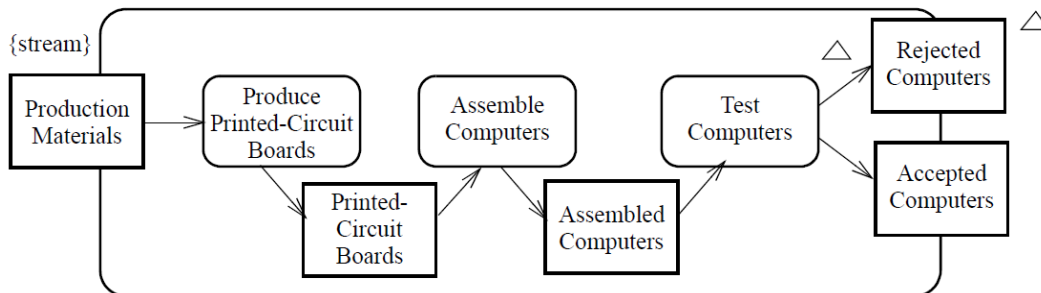


Figure 15.57 Example of ActivityParameterNodes for streaming and exceptions

### Central Buffer and Data Store Nodes

In Figure 15.58, the Behaviors for making parts at two factories produce finished parts. The CentralBufferNode collects the parts, and Behaviors after it in the flow use them as needed. All the parts that are not used will be packed as spares, and vice versa, because each token can only be drawn from the CentralBufferNode by one outgoing edge. The choice in this example is non-deterministic.

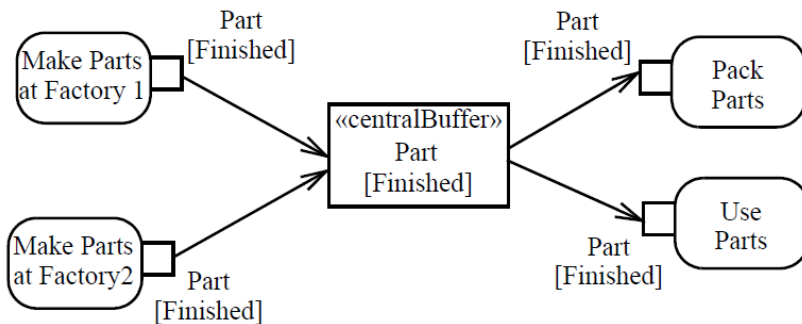


Figure 15.58 CentralBufferNode example

Figure 15.59 is an example of using a DataStoreNode. Records for hired employees are persisted in the Personnel Database. If an employee has no assignment, then one is made using Assign Employee. Once a year, all employees have their performance reviewed. The JoinNode blocks the flow of tokens to Review Employee except when the AcceptEventAction (see sub clause 16.10) is triggered “Once a year”. When the AcceptEventAction generates its yearly control token, this satisfies the join condition on the JoinNode and, as the outgoing edge from the Personnel Database has “{weight=\*}”, object tokens for all the persisted employee records can then flow to Review Employee.

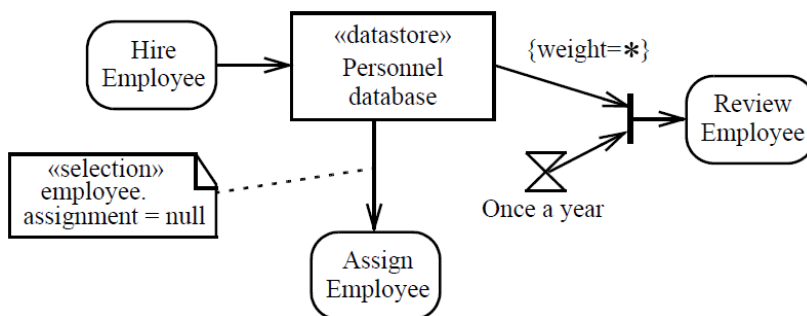


Figure 15.59 DataStoreNode example

## 15.5 Executable Nodes

### 15.5.1 Summary

An ExecutableNode is a kind of ActivityNode that may be executed to produce a step in the overall desired behavior of the containing Activity. Generally, the ControlNodes and ObjectNodes in an Activity are largely there to control the sequencing and to manage the flow of data between the ExecutableNodes of the Activity.

All concrete kinds of ExecutableNodes are Actions, which are described in Clause 16. This subclause discusses the general semantics of ExecutableNodes within an Activity and the capability for any ExecutableNode to have an ExceptionHandler attached to it.

## 15.5.2 Abstract Syntax

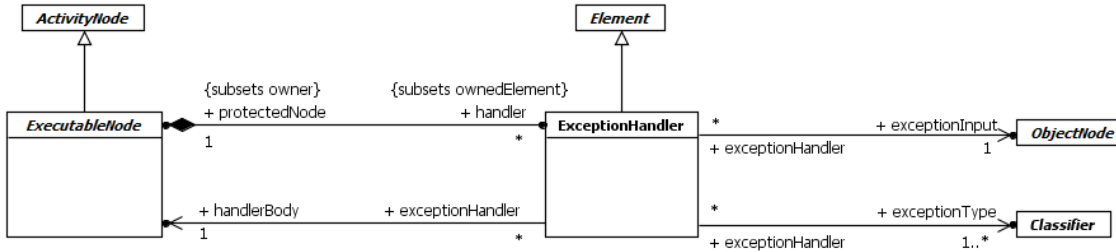


Figure 15.60 Executable Nodes

## 15.5.3 Semantics

### Executable Nodes

An ExecutableNode is an ActivityNode that carries out a substantive behavioral step of the Activity that contains it. All the incoming and outgoing ActivityEdges of an ExecutableNode shall be ControlFlows. An ExecutableNode may also consume and produce data, but it must do so through related ObjectNodes (Actions use Pins for this purpose; see Clause [16](#).)

An ExecutableNode shall not execute until all incoming ControlFlows (if any) are offering tokens. That is, there is an *implicit join* on the incoming Control Flows. Specific kinds of ExecutableNodes may have additional prerequisites that must be satisfied before the node can execute.

Before an ExecutableNode begins executing, it accepts all tokens offered on incoming ControlFlows. If multiple tokens are being offered on a ControlFlow, they are all consumed. The effect of object tokens accepted from ControlFlows is not specified (see isControlType for ObjectNodes in sub clause 15.4), but the semantics above applies if the effect is to execute the ExecutableNode.

While the ExecutableNode is executing, it is considered to hold a single control indicating it is execution. In some cases, multiple concurrent executions of an ExecutableNode may be ongoing at one time (see the semantics of isLocallyReentrant=true for Actions in sub clause [16.2](#)). In this case, the ExecutableNode holds one control token for each concurrent execution.

When an ExecutableNode completes an execution, the control token representing that execution is removed from the ExecutableNode and control tokens are offered on all outgoing ControlFlows of the ExecutableNode. That is, there is an *implicit fork* of the flow of control from the ExecutableNode to its outgoing ControlFlows.

### Exceptions and Exception Handlers

An exception is a value used to identify a non-normal completion mode of an execution. If an exception is raised (e.g., using a RaiseExceptionAction; see sub clause [16.13](#)) within the execution of an ExecutableNode and not handled within that execution, then the execution is terminated and the exception is propagated out of the ExecutableNode.

An ExecutableNode may have one or more ExceptionHandlers that are used to deal with exceptions that may be propagated out of the ExecutableNode, which is the protectedNode of those handlers. If an exception is propagated out of the protectedNode, then the set of handlers is examined for a handler that matches the exception. A handler matches if the type of the exception is the same as, or a (direct or indirect) subtype of, one of the exceptionTypes of the handler. If there is a match, the handler catches the exception. If there are multiple matches, exactly one handler catches the exception, but it is not defined which does.

If an ExceptionHandler catches an exception, the exception is wrapped in an object token that is placed on the exceptionInput ObjectNode for the handler. The handlerBody of the ExceptionHandler is then executed. The execution of the handlerBody may access the caught exception via the exceptionInput node.

When the handlerBody of an ExceptionHandler completes execution after an exception is caught, control tokens are offered on the outgoing ControlFlows of the protectedNode of the ExceptionHandler, in exactly the same way as if the protectedNode completed normally. If the protectedNode is an Action with OutputPins, then the handlerBody must also be an Action with matching OutputPins, and any tokens placed on the OutputPins of the handlerBody are transferred to the OutputPins of the protectedNode (see also sub clause [16.2](#) on Actions and Pins).

A handlerBody shall have no incoming or outgoing ActivityEdges. An ExecutableNode acting as a handlerBody is not enabled to execute in any case other than in response to an exception being caught by its handler.

The handlerBody of an ExceptionHandler shall have the same owner as the protectedNode of the ExceptionHandler and shall own the exceptionInput of the ExceptionHandler. The exceptionInput shall either be untyped or have a type that is either the same as or a (direct or indirect) generalization of all the exceptionTypes of the ExceptionHandler. Typically, the handlerBody will be a StructuredActivityNode and the exceptionInput will be an InputPin for it (see sub clause [16.11](#) on StructuredActivityNodes).

If an ExecutableNode propagates an exception and the node either has no handlers, or no handler matches the propagated exception, then the exception continues to propagate outward. If the exception is not caught at all within the execution of the containing Activity, then the Activity execution terminates and the exception is propagated out of the Activity. If the Activity was invoked synchronously, then the exception is propagated to the caller. If the Activity was invoked asynchronously, then the exception is lost and not propagated further.

## 15.5.4 Notation

### Executable Nodes

An ExecutableNode is drawn generically as a rectangle with rounded corners, as shown in Figure 15.61. More specialized notations for various kinds of Actions are described in Clause [16](#).

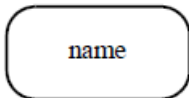


Figure 15.61 ExecutableNode notation

### Exception Handlers

The notation for ExceptionHandlers is illustrated in Figure 15.62. An ExceptionHandler is shown by drawing a “lightning bolt” symbol from the boundary of the protectedNode to a small square on the boundary of the ExceptionHandler. The name of the exceptionType is placed next to the lightning bolt. The small square is the exceptionInput node. Multiple ExceptionHandlers may be attached to the same protectedNode, each by its own lightning bolt.

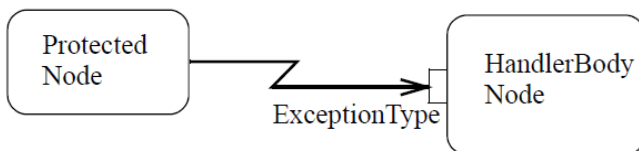


Figure 15.62 ExceptionHandler notation

An option for notating an ExceptionHandler is a “zig-zag” adornment on a straight line, as shown in Figure 15.63.



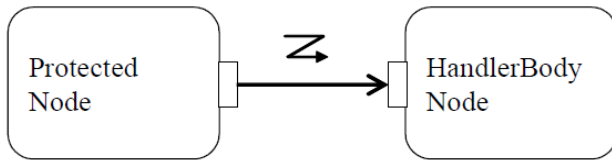


Figure 15.63 Alternative ExceptionHandler notation

## 15.5.5 Examples

Figure 15.64 shows a matrix calculation. First a matrix is inverted, and then it is multiplied by a vector to produce a vector. If the matrix is singular, the inversion will fail and a SingularMatrix exception is raised. This exception is handled by the ExceptionHandler for the exceptionType SingularMatrix, which executes the region containing the Substitute Vector1 Action. If an Overflow exception occurs during either the matrix inversion or the vector multiplication, the region containing the Substitute Vector2 Action is executed. Regardless of whether the matrix operations complete without exception or whether one of the ExceptionHandlers is triggered, the action Print Results is executed next.

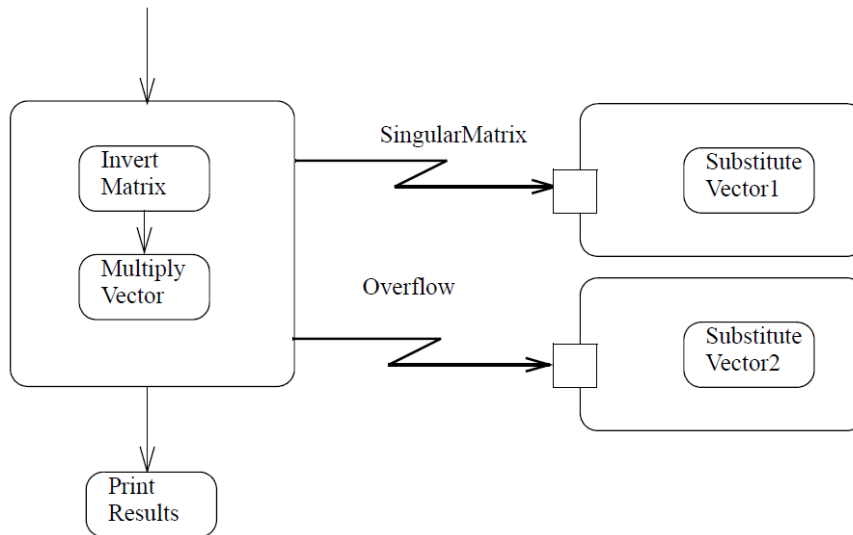


Figure 15.64 ExceptionHandler example

## 15.6 Activity Groups

### 15.6.1 Summary

ActivityGroups are a grouping constructs for ActivityNodes and ActivityEdges. Nodes and edges can belong to more than one group. This subclause describes two concrete kinds of ActivityGroups, ActivityPartitions and InterruptibleActivityRegions. StructuredActivityNodes are a third kind of ActivityGroup, but they are also Actions and are discussed in sub clause [16.11](#) of Clause [16](#) on Actions.

## 15.6.2 Abstract Syntax

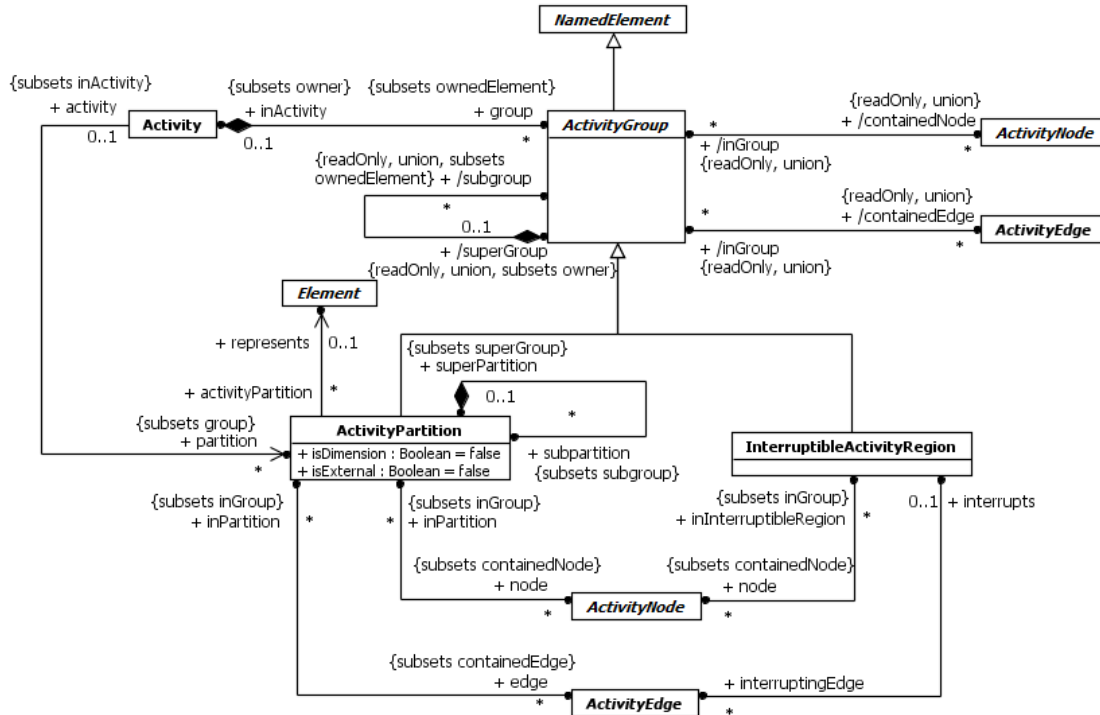


Figure 15.65 ActivityGroups

## 15.6.3 Semantics

### Activity Partitions

An ActivityPartition is a kind of ActivityGroup for identifying ActivityNodes that have some characteristics in common. ActivityPartitions 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.

ActivityPartitions do not affect the token flow of the model. They constrain and provide a view on the Behaviors invoked due to the execution of the containedNodes and containedEdges of the partition, including Operation calls and Signal sends. This may be due not only to the execution of explicit InvocationActions (see sub clause 16.3) but also the implicit invocation of, e.g., transformation and specification Behaviors. Constraints vary according to the kind of element that the partition represents as listed below (These following constraints are normative).

- *Classifier*. Behaviors invoked within the partition are the responsibility of instances of the Classifier that that the partition represents. The context of all invoked Behaviors shall be the Classifier. Operation calls and Signal sends within the partition shall target objects at runtime that are instances of the Classifier.
- *InstanceSpecification*. Behaviors invoked within the partition are the responsibility of the instance modeled by the InstanceSpecification that the partition represents. The context of all invoked Behaviors shall be a Classifier of the InstanceSpecification. Operation calls and Signal sends within the partition shall target the instances represented modeled by the InstanceSpecification.

- *Property*. Behaviors invoked within the partition are the responsibility of the instance or instances held by the Property that the partition represents. The context of all invoked Behaviors shall be the type of the Property. Operation calls and Signal sends within the partition shall target an instance held by the Property at runtime. If the Property holds more than one value, the invocations are treated as if they were made concurrently on each value and the invocation does not complete until all the concurrent instances of it complete.

An ActivityPartition may represent other kinds of Elements than the above, but the semantics for these are not defined in this specification.

An ActivityPartition may have subpartitions. If an ActivityPartition has `isDimension=true`, then it is a dimension partition for its subpartitions. Dimension partitions shall not be contained in any other ActivityPartitions. For example, an Activity may have one dimension partition for the location at which invoked Behaviors are carried out (represented by each of the subpartitions) and another for the cost of performing them.

If an ActivityPartition represents a Property, and its subpartitions represent InstanceSpecifications, then the InstanceSpecifications shall model values held by that Property. Behaviors invoked in the subpartitions shall be consistent with the constraints required for both the InstanceSpecification of the subpartition and the Property of the containing ActivityPartition. For example, a partition may represent the location at which an invoked Behavior is carried out, and the subpartitions would represent specific values for that Property, such as Chicago. The location Property might be an attribute of the context BehaviorClassifier of the enclosing Activity or even of the Activity itself.

If an ActivityPartition represents a Property that is an attribute of a Classifier (see sub clause [9.2](#)), and is contained by another partition, then the superPartition shall represent that Classifier or a Property whose type is that Classifier. All other non-external subpartitions in the superPartition shall also represent attributes of the Classifier. At runtime, the targets of invocations of Behaviors in the subpartitions shall be the values of the represented attribute of the same instance of the Classifier represented by the superPartition, including Operation calls and Signal sends. If the superPartition is a partition of an enclosing Activity that has the Classifier as its context, then, at runtime, the context object of the executing Activity shall be the same instance of the Classifier represented by the superPartition that are targets of invocations in the subpartitions.

If a non-external ActivityPartition represents a Classifier and is contained in another partition, then the superPartition shall also represent a Classifier, and the Classifier of the subpartition must be a nestedClassifier or ownedBehavior of the Classifier represented by the superPartition or be at the contained end of a composition Association with the Classifier represented by the superPartition. If the latter, then, at runtime, the target for invocations of Behaviors in the subpartition, including Operation calls and Signal sends, shall be considered to be an instance of the Classifier represented by the subpartition that is linked to an instance of the Classifier represented by the superPartition by the composition Association.

An external ActivityPartition is one with `isExternal=true`. External partitions are intentional exceptions to the rules for partition structure. For example, a dimension partition may have partitions showing the parts of a StructuredClassifier. It can then have an external partition that does not represent one of the parts, but a completely separate Classifier. In business modeling, external partitions can be used to model entities outside a business.

ActivityPartitions may be used in a way that provides enough information for review by high-level modelers, though not enough for execution. For example, if a partition represents a Classifier, then Behaviors invoked in that partition are the responsibility of instances of the Classifier, but the model may or may not say which instance in particular. Thus, an invocation of an Operation would be limited to an Operation on that Classifier, but an input ObjectFlow to the invocation might not be specified to tell which instance should be the target at runtime. The ObjectFlow could be specified in a later stage of development to support execution. Another option would be to use ActivationPartitions that represent parts. Then, when the Activity executes in the context of a particular object, the parts of that object at runtime will be used as targets for the Operation calls and Signal sends, as described above.

### Interruptible Activity Regions

An InterruptibleActivityRegion is an ActivityGroup that supports termination of a portion of an Activity. An InterruptibleActivityRegion contains only ActivityNodes. It also identifies as `interruptingEdges` certain ActivityEdges that have their source within the region and their target outside the region. When a token offered along an `interruptingEdge` is accepted and

traverses that edge, then the execution of all containedNodes of the region is terminated and all tokens are removed from them. However, the token traversing the interruptingEdge still arrives at its target and, further, any accepted tokens traversing non-interrupting edges from a source node within the region to a target node outside the region also still arrive at the target nodes, even if the interruption occurs during the traversal.

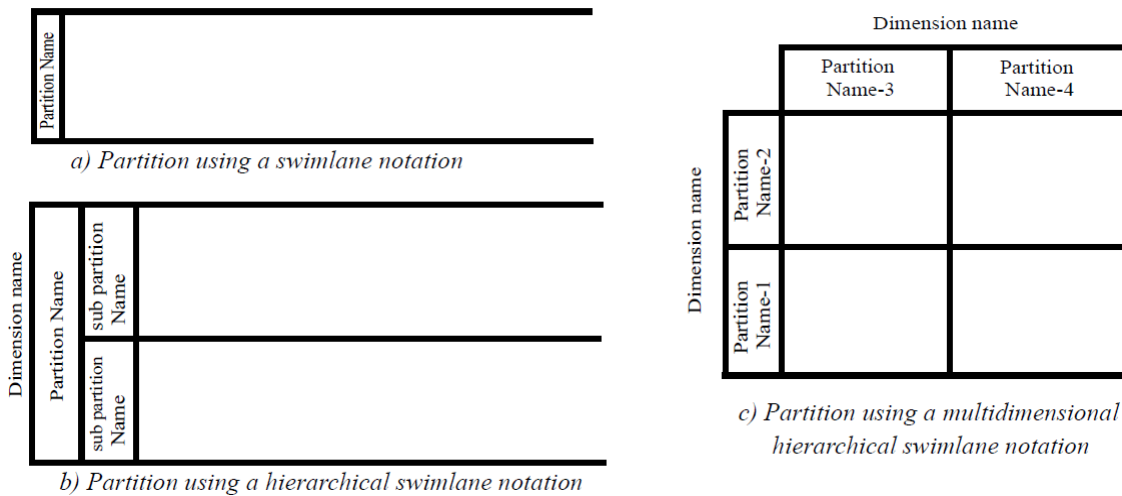
AcceptEventActions in the region that do not have incoming edges are enabled only when a token enters the region, even if the token is not directed at the AcceptEventAction (see sub clause 16.10 for a full description of AcceptEventActions).

Do not use an InterruptibleActivityRegion if it is not desired to abort all flows in the region in some cases. For example, if a single execution of an Activity is being used for all its invocations (i.e., isSingleExecution=true), then multiple streams of tokens will be flowing through the same Activity. In this case, it is probably not desired to abort all flows in a region just because one token leaves the region. Arrange for separate invocations of the Activity to use separate executions of the Activity (i.e., isSingleExecution=false) when employing InterruptibleActivityRegions, so tokens from different invocations will not affect each other.

## 15.6.4 Notation

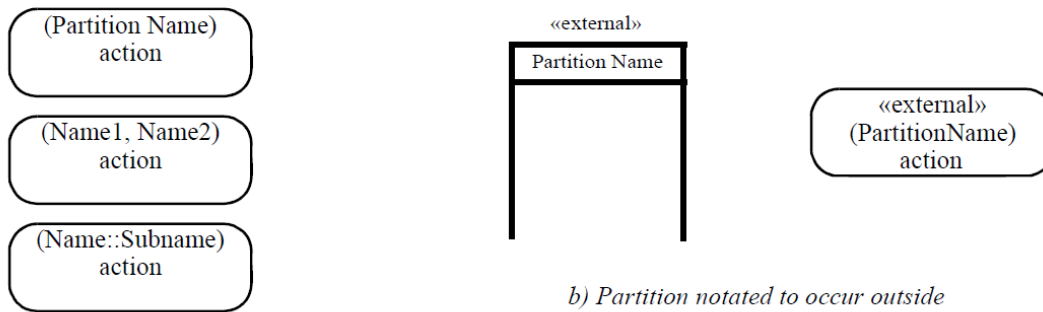
### Activity Partitions

An ActivityPartition is notated with two, usually parallel lines, either horizontal or vertical, and a name labeling the partition in a box at one end. Any ActivityNodes and ActivityEdges placed between these lines are considered to be contained within the partition. This notation for an ActivityPartition is colloquially known as a swimlane, as shown in Figure 15.66(a). Swimlanes can express hierarchical partitioning by representing the subpartitions as further partitioning of the superpartition, as illustrated in Figure 15.66(b). Diagrams can also be partitioned multidimensionally, as depicted in Figure 15.66(c), where each “swim cell” is an intersection of multiple partitions. The partitions within each dimension may be grouped into an enclosing activity partition with isDimension=true, whose name is the dimension name. Rather than being shown as a partition itself, however, the dimension is indicated by placing its name alongside the set of partitions in the dimension, as shown in Figure 15.66(c).



**Figure 15.66 ActivityPartition notations**

In some diagramming situations, using parallel lines to delineate ActivityPartitions is not practical. An alternative is to place the partition name in parenthesis above the ActivityNode name, as illustrated in Figure 15.67(a), below. A comma-separated list of partition names means that the node is contained in more than one partition. A double colon within a partition name indicates that the partition is nested, with the superPartitions coming earlier in the name. An external partition (with isExternal=true) is labeled with the keyword «external», as illustrated in Figure 15.67(b). If an ActivityNode within a non-external swimlane is given a specific external partition name, as shown on the right in Figure 15.67(b), then the ActivityNode is considered to be contained in the named external partition, rather than the partition denoted by the swimlane.



a) Partition notated on a specific activity

b) Partition notated to occur outside the primary concern of the model.

**Figure 15.67 ActivityPartition notations**

When ActivityPartition swimlane notation is combined with the frame notation for Activity (see sub clause 15.2.4), the outside edges of the top level partition swimlanes can be merged with the Activity frame.

### Interruptible Activity Regions

An InterruptibleActivityRegion is notated by a dashed, round-cornered rectangle drawn around the nodes contained by the region, as shown in Figure 15.68. An interruptingEdge is notated with a “lightning bolt” ActivityEdge.



**Figure 15.68 InterruptibleActivityRegion**

An alternative option for notating an interruptingEdge is a “zig-zag” adornment on a straight line, as shown in Figure 15.70.



**Figure 15.69 InterruptibleActivityRegion alternative notation**

## 15.6.5 Examples

### Activity Partitions

Figure 15.70 illustrates an example of partitioning an order processing Activity diagram into swimlanes. The top partition contains the portion of an Activity for which the Order Department is responsible; the middle partition, the Accounting Department, and the bottom the Customer. Accounting Department and Order Department are values of the performingDept attribute. Customer, on the other hand, is external to the domain.

**NOTE.** ActivityEdges that cross between partitions are not contained in any of the subpartitions.

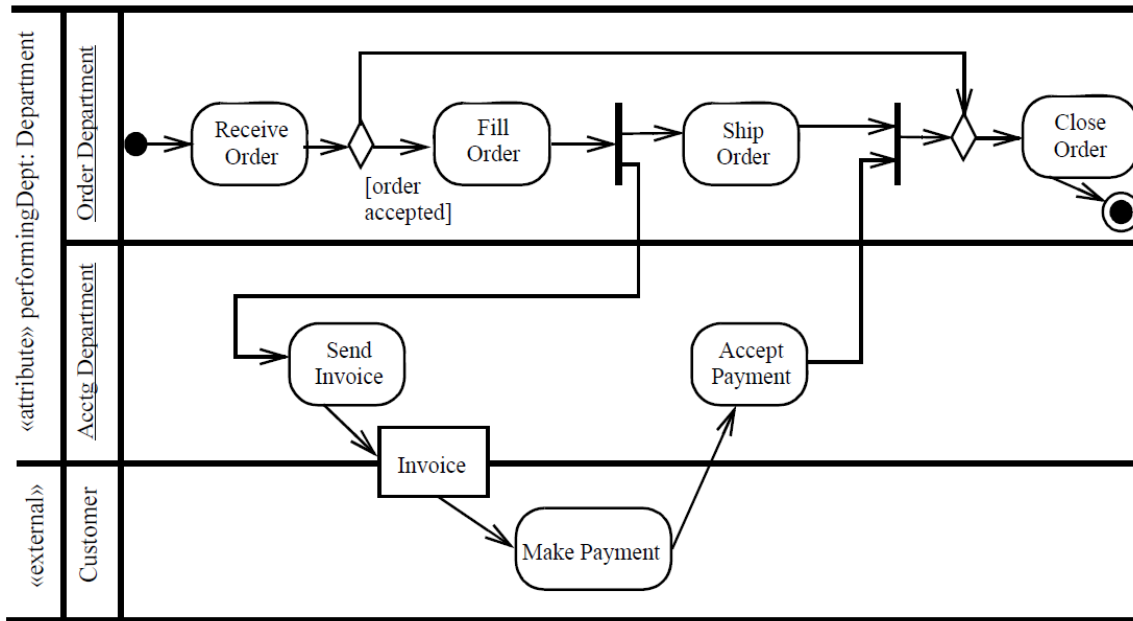


Figure 15.70 ActivityPartitions using swimlane notation

Figure 15.71 shows the same partitioning as in Figure 15.70, but using annotated ActivityNodes rather than swimlanes.

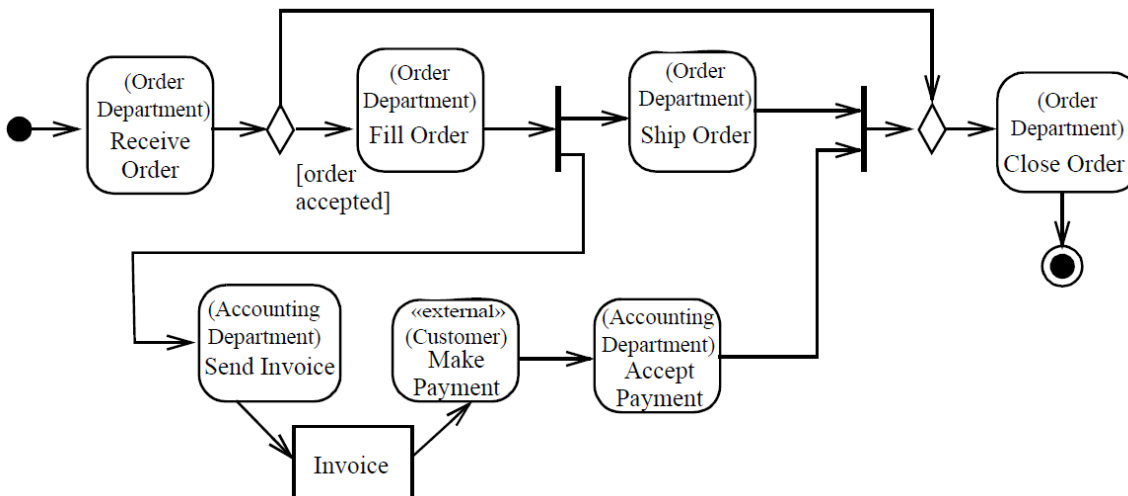


Figure 15.71 ActivityPartitions using annotation

Figure 15.71 depicts multidimensional swimlanes. The Receive Order and Fill Order Behaviors are performed by an instance of the Order Processor class, situated in Seattle, but not necessarily the same instance for both Behaviors. Even though the Make Payment node is contained within the Seattle/Accounting Clerk swim cell, its performer and location are not specified by the containing swimlanes, because it has an overriding external partition annotation.

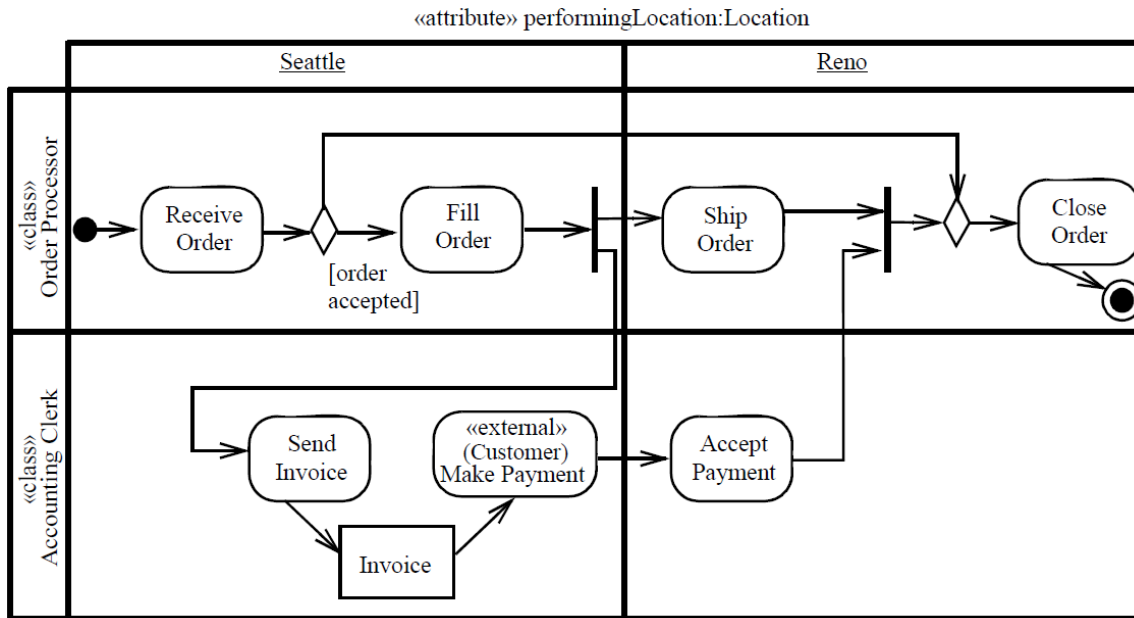


Figure 15.72 ActivityPartitions using multidimensional swimlane notation

### Interruptible Activity Regions

Figure 15.73 illustrates the use of an InterruptibleActivityRegion. When an order cancellation request is made while receiving, filling, or shipping orders, that flow is terminated and the Cancel Order node is executed.

**NOTE.** If this happens after Fill Order is finished, invoicing might have already been initiated (due to the ForkNode after Fill Order).

As this flow is outside the InterruptibleActivityRegion, it will not be terminated by an order cancellation request, even though Ship Order will be.

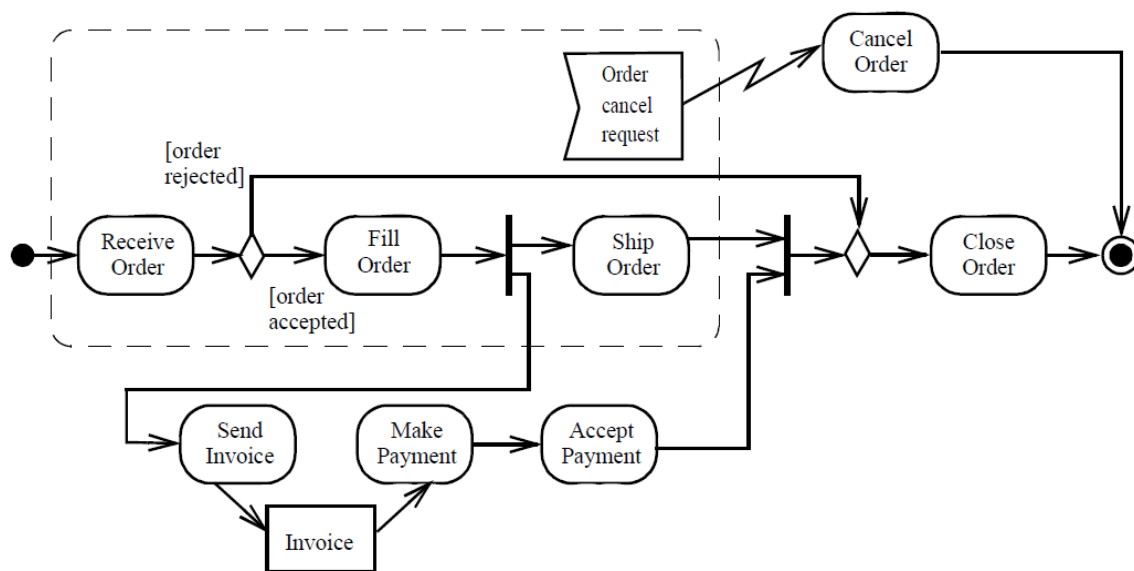


Figure 15.73 InterruptibleActivityRegion example





## 15.7 Classifier Descriptions

### Activity [Class]

#### Description

An Activity is the specification of parameterized Behavior as the coordinated sequencing of subordinate units.

#### Diagrams

[Activities](#), [Activity Groups](#), [Structured Actions](#)

#### Generalizations

[Behavior](#)

#### Attributes

- isReadOnly : [Boolean](#) [1..1] = false  
If true, this Activity must not make any changes to objects. The default is false (an Activity may make nonlocal changes). (This is an assertion, not an executable property. It may be used by an execution engine to optimize model execution. If the assertion is violated by the Activity, then the model is ill-formed.)
- isSingleExecution : [Boolean](#) [1..1] = false  
If true, all invocations of the Activity are handled by the same execution.

#### Association Ends

- ♦ edge : [ActivityEdge](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [ActivityEdge::activity](#))  
ActivityEdges expressing flow between the nodes of the Activity.
- ♦ group : [ActivityGroup](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [ActivityGroup::inActivity](#))  
Top-level ActivityGroups in the Activity.
- ♦ node : [ActivityNode](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [ActivityNode::activity](#))  
ActivityNodes coordinated by the Activity.
- partition : [ActivityPartition](#) [0..\*]{subsets [Activity::group](#)} (opposite [A\\_partition\\_activity::activity](#))  
Top-level ActivityPartitions in the Activity.
- ♦ structuredNode : [StructuredActivityNode](#) [0..\*]{subsets [Activity::group](#), subsets [Activity::node](#)} (opposite [StructuredActivityNode::activity](#))  
Top-level StructuredActivityNodes in the Activity.
- ♦ variable : [Variable](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [Variable::activityScope](#))  
Top-level Variables defined by the Activity.

## Constraints

- **maximum\_one\_parameter\_node**  
A Parameter with direction other than inout must have exactly one ActivityParameterNode in an Activity.

```
inv: ownedParameter->forall(p |
    p.direction <> ParameterDirectionKind::inout implies node->select(
        oclIsKindOf(ActivityParameterNode) and oclAsType(ActivityParameterNode).parameter = p) -
    >size() = 1)
```

- **maximum\_two\_parameter\_nodes**  
A Parameter with direction inout must have exactly two ActivityParameterNodes in an Activity, at most one with incoming ActivityEdges and at most one with outgoing ActivityEdges.

```
inv: ownedParameter->forall(p |
    p.direction = ParameterDirectionKind::inout implies
    let associatedNodes : Set(ActivityNode) = node->select(
        oclIsKindOf(ActivityParameterNode) and oclAsType(ActivityParameterNode).parameter = p) in
    associatedNodes->size()=2 and
    associatedNodes->select(incoming->notEmpty())->size()<=1 and
    associatedNodes->select(outgoing->notEmpty())->size()<=1
)
```

## ActivityEdge [Abstract Class]

### Description

An ActivityEdge is an abstract class for directed connections between two ActivityNodes.

### Diagrams

[Activities](#), [Activity Groups](#), [Information Flows](#), [Structured Actions](#)

### Generalizations

[RedefinableElement](#)

### Specializations

[ControlFlow](#), [ObjectFlow](#)

### Attributes

### Association Ends

- activity : [Activity](#) [0..1]{subsets [Element::owner](#)} (opposite [Activity::edge](#))  
The Activity containing the ActivityEdge, if it is directly owned by an Activity.
- ♦ guard : [ValueSpecification](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_guard\\_activityEdge::activityEdge](#))  
A ValueSpecification that is evaluated to determine if a token can traverse the ActivityEdge. If an ActivityEdge has no guard, then there is no restriction on tokens traversing the edge.

- /inGroup : [ActivityGroup](#) [0..\*]{union} (opposite [ActivityGroup::containedEdge](#))  
ActivityGroups containing the ActivityEdge.
- inPartition : [ActivityPartition](#) [0..\*]{subsets [ActivityEdge::inGroup](#)} (opposite [ActivityPartition::edge](#))  
ActivityPartitions containing the ActivityEdge.
- inStructuredNode : [StructuredActivityNode](#) [0..1]{subsets [ActivityEdge::inGroup](#), subsets [Element::owner](#)} (opposite [StructuredActivityNode::edge](#))  
The StructuredActivityNode containing the ActivityEdge, if it is owned by a StructuredActivityNode.
- interrupts : [InterruptibleActivityRegion](#) [0..1] (opposite [InterruptibleActivityRegion::interruptingEdge](#))  
The InterruptibleActivityRegion for which this ActivityEdge is an interruptingEdge.
- redefinedEdge : [ActivityEdge](#) [0..\*]{subsets [RedefinableElement::redefinedElement](#)} (opposite [A\\_redefinedEdge\\_activityEdge::activityEdge](#))  
ActivityEdges from a generalization of the Activity containing this ActivityEdge that are redefined by this ActivityEdge.
- source : [ActivityNode](#) [1..1] (opposite [ActivityNode::outgoing](#))  
The ActivityNode from which tokens are taken when they traverse the ActivityEdge.
- target : [ActivityNode](#) [1..1] (opposite [ActivityNode::incoming](#))  
The ActivityNode to which tokens are put when they traverse the ActivityEdge.
- ♦ weight : [ValueSpecification](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_weight\\_activityEdge::activityEdge](#))  
The minimum number of tokens that must traverse the ActivityEdge at the same time. If no weight is specified, this is equivalent to specifying a constant value of 1.

## Operations

- isConsistentWith(redefinee : [RedefinableElement](#)) : [Boolean](#)

body: `redefinee.ooclIsKindOf(ActivityEdge)`

## Constraints

- source\_and\_target  
If an ActivityEdge is directly owned by an Activity, then its source and target must be directly or indirectly contained in the same Activity.

inv: `activity<>null implies source.containingActivity() = activity and target.containingActivity() = activity`

## ActivityFinalNode [Class]

### Description

An ActivityFinalNode is a FinalNode that terminates the execution of its owning Activity or StructuredActivityNode.

### Diagrams

[Control Nodes](#)

## Generalizations

[FinalNode](#)

## ActivityGroup [Abstract Class]

### Description

ActivityGroup is an abstract class for defining sets of ActivityNodes and ActivityEdges in an Activity.

### Diagrams

[Activity Groups](#), [Structured Actions](#)

## Generalizations

[NamedElement](#)

## Specializations

[ActivityPartition](#), [InterruptibleActivityRegion](#), [StructuredActivityNode](#)

## Association Ends

- /containedEdge : [ActivityEdge](#) [0..\*]{union} (opposite [ActivityEdge::inGroup](#))  
ActivityEdges immediately contained in the ActivityGroup.
- /containedNode : [ActivityNode](#) [0..\*]{union} (opposite [ActivityNode::inGroup](#))  
ActivityNodes immediately contained in the ActivityGroup.
- inActivity : [Activity](#) [0..1]{subsets [Element::owner](#)} (opposite [Activity::group](#))  
The Activity containing the ActivityGroup, if it is directly owned by an Activity.
- ♦ /subgroup : [ActivityGroup](#) [0..\*]{union, subsets [Element::ownedElement](#)} (opposite [ActivityGroup::superGroup](#))  
Other ActivityGroups immediately contained in this ActivityGroup.
- /superGroup : [ActivityGroup](#) [0..1]{union, subsets [Element::owner](#)} (opposite [ActivityGroup::subgroup](#))  
The ActivityGroup immediately containing this ActivityGroup, if it is directly owned by another ActivityGroup.

## Operations

- containingActivity() : [Activity](#) [0..1]  
The Activity that directly or indirectly contains this ActivityGroup.

```
body: if superGroup<>null then superGroup.containingActivity()  
else inActivity  
endif
```

## Constraints

- nodes\_and\_edges  
All containedNodes and containedEdges of an ActivityGroup must be in the same Activity as the group.

```
inv: containedNode->forall(activity = self.containingActivity()) and
containedEdge->forall(activity = self.containingActivity())
```

- not\_contained  
No containedNode or containedEdge of an ActivityGroup may be contained by its subgroups or its superGroups, transitively.

```
inv: subgroup->closure(subgroup).containedNode->excludesAll(containedNode) and
superGroup->closure(superGroup).containedNode->excludesAll(containedNode) and
subgroup->closure(subgroup).containedEdge->excludesAll(containedEdge) and
superGroup->closure(superGroup).containedEdge->excludesAll(containedEdge)
```

## ActivityNode [Abstract Class]

### Description

ActivityNode is an abstract class for points in the flow of an Activity connected by ActivityEdges.

### Diagrams

[Object Nodes](#), [Activities](#), [Activity Groups](#), [Control Nodes](#), [Executable Nodes](#), [Structured Actions](#)

### Generalizations

[RedefinableElement](#)

### Specializations

[ControlNode](#), [ExecutableNode](#), [ObjectNode](#)

### Attributes

### Association Ends

- activity : [Activity](#) [0..1]{subsets [Element::owner](#)} (opposite [Activity::node](#))  
The Activity containing the ActivityNode, if it is directly owned by an Activity.
- /inGroup : [ActivityGroup](#) [0..\*]{union} (opposite [ActivityGroup::containedNode](#))  
ActivityGroups containing the ActivityNode.
- inInterruptibleRegion : [InterruptibleActivityRegion](#) [0..\*]{subsets [ActivityNode::inGroup](#)} (opposite [InterruptibleActivityRegion::node](#))  
InterruptibleActivityRegions containing the ActivityNode.
- inPartition : [ActivityPartition](#) [0..\*]{subsets [ActivityNode::inGroup](#)} (opposite [ActivityPartition::node](#))  
ActivityPartitions containing the ActivityNode.
- inStructuredNode : [StructuredActivityNode](#) [0..1]{subsets [Element::owner](#), subsets [ActivityNode::inGroup](#)} (opposite [StructuredActivityNode::node](#))  
The StructuredActivityNode containing the ActivityNode, if it is directly owned by a StructuredActivityNode.

- incoming : [ActivityEdge](#) [0..\*] (opposite [ActivityEdge::target](#))  
ActivityEdges that have the ActivityNode as their target.
- outgoing : [ActivityEdge](#) [0..\*] (opposite [ActivityEdge::source](#))  
ActivityEdges that have the ActivityNode as their source.
- redefinedNode : [ActivityNode](#) [0..\*]{subsets [RedefinableElement::redefinedElement](#)} (opposite [A\\_redefinedNode\\_activityNode::activityNode](#))  
ActivityNodes from a generalization of the Activity containing this ActivityNode that are redefined by this ActivityNode.

## Operations

- containingActivity() : [Activity](#) [0..1]  
The Activity that directly or indirectly contains this ActivityNode.

```
body: if inStructuredNode<>null then inStructuredNode.containingActivity()
else activity
endif
```

- isConsistentWith(redefinee : [RedefinableElement](#)) : [Boolean](#)

```
body: redefinee.oclIsKindOf(ActivityNode)
```

## ActivityParameterNode [Class]

### Description

An ActivityParameterNode is an ObjectNode for accepting values from the input Parameters or providing values to the output Parameters of an Activity.

### Diagrams

[Object Nodes](#)

### Generalizations

[ObjectNode](#)

### Association Ends

- parameter : [Parameter](#) [1..1] (opposite [A\\_parameter\\_activityParameterNode::activityParameterNode](#))  
The Parameter for which the ActivityParameterNode will be accepting or providing values.

### Constraints

- no\_outgoing\_edges  
An ActivityParameterNode with no outgoing ActivityEdges and one or more incoming ActivityEdges must have a parameter with direction out, inout, or return.

```
inv: (incoming->notEmpty() and outgoing->isEmpty()) implies
(parameter.direction = ParameterDirectionKind::out or
parameter.direction = ParameterDirectionKind::inout or
parameter.direction = ParameterDirectionKind::return)
```

- **has\_parameters**  
The parameter of an ActivityParameterNode must be from the containing Activity.

```
inv: activity.ownedParameter->includes(parameter)
```

- **same\_type**  
The type of an ActivityParameterNode is the same as the type of its parameter.

```
inv: type = parameter.type
```

- **no\_incoming\_edges**  
An ActivityParameterNode with no incoming ActivityEdges and one or more outgoing ActivityEdges must have a parameter with direction in or inout.

```
inv: (outgoing->notEmpty() and incoming->isEmpty()) implies
(parameter.direction = ParameterDirectionKind::_'in' or
parameter.direction = ParameterDirectionKind::inout)
```

- **no\_edges**  
An ActivityParameterNode may have all incoming ActivityEdges or all outgoing ActivityEdges, but it must not have both incoming and outgoing ActivityEdges.

```
inv: incoming->isEmpty() or outgoing->isEmpty()
```

## ActivityPartition [Class]

### Description

An ActivityPartition is a kind of ActivityGroup for identifying ActivityNodes that have some characteristic in common.

### Diagrams

[Activity Groups](#)

### Generalizations

[ActivityGroup](#)

### Attributes

- **isDimension** : [Boolean](#) [1..1] = false  
Indicates whether the ActivityPartition groups other ActivityPartitions along a dimension.
- **isExternal** : [Boolean](#) [1..1] = false  
Indicates whether the ActivityPartition represents an entity to which the partitioning structure does not apply.

## Association Ends

- edge : [ActivityEdge](#) [0..\*]{subsets [ActivityGroup::containedEdge](#)} (opposite [ActivityEdge::inPartition](#))  
ActivityEdges immediately contained in the ActivityPartition.
- node : [ActivityNode](#) [0..\*]{subsets [ActivityGroup::containedNode](#)} (opposite [ActivityNode::inPartition](#))  
ActivityNodes immediately contained in the ActivityPartition.
- represents : [Element](#) [0..1] (opposite [A represents activityPartition::activityPartition](#))  
An Element represented by the functionality modeled within the ActivityPartition.
- ♦ subpartition : [ActivityPartition](#) [0..\*]{subsets [ActivityGroup::subgroup](#)} (opposite [ActivityPartition::superPartition](#))  
Other ActivityPartitions immediately contained in this ActivityPartition (as its subgroups).
- superPartition : [ActivityPartition](#) [0..1]{subsets [ActivityGroup::superGroup](#)} (opposite [ActivityPartition::subpartition](#))  
Other ActivityPartitions immediately containing this ActivityPartition (as its superGroups).

## Constraints

- represents\_classifier  
If a non-external ActivityPartition represents a Classifier and has a superPartition, then the superPartition must represent a Classifier, and the Classifier of the subpartition must be nested (nestedClassifier or ownedBehavior) in the Classifier represented by the superPartition, or be at the contained end of a composition Association with the Classifier represented by the superPartition.

```
inv: (not isExternal and represents.ocIsKindOf(Classifier) and superPartition->notEmpty()) implies
(
  let representedClassifier : Classifier = represents.ocAsType(Classifier) in
  superPartition.represents.ocIsKindOf(Classifier) and
  let representedSuperClassifier : Classifier = superPartition.represents.ocAsType(Classifier)
in
  (representedSuperClassifier.ocIsKindOf(BehavioedClassifier) and
  representedClassifier.ocIsKindOf(Behavior) and
  representedSuperClassifier.ocAsType(BehavioedClassifier).ownedBehavior-
>includes(representedClassifier.ocAsType(Behavior)))
  or
  (representedSuperClassifier.ocIsKindOf(Class) and
  representedSuperClassifier.ocAsType(Class).nestedClassifier->includes(representedClassifier))
  or
  (Association.allInstances()->exists(a | a.memberEnd->exists(end1 | end1.isComposite and
  end1.type = representedClassifier and
  a.memberEnd->exists(end2 |
  end1<>end2 and end2.type = representedSuperClassifier))))
)
```

- represents\_property\_and\_is\_contained  
If an ActivityPartition represents a Property and has a superPartition, then the Property must be of a Classifier represented by the superPartition, or of a Classifier that is the type of a Property represented by the superPartition.

```
inv: (represents.ocIsKindOf(Property) and superPartition->notEmpty()) implies
(
  (superPartition.represents.ocIsKindOf(Classifier) and represents.owner =
  superPartition.represents) or
  (superPartition.represents.ocIsKindOf(Property) and represents.owner =
  superPartition.represents.ocAsType(Property).type)
)
```



)

- **represents\_property**  
If an ActivityPartition represents a Property and has a superPartition representing a Classifier, then all the other non-external subpartitions of the superPartition must represent Properties directly owned by the same Classifier.

```
inv: (represents.ocIsKindOf(Property) and superPartition->notEmpty() and
superPartition.represents.ocIsKindOf(Classifier)) implies
(
  let representedClassifier : Classifier = superPartition.represents.ocAsType(Classifier)
  in
    superPartition.subpartition->reject(isExternal)->forall(p |
      p.represents.ocIsKindOf(Property) and p.owner=representedClassifier)
)
```

- **dimension\_not\_contained**  
An ActivityPartition with isDimension = true may not be contained by another ActivityPartition.

```
inv: isDimension implies superPartition->isEmpty()
```

## CentralBufferNode [Class]

### Description

A CentralBufferNode is an ObjectNode for managing flows from multiple sources and targets.

### Diagrams

[Object Nodes](#)

### Generalizations

[ObjectNode](#)

### Specializations

[DataStoreNode](#)

## ControlFlow [Class]

### Description

A ControlFlow is an ActivityEdge traversed by control tokens or object tokens of control type, which are use to control the execution of ExecutableNodes.

### Diagrams

[Activities](#)

### Generalizations

[ActivityEdge](#)

## Constraints

- `object_nodes`  
ControlFlows may not have ObjectNodes at either end, except for ObjectNodes with control type.

```
inv: (source.oclIsKindOf(ObjectNode) implies source.oclAsType(ObjectNode).isControlType) and  
(target.oclIsKindOf(ObjectNode) implies target.oclAsType(ObjectNode).isControlType)
```

## ControlNode [Abstract Class]

### Description

A ControlNode is an abstract ActivityNode that coordinates flows in an Activity.

### Diagrams

[Control Nodes](#)

### Generalizations

[ActivityNode](#)

### Specializations

[DecisionNode](#), [FinalNode](#), [ForkNode](#), [InitialNode](#), [JoinNode](#), [MergeNode](#)

## DataStoreNode [Class]

### Description

A DataStoreNode is a CentralBufferNode for persistent data.

### Diagrams

[Object Nodes](#)

### Generalizations

[CentralBufferNode](#)

## DecisionNode [Class]

### Description

A DecisionNode is a ControlNode that chooses between outgoing ActivityEdges for the routing of tokens.

### Diagrams

[Control Nodes](#)

### Generalizations

[ControlNode](#)

## Association Ends

- decisionInput : [Behavior](#) [0..1] (opposite [A decisionInput decisionNode::decisionNode](#))  
A Behavior that is executed to provide an input to guard ValueSpecifications on ActivityEdges outgoing from the DecisionNode.
- decisionInputFlow : [ObjectFlow](#) [0..1] (opposite [A decisionInputFlow decisionNode::decisionNode](#))  
An additional ActivityEdge incoming to the DecisionNode that provides a decision input value for the guards ValueSpecifications on ActivityEdges outgoing from the DecisionNode.

## Constraints

- zero\_input\_parameters  
If the DecisionNode has no decisionInputFlow and an incoming ControlFlow, then any decisionInput Behavior has no in parameters.

```
inv: (decisionInput<>null and decisionInputFlow=null and incoming->exists(oclIsKindOf(ControlFlow)))  
implies  
    decisionInput.inputParameters()->isEmpty()
```

- edges  
The ActivityEdges incoming to and outgoing from a DecisionNode, other than the decisionInputFlow (if any), must be either all ObjectFlows or all ControlFlows.

```
inv: let allEdges: Set(ActivityEdge) = incoming->union(outgoing) in  
let allRelevantEdges: Set(ActivityEdge) = if decisionInputFlow->notEmpty() then allEdges-  
>excluding(decisionInputFlow) else allEdges endif in  
allRelevantEdges->forall(oclIsKindOf(ControlFlow)) or allRelevantEdges-  
>forall(oclIsKindOf(ObjectFlow))
```

- decision\_input\_flow\_incoming  
The decisionInputFlow of a DecisionNode must be an incoming ActivityEdge of the DecisionNode.

```
inv: incoming->includes(decisionInputFlow)
```

- two\_input\_parameters  
If the DecisionNode has a decisionInputFlow and an second incoming ObjectFlow, then any decisionInputBehavior has two in Parameters, the first of which has a type that is the same as or a supertype of the type of object tokens offered on the non-decisionInputFlow and the second of which has a type that is the same as or a supertype of the type of object tokens offered on the decisionInputFlow.

```
inv: (decisionInput<>null and decisionInputFlow<>null and incoming->forall(oclIsKindOf(ObjectFlow)))  
implies  
    decisionInput.inputParameters()->size()=2
```

- incoming\_outgoing\_edges  
A DecisionNode has one or two incoming ActivityEdges and at least one outgoing ActivityEdge.

```
inv: (incoming->size() = 1 or incoming->size() = 2) and outgoing->size() > 0
```

- **incoming\_control\_one\_input\_parameter**

If the DecisionNode has a decisionInputFlow and an incoming ControlFlow, then any decisionInput Behavior has one in Parameter whose type is the same as or a supertype of the type of object tokens offered on the decisionInputFlow.

```
inv: (decisionInput<>null and decisionInputFlow<>null and incoming->exists(oclIsKindOf(ControlFlow)))
implies
    decisionInput.inputParameters()->size()==1
```

- **parameters**

A decisionInput Behavior has no out parameters, no inout parameters, and one return parameter.

```
inv: decisionInput<>null implies
    (decisionInput.ownedParameter->forall(par |
        par.direction <> ParameterDirectionKind::out and
        par.direction <> ParameterDirectionKind::inout ) and
    decisionInput.ownedParameter->one(par |
        par.direction <> ParameterDirectionKind::return))
```

- **incoming\_object\_one\_input\_parameter**

If the DecisionNode has no decisionInputFlow and an incoming ObjectFlow, then any decisionInput Behavior has one in Parameter whose type is the same as or a supertype of the type of object tokens offered on the incoming ObjectFlow.

```
inv: (decisionInput<>null and decisionInputFlow=null and incoming->forall(oclIsKindOf(ObjectFlow)))
implies
    decisionInput.inputParameters()->size()==1
```

## ExceptionHandler [Class]

### Description

An ExceptionHandler is an Element that specifies a handlerBody ExecutableNode to execute in case the specified exception occurs during the execution of the protected ExecutableNode.

### Diagrams

[Executable Nodes](#)

### Generalizations

[Element](#)

### Association Ends

- exceptionInput : [ObjectNode](#) [1..1] (opposite [A\\_exceptionInput\\_exceptionHandler::exceptionHandler](#))  
An ObjectNode within the handlerBody. When the ExceptionHandler catches an exception, the exception token is placed on this ObjectNode, causing the handlerBody to execute.
- exceptionType : [Classifier](#) [1..\*] (opposite [A\\_exceptionType\\_exceptionHandler::exceptionHandler](#))  
The Classifiers whose instances the ExceptionHandler catches as exceptions. If an exception occurs whose type is any exceptionType, the ExceptionHandler catches the exception and executes the handlerBody.

- handlerBody : [ExecutableNode](#) [1..1] (opposite [A handlerBody exceptionHandler::exceptionHandler](#))  
An ExecutableNode that is executed if the ExceptionHandler catches an exception.
- protectedNode : [ExecutableNode](#) [1..1]{subsets [Element::owner](#)} (opposite [ExecutableNode::handler](#))  
The ExecutableNode protected by the ExceptionHandler. If an exception propagates out of the protectedNode and has a type matching one of the exceptionTypes, then it is caught by this ExceptionHandler.

## Constraints

- handler\_body\_edges  
The handlerBody has no incoming or outgoing ActivityEdges and the exceptionInput has no incoming ActivityEdges.

```
inv: handlerBody.incoming->isEmpty() and handlerBody.outgoing->isEmpty() and exceptionInput.incoming->isEmpty()
```

- output\_pins  
If the protectedNode is an Action with OutputPins, then the handlerBody must also be an Action with the same number of OutputPins, which are compatible in type, ordering, and multiplicity to those of the protectedNode.

```
inv: (protectedNode.oclIsKindOf(Action) and protectedNode.oclAsType(Action).output->notEmpty())
implies
(
  handlerBody.oclIsKindOf(Action) and
  let protectedNodeOutput : OrderedSet(OutputPin) = protectedNode.oclAsType(Action).output,
      handlerBodyOutput : OrderedSet(OutputPin) = handlerBody.oclAsType(Action).output in
  protectedNodeOutput->size() = handlerBodyOutput->size() and
  Sequence{1..protectedNodeOutput->size()}->forall(i |
    handlerBodyOutput->at(i).type.conformsTo(protectedNodeOutput->at(i).type) and
    handlerBodyOutput->at(i).isOrdered=protectedNodeOutput->at(i).isOrdered and
    handlerBodyOutput->at(i).compatibleWith(protectedNodeOutput->at(i)))
)
```

- one\_input  
The handlerBody is an Action with one InputPin, and that InputPin is the same as the exceptionInput.

```
inv: handlerBody.oclIsKindOf(Action) and
let inputs: OrderedSet(InputPin) = handlerBody.oclAsType(Action).input in
inputs->size()==1 and inputs->first()==exceptionInput
```

- edge\_source\_target  
An ActivityEdge that has a source within the handlerBody of an ExceptionHandler must have its target in the handlerBody also, and vice versa.

```
inv: let nodes:Set(ActivityNode) = handlerBody.oclAsType(Action).allOwnedNodes() in
nodes.outgoing->forall(nodes->includes(target)) and
nodes.incoming->forall(nodes->includes(source))
```

- handler\_body\_owner  
The handlerBody must have the same owner as the protectedNode.

```
inv: handlerBody.owner=protectedNode.owner
```

- `exception_input_type`  
The `exceptionInput` must either have no type or every `exceptionType` must conform to the `exceptionInput` type.

```
inv: exceptionInput.type=null or
exceptionType->forAll(conformsTo(exceptionInput.type.oclassType(Classifier)))
```

## ExecutableNode [Abstract Class]

### Description

An `ExecutableNode` is an abstract class for `ActivityNodes` whose execution may be controlled using `ControlFlows` and to which `ExceptionHandler`s may be attached.

### Diagrams

[Executable Nodes](#), [Actions](#), [Structured Actions](#)

### Generalizations

[ActivityNode](#)

### Specializations

[Action](#)

### Association Ends

- ♦ handler : [ExceptionHandler](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [ExceptionHandler::protectedNode](#))  
A set of `ExceptionHandler`s that are examined if an exception propagates out of the `ExceptionNode`.

## FinalNode [Abstract Class]

### Description

A `FinalNode` is an abstract `ControlNode` at which a flow in an `Activity` stops.

### Diagrams

[Control Nodes](#)

### Generalizations

[ControlNode](#)

### Specializations

[ActivityFinalNode](#), [FlowFinalNode](#)

### Constraints

- `no_outgoing_edges`  
A `FinalNode` has no outgoing `ActivityEdges`.

```
inv: outgoing->isEmpty()
```

## FlowFinalNode [Class]

### Description

A FlowFinalNode is a FinalNode that terminates a flow by consuming the tokens offered to it.

### Diagrams

[Control Nodes](#)

### Generalizations

[FinalNode](#)

## ForkNode [Class]

### Description

A ForkNode is a ControlNode that splits a flow into multiple concurrent flows.

### Diagrams

[Control Nodes](#)

### Generalizations

[ControlNode](#)

### Constraints

- edges  
The ActivityEdges incoming to and outgoing from a ForkNode must be either all ObjectFlows or all ControlFlows.

```
inv: let allEdges : Set(ActivityEdge) = incoming->union(outgoing) in  
allEdges->forAll(oclIsKindOf(ControlFlow)) or allEdges->forAll(oclIsKindOf(ObjectFlow))
```

- one\_incoming\_edge  
A ForkNode has one incoming ActivityEdge.

```
inv: incoming->size()==1
```

## InitialNode [Class]

### Description

An InitialNode is a ControlNode that offers a single control token when initially enabled.

### Diagrams

[Control Nodes](#)

## Generalizations

[ControlNode](#)

## Constraints

- `no_incoming_edges`  
An InitialNode has no incoming ActivityEdges.
- `control_edges`  
All the outgoing ActivityEdges from an InitialNode must be ControlFlows.

```
inv: incoming->isEmpty()
```

```
inv: outgoing->forAll(oclIsKindOf(ControlFlow))
```

## InterruptibleActivityRegion [Class]

### Description

An InterruptibleActivityRegion is an ActivityGroup that supports the termination of tokens flowing in the portions of an activity within it.

### Diagrams

[Activity Groups](#)

### Generalizations

[ActivityGroup](#)

### Association Ends

- `interruptingEdge` : [ActivityEdge](#) [0..\*] (opposite [ActivityEdge::interrupts](#))  
The ActivityEdges leaving the InterruptibleActivityRegion on which a traversing token will result in the termination of other tokens flowing in the InterruptibleActivityRegion.
- `node` : [ActivityNode](#) [0..\*]{subsets [ActivityGroup::containedNode](#)} (opposite [ActivityNode::inInterruptibleRegion](#))  
ActivityNodes immediately contained in the InterruptibleActivityRegion.

### Constraints

- `interrupting_edges`  
The interruptingEdges of an InterruptibleActivityRegion must have their source in the region and their target outside the region, but within the same Activity containing the region.

```
inv: interruptingEdge->forAll(edge |  
  node->includes(edge.source) and node->excludes(edge.target) and edge.target.containingActivity() =  
  inActivity)
```



## JoinNode [Class]

### Description

A JoinNode is a ControlNode that synchronizes multiple flows.

### Diagrams

[Control Nodes](#)

### Generalizations

[ControlNode](#)

### Attributes

- isCombineDuplicate : [Boolean](#) [1..1] = true  
Indicates whether incoming tokens having objects with the same identity are combined into one by the JoinNode.

### Association Ends

- ♦ joinSpec : [ValueSpecification](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_joinSpec\\_joinNode::joinNode](#))  
A ValueSpecification giving the condition under which the JoinNode will offer a token on its outgoing ActivityEdge. If no joinSpec is specified, then the JoinNode will offer an outgoing token if tokens are offered on all of its incoming ActivityEdges (an "and" condition).

### Constraints

- one\_outgoing\_edge  
A JoinNode has one outgoing ActivityEdge.  

```
inv: outgoing->size() = 1
```
- incoming\_object\_flow  
If one of the incoming ActivityEdges of a JoinNode is an ObjectFlow, then its outgoing ActivityEdge must be an ObjectFlow. Otherwise its outgoing ActivityEdge must be a ControlFlow.  

```
inv: if incoming->exists(oclIsKindOf(ObjectFlow)) then outgoing->forall(oclIsKindOf(ObjectFlow))
else outgoing->forall(oclIsKindOf(ControlFlow))
endif
```

## MergeNode [Class]

### Description

A merge node is a control node that brings together multiple alternate flows. It is not used to synchronize concurrent flows but to accept one among several alternate flows.

## Diagrams

[Control Nodes](#)

## Generalizations

[ControlNode](#)

## Constraints

- **one\_outgoing\_edge**  
A MergeNode has one outgoing ActivityEdge.  
  
`inv: outgoing->size()=1`
- **edges**  
The ActivityEdges incoming to and outgoing from a MergeNode must be either all ObjectFlows or all ControlFlows.

```
inv: let allEdges : Set(ActivityEdge) = incoming->union(outgoing) in  
allEdges->forAll(oclIsKindOf(ControlFlow)) or allEdges->forAll(oclIsKindOf(ObjectFlow))
```

## ObjectFlow [Class]

### Description

An ObjectFlow is an ActivityEdge that is traversed by object tokens that may hold values. Object flows also support multicast/receive, token selection from object nodes, and transformation of tokens.

### Diagrams

[Activities](#), [Control Nodes](#)

### Generalizations

[ActivityEdge](#)

### Attributes

- **isMulticast** : [Boolean](#) [1..1] = false  
Indicates whether the objects in the ObjectFlow are passed by multicasting.
- **isMultireceive** : [Boolean](#) [1..1] = false  
Indicates whether the objects in the ObjectFlow are gathered from respondents to multicasting.

### Association Ends

- **selection** : [Behavior](#) [0..1] (opposite [A\\_selection\\_objectFlow::objectFlow](#))  
A Behavior used to select tokens from a source ObjectNode.

- transformation : [Behavior](#) [0..1] (opposite [A transformation objectFlow::objectFlow](#))  
A Behavior used to change or replace object tokens flowing along the ObjectFlow.

## Constraints

- input\_and\_output\_parameter  
A selection Behavior has one input Parameter and one output Parameter. The input Parameter must have the same as or a supertype of the type of the source ObjectNode, be non-unique and have multiplicity 0..\*. The output Parameter must be the same or a subtype of the type of source ObjectNode. The Behavior cannot have side effects.

```
inv: selection<>null implies
    selection.inputParameters()->size()=1 and
    selection.inputParameters()->forAll(not isUnique and is(0,*)) and
    selection.outputParameters()->size()=1
```

- no\_executable\_nodes  
ObjectFlows may not have ExecutableNodes at either end.

```
inv: not (source.oclIsKindOf(ExecutableNode) or target.oclIsKindOf(ExecutableNode))
```

- transformation\_behavior  
A transformation Behavior has one input Parameter and one output Parameter. The input Parameter must be the same as or a supertype of the type of object token coming from the source end. The output Parameter must be the same or a subtype of the type of object token expected downstream. The Behavior cannot have side effects.

```
inv: transformation<>null implies
    transformation.inputParameters()->size()=1 and
    transformation.outputParameters()->size()=1
```

- selection\_behavior  
An ObjectFlow may have a selection Behavior only if it has an ObjectNode as its source.

```
inv: selection<>null implies source.oclIsKindOf(ObjectNode)
```

- compatible\_types  
ObjectNodes connected by an ObjectFlow, with optionally intervening ControlNodes, must have compatible types. In particular, the downstream ObjectNode type must be the same or a supertype of the upstream ObjectNode type.

Cannot be expressed in OCL

- same\_upper\_bounds  
ObjectNodes connected by an ObjectFlow, with optionally intervening ControlNodes, must have the same upperBounds.

Cannot be expressed in OCL

- target  
An ObjectFlow with a constant weight may not target an ObjectNode, with optionally intervening ControlNodes, that has an upper bound less than the weight.

Cannot be expressed in OCL

- `is_multicast_or_is_multireceive`  
`isMulticast` and `isMultireceive` cannot both be true.

```
inv: not (isMulticast and isMultireceive)
```

## ObjectNode [Abstract Class]

### Description

An `ObjectNode` is an abstract `ActivityNode` that may hold tokens within the object flow in an `Activity`. `ObjectNodes` also support token selection, limitation on the number of tokens held, specification of the state required for tokens being held, and carrying control values.

### Diagrams

[Object Nodes](#), [Executable Nodes](#), [Actions](#), [Expansion Regions](#)

### Generalizations

[TypedElement](#), [ActivityNode](#)

### Specializations

[ActivityParameterNode](#), [CentralBufferNode](#), [ExpansionNode](#), [Pin](#)

### Attributes

- `isControlType` : [Boolean](#) [1..1] = false  
 Indicates whether the type of the `ObjectNode` is to be treated as representing control values that may traverse `ControlFlows`.
- `ordering` : [ObjectNodeOrderingKind](#) [1..1] = FIFO  
 Indicates how the tokens held by the `ObjectNode` are ordered for selection to traverse `ActivityEdges` outgoing from the `ObjectNode`.

### Association Ends

- `inState` : [State](#) [0..\*] (opposite [A\\_inState\\_objectNode::objectNode](#))  
 The States required to be associated with the values held by tokens on this `ObjectNode`.
- `selection` : [Behavior](#) [0..1] (opposite [A\\_selection\\_objectNode::objectNode](#))  
 A `Behavior` used to select tokens to be offered on outgoing `ActivityEdges`.
- ♦ `upperBound` : [ValueSpecification](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_upperBound\\_objectNode::objectNode](#))  
 The maximum number of tokens that may be held by this `ObjectNode`. Tokens cannot flow into the `ObjectNode` if the `upperBound` is reached. If no `upperBound` is specified, then there is no limit on how many tokens the `ObjectNode` can hold.

## Constraints

- **input\_output\_parameter**  
A selection Behavior has one input Parameter and one output Parameter. The input Parameter must have the same type as or a supertype of the type of ObjectNode, be non-unique, and have multiplicity 0..\*. The output Parameter must be the same or a subtype of the type of ObjectNode. The Behavior cannot have side effects.

```
inv: selection<>null implies
    selection.inputParameters()->size()=1 and
    selection.inputParameters()->forall(p | not p.isUnique and p.is(0,*) and
    self.type.conformsTo(p.type)) and
    selection.outputParameters()->size()=1 and
    selection.inputParameters()->forall(p | self.type.conformsTo(p.type))
```

- **selection\_behavior**  
If an ObjectNode has a selection Behavior, then the ordering of the object node is ordered, and vice versa.

```
inv: (selection<>null) = (ordering=ObjectNodeOrderingKind::ordered)
```

- **object\_flow\_edges**  
If isControlType=false, the ActivityEdges incoming to or outgoing from an ObjectNode must all be ObjectFlows.

```
inv: (not isControlType) implies incoming->union(outgoing)->forall(oclIsKindOf(ObjectFlow))
```

## ObjectNodeOrderingKind [Enumeration]

### Description

ObjectNodeOrderingKind is an enumeration indicating queuing order for offering the tokens held by an ObjectNode.

### Diagrams

- [Object Nodes](#)

### Literals

- **unordered**  
Indicates that tokens are unordered.
- **ordered**  
Indicates that tokens are ordered.
- **LIFO**  
Indicates that tokens are queued in a last in, first out manner.
- **FIFO**  
Indicates that tokens are queued in a first in, first out manner.

## Variable [Class]

### Description

A Variable is a ConnectableElement that may store values during the execution of an Activity. Reading and writing the values of a Variable provides an alternative means for passing data than the use of ObjectFlows. A Variable may be owned directly by an Activity, in which case it is accessible from anywhere within that activity, or it may be owned by a StructuredActivityNode, in which case it is only accessible within that node.

### Diagrams

[Activities](#), [Variable Actions](#), [Structured Actions](#)

### Generalizations

[ConnectableElement](#), [MultiplicityElement](#)

### Attributes

### Association Ends

- activityScope : [Activity](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [Activity::variable](#))  
An Activity that owns the Variable.
- scope : [StructuredActivityNode](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [StructuredActivityNode::variable](#))  
A StructuredActivityNode that owns the Variable.

### Operations

- isAccessibleBy(a : [Action](#)) : [Boolean](#)  
A Variable is accessible by Actions within its scope (the Activity or StructuredActivityNode that owns it).

```
body: if scope<>null then scope.allOwnedNodes()->includes(a)
else a.containingActivity()==activityScope
endif
```

## 15.8 Association Descriptions

### A\_containedEdge\_inGroup [Association]

#### Diagrams

[Activity Groups](#)

#### Member Ends

- [ActivityGroup::containedEdge](#)
- [ActivityEdge::inGroup](#)

## A\_containedNode\_inGroup [Association]

### Diagrams

[Activity Groups](#)

### Member Ends

- [ActivityGroup::containedNode](#)
- [ActivityNode::inGroup](#)

## A\_decisionInputFlow\_decisionNode [Association]

### Diagrams

[Control Nodes](#)

### Owned Ends

- decisionNode : [DecisionNode](#) [0..1] (opposite [DecisionNode::decisionInputFlow](#))

## A\_decisionInput\_decisionNode [Association]

### Diagrams

[Control Nodes](#)

### Owned Ends

- decisionNode : [DecisionNode](#) [0..\*] (opposite [DecisionNode::decisionInput](#))

## A\_edge\_activity [Association]

### Diagrams

[Activities](#)

### Member Ends

- [Activity::edge](#)
- [ActivityEdge::activity](#)

## A\_edge\_inPartition [Association]

### Diagrams

[Activity Groups](#)

### Member Ends

- [ActivityPartition::edge](#)
- [ActivityEdge::inPartition](#)

## A\_exceptionInput\_exceptionHandler [Association]

### Diagrams

[Executable Nodes](#)

### Owned Ends

- exceptionHandler : [ExceptionHandler](#) [0..\*] (opposite [ExceptionHandler::exceptionInput](#))

## A\_exceptionType\_exceptionHandler [Association]

### Diagrams

[Executable Nodes](#)

### Owned Ends

- exceptionHandler : [ExceptionHandler](#) [0..\*] (opposite [ExceptionHandler::exceptionType](#))

## A\_group\_inActivity [Association]

### Diagrams

[Activity Groups](#)

### Member Ends

- [Activity::group](#)
- [ActivityGroup::inActivity](#)



## A\_guard\_activityEdge [Association]

### Diagrams

[Activities](#)

### Owned Ends

- activityEdge : [ActivityEdge](#) [0..1]{subsets [Element::owner](#)} (opposite [ActivityEdge::guard](#))

## A\_handlerBody\_exceptionHandler [Association]

### Diagrams

[Executable Nodes](#)

### Owned Ends

- exceptionHandler : [ExceptionHandler](#) [0..\*] (opposite [ExceptionHandler::handlerBody](#))

## A\_handler\_protectedNode [Association]

### Diagrams

[Executable Nodes](#)

### Member Ends

- [ExecutableNode::handler](#)
- [ExceptionHandler::protectedNode](#)

## A\_inInterruptibleRegion\_node [Association]

### Diagrams

[Activity Groups](#)

### Member Ends

- [ActivityNode::inInterruptibleRegion](#)
- [InterruptibleActivityRegion::node](#)

## A\_inPartition\_node [Association]

### Diagrams

[Activity Groups](#)

### Member Ends

- [ActivityNode::inPartition](#)
- [ActivityPartition::node](#)

## A\_inState\_objectNode [Association]

### Diagrams

[Object Nodes](#)

### Owned Ends

- objectNode : [ObjectNode](#) [0..\*] (opposite [ObjectNode::inState](#))

## A\_incoming\_target\_node [Association]

### Diagrams

[Activities](#)

### Member Ends

- [ActivityNode::incoming](#)
- [ActivityEdge::target](#)

## A\_interruptingEdge\_interrupts [Association]

### Diagrams

[Activity Groups](#)

### Member Ends

- [InterruptibleActivityRegion::interruptingEdge](#)
- [ActivityEdge::interrupts](#)

## A\_joinSpec\_joinNode [Association]

### Diagrams

[Control Nodes](#)

## Owned Ends

- joinNode : [JoinNode](#) [0..1]{subsets [Element::owner](#)} (opposite [JoinNode::joinSpec](#))

## A\_node\_activity [Association]

### Diagrams

[Activities](#)

### Member Ends

- [Activity::node](#)
- [ActivityNode::activity](#)

## A\_outgoing\_source\_node [Association]

### Diagrams

[Activities](#)

### Member Ends

- [ActivityNode::outgoing](#)
- [ActivityEdge::source](#)

## A\_parameter\_activityParameterNode [Association]

### Diagrams

[Object Nodes](#)

### Owned Ends

- activityParameterNode : [ActivityParameterNode](#) [0..\*] (opposite [ActivityParameterNode::parameter](#))

## A\_partition\_activity [Association]

### Diagrams

[Activity Groups](#)

## Owned Ends

- activity : [Activity](#) [0..1]{subsets [ActivityGroup::inActivity](#)} (opposite [Activity::partition](#))

## A\_redefinedEdge\_activityEdge [Association]

### Diagrams

[Activities](#)

## Owned Ends

- activityEdge : [ActivityEdge](#) [0..\*]{subsets [A\\_redefinedElement\\_redefinableElement::redefinableElement](#)} (opposite [ActivityEdge::redefinedEdge](#))

## A\_redefinedNode\_activityNode [Association]

### Diagrams

[Activities](#)

## Owned Ends

- activityNode : [ActivityNode](#) [0..\*]{subsets [A\\_redefinedElement\\_redefinableElement::redefinableElement](#)} (opposite [ActivityNode::redefinedNode](#))

## A\_represents\_activityPartition [Association]

### Diagrams

[Activity Groups](#)

## Owned Ends

- activityPartition : [ActivityPartition](#) [0..\*] (opposite [ActivityPartition::represents](#))

## A\_selection\_objectFlow [Association]

### Diagrams

[Activities](#)

### Owned Ends

- objectFlow : [ObjectFlow](#) [0..\*] (opposite [ObjectFlow::selection](#))

## A\_selection\_objectNode [Association]

### Diagrams

[Object Nodes](#)

### Owned Ends

- objectNode : [ObjectNode](#) [0..\*] (opposite [ObjectNode::selection](#))

## A\_structuredNode\_activity [Association]

### Diagrams

[Structured Actions](#)

### Member Ends

- [Activity::structuredNode](#)
- [StructuredActivityNode::activity](#)

## A\_subgroup\_superGroup [Association]

### Diagrams

[Activity Groups](#)

### Member Ends

- [ActivityGroup::subgroup](#)
- [ActivityGroup::superGroup](#)

## A\_subpartition\_superPartition [Association]

### Diagrams

[Activity Groups](#)

### Member Ends

- [ActivityPartition::subpartition](#)
- [ActivityPartition::superPartition](#)

## A\_transformation\_objectFlow [Association]

### Diagrams

[Activities](#)

### Owned Ends

- objectFlow : [ObjectFlow](#) [0..\*] (opposite [ObjectFlow::transformation](#))

## A\_upperBound\_objectNode [Association]

### Diagrams

[Object Nodes](#)

### Owned Ends

- objectNode : [ObjectNode](#) [0..1]{subsets [Element::owner](#)} (opposite [ObjectNode::upperBound](#))

## A\_variable\_activityScope [Association]

### Diagrams

[Activities](#)

### Member Ends

- [Activity::variable](#)
- [Variable::activityScope](#)

## A\_weight\_activityEdge [Association]

### Diagrams

[Activities](#)

### Owned Ends

- activityEdge : [ActivityEdge](#) [0..1]{subsets [Element::owner](#)} (opposite [ActivityEdge::weight](#))

# 16 Actions

## 16.1 Summary

An *Action* is the fundamental unit of behavior specification in UML. An Action may take a set of inputs and produce a set of outputs, though either or both of these sets may be empty. Some Actions may modify the state of the system in which the Action executes.

Actions are contained in Behaviors, specifically Activities (as ExecutableNodes, see Clause [15](#)) and Interactions (see Clause [17](#)). These Behaviors determine when Actions execute and what inputs they have. However, the abstract syntax and semantics of Actions are very dependent on Activities, because they specialize elements and semantics from Activities to accept inputs and provide outputs and to define Actions that coordinate other Actions (Structured Actions, see sub clause 16.11). In addition, the concrete syntax for Actions only appears in Activity diagrams (all the examples in this Clause use Activity notation), and some of the notation for Actions is specified Clause 15. This Clause focuses on the syntax and semantics of Actions specifically, rather than the Behaviors that contain them, but must be read in conjunction with Clause 15.

### Concrete Syntax

The UML specification provides a relatively minimal set of graphical notations for Actions. However, conforming tools may provide tool-specific graphical or textual representations that *map* into the standard Action abstract syntax. Such representations are referred to as *concrete syntaxes*. For example, a textual concrete syntax may be used to notate Actions in Behaviors attached to Transitions in StateMachines (see sub clause [14.2.4](#)). Concrete syntaxes generally encompass both primitive Actions and the control mechanisms provided by Behaviors.

Concrete syntaxes can map higher-level constructs to the Actions specified in the UML abstract syntax. For example, creating an object may involve initializing attribute values or creating objects for mandatory Associations. The UML specification defines the CreateObjectAction to only create the object and requires further Actions to initialize attribute values and create objects for mandatory Associations. A concrete syntax can support a creation operation including initialization as a single unit as a shorthand for several underlying Actions. In general, concrete syntaxes can implement each Action one-to-one, or define higher-level, composite constructs to offer the modeler more power and convenience. This specification provides abstract syntax and semantics for primitive behavioral concepts that are simple to understand and implement. Modelers can work in terms of higher-level constructs as provided by their chosen concrete syntax.

The most primitive Actions in this specification are defined to enable the maximum range of concrete syntax mappings. Specifically, a primitive Action either carries out a computation or accesses object memory, but never both. This approach enables clean mappings to a physical implementation, even those with data organizations different from that suggested by the specification. In addition, any re-organization of the data structure will leave the specification of the computation unaffected.

### Execution Engines

An *execution engine* is a tool that executes UML Actions. Actions are defined to enable the construction of various execution engines with different performance characteristics. An execution engine can optimize the execution of a model to meet specific performance requirements, so long as the engine stays within the semantics specified for Actions in UML. For example, one execution engine might operate fully sequentially within a single task, while another might assign classes to different processors based on how much they interact, and yet others might assign classes in a client-server, or even a three-tier architecture. These are all valid execution engines for UML if they are optimized only to the extent allowed by UML semantics.

Modelers can provide “hints” to the execution engine when they have special knowledge of the domain solution that could be of value in optimizing the execution. For example, instances could—by design—be partitioned to match the assignment of classes, so tests based on this partitioning can be optimized on each processor. An execution engine is not required to check or enforce such hints. It can either assume that the modeler is correct, or just ignore it. An execution engine is not required to verify that the modeler’s assertion is true.

When the execution of Actions violates aspects of UML structural semantics that constrain runtime behavior, the semantics of the Actions are left undefined. For example, linking an instance to multiple owners via composite Associations is undefined – some concrete syntaxes may make this Action illegal, others may allow it until a single owner is established. The semantics are also left undefined in situations that require Classes as values at runtime. However, in the execution of Actions the lower multiplicity bound is ignored and the semantics is still as defined. Otherwise, it is impossible to use Actions to pass through intermediate stages necessary to construct object configurations that satisfy multiplicity constraints. The modeler must determine the points at which minimum multiplicity will be enforced, and these points cannot be everywhere or the object configuration would not be able to change.

## 16.2 Actions

### 16.2.1 Summary

This subclause defines the basic abstract syntax for Actions and Pins, where Pins are used to specify the inputs and outputs for Actions. Other than `OpaqueAction`, the various concrete kinds of Actions are described in subsequent subclauses.

### 16.2.2 Abstract Syntax

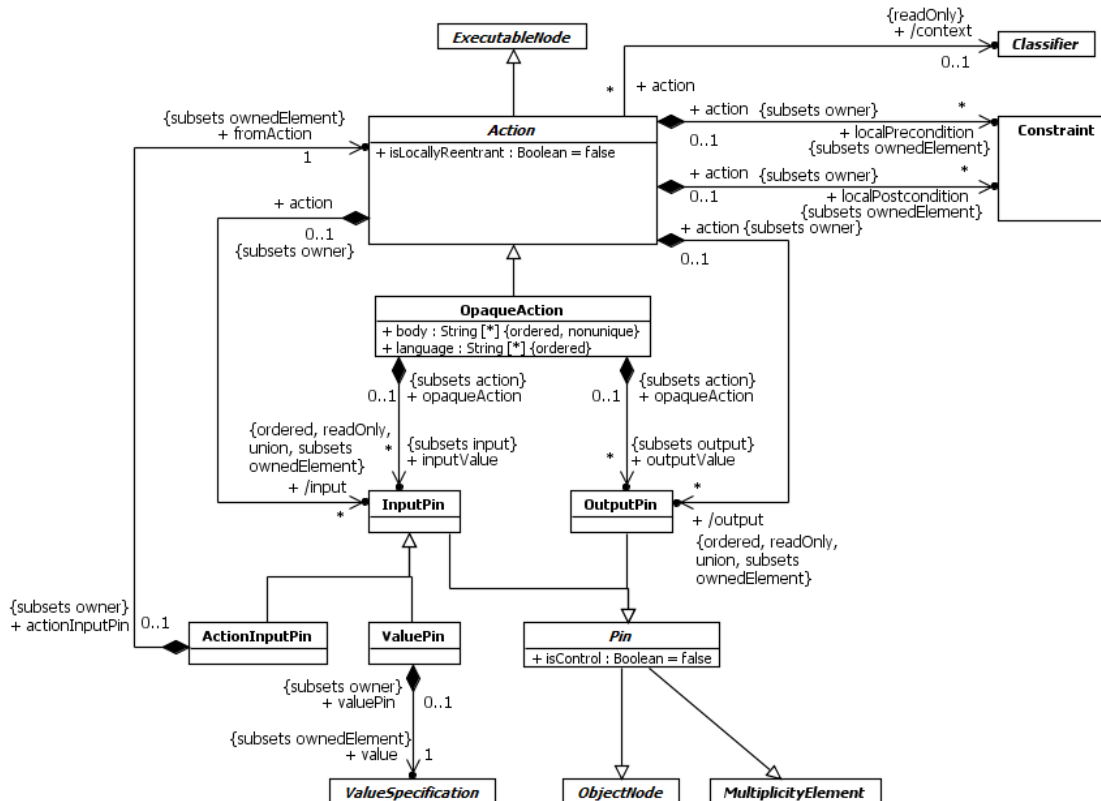


Figure 16.1 Actions



## 16.2.3 Semantics

### Actions

An Action is a fundamental unit of executable functionality contained, directly or indirectly, within a Behavior. The execution of an Action represents some transformation or processing in the modeled system, be it a computer system or otherwise. However, an Action execution may also result in the invocation of another Behavior (see sub clause 16.3 on Invocation Actions). An Action is therefore simple from the point of view of the Behavior containing it but may be complex in its effect and not atomic.

If the Behavior containing an Action has a context BehavioredClassifier (see sub clause 13.2), then that BehavioredClassifier is also the context Classifier for the Action. When the Action executes, it executes in the context of an instance of the context Classifier or, if there is no context Classifier, then directly in the context of the execution instance of the Behavior. (See also the discussion of context objects under BehavioredClassifiers in sub clause 13.2.3.)

An Action may accept inputs and produce outputs, as specified by InputPins and OutputPins of the Action, respectively. Each Pin on an Action specifies the type and multiplicity for a specific input or output of that Action.

The time at which an Action executes and what inputs are accepted by each execution are determined by the kind of Action it is, characteristics of its InputPins, and the Behavior in which it is used. Once it has been determined that an Action will execute, the general steps for that execution are as follows:

1. The Action execution consumes input data on all InputPins on the Action up to the upper multiplicity for each InputPin. For structured Actions (StructuredActivityNodes, see sub clause 16.11), data can remain on InputPins during Action execution, otherwise they are immediately removed from the InputPins by the ActionExecution. If the Action is an invocation of a Behavior with streaming Parameters (see sub clause 13.2.3), then the Action execution may consume additional data supplied to InputPins corresponding to streaming input Parameters (see sub clause 16.3.3 on the semantics of InvocationActions). Otherwise, once an Action execution has started, any additional data on InputPins has no effect on it.
2. An Action continues executing until it has completed. The detailed semantics of the execution of an Action and the definition of its completion depend on the particular kind of Action being executed.
3. When completed, an Action execution provides any output data on the OutputPins of the Action, and it terminates. However, if the Action is an invocation of a Behavior with streaming Parameters (see sub clause 13.2.3), then the Action execution may also post data to OutputPins corresponding to streaming output Parameters before completion of the execution (see sub clause 16.3.3 on the semantics of InvocationActions).

**NOTE.** After the execution of an Action has terminated, any resources used in executing it may be reclaimed by an execution engine implementing its execution. However, the details of resource management are implementation-specific and not defined in this specification.

In the specification of semantics for Actions, Behaviors may reuse the same kind of Action multiple times, and the semantics of the Action applies to each usage separately. For example, Activities may have multiple nodes that are instances of the same Action metaclass, and each instance has its own values for the properties described below, affecting the execution of that particular usage of the Action only. The same applies to Interactions with multiple actions that are instances of the same Action metaclass. The phrase “an Action” or “the Action” in this specification refers to a single instance of an Action metaclass used in a Behavior, separate from the other instances of the same Action metaclass that may be used in the Behavior or other Behaviors.

If an Action is not locally reentrant (`isLocallyReentrant=false`, the default), then no more than one execution of it may exist at any given time within a single execution of the containing Behavior. Even if the Action would normally begin executing, the new execution may not start if there is already one ongoing within the same Behavior execution, but the new execution may start when the current execution of the Action terminates. On the other hand, if an Action is locally reentrant (`isLocallyReentrant=true`), then a new execution of it may begin any time the normal rules above allow it, even if there are one or more executions already ongoing within the same Behavior execution. This means that there may be, within any one execution of the containing Behavior, more than one concurrent execution of the Action ongoing at any given time.

A CallAction for a non-reentrant Behavior (`isReentrant=true`) will also act as if the CallAction were locally non-reentrant, whatever the value of the `isLocallyReentrant` property for the action. Moreover, such an Action will not execute if there is any currently running execution for the Behavior, whether invoked by this Action or any other within the same containing Behavior or in any other (see the semantics of `isReentrant` in sub clause [13.2.3](#) and of CallAction in sub clause [16.3.3](#)).

The `localPrecondition` and `localPostcondition` for an Action are Constraints that should hold when an execution of an Action starts and completes, respectively. As a `localPrecondition` or `localPostcondition` is a modeler-defined Constraint, violations do not mean that the UML semantics of the Action execution are undefined. They only mean that the execution trace for the Behavior containing the Action does not conform to the modeler's intention (although in most cases this indicates a serious modeling error that calls into question the validity of the model). Specifically how a `localPrecondition` or `localPostcondition` is enforced is not defined in this specification. An execution engine may detect violations statically, if possible, or at runtime. The runtime effect of a violation may be an error that stops execution, just a warning, or no effect at all, as determined by the execution engine.

### Opaque Actions

An OpaqueAction is an Action whose specification may be given in a textual concrete syntax other than UML. An OpaqueAction may also be used as a temporary placeholder before some other kind of Action is chosen.

An OpaqueAction has a body that consists of a sequence of text Strings representing alternative means of specifying the behavior of the Action. A corresponding sequence of language Strings may be used to specify the languages in which each of the body Strings is to be interpreted. Languages are matched to body Strings by order. The UML specification does not define how body Strings are interpreted relative to any language, though other specifications may define specific language Strings to be used to indicate interpretation with respect to those specifications. It is not required to specify the languages. If they are unspecified, then the interpretation of any body Strings shall be determined implicitly from the form of the bodies or how the OpaqueAction is used.

If an OpaqueAction has more than one body String, then any one of the bodies can be used to determine the behavior of the OpaqueAction. The UML specification does not determine how this choice is made.

### Pins

A Pin represents an input to an Action or an output from an Action. An InputPin represents an input, while an OutputPin represents an output. Each of the sets of inputs and outputs owned by an Action are ordered. The InputPins and OutputPins of an Action are determined by the kind of Action it is.

A Pin is a kind of ObjectNode (see sub clause [15.4](#)), so it holds object tokens that contain values of a specified Type (see sub clause [15.2](#) about tokens). Values held in the tokens of the InputPins of an Action provide the input data for executions of the Action, and the output data from Action executions are placed on the OutputPins of the Action wrapped in object tokens. A Pin is also a MultiplicityElement. The multiplicity bounds on a Pin constrain the total number of values that may be input or output by a single execution of an Action, not the number of tokens it contains (see the `upperBound` property inherited from ObjectNode). A Pin may hold null tokens that contain no values. Pin multiplicity is not unique because it may hold multiple tokens with the same value.

Pin inherits both an `ordering` attribute from ObjectNode and an `isOrdered` attribute from MultiplicityElement. The values of these attributes may be set independently. However, if `isOrdered` is true, then the ordering of values on the Pin considered as a MultiplicityElement is the order in which the values were placed onto the Pin. The value of the `ordering` attribute, however, determines the order in which values are taken from the Pin. For example, if `isOrdered` is true and the `ordering` is FIFO, then values will be taken from the Pin in the same order as the MultiplicityElement ordering. However, if the `ordering` is LIFO, then values will be taken from the pin in reverse order to the MultiplicityElement ordering. On the other hand, if `isOrdered` is false, then the order in which values are placed on the pin is indeterminate, and the effect of different orderings is not defined.

An InputPin is a Pin that holds input values to be consumed by its Action. An Action cannot start execution if one of its InputPins has fewer values than the lower multiplicity of that InputPin. The upper multiplicity determines the maximum number of values that can be consumed from an InputPin by a single execution of its Action. Tokens consumed by an Action are immediately removed from its InputPins when the action begins an execution (except in some cases for StructuredActivityNodes, where tokens may remain on InputPins during the Action execution – see sub clause [16.11](#)).

An OutputPin is a Pin that holds output values produced by an Action. For each execution, an Action cannot terminate itself unless it can put at least as many values into its outputs as required by the multiplicity lower bounds on those OutputPins. Values that may remain on the OutputPins from previous executions are not included in meeting this minimum multiplicity requirement. An Action may not put more values into an output in a single execution than the multiplicity of that OutputPin.

ValuePins and ActionInputPins are InputPins, but are not used in the determination of whether an Action is enabled for execution. If an Action has no other way to start execution, simply having ValuePins or ActionInputPins for its inputs will not enable execution of the Action. When the Action is enabled by other means, values are computed as specified for the ValuePins and ActionInputPins owned by an Action, and the results are provided as inputs to the Action when it begins execution.

A ValuePin provides a value by evaluating a ValueSpecification (e.g., this may be used as a simple way to specify constant inputs to an Action.) When the Action is enabled by other means, the ValueSpecification of the ValuePin is evaluated, and the result is provided as an input to the Action when it begins execution.

An ActionInputPin provides values by executing another Action. When an Action is enabled by other means, the fromActions on any ActionInputPins owned by the Action are also enabled. The fromActions must execute before the Action owning the ActionInputPins, and the outputs of the fromActions are placed in the corresponding ActionInputPins. The process recurs on any ActionInputPins of the fromActions. In the case that ActionInputPins are used for all inputs, this forms a tree structure that is an Action model of nested expressions, bottoming out at Actions that have no inputs (such as ReadVariableActions or ReadSelfActions).

### Actions and Pins in Activities

If an Action (and so its Pins) are contained in an Activity, when tokens move in and out of the Pins and when the Action executes are determined by the semantics of Activities as well as Actions. For example, Activities include ControlFlows between ExecutableNodes (a generalization of Actions, see sub clause [15.5](#)), which affects when Actions may execute, and Pins are ObjectNodes that may hold tokens accepted and offered according to the semantics of ObjectFlows between ObjectNodes (see sub clause [15.4](#)).

Executing an Action in an Activity requires all of its InputPins to be offered all necessary tokens, as specified by their minimum multiplicity (except for the special cases of ActionInputPins and ValuePins, as discussed above). When the Action begins executing, all the InputPins accept tokens offered to them at once, up to the maximum multiplicity allowed on each InputPin. (InputPins cannot accept more tokens than will be consumed immediately by their Actions during a single execution. This ensures that InputPins on separate Actions competing for the same tokens do not accept any tokens they cannot immediately consume, causing deadlock or starvation as Actions wait for tokens accepted by the InputPins of other Actions but not used.) Tokens accepted by the InputPins of one Action cannot be consumed by any other Action.

When an Action in an Activity completes execution, object tokens for output data placed on its OutputPins may be offered on any outgoing ObjectFlows from those Pins (per the semantics of ObjectNodes, see sub clause [15.4](#)). In addition, control tokens shall be offered on any outgoing ControlFlows from the Action (per the semantics of ExecutableNodes, see sub clause [15.5](#)).

If an Action is not locally reentrant (`isLocallyReentrant=false`), then once it starts executing, the Action and its InputPins do not accept any tokens offered to them until the execution has finished. At this point, if the required tokens are still available, the Action may accept the offers and begin a new execution. On the other hand, if the Action is locally reentrant (`isLocallyReentrant=true`), then it may begin a new execution any time the above rules allow it.

A control Pin (with `isControl=true`) must have a control type (`isControlType=true`), so that they may be used with ControlFlows. Control Pins are ignored in the constraints that Actions place on Pins (including matching to parameters for InvocationActions – see sub clause [16.3](#)). Tokens arriving at a control InputPin have the same semantics as control tokens arriving at the Action, except that control tokens can be buffered in control Pins. Tokens are placed on control OutputPins according to the same semantics as tokens placed on ControlFlows coming out of an Actions.

## 16.2.4 Notation

This subclause specifies a graphical notation for the Actions used within Activities. This notation is optional, in that a conforming tool may use a textual concrete syntax instead. However, the notation given in this and subsequent notation subclauses within this clause is the only graphical notation for Actions conformant with this specification.

### Actions

Actions are notated as round-cornered rectangles, as shown in Figure 16.2. The name of the action or other description of it may appear in the symbol. (Specialized notations for certain specific kinds of Actions are described in subsequent subclauses.)

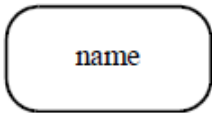


Figure 16.2 Action

Local pre- and post-conditions are shown as notes attached to the invocation with the keywords «localPrecondition» and «localPostcondition», respectively, as shown in Figure 16.3.

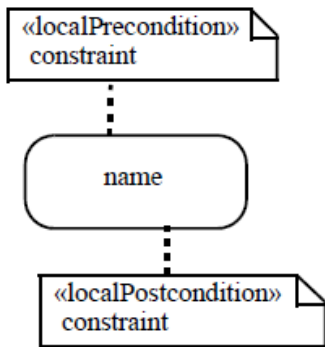


Figure 16.3 Local pre- and post-conditions

### Pins

As ObjectNodes, Pins are notated as rectangles (see sub clause 15.4.4). However, Pin rectangles may be notated as small rectangles that are attached to the symbol for the Action that owns them (see Figure 16.4). The name of the Pin may be displayed near the pin. The name is not restricted, but it often just shows the type of object or data that flows through the Pin. Both the name and type may be shown for a pin using a label of the form “name: type”. The label may also be a full specification of the Parameter corresponding to a Pin for InvocationActions, using the same textual notation as for the Parameters for BehavioralFeatures on Classes (see sub clause 16.3.4). The Pins of an Action may be elided in the notation for the Action, even though they are present in the model.



Figure 16.4 Pin notations

When ActivityEdges are not present to distinguish InputPins and OutputPins, an optional arrow may be placed inside the Pin rectangle, as shown below. InputPins have the arrow pointing toward the Action and OutputPins have the arrow pointing away from the Action.



Figure 16.5 Pin notations, with arrows

The situation in which the OutputPin of one Action is connected to the InputPin of the same name in another Action via an ObjectFlow may be shown by the optional notations of Figure 16.6. The standalone Pin in the notation maps to an OutputPin and an InputPin and one ObjectFlow edge between them in the underlying model. This form should be avoided if the Pins are not of the same type. Multiple arrows coming out of a standalone Pin rectangle is an optional notation for multiple edges coming out of an OutputPin. (See other ObjectFlow and Pin notations in sub clause [15.2](#). The specific notational variant used shall be preserved when the diagram is interchanged, see Annex [B](#).)

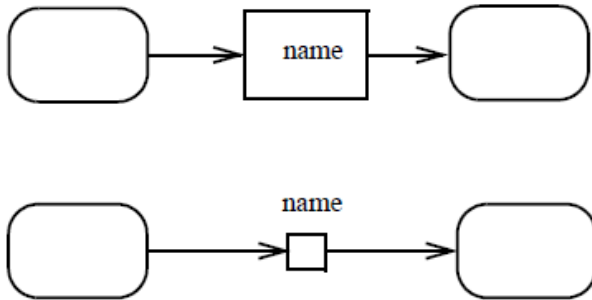


Figure 16.6 Standalone Pin notations

Control Pins are shown with the textual annotation {control} placed near the Pin symbol.

A ValuePin is notated as an InputPin with the ValueSpecification written beside it (see sub clause [8.2.4](#) on textual notation for ValueSpecifications).

## 16.2.5 Examples

### Actions

Figure 16.7 illustrates two actions. These perform behaviors called Send Payment and Accept Payment. (See other examples in sub clause [15.2](#).)

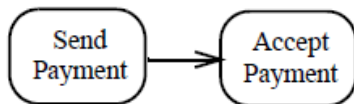


Figure 16.7 Examples of Actions

Figure 16.8 is an example of an action expressed using a tool-specific concrete syntax:

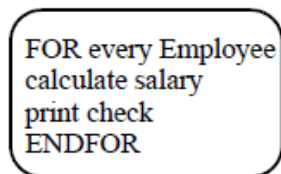
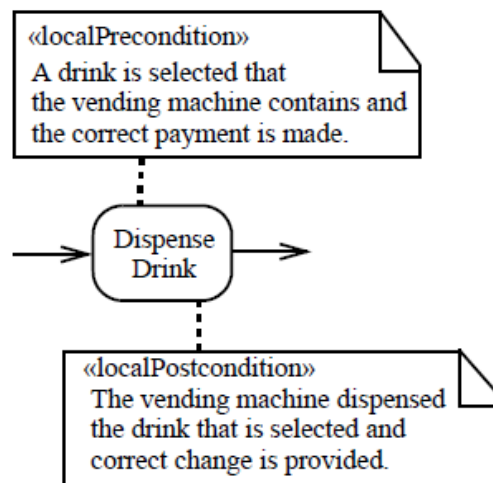


Figure 16.8 Example of action using a tool-specific concrete syntax

Figure 16.9 below illustrates local pre- and postconditions for the Action of a drink-dispensing machine. This is considered “local” because a drink-dispensing machine is constrained to operate under these conditions for this particular Action. For a machine technician scenario, the situation would be different. Here, a machine technician would have a key to open up the machine, and therefore no money need be inserted to dispense the drink, nor change need be given. In such a situation, the global pre- and post-conditions on the underlying Dispense Drink Behavior called by this action would be all that is required. For example, a global pre-condition for a Dispense Drink Behavior could be “A drink is selected that the vending machine dispenses.” The post-condition, then, would be “The vending machine dispensed the drink that was selected.” In other words, there is no global requirement for money and correct change.



**Figure 16.9 Example of an action with local pre- and post-conditions**

### Pins

In Figure 16.10, the Pin named “Order” represents Order objects. In this example at the upper left, Fill Order produces filled orders and Ship Order consumes them. The Fill Order Action must complete for Ship Order to begin. The pin symbols have been elided from the action symbols; both pins are represented by the single Order rectangle. The example on the upper right shows the same thing with explicit Pin symbols on the Actions. The example at the bottom of the figure illustrates the use of multiple pins. (See other examples in sub clause [15.2.](#))

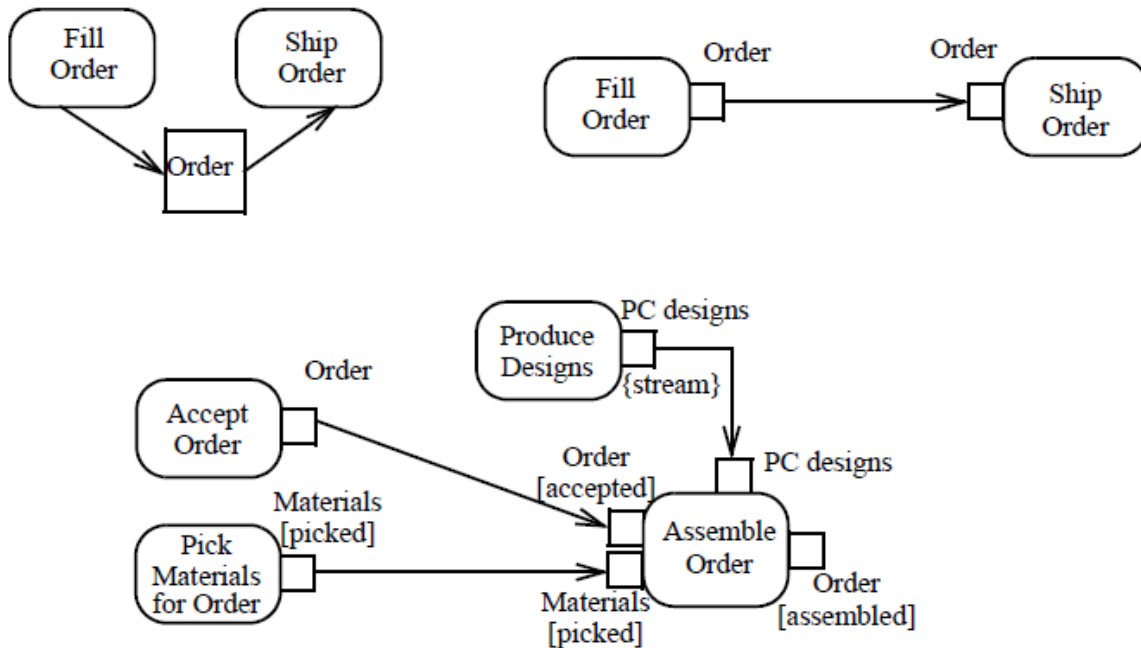


Figure 16.10 Pin examples

Figure 16.11 shows two examples of ObjectNode selection behavior (see sub clause 15.4) as applied to Pins. Both examples indicate that orders are to be shipped based on order priority—and those with the same priority should be filled on a first-in/first-out (FIFO) basis.

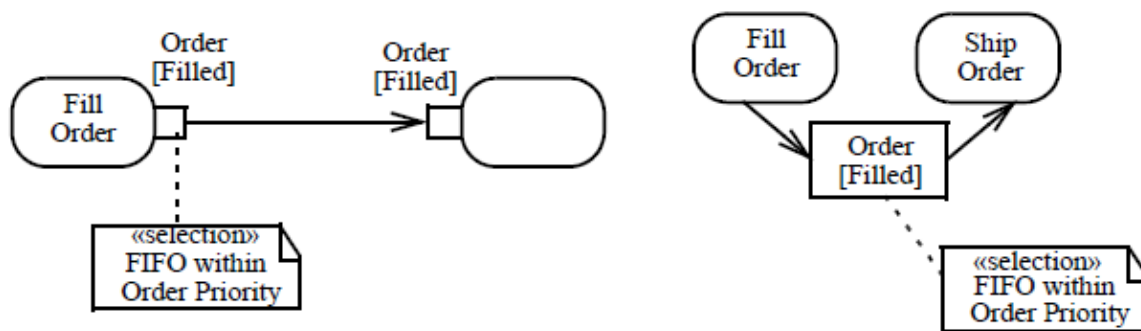


Figure 16.11 Specifying selection behavior on an ObjectFlow

For an example of the use of ActionInputPins, consider the abstract syntax mapping for the following expression in a textual concrete syntax: `self.foo->bar(self.baz)` (while this notation is OCL-like, it is not intended to be normative). The meaning of this expression is to get the value of the `foo` attribute of `self`, then send a `bar` signal to that value with an argument that is the value of the `baz` attribute of `self`. Figure 16.12 shows a UML abstract syntax representation for this expression.

**NOTE.** Subexpressions are linked using ActionInputPins.

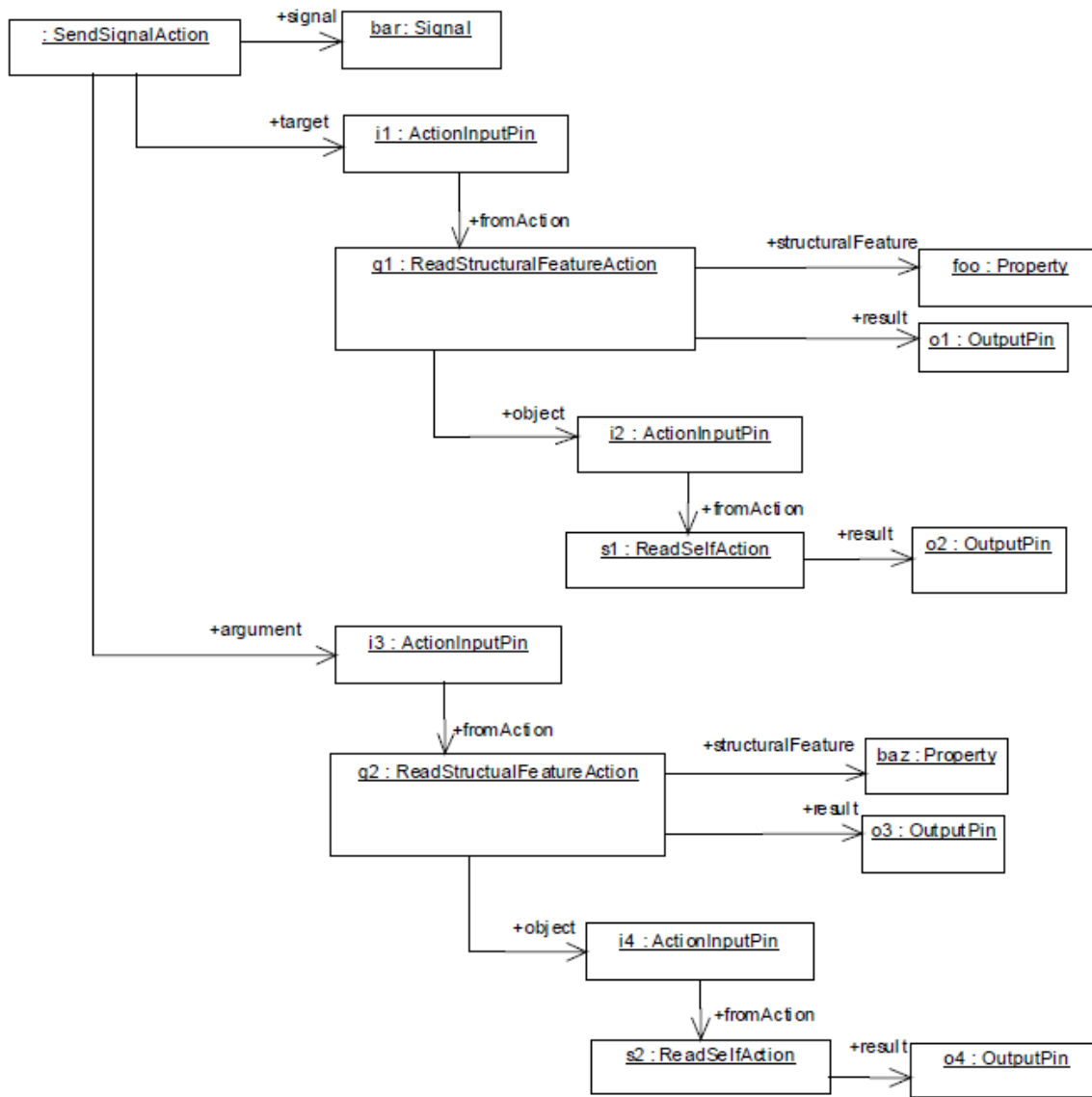


Figure 16.12 Example abstract syntax model showing the use of ActionInputPins

## 16.3 Invocation Actions

### 16.3.1 Summary

An InvocationAction is an Action that results, directly or indirectly, in the invocation of a Behavior (see sub clause 13.2). InvocationActions include the CallActions for calling Operations or Behaviors and for starting Behaviors that have been previously instantiated. Additional kinds of InvocationActions allow for the targeted sending of signals and other objects and the ability for broadcasting signals to available receivers.



## 16.3.2 Abstract Syntax

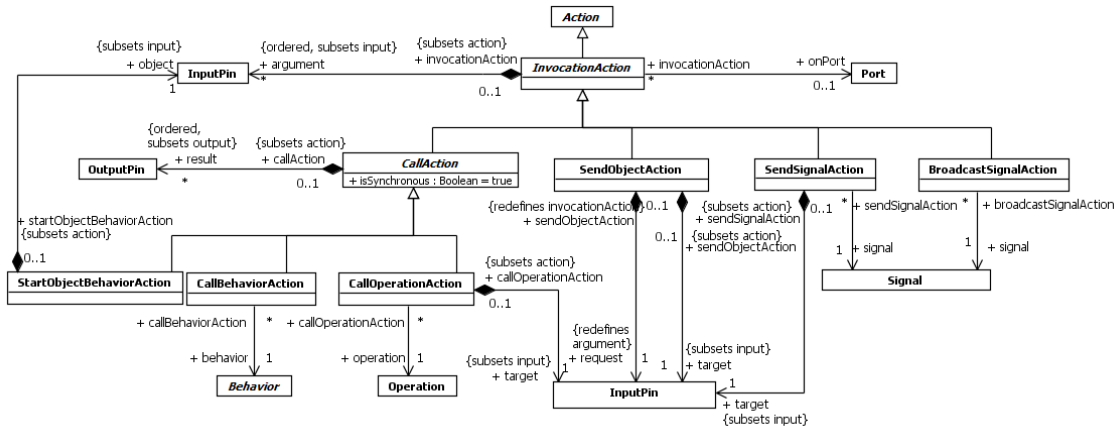


Figure 16.13 Invocation Actions

## 16.3.3 Semantics

### Call Actions

A **CallAction** is an **InvocationAction** that calls a **Behavior** or an **Operation**. There are three kinds of **CallActions**:

1. A **CallBehaviorAction** is a **CallAction** that invokes a **Behavior** directly, rather than calling a **BehavioralFeature** that, in turn, results in the invocation of a **Behavior**.
2. A **CallOperationAction** is a **CallAction** that transmits an **Operation** call request message to the target object, where it may cause the invocation of an associated **Behavior**. The target object is taken from the target **InputPin** of the **CallOperationAction**. The handling of an **Operation** call request by the target object is described under **Behavioral Features and Methods** in sub clause 13.2.3 and **Message Event** in sub clause 13.3.3.
3. A **StartObjectBehaviorAction** is a **CallAction** that starts the execution either of a directly instantiated **Behavior** or of the classifierBehavior of an object. The object to be started is taken from the object **InputPin**. If the input object is an instantiated **Behavior** that is not already executing, then it begins executing. If the input object has a classifierBehavior that is not already executing, then it is instantiated and started. In either case, if the identified **Behavior** is already executing, then the **StartObjectBehaviorAction** has no effect.

**NOTE.** If the input object is not an instantiated **Behavior**, then it must have a classifierBehavior. If the input object is an instantiated **Behavior**, then it may also have a classifierBehavior, which is also started. If this classifierBehavior itself has a classifierBehavior, then this is also recursively started, and so on.

A **CallAction** may result in either a synchronous or asynchronous **Behavior** invocation, either directly or through an **Operation** call.

- If the call is synchronous (**isSynchronous=true**), then the execution of the **Action** does not complete until the execution of the invoked **Behavior** completes, normally or otherwise. (If the **Behavior** execution does not complete normally but, instead, raises an exception, then that exception is propagated out of the **CallAction**, (see sub clauses 15.5.3 and 16.13 on exceptions and how they are handled).
- If the call is asynchronous (**isSynchronous=false**), then the execution of the **Action** completes as soon as the **Behavior** has been invoked. When an asynchronous call is complete, execution of the **Behavior** containing the **CallAction** proceeds

independently of and concurrently with the execution of the invoked Behavior, including the handling of any exceptional conditions that occur while the Behavior is executing.

If the Behavior invoked by a CallAction is not reentrant, then no more than one execution of it shall exist at any given time (see sub clause [13.2.2](#)). An invocation of a non-reentrant Behavior may not start a new execution of the Behavior when the Behavior is already executing, but may start a new execution when the current execution terminates. An invocation of a reentrant Behavior may start a new execution of the Behavior, even if the Behavior is already executing. However, a CallAction with `isLocallyReentrant=false` shall not start a new execution of the Behavior if there is a Behavior execution ongoing that was started by the same Action within the same containing Behavior execution, but may start it when the current execution terminates (see also sub clause [16.2.3](#)).

Behaviors and Operations have totally ordered lists of owned Parameters, and these Parameters are matched to Pins on a CallAction using that ordering. The argument Pins of a CallAction are matched, in order, to the sublist of input Parameters, while the result Pins are matched, in order, to the sublist of output Parameters. The type, ordering, and multiplicity of the argument and result pins of a CallAction shall be the same as the corresponding Parameters.

When a CallAction executes, it passes the values on its argument Pins to the invoked Behavior or Operation on the corresponding input Parameters. If the call is synchronous, then values returned on output Parameters are placed in tokens on the corresponding result Pins of the CallAction. If an output Parameter has no values, then a null token is placed on the corresponding result Pin. If the call is asynchronous, then result values cannot be returned.

If any of the Parameters are streaming (`isStreaming=true`), then the call must be synchronous (see also the discussion of streaming Behavior parameters in sub clause [13.2.3](#)). In this case, while the invoked Behavior executes, the CallAction continues to accept tokens offered to the argument Pins corresponding to streaming input Parameters, up to the upper multiplicity of each Pin, which are consumed by the Action and posted to the executing Behavior as they arrive. Values posted to streaming output Parameters are offered on the corresponding result Pins (which can only be accepted up to the upper multiplicity of each Pin). In effect, streaming Parameters give the invoked Behavior access to token flows to and from the invoking CallAction while the Behavior is executing.

In addition to the execution rules given in sub clause [16.2.3](#) for Actions, the following rules also apply to a CallAction invoking a Behavior or Operation with streaming Parameters:

- All InputPins for the CallAction shall have been offered a number of values equal to or greater than the lower multiplicity of each Pin before the CallAction can execute, per the usual rules. If all the argument Pins of the CallAction are for streaming Parameters with a lower multiplicity greater than 0, then at least one shall have an offered value before the CallAction can execute.
- A number of values equal to or greater than the lower multiplicity of each argument Pin of the CallAction corresponding to a streaming Parameter must be accepted before the CallAction can complete execution normally.
- Before an invoked Behavior execution completes normally (no exception raised, see sub clause [16.13](#), and no values for any exception Parameters, `isException=true`), a number of values equal to at least the lower multiplicity of each result Pin of the CallAction shall be posted by the time the execution completes. (Values may be posted to result Pins corresponding to streaming output Parameters before the execution completes.) If the invoked Behavior does not complete normally, however, then result Pins may have a smaller number of values posted to them than their lower multiplicity.

Special rules also apply to a CallAction invoking a Behavior or Operation with Parameters grouped into ParameterSets.

- If the Behavior or Operation has input ParameterSets, then the rules from sub clause [16.2.3](#) on when the CallAction may execute are applied separately to the (possibly overlapping) sets of InputPins corresponding to the Parameters in each input ParameterSet. If data is available to one of these input sets sufficient to allow the CallAction to execute, then the CallAction may execute accepting tokens only on the InputPins in that input set, and the invoked Behavior or Operation

is passed this data on Parameters only from the corresponding ParameterSet. If sufficient data is available to more than one input set, then one is chosen non-deterministically.

- If the Behavior or Operation has output Parameter sets, and the CallAction completes normally, the Call Action shall produce output on only the OutputPins corresponding to one output ParameterSet, sufficient to meet the lower multiplicity of those OutputPins, and object tokens are offered to ActivityEdges outgoing from those OutputPins. No object tokens are offered from any other OutputPins, not even null tokens.

## Send Actions

A send Action is an action that transmits an object asynchronously to one or more target objects. As a send Action is always asynchronous, it may have argument inputs, but it has no result outputs. The Action completes as soon as the object is sent, whether or not it has been received yet.

There are three kinds of send Actions:

1. A `SendSignalAction` is an `InvocationAction` that creates a `Signal` instance and transmits the instance to the object given on its target `InputPin`. A `SendSignalAction` must have argument `InputPins` corresponding, in order, to each of the (owned and inherited) `Properties` of the `Signal` being sent, with the same type, ordering and multiplicity as the corresponding attribute. The values of each `Property` of the transmitted `Signal` instance are taken from the argument `InputPin` corresponding to the `Property`. The handling of the `Signal` instance by the target object is described under `Behavioral Features and Methods` in sub clause [13.2.3](#) and `Message Events` in sub clause [13.3.3](#).
2. A `BroadcastSignalAction` is an `InvocationAction` that creates a `Signal` instance using values from its argument `InputPins`, similarly to a `SendSignalAction`. However, instead of sending the `Signal` instance to a single target object, it transmits the instance potentially to all available target objects in the system. The manner of identifying the exact set of objects that are broadcast targets is not defined in this specification, however, and may be limited to some subset of all the objects that exist.
3. A `SendObjectAction` is an `InvocationAction` that transmits any kind of object to the object given on its target `InputPin`. The object to be transmitted is given on the single request `InputPin` of the `SendObjectAction`. If the object is a `Signal` instance, then it may be handled by the target object in the same way as an instance sent from a `SendSignalAction` or `BroadcastSignalAction`. Otherwise, the reception of the object can only be handled using an `AnyReceiveEvent` (as described under `Message Events` in sub clause [13.3.3](#)).

For `SendSignalActions` and `BroadcastSignalActions`, argument `InputPins` are matched to `Properties` of the `Signal` being sent by order. The owned `Properties` of a `Signal` are ordered, but a `Signal` may also inherit `Properties` from other `Signals` due to `Generalization` relationships. In this case, the `Properties` of a `Signal` are ordered such that all owned `Properties` come before any inherited `Properties`. Further, if two ancestors of the `Signal` are related directly or indirectly by `Generalization` relationships, then the owned `Properties` of the more specific `Signal` are ordered before the owned `Properties` of the more general `Signal`. However, in the presence of multiple `Generalization`, some ancestors of the `Signal` may not have any such transitive `Generalization` relationship, and no standard ordering is defined between the `Properties` of such ancestors.

The target object(s) of a send Action may be local or remote. The transmitted object may be copied during transmission, so identity may not be preserved. The manner of transmitting the object, the amount of time required to transmit it, the order in which the transmissions reach various target objects and the path for reaching the target objects are all undefined.

## Invocation Actions and Ports

A `CallOperationAction`, `SendSignalAction`, or `SendObjectAction` may send a request through a `Port` by targeting an object having the `Port` and identifying the `Port` with the Action's `onPort` attribute. Other kinds of `InvocationActions` shall not have a value for the `onPort` attribute.

If `onPort` is given, then the `Port` shall be an owned or inherited feature of the type of the target `InputPin` of the Action. When the Action executes, rather than sending a message to the target object itself, it sends the message through the given `Port` of the target

object, which is then handled as described in sub clause [11.3.3](#). Such a message may be sent *into* the target object from the outside, through a provided Interface of the Port, or it may be sent *out of* the target object from the inside, through a required Interface of the Port.

An Action is said to execute *inside* an object if the context object of the Behavior execution within which the Action is executing is either the same as or is directly or indirectly owned (in the sense of transitive composition links) by the given object. This includes, for example, when the Action is executed in a method or classifierBehavior of the given object or in a method or classifierBehavior of a part of that object.

In the case of a CallOperationAction, a provided or required Interface of the given Port shall have the called Operation as a feature. In the case of a SendSignalAction or a SendObjectAction whose object InputPin has a Signal as its type, a provided or required Interface of the given Port may have a Reception for the identified Signal, but this is not required. In either case, the relevant Operation or Reception (if any) is referred to as the *invoked BehavioralFeature* in the following rules.

- If the invoked BehavioralFeature is on a provided Interface but not on any required Interface, then, when the InvocationAction is executed, the invocation is made into the object given on the target InputPin through the given Port, and its reception is handled as described in sub clause [11.3.3](#).

**NOTE.** This allows an InvocationAction executed inside its target object to potentially send a message back into the target object through a provided Interface of one of its own Ports.

- If the invoked BehavioralFeature is on a required Interface but not on any provided Interface, then, if the InvocationAction is being executed inside the object given on the target InputPin, the invocation is forwarded out of the target object through the given Port as described in sub clause [11.3.3](#). If the InvocationAction is being executed other than inside the given target object, the semantics are undefined.
- If the invoked BehavioralFeature is on both a provided and a required Interface or if there is no invoked BehavioralFeature, then, if the InvocationAction is being executed inside the object given on the target InputPin, the invocation is made out of the target object through the given Port. Otherwise the invocation is made into the target object through the given Port.

**NOTE.** In this case, if the InvocationAction executes inside its target object, it cannot send a message back into the target object, because such a message would go out through the required Interface. However, the same effect can be achieved by having a Connector that loops from the Port in question back to that same Port.

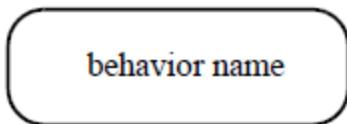
It is also possible to use an “interaction point” object (that is, an object instantiated in a Port) as the target object for a CallOperationAction, SendSignalAction, or SendObjectAction without specifying onPort. In this case, the request is sent directly to the interaction point and is then routed internally within the owner of the Port (as also described in sub clause [11.3.3](#)). The request goes into the owner of the interaction point through one of the provided Interfaces of the Port.

## 16.3.4 Notation

The value of onPort is shown by the phrase “via <port>” in the name string of the symbol denoting the particular InvocationAction.

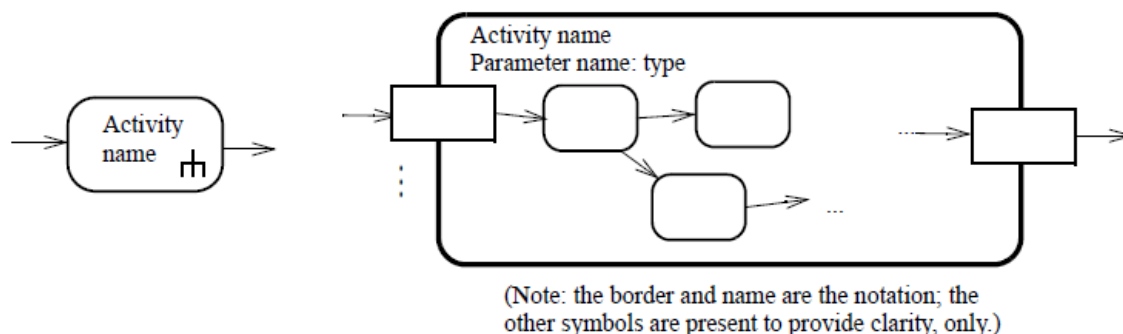
### Call Behavior Actions

A CallBehaviorAction shall be notated as an Action with the name of the invoked Behavior placed inside the Action symbol (see Figure 16.14). If the name of the Action is non-empty and different than the Behavior name, then the Action name shall appear in the symbol instead. preconditions and postconditions of the Behavior can be shown similarly to Figure 16.3, but using the keywords «precondition» and «postcondition».



**Figure 16.14 Calling a Behavior**

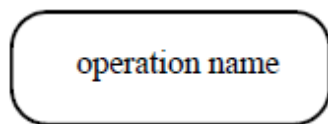
The call of an Activity is indicated by placing a rake-style symbol within the Action symbol (see Figure 16.15, left). The rake resembles a miniature hierarchy, indicating that this invocation starts another Activity that represents a further decomposition. An alternative notation in the case of an invoked Activity is to show the contents of the invoked Activity inside a large round-cornered rectangle symbol (see Figure 16.15, right). The ActivityParameterNodes are shown on the border of the invoked Activity. ObjectFlows are shown as linked to the ActivityParameterNodes in the called Activity corresponding to the Pins of the CallBehaviorAction, even though they still link to Pins in the abstract syntax. The abstract syntax is the same regardless of the choice of notation. (The specific notational variant used shall be preserved when the diagram is interchanged, see Annex B.)



**Figure 16.15 Calling an Activity**

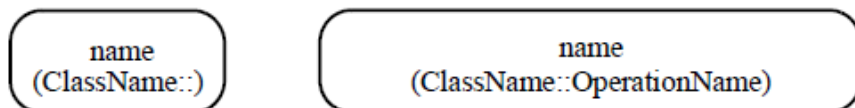
### Call Operation Actions

A CallOperationAction is notated as an Action with the name of the invoked Operation placed inside the Action symbol (see Figure 16.16). If the name of the Action is non-empty and different than the Operation name, then the Action name shall appear in the symbol instead. preconditions and postconditions on the Operation can be shown similarly to Figure 16.3, but using the keywords «precondition» and «postcondition».



**Figure 16.16 Calling an Operation**

The name of the owner of the Operation may optionally appear below the name of the Operation, in parentheses postfixed by a double colon (see Figure 16.17, left). If the Action name is shown instead of the Operation name, then the Operation name may be shown after the double colon.



**Figure 16.17 Calling an Operation, showing the owner name**

### Send Signal and Send Object Actions

A SendSignalAction is notated as a convex pentagon with the name of the Signal placed inside it.

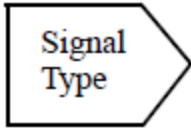
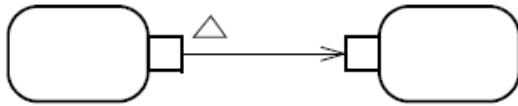


Figure 16.18 Sending a Signal

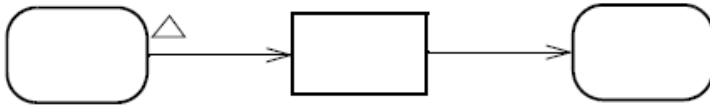
If a SendObjectAction is used in a way that will always result in the sending of a Signal (e.g., the type of the object InputPin is a Signal), then the SendSignalAction notation can be used for the SendObjectAction.

**Pin Annotations**

Pins corresponding to Parameters with isException=true are shown with a small triangle annotating the source end of the edge that comes out of the exception pin. The annotation is the same if the standalone ObjectNode notation is used. (See Figure 16.19.)



*Output pin, pin style, exception*



*Input and output pin, standalone style, exception*

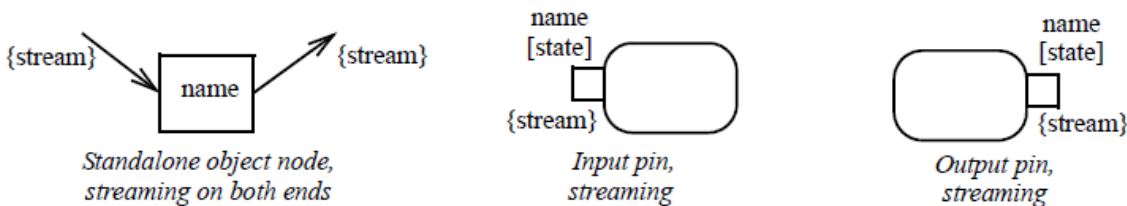
Figure 16.19 Exception Pin annotations

If the Parameter corresponding to a Pin has an effect specified, this is shown by placing the effect in braces near the edge leading to or from the Pin (see Figure 16.20).



Figure 16.20 Effect Pin annotations

Whether the Parameter corresponding to a Pin has isStreaming=true or false is shown by placing a textual annotation near the Pin symbol: {stream} or {nonstream}, respectively (see Figure 16.21). The annotation is the same if the standalone ObjectNode notation is used. {nonstream} is the default where the annotation is omitted.



*Standalone object node, streaming on both ends*

*Input pin, streaming*

*Output pin, streaming*

Figure 16.21 Stream Pin annotations

Additional emphasis may be added for streaming Parameters by using a graphical notation instead of the textual adornment. A standalone Pin can be connected with arrows having filled arrowheads to indicate streaming. Otherwise, Pins corresponding to streaming parameters can be shown as filled rectangles. When combined with the “pins with arrows inside” option above, the arrows inside are shown using the color of the inside of an unfilled Pin rectangle. (See Figure 16.22.)

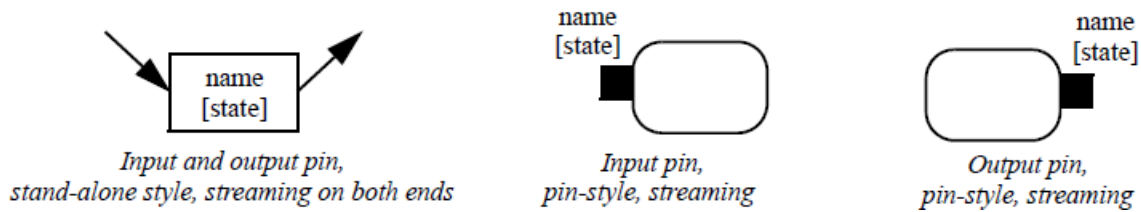


Figure 16.22 Stream Pin annotations, with filled arrows and rectangles

### Parameter Sets

Multiple ObjectFlows entering or leaving the Pins of an InvocationAction are typically treated as “and” conditions. However, sometimes one group of flows is permitted to the exclusion of another. This is modeled with ParameterSets and notated as rectangles surrounding one or more pins. The notation shown in Figure 16.23 expresses a disjunctive normal form for the inputs where one group of “and” flows are separated by “or” groupings.



Figure 16.23 Alternative input/outputs using ParameterSet notation

## 16.3.5 Examples

### Call Behavior Actions

Figure 16.24 is an example of invoking an activity called FillOrder.

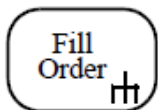


Figure 16.24 Invoking an Activity

### Send Signal Actions

Figure 16.25 shows part of an order-processing workflow in which two Signals are sent. An order is created (in response to some previous request that is not shown in the example). A Signal is sent to the warehouse to fill and ship the order. Then an invoice is created and sent to the customer.

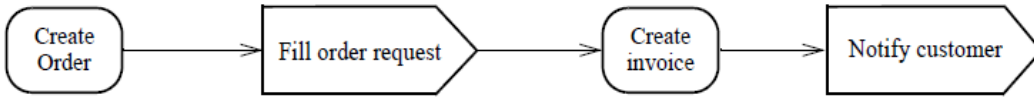


Figure 16.25 Sending Signals

### Pin Annotations

In Figure 16.26, Order Filling is a continuous Behavior that periodically emits (streams out) filled-order objects, without necessarily completing. Order Shipping is also a continuous Behavior that periodically receives filled-order objects as they are produced. Order Shipping is invoked when the first order arrives and does not terminate, processing orders as they arrive.

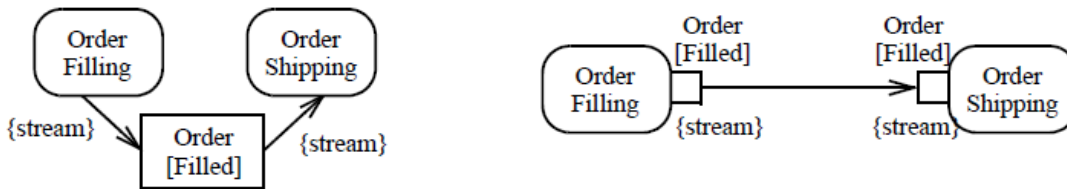


Figure 16.26 Streaming Pin examples

Figure 16.27 shows an example of exception notation. Accept Payment normally completes with a payment as being accepted and the account is then credited. However, when something goes wrong in the acceptance process, an exception can be returned that the payment is not valid, and the payment is rejected.

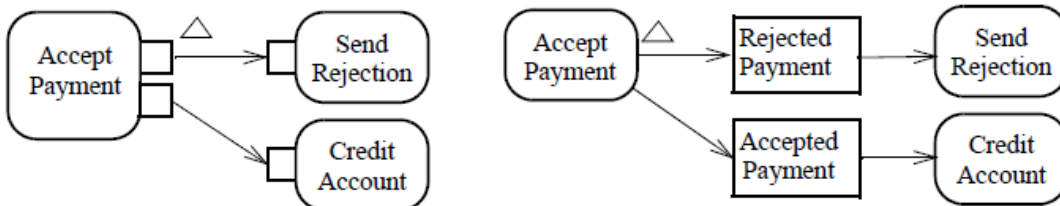


Figure 16.27 Exception Pin examples

Figure 16.28 depicts a Place Order Action that creates orders and a Fill Order Action that reads these placed orders and fills them.

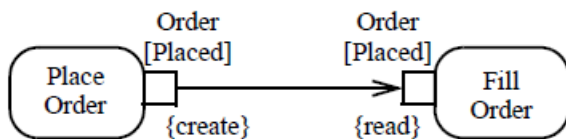


Figure 16.28 Pin example with effects

### Parameter Sets

In Figure 16.29, the Ship Item activity begins whenever it receives a bought item or a made item.

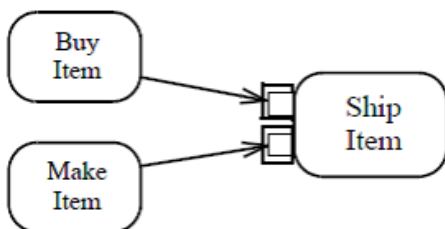




Figure 16.29 Alternative input/outputs using ParameterSets

## 16.4 Object Actions

### 16.4.1 Summary

Object Actions deal with the creation, destruction and comparison of instances of Classifiers. They also include Actions to read the instances of a given Classifier, check how an instance is classified, and to change the classification of an instance.

### 16.4.2 Abstract Syntax

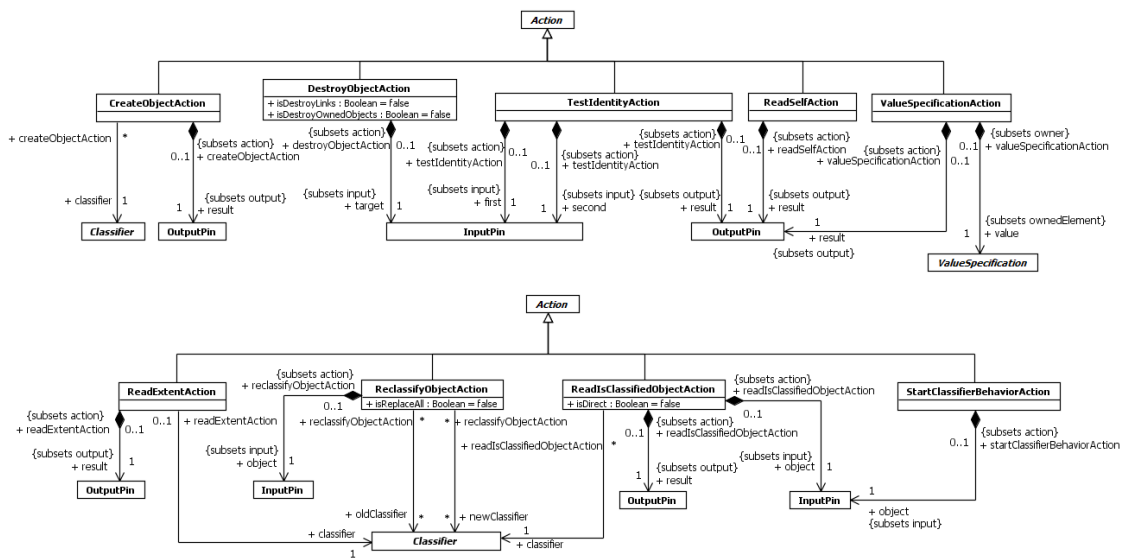


Figure 16.30 Object Actions

### 16.4.3 Semantics

#### Create Object Actions

A CreateObjectAction is an Action that creates a direct instance of a given Classifier and places the new instance on its result OutputPin. The Action has no other effect. In particular, no Behaviors are executed, no default value expressions are evaluated, and no StateMachine transitions are triggered. The new instance has no values for its StructuralFeatures and participates in no links.

If the Classifier being instantiated is a Behavior, then the instantiated object is an execution of that Behavior. However, this execution does not start automatically on instantiation. It must be explicitly started using a StartObjectBehaviorAction (see sub clause 16.3.3).

#### Destroy Object Actions

A DestroyObjectAction is an Action that destroys the object on its target InputPin. The object may be a link object, in which case the semantics of DestroyLinkAction also applies (see sub clause 16.6). When an object is destroyed, it is no longer classified under any Classifier. Other than the options described below for isDestroyLinks and isDestroyOwnedObjects, the Action has no other effect. In particular, no behaviors are executed, no state machine transitions are triggered, and references to the destroyed objects

are unchanged (except see `isDestroyLinks` below). The result of referencing an object that has been destroyed is not defined in this specification.

If `isDestroyLinks` is *true*, links in which the object participates are destroyed along with the object according to the semantics of `DestroyLinkAction`, except for link objects, which are destroyed according to the semantics of `DestroyObjectAction` with the same attribute values as the original `DestroyObjectAction`. If `isDestroyOwnedObjects` is *true*, objects owned by the object through composite aggregation are destroyed according to the semantics of `DestroyObjectAction` with the same attribute values as the original `DestroyObjectAction`.

Destroying an object that is already destroyed has no effect.

### Test Identity Actions

A `TestIdentityAction` is an action that tests if the two values given on its `InputPins` are identical objects. If the two values represent the same object, the Boolean value *true* is placed on the result `OutputPin`. Otherwise the value *false* is placed on the result `OutputPin`.

If an object is classified solely as an instance of one or more `Classes`, then testing whether it is the “same object” as another object is based on the identity of the object, independent of the current values for its `StructuralFeatures` or any links in which it participates (see sub clause [11.4.2](#)).

If an object is classified solely as an instance of one or more `DataTypes`, then testing whether it is the “same object” as another object is based on whether it has the same value (see sub clause [10.2.3](#)). For a `PrimitiveType`, equality of value is determined by the definition of the type, outside of UML. For an `Enumeration`, an `EnumerationLiteral` is equal to another value if it is the same `EnumerationLiteral`. Otherwise, an instance of a `DataType` is equal to another value if it is a direct instance of the same `DataType` with identical values for corresponding attributes (with “identical values” determined recursively as in a `Test Identity Action`).

The result of a `TestIdentityAction` for objects that are classified by both `Classes` and `DataTypes`, or by other kinds of `Classifiers`, is not defined, but, in all cases the Action produces a Boolean result.

### Read Self Actions

A `ReadSelfAction` is an Action that places the context object of the Action execution on its result `OutputPin`. (See sub clause [16.2.3](#) about context objects.)

For example, if a `ReadSelfAction` is contained in a `Behavior` that is the method of an `Operation`, then the context object it returns will be the instance of the owning `Classifier` of the `Operation` that was the target of the `Operation` invocation. However, if the `ReadSelfAction` is contained in a `Behavior` not owned by any `BehavedClassifier`, then the context object will be the instance of the `Behavior` in which the Action is executing.

### Value Specification Actions

A `ValueSpecificationAction` is an Action that evaluates a `ValueSpecification` and places the resulting value on its result `OutputPin`. See Clause [8](#) on the semantics of evaluating `ValueSpecifications`. In particular, a `LiteralSpecification` may be used in a `ValueSpecificationAction` to produce a constant value. Also see sub clause [9.8](#) on the semantics of evaluating `InstanceValues` to produce instances of `Classifiers`. Using an `InstanceValue` in a `ValueSpecificationAction` is similar to creating an instance using a `CreateObjectAction`, except that values may be given for the `StructuralFeatures` of the instance using slots on the `InstanceSpecification` of the `InstanceValue`.

### Read Extent Actions

A `ReadExtentAction` is an Action that retrieves the objects in the current extent of a `Classifier` and places them on its result `OutputPin`. The *extent* of a classifier is the set of all instances of a `Classifier` (including instances of any specializations) that exist at any one time.

It is not generally practical for implementations of ReadExtentAction to produce all the instances of the Classifier that exist everywhere. An execution engine typically manages only a limited subset of the instances of any Classifier and may manage multiple distributed extents for any one Classifier. It is not defined which managed extent is actually read by a ReadExtentAction.

### Reclassify Object Actions

A ReclassifyObjectAction is an Action that changes which Classifiers classify the object given on its object InputPin. It may both add and remove Classifiers from the object. Multiple Classifiers may be added and removed at one time.

After the Action completes, the input object is classified by each newClassifier and no oldClassifier. If the object was previously classified by any Classifier that is not an oldClassifier, then it is still classified by that Classifier. Neither specifying a newClassifier that duplicates an already existing Classifier, specifying an oldClassifier that is not classifying the input object, nor specifying a Classifier as both a newClassifier and an oldClassifier has any effect. A newClassifier shall not be abstract.

The identity of the input object is preserved, no behaviors are executed, and no default value expressions are evaluated. The newClassifiers replace existing classifiers in an atomic step, so that structural feature values and links are not lost during the reclassification when the oldClassifiers and newClassifiers have structural features and associations in common.

If isReplaceAll is *true*, then all existing Classifiers for the object are removed before the newClassifiers are added, except if a newClassifier already classifies the input object, in which case this Classifier is not removed.

The effect of removing all Classifiers from an object and not adding any new ones is not defined.

### Read-Is-Classified-Object Actions

A ReadIsClassifiedObjectAction is an Action that determines whether the object given on its object InputPin is classified by a given Classifier. If it is so classified, then a Boolean *true* value is placed on the result OutputPin. Otherwise a *false* value is placed on the result OutputPin.

If isDirect is *true*, then the test is whether the object is directly classified by the specified Classifier and not by any specializations of it. If isDirect is *false*, then the object may be classified by the specified Classifier or any (direct or indirect) specialization of it.

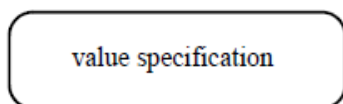
### Start Classifier Behavior Actions

A StartClassifierBehaviorAction is an Action that starts the execution of the classifierBehavior of the object given on its object InputPin. The Action completes as soon as the execution of the Behavior has started, after which the Behavior executes asynchronously. If the Behavior is already executing, or if the given object is not classified by a Classifier with a classifierBehavior, the StartClassifierBehaviorAction has no effect.

**NOTE.** StartClassifierBehaviorAction is provided for compatibility with older versions of UML. It is generally preferable to use a StartObjectBehaviorAction instead of a StartClassifierBehaviorAction, as a StartObjectBehaviorAction allows for the passing of Parameter values and for synchronous invocation (see sub clause 16.3).

## 16.4.4 Notation

A ValueSpecificationAction is labeled with its ValueSpecification, as shown in Figure 16.31.



**Figure 16.31 ValueSpecificationAction notation**

There is no specific notation defined for other kinds of object actions.

## 16.4.5 Examples

Figure 16.32 shows ValueSpecificationActions used to output a constant from an activity.

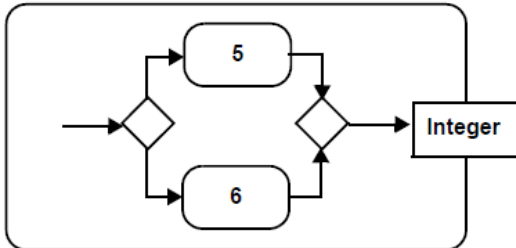


Figure 16.32 ValueSpecificationActions

## 16.5 Link End Data

### 16.5.1 Summary

Links (instances of Associations) that are not link objects (instances of AssociationClasses) cannot be passed as runtime values to or from an Action. Instead, a link is identified by the values on its ends. LinkEndData is a specification of such values for one end of a link, used in the identification of links by LinkActions (as further described in sub clause 16.6).

### 16.5.2 Abstract Syntax

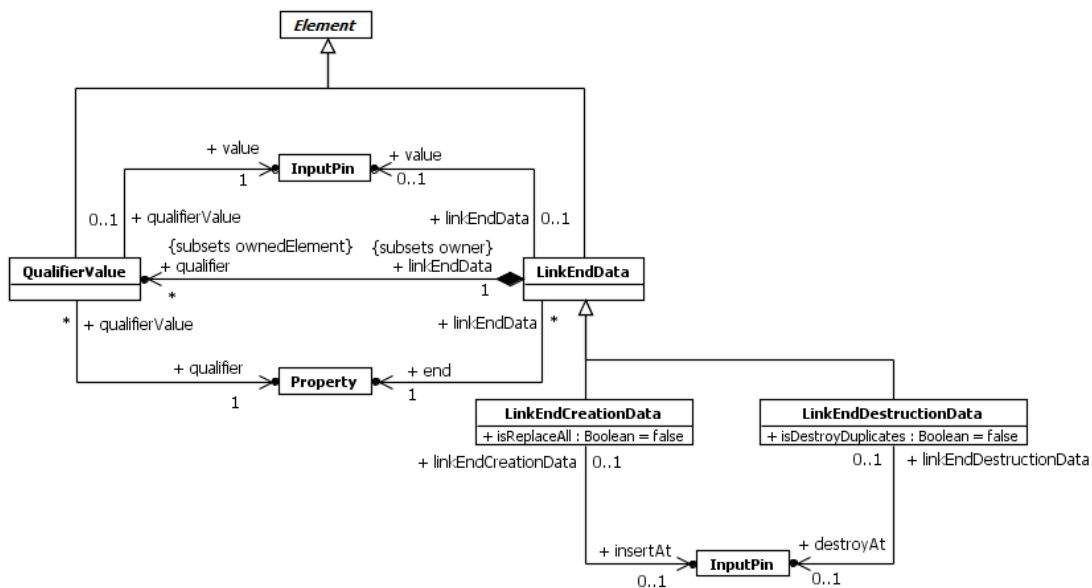


Figure 16.33 Link End Data

## 16.5.3 Semantics

### Link End Data

LinkEndData is an Element that specifies the inputs to be used to match the values on one end of a link. Each Association end is identified separately with a LinkEndData element.

Each LinkEndData element has three parts:

1. Identification of the end of the link that is being matched. This Property must be a memberEnd of an Association (see sub clause [11.4](#)).
2. A value InputPin providing the value that is expected on the given end of the link. This InputPin has the same type as the given end and multiplicity 1..1. Having a value InputPin is optional, to allow for the specification of an “open end” on a ReadLinkAction (see sub clause 16.6).
3. Optional QualifierValues identifying InputPins that provide the expected value of qualifiers of the given end of the link. The qualifier Properties must be qualifiers of the end Property (see sub clause [9.5](#)). The InputPin has the same type as the given qualifier and multiplicity 1..1.

### Link End Creation Data

LinkEndCreationData is a specialized kind of LinkEndData used to identify one end of a link to be created by CreateLinkAction (see sub clause 16.6). In addition to what is included in regular LinkEndData, LinkEndCreationData includes the following:

- An isReplaceAll option that, if true, specifies that the new link replaces all links that previously matched the values at this end.
- If the given end is ordered, then an insertAt InputPin, with a type of UnlimitedNatural and a multiplicity of 1..1, must be specified to provide the insertion point of the new link in the ordered values on this end. (See sub clause 16.6 for more about insertAt.)

### Link End Destruction Data

LinkEndDestructionData is a specialized kind of LinkEndData used to identify one end of a link to be destroyed by DestroyLinkAction (see sub clause 16.6). In addition to what is included in regular LinkEndData, LinkEndDestructionData includes the following:

- An isDestroyDuplicates option that, if true, specifies that all links matching the values on this end shall be destroyed.
- If the given end is ordered and non-unique, and isDestroyDuplicates=false, then a destroyAt InputPin, with a type of UnlimitedNatural and a multiplicity of 1..1, must be specified to provide a value for the position in this end of the link to be destroyed. (See sub clause 16.6 for more about destroyAt.)

## 16.5.4 Notation

There is no specific notation for LinkEndData.

## 16.5.5 Examples

None.

## 16.6 Link Actions

### 16.6.1 Summary

LinkActions (and ClearAssociationAction) operate on Associations and their instances, links. This includes Associations that are AssociationClasses (but see also sub clause 16.7 for LinkObjectActions that operate specifically on instances of AssociationClasses).

### 16.6.2 Abstract Syntax

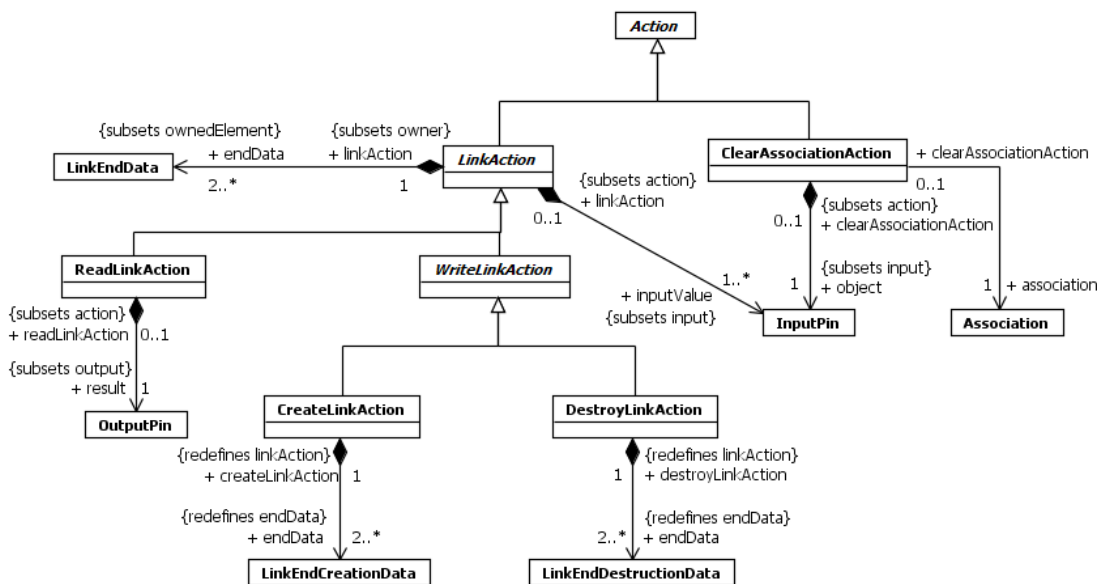


Figure 16.34 Link Actions

### 16.6.3 Semantics

#### Link Actions

A LinkAction is an Action that reads, creates, or destroys links of an Association, which may be an AssociationClass. The links acted on are identified using LinkEndData, which specifies the values expected on the ends of those links (see sub clause 16.5). All the ends of the LinkEndData elements of a LinkAction must be memberEnds of the same Association. All the InputPins identified in the LinkEndData elements of a LinkAction must be inputValue InputPins of the LinkAction. The semantics are undefined for Associations that have an end with isStatic=true.

#### Read Link Actions

A ReadLinkAction is a LinkAction that retrieves values on one end of an Association (given the values on all the other ends) and places them on its result OutputPin. The end being read is called the *open end* for the ReadLinkAction. Exactly one endData element for a ReadLinkAction must *not* have a value InputPin identified, and its corresponding end is the open end. The semantics are undefined for reading a link whose open end is not navigable or is not visible from the context Classifier of the ReadLinkAction or from the containing Behavior, if there is no context Classifier (see sub clause 16.2.3 about context objects).

When a `ReadLinkAction` executes, it identifies the subset of all existing links of the Association that match the `endData` object values and qualifier values for all ends other than the open end and any qualifier values given for the open end. The values placed on the result `OutputPin` are the ones on the open end of this subset of links. If the open end is ordered, then values are placed onto the result `OutputPin` in order. If there are no matching links, then the `ReadLinkAction` produces a single null token on its result `OutputPin`.

The result `OutputPin` of a `ReadLinkAction` must have the same type and ordering as the open end. The multiplicity of the open end must be compatible with that of the `OutputPin`, but they do not have to be the same. For example, a modeler can set the multiplicity of the result `Pin` to support multiple values even when the open end only allows a single value (this way, the `ReadLinkAction` as modeled will be unaffected by changes in the multiplicity of the open end).

### Create Link Action

A `CreateLinkAction` is a `LinkAction` for creating links. It has no `OutputPin`, because links are not necessarily values that can be passed to and from Actions. A `CreateLinkAction` may also be used to create a link object, though there is no output value in this case, either. (The `CreateLinkAction` does not need to be changed if the Association is changed to an `AssociationClass`, or vice versa.) The semantics of `CreateLinkObjectAction` applies to creating link objects with `CreateLinkAction` (see sub clause 16.7).

`CreateLinkAction` uses a specialization of `LinkEndData` called `LinkEndCreationData`. It supports the destruction of existing links of the Association that connect any of the objects of the new link. When the link is created, this `isReplaceAll` option is available on an end-by-end basis, and causes all links of the association from the specified ends to be destroyed before the new link is created. If the link being created already exists, then it is not destroyed under this option. If all the Association ends are unordered and unique, creating a link between objects that are already linked by the same Association has no effect.

Associations with ordered ends are supported with an insertion point specified at runtime by an additional `insertAt` `InputPin` on `LinkEndCreationData`, which is required for ordered Association ends when `isReplaceAll` is `false` and omitted for unordered ends.

**NOTE.** Association ends may be ordered even if the upper multiplicity is 1.

The `insertAt` `Pin` is of type `UnlimitedNatural` with multiplicity of 1..1. An insertion point that is a positive integer less than or equal to the current number of links means to insert the new link at that position in the sequence of existing links, with the integer 1 meaning the new link will be first in the sequence. A value of unlimited (“\*”) for the insertion point means to insert the new link at the end of the sequence. The semantics are undefined for a value of 0 and for an integer greater than the number of existing links. Reinserting an existing link at a new position in an ordered, unique end moves the link to that position.

The semantics is undefined for creating a link for an Association that is abstract. The semantics is undefined for creating a link that violates the upper multiplicity of one of its Association ends. A new link violates the upper multiplicity of an end if the cardinality of that end after the link is created would be greater than the upper multiplicity of that end. The cardinality of an end is equal to the number of links with objects participating in the other ends that are the same as those participating in those other ends in the new link, and with qualifier values on all ends the same as the new link, if any.

The semantics is undefined for creating a link that has an Association end with `isReadOnly=true` after initialization of the other end objects, unless the link being created already exists (in which case the `CreateLinkAction` has no effect). Links may be created as long as the objects that will participate in the new links across from all read-only ends are still being initialized. This means that Associations with two or more read-only ends cannot have links created unless all the objects to be linked are being initialized.

### Destroy Link Actions

A `DestroyLinkAction` is an Action that destroys links matching specified `LinkEndData`. If there are no matching links, then the `DestroyLinkAction` has no effect.

A `DestroyLinkAction` may also be used to destroy link objects, though the objects to destroy are still specified in the same way, using `LinkEndData`. (The `DestroyLinkAction` thus does not need to be changed if the Association is changed to an `AssociationClass`, or vice versa.) The semantics of `DestroyObjectAction` applies to destroying a link object with `DestroyLinkAction`.

DestroyLinkAction uses a specialization of LinkEndData called LinkEndDestructionData. It supports destruction of duplicate links of Association on ends that are non-unique. This isDestroyDuplicates option is available on an end-by-end basis, and causes all duplicate links of the Association from the specified ends to be destroyed.

Associations with ordered, non-unique ends are supported by a deletion position specified at runtime by an additional destroyAt InputPin on LinkEndDestructionData, which is required for ordered, non-unique Association ends when isDestroyDuplicates is false, and omitted for other ends. The destroyAt Pin is of type UnlimitedNatural with multiplicity of 1..1. A deletion position that is a positive integer less than or equal to the current number of links means to destroy the link at that position in the sequence of existing links, with the integer 1 meaning the first link in the sequence. The semantics are undefined for a value of 0, an integer greater than the number existing links, and unlimited (“\*”).

The semantics are undefined for destroying a link that has an Association end with isReadOnly = true after initialization of the other end objects, unless no link matches the endData (in which case the DestroyLinkAction has no effect). Links may be destroyed as long as the objects participating in the links across from all read-only ends are still being initialized. This means links with two or more read-only ends cannot be destroyed, unless all the participating objects are being initialized.

### **Clear Association Actions**

A ClearAssociationAction is an Action that destroys all links of an Association in which a particular object participates, including link objects of an AssociationClass. The Association to be cleared is statically specified. The ClearAssociationAction has an InputPin for a runtime object that must be of the same or more specialized type as at least one of the memberEnds of the Association. All links of the Association that have the given object on any end are destroyed, even when this violates the minimum multiplicity of any of the Association ends. The semantics of DestroyObjectAction applies to destroying link objects with ClearAssociationAction.

### **16.6.4 Notation**

No specialized notation is defined for LinkActions or ClearAssociationActions.

### **16.6.5 Examples**

None.

## **16.7 Link Object Actions**

### **16.7.1 Summary**

Link object Actions operate on link objects, which are instances of AssociationClasses. LinkActions also operate on link objects, but identify them differently (see sub clause 16.7).



## 16.7.2 Abstract Syntax

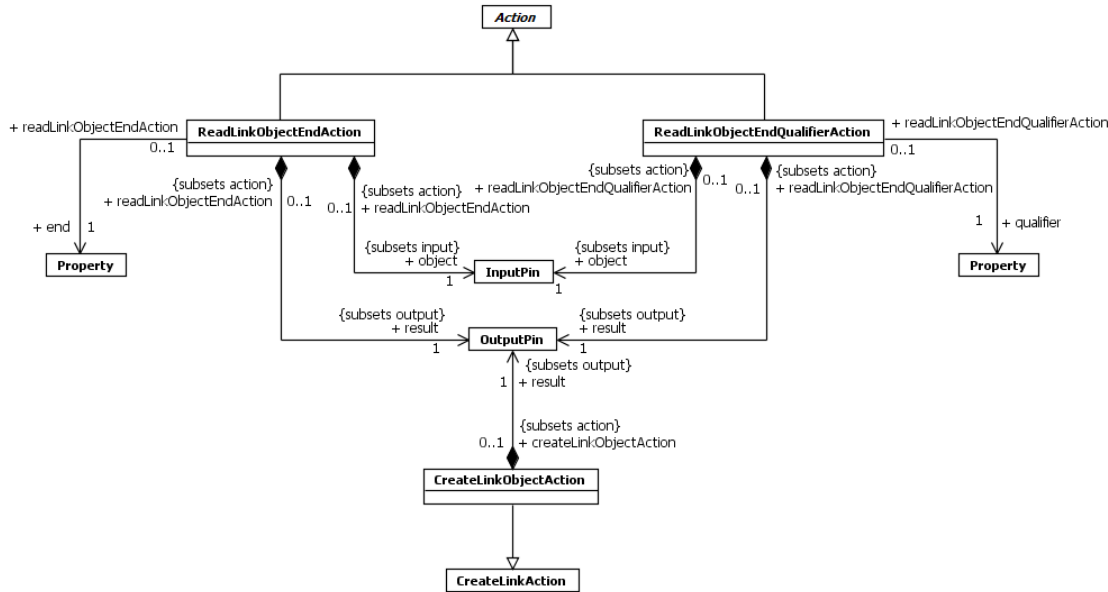


Figure 16.35 Link Object Actions

## 16.7.3 Semantics

### Read Link Object End Actions

A `ReadLinkObjectEndAction` is an action that retrieves an end object from a link object. The `AssociationClass` end from which to retrieve the object is specified statically. The link object to be read is given on the object `InputPin`. The value of the specified end of the link object is placed on the result `OutputPin`.

**NOTE.** This is *not* the same as reading links of the link object's `Association` with the specified end as the open end (as described for `ReadLinkAction` in sub clause 16.6). The link object being read is identified like any other object, rather than by the values of link ends, as in `ReadLinkAction`. There is exactly one object at each end of a link object, even if the multiplicity of the end is different from 1..1 in the `Association`. This is why the result `OutputPin` of a `ReadLinkObjectEndAction` always has a multiplicity of 1..1.

### Read Link Object End Qualifier Actions

A `ReadLinkObjectEndQualifierAction` is an action that retrieves a qualifier end value from a link object. The qualifier whose value is to be retrieved is specified statically. The owner of the qualifier must be an end of an `AssociationClass`. The link object to be read is given on the object `InputPin`. The value of the specified qualifier for the given link object is placed in the result `OutputPin`.

### Create Link Object Actions

A `CreateLinkObjectAction` is a specialized `CreateLinkAction` (see sub clause 16.6) for creating a link object (an instance of an `AssociationClass`). A `CreateLinkObjectAction` has the same semantics for link creation as a `CreateLinkAction`, except that its `endData` must be for an `AssociationClass` and the new link is a link object that is placed on the result `OutputPin`. If a link object matching the given `endData` already exists, and all the `Association` ends are unique, then this is placed on the result `OutputPin`, and no new link object is created.

## 16.7.4 Notation

No specialized notation is defined for Actions that operate on link objects.

## 16.7.5 Examples

None.

## 16.8 Structural Feature Actions

### 16.8.1 Summary

StructuralFeatureActions support the reading and writing of StructuralFeatures.

### 16.8.2 Abstract Syntax

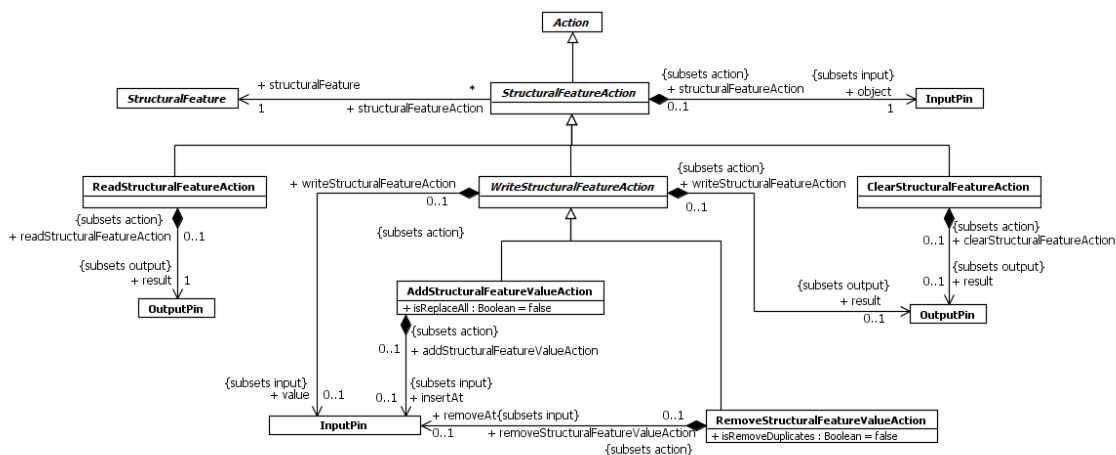


Figure 16.36 Structural Feature Actions

### 16.8.3 Semantics

#### Structural Feature Actions

A StructuralFeatureAction is given a statically-specified StructuralFeature of a Classifier, and an object on which to act on its object InputPin. This object is either an instance (direct or indirect) of the Classifier that owns the StructuralFeature or, if the StructuralFeature is an ownedEnd of a binary Association, an instance of the type of the opposite end of the Association. If the StructuralFeature is an Association end, then a StructuralFeatureAction has the same semantics as a LinkAction on an Association that has the StructuralFeature as an end (see specializations of StructuralFeatureAction). The semantics are undefined if the StructuralFeature is not visible from the context Classifier of the StructuralFeatureAction or from the containing Behavior, if there is no context Classifier (see sub clause 16.2.3 about context objects), or if the StructuralFeature has isStatic=true.

The StructuralFeatures and Associations in which an object participates may change over time due to dynamic classification (see ReclassifyObjectAction in sub clause 16.4). However, the type of the object InputPin of a StructuralFeatureAction is a single Classifier, and the semantics are defined only when the object passed to the Action is classified by that Classifier (directly or indirectly) at the time the Action accepts it and while the Action is executing. The StructuralFeature is referenced from a

StructuralFeatureAction as a model Element, so it is uniquely identified, even if there are other StructuralFeatures of the same name on other Classifiers.

A ReadStructuralFeatureAction reads the values of a StructuralFeature and places these values on its result OutputPin. The other kinds of StructuralFeatureActions, WriteStructuralFeatureActions (including AddStructuralFeatureValueActions and RemoveStructuralFeatureValueActions) and ClearStructuralFeatureActions modify the values of a StructuralFeature. These Actions may optionally have a result OutputPin. If a result OutputPin is provided, then the input object, as modified, is placed on this OutputPin. If the input object is a data value (an instance of a DataType), then a *copy* of the input data value is placed on the output pin, but with the appropriate structural feature modified. As a data value does not have an independent identity, the only way to obtain the modified data value is through the use of the result OutputPin.

### Read Structural Feature Actions

A ReadStructuralFeatureAction is a StructuralFeatureAction that retrieves the values of a StructuralFeature and places them on its result OutputPin. If the StructuralFeature is ordered, then the values are placed on the OutputPin in order. If the StructuralFeature is an Association end, the semantics are the same as a ReadLinkAction with the StructuralFeature as the open end (see sub clause 16.6). If there are no retrieved values (that is, the StructuralFeature is empty), then a ReadStructuralFeatureAction produces a single null token on its result OutputPin.

The type and ordering of the result OutputPin are the same as those of the StructuralFeature. The multiplicity of the StructuralFeature must be compatible with the multiplicity of the result OutputPin, but does not have to be the same. For example, the modeler can set the multiplicity of this OutputPin to support multiple values even when the StructuralFeature only allows a single value (this way, the ReadStructuralFeatureAction as modeled will be unaffected by changes in the multiplicity of the StructuralFeature).

### Add Structural Feature Value Actions

An AddStructuralFeatureValueAction is a StructuralFeatureAction for adding a value to a StructuralFeature of an object. The value to be added is given on the value InputPin, which is required. This InputPin has the same type as the StructuralFeature and a multiplicity of 1..1 (that is, a single value is added). If the StructuralFeature is an Association end, the semantics are the same as for a CreateLinkAction (see sub clause 16.6), where the participants in the link are the object being acted on and the new value.

If isReplaceAll is true, then the existing values of the StructuralFeature are removed before the new value is added, except if the StructuralFeature already contains the new value, in which case it is not removed under this option. The StructuralFeature always has a single value when the Action completes, even if the lower multiplicity of the StructuralFeature is greater than 1. If isReplaceAll is false and the StructuralFeature is unordered and unique, then adding a value that is already contained in the StructuralFeature has no effect.

Adding a value to an ordered StructuralFeature requires an insertion point for the new value given in the insertAt InputPin, which is required for ordered StructuralFeatures when isReplaceAll is false and omitted for unordered StructuralFeatures.

**NOTE.** Values of StructuralFeatures may be ordered even if the upper multiplicity is 1.

If the insertAt InputPin is present, it has type UnlimitedNatural and multiplicity 1..1. An insertion point that is a positive integer less than or equal to the current number of values means to insert the new value at that position in the sequence of existing values, with the integer 1 meaning the new value will be first in the sequence. A value of unlimited (“\*”) for the insertion point means to insert the new value at the end of the sequence. The semantics are undefined for a value of 0 or an integer greater than the number of existing values. Reinserting an existing value at a new position in an ordered, unique StructuralFeature moves the value to that position (this works because such StructuralFeature values are ordered sets). The insertion point is ignored if it is used when isReplaceAll=true.

The semantics are undefined for adding a value that violates the upper multiplicity of the StructuralFeature, and for adding a new value to a StructuralFeature with isReadOnly=true after initialization of the object that would have the value.

## Remove Structural Feature Value Actions

A `RemoveStructuralFeatureValueAction` is a `StructuralFeatureAction` for removing a value from a `StructuralFeature` of an object. If the feature is an Association end, the semantics are the same as for `DestroyLinkAction`, where the participants in the link are the object the value being removed.

Except as given below, the value to be removed is given on the value `InputPin`, which has the same type as the `StructuralFeature` and a multiplicity of 1..1. The value is removed even when this results in a violation of the lower multiplicity of the `StructuralFeature`. Removing a value that is not contained in the `StructuralFeature` has no effect. The `isRemoveDuplicates` option indicates whether to remove all duplicates of the specified value in non-unique `StructuralFeatures`.

If `isRemoveDuplicates` is false and the `StructuralFeature` is ordered and non-unique, then there is no value `InputPin`, and the value to be removed is instead specified by giving its position on the `removeAt` `InputPin`, which has type `UnlimitedNatural` and a multiplicity of 1..1. A removal position that is a positive integer less than or equal to the current number of values means to remove the value at that position in the sequence of existing values, with the integer 1 meaning the first value in the sequence. The semantics are undefined for 0, an integer greater than the number of existing values, and unlimited (“\*”).

The semantics are undefined for removing an existing value of a `StructuralFeature` with `readOnly=true` after initialization of the owning object.

## Clear Structural Feature Actions

A `ClearStructuralFeatureAction` is a `StructuralFeatureAction` that removes all values of a `StructuralFeature`, even if lower multiplicity of the `StructuralFeature` is greater than 0. The action has no effect if the `StructuralFeature` has no values. If the `StructuralFeature` is an Association end, the semantics are the same as for a `ClearAssociationAction` on the given object.

The semantics are undefined for a `StructuralFeature` with `isReadOnly = true` after initialization of the object owning the `StructuralFeature`, unless the `StructuralFeature` has no values.

### 16.8.4 Notation

No specialized notation is defined for `StructuralFeatureActions`.

### 16.8.5 Examples

None.

## 16.9 Variable Actions

### 16.9.1 Summary

`VariableActions` support the reading and writing of `Variables`.

## 16.9.2 Abstract Syntax

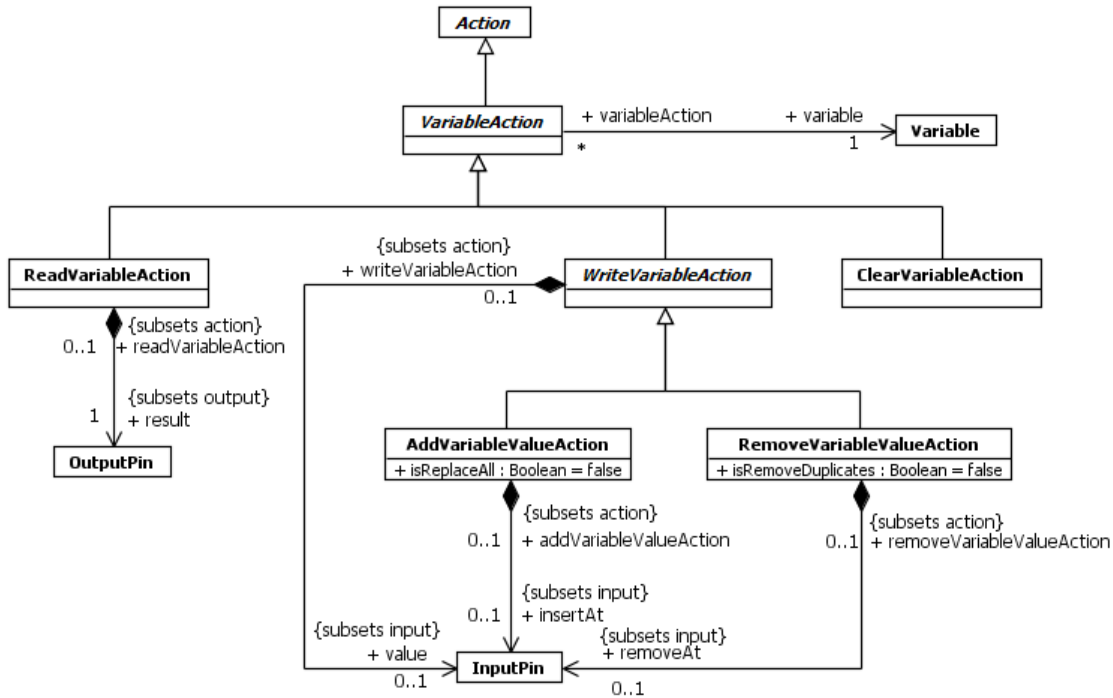


Figure 16.37 Variable Actions

## 16.9.3 Semantics

### Variable Action

A VariableAction operates on a statically-specified Variable. The Variable must be one that is defined either by an Activity (see sub clause 15.2) or a StructuredActivityNode (see sub clause 16.11) containing the VariableAction.

### Read Variable Actions

A ReadVariableAction is a VariableAction that retrieves the values of a Variable and places them on its result OutputPin. If the Variable is ordered, then the values are placed on the OutputPin in order. If there are no retrieved values (that is, the Variable is empty), then a ReadVariableAction produces a single null token on its result OutputPin.

The type and ordering of the result OutputPin are the same as those of the Variable. The multiplicity of the Variable must be compatible with the multiplicity of the result OutputPin, but does not have to be the same. For example, the modeler can set the multiplicity of this OutputPin to support multiple values even when the Variable only allows a single value (this way, the ReadVariableAction as modeled will be unaffected by changes in the multiplicity of the Variable).

### Add Variable Value Action

An AddVariableValueAction is a VariableAction for adding a value to a Variable. The value to be added is given on the value InputPin, which is required. This InputPin has the same type as the Variable and a multiplicity of 1..1 (that is, a single value is added).

If `isReplaceAll` is true, then the existing values of the Variable are removed before the new value is added, except if the Variable already contains the new value, in which case it is not removed under this option. The Variable always has a single value when the Action completes, even if the lower multiplicity of the Variable is greater than 1. If `isReplaceAll` is false and the Variable is unordered and unique, then adding a value that is already contained in the Variable has no effect.

Adding a value to an ordered Variable requires an insertion point for the new value using the `insertAt` InputPin, which is required for ordered Variable when `isReplaceAll` is false and omitted for unordered Variable (values of a Variable may be ordered or unordered, even if the multiplicity upper bound is 1.) If the `insertAt` InputPin is present, it has type `UnlimitedNatural` and multiplicity 1..1. An insertion point that is a positive integer less than or equal to the current number of values means to insert the new value at that position in the sequence of existing values, with the integer 1 meaning the new value will be first in the sequence. A value of unlimited (“\*”) for the insertion point means to insert the new value at the end of the sequence. The semantics are undefined for a value of 0 or an integer greater than the number of existing values. Reinserting an existing value at a new position in an ordered, unique Variable moves the value to that position (this works because such Variable values are ordered sets). The insertion point is ignored if it is used when `isReplaceAll=true`.

The semantics are undefined for adding a value that violates the upper multiplicity of the Variable.

### Remove Variable Value Actions

A `RemoveVariableValueAction` is a `VariableAction` for removing a value from a Variable.

The value to be removed is given on the value InputPin, which has the same type as the Variable and a multiplicity of 1..1. The value is removed even when this results in a violation of the lower multiplicity of the Variable. Attempting to remove a value that is not contained in the Variable has no effect. The `isRemoveDuplicates` option indicates whether to remove all duplicates of the specified value in non-unique Variables.

If `isRemoveDuplicates` is false and the Variable is ordered and non-unique, then there is no value InputPin, and the value to be removed is specified by giving its position on the `removeAt` InputPin, which has type `UnlimitedNatural` and a multiplicity of 1..1. A removal position that is a positive integer less than or equal to the current number of values means to remove the value at that position in the sequence of existing values, with the integer 1 meaning the first value in the sequence. The semantics are undefined for 0, an integer greater than the number of existing values, and unlimited (“\*”).

### Clear Variable Actions

A `ClearVariableAction` is a `VariableAction` that removes all values of a Variable, even if the lower multiplicity of the Variable is greater than 0. The action has no effect if the Variable has no values.

## 16.9.4 Notation

The presentation option at the top of Figure 16.38 may be used as notation for abstract syntax corresponding to the notation at the bottom of the figure. If the `AddVariableValueAction` has `isReplaceAll=true`, this can be shown with the textual annotation `{replaceAll}` near the variable name.

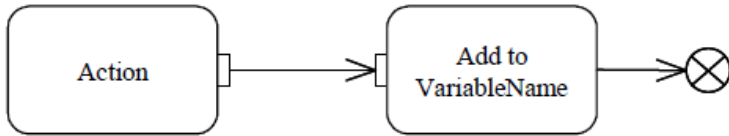
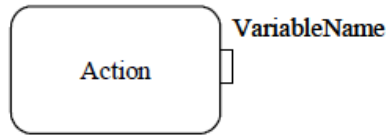


Figure 16.38 Presentation option for AddVariableValueAction

### 16.9.5 Examples

None.

## 16.10 Accept Event Actions

### 16.10.1 Summary

An AcceptEventAction waits for the occurrence of one or more Events. If an accepted Event occurrence is for a CallEvent, then a ReplyAction may be used to reply to it. If an accepted Event occurrence is for a SignalEvent, then the received Signal instance may either be unmarshalled immediately into its attribute values, or this may be done later using an UnmarshallAction.

### 16.10.2 Abstract Syntax

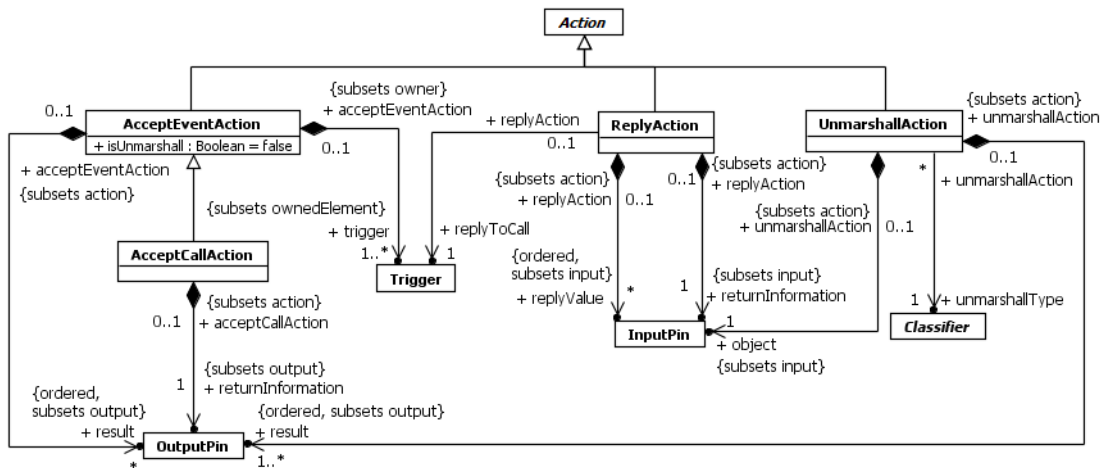


Figure 16.39 Accept Event Actions

## 16.10.3 Semantics

### Accept Event Action

AcceptEventAction is an Action with Triggers for one or more Events. When an AcceptEventAction is executed, it waits for an Event occurrence to be dispatched from the event pool of the context object for its execution that matches one of its Triggers. The context object for an AcceptEventAction is the context object of the Behavior execution within which the AcceptEventAction is executing (which may be the Behavior execution itself, see sub clause [13.2.3](#)). An AcceptEventAction is a *wait point* in the sense discussed in sub clause [13.3.3](#), except only the AcceptEventAction waits, rather than the whole Activity (Activities can have other Actions executing while AcceptEventActions are waiting).

If a matching Event occurrence for an AcceptEventAction is dispatched from the event pool, then the AcceptEventAction is enabled to continue. However, if the containing Behavior execution has more than one waiting Trigger that matches the Event occurrence, only one of them will be selected to actually trigger. If the Trigger on the AcceptEventAction is chosen, then it completes and produces output on any result OutputPins.

An AcceptEventAction with a trigger for a SignalEvent is informally called an *accept signal action*. If an accept signal action has isUnmarshall=false, then it must have a single result OutputPin on which the Signal instance associated with an accepted SignalEvent occurrence is placed. If it has isUnmarshall=true, then it must have result OutputPins corresponding to each of the attributes of the Signal of the SignalEvent (in order), and the attribute values of the Signal instance associated with an accepted SignalEvent occurrence are placed on these OutputPins.

An AcceptEventAction with a trigger for a TimeEvent is informally called a *wait time action*. A wait time action must have a single result OutputPin. When it accepts a TimeEvent occurrence, then the time value of when the occurrence transpired is placed on the result OutputPin.

If the triggers of an AcceptEventAction are all for ChangeEvents and/or CallEvents, then the AcceptEventAction has no result OutputPins (unless the AcceptEventAction is an AcceptCallAction, see below). If the triggers include SignalEvents and/or TimeEvents along with ChangeEvents and/or CallEvents, then the AcceptEventAction must have isUnmarshall=false and a single result OutputPin, on which a null token is placed in the case of the acceptance of an occurrence of a ChangeEvent or a CallEvent.

**NOTE.** While an AcceptEventAction can, in general, contain triggers for CallEvents, it cannot accept synchronous calls unless it is an AcceptCallAction (see below).

If one of the triggers of an AcceptEventAction is an AnyReceiveEvent, and the Event occurrence is for a message that is not matched by a SignalEvent or CallEvent trigger on the same AcceptEventAction, then the Event occurrence matches the trigger for the AnyReceiveEvent (see also the discussion under Message Events in sub clause [13.3.3](#)).

If an AcceptEventAction is used in an Activity, there are special rules for when it is enabled (see sub clause [13.2.3](#) for a general discussion of the enabling of ActivityNodes). If the AcceptEventAction has no incoming edges, by the usual rules, it is enabled when its immediately containing Activity (or StructuredActivityNode) begins execution. However, in addition, an AcceptEventAction with no incoming edges remains enabled after it accepts an Event occurrence. That is, it does not terminate after accepting an Event occurrence and outputting any values (as described above), but continues to wait for another Event occurrence. Such an AcceptEventAction is terminated when its immediately containing Activity (or StructuredActivityNode) is terminated.

### Accept Call Actions

An AcceptCallAction is an AcceptEventAction specialized for the handling of CallEvent occurrences. An AcceptCallAction must have a single Trigger that is for a CallEvent. It has result OutputPins corresponding to each of the in and inout ownedParameters of the Operation identified in the CallEvent, in order. In addition, it has a returnInformation OutputPin that provides the information necessary to return from a synchronous call.

An AcceptCallAction is triggered in the same way as a normal AcceptEventAction on matching CallEvent occurrences. The argument values associated with the call message are then placed on the result OutputPins. In addition, information sufficient to



perform a subsequent ReplyAction (see below) is placed on the returnInformation OutputPin. The contents of the return information value is not defined by this specification and may only be used by a ReplyAction.

An AcceptCallAction triggered by an asynchronous call will still produce a value on its returnInformation OutputPin, but a ReplyAction accepting the value will complete immediately when given a return information value for an asynchronous call, with no effect.

**NOTE.** Operation referenced in the CallEvent of an AcceptCallAction should not have an associated method Behavior. Otherwise, a call to the Operation will have the immediate effect of executing the method and will not be placed into the event pool for the context object. Thus, a call to the Operation will never be dispatched to the AcceptCallAction. (See also the discussion of Event Dispatching in sub clause [13.3.3](#)).

## Reply Actions

A ReplyAction is an Action that completes the handling of a call that was accepted by a previous AcceptCallAction (see above). The two are connected by a return information value, which is produced by the AcceptCallAction in its returnInformation OutputPin and placed on the returnInformation InputPin of the ReplyAction by the containing Behavior (for example, with an ObjectFlow in an Activity). The ReplyAction also identifies a Trigger, which should be the same CallEvent Trigger owned by the AcceptCallAction that the ReplyAction is getting its return information value from. The replyValue InputPins of the ReplyAction shall correspond, in order, to the out, inout, and return ownedParameters of the Operation identified by the CallEvent.

When a ReplyAction executes, it generates a reply message to the original call request message, using the values from its replyValue InputPins. For a synchronous call, the value on its returnInformation InputPin is used to identify the caller to which the reply message is sent. However, if the return information value is the result of an asynchronous call, then no reply message is sent and the ReplyAction completes with no effect. The details of transmitting call requests, encoding return information, and transmitting replies are not defined in this specification.

Return information may be copied, stored in objects, and passed around, but it may only be used in a ReplyAction once. The semantics are undefined if the same return information value is supplied to a second ReplyAction. The semantics are also undefined if the return information value is not for a call to the same Operation as identified by the replyToCall Trigger of the ReplyAction. However, it is not intended that any profile or execution engine give any meaning to these undefined cases other than errors.

**NOTE.** If a ReplyAction is never executed on the return information from a synchronous call, then the caller will never receive a reply and, therefore, will never complete execution. This is not illegal, but it is usually undesirable.

## Unmarshall Actions

An UnmarshallAction is an Action that retrieves the values of the StructuralFeatures of an object and places them on OutputPins. The object is given on the object InputPin, which has the type given by the unmarshallType Classifier and a multiplicity of 1..1. The UnmarshallAction then has result OutputPins that correspond, in order, to each of the Properties of the unmarshallType (which must have at least one attribute).

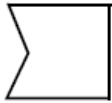
The owned Properties of a Signal are ordered, but a Classifier may also inherit Properties from other Classifiers due to Generalization relationships. In this case, the Properties of the unmarshallType are ordered such that all owned Properties come before any inherited Properties. Further, if two ancestors of the unmarshallType are related directly or indirectly by Generalization relationships, then the owned Properties of the more specific Classifier are ordered before the owned Properties of the more general Classifier. However, in the presence of multiple Generalization, some ancestors of the unmarshallType may not have any such transitive Generalization relationship, and no standard ordering is defined between the Properties of such ancestors.

When an UnmarshallAction executes, it takes the object from its InputPin, retrieves the values of the Properties of the unmarshallType from the object and places these values on the corresponding OutputPins. If a Property is ordered and has multiple values, then those values are placed in order on the corresponding OutputPin.

An UnmarshallAction is useful, for example, to obtain the attribute values of a Signal instance produced by an AcceptEventAction with isUnmarshall=false.

## 16.10.4 Notation

An AcceptEventAction in general is notated with a concave pentagon symbol (see Figure 16.40, left). The name of the Action may be placed within the symbol. A wait time action (i.e., an AcceptEventAction with a single TimeEvent trigger) is notated with an hour glass symbol (see Figure 16.40, right). The name of the Action may be placed below the symbol.



*Accept event action*



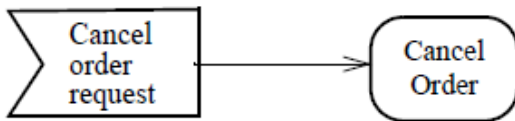
*Accept time event action*

**Figure 16.40** AcceptEventAction notations

## 16.10.5 Examples

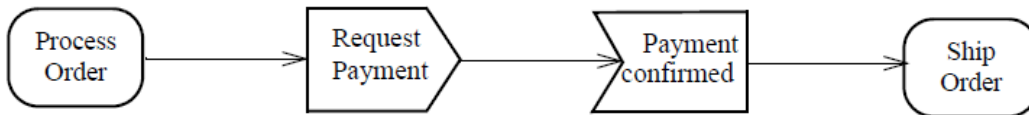
### Accept Event Actions

Figure 16.41 is an example of an accept signal action (i.e., an AcceptEventAction with a single SignalEvent trigger) that accepts a Signal indicating the cancellation of an order. The acceptance of the Signal causes an invocation of a cancellation behavior. This Action is enabled on entry to the Activity containing it, therefore it has no incoming ControlFlow.



**Figure 16.41** Implicitly enabled AcceptEventAction

In Figure 16.42, a request payment Signal is sent after an order is processed (see sub clause 16.3.4 on the notation used for a SendSignalAction). The Activity then waits to receive a payment confirmed Signal. Acceptance of the payment confirmed Signal is enabled only after the request for payment is sent; no confirmation is accepted until then. When the confirmation is received, the order is shipped.



**Figure 16.42** Explicitly enabled AcceptEventAction

In Figure 16.43, the end-of-month wait time action (i.e., an AcceptEventAction with a single TimeEvent trigger) executes at the end of the month. As there are no incoming edges to the time event action, it is enabled as long as its containing Activity (or StructuredActivityNode) is. It will execute at the end of every month.



**Figure 16.43** Repetitive time event

### Unmarshall Actions

In Figure 16.44, an order is unmarshalled into the values of its name, shipping address, and product attributes.



### 16.11.3 Semantics

#### Structured Activity Nodes

A `StructuredActivityNode` is an `Action` that is also an `ActivityGroup` (see sub clause [15.6](#)) and whose behavior is specified by the `ActivityNodes` and `ActivityEdges` it so contains. Unlike other kinds of `ActivityGroup`, a `StructuredActivityNode` owns the `ActivityNodes` and `ActivityEdges` it contains, and so a node or edge can only be directly contained in one `StructuredActivityNode`. `StructuredActivityNodes` may be nested (as a `StructuredActivityNode`, as an `Action`, is also an `ActivityNode`), however, so an edge or node may be indirectly contained in a number of nested `StructuredActivityNodes`.

A `StructuredActivityNode` may also include the definition of `Variables`. These `Variables` may be operated on only by `VariableActions` (see sub clause 16.9) within the `StructuredActivityNode` and any nested `StructuredActivityNodes`, in addition to any `Variables` defined in any surrounding `StructuredActivityNode` or `Activity`. When a `StructuredActivityNode` begins executing, all its `Variables` are initially empty.

The immediately following discussion under this heading is for the semantics of a `StructuredActivityNode` that is *not* an instance of one of the specializations of `StructuredActivityNodes`. The semantics for specialized kinds of `StructuredActivityNodes` are then described in subsequent sections of this subclause (and in sub clause 16.9 for `ExpansionRegions`).

A `StructuredActivityNode` is enabled and begins executing following the normal rules for an `Action` (see sub clause 16.2.3). None of the nodes within a `StructuredActivityNode` are enabled until the containing `StructuredActivityNode` begins executing (including `InitialNodes` and `AcceptEventActions`). At that point, which nodes are enabled is determined in the same way as for the nodes of an `Activity` when it begins execution (see sub clause [15.2.3](#)). Execution then proceeds per the semantics of `Activity` models, as described in Clause [15](#).

The `InputPins` of a `StructuredActivityNode` may be the sources for `ActivityEdges` contained in the `StructuredActivityNode`. This allows tokens placed on those `InputPins` to be made available to `ActivityNodes` within the `StructuredActivityNode`. The `InputPins` offer their tokens on outgoing flows when the `StructuredActivityNode` begins execution.

Similarly, the `OutputPins` of a `StructuredActivityNode` may be the targets of `ActivityEdges` contained in the `StructuredActivityNode`. This allows tokens produced within the `StructuredActivityNode` to be made available as outputs. An `OutputPin` accepts tokens offered to it according to the general rules for `ObjectNodes`.

An `ActivityEdge` contained by a `StructuredActivityNode` must have both its source and target either be contained (directly or indirectly) in the `StructuredActivityNode` or be a `Pin` owned by the `StructuredActivityNode`. Conversely, an `ActivityEdge` that is not contained by a `StructuredActivityNode` may not have both its source and target contained in the `StructuredActivityNode`. `ActivityEdges` that cross into and out of a `StructuredActivityNode` are owned by a container, direct or indirect, of the `StructuredActivityNode`.

If an `ActivityEdge` has a source outside a `StructuredActivityNode` and a target within it, then any offers made on that edge remain pending until the `StructuredActivityNode` begins execution. While the `StructuredActivityNode` is executing, the target of the `ActivityEdge` may accept any offers pending from before the execution of the `StructuredActivityNode`, as well as any additional offers made while the `StructuredActivityNode` is executing, per the usual semantics of `ActivityNodes`.

If an `ActivityEdge` has a source inside a `StructuredActivityNode` and a target outside it, then no offers can be made on that edge unless the `StructuredActivityNode` is executing. During the execution of the `StructuredActivityNode`, any offers made from the `ActivityEdge` are immediately propagated out of the `StructuredActivityNode` and handled by the target of the edge per the usual semantics of `ActivityNodes`.

A `StructuredActivityNode` completes execution according to the same rules as for the completion of the execution of an `Activity` (see sub clause [15.2.3](#)), including terminating execution due to an `ActivityFinalNode` (see sub clause [15.3.3](#)). However, an `ActivityFinalNode` contained in a `StructuredActivityNode` will terminate *only* the immediately containing `StructuredActivityNode` and its contents, not any `Activity` or other `StructuredActivityNodes` in which that may be contained.

When a `StructuredActivityNode` completes its execution, all executions ongoing within it are terminated and all tokens contained in it are destroyed, except those on `OutputPins` of the `StructuredActivityNode`, which are offered on any outgoing edges. Tokens

may accumulate on the OutputPins during the execution of the StructuredActivityNode, but they are only offered on outgoing edges once the execution completes. Any OutputPins that do not hold any tokens when the StructuredActivityNode completes offer null tokens on their outgoing edges.

## Isolation

Because of the concurrent nature of the execution of Actions within and across Behaviors, it can be difficult to guarantee consistent access and modification of object memory. In order to avoid race conditions or other concurrency-related problems, it is sometimes necessary to isolate the effects of a group of Actions from the effects of Actions outside the group. This may be indicated by setting the `mustisolate` attribute to *true* on a StructuredActivityNode (including any of the specialized kinds of StructuredActivityNode).

If the `mustisolate` flag is true for a StructuredActivityNode, then any access to an object by an Action within the node must not conflict with access to the object by an Action outside the node. A conflict is defined as an attempt to write to the object by one or both of the Actions. If such a conflict potentially exists, then no such access by an Action outside the isolated StructuredActivityNode may be interleaved with the execution of the StructuredActivityNode. This specification does not define the ways in which this rule may be enforced. An execution engine may achieve isolation using a locking mechanism, or it may simply sequentialize execution to avoid concurrency conflicts, or it may use some other method. If it is impossible to execute a model in accordance with these rules, then it is ill-formed.

**NOTE.** Isolation is different from the property of “atomicity,” which is the guarantee that a group of Actions either all complete successfully or have no effect at all. Atomicity generally requires a rollback mechanism to prevent committing partial results.

## Conditional Nodes

A ConditionalNode is a StructuredActivityNode that chooses one among some number of alternative collections of ExecutableNodes to execute. A ConditionalNode consists of one or more Clauses, each of which represents a single branch of the conditional. A Clause consists of a test section and a body section, which identify disjoint subsets of the ExecutableNodes contained in the ConditionalNode. Any ExecutableNode in the ConditionalNode must be included in the test section or body section of exactly one Clause.

When a ConditionalNode begins execution, any InitialNodes within it are immediately enabled. An ExecutableNode contained in the ConditionalNode, however, can only become enabled when the test section or body section that contains it is executed, as described below. When a test or body section is executed, any ExecutableNode in the section that has no mandatory input data and no incoming ControlFlow *with a source in the same section* is enabled and receives a single control token. Execution then proceeds according to the usual semantics of Activities, except that any offers made to an ExecutableNode in a section that is not executing are not immediately delivered but remain pending. The target ExecutableNode may accept any pending offers if it eventually executes as part of a later execution of the section that contains it.

Once a ConditionalNode is executing, the test sections of any of its Clauses that have no predecessorClauses are executed (whether serially or concurrently or some combination is not defined). Each test section has an Action owning the decider OutputPin with type Boolean identified by the Clause. The result of a test section is the value placed on the decider OutputPin. If one or more test sections result in a true value, then the corresponding body sections are enabled for execution. Next, any Clauses for which all predecessorClauses have test sections that did *not* result in a true value have their test sections executed. This process continues until there are no more test sections to be executed.

**NOTE.** Where the execution of test sections is specified as being “concurrent” above, this means that the model does not impose any order on their execution. In general, such test sections may be executed in any order, including simultaneously (if the execution engine supports this). To enforce ordering of execution, predecessor/successor constraints may be specified among the Clauses of a ConditionalNode. One frequent case is a total ordering of the Clauses, in which case the test section execution order is determinate. If it is impossible for more than one test section to simultaneously result in true, then the result is deterministic anyway and it is unnecessary to order the Clauses, as ordering might impose undesirable and unnecessary restrictions on the implementation.

If `isAssured` is true for a `ConditionalNode`, this asserts that at least one test section will yield a true value. If `isDeterminate` is true, this asserts that at most one test section will yield a true value (the `predecessorClause` relationship may be used to enforce this assertion).

**NOTE.** It is impossible to automatically verify these assertions in general and it is not required to enforce them, but they may provide useful information to an execution engine. If the assertions are violated, then the model is ill formed.

Once the process of executing test sections finishes, if there is exactly one body section enabled for execution, that body section is executed. If more than one body section is enabled, only one is actually executed, but the choice of which one is non-deterministic. If there is no body section enabled for execution, then the execution of the `ConditionalNode` completes with no additional effect.

An “else” Clause is a Clause that is a successor to all other clauses in the `ConditionalNode` and whose test section always results in true. The body section for such a Clause will be enabled if and only if no other body section in the `ConditionalNode` is enabled. This ensures that at least one body will always be executed for the `ConditionalNode`.

Whenever a body section becomes enabled for execution, it may actually be executed before the completion of any further test section executions. In this case, any ongoing test section executions are terminated and no further test sections are executed. (If some test sections have external effects, terminating them may be a source of indeterminacy. Although test sections are permitted to have side effects, avoiding side effects in the tests reduces the chance of logical errors and race conditions.)

A `ConditionalNode` has an ordered set of result `OutputPins`. Each Clause of the `ConditionalNode` must have a matching set of body `OutputPins`, which must identify `OutputPins` owned by Actions in the body section of the Clause. Every Clause must have a body `Output` for each one of the result `OutputPins`. If the body section of a Clause is executed, then, once that execution completes, any tokens on the body `Outputs` of the Clause are moved to the corresponding result `OutputPins` of the `ConditionalNode`. The execution of the `ConditionalNode` then completes, and the tokens on its `OutputPins` are offered to any outgoing edges. Any `OutputPins` that do not hold any tokens when the `StructuredActivityNode` completes offer null tokens on their outgoing edges. If no test section of a `ConditionalNode` results in a true value, then no body section is executed, no tokens are placed on any result `OutputPins`, and null tokens are offered from all these `OutputPins`.

Once a `ConditionalNode` completes execution, all executions ongoing within it are terminated and all tokens within it are destroyed. If a `ConditionalNode` directly contains an `ActivityFinalNode` that accepts a token during execution of the `ConditionalNode`, then the `ConditionalNode` immediately completes. *Only* the immediately containing `ConditionalNode` is terminated in this case, not any `Activity` or other `StructuredActivityNodes` containing the `ConditionalNode`, per the rules for `ActivityFinalNodes` in `StructuredActivityNodes`.

A `ConditionalNode` may not have `InputPins`. However, `ActivityEdges` may cross into and out of a `ConditionalNode`, as for `StructuredActivityNodes` in general, and the semantics are the same (see above), except that the execution of `ExecutableNodes` within the `ConditionalNode` is specified by the `ConditionalNode` semantics described under this heading.

## Loop Nodes

A `LoopNode` is a `StructuredActivityNode` that represents an iterative loop. A `LoopNode` consists of a `setupPart`, a test and a `bodyPart`, which identify subsets of the `ExecutableNodes` contained in the `LoopNode`. Any `ExecutableNode` in the `LoopNode` must be included in the `setupPart`, test or `bodyPart` for the `LoopNode`.

When a `LoopNode` begins execution, any `InitialNodes` within it are immediately enabled. An `ExecutableNode` contained in the `LoopNode`, however, can only become enabled when the `setupPart`, test or `bodyPart` section that contains it is executed (as described below). When a section is executed, any `ExecutableNode` in the section that has no mandatory input data and no incoming `ControlFlow` with a source in the same section is enabled and receives a single control token. Execution then proceeds according to the usual semantics of `Activities`, except that any offers made to an `ExecutableNode` in a section that is not executing are not immediately delivered but remain pending. The target `ExecutableNode` may accept any pending offers if it eventually executes as part of a later execution of the section that contains it.

The `setupPart` of a `LoopNode` is executed first. When the `setupPart` has completed execution, the iterative execution of the loop begins. Execution of the test section may precede or follow execution of the `bodyPart`, depending on whether `isTestFirst` is true or false, respectively. The following description assumes that the test section is executed first (`isTestFirst=true`). If the `bodyPart` is

executed first (isTestFirst=false), it is always executed at least once, after which the following description applies to subsequent iterations.

The test section has an Action owning the decider OutputPin with type Boolean identified by the LoopNode. When the test section has completed execution, if the value on the decider OutputPin is true, then the bodyPart is executed. Otherwise, execution of the LoopNode is complete.

After each execution of the bodyPart, the test section is executed again, for the next iteration of the loop.

A LoopNode may also define a set of loopVariable OutputPins used to hold intermediate values during each loop iteration. These OutputPins may have outgoing ActivityEdges, in order to make the values they hold available within the test and bodyPart sections of a loop during an iteration. If a LoopNode has loopVariable OutputPins, then it must also have matching sets of loopVariableInput InputPins, bodyOutput OutputPins (owned by Actions within the bodyPart), and result OutputPins.

When the LoopNode begins executing, the tokens on the loopVariableInput InputPins are moved to the corresponding loopVariable OutputPins before the first iteration of the loop. After the completion of each execution of the bodyPart of the LoopNode, any remaining tokens on the loopVariable OutputPins are destroyed and tokens on the bodyOutput OutputPins are copied to the corresponding loopVariable OutputPins so that they are available for the next iteration. Once the test fails and the loop is completed, the tokens on the bodyOutput OutputPins from the last iteration are moved to the result OutputPins and offered on any edges outgoing from those OutputPins.

A LoopNode may not have any other InputPins or OutputPins than those described above. However, ActivityEdges may cross into and out of a LoopNode, as for StructuredActivityNodes in general, and the semantics are the same (see above), except that the execution of ExecutableNodes within the LoopNode is specified by the iterative looping semantics described above. In particular, a token accepted from an ActivityEdge crossing into a LoopNode on one iteration will be consumed and will *not* be available on the next iteration.

### Sequence Nodes

A SequenceNode defines a complete, sequential ordering of all the ActivityNodes it contains, which must all be ExecutableNodes. When the SequenceNode executes, each of the nodes within it are executed in sequential order. The SequenceNode may also contain ActivityEdges between its nodes, and ActivityEdges may cross into and out of the SequenceNode. The semantics are equivalent to a general StructuredActivityNode containing the same nodes and edges, but with ControlFlows added to sequentially order the nodes as specified for the SequenceNode.

#### 16.11.4 Notation

A StructuredActivityNode is notated with a dashed round cornered rectangle enclosing its nodes and edges, with the keyword «structured» at the top.

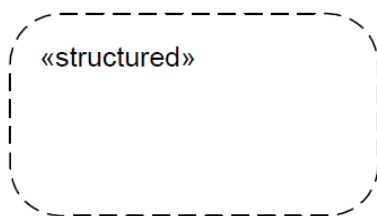


Figure 16.46 Notation for StructuredActivityNode

No standard notation is defined for ConditionalNodes, LoopNodes or SequenceNodes.

#### 16.11.5 Examples

None.

## 16.12 Expansion Regions

### 16.12.1 Summary

An ExpansionRegion is a StructuredActivityNode that executes its contained elements multiple times corresponding to elements of an input collection.

### 16.12.2 Abstract Syntax

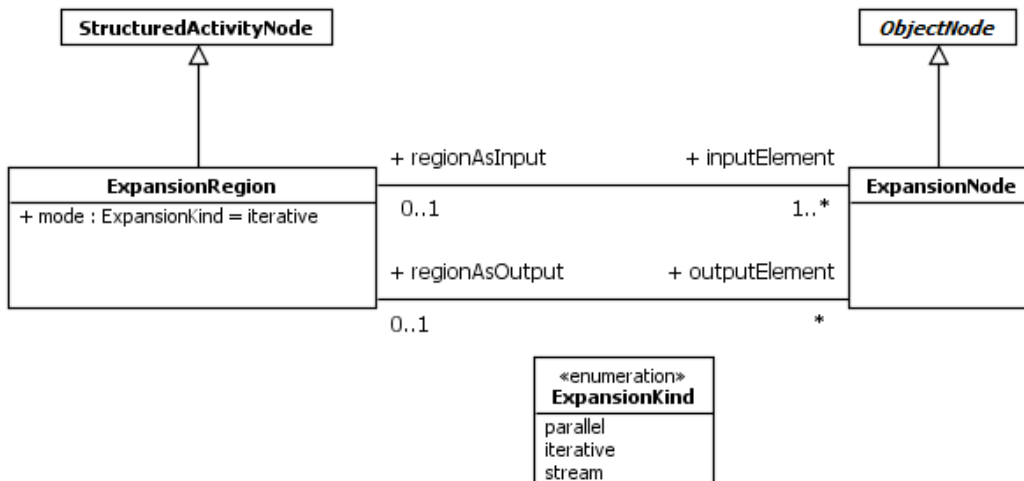


Figure 16.47 Expansion Regions

### 16.12.3 Semantics

An ExpansionRegion is a StructuredActivityNode that takes as input one or more collections of values and executes its contained ActivityNodes and ActivityEdges on each value in those collections. If the computation produces results, these may be collected into output collections. The number of output collections can differ from the number of input collections.

An ExpansionNode is an ObjectNode used to indicate a flow across the boundary of an ExpansionRegion. From “outside” of the region, the values on these nodes appear as collections. From “inside” the region the values appear as elements of the collections. A “collection” is defined to be any construct supported by an execution engine that may be treated either as a whole or as a well-defined set of element values.

An execution engine may define various kinds of collection types that it supports (sets, bags, and so on), individual instances of which may be constructed from element values and from which those element values may later be obtained. Such a collection instance is passed as a single value on a single token. An execution engine may alternatively support collections implicitly as the set of values passed in a group of tokens placed together on an ExpansionNode.

If an ExpansionRegion has multiple input ExpansionNodes, then each one must handle the same kind of collection (set, bag, or so on), although the types of the elements in different collections may vary. If the kind of collection is represented as a collection type, then this is used as the type of the ExpansionNodes. Otherwise, the type of the ExpansionNodes reflects the type of the elements in the collections.

An ExpansionRegion begins executing according to the normal rules for an Action (see sub clause 16.2.3). In addition, if the input ExpansionNodes for the ExpansionRegion have collection types, then a collection instance must be placed on each ExpansionNode before the ExpansionRegion may begin executing. Otherwise, there is no constraint on whether any input



ExpansionNode contains any tokens (as an ExpansionNode with no token is interpreted as the empty collection in this case). When the ExpansionRegion starts executing it removes all tokens in its input ExpansionNodes.

Then the group of ActivityNodes and ActivityEdges contained in the ExpansionRegion is executed once for each element of the input collections. These will be referred to as the *expansion executions* for the ExpansionRegion. If the collections have different numbers of elements, then the number of expansion executions is equal to the size of the smallest collection (except in the case of mode=stream, in which case there is only one expansion execution, as discussed later). Each of the expansion executions proceeds independently from the other executions, with the same semantics as the execution of a general StructureActivityNode, except for the following special rules:

- Within each expansion execution, a single token is offered on each ActivityEdge with an input ExpansionNode as its source and its target inside the ExpansionRegion. This token contains as its value an individual element of the collection on the input ExpansionNode. For each collection on each such input ExpansionNode, a different element is offered for each expansion execution. If the collection is not a set (non-unique), duplicate values are considered to be different elements. If the collections are ordered, then the elements from each collection are aligned in order for each execution (one execution gets all elements from position 1 in the input collections, another gets all elements from position 2, and so on), up to the number of executions (this provides an effective ordering of the expansion executions). If the collections are not ordered, then it is undefined which individual elements of a collection are delivered to which execution (except that no element is delivered to more than one execution).
- Each expansion execution may result in tokens offered to an ActivityEdge with its source inside the ExpansionRegion and with an output ExpansionNode as its target. Such tokens are immediately accepted by the ExpansionNode and inserted into the output collection for that ExpansionNode. If the input and output collections are both ordered, then the values provided by each execution are concatenated in the same order as is induced on the executions by the input collections. If each execution produces a single value, then the output collection will have the same number of elements as the smallest input collection, and, if the output collection is ordered, it will have an output at each position corresponding to the input at the same position of the input collections. On the other hand, if each execution may or may not produce a value, then the output collection will have fewer elements than the input collections and the ExpansionRegion will act as a kind of filter. Finally, if each execution can produce more than one value, then the output collection may end up with a greater number of elements than the input collections.
- Tokens placed on the InputPins of an ExpansionRegion are duplicated for each expansion execution, so that each execution is offered a different copy of the tokens on outgoing ActivityEdges from the InputPins. In this way, tokens consumed from an InputPin in one expansion execution do not affect the tokens available from the InputPin in other executions (the tokens on the InputPin effectively appear to be “constant” across the executions). Similarly, tokens offered on ActivityEdges that cross into an ExpansionRegion from outside it (other than those to or from ExpansionNodes of the ExpansionRegion) are duplicated for each expansion execution (the target of each ActivityEdge is offered a separate copy of the tokens within each expansion execution).
- ExpansionRegions may also have OutputPins and ActivityEdges that cross out of the ExpansionRegion from inside it. However, the semantics are undefined for offering tokens to such OutputPins or ActivityEdges from within the expansion executions of the ExpansionRegion (other than for ActivityEdges to or from ExpansionNodes of the ExpansionRegion).

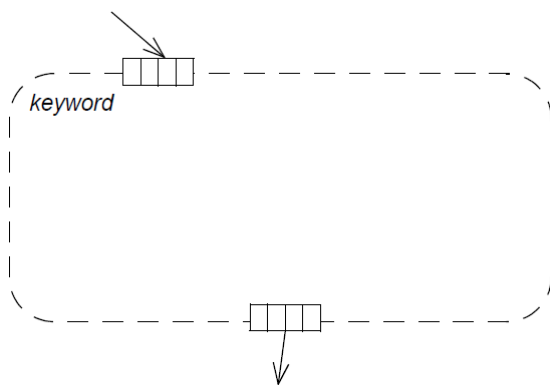
When the ExpansionRegion completes all expansion executions, it offers the output collections on its output ExpansionNodes on any ActivityEdges outgoing from those nodes (they are not offered during the execution of the ExpansionRegion). If the ExpansionRegion contains an ActivityFinalNode immediately within it, then, if the ActivityFinalNode accepts a token within *any* expansion execution, *all* currently ongoing expansion executions are terminated and the ExpansionRegion as a whole completes its execution. In this case, output collections are still offered from the output ExpansionNodes, though the collections may only be partially filled.

The mode of an ExpansionRegion controls how its expansion executions proceed.

- If the value is *parallel*, the expansion executions proceed concurrently. This allows an execution engine to run the executions in parallel, or otherwise overlapping in time, but this is not required. However, if the executions are run sequentially, then the order in which they are run is not defined.
- If the value is *iterative*, the expansion executions must occur in an iterative sequence, with one completing before another can begin. The first expansion execution begins immediately when the ExpansionRegion starts executing, with subsequent executions starting when the previous execution is completed. If the input collections are ordered, then the expansion executions are sequenced in the order induced by the input collection. Otherwise, the order of the expansion executions is not defined.
- If the value is *stream*, there is exactly one expansion execution, and element values are offered to this execution in a stream from each collection. That is, each element of a collection on an input ElementNode is offered separately as a token, one by one, on all outgoing ActivityEdges from the ExpansionRegion (up to a number of tokens equal to the size of the smallest input collection). If the input collections are ordered, then this sequence of offers is made in the same order as the elements of each collection; otherwise the order is not defined. During the course of the single expansion execution, multiple tokens may be accepted by each output ExpansionNode in order to construct the output collections from the ExpansionRegion. If an output collection is ordered, then the elements of the collection are ordered corresponding to the order in which tokens are received by the ExpansionNode.

#### 16.12.4 Notation

An ExpansionRegion is shown as a dashed rounded box with one of the keywords «parallel», «iterative» or «stream» in the upper left corner (see Figure 16.48). Input and output ExpansionNodes are drawn as small rectangles divided by vertical bars into small compartments. (The symbol is meant to suggest a list of elements.) The ExpansionNode symbols are placed on the boundary of the dashed box. Usually, ActivityEdge arrows inside and outside the ExpansionRegion will distinguish input and output expansion nodes. If not, then a small arrow can be used as with Pins (Figure 16.5).



**Figure 16.48 Expansion Region**

As a shorthand notation, the ExpansionNode “list box” notation may be placed directly on an Action symbol, replacing the pins of the action (Figure 16.49). This indicates an expansion region containing a single Action. The equivalent full form is shown in Figure 16.50. In the shorthand notation, there must be one input ExpansionNode corresponding to each in or inout parameter of the behavior (which must have at least one such parameter) and one output ExpansionNode corresponding to each out, inout, or return parameter of the behavior.

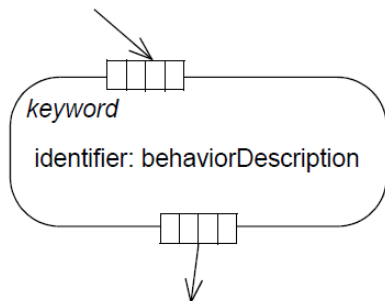


Figure 16.49 Shorthand notation for expansion region containing single node

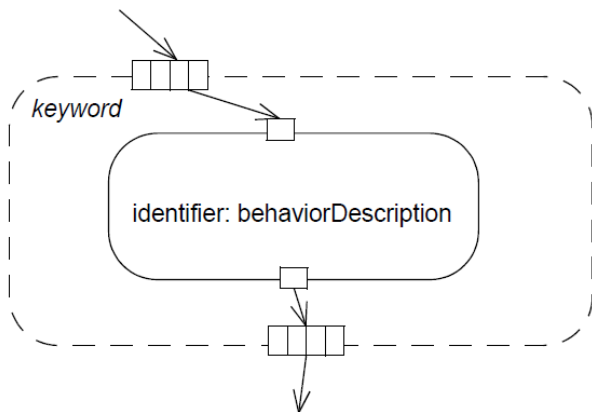


Figure 16.50 Full form of previous shorthand notation

Figure 16.51 shows a further shorthand for an ExpansionRegion that contains a single CallBehaviorAction. This is shown using the shorthand notation of Figure 16.49, but, instead of using a mode keyword, a “\*” is placed in the upper right-hand corner of the symbol (this is intended to indicate “multiple execution.” The notation maps to an expansion region containing the CallBehaviorAction (as in Figure 16.50) with mode=parallel.

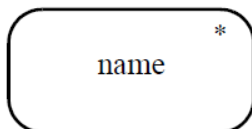
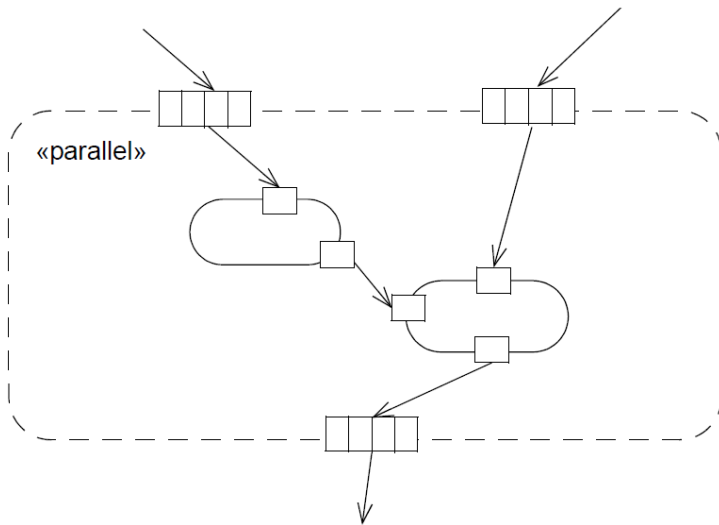


Figure 16.51 Notation for expansion region with one behavior invocation

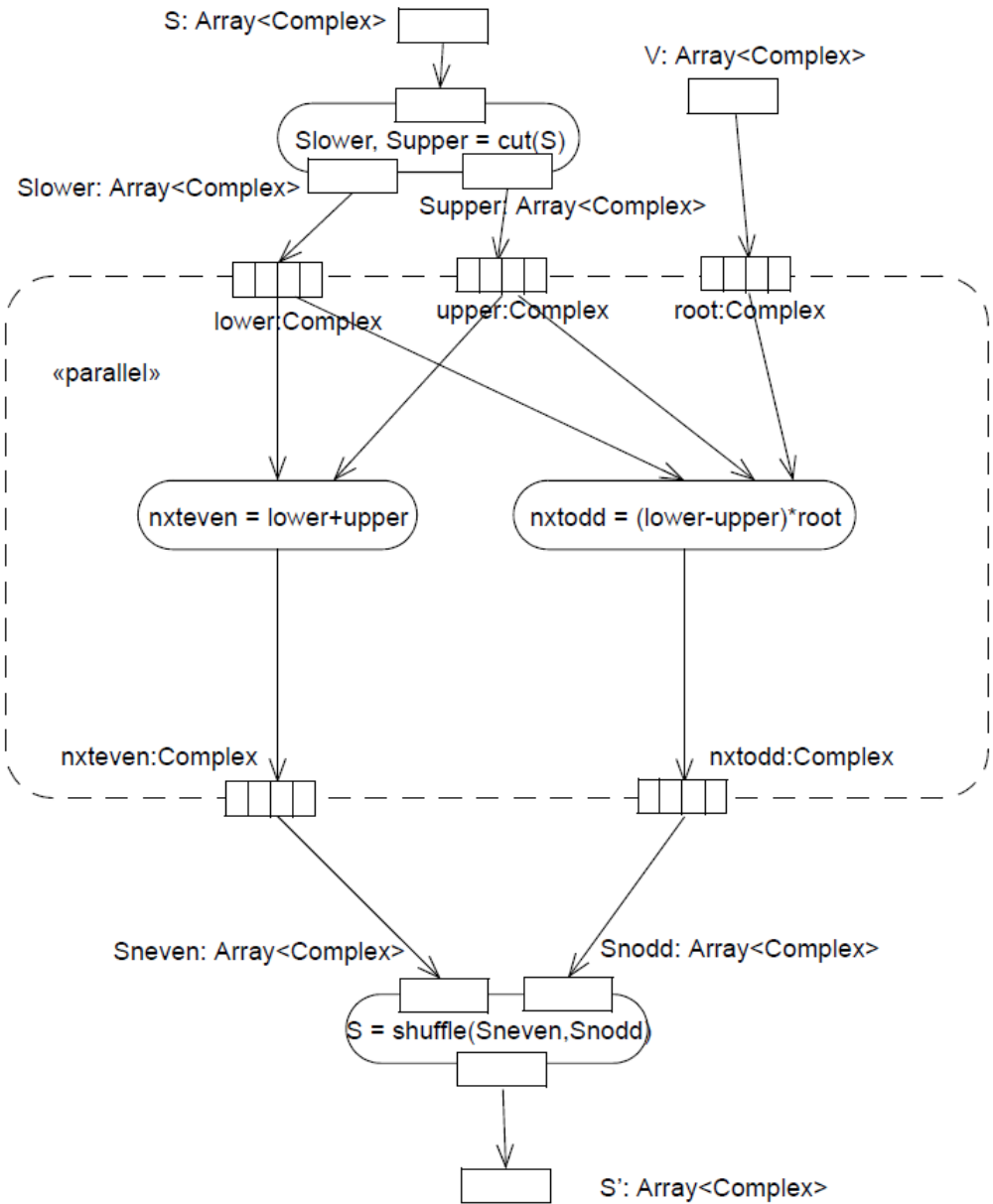
### 16.12.5 Examples

Figure 16.52 shows an ExpansionRegion with two inputs and one output that is executed in parallel. Execution of the ExpansionRegion does not begin until both input collections are available. Both collections are expected to have the same number of elements. The interior of the ExpansionRegion is executed once for each element in the input collections. During each execution of the region, a pair of values, one from each collection, is available to the interior from the input ExpansionNodes. Each expansion execution produces a result value on the output ExpansionNode. All of the result values are formed into a collection of the same size as the input collections. This output collection is available outside the ExpansionRegion on the output ExpansionNode after all the parallel expansionexecutions have completed.



**Figure 16.52 Expansion region with two inputs and one output**

Figure 16.53 shows a fragment of a Fast Fourier Transform (FFT) computation containing an ExpansionRegion. Outside the ExpansionRegion, there are operations on arrays of complex numbers. S, Slower, Supper, and V are arrays. Cut and shuffle are operations on arrays. Inside the region, two arithmetic operations are performed on elements of the three input arrays, yielding two output arrays. Different positions in the arrays do not interact, therefore the ExpansionRegion can be executed in parallel on all positions.



**Figure 16.53 Expansion Region**

Figure 16.54 shows a use of the shorthand notation for an ExpansionRegion with a single Action. In this example, the Specify Trip Route action outputs sets of flights and sets of hotels to book. The hotels may be booked independently and in parallel with each other and with booking the flight.

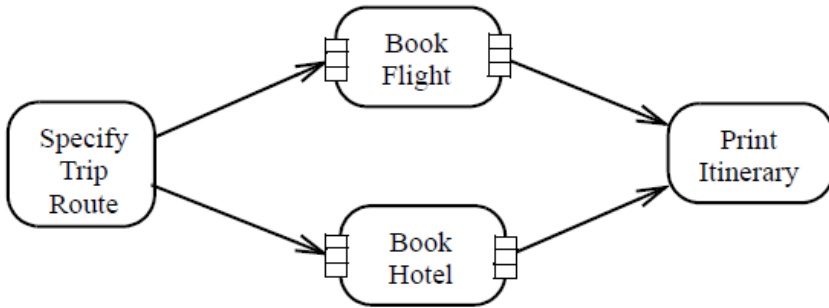


Figure 16.54 Examples of expansion region shorthand

In Figure 16.55, Specify Trip Route can result in multiple flight segments, each of which must be booked separately. The Book Flight action will invoke the Book Flight Behavior multiple times, once for each flight segment in the set passed to Book Flight.

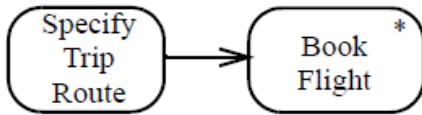


Figure 16.55 Shorthand notation for expansion region

## 16.13 Other Actions

### 16.13.1 Summary

This subclause covers two additional kinds of Actions: ReduceActions for repeatedly invoking a Behavior to reduce a collection of values to a single value, and RaiseExceptionAction for raising exceptions.

### 16.13.2 Abstract Syntax

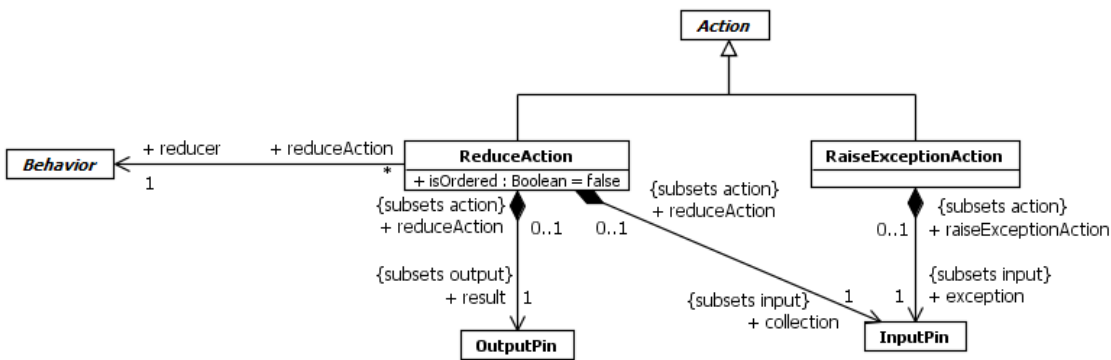


Figure 16.56 Other Actions

### 16.13.3 Semantics

#### Reduce Actions

A ReduceAction is an Action that reduces a collection to a single value by combining the elements of the collection. The input collection is given on the collection InputPin. The reducer Behavior must have two in parameters and a single out or return parameter, with the same type as the elements of the collection. It has a single result OutputPin which also has the collection element type.

An execution engine may define various kinds of collection types that it supports (set, bag, and so on), individual instances of which may be constructed from element values and from which those element values may later be obtained. Such a collection instance is passed as a single value on a single token. In this case, the collection InputPin of a ReduceAction should have a collection type as its type and a multiplicity of 1..1.

Alternatively, an execution engine may support collections implicitly as the set of values passed in a group of tokens placed together on an InputPin. In this case, the collection InputPin of a ReduceAction should have the same type as the elements of the collection and an upper multiplicity greater than 1 (usually \*).

A ReduceAction executes by repeatedly invoking the reducer Behavior on an intermediate copy of the input collection. For each invocation of the reducer, two elements are removed from the intermediate collection to act as arguments for the invocation. The value returned from the invocation is then inserted into the intermediate collection, so its size is reduced by one from before the invocation, and the reducer Behavior is invoked again. This process continues until the collection has only one element. This element is then placed on the result OutputPin and the ReduceAction execution completes.

If the input collection is unordered, or if isOrdered for the InputPin is false, it is indeterminate which elements of the intermediate collection are selected as arguments for the reducer invocation. If the input collection is ordered, or if isOrdered for the InputPin is true, then the first two elements of the intermediate collection are always used as the arguments of the reducer invocation, and the result of the invocation is always inserted as the first element of the collection.

If the reducer Behavior is a commutative and associative operation, then an unordered collection or isOrdered=false will not normally affect the result of a ReduceAction, giving greater freedom in how the reduction computation may be carried out. For example, addition is commutative ( $a + b = b + a$ ) and associative ( $(a + b) + c = a + (b + c)$ ), so reducing a collection using addition will produce the same result regardless of how elements are paired at each reducer invocation. If the reducer Behavior is not a commutative and associative operation (as, for example, with matrix multiplication), the order in which elements of the intermediate collection are selected will affect the result of the reduction computation. If it is desired to avoid nondeterminacy in this case, collections may be ordered or isOrdered set to true, so the reducer Behavior will be applied to adjacent pairs according to the collection order.

If the reducer Behavior has side effects such that invocations of it may affect each other, then the result a ReduceAction with isOrdered=false may be unpredictable.

#### Raise Exception Actions

A RaiseExceptionAction is an Action that causes an exception to occur. A RaiseExceptionAction always completes by raising an exception, rather than normally. The value given on the exception InputPin is raised as the exception.

If the RaiseExceptionAction itself has an ExceptionHandler (see sub clause [15.5](#)) that matches the raised exception, then the exception is caught by that handler. Otherwise, the exception propagates outward to the innermost containing StructuredActivityNode of the RaiseExceptionAction. If this StructuredActivityNode has an ExceptionHandler that matches the raised exception, then the exception is caught by that handler, otherwise the exception continues to be propagated outward.

When an exception is propagated out of a StructuredActivityNode (including being caught by an ExceptionHandler on the node), the StructuredActivityNode is terminated. If the exception is caught by an ExceptionHandler on the StructuredActivityNode, then after execution of the handler, control and object tokens are offered from the StructuredActivityNode as described in sub clause

[15.5.3](#). Otherwise, the StructuredActivityNode does not offer any control tokens and its OutputPins do not offer any object tokens.

If the exception is not caught by any ExceptionHandler at some level within the Behavior in which the RaiseExceptionAction was executed, then the execution of the Behavior is terminated. If the Behavior was invoked synchronously, then the exception is propagated to the caller of the Behavior, out from that invocation (see also the discussion in sub clause 16.3.3 of exceptions raised by CallActions making synchronous calls). If the Behavior was invoked asynchronously, then the exception propagation ends with the termination of the Behavior execution.

#### **16.13.4 Notation**

No specialized notation is defined for ReduceActions and RaiseExceptionActions.

#### **16.13.5 Examples**

A ReduceAction can be used to reduce a list of numbers to the sum of the numbers. Such a ReduceAction has one InputPin for a collection of numbers, one OutputPin for a number and an addition function as the reducer Behavior. For example, suppose the input collection has four integers: (2, 7, 5, 3). The result of applying the ReduceAction to this collection with an addition function is 11. With the default of `isOrdered=false`, this can be computed in a number of ways, for example,  $((2+7) + 5) + 3$ ,  $2 + (7 + (5 + 3))$ ,  $((2 + 7) + (5 + 3))$ .



## 16.14 Classifier Descriptions

### AcceptCallAction [Class]

#### Description

An AcceptCallAction is an AcceptEventAction that handles the receipt of a synchronous call request. In addition to the values from the Operation input parameters, the Action produces an output that is needed later to supply the information to the ReplyAction necessary to return control to the caller. An AcceptCallAction is for synchronous calls. If it is used to handle an asynchronous call, execution of the subsequent ReplyAction will complete immediately with no effect.

#### Diagrams

[Accept Event Actions](#)

#### Generalizations

[AcceptEventAction](#)

#### Association Ends

- ◆ returnInformation : [OutputPin](#) [1..1]{subsets [Action::output](#)} (opposite [A\\_returnInformation\\_acceptCallAction::acceptCallAction](#))  
An OutputPin where a value is placed containing sufficient information to perform a subsequent ReplyAction and return control to the caller. The contents of this value are opaque. It can be passed and copied but it cannot be manipulated by the model.

#### Constraints

- result\_pins  
The number of result OutputPins must be the same as the number of input (in and inout) ownedParameters of the Operation specified by the trigger Event. The type, ordering and multiplicity of each result OutputPin must be consistent with the corresponding input Parameter.

```
inv: let parameter: OrderedSet(Parameter) = trigger.event->asSequence()->first().oclAsType(CallEvent).operation.inputParameters() in
result->size() = parameter->size() and
Sequence{1..result->size()}->forall(i |
  parameter->at(i).type.conformsTo(result->at(i).type) and
  parameter->at(i).isOrdered = result->at(i).isOrdered and
  parameter->at(i).compatibleWith(result->at(i)))
```

- trigger\_call\_event  
The action must have exactly one trigger, which must be for a CallEvent.

```
inv: trigger->size()==1 and
trigger->asSequence()->first().event.oclIsKindOf(CallEvent)
```

- unmarshall  
isUnmrashall must be true for an AcceptCallAction.

```
inv: isUnmarshall = true
```

## AcceptEventAction [Class]

### Description

An AcceptEventAction is an Action that waits for the occurrence of one or more specific Events.

### Diagrams

[Accept Event Actions](#)

### Generalizations

[Action](#)

### Specializations

[AcceptCallAction](#)

### Attributes

- isUnmarshall : [Boolean](#) [1..1] = false  
Indicates whether there is a single OutputPin for a SignalEvent occurrence, or multiple OutputPins for attribute values of the instance of the Signal associated with a SignalEvent occurrence.

### Association Ends

- ♦ result : [OutputPin](#) [0..\*]{ordered, subsets [Action::output](#)} (opposite [A result acceptEventAction::acceptEventAction](#))  
OutputPins holding the values received from an Event occurrence.
- ♦ trigger : [Trigger](#) [1..\*]{subsets [Element::ownedElement](#)} (opposite [A trigger acceptEventAction::acceptEventAction](#))  
The Triggers specifying the Events of which the AcceptEventAction waits for occurrences.

### Constraints

- one\_output\_pin  
If isUnmarshall=false and any of the triggers are for SignalEvents or TimeEvents, there must be exactly one result OutputPin with multiplicity 1..1.

```
inv: not isUnmarshall and trigger->exists(event.oclIsKindOf(SignalEvent) or
event.oclIsKindOf(TimeEvent)) implies
  output->size() = 1 and output->first().is(1,1)
```

- no\_input\_pins  
AcceptEventActions may have no input pins.  
  
inv: input->size() = 0
- no\_output\_pins  
There are no OutputPins if the trigger events are only ChangeEvents and/or CallEvents when this action is an instance of AcceptEventAction and not an instance of a descendant of AcceptEventAction (such as AcceptCallAction).

```

inv: (self.oclIsTypeOf(AcceptEventAction) and
      (trigger->forall(event.oclIsKindOf(ChangeEvent) or
                      event.oclIsKindOf(CallEvent))))
implies output->size() = 0

```

- **unmarshall\_signal\_events**

If isUnmarshall is true (and this is not an AcceptCallAction), there must be exactly one trigger, which is for a SignalEvent. The number of result output pins must be the same as the number of attributes of the signal. The type and ordering of each result output pin must be the same as the corresponding attribute of the signal. The multiplicity of each result output pin must be compatible with the multiplicity of the corresponding attribute.

```

inv: isUnmarshall and self.oclIsTypeOf(AcceptEventAction) implies
      trigger->size()=1 and
      trigger->asSequence()->first().event.oclIsKindOf(SignalEvent) and
      let attribute: OrderedSet(Property) = trigger->asSequence()-
      >first().event.oclAsType(SignalEvent).signal.allAttributes() in
      attribute->size()>0 and result->size() = attribute->size() and
      Sequence{1..result->size()}->forall(i |
        result->at(i).type = attribute->at(i).type and
        result->at(i).isOrdered = attribute->at(i).isOrdered and
        result->at(i).includesMultiplicity(attribute->at(i)))

```

- **conforming\_type**

If isUnmarshall=false and all the triggers are for SignalEvents, then the type of the single result OutputPin must either be null or all the signals must conform to it.

```

inv: not isUnmarshall implies
      result->isEmpty() or
      let type: Type = result->first().type in
      type=null or
      (trigger->forall(event.oclIsKindOf(SignalEvent)) and
       trigger.event.oclAsType(SignalEvent).signal->forall(s | s.conformsTo(type)))

```

## Action [Abstract Class]

### Description

An Action is the fundamental unit of executable functionality. The execution of an Action represents some transformation or processing in the modeled system. Actions provide the ExecutableNodes within Activities and may also be used within Interactions.

### Diagrams

[Actions](#), [Invocation Actions](#), [Link Actions](#), [Link Object Actions](#), [Structural Feature Actions](#), [Accept Event Actions](#), [Other Actions](#), [Variable Actions](#), [Structured Actions](#), [Object Actions](#), [Interactions](#), [Occurrences](#)

### Generalizations

[ExecutableNode](#)

### Specializations

[ValueSpecificationAction](#), [VariableAction](#), [AcceptEventAction](#), [ClearAssociationAction](#), [CreateObjectAction](#), [DestroyObjectAction](#), [InvocationAction](#), [LinkAction](#), [OpaqueAction](#), [RaiseExceptionAction](#), [ReadExtentAction](#), [ReadIsClassifiedObjectAction](#), [ReadLinkObjectEndAction](#), [ReadLinkObjectEndQualifierAction](#), [ReadSelfAction](#),

[ReclassifyObjectAction](#), [ReduceAction](#), [ReplyAction](#), [StartClassifierBehaviorAction](#), [StructuralFeatureAction](#), [StructuredActivityNode](#), [TestIdentityAction](#), [UnmarshallAction](#)

## Attributes

- isLocallyReentrant : [Boolean](#) [1..1] = false  
If true, the Action can begin a new, concurrent execution, even if there is already another execution of the Action ongoing. If false, the Action cannot begin a new execution until any previous execution has completed.

## Association Ends

- /context : [Classifier](#) [0..1]{ } (opposite [A\\_context\\_action::action](#))  
The context Classifier of the Behavior that contains this Action, or the Behavior itself if it has no context.
- ♦ /input : [InputPin](#) [0..\*]{ordered, union, subsets [Element::ownedElement](#)} (opposite [A\\_input\\_action::action](#))  
The ordered set of InputPins representing the inputs to the Action.
- ♦ localPostcondition : [Constraint](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [A\\_localPostcondition\\_action::action](#))  
A Constraint that must be satisfied when execution of the Action is completed.
- ♦ localPrecondition : [Constraint](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [A\\_localPrecondition\\_action::action](#))  
A Constraint that must be satisfied when execution of the Action is started.
- ♦ /output : [OutputPin](#) [0..\*]{ordered, union, subsets [Element::ownedElement](#)} (opposite [A\\_output\\_action::action](#))  
The ordered set of OutputPins representing outputs from the Action.

## Operations

- context() : [Classifier](#) [0..1]  
The derivation for the context property.  

```
body: let behavior: Behavior = self.containingBehavior() in
if behavior=null then null
else if behavior._'context' = null then behavior
else behavior._'context'
endif
endif
```
- allActions() : [Action](#) [0..\*]  
Return this Action and all Actions contained directly or indirectly in it. By default only the Action itself is returned, but the operation is overridden for StructuredActivityNodes.

```
body: self->asSet()
```

- allOwnedNodes() : [ActivityNode](#) [0..\*]  
Returns all the ActivityNodes directly or indirectly owned by this Action. This includes at least all the Pins of the Action.

```
body: input.oclassType(Pin)->asSet()->union(output->asSet())
```

- containingBehavior() : [Behavior](#) [0..1]

```
body: if inStructuredNode<>null then inStructuredNode.containingBehavior()
else if activity<>null then activity
else interaction
endif
endif
```

## ActionInputPin [Class]

### Description

An ActionInputPin is a kind of InputPin that executes an Action to determine the values to input to another Action.

### Diagrams

[Actions](#)

### Generalizations

[InputPin](#)

### Association Ends

- ♦ fromAction : [Action](#) [1..1]{subsets [Element::ownedElement](#)} (opposite [A fromAction actionInputPin::actionInputPin](#))  
The Action used to provide the values of the ActionInputPin.

### Constraints

- input\_pin  
The fromAction of an ActionInputPin must only have ActionInputPins as InputPins.

```
inv: fromAction.input->forall(oclIsKindOf(ActionInputPin))
```

- one\_output\_pin  
The fromAction of an ActionInputPin must have exactly one OutputPin.

```
inv: fromAction.output->size() = 1
```

- no\_control\_or\_object\_flow  
The fromAction of an ActionInputPin cannot have ActivityEdges coming into or out of it or its Pins.

```
inv: fromAction.incoming->union(outgoing)->isEmpty() and
fromAction.input.incoming->isEmpty() and
fromAction.output.outgoing->isEmpty()
```

## AddStructuralFeatureValueAction [Class]

### Description

An AddStructuralFeatureValueAction is a WriteStructuralFeatureAction for adding values to a StructuralFeature.

## Diagrams

[Structural Feature Actions](#)

## Generalizations

[WriteStructuralFeatureAction](#)

## Attributes

- isReplaceAll : [Boolean](#) [1..1] = false  
Specifies whether existing values of the StructuralFeature should be removed before adding the new value.

## Association Ends

- ♦ insertAt : [InputPin](#) [0..1]{subsets [Action::input](#)} (opposite [A\\_insertAt\\_addStructuralFeatureValueAction::addStructuralFeatureValueAction](#))  
The InputPin that gives the position at which to insert the value in an ordered StructuralFeature. The type of the insertAt InputPin is UnlimitedNatural, but the value cannot be zero. It is omitted for unordered StructuralFeatures.

## Constraints

- required\_value  
A value InputPin is required.  
  
`inv: value<>null`
- insertAt\_pin  
AddStructuralFeatureActions adding a value to ordered StructuralFeatures must have a single InputPin for the insertion point with type UnlimitedNatural and multiplicity of 1..1 if isReplaceAll=false, and must have no Input Pin for the insertion point when the StructuralFeature is unordered.

```
inv: if not structuralFeature.isOrdered then insertAt = null
else
  not isReplaceAll implies
    insertAt<>null and
    insertAt->forall(type=UnlimitedNatural and is(1,1.oclAsType(UnlimitedNatural)))
endif
```

## AddVariableValueAction [Class]

### Description

An AddVariableValueAction is a WriteVariableAction for adding values to a Variable.

### Diagrams

[Variable Actions](#)

## Generalizations

[WriteVariableAction](#)

## Attributes

- isReplaceAll : [Boolean](#) [1..1] = false  
Specifies whether existing values of the Variable should be removed before adding the new value.

## Association Ends

- ♦ insertAt : [InputPin](#) [0..1]{subsets [Action::input](#)} (opposite [A insertAt addVariableValueAction::addVariableValueAction](#))  
The InputPin that gives the position at which to insert a new value or move an existing value in ordered Variables. The type of the insertAt InputPin is UnlimitedNatural, but the value cannot be zero. It is omitted for unordered Variables.

## Constraints

- required\_value  
A value InputPin is required.  
  
`inv: value <> null`
- insertAt\_pin  
AddVariableValueActions for ordered Variables must have a single InputPin for the insertion point with type UnlimitedNatural and multiplicity of 1..1 if isReplaceAll=false, otherwise the Action has no InputPin for the insertion point.

```
inv: if not variable.isOrdered then insertAt = null
else
  not isReplaceAll implies
    insertAt<>null and
    insertAt->forall(type=UnlimitedNatural and is(1,1.oclAsType(UnlimitedNatural)))
endif
```

## BroadcastSignalAction [Class]

### Description

A BroadcastSignalAction is an InvocationAction that transmits a Signal instance to all the potential target objects in the system. Values from the argument InputPins are used to provide values for the attributes of the Signal. The requestor continues execution immediately after the Signal instances are sent out and cannot receive reply values.

### Diagrams

[Invocation Actions](#)

## Generalizations

[InvocationAction](#)

## Association Ends

- signal : [Signal](#) [1..1] (opposite [A signal broadcastSignalAction::broadcastSignalAction](#))  
The Signal whose instances are to be sent.

## Constraints

- number\_of\_arguments  
The number of argument InputPins must be the same as the number of attributes in the signal.

```
inv: argument->size() = signal.allAttributes()->size()
```

- type\_ordering\_multiplicity  
The type, ordering, and multiplicity of an argument InputPin must be the same as the corresponding attribute of the signal.

```
inv: let attribute: OrderedSet(Property) = signal.allAttributes() in  
Sequence{1..argument->size()}->forall(i |  
  argument->at(i).type.conformsTo(attribute->at(i).type) and  
  argument->at(i).isOrdered = attribute->at(i).isOrdered and  
  argument->at(i).compatibleWith(attribute->at(i)))
```

- no\_onport  
A BroadcastSignalAction may not specify onPort.

```
inv: onPort=null
```

## CallAction [Abstract Class]

### Description

CallAction is an abstract class for Actions that invoke a Behavior with given argument values and (if the invocation is synchronous) receive reply values.

### Diagrams

[Invocation Actions](#)

### Generalizations

[InvocationAction](#)

### Specializations

[CallBehaviorAction](#), [CallOperationAction](#), [StartObjectBehaviorAction](#)

### Attributes

- isSynchronous : [Boolean](#) [1..1] = true  
If true, the call is synchronous and the caller waits for completion of the invoked Behavior. If false, the call is



asynchronous and the caller proceeds immediately and cannot receive return values.

## Association Ends

- ♦ result : [OutputPin](#) [0..\*]{ordered, subsets [Action::output](#)} (opposite [A\\_result callAction::callAction](#))  
The OutputPins on which the reply values from the invocation are placed (if the call is synchronous).

## Operations

- inputParameters() : [Parameter](#) [0..\*]  
Return the in and inout ownedParameters of the Behavior or Operation being called. (This operation is abstract and should be overridden by subclasses of CallAction.)
- outputParameters() : [Parameter](#) [0..\*]  
Return the inout, out and return ownedParameters of the Behavior or Operation being called. (This operation is abstract and should be overridden by subclasses of CallAction.)

## Constraints

- argument\_pins  
The number of argument InputPins must be the same as the number of input (in and inout) ownedParameters of the called Behavior or Operation. The type, ordering and multiplicity of each argument InputPin must be consistent with the corresponding input Parameter.

```
inv: let parameter: OrderedSet(Parameter) = self.inputParameters() in
argument->size() = parameter->size() and
Sequence{1..argument->size()}->forall(i |
  argument->at(i).type.conformsTo(parameter->at(i).type) and
  argument->at(i).isOrdered = parameter->at(i).isOrdered and
  argument->at(i).compatibleWith(parameter->at(i)))
```

- result\_pins  
The number of result OutputPins must be the same as the number of output (inout, out and return) ownedParameters of the called Behavior or Operation. The type, ordering and multiplicity of each result OutputPin must be consistent with the corresponding input Parameter.

```
inv: let parameter: OrderedSet(Parameter) = self.outputParameters() in
result->size() = parameter->size() and
Sequence{1..result->size()}->forall(i |
  parameter->at(i).type.conformsTo(result->at(i).type) and
  parameter->at(i).isOrdered = result->at(i).isOrdered and
  parameter->at(i).compatibleWith(result->at(i)))
```

- synchronous\_call  
Only synchronous CallActions can have result OutputPins.

```
inv: result->notEmpty() implies isSynchronous
```

## CallBehaviorAction [Class]

### Description

A CallBehaviorAction is a CallAction that invokes a Behavior directly. The argument values of the CallBehaviorAction are passed on the input Parameters of the invoked Behavior. If the call is synchronous, the execution of the CallBehaviorAction waits until the execution of the invoked Behavior completes and the values of output Parameters of the Behavior are placed on the result OutputPins. If the call is asynchronous, the CallBehaviorAction completes immediately and no results values can be provided.

### Diagrams

[Invocation Actions](#)

### Generalizations

[CallAction](#)

### Association Ends

- behavior : [Behavior](#) [1..1] (opposite [A\\_behavior\\_callBehaviorAction::callBehaviorAction](#))  
The Behavior being invoked.

### Operations

- outputParameters() : [Parameter](#) [0..\*]  
Return the inout, out and return ownedParameters of the Behavior being called.

```
body: behavior.outputParameters()
```

- inputParameters() : [Parameter](#) [0..\*]  
Return the in and inout ownedParameters of the Behavior being called.

```
body: behavior.inputParameters()
```

### Constraints

- no\_onport  
A CallBehaviorAction may not specify onPort.

```
inv: onPort=null
```

## CallOperationAction [Class]

### Description

A CallOperationAction is a CallAction that transmits an Operation call request to the target object, where it may cause the invocation of associated Behavior. The argument values of the CallOperationAction are passed on the input Parameters of the Operation. If call is synchronous, the execution of the CallOperationAction waits until the execution of the invoked Operation

completes and the values of output Parameters of the Operation are placed on the result OutputPins. If the call is asynchronous, the CallOperationAction completes immediately and no results values can be provided.

## Diagrams

[Invocation Actions](#)

## Generalizations

[CallAction](#)

## Association Ends

- operation : [Operation](#) [1..1] (opposite [A operation callOperationAction::callOperationAction](#))  
The Operation being invoked.
- ♦ target : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A target callOperationAction::callOperationAction](#))  
The InputPin that provides the target object to which the Operation call request is sent.

## Operations

- outputParameters() : [Parameter](#) [0..\*]  
Return the inout, out and return ownedParameters of the Operation being called.

```
body: operation.outputParameters()
```

- inputParameters() : [Parameter](#) [0..\*]  
Return the in and inout ownedParameters of the Operation being called.

```
body: operation.inputParameters()
```

## Constraints

- type\_target\_pin  
If onPort has no value, the operation must be an owned or inherited feature of the type of the target InputPin, otherwise the Port given by onPort must be an owned or inherited feature of the type of the target InputPin, and the Port must have a required or provided Interface with the operation as an owned or inherited feature.

```
inv: if onPort=null then target.type.oclassType(Classifier).allFeatures()->includes(operation)
else target.type.oclassType(Classifier).allFeatures()->includes(onPort) and onPort.provided-
>union(onPort.required).allFeatures()->includes(operation)
endif
```

## Clause [Class]

### Description

A Clause is an Element that represents a single branch of a ConditionalNode, including a test and a body section. The body section is executed only if (but not necessarily if) the test section evaluates to true.

## Diagrams

[Structured Actions](#)

## Generalizations

[Element](#)

## Association Ends

- body : [ExecutableNode](#) [0..\*] (opposite [A\\_body\\_clause::clause](#))  
The set of ExecutableNodes that are executed if the test evaluates to true and the Clause is chosen over other Clauses within the ConditionalNode that also have tests that evaluate to true.
- bodyOutput : [OutputPin](#) [0..\*]{ordered} (opposite [A\\_bodyOutput\\_clause::clause](#))  
The OutputPins on Actions within the body section whose values are moved to the result OutputPins of the containing ConditionalNode after execution of the body.
- decider : [OutputPin](#) [1..1] (opposite [A\\_decider\\_clause::clause](#))  
An OutputPin on an Action in the test section whose Boolean value determines the result of the test.
- predecessorClause : [Clause](#) [0..\*] (opposite [Clause::successorClause](#))  
A set of Clauses whose tests must all evaluate to false before this Clause can evaluate its test.
- successorClause : [Clause](#) [0..\*] (opposite [Clause::predecessorClause](#))  
A set of Clauses that may not evaluate their tests unless the test for this Clause evaluates to false.
- test : [ExecutableNode](#) [1..\*] (opposite [A\\_test\\_clause::clause](#))  
The set of ExecutableNodes that are executed in order to provide a test result for the Clause.

## Constraints

- body\_output\_pins  
The bodyOutput Pins are OutputPins on Actions in the body of the Clause.  

```
inv: _('body'.oclAsType(Action).allActions().output->includesAll(bodyOutput)
```
- decider\_output  
The decider Pin must be on an Action in the test section of the Clause and must be of type Boolean with multiplicity 1..1.  

```
inv: test.oclAsType(Action).allActions().output->includes(decider) and  
decider.type = Boolean and  
decider.is(1,1)
```
- test\_and\_body  
The test and body parts of a ConditionalNode must be disjoint with each other.  

```
inv: test->intersection(_('body')->isEmpty()
```

## ClearAssociationAction [Class]

### Description

A ClearAssociationAction is an Action that destroys all links of an Association in which a particular object participates.

### Diagrams

[Link Actions](#)

### Generalizations

[Action](#)

### Association Ends

- association : [Association](#) [1..1] (opposite [A\\_association\\_clearAssociationAction::clearAssociationAction](#))  
The Association to be cleared.
- ♦ object : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A\\_object\\_clearAssociationAction::clearAssociationAction](#))  
The InputPin that gives the object whose participation in the Association is to be cleared.

### Constraints

- multiplicity  
The multiplicity of the object InputPin is 1..1.  
  
`inv: object.is(1,1)`
- same\_type  
The type of the InputPin must conform to the type of at least one of the memberEnds of the association.

```
inv: association.memberEnd->exists(self.object.type.conformsTo(type))
```

## ClearStructuralFeatureAction [Class]

### Description

A ClearStructuralFeatureAction is a StructuralFeatureAction that removes all values of a StructuralFeature.

### Diagrams

[Structural Feature Actions](#)

### Generalizations

[StructuralFeatureAction](#)

## Association Ends

- ♦ result : [OutputPin](#) [0..1]{subsets [Action::output](#)} (opposite [A\\_result\\_clearStructuralFeatureAction::clearStructuralFeatureAction](#))  
The OutputPin on which is put the input object as modified by the ClearStructuralFeatureAction.

## Constraints

- type\_of\_result  
The type of the result OutputPin is the same as the type of the inherited object InputPin.

```
inv: result<>null implies result.type = object.type
```

- multiplicity\_of\_result  
The multiplicity of the result OutputPin must be 1..1.

```
inv: result<>null implies result.is(1,1)
```

## ClearVariableAction [Class]

### Description

A ClearVariableAction is a VariableAction that removes all values of a Variable.

### Diagrams

[Variable Actions](#)

### Generalizations

[VariableAction](#)

## ConditionalNode [Class]

### Description

A ConditionalNode is a StructuredActivityNode that chooses one among some number of alternative collections of ExecutableNodes to execute.

### Diagrams

[Structured Actions](#)

### Generalizations

[StructuredActivityNode](#)

## Attributes

- `isAssured` : [Boolean](#) [1..1] = false  
If true, the modeler asserts that the test for at least one Clause of the ConditionalNode will succeed.
- `isDeterminate` : [Boolean](#) [1..1] = false  
If true, the modeler asserts that the test for at most one Clause of the ConditionalNode will succeed.

## Association Ends

- `clause` : [Clause](#) [1..\*]{subsets [Element::ownedElement](#)} (opposite [A\\_clause\\_conditionalNode::conditionalNode](#))  
The set of Clauses composing the ConditionalNode.
- `result` : [OutputPin](#) [0..\*]{ordered, redefines [StructuredActivityNode::structuredNodeOutput](#)} (opposite [A\\_result\\_conditionalNode::conditionalNode](#))  
The OutputPins that onto which are moved values from the bodyOutputs of the Clause selected for execution.

## Operations

- `allActions()` : [Action](#) [0..\*]  
Return only this ConditionalNode. This prevents Actions within the ConditionalNode from having their OutputPins used as bodyOutputs or decider Pins in containing LoopNodes or ConditionalNodes.

```
body: self->asSet()
```

## Constraints

- `result_no_incoming`  
The result OutputPins have no incoming edges.

```
inv: result.incoming->isEmpty()
```

- `no_input_pins`  
A ConditionalNode has no InputPins.

```
inv: input->isEmpty()
```

- `one_clause_with_executable_node`  
No ExecutableNode in the ConditionNode may appear in the test or body part of more than one clause of a ConditionalNode.

```
inv: node->select(oclIsKindOf(ExecutableNode)).oclAsType(ExecutableNode)->forall(n | self.clause->select(test->union(_'body')->includes(n))->size()==1)
```

- `matching_output_pins`  
Each clause of a ConditionalNode must have the same number of bodyOutput pins as the ConditionalNode has result OutputPins, and each clause bodyOutput Pin must be compatible with the corresponding result OutputPin (by positional order) in type, multiplicity, ordering, and uniqueness.

```

inv: clause->forall(
  bodyOutput->size()=self.result->size() and
  Sequence{1..self.result->size()}->forall(i |
    bodyOutput->at(i).type.conformsTo(result->at(i).type) and
    bodyOutput->at(i).isOrdered = result->at(i).isOrdered and
    bodyOutput->at(i).isUnique = result->at(i).isUnique and
    bodyOutput->at(i).compatibleWith(result->at(i)))

```

- **executable\_nodes**  
The union of the ExecutableNodes in the test and body parts of all clauses must be the same as the subset of nodes contained in the ConditionalNode (considered as a StructuredActivityNode) that are ExecutableNodes.

```

inv: clause.test->union(clause._'body') = node-
>select (oclIsKindOf (ExecutableNode)) .oclAsType (ExecutableNode)

```

- **clause\_no\_predecessor**  
No two clauses within a ConditionalNode may be predecessorClauses of each other, either directly or indirectly.

```

inv: clause->closure(predecessorClause)->intersection(clause)->isEmpty()

```

## CreateLinkAction [Class]

### Description

A CreateLinkAction is a WriteLinkAction for creating links.

### Diagrams

[Link Actions](#), [Link Object Actions](#)

### Generalizations

[WriteLinkAction](#)

### Specializations

[CreateLinkObjectAction](#)

### Association Ends

- ♦ endData : [LinkEndCreationData](#) [2..\*]{redefines [LinkAction::endData](#)} (opposite [A\\_endData\\_createLinkAction::createLinkAction](#))  
The LinkEndData that specifies the values to be placed on the Association ends for the new link.

### Constraints

- **association\_not\_abstract**  
The Association cannot be an abstract Classifier.

```

inv: not self.association().isAbstract

```



## CreateLinkObjectAction [Class]

### Description

A CreateLinkObjectAction is a CreateLinkAction for creating link objects (AssociationClass instances).

### Diagrams

[Link Object Actions](#)

### Generalizations

[CreateLinkAction](#)

### Association Ends

- **result** : [OutputPin](#) [1..1]{subsets [Action::output](#)} (opposite [A\\_result\\_createLinkObjectAction::createLinkObjectAction](#))  
The output pin on which the newly created link object is placed.

### Constraints

- **multiplicity**  
The multiplicity of the OutputPin is 1..1.
- **type\_of\_result**  
The type of the result OutputPin must be the same as the Association of the CreateLinkObjectAction.

```
inv: result.is(1,1)
```

```
inv: result.type = association()
```

- **association\_class**  
The Association must be an AssociationClass.

```
inv: self.association().oclIsKindOf(AssociationClass)
```

## CreateObjectAction [Class]

### Description

A CreateObjectAction is an Action that creates an instance of the specified Classifier.

### Diagrams

[Object Actions](#)

### Generalizations

[Action](#)

## Association Ends

- classifier : [Classifier](#) [1..1] (opposite [A\\_classifier\\_createObjectAction::createObjectAction](#))  
The Classifier to be instantiated.
- ♦ result : [OutputPin](#) [1..1]{subsets [Action::output](#)} (opposite [A\\_result\\_createObjectAction::createObjectAction](#))  
The OutputPin on which the newly created object is placed.

## Constraints

- classifier\_not\_abstract  
The classifier cannot be abstract.  
  
`inv: not classifier.isAbstract`
- multiplicity  
The multiplicity of the result OutputPin is 1..1.  
  
`inv: result.is(1,1)`
- classifier\_not\_association\_class  
The classifier cannot be an AssociationClass.  
  
`inv: not classifier.oclIsKindOf(AssociationClass)`
- same\_type  
The type of the result OutputPin must be the same as the classifier of the CreateObjectAction.  
  
`inv: result.type = classifier`

## DestroyLinkAction [Class]

### Description

A DestroyLinkAction is a WriteLinkAction that destroys links (including link objects).

### Diagrams

[Link Actions](#)

### Generalizations

[WriteLinkAction](#)

## Association Ends

- ♦ endData : [LinkEndDestructionData](#) [2..\*]{redefines [LinkAction::endData](#)} (opposite [A\\_endData\\_destroyLinkAction::destroyLinkAction](#))  
The LinkEndData that the values of the Association ends for the links to be destroyed.

## DestroyObjectAction [Class]

### Description

A DestroyObjectAction is an Action that destroys objects.

### Diagrams

[Object Actions](#)

### Generalizations

[Action](#)

### Attributes

- isDestroyLinks : [Boolean](#) [1..1] = false  
Specifies whether links in which the object participates are destroyed along with the object.
- isDestroyOwnedObjects : [Boolean](#) [1..1] = false  
Specifies whether objects owned by the object (via composition) are destroyed along with the object.

### Association Ends

- ♦ target : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A\\_target\\_destroyObjectAction::destroyObjectAction](#))  
The InputPin providing the object to be destroyed.

### Constraints

- multiplicity  
The multiplicity of the target InputPin is 1..1.

```
inv: target.is(1,1)
```

- no\_type  
The target InputPin has no type.

```
inv: target.type= null
```

## ExpansionKind [Enumeration]

### Description

ExpansionKind is an enumeration type used to specify how an ExpansionRegion executes its contents.

### Diagrams

- [Expansion Regions](#)

## Literals

- parallel  
The content of the ExpansionRegion is executed concurrently for the elements of the input collections.
- iterative  
The content of the ExpansionRegion is executed iteratively for the elements of the input collections, in the order of the input elements, if the collections are ordered.
- stream  
A stream of input collection elements flows into a single execution of the content of the ExpansionRegion, in the order of the collection elements if the input collections are ordered.

## ExpansionNode [Class]

### Description

An ExpansionNode is an ObjectNode used to indicate a collection input or output for an ExpansionRegion. A collection input of an ExpansionRegion contains a collection that is broken into its individual elements inside the region, whose content is executed once per element. A collection output of an ExpansionRegion combines individual elements produced by the execution of the region into a collection for use outside the region.

### Diagrams

[Expansion Regions](#)

### Generalizations

[ObjectNode](#)

### Association Ends

- regionAsInput : [ExpansionRegion](#) [0..1] (opposite [ExpansionRegion::inputElement](#))  
The ExpansionRegion for which the ExpansionNode is an input.
- regionAsOutput : [ExpansionRegion](#) [0..1] (opposite [ExpansionRegion::outputElement](#))  
The ExpansionRegion for which the ExpansionNode is an output.

### Constraints

- region\_as\_input\_or\_output  
One of regionAsInput or regionAsOutput must be non-empty, but not both.

```
inv: regionAsInput->notEmpty() xor regionAsOutput->notEmpty()
```

## ExpansionRegion [Class]

### Description

An ExpansionRegion is a StructuredActivityNode that executes its content multiple times corresponding to elements of input collection(s).

### Diagrams

[Expansion Regions](#)

### Generalizations

[StructuredActivityNode](#)

### Attributes

- mode : [ExpansionKind](#) [1..1] = iterative  
The mode in which the ExpansionRegion executes its contents. If parallel, executions are concurrent. If iterative, executions are sequential. If stream, a stream of values flows into a single execution.

### Association Ends

- inputElement : [ExpansionNode](#) [1..\*] (opposite [ExpansionNode::regionAsInput](#))  
The ExpansionNodes that hold the input collections for the ExpansionRegion.
- outputElement : [ExpansionNode](#) [0..\*] (opposite [ExpansionNode::regionAsOutput](#))  
The ExpansionNodes that form the output collections of the ExpansionRegion.

## InputPin [Class]

### Description

An InputPin is a Pin that holds input values to be consumed by an Action.

### Diagrams

[Actions](#), [Invocation Actions](#), [Link End Data](#), [Link Actions](#), [Link Object Actions](#), [Structural Feature Actions](#), [Accept Event Actions](#), [Other Actions](#), [Variable Actions](#), [Structured Actions](#), [Object Actions](#)

### Generalizations

[Pin](#)

### Specializations

[ActionInputPin](#), [ValuePin](#)

## Constraints

- `outgoing_edges_structured_only`  
An InputPin may have outgoing ActivityEdges only when it is owned by a StructuredActivityNode, and these edges must target a node contained (directly or indirectly) in the owning StructuredActivityNode.

```
inv: outgoing->notEmpty() implies
    action<>null and
    action.oclIsKindOf(StructuredActivityNode) and
    action.oclAsType(StructuredActivityNode).allowedNodes()->includesAll(outgoing.target)
```

## InvocationAction [Abstract Class]

### Description

InvocationAction is an abstract class for the various actions that request Behavior invocation.

### Diagrams

[Invocation Actions](#)

### Generalizations

[Action](#)

### Specializations

[BroadcastSignalAction](#), [CallAction](#), [SendObjectAction](#), [SendSignalAction](#)

### Association Ends

- ♦ argument : [InputPin](#) [0..\*]{ordered, subsets [Action::input](#)} (opposite [A\\_argument invocationAction::invocationAction](#))  
The InputPins that provide the argument values passed in the invocation request.
- onPort : [Port](#) [0..1] (opposite [A\\_onPort invocationAction::invocationAction](#))  
For CallOperationActions, SendSignalActions, and SendObjectActions, an optional Port of the target object through which the invocation request is sent.

## LinkAction [Abstract Class]

### Description

LinkAction is an abstract class for all Actions that identify the links to be acted on using LinkEndData.

### Diagrams

[Link Actions](#)

### Generalizations

[Action](#)

## Specializations

[WriteLinkAction](#), [ReadLinkAction](#)

## Association Ends

- ◆ endData : [LinkEndData](#) [2..\*]{subsets [Element::ownedElement](#)} (opposite [A\\_endData linkAction::linkAction](#))  
The LinkEndData identifying the values on the ends of the links acting on by this LinkAction.
- ◆ inputValue : [InputPin](#) [1..\*]{subsets [Action::input](#)} (opposite [A\\_inputValue linkAction::linkAction](#))  
InputPins used by the LinkEndData of the LinkAction.

## Operations

- association() : [Association](#)  
Returns the Association acted on by this LinkAction.

```
body: endData->asSequence()->first().end.association
```

## Constraints

- same\_pins  
The inputValue InputPins is the same as the union of all the InputPins referenced by the endData.

```
inv: inputValue->asBag()=endData.allPins()
```

- same\_association  
The ends of the endData must all be from the same Association and include all and only the memberEnds of that association.

```
inv: endData.end = self.association().memberEnd->asBag()
```

- not\_static  
The ends of the endData must not be static.

```
inv: endData->forAll(not end.isStatic)
```

## LinkEndCreationData [Class]

### Description

LinkEndCreationData is LinkEndData used to provide values for one end of a link to be created by a CreateLinkAction.

### Diagrams

[Link End Data](#), [Link Actions](#)

### Generalizations

[LinkEndData](#)

## Attributes

- `isReplaceAll` : [Boolean](#) [1..1] = false  
Specifies whether the existing links emanating from the object on this end should be destroyed before creating a new link.

## Association Ends

- `insertAt` : [InputPin](#) [0..1] (opposite [A insertAt linkEndCreationData::linkEndCreationData](#))  
For ordered Association ends, the `InputPin` that provides the position where the new link should be inserted or where an existing link should be moved to. The type of the `insertAt` `InputPin` is `UnlimitedNatural`, but the input cannot be zero. It is omitted for Association ends that are not ordered.

## Operations

- `allPins()` : [InputPin](#) [0..\*]  
Adds the `insertAt` `InputPin` (if any) to the set of all Pins.

```
body: self.LinkEndData::allPins()->including(insertAt)
```

## Constraints

- `insertAt_pin`  
`LinkEndCreationData` for ordered Association ends must have a single `insertAt` `InputPin` for the insertion point with type `UnlimitedNatural` and multiplicity of 1..1, if `isReplaceAll=false`, and must have no `InputPin` for the insertion point when the association ends are unordered.

```
inv: if not end.isOrdered
then insertAt = null
else
  not isReplaceAll=false implies
    insertAt <> null and insertAt->forall(type=UnlimitedNatural and is(1,1))
endif
```

## LinkEndData [Class]

### Description

`LinkEndData` is an `Element` that identifies one end of a link to be read or written by a `LinkAction`. As a link (that is not a link object) cannot be passed as a runtime value to or from an `Action`, it is instead identified by its end objects and qualifier values, if any. A `LinkEndData` instance provides these values for a single Association end.

### Diagrams

[Link End Data](#), [Link Actions](#)

### Generalizations

[Element](#)



## Specializations

[LinkEndCreationData](#), [LinkEndDestructionData](#)

## Association Ends

- end : [Property](#) [1..1] (opposite [A\\_end\\_linkEndData::linkEndData](#))  
The Association end for which this LinkEndData specifies values.
- ♦ qualifier : [QualifierValue](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [A\\_qualifier\\_linkEndData::linkEndData](#))  
A set of QualifierValues used to provide values for the qualifiers of the end.
- value : [InputPin](#) [0..1] (opposite [A\\_value\\_linkEndData::linkEndData](#))  
The InputPin that provides the specified value for the given end. This InputPin is omitted if the LinkEndData specifies the "open" end for a ReadLinkAction.

## Operations

- allPins() : [InputPin](#) [0..\*]  
Returns all the InputPins referenced by this LinkEndData. By default this includes the value and qualifier InputPins, but subclasses may override the operation to add other InputPins.

```
body: value->asBag()->union(qualifier.value)
```

## Constraints

- same\_type  
The type of the value InputPin conforms to the type of the Association end.

```
inv: value<>null implies value.type.conformsTo(end.type)
```

- multiplicity  
The multiplicity of the value InputPin must be 1..1.

```
inv: value<>null implies value.is(1,1)
```

- end\_object\_input\_pin  
The value InputPin is not also the qualifier value InputPin.

```
inv: value->excludesAll(qualifier.value)
```

- property\_is\_association\_end  
The Property must be an Association memberEnd.

```
inv: end.association <> null
```

- qualifiers  
The qualifiers must be qualifiers of the Association end.

```
inv: end.qualifier->includesAll(qualifier.qualifier)
```

## LinkEndDestructionData [Class]

### Description

LinkEndDestructionData is LinkEndData used to provide values for one end of a link to be destroyed by a DestroyLinkAction.

### Diagrams

[Link End Data](#), [Link Actions](#)

### Generalizations

[LinkEndData](#)

### Attributes

- isDestroyDuplicates : [Boolean](#) [1..1] = false  
Specifies whether to destroy duplicates of the value in nonunique Association ends.

### Association Ends

- destroyAt : [InputPin](#) [0..1] (opposite [A\\_destroyAt\\_linkEndDestructionData::linkEndDestructionData](#))  
The InputPin that provides the position of an existing link to be destroyed in an ordered, nonunique Association end. The type of the destroyAt InputPin is UnlimitedNatural, but the value cannot be zero or unlimited.

### Operations

- allPins() : [InputPin](#) [0..\*]  
Adds the destroyAt InputPin (if any) to the set of all Pins.

```
body: self.LinkEndData::allPins()->including(destroyAt)
```

### Constraints

- destroyAt\_pin  
LinkEndDestructionData for ordered, nonunique Association ends must have a single destroyAt InputPin if isDestroyDuplicates is false, which must be of type UnlimitedNatural and have a multiplicity of 1..1. Otherwise, the action has no destroyAt input pin.

```
inv: if not end.isOrdered or end.isUnique or isDestroyDuplicates  
then destroyAt = null  
else  
  destroyAt <> null and  
  destroyAt->forAll(type=UnlimitedNatural and is(1,1))  
endif
```

## LoopNode [Class]

### Description

A LoopNode is a StructuredActivityNode that represents an iterative loop with setup, test, and body sections.

### Diagrams

[Structured Actions](#)

### Generalizations

[StructuredActivityNode](#)

### Attributes

- isTestedFirst : [Boolean](#) [1..1] = false  
If true, the test is performed before the first execution of the bodyPart. If false, the bodyPart is executed once before the test is performed.

### Association Ends

- bodyOutput : [OutputPin](#) [0..\*]{ordered} (opposite [A\\_bodyOutput\\_loopNode::loopNode](#))  
The OutputPins on Actions within the bodyPart, the values of which are moved to the loopVariable OutputPins after the completion of each execution of the bodyPart, before the next iteration of the loop begins or before the loop exits.
- bodyPart : [ExecutableNode](#) [0..\*] (opposite [A\\_bodyPart\\_loopNode::loopNode](#))  
The set of ExecutableNodes that perform the repetitive computations of the loop. The bodyPart is executed as long as the test section produces a true value.
- decider : [OutputPin](#) [1..1] (opposite [A\\_decider\\_loopNode::loopNode](#))  
An OutputPin on an Action in the test section whose Boolean value determines whether to continue executing the loop bodyPart.
- ♦ loopVariable : [OutputPin](#) [0..\*]{ordered, subsets [Element::ownedElement](#)} (opposite [A\\_loopVariable\\_loopNode::loopNode](#))  
A list of OutputPins that hold the values of the loop variables during an execution of the loop. When the test fails, the values are moved to the result OutputPins of the loop.
- ♦ loopVariableInput : [InputPin](#) [0..\*]{ordered, redefines [StructuredActivityNode::structuredNodeInput](#)} (opposite [A\\_loopVariableInput\\_loopNode::loopNode](#))  
A list of InputPins whose values are moved into the loopVariable Pins before the first iteration of the loop.
- ♦ result : [OutputPin](#) [0..\*]{ordered, redefines [StructuredActivityNode::structuredNodeOutput](#)} (opposite [A\\_result\\_loopNode::loopNode](#))  
A list of OutputPins that receive the loopVariable values after the last iteration of the loop and constitute the output of the LoopNode.
- setupPart : [ExecutableNode](#) [0..\*] (opposite [A\\_setupPart\\_loopNode::loopNode](#))  
The set of ExecutableNodes executed before the first iteration of the loop, in order to initialize values or perform other

setup computations.

- test : [ExecutableNode](#) [1..\*] (opposite [A\\_test\\_loopNode::loopNode](#))  
The set of ExecutableNodes executed in order to provide the test result for the loop.

## Operations

- allActions() : [Action](#) [0..\*]  
Return only this LoopNode. This prevents Actions within the LoopNode from having their OutputPins used as bodyOutputs or decider Pins in containing LoopNodes or ConditionalNodes.

```
body: self->asSet()
```

- sourceNodes() : [ActivityNode](#) [0..\*]  
Return the loopVariable OutputPins in addition to other source nodes for the LoopNode as a StructuredActivityNode.

```
body: self.StructuredActivityNode::sourceNodes()->union(loopVariable)
```

## Constraints

- result\_no\_incoming  
The result OutputPins have no incoming edges.

```
inv: result.incoming->isEmpty()
```

- input\_edges  
The loopVariableInputs must not have outgoing edges.

```
inv: loopVariableInput.outgoing->isEmpty()
```

- executable\_nodes  
The union of the ExecutableNodes in the setupPart, test and bodyPart of a LoopNode must be the same as the subset of nodes contained in the LoopNode (considered as a StructuredActivityNode) that are ExecutableNodes.

```
inv: setupPart->union(test)->union(bodyPart)=node->select(oclIsKindOf(ExecutableNode)).oclAsType(ExecutableNode)->asSet()
```

- body\_output\_pins  
The bodyOutput pins are OutputPins on Actions in the body of the LoopNode.

```
inv: bodyPart.oclAsType(Action).allActions().output->includesAll(bodyOutput)
```

- setup\_test\_and\_body  
The test and body parts of a ConditionalNode must be disjoint with each other.

```
inv: setupPart->intersection(test)->isEmpty() and  
setupPart->intersection(bodyPart)->isEmpty() and  
test->intersection(bodyPart)->isEmpty()
```

- **matching\_output\_pins**  
A LoopNode must have the same number of bodyOutput Pins as loopVariables, and each bodyOutput Pin must be compatible with the corresponding loopVariable (by positional order) in type, multiplicity, ordering and uniqueness.

```
inv: bodyOutput->size()=loopVariable->size() and
Sequence{1..loopVariable->size()}->forall(i |
  bodyOutput->at(i).type.conformsTo(loopVariable->at(i).type) and
  bodyOutput->at(i).isOrdered = loopVariable->at(i).isOrdered and
  bodyOutput->at(i).isUnique = loopVariable->at(i).isUnique and
  loopVariable->at(i).includesMultiplicity(bodyOutput->at(i)))
```

- **matching\_loop\_variables**  
A LoopNode must have the same number of loopVariableInputs and loopVariables, and they must match in type, uniqueness and multiplicity.

```
inv: loopVariableInput->size()=loopVariable->size() and
loopVariableInput.type=loopVariable.type and
loopVariableInput.isUnique=loopVariable.isUnique and
loopVariableInput.lower=loopVariable.lower and
loopVariableInput.upper=loopVariable.upper
```

- **matching\_result\_pins**  
A LoopNode must have the same number of result OutputPins and loopVariables, and they must match in type, uniqueness and multiplicity.

```
inv: result->size()=loopVariable->size() and
result.type=loopVariable.type and
result.isUnique=loopVariable.isUnique and
result.lower=loopVariable.lower and
result.upper=loopVariable.upper
```

- **loop\_variable\_outgoing**  
All ActivityEdges outgoing from loopVariable OutputPins must have targets within the LoopNode.

```
inv: allOwnedNodes()->includesAll(loopVariable.outgoing.target)
```

## OpaqueAction [Class]

### Description

An OpaqueAction is an Action whose functionality is not specified within UML.

### Diagrams

[Actions](#)

### Generalizations

[Action](#)

## Attributes

- body : [String](#) [0..\*]  
Provides a textual specification of the functionality of the Action, in one or more languages other than UML.
- language : [String](#) [0..\*]  
If provided, a specification of the language used for each of the body Strings.

## Association Ends

- ♦ inputValue : [InputPin](#) [0..\*]{subsets [Action::input](#)} (opposite [A\\_inputValue\\_opaqueAction::opaqueAction](#))  
The InputPins providing inputs to the OpaqueAction.
- ♦ outputValue : [OutputPin](#) [0..\*]{subsets [Action::output](#)} (opposite [A\\_outputValue\\_opaqueAction::opaqueAction](#))  
The OutputPins on which the OpaqueAction provides outputs.

## Constraints

- language\_body\_size  
If the language attribute is not empty, then the size of the body and language lists must be the same.

```
inv: language->notEmpty() implies (_'body'->size() = language->size())
```

## OutputPin [Class]

### Description

An OutputPin is a Pin that holds output values produced by an Action.

### Diagrams

[Actions](#), [Invocation Actions](#), [Link Actions](#), [Link Object Actions](#), [Structural Feature Actions](#), [Accept Event Actions](#), [Other Actions](#), [Variable Actions](#), [Structured Actions](#), [Object Actions](#)

### Generalizations

[Pin](#)

### Constraints

- incoming\_edges\_structured\_only  
An OutputPin may have incoming ActivityEdges only when it is owned by a StructuredActivityNode, and these edges must have sources contained (directly or indirectly) in the owning StructuredActivityNode.

```
inv: incoming->notEmpty() implies  
  action<>null and  
  action.oclIsKindOf(StructuredActivityNode) and  
  action.oclAsType(StructuredActivityNode).allowedNodes()->includesAll(incoming.source)
```

## Pin [Abstract Class]

### Description

A Pin is an ObjectNode and MultiplicityElement that provides input values to an Action or accepts output values from an Action.

### Diagrams

[Actions](#)

### Generalizations

[ObjectNode](#), [MultiplicityElement](#)

### Specializations

[InputPin](#), [OutputPin](#)

### Attributes

- isControl : [Boolean](#) [1..1] = false  
Indicates whether the Pin provides data to the Action or just controls how the Action executes.

### Constraints

- control\_pins  
A control Pin has a control type.  
  
`inv: isControl implies isControlType`
- not\_unique  
Pin multiplicity is not unique.  
  
`inv: not isUnique`

## QualifierValue [Class]

### Description

A QualifierValue is an Element that is used as part of LinkEndData to provide the value for a single qualifier of the end given by the LinkEndData.

### Diagrams

[Link End Data](#)

### Generalizations

[Element](#)

## Association Ends

- `qualifier` : [Property](#) [1..1] (opposite [A qualifier qualifierValue::qualifierValue](#))  
The qualifier Property for which the value is to be specified.
- `value` : [InputPin](#) [1..1] (opposite [A value qualifierValue::qualifierValue](#))  
The InputPin from which the specified value for the qualifier is taken.

## Constraints

- `multiplicity_of_qualifier`  
The multiplicity of the value InputPin is 1..1.  

```
inv: value.is(1,1)
```
- `type_of_qualifier`  
The type of the value InputPin conforms to the type of the qualifier Property.  

```
inv: value.type.conformsTo(qualifier.type)
```
- `qualifier_attribute`  
The qualifier must be a qualifier of the Association end of the linkEndData that owns this QualifierValue.  

```
inv: linkEndData.end.qualifier->includes(qualifier)
```

## RaiseExceptionAction [Class]

### Description

A RaiseExceptionAction is an Action that causes an exception to occur. The input value becomes the exception object.

### Diagrams

[Other Actions](#)

### Generalizations

[Action](#)

### Association Ends

- ♦ `exception` : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A exception raiseExceptionAction::raiseExceptionAction](#))  
An InputPin whose value becomes the exception object.



## ReadExtentAction [Class]

### Description

A ReadExtentAction is an Action that retrieves the current instances of a Classifier.

### Diagrams

[Object Actions](#)

### Generalizations

[Action](#)

### Association Ends

- classifier : [Classifier](#) [1..1] (opposite [A\\_classifier\\_readExtentAction::readExtentAction](#))  
The Classifier whose instances are to be retrieved.
- ♦ result : [OutputPin](#) [1..1]{subsets [Action::output](#)} (opposite [A\\_result\\_readExtentAction::readExtentAction](#))  
The OutputPin on which the Classifier instances are placed.

### Constraints

- type\_is\_classifier  
The type of the result OutputPin is the classifier.

```
inv: result.type = classifier
```

- multiplicity\_of\_result  
The multiplicity of the result OutputPin is 0..\*.

```
inv: result.is(0,*)
```

## ReadIsClassifiedObjectAction [Class]

### Description

A ReadIsClassifiedObjectAction is an Action that determines whether an object is classified by a given Classifier.

### Diagrams

[Object Actions](#)

### Generalizations

[Action](#)

## Attributes

- isDirect : [Boolean](#) [1..1] = false  
Indicates whether the input object must be directly classified by the given Classifier or whether it may also be an instance of a specialization of the given Classifier.

## Association Ends

- classifier : [Classifier](#) [1..1] (opposite [A\\_classifier\\_readIsClassifiedObjectAction::readIsClassifiedObjectAction](#))  
The Classifier against which the classification of the input object is tested.
- ♦ object : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A\\_object\\_readIsClassifiedObjectAction::readIsClassifiedObjectAction](#))  
The InputPin that holds the object whose classification is to be tested.
- ♦ result : [OutputPin](#) [1..1]{subsets [Action::output](#)} (opposite [A\\_result\\_readIsClassifiedObjectAction::readIsClassifiedObjectAction](#))  
The OutputPin that holds the Boolean result of the test.

## Constraints

- no\_type  
The object InputPin has no type.  

```
inv: object.type = null
```
- multiplicity\_of\_output  
The multiplicity of the result OutputPin is 1..1.  

```
inv: result.is(1,1)
```
- boolean\_result  
The type of the result OutputPin is Boolean.  

```
inv: result.type = Boolean
```
- multiplicity\_of\_input  
The multiplicity of the object InputPin is 1..1.  

```
inv: object.is(1,1)
```

## ReadLinkAction [Class]

### Description

A ReadLinkAction is a LinkAction that navigates across an Association to retrieve the objects on one end.

## Diagrams

[Link Actions](#)

## Generalizations

[LinkAction](#)

## Association Ends

- ♦ result : [OutputPin](#) [1..1]{subsets [Action::output](#)} (opposite [A\\_result\\_readLinkAction::readLinkAction](#))  
The OutputPin on which the objects retrieved from the "open" end of those links whose values on other ends are given by the endData.

## Operations

- openEnd() : [Property](#) [0..\*]  
Returns the ends corresponding to endData with no value InputPin. (A well-formed ReadLinkAction is constrained to have only one of these.)

```
body: endData->select(value=null).end->asOrderedSet()
```

## Constraints

- type\_and\_ordering  
The type and ordering of the result OutputPin are same as the type and ordering of the open Association end.
- compatible\_multiplicity  
The multiplicity of the open Association end must be compatible with the multiplicity of the result OutputPin.

```
inv: self.openEnd()->forall(type=result.type and isOrdered=result.isOrdered)
```

```
inv: self.openEnd()->first().compatibleWith(result)
```

- visibility  
Visibility of the open end must allow access from the object performing the action.

```
inv: let openEnd : Property = self.openEnd()->first() in
  openEnd.visibility = VisibilityKind::public or
  endData->exists(oed |
    oed.end<>openEnd and
    ('context' = oed.end.type or
     (openEnd.visibility = VisibilityKind::protected and
      '_context'.conformsTo(oed.end.type.oclAsType(Classifier))))))
```

- one\_open\_end  
Exactly one linkEndData specification (corresponding to the "open" end) must not have an value InputPin.

```
inv: self.openEnd()->size() = 1
```

- `navigable_open_end`  
The open end must be navigable.

```
inv: self.openEnd()->first().isNavigable()
```

## ReadLinkObjectEndAction [Class]

### Description

A ReadLinkObjectEndAction is an Action that retrieves an end object from a link object.

### Diagrams

[Link Object Actions](#)

### Generalizations

[Action](#)

### Association Ends

- `end` : [Property](#) [1..1] (opposite [A\\_end\\_readLinkObjectEndAction::readLinkObjectEndAction](#))  
The Association end to be read.
- `object` : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A\\_object\\_readLinkObjectEndAction::readLinkObjectEndAction](#))  
The input pin from which the link object is obtained.
- `result` : [OutputPin](#) [1..1]{subsets [Action::output](#)} (opposite [A\\_result\\_readLinkObjectEndAction::readLinkObjectEndAction](#))  
The OutputPin where the result value is placed.

### Constraints

- `property`  
The end Property must be an Association memberEnd.

```
inv: end.association <> null
```

- `multiplicity_of_object`  
The multiplicity of the object InputPin is 1..1.

```
inv: object.is(1,1)
```

- `ends_of_association`  
The ends of the association must not be static.

```
inv: end.association.memberEnd->forall(e | not e.isStatic)
```

- **type\_of\_result**  
The type of the result OutputPin is the same as the type of the end Property.

```
inv: result.type = end.type
```

- **multiplicity\_of\_result**  
The multiplicity of the result OutputPin is 1..1.

```
inv: result.is(1,1)
```

- **type\_of\_object**  
The type of the object InputPin is the AssociationClass that owns the end Property.

```
inv: object.type = end.association
```

- **association\_of\_association**  
The association of the end must be an AssociationClass.

```
inv: end.association.oclIsKindOf(AssociationClass)
```

## ReadLinkObjectEndQualifierAction [Class]

### Description

A ReadLinkObjectEndQualifierAction is an Action that retrieves a qualifier end value from a link object.

### Diagrams

[Link Object Actions](#)

### Generalizations

[Action](#)

### Association Ends

- ♦ object : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A\\_object\\_readLinkObjectEndQualifierAction::readLinkObjectEndQualifierAction](#))  
The InputPin from which the link object is obtained.
- qualifier : [Property](#) [1..1] (opposite [A\\_qualifier\\_readLinkObjectEndQualifierAction::readLinkObjectEndQualifierAction](#))  
The qualifier Property to be read.
- ♦ result : [OutputPin](#) [1..1]{subsets [Action::output](#)} (opposite [A\\_result\\_readLinkObjectEndQualifierAction::readLinkObjectEndQualifierAction](#))  
The OutputPin where the result value is placed.

## Constraints

- **multiplicity\_of\_object**  
The multiplicity of the object InputPin is 1..1.  
  
`inv: object.is(1,1)`
- **type\_of\_object**  
The type of the object InputPin is the AssociationClass that owns the Association end that has the given qualifier Property.  
  
`inv: object.type = qualifier.associationEnd.association`
- **multiplicity\_of\_qualifier**  
The multiplicity of the qualifier Property is 1..1.  
  
`inv: qualifier.is(1,1)`
- **ends\_of\_association**  
The ends of the Association must not be static.  
  
`inv: qualifier.associationEnd.association.memberEnd->forall(e | not e.isStatic)`
- **multiplicity\_of\_result**  
The multiplicity of the result OutputPin is 1..1.  
  
`inv: result.is(1,1)`
- **same\_type**  
The type of the result OutputPin is the same as the type of the qualifier Property.  
  
`inv: result.type = qualifier.type`
- **association\_of\_association**  
The association of the Association end of the qualifier Property must be an AssociationClass.  
  
`inv: qualifier.associationEnd.association.oclIsKindOf(AssociationClass)`
- **qualifier\_attribute**  
The qualifier Property must be a qualifier of an Association end.  
  
`inv: qualifier.associationEnd <> null`

## ReadSelfAction [Class]

### Description

A ReadSelfAction is an Action that retrieves the context object of the Behavior execution within which the ReadSelfAction execution is taking place.

## Diagrams

[Object Actions](#)

## Generalizations

[Action](#)

## Association Ends

- ♦ result : [OutputPin](#) [1..1]{subsets [Action::output](#)} (opposite [A\\_result\\_readSelfAction::readSelfAction](#))  
The OutputPin on which the context object is placed.

## Constraints

- contained  
A ReadSelfAction must have a context Classifier.

```
inv: _('context' <> null
```

- multiplicity  
The multiplicity of the result OutputPin is 1..1.

```
inv: result.is(1,1)
```

- not\_static  
If the ReadSelfAction is contained in an Behavior that is acting as a method, then the Operation of the method must not be static.

```
inv: let behavior: Behavior = self.containingBehavior() in  
behavior.specification<>null implies not behavior.specification.isStatic
```

- type  
The type of the result OutputPin is the context Classifier.

```
inv: result.type = _('context'
```

## ReadStructuralFeatureAction [Class]

### Description

A ReadStructuralFeatureAction is a StructuralFeatureAction that retrieves the values of a StructuralFeature.

### Diagrams

[Structural Feature Actions](#)

### Generalizations

[StructuralFeatureAction](#)

## Association Ends

- ♦ result : [OutputPin](#) [1..1]{subsets [Action::output](#)} (opposite [A\\_result\\_readStructuralFeatureAction::readStructuralFeatureAction](#))  
The OutputPin on which the result values are placed.

## Constraints

- multiplicity  
The multiplicity of the StructuralFeature must be compatible with the multiplicity of the result OutputPin.  
  
`inv: structuralFeature.compatibleWith(result)`
- type\_and\_ordering  
The type and ordering of the result OutputPin are the same as the type and ordering of the StructuralFeature.

```
inv: result.type =structuralFeature.type and  
result.isOrdered = structuralFeature.isOrdered
```

## ReadVariableAction [Class]

### Description

A ReadVariableAction is a VariableAction that retrieves the values of a Variable.

### Diagrams

[Variable Actions](#)

### Generalizations

[VariableAction](#)

## Association Ends

- ♦ result : [OutputPin](#) [1..1]{subsets [Action::output](#)} (opposite [A\\_result\\_readVariableAction::readVariableAction](#))  
The OutputPin on which the result values are placed.

## Constraints

- type\_and\_ordering  
The type and ordering of the result OutputPin are the same as the type and ordering of the variable.  
  
`inv: result.type =variable.type and  
result.isOrdered = variable.isOrdered`
- compatible\_multiplicity  
The multiplicity of the variable must be compatible with the multiplicity of the output pin.



```
inv: variable.compatibleWith(result)
```

## ReclassifyObjectAction [Class]

### Description

A `ReclassifyObjectAction` is an Action that changes the Classifiers that classify an object.

### Diagrams

[Object Actions](#)

### Generalizations

[Action](#)

### Attributes

- `isReplaceAll` : [Boolean](#) [1..1] = false  
Specifies whether existing Classifiers should be removed before adding the new Classifiers.

### Association Ends

- `newClassifier` : [Classifier](#) [0..\*] (opposite [A\\_newClassifier\\_reclassifyObjectAction::reclassifyObjectAction](#))  
A set of Classifiers to be added to the Classifiers of the given object.
- ♦ `object` : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A\\_object\\_reclassifyObjectAction::reclassifyObjectAction](#))  
The InputPin that holds the object to be reclassified.
- `oldClassifier` : [Classifier](#) [0..\*] (opposite [A\\_oldClassifier\\_reclassifyObjectAction::reclassifyObjectAction](#))  
A set of Classifiers to be removed from the Classifiers of the given object.

### Constraints

- `input_pin`  
The object InputPin has no type.  

```
inv: object.type = null
```
- `classifier_not_abstract`  
None of the newClassifiers may be abstract.  

```
inv: not newClassifier->exists(isAbstract)
```
- `multiplicity`  
The multiplicity of the object InputPin is 1..1.  

```
inv: object.is(1,1)
```

## ReduceAction [Class]

### Description

A ReduceAction is an Action that reduces a collection to a single value by repeatedly combining the elements of the collection using a reducer Behavior.

### Diagrams

[Other Actions](#)

### Generalizations

[Action](#)

### Attributes

- isOrdered : [Boolean](#) [1..1] = false  
Indicates whether the order of the input collection should determine the order in which the reducer Behavior is applied to its elements.

### Association Ends

- ♦ collection : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A\\_collection\\_reduceAction::reduceAction](#))  
The InputPin that provides the collection to be reduced.
- reducer : [Behavior](#) [1..1] (opposite [A\\_reducer\\_reduceAction::reduceAction](#))  
A Behavior that is repeatedly applied to two elements of the input collection to produce a value that is of the same type as elements of the collection.
- ♦ result : [OutputPin](#) [1..1]{subsets [Action::output](#)} (opposite [A\\_result\\_reduceAction::reduceAction](#))  
The output pin on which the result value is placed.

### Constraints

- reducer\_inputs\_output  
The reducer Behavior must have two input ownedParameters and one output ownedParameter, where the type of the output Parameter and the type of elements of the input collection conform to the types of the input Parameters.

```
inv: let inputs: OrderedSet(Parameter) = reducer.inputParameters() in
let outputs: OrderedSet(Parameter) = reducer.outputParameters() in
inputs->size()=2 and outputs->size()=1 and
inputs.type->forall(t |
  outputs.type->forall(conformsTo(t)) and
  -- Note that the following only checks the case when the collection is via multiple tokens.
  collection.upperBound()>1 implies collection.type.conformsTo(t))
```

- input\_type\_is\_collection  
The type of the collection InputPin must be a collection.

Cannot be expressed in OCL

- `output_types_are_compatible`  
The type of the output of the reducer Behavior must conform to the type of the result OutputPin.

```
inv: reducer.outputParameters().type->forAll(conformsTo(result.type))
```

## RemoveStructuralFeatureValueAction [Class]

### Description

A RemoveStructuralFeatureValueAction is a WriteStructuralFeatureAction that removes values from a StructuralFeature.

### Diagrams

[Structural Feature Actions](#)

### Generalizations

[WriteStructuralFeatureAction](#)

### Attributes

- `isRemoveDuplicates` : [Boolean](#) [1..1] = false  
Specifies whether to remove duplicates of the value in nonunique StructuralFeatures.

### Association Ends

- ♦ `removeAt` : [InputPin](#) [0..1]{subsets [Action::input](#)} (opposite [A\\_removeAt\\_removeStructuralFeatureValueAction::removeStructuralFeatureValueAction](#))  
An InputPin that provides the position of an existing value to remove in ordered, nonunique structural features. The type of the removeAt InputPin is UnlimitedNatural, but the value cannot be zero or unlimited.

### Constraints

- `removeAt_and_value`  
RemoveStructuralFeatureValueActions removing a value from ordered, non-unique StructuralFeatures must have a single removeAt InputPin and no value InputPin, if isRemoveDuplicates is false. The removeAt InputPin must be of type Unlimited Natural with multiplicity 1..1. Otherwise, the Action has a value InputPin and no removeAt InputPin.

```
inv: if structuralFeature.isOrdered and not structuralFeature.isUnique and not isRemoveDuplicates
then
  value = null and
  removeAt <> null and
  removeAt.type = UnlimitedNatural and
  removeAt.is(1,1)
else
  removeAt = null and value <> null
endif
```

## RemoveVariableValueAction [Class]

### Description

A RemoveVariableValueAction is a WriteVariableAction that removes values from a Variables.

### Diagrams

[Variable Actions](#)

### Generalizations

[WriteVariableAction](#)

### Attributes

- isRemoveDuplicates : [Boolean](#) [1..1] = false  
Specifies whether to remove duplicates of the value in nonunique Variables.

### Association Ends

- ♦ removeAt : [InputPin](#) [0..1]{subsets [Action::input](#)} (opposite [A\\_removeAt\\_removeVariableValueAction::removeVariableValueAction](#))  
An InputPin that provides the position of an existing value to remove in ordered, nonunique Variables. The type of the removeAt InputPin is UnlimitedNatural, but the value cannot be zero or unlimited.

### Constraints

- removeAt\_and\_value  
ReadVariableActions removing a value from ordered, non-unique Variables must have a single removeAt InputPin and no value InputPin, if isRemoveDuplicates is false. The removeAt InputPin must be of type Unlimited Natural with multiplicity 1..1. Otherwise, the Action has a value InputPin and no removeAt InputPin.

```
inv: if variable.isOrdered and not variable.isUnique and not isRemoveDuplicates then
    value = null and
    removeAt <> null and
    removeAt.type = UnlimitedNatural and
    removeAt.is(1,1)
else
    removeAt = null and value <> null
endif
```

## ReplyAction [Class]

### Description

A ReplyAction is an Action that accepts a set of reply values and a value containing return information produced by a previous AcceptCallAction. The ReplyAction returns the values to the caller of the previous call, completing execution of the call.

## Diagrams

[Accept Event Actions](#)

## Generalizations

[Action](#)

## Association Ends

- replyToCall : [Trigger](#) [1..1] (opposite [A\\_replyToCall\\_replyAction::replyAction](#))  
The Trigger specifying the Operation whose call is being replied to.
- ♦ replyValue : [InputPin](#) [0..\*]{ordered, subsets [Action::input](#)} (opposite [A\\_replyValue\\_replyAction::replyAction](#))  
A list of InputPins providing the values for the output (inout, out, and return) Parameters of the Operation. These values are returned to the caller.
- ♦ returnInformation : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A\\_returnInformation\\_replyAction::replyAction](#))  
An InputPin that holds the return information value produced by an earlier AcceptCallAction.

## Constraints

- pins\_match\_parameter  
The replyValue InputPins must match the output (return, out, and inout) parameters of the operation of the event of the replyToCall Trigger in number, type, ordering, and multiplicity.

```
inv: let parameter:OrderedSet(Parameter) =
replyToCall.event.oclAsType(CallEvent).operation.outputParameters() in
replyValue->size()=parameter->size() and
Sequence{1..replyValue->size()}->forall(i |
  replyValue->at(i).type.conformsTo(parameter->at(i).type) and
  replyValue->at(i).isOrdered=parameter->at(i).isOrdered and
  replyValue->at(i).compatibleWith(parameter->at(i)))
```

- event\_on\_reply\_to\_call\_trigger  
The event of the replyToCall Trigger must be a CallEvent.

```
inv: replyToCall.event.oclIsKindOf(CallEvent)
```

## SendObjectAction [Class]

### Description

A SendObjectAction is an InvocationAction that transmits an input object to the target object, which is handled as a request message by the target object. The requestor continues execution immediately after the object is sent out and cannot receive reply values.

## Diagrams

[Invocation Actions](#)

## Generalizations

[InvocationAction](#)

## Association Ends

- ♦ request : [InputPin](#) [1..1]{redefines [InvocationAction::argument](#)} (opposite [A\\_request\\_sendObjectAction::sendObjectAction](#))  
The request object, which is transmitted to the target object. The object may be copied in transmission, so identity might not be preserved.
- ♦ target : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A\\_target\\_sendObjectAction::sendObjectAction](#))  
The target object to which the object is sent.

## Constraints

- type\_target\_pin  
If onPort is not empty, the Port given by onPort must be an owned or inherited feature of the type of the target InputPin.

```
inv: onPort<>null implies target.type.oclAsType(Classifier).allFeatures()->includes(onPort)
```

## SendSignalAction [Class]

### Description

A SendSignalAction is an InvocationAction that creates a Signal instance and transmits it to the target object. Values from the argument InputPins are used to provide values for the attributes of the Signal. The requestor continues execution immediately after the Signal instance is sent out and cannot receive reply values.

### Diagrams

[Invocation Actions](#)

## Generalizations

[InvocationAction](#)

## Association Ends

- signal : [Signal](#) [1..1] (opposite [A\\_signal\\_sendSignalAction::sendSignalAction](#))  
The Signal whose instance is transmitted to the target.
- ♦ target : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A\\_target\\_sendSignalAction::sendSignalAction](#))  
The InputPin that provides the target object to which the Signal instance is sent.

## Constraints

- **type\_ordering\_multiplicity**  
The type, ordering, and multiplicity of an argument InputPin must be the same as the corresponding attribute of the signal.

```
inv: let attribute: OrderedSet(Property) = signal.allAttributes() in
Sequence{1..argument->size()}->forall(i |
  argument->at(i).type.conformsTo(attribute->at(i).type) and
  argument->at(i).isOrdered = attribute->at(i).isOrdered and
  argument->at(i).compatibleWith(attribute->at(i)))
```

- **number\_order**  
The number and order of argument InputPins must be the same as the number and order of attributes of the signal.

```
inv: argument->size()=signal.allAttributes()->size()
```

- **type\_target\_pin**  
If onPort is not empty, the Port given by onPort must be an owned or inherited feature of the type of the target InputPin.

```
inv: not onPort->isEmpty() implies target.type.oclAsType(Classifier).allFeatures()->includes(onPort)
```

## SequenceNode [Class]

### Description

A SequenceNode is a StructuredActivityNode that executes a sequence of ExecutableNodes in order.

### Diagrams

[Structured Actions](#)

### Generalizations

[StructuredActivityNode](#)

### Association Ends

- ♦ executableNode : [ExecutableNode](#) [0..\*]{ordered, redefines [StructuredActivityNode::node](#)} (opposite [A\\_executableNode\\_sequenceNode::sequenceNode](#))  
The ordered set of ExecutableNodes to be sequenced.

## StartClassifierBehaviorAction [Class]

### Description

A StartClassifierBehaviorAction is an Action that starts the classifierBehavior of the input object.

### Diagrams

[Object Actions](#)

## Generalizations

[Action](#)

## Association Ends

- ♦ object : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A object\\_startClassifierBehaviorAction::startClassifierBehaviorAction](#))  
The InputPin that holds the object whose classifierBehavior is to be started.

## Constraints

- multiplicity  
The multiplicity of the object InputPin is 1..1
- type\_has\_classifier  
If the InputPin has a type, then the type or one of its ancestors must have a classifierBehavior.

```
inv: object.is(1,1)
```

```
inv: object.type->notEmpty() implies  
    (object.type.oclIsKindOf(BehavioredClassifier) and  
    object.type.oclAsType(BehavioredClassifier).classifierBehavior<>null)
```

## StartObjectBehaviorAction [Class]

### Description

A StartObjectBehaviorAction is an InvocationAction that starts the execution either of a directly instantiated Behavior or of the classifierBehavior of an object. Argument values may be supplied for the input Parameters of the Behavior. If the Behavior is invoked synchronously, then output values may be obtained for output Parameters.

### Diagrams

[Invocation Actions](#)

## Generalizations

[CallAction](#)

## Association Ends

- ♦ object : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A object\\_startObjectBehaviorAction::startObjectBehaviorAction](#))  
An InputPin that holds the object that is either a Behavior to be started or has a classifierBehavior to be started.

## Operations

- outputParameters() : [Parameter](#) [0..\*]  
Return the inout, out and return ownedParameters of the Behavior being called.



```
body: self.behavior().outputParameters()
```

- `inputParameters() : Parameter [0..*]`  
Return the in and inout ownedParameters of the Behavior being called.

```
body: self.behavior().inputParameters()
```

- `behavior() : Behavior [0..1]`  
If the type of the object InputPin is a Behavior, then that Behavior. Otherwise, if the type of the object InputPin is a BehavioredClassifier, then the classifierBehavior of that BehavioredClassifier.

```
body: if object.type.ocIsKindOf(Behavior) then
  object.type.ocAsType(Behavior)
else if object.type.ocIsKindOf(BehavioredClassifier) then
  object.type.ocAsType(BehavioredClassifier).classifierBehavior
else
  null
endif
endif
```

## Constraints

- `multiplicity_of_object`  
The multiplicity of the object InputPin must be 1..1.

```
inv: object.is(1,1)
```

- `type_of_object`  
The type of the object InputPin must be either a Behavior or a BehavioredClassifier with a classifierBehavior.

```
inv: self.behavior() <> null
```

- `no_onport`  
A StartObjectBehaviorAction may not specify onPort.

```
inv: onPort->isEmpty()
```

## StructuralFeatureAction [Abstract Class]

### Description

StructuralFeatureAction is an abstract class for all Actions that operate on StructuralFeatures.

### Diagrams

[Structural Feature Actions](#)

### Generalizations

[Action](#)

## Specializations

[WriteStructuralFeatureAction](#), [ClearStructuralFeatureAction](#), [ReadStructuralFeatureAction](#)

## Association Ends

- ♦ object : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A\\_object\\_structuralFeatureAction::structuralFeatureAction](#))  
The InputPin from which the object whose StructuralFeature is to be read or written is obtained.
- structuralFeature : [StructuralFeature](#) [1..1] (opposite [A\\_structuralFeature\\_structuralFeatureAction::structuralFeatureAction](#))  
The StructuralFeature to be read or written.

## Constraints

- multiplicity  
The multiplicity of the object InputPin must be 1..1.

```
inv: object.is(1,1)
```

- object\_type  
The structuralFeature must either be an owned or inherited feature of the type of the object InputPin, or it must be an owned end of a binary Association whose opposite end had as a type to which the type of the object InputPin conforms.

```
inv: object.type.oclassType(Classifier).allFeatures()->includes(structuralFeature) or  
object.type.conformsTo(structuralFeature.oclassType(Property).opposite.type)
```

- visibility  
The visibility of the structuralFeature must allow access from the object performing the ReadStructuralFeatureAction.

```
inv: structuralFeature.visibility = VisibilityKind::public or  
_'context'.allFeatures()->includes(structuralFeature) or  
structuralFeature.visibility=VisibilityKind::protected and  
_'context'.conformsTo(structuralFeature.oclassType(Property).opposite.type.oclassType(Classifier))
```

- not\_static  
The structuralFeature must not be static.

```
inv: not structuralFeature.isStatic
```

- one\_featuring\_classifier  
The structuralFeature must have exactly one featuringClassifier.

```
inv: structuralFeature.featuringClassifier->size() = 1
```

## StructuredActivityNode [Class]

### Description

A StructuredActivityNode is an Action that is also an ActivityGroup and whose behavior is specified by the ActivityNodes and ActivityEdges it so contains. Unlike other kinds of ActivityGroup, a StructuredActivityNode owns the ActivityNodes and

ActivityEdges it contains, and so a node or edge can only be directly contained in one StructuredActivityNode, though StructuredActivityNodes may be nested.

## Diagrams

[Structured Actions](#), [Expansion Regions](#)

## Generalizations

[Namespace](#), [ActivityGroup](#), [Action](#)

## Specializations

[ConditionalNode](#), [ExpansionRegion](#), [LoopNode](#), [SequenceNode](#)

## Attributes

- mustIsolate : [Boolean](#) [1..1] = false  
If true, then any object used by an Action within the StructuredActivityNode cannot be accessed by any Action outside the node until the StructuredActivityNode as a whole completes. Any concurrent Actions that would result in accessing such objects are required to have their execution deferred until the completion of the StructuredActivityNode.

## Association Ends

- activity : [Activity](#) [0..1]{redefines [ActivityGroup::inActivity](#), redefines [ActivityNode::activity](#)} (opposite [Activity::structuredNode](#))  
The Activity immediately containing the StructuredActivityNode, if it is not contained in another StructuredActivityNode.
- ♦ edge : [ActivityEdge](#) [0..\*]{subsets [ActivityGroup::containedEdge](#), subsets [Element::ownedElement](#)} (opposite [ActivityEdge::inStructuredNode](#))  
The ActivityEdges immediately contained in the StructuredActivityNode.
- ♦ node : [ActivityNode](#) [0..\*]{subsets [Element::ownedElement](#), subsets [ActivityGroup::containedNode](#)} (opposite [ActivityNode::inStructuredNode](#))  
The ActivityNodes immediately contained in the StructuredActivityNode.
- ♦ structuredNodeInput : [InputPin](#) [0..\*]{subsets [Action::input](#)} (opposite [A\\_structuredNodeInput\\_structuredActivityNode::structuredActivityNode](#))  
The InputPins owned by the StructuredActivityNode.
- ♦ structuredNodeOutput : [OutputPin](#) [0..\*]{subsets [Action::output](#)} (opposite [A\\_structuredNodeOutput\\_structuredActivityNode::structuredActivityNode](#))  
The OutputPins owned by the StructuredActivityNode.
- ♦ variable : [Variable](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [Variable::scope](#))  
The Variables defined in the scope of the StructuredActivityNode.

## Operations

- `allActions() : Action [0..*]`  
Returns this `StructuredActivityNode` and all `Actions` contained in it.  
  
`body: node->select(oclIsKindOf(Action)).oclAsType(Action).allActions()->including(self)->asSet()`
- `allOwnedNodes() : ActivityNode [0..*]`  
Returns all the `ActivityNodes` contained directly or indirectly within this `StructuredActivityNode`, in addition to the `Pins` of the `StructuredActivityNode`.  
  
`body: self.Action::allOwnedNodes()->union(node)->union(node->select(oclIsKindOf(Action)).oclAsType(Action).allOwnedNodes())->asSet()`
- `sourceNodes() : ActivityNode [0..*]`  
Return those `ActivityNodes` contained immediately within the `StructuredActivityNode` that may act as sources of edges owned by the `StructuredActivityNode`.  
  
`body: node->union(input.oclAsType(ActivityNode)->asSet())->union(node->select(oclIsKindOf(Action)).oclAsType(Action).output)->asSet()`
- `targetNodes() : ActivityNode [0..*]`  
Return those `ActivityNodes` contained immediately within the `StructuredActivityNode` that may act as targets of edges owned by the `StructuredActivityNode`.  
  
`body: node->union(output.oclAsType(ActivityNode)->asSet())->union(node->select(oclIsKindOf(Action)).oclAsType(Action).input)->asSet()`
- `containingActivity() : Activity [0..1]`  
The `Activity` that directly or indirectly contains this `StructuredActivityNode` (considered as an `Action`).  
  
`body: self.Action::containingActivity()`

## Constraints

- `output_pin_edges`  
The outgoing `ActivityEdges` of the `OutputPins` of a `StructuredActivityNode` must have targets that are not within the `StructuredActivityNode`.  
  
`inv: output.outgoing.target->excludesAll(allOwnedNodes()-input)`
- `edges`  
The edges of a `StructuredActivityNode` are all the `ActivityEdges` with source and target `ActivityNodes` contained directly or indirectly within the `StructuredActivityNode` and at least one of the source or target not contained in any more deeply nested `StructuredActivityNode`.  
  
`inv: edge=self.sourceNodes().outgoing->intersection(self.allOwnedNodes().incoming)->union(self.targetNodes().incoming->intersection(self.allOwnedNodes().outgoing))->asSet()`

- **input\_pin\_edges**  
The incoming ActivityEdges of an InputPin of a StructuredActivityNode must have sources that are not within the StructuredActivityNode.

```
inv: input.incoming.source->excludesAll (allOwnedNodes()-output)
```

## TestIdentityAction [Class]

### Description

A TestIdentityAction is an Action that tests if two values are identical objects.

### Diagrams

[Object Actions](#)

### Generalizations

[Action](#)

### Association Ends

- **first** : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A\\_first\\_testIdentityAction::testIdentityAction](#))  
The InputPin on which the first input object is placed.
- **result** : [OutputPin](#) [1..1]{subsets [Action::output](#)} (opposite [A\\_result\\_testIdentityAction::testIdentityAction](#))  
The OutputPin whose Boolean value indicates whether the two input objects are identical.
- **second** : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A\\_second\\_testIdentityAction::testIdentityAction](#))  
The OutputPin on which the second input object is placed.

### Constraints

- **multiplicity**  
The multiplicity of the InputPins is 1..1.

```
inv: first.is(1,1) and second.is(1,1)
```

- **no\_type**  
The InputPins have no type.

```
inv: first.type= null and second.type = null
```

- **result\_is\_boolean**  
The type of the result OutputPin is Boolean.

```
inv: result.type=Boolean
```

## UnmarshallAction [Class]

### Description

An UnmarshallAction is an Action that retrieves the values of the StructuralFeatures of an object and places them on OutputPins.

### Diagrams

[Accept Event Actions](#)

### Generalizations

[Action](#)

### Association Ends

- ♦ object : [InputPin](#) [1..1]{subsets [Action::input](#)} (opposite [A\\_object\\_unmarshallAction::unmarshallAction](#))  
The InputPin that gives the object to be unmarshalled.
- ♦ result : [OutputPin](#) [1..\*]{ordered, subsets [Action::output](#)} (opposite [A\\_result\\_unmarshallAction::unmarshallAction](#))  
The OutputPins on which are placed the values of the StructuralFeatures of the input object.
- unmarshallType : [Classifier](#) [1..1] (opposite [A\\_unmarshallType\\_unmarshallAction::unmarshallAction](#))  
The type of the object to be unmarshalled.

### Constraints

- structural\_feature  
The unmarshallType must have at least one StructuralFeature.  

```
inv: unmarshallType.allAttributes()->size() >= 1
```
- number\_of\_result  
The number of result outputPins must be the same as the number of attributes of the unmarshallType.  

```
inv: unmarshallType.allAttributes()->size() = result->size()
```
- type\_ordering\_and\_multiplicity  
The type, ordering and multiplicity of each attribute of the unmarshallType must be compatible with the type, ordering and multiplicity of the corresponding result OutputPin.  

```
inv: let attribute:OrderedSet(Property) = unmarshallType.allAttributes() in  
Sequence{1..result->size()->forall(i |  
  attribute->at(i).type.conformsTo(result->at(i).type) and  
  attribute->at(i).isOrdered=result->at(i).isOrdered and  
  attribute->at(i).compatibleWith(result->at(i)))
```
- multiplicity\_of\_object  
The multiplicity of the object InputPin is 1..1  

```
inv: object.is(1,1)
```

- `object_type`  
The type of the object `InputPin` conform to the `unmarshallType`.

```
inv: object.type.conformsTo(unmarshallType)
```

## ValuePin [Class]

### Description

A `ValuePin` is an `InputPin` that provides a value by evaluating a `ValueSpecification`.

### Diagrams

[Actions](#)

### Generalizations

[InputPin](#)

### Association Ends

- ♦ `value` : [ValueSpecification](#) [1..1]{subsets [Element::ownedElement](#)} (opposite [A\\_value\\_valuePin::valuePin](#))  
The `ValueSpecification` that is evaluated to obtain the value that the `ValuePin` will provide.

### Constraints

- `no_incoming_edges`  
A `ValuePin` may have no incoming `ActivityEdges`.

```
inv: incoming->isEmpty()
```

- `compatible_type`  
The type of the value `ValueSpecification` must conform to the type of the `ValuePin`.

```
inv: value.type.conformsTo(type)
```

## ValueSpecificationAction [Class]

### Description

A `ValueSpecificationAction` is an `Action` that evaluates a `ValueSpecification` and provides a result.

### Diagrams

[Object Actions](#)

### Generalizations

[Action](#)

## Association Ends

- ♦ result : [OutputPin](#) [1..1]{subsets [Action::output](#)} (opposite [A\\_result\\_valueSpecificationAction::valueSpecificationAction](#))  
The OutputPin on which the result value is placed.
- ♦ value : [ValueSpecification](#) [1..1]{subsets [Element::ownedElement](#)} (opposite [A\\_value\\_valueSpecificationAction::valueSpecificationAction](#))  
The ValueSpecification to be evaluated.

## Constraints

- multiplicity  
The multiplicity of the result OutputPin is 1..1  

```
inv: result.is(1,1)
```
- compatible\_type  
The type of the value ValueSpecification must conform to the type of the result OutputPin.  

```
inv: value.type.conformsTo(result.type)
```

## VariableAction [Abstract Class]

### Description

VariableAction is an abstract class for Actions that operate on a specified Variable.

### Diagrams

[Variable Actions](#)

### Generalizations

[Action](#)

### Specializations

[WriteVariableAction](#), [ClearVariableAction](#), [ReadVariableAction](#)

## Association Ends

- variable : [Variable](#) [1..1] (opposite [A\\_variable\\_variableAction::variableAction](#))  
The Variable to be read or written.

## Constraints

- scope\_of\_variable  
The VariableAction must be in the scope of the variable.



```
inv: variable.isAccessibleBy(self)
```

## WriteLinkAction [Abstract Class]

### Description

WriteLinkAction is an abstract class for LinkActions that create and destroy links.

### Diagrams

[Link Actions](#)

### Generalizations

[LinkAction](#)

### Specializations

[CreateLinkAction](#), [DestroyLinkAction](#)

### Constraints

- allow\_access  
The visibility of at least one end must allow access from the context Classifier of the WriteLinkAction.

```
inv: endData.end->exists(end |
  end.type='_context' or
  end.visibility=VisibilityKind::public or
  end.visibility=VisibilityKind::protected and
  endData.end->exists(other |
    other<>end and '_context'.conformsTo(other.type.oclAsType(Classifier))))
```

## WriteStructuralFeatureAction [Abstract Class]

### Description

WriteStructuralFeatureAction is an abstract class for StructuralFeatureActions that change StructuralFeature values.

### Diagrams

[Structural Feature Actions](#)

### Generalizations

[StructuralFeatureAction](#)

### Specializations

[AddStructuralFeatureValueAction](#), [RemoveStructuralFeatureValueAction](#)

### Association Ends

- ♦ result : [OutputPin](#) [0..1]{subsets [Action::output](#)} (opposite [A result writeStructuralFeatureAction::writeStructuralFeatureAction](#))

The OutputPin on which is put the input object as modified by the WriteStructuralFeatureAction.

- ♦ value : [InputPin](#) [0..1]{subsets [Action::input](#)} (opposite [A value writeStructuralFeatureAction::writeStructuralFeatureAction](#))  
The InputPin that provides the value to be added or removed from the StructuralFeature.

## Constraints

- multiplicity\_of\_result  
The multiplicity of the result OutputPin must be 1..1.  

```
inv: result <> null implies result.is(1,1)
```
- type\_of\_value  
The type of the value InputPin must conform to the type of the structuralFeature.  

```
inv: value <> null implies value.type.conformsTo(structuralFeature.type)
```
- multiplicity\_of\_value  
The multiplicity of the value InputPin is 1..1.  

```
inv: value<>null implies value.is(1,1)
```
- type\_of\_result  
The type of the result OutputPin is the same as the type of the inherited object InputPin.  

```
inv: result <> null implies result.type = object.type
```

## WriteVariableAction [Abstract Class]

### Description

WriteVariableAction is an abstract class for VariableActions that change Variable values.

### Diagrams

[Variable Actions](#)

### Generalizations

[VariableAction](#)

### Specializations

[AddVariableValueAction](#), [RemoveVariableValueAction](#)

## Association Ends

- ♦ value : [InputPin](#) [0..1]{subsets [Action::input](#)} (opposite [A\\_value\\_writeVariableAction::writeVariableAction](#))  
The InputPin that gives the value to be added or removed from the Variable.

## Constraints

- value\_type  
The type of the value InputPin must conform to the type of the variable.

```
inv: value <> null implies value.type.conformsTo(variable.type)
```

- multiplicity  
The multiplicity of the value InputPin is 1..1.

```
inv: value<>null implies value.is(1,1)
```

## 16.15 Association Descriptions

### A\_argument\_invocationAction [Association]

#### Diagrams

[Invocation Actions](#)

#### Specializations

[A\\_request\\_sendObjectAction](#)

#### Owned Ends

- invocationAction : [InvocationAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [InvocationAction::argument](#))

### A\_association\_clearAssociationAction [Association]

#### Diagrams

[Link Actions](#)

#### Owned Ends

- clearAssociationAction : [ClearAssociationAction](#) [0..1] (opposite [ClearAssociationAction::association](#))

## A\_behavior\_callBehaviorAction [Association]

### Diagrams

[Invocation Actions](#)

### Owned Ends

- callBehaviorAction : [CallBehaviorAction](#) [0..\*] (opposite [CallBehaviorAction::behavior](#))

## A\_bodyOutput\_clause [Association]

### Diagrams

[Structured Actions](#)

### Owned Ends

- clause : [Clause](#) [0..\*] (opposite [Clause::bodyOutput](#))

## A\_bodyOutput\_loopNode [Association]

### Diagrams

[Structured Actions](#)

### Owned Ends

- loopNode : [LoopNode](#) [0..\*] (opposite [LoopNode::bodyOutput](#))

## A\_bodyPart\_loopNode [Association]

### Diagrams

[Structured Actions](#)

### Owned Ends

- loopNode : [LoopNode](#) [0..1] (opposite [LoopNode::bodyPart](#))

## A\_body\_clause [Association]

### Diagrams

[Structured Actions](#)

### Owned Ends

- clause : [Clause](#) [0..1] (opposite [Clause::body](#))

## A\_classifier\_createObjectAction [Association]

### Diagrams

[Object Actions](#)

### Owned Ends

- createObjectAction : [CreateObjectAction](#) [0..\*] (opposite [CreateObjectAction::classifier](#))

## A\_classifier\_readExtentAction [Association]

### Diagrams

[Object Actions](#)

### Owned Ends

- readExtentAction : [ReadExtentAction](#) [0..1] (opposite [ReadExtentAction::classifier](#))

## A\_classifier\_readIsClassifiedObjectAction [Association]

### Diagrams

[Object Actions](#)

### Owned Ends

- readIsClassifiedObjectAction : [ReadIsClassifiedObjectAction](#) [0..\*] (opposite [ReadIsClassifiedObjectAction::classifier](#))

## A\_clause\_conditionalNode [Association]

### Diagrams

[Structured Actions](#)

### Owned Ends

- conditionalNode : [ConditionalNode](#) [1..1]{subsets [Element::owner](#)} (opposite [ConditionalNode::clause](#))

## A\_collection\_reduceAction [Association]

### Diagrams

[Other Actions](#)

### Owned Ends

- reduceAction : [ReduceAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [ReduceAction::collection](#))

## A\_context\_action [Association]

### Diagrams

[Actions](#)

### Owned Ends

- action : [Action](#) [0..\*] (opposite [Action::context](#))

## A\_decider\_clause [Association]

### Diagrams

[Structured Actions](#)

### Owned Ends

- clause : [Clause](#) [0..1] (opposite [Clause::decider](#))

## A\_decider\_loopNode [Association]

### Diagrams

[Structured Actions](#)

### Owned Ends

- loopNode : [LoopNode](#) [0..1] (opposite [LoopNode::decider](#))

## A\_destroyAt\_linkEndDestructionData [Association]

### Diagrams

[Link End Data](#)

### Owned Ends

- linkEndDestructionData : [LinkEndDestructionData](#) [0..1] (opposite [LinkEndDestructionData::destroyAt](#))

## A\_edge\_inStructuredNode [Association]

### Diagrams

[Structured Actions](#)

### Member Ends

- [StructuredActivityNode::edge](#)
- [ActivityEdge::inStructuredNode](#)

## A\_endData\_createLinkAction [Association]

### Diagrams

[Link Actions](#)

### Generalizations

[A\\_endData\\_linkAction](#)

### Owned Ends

- createLinkAction : [CreateLinkAction](#) [1..1]{redefines [A\\_endData\\_linkAction::linkAction](#)} (opposite [CreateLinkAction::endData](#))

## A\_endData\_destroyLinkAction [Association]

### Diagrams

[Link Actions](#)

### Generalizations

[A\\_endData\\_linkAction](#)

### Owned Ends

- destroyLinkAction : [DestroyLinkAction](#) [1..1]{redefines [A\\_endData\\_linkAction::linkAction](#)} (opposite [DestroyLinkAction::endData](#))

## A\_endData\_linkAction [Association]

### Diagrams

[Link Actions](#)

### Specializations

[A\\_endData\\_destroyLinkAction](#), [A\\_endData\\_createLinkAction](#)

### Owned Ends

- linkAction : [LinkAction](#) [1..1]{subsets [Element::owner](#)} (opposite [LinkAction::endData](#))

## A\_end\_linkEndData [Association]

### Diagrams

[Link End Data](#)

### Owned Ends

- linkEndData : [LinkEndData](#) [0..\*] (opposite [LinkEndData::end](#))

## A\_end\_readLinkObjectEndAction [Association]

### Diagrams

[Link Object Actions](#)



## Owned Ends

- readLinkObjectEndAction : [ReadLinkObjectEndAction](#) [0..1] (opposite [ReadLinkObjectEndAction::end](#))

## A\_exception\_raiseExceptionAction [Association]

### Diagrams

[Other Actions](#)

## Owned Ends

- raiseExceptionAction : [RaiseExceptionAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [RaiseExceptionAction::exception](#))

## A\_executableNode\_sequenceNode [Association]

### Diagrams

[Structured Actions](#)

## Owned Ends

- sequenceNode : [SequenceNode](#) [0..1]{subsets [ActivityNode::inStructuredNode](#)} (opposite [SequenceNode::executableNode](#))

## A\_first\_testIdentityAction [Association]

### Diagrams

[Object Actions](#)

## Owned Ends

- testIdentityAction : [TestIdentityAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [TestIdentityAction::first](#))

## A\_fromAction\_actionInputPin [Association]

### Diagrams

[Actions](#)

## Owned Ends

- actionInputPin : [ActionInputPin](#) [0..1]{subsets [Element::owner](#)} (opposite [ActionInputPin::fromAction](#))

## A\_inputElement\_regionAsInput [Association]

### Diagrams

[Expansion Regions](#)

### Member Ends

- [ExpansionRegion::inputElement](#)
- [ExpansionNode::regionAsInput](#)

## A\_inputValue\_linkAction [Association]

### Diagrams

[Link Actions](#)

### Owned Ends

- linkAction : [LinkAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [LinkAction::inputValue](#))

## A\_inputValue\_opaqueAction [Association]

### Diagrams

[Actions](#)

### Owned Ends

- opaqueAction : [OpaqueAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [OpaqueAction::inputValue](#))

## A\_input\_action [Association]

### Diagrams

[Actions](#)

## Owned Ends

- action : [Action](#) [0..1]{subsets [Element::owner](#)} (opposite [Action::input](#))

## A\_insertAt\_addStructuralFeatureValueAction [Association]

### Diagrams

[Structural Feature Actions](#)

## Owned Ends

- addStructuralFeatureValueAction : [AddStructuralFeatureValueAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [AddStructuralFeatureValueAction::insertAt](#))

## A\_insertAt\_addVariableValueAction [Association]

### Diagrams

[Variable Actions](#)

## Owned Ends

- addVariableValueAction : [AddVariableValueAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [AddVariableValueAction::insertAt](#))

## A\_insertAt\_linkEndCreationData [Association]

### Diagrams

[Link End Data](#)

## Owned Ends

- linkEndCreationData : [LinkEndCreationData](#) [0..1] (opposite [LinkEndCreationData::insertAt](#))

## A\_localPostcondition\_action [Association]

### Diagrams

[Actions](#)

## Owned Ends

- action : [Action](#) [0..1]{subsets [Element::owner](#)} (opposite [Action::localPostcondition](#))

## A\_localPrecondition\_action [Association]

### Diagrams

[Actions](#)

## Owned Ends

- action : [Action](#) [0..1]{subsets [Element::owner](#)} (opposite [Action::localPrecondition](#))

## A\_loopVariableInput\_loopNode [Association]

### Diagrams

[Structured Actions](#)

## Owned Ends

- loopNode : [LoopNode](#) [0..1]{subsets [A\\_structuredNodeInput\\_structuredActivityNode::structuredActivityNode](#)} (opposite [LoopNode::loopVariableInput](#))

## A\_loopVariable\_loopNode [Association]

### Diagrams

[Structured Actions](#)

## Owned Ends

- loopNode : [LoopNode](#) [0..1]{subsets [Element::owner](#)} (opposite [LoopNode::loopVariable](#))

## A\_newClassifier\_reclassifyObjectAction [Association]

### Diagrams

[Object Actions](#)

## Owned Ends

- reclassifyObjectAction : [ReclassifyObjectAction](#) [0..\*] (opposite [ReclassifyObjectAction::newClassifier](#))

## A\_node\_inStructuredNode [Association]

### Diagrams

[Structured Actions](#)

### Member Ends

- [StructuredActivityNode::node](#)
- [ActivityNode::inStructuredNode](#)

## A\_object\_clearAssociationAction [Association]

### Diagrams

[Link Actions](#)

### Owned Ends

- clearAssociationAction : [ClearAssociationAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [ClearAssociationAction::object](#))

## A\_object\_readIsClassifiedObjectAction [Association]

### Diagrams

[Object Actions](#)

### Owned Ends

- readIsClassifiedObjectAction : [ReadIsClassifiedObjectAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [ReadIsClassifiedObjectAction::object](#))

## A\_object\_readLinkObjectEndAction [Association]

### Diagrams

[Link Object Actions](#)

## Owned Ends

- readLinkObjectEndAction : [ReadLinkObjectEndAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [ReadLinkObjectEndAction::object](#))

## A\_object\_readLinkObjectEndQualifierAction [Association]

### Diagrams

[Link Object Actions](#)

## Owned Ends

- readLinkObjectEndQualifierAction : [ReadLinkObjectEndQualifierAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [ReadLinkObjectEndQualifierAction::object](#))

## A\_object\_reclassifyObjectAction [Association]

### Diagrams

[Object Actions](#)

## Owned Ends

- reclassifyObjectAction : [ReclassifyObjectAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [ReclassifyObjectAction::object](#))

## A\_object\_startClassifierBehaviorAction [Association]

### Diagrams

[Object Actions](#)

## Owned Ends

- startClassifierBehaviorAction : [StartClassifierBehaviorAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [StartClassifierBehaviorAction::object](#))

## A\_object\_startObjectBehaviorAction [Association]

### Diagrams

[Invocation Actions](#)

### Owned Ends

- startObjectBehaviorAction : [StartObjectBehaviorAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [StartObjectBehaviorAction::object](#))

## A\_object\_structuralFeatureAction [Association]

### Diagrams

[Structural Feature Actions](#)

### Owned Ends

- structuralFeatureAction : [StructuralFeatureAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [StructuralFeatureAction::object](#))

## A\_object\_unmarshallAction [Association]

### Diagrams

[Accept Event Actions](#)

### Owned Ends

- unmarshallAction : [UnmarshallAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [UnmarshallAction::object](#))

## A\_oldClassifier\_reclassifyObjectAction [Association]

### Diagrams

[Object Actions](#)

### Owned Ends

- reclassifyObjectAction : [ReclassifyObjectAction](#) [0..\*] (opposite [ReclassifyObjectAction::oldClassifier](#))

## A\_onPort\_invocationAction [Association]

### Diagrams

[Invocation Actions](#)

### Owned Ends

- invocationAction : [InvocationAction](#) [0..\*] (opposite [InvocationAction::onPort](#))

## A\_operation\_callOperationAction [Association]

### Diagrams

[Invocation Actions](#)

### Owned Ends

- callOperationAction : [CallOperationAction](#) [0..\*] (opposite [CallOperationAction::operation](#))

## A\_outputElement\_regionAsOutput [Association]

### Diagrams

[Expansion Regions](#)

### Member Ends

- [ExpansionRegion::outputElement](#)
- [ExpansionNode::regionAsOutput](#)

## A\_outputValue\_opaqueAction [Association]

### Diagrams

[Actions](#)

### Owned Ends

- opaqueAction : [OpaqueAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [OpaqueAction::outputValue](#))



## A\_output\_action [Association]

### Diagrams

[Actions](#)

### Owned Ends

- action : [Action](#) [0..1]{subsets [Element::owner](#)} (opposite [Action::output](#))

## A\_predecessorClause\_successorClause [Association]

### Diagrams

[Structured Actions](#)

### Member Ends

- [Clause::predecessorClause](#)
- [Clause::successorClause](#)

## A\_qualifier\_linkEndData [Association]

### Diagrams

[Link End Data](#)

### Owned Ends

- linkEndData : [LinkEndData](#) [1..1]{subsets [Element::owner](#)} (opposite [LinkEndData::qualifier](#))

## A\_qualifier\_qualifierValue [Association]

### Diagrams

[Link End Data](#)

### Owned Ends

- qualifierValue : [QualifierValue](#) [0..\*] (opposite [QualifierValue::qualifier](#))

## A\_qualifier\_readLinkObjectEndQualifierAction [Association]

### Diagrams

[Link Object Actions](#)

### Owned Ends

- readLinkObjectEndQualifierAction : [ReadLinkObjectEndQualifierAction](#) [0..1] (opposite [ReadLinkObjectEndQualifierAction::qualifier](#))

## A\_reducer\_reduceAction [Association]

### Diagrams

[Other Actions](#)

### Owned Ends

- reduceAction : [ReduceAction](#) [0..\*] (opposite [ReduceAction::reducer](#))

## A\_removeAt\_removeStructuralFeatureValueAction [Association]

### Diagrams

[Structural Feature Actions](#)

### Owned Ends

- removeStructuralFeatureValueAction : [RemoveStructuralFeatureValueAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [RemoveStructuralFeatureValueAction::removeAt](#))

## A\_removeAt\_removeVariableValueAction [Association]

### Diagrams

[Variable Actions](#)

### Owned Ends

- removeVariableValueAction : [RemoveVariableValueAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [RemoveVariableValueAction::removeAt](#))

## A\_replyToCall\_replyAction [Association]

### Diagrams

[Accept Event Actions](#)

### Owned Ends

- replyAction : [ReplyAction](#) [0..1] (opposite [ReplyAction::replyToCall](#))

## A\_replyValue\_replyAction [Association]

### Diagrams

[Accept Event Actions](#)

### Owned Ends

- replyAction : [ReplyAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [ReplyAction::replyValue](#))

## A\_request\_sendObjectAction [Association]

### Diagrams

[Invocation Actions](#)

### Generalizations

[A\\_argument\\_invocationAction](#)

### Owned Ends

- sendObjectAction : [SendObjectAction](#) [0..1]{redefines [A\\_argument\\_invocationAction::invocationAction](#)} (opposite [SendObjectAction::request](#))

## A\_result\_acceptEventAction [Association]

### Diagrams

[Accept Event Actions](#)

## Owned Ends

- acceptEventAction : [AcceptEventAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [AcceptEventAction::result](#))

## A\_result\_callAction [Association]

### Diagrams

[Invocation Actions](#)

## Owned Ends

- callAction : [CallAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [CallAction::result](#))

## A\_result\_clearStructuralFeatureAction [Association]

### Diagrams

[Structural Feature Actions](#)

## Owned Ends

- clearStructuralFeatureAction : [ClearStructuralFeatureAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [ClearStructuralFeatureAction::result](#))

## A\_result\_conditionalNode [Association]

### Diagrams

[Structured Actions](#)

## Owned Ends

- conditionalNode : [ConditionalNode](#) [0..1]{subsets [A\\_structuredNodeOutput\\_structuredActivityNode::structuredActivityNode](#)} (opposite [ConditionalNode::result](#))

## A\_result\_createLinkObjectAction [Association]

### Diagrams

[Link Object Actions](#)

## Owned Ends

- createLinkObjectAction : [CreateLinkObjectAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [CreateLinkObjectAction::result](#))

## A\_result\_createObjectAction [Association]

### Diagrams

[Object Actions](#)

## Owned Ends

- createObjectAction : [CreateObjectAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [CreateObjectAction::result](#))

## A\_result\_loopNode [Association]

### Diagrams

[Structured Actions](#)

## Owned Ends

- loopNode : [LoopNode](#) [0..1]{subsets [A\\_structuredNodeOutput\\_structuredActivityNode::structuredActivityNode](#)} (opposite [LoopNode::result](#))

## A\_result\_readExtentAction [Association]

### Diagrams

[Object Actions](#)

## Owned Ends

- readExtentAction : [ReadExtentAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [ReadExtentAction::result](#))

## A\_result\_readsClassifiedObjectAction [Association]

### Diagrams

[Object Actions](#)

## Owned Ends

- readIsClassifiedObjectAction : [ReadIsClassifiedObjectAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [ReadIsClassifiedObjectAction::result](#))

## A\_result\_readLinkAction [Association]

### Diagrams

[Link Actions](#)

## Owned Ends

- readLinkAction : [ReadLinkAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [ReadLinkAction::result](#))

## A\_result\_readLinkObjectEndAction [Association]

### Diagrams

[Link Object Actions](#)

## Owned Ends

- readLinkObjectEndAction : [ReadLinkObjectEndAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [ReadLinkObjectEndAction::result](#))

## A\_result\_readLinkObjectEndQualifierAction [Association]

### Diagrams

[Link Object Actions](#)

## Owned Ends

- readLinkObjectEndQualifierAction : [ReadLinkObjectEndQualifierAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [ReadLinkObjectEndQualifierAction::result](#))

## A\_result\_readSelfAction [Association]

### Diagrams

[Object Actions](#)

## Owned Ends

- readSelfAction : [ReadSelfAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [ReadSelfAction::result](#))

## A\_result\_readStructuralFeatureAction [Association]

### Diagrams

[Structural Feature Actions](#)

## Owned Ends

- readStructuralFeatureAction : [ReadStructuralFeatureAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [ReadStructuralFeatureAction::result](#))

## A\_result\_readVariableAction [Association]

### Diagrams

[Variable Actions](#)

## Owned Ends

- readVariableAction : [ReadVariableAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [ReadVariableAction::result](#))

## A\_result\_reduceAction [Association]

### Diagrams

[Other Actions](#)

## Owned Ends

- reduceAction : [ReduceAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [ReduceAction::result](#))

## A\_result\_testIdentityAction [Association]

### Diagrams

[Object Actions](#)

## Owned Ends

- testIdentityAction : [TestIdentityAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [TestIdentityAction::result](#))

## A\_result\_unmarshallAction [Association]

### Diagrams

[Accept Event Actions](#)

## Owned Ends

- unmarshallAction : [UnmarshallAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [UnmarshallAction::result](#))

## A\_result\_valueSpecificationAction [Association]

### Diagrams

[Object Actions](#)

## Owned Ends

- valueSpecificationAction : [ValueSpecificationAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [ValueSpecificationAction::result](#))

## A\_result\_writeStructuralFeatureAction [Association]

### Diagrams

[Structural Feature Actions](#)

## Owned Ends

- writeStructuralFeatureAction : [WriteStructuralFeatureAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [WriteStructuralFeatureAction::result](#))

## A\_returnInformation\_acceptCallAction [Association]

### Diagrams

[Accept Event Actions](#)



## Owned Ends

- acceptCallAction : [AcceptCallAction](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [AcceptCallAction::returnInformation](#))

## A\_returnInformation\_replyAction [Association]

### Diagrams

[Accept Event Actions](#)

## Owned Ends

- replyAction : [ReplyAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [ReplyAction::returnInformation](#))

## A\_second\_testIdentityAction [Association]

### Diagrams

[Object Actions](#)

## Owned Ends

- testIdentityAction : [TestIdentityAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [TestIdentityAction::second](#))

## A\_setupPart\_loopNode [Association]

### Diagrams

[Structured Actions](#)

## Owned Ends

- loopNode : [LoopNode](#) [0..1] (opposite [LoopNode::setupPart](#))

## A\_signal\_broadcastSignalAction [Association]

### Diagrams

[Invocation Actions](#)

## Owned Ends

- broadcastSignalAction : [BroadcastSignalAction](#) [0..\*] (opposite [BroadcastSignalAction::signal](#))

## A\_signal\_sendSignalAction [Association]

### Diagrams

[Invocation Actions](#)

## Owned Ends

- sendSignalAction : [SendSignalAction](#) [0..\*] (opposite [SendSignalAction::signal](#))

## A\_structuralFeature\_structuralFeatureAction [Association]

### Diagrams

[Structural Feature Actions](#)

## Owned Ends

- structuralFeatureAction : [StructuralFeatureAction](#) [0..\*] (opposite [StructuralFeatureAction::structuralFeature](#))

## A\_structuredNodeInput\_structuredActivityNode [Association]

### Diagrams

[Structured Actions](#)

## Owned Ends

- structuredActivityNode : [StructuredActivityNode](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [StructuredActivityNode::structuredNodeInput](#))

## A\_structuredNodeOutput\_structuredActivityNode [Association]

### Diagrams

[Structured Actions](#)

## Owned Ends

- structuredActivityNode : [StructuredActivityNode](#) [0..1]{subsets [A\\_output\\_action::action](#)} (opposite [StructuredActivityNode::structuredNodeOutput](#))

## A\_target\_callOperationAction [Association]

### Diagrams

[Invocation Actions](#)

## Owned Ends

- callOperationAction : [CallOperationAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [CallOperationAction::target](#))

## A\_target\_destroyObjectAction [Association]

### Diagrams

[Object Actions](#)

## Owned Ends

- destroyObjectAction : [DestroyObjectAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [DestroyObjectAction::target](#))

## A\_target\_sendObjectAction [Association]

### Diagrams

[Invocation Actions](#)

## Owned Ends

- sendObjectAction : [SendObjectAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [SendObjectAction::target](#))

## A\_target\_sendSignalAction [Association]

### Diagrams

[Invocation Actions](#)

## Owned Ends

- sendSignalAction : [SendSignalAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [SendSignalAction::target](#))

## A\_test\_clause [Association]

### Diagrams

[Structured Actions](#)

## Owned Ends

- clause : [Clause](#) [0..1] (opposite [Clause::test](#))

## A\_test\_loopNode [Association]

### Diagrams

[Structured Actions](#)

## Owned Ends

- loopNode : [LoopNode](#) [0..1] (opposite [LoopNode::test](#))

## A\_trigger\_acceptEventAction [Association]

### Diagrams

[Accept Event Actions](#)

## Owned Ends

- acceptEventAction : [AcceptEventAction](#) [0..1]{subsets [Element::owner](#)} (opposite [AcceptEventAction::trigger](#))

## A\_unmarshallType\_unmarshallAction [Association]

### Diagrams

[Accept Event Actions](#)

## Owned Ends

- unmarshallAction : [UnmarshallAction](#) [0..\*] (opposite [UnmarshallAction::unmarshallType](#))

## A\_value\_linkEndData [Association]

### Diagrams

[Link End Data](#)

## Owned Ends

- linkEndData : [LinkEndData](#) [0..1] (opposite [LinkEndData::value](#))

## A\_value\_qualifierValue [Association]

### Diagrams

[Link End Data](#)

## Owned Ends

- qualifierValue : [QualifierValue](#) [0..1] (opposite [QualifierValue::value](#))

## A\_value\_valuePin [Association]

### Diagrams

[Actions](#)

## Owned Ends

- valuePin : [ValuePin](#) [0..1]{subsets [Element::owner](#)} (opposite [ValuePin::value](#))

## A\_value\_valueSpecificationAction [Association]

### Diagrams

[Object Actions](#)

## Owned Ends

- valueSpecificationAction : [ValueSpecificationAction](#) [0..1]{subsets [Element::owner](#)} (opposite [ValueSpecificationAction::value](#))

## A\_value\_writeStructuralFeatureAction [Association]

### Diagrams

[Structural Feature Actions](#)

## Owned Ends

- writeStructuralFeatureAction : [WriteStructuralFeatureAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [WriteStructuralFeatureAction::value](#))

## A\_value\_writeVariableAction [Association]

### Diagrams

[Variable Actions](#)

## Owned Ends

- writeVariableAction : [WriteVariableAction](#) [0..1]{subsets [A\\_input\\_action::action](#)} (opposite [WriteVariableAction::value](#))

## A\_variable\_scope [Association]

### Diagrams

[Structured Actions](#)

## Member Ends

- [StructuredActivityNode::variable](#)
- [Variable::scope](#)

## A\_variable\_variableAction [Association]

### Diagrams

[Variable Actions](#)

## Owned Ends

- variableAction : [VariableAction](#) [0..\*] (opposite [VariableAction::variable](#))

# 17 Interactions

## 17.1 Summary

### 17.1.1 Overview

Interactions are used in a number of different situations. They are used to get a better grip of an interaction situation for an individual designer or for a group that needs to achieve a common understanding of the situation. Interactions are also used during the more detailed design phase where the precise inter-process communication must be set up according to formal protocols. When testing is performed, the traces of the system can be described as interactions and compared with those of the earlier phases.

In this clause we use the term *trace* to mean “sequence of event occurrences,” which corresponds well with common use in the area of trace-semantics, which is a preferred way to describe the semantics of Interactions. We may denote this by  $\langle \text{eventoccurrence1, eventoccurrence2, ..., eventoccurrence-n} \rangle$ . We are aware that other parts of the UML language definition use the term “trace” for other purposes.

Interaction specifications place partial ordering constraints on allowed and disallowed traces. A partial order restricts the order in which events can (or cannot) occur in any given system trace.

The Interaction package describes the concepts needed to express Interactions, depending on their purpose. An interaction can be displayed in several different types of diagrams: Sequence Diagrams, Interaction Overview Diagrams, and Communication Diagrams. Optional diagram types such as Timing Diagrams and Interaction Tables come in addition. Each type of diagram provides slightly different capabilities that make it more appropriate for certain situations.

Interactions are a common mechanism for describing systems that can be understood and produced, at varying levels of detail, by both professionals of computer systems design, as well as potential end users and stakeholders of (future) systems.

Typically when interactions are produced by designers or by running systems, the case is that the interactions do not tell the complete story. There are normally other legal and possible traces that are not contained within the described interactions. Some projects may, however, request that all possible traces of a system shall be documented through interactions in the form of (e.g., sequence diagrams or similar notations).

The most visible aspects of an Interaction are the messages between lifelines. The sequence of the messages is considered important for the understanding of the situation. The data that the messages convey and the lifelines store may also be very important, but the Interactions do not focus on the manipulation of data even though data can be used to decorate the diagrams.

By *interleaving* we mean the merging of two or more traces such that the events from different traces may come in any order in the resulting trace, while events within the same trace retain their order. Interleaving semantics is different from a semantics where it is perceived that two events may occur at exactly the same time. To explain Interactions we apply an Interleaving Semantics.

### 17.1.2 Basic trace model

The semantics of an Interaction are expressed in terms of a pair  $[P, I]$  where  $P$  is the set of valid traces and  $I$  is the set of invalid traces.  $P \cup I$  need not be the whole universe of traces.

A trace is a sequence of event occurrences denoted  $\langle e1, e2, \dots, en \rangle$ .

An event occurrence may also include information about the values of all relevant objects at this point in time.

Each construct of Interactions (such as CombinedFragments of different kinds) are expressed in terms of how it relates to a pair of sets of traces. For simplicity we normally refer only to the set of valid traces as these traces are those mostly modeled.



Two Interactions are equivalent if their pair of trace-sets are equal.

### 17.1.3 Partial ordering constraints on valid and invalid traces

The set of valid traces is constrained by a partial ordering of the event occurrences in the traces. Likewise, the set of invalid traces is also constrained by a partial ordering of the event occurrences in the trace.

In an interaction diagram each vertical line describes the time-line for a process, where time increases down the page. The distance between two events on a time-line does not represent any literal measurement of time, only that non-zero time has passed.

The instances in an interaction in principle operate independently of each other. No global notion of time is assumed. The only dependencies between the timing of the instances come from the restriction that a message must be sent before it is received.

Along each instance axis the time is running from top to bottom, however, a proper time scale is not assumed. If no coregion or parallel operator is introduced, a total time ordering of events is assumed along each instance.

Events of different instances are ordered via messages, or via the generalized ordering mechanism. See 17.4.3 (Message). A message must first be sent before it is consumed. With the generalized ordering mechanism "orderable events" on different instances (even in different interactions) can be ordered explicitly. No other ordering is prescribed. An interaction specification, therefore, imposes a partial ordering on the set of events being contained. A binary relation which is transitive, antisymmetric and irreflexive is called partial order.

### 17.1.4 Relation of trace model to execution model

In "Common Behavior" Clause [13](#) we find an Execution model, and this is how the Interactions Trace Model relates to the Execution model.

An Interaction is an Emergent Behavior.

An InvocationOccurrence in the Execution model corresponds with an (event) Occurrence in a trace. Occurrences are modeled in an Interaction by OccurrenceSpecifications. Normally in Interaction the action leading to the invocation as such is not described (such as the sending action). However, if it is desirable to go into details, a Behavior (such as an Activity) may be associated with an OccurrenceSpecification. An occurrence in Interactions is normally interpreted to take zero time. Duration is always between occurrences.

Likewise a ReceiveOccurrence in the Execution model is modeled by an OccurrenceSpecification. Similarly the detailed actions following immediately from this reception are often omitted in Interactions, but may also be described explicitly with a Behavior associated with that OccurrenceSpecification.

A Request in the Execution model is modeled by the Message in Interactions.

An Execution in the Execution model is modeled by an ExecutionSpecification in Interactions. An Execution is defined in the trace by two Occurrences, one at the start and one at the end. This corresponds to the StartOccurrence and the CompletionOccurrence of the Execution model.

### 17.1.5 Interaction Diagram Variants

Interaction diagrams come in different variants. A separate subclause defines notation for each of the following Interaction Diagram variants:

- 17.8 Sequence Diagrams - The most common variant is the Sequence Diagram that focuses on the Message interchange between a number of Lifelines.

- 17.9 Communication Diagrams - Communication Diagrams show interactions through an architectural view where the arcs between the communicating Lifelines are decorated with description of the passed Messages and their sequencing.
- 17.10 Interaction Overview Diagrams - Interaction Overview Diagrams define interactions in a way that promotes overview of the control flow. Overview diagrams have notational elements that are similar to certain elements used in Activity diagrams (flow lines, forks, joins, etc.); however, although the notation and the general purpose of these elements is the same in both cases, their detailed semantics are quite different and modelers should not interpret Overview diagrams as if they were Activity diagrams.
- 17.11 Timing Diagrams - Timing Diagrams are used to show interactions when a primary purpose of the diagram is to reason about time. Conformant UML 2.5 tools are not required to implement Timing Diagrams.

In addition to the Interaction Diagram variants in this clause, there is also an optional notation using Interaction Tables ([Annex D](#)).

## 17.2 Interactions

### 17.2.1 Summary

The Interactions subclause specifies the abstract syntax, semantics, and notation for the following metaclasses:

- Interaction
- InteractionFragment
- OccurrenceSpecification
- ExecutionSpecification
- StateInvariant

### 17.2.2 Abstract Syntax

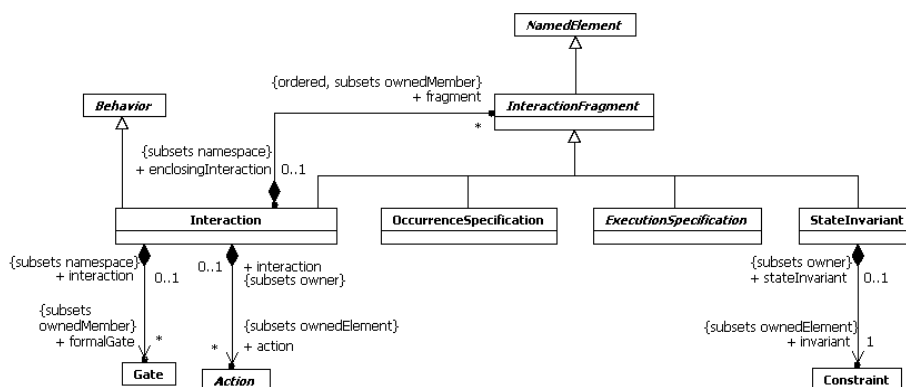


Figure 17.1 - Interactions

## 17.2.3 Semantics

### Interactions

Interactions are units of behavior of an enclosing Classifier. Interactions focus on the passing of information with Messages between the ConnectableElements of the Classifier.

The semantics of an Interaction is given as a pair of sets of traces. The two trace sets represent valid traces and invalid traces. The union of these two sets need not necessarily cover the whole universe of traces. The traces that are not included are not described by this Interaction at all, and we cannot know whether they are valid or invalid.

A trace is a sequence of event occurrences, each of which is described by an OccurrenceSpecification in a model. The semantics of Interactions are compositional in the sense that the semantics of an Interaction is mechanically built from the semantics of its constituent InteractionFragments. The constituent InteractionFragments are ordered and combined by the seq operation (weak sequencing) as explained in 17.6.3 (Weak Sequencing)

The invalid set of traces is associated only with the use of a Negative CombinedInteraction. For simplicity we describe only valid traces for all other constructs.

As Behavior an Interaction is generalizable and redefineable. Specializing an Interaction is simply to add more traces to those of the original. The traces defined by the specialization is combined with those of the inherited Interaction with a union.

The classifier owning an Interaction may be specialized, and in the specialization the Interaction may be redefined. Redefining an Interaction simply means to exchange the redefining Interaction for the redefined one, and this exchange takes effect also for InteractionUses within the supertype of the owner. This is similar to redefinition of other kinds of Behavior.

A formal Gate may be attached to the inner boundary of an Interaction to provide a link point to establish the concrete sender and receiver through an InteractionUse of that Interaction.

### Interaction Fragments

The semantics of an InteractionFragment is a pair of set of traces. See 17.1.2 for explanation of how to calculate the traces.

An InteractionFragment may either be contained directly in an enclosing Interaction, or may be contained within an InteractionOperator of a CombinedFragment. As a CombinedFragment is itself an InteractionFragment, there may be multiple nesting levels of InteractionFragments within an Interaction.

### Occurrence Specifications

The semantics of an OccurrenceSpecification is just the trace of that single OccurrenceSpecification.

The understanding and deeper meaning of the OccurrenceSpecification is dependent upon the associated Message and the information that it conveys.

### Execution Specifications

The trace semantics of Interactions merely see an Execution as the trace <start, finish>. There may be occurrences between these. Typically the start occurrence and the finish occurrence will represent OccurrenceSpecifications such as a receive OccurrenceSpecification (of a Message) and the send OccurrenceSpecification (of a reply Message).

### State Invariants

The Constraint is assumed to be evaluated during runtime. The Constraint is evaluated immediately prior to the execution of the next OccurrenceSpecification such that all actions that are not explicitly modeled have been executed. If the Constraint is true, the trace is a valid trace; if the Constraint is false, the trace is an invalid trace. In other words all traces that have a StateInvariant with a false Constraint are considered invalid.

## 17.2.4 Notation

### Interaction

The notation for an Interaction in a Sequence Diagram is a solid-outline rectangle. The keyword `sd` followed by the Interaction name and parameters is in a pentagon in the upper left corner of the rectangle. The notation within this rectangular frame comes in several forms: Sequence Diagrams, Communication Diagrams, Interaction Overview Diagrams, and Timing Diagrams.

The notation within the pentagon descriptor follows the general notation for the name of Behaviors. In addition the Interaction Overview Diagrams may include a list of Lifelines through a lifeline-clause as shown in Figure 17.27. The list of lifelines is simply a listing of the Lifelines involved in the Interaction. An Interaction Overview Diagram does not in itself show the involved lifelines even though the lifelines may occur explicitly within inline Interactions in the graph nodes.

An Interaction diagram may also include definitions of local attributes with the same syntax as attributes in general are shown within class symbol compartments. These attribute definitions may appear near the top of the diagram frame or within note symbols at other places in the diagram.

### InteractionFragment

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

### OccurrenceSpecification

OccurrenceSpecifications are merely syntactic points at the ends of Messages or at the beginning/end of an ExecutionSpecification.

### ExecutionSpecification

ExecutionSpecifications are represented as thin rectangles (gray or white) on the lifeline (see 17.3.4 (Lifeline)).

We may also represent an ExecutionSpecification by a wider labeled rectangle, where the label usually identifies the action that was executed. An example of this can be seen in Figure 17.16.

For ExecutionSpecifications that refer to atomic actions such as reading attributes of a Signal (conveyed by the Message), the Action symbol may be associated with the reception OccurrenceSpecification with a line in order to emphasize that the whole Action is associated with only one OccurrenceSpecification (and start and finish associations refer to the very same OccurrenceSpecification).

Overlapping ExecutionSpecifications on the same lifeline are represented by overlapping rectangles as shown in Figure 17.2.

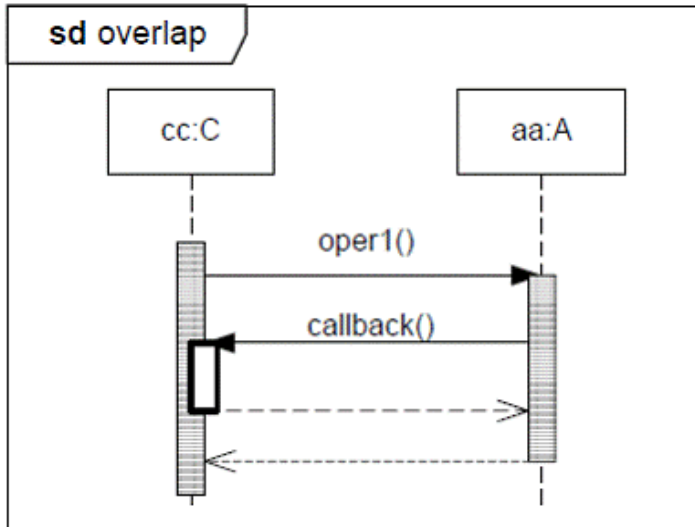


Figure 17.2 - Overlapping Execution Specifications

### StateInvariant

The possible associated Constraint is shown as text in curly brackets on the lifeline. See example in Figure 17.17.

A conforming tool may show a StateInvariant as a Note associated with an OccurrenceSpecification.

The state symbol represents the equivalent of a constraint that checks the state of the object represented by the Lifeline. This could be the internal state of the classifierBehavior of the corresponding Classifier, or it could be some external state based on a “black-box” view of the Lifeline. In the former case, and if the classifierBehavior is described by a state machine, the name of the state should match the hierarchical name of the corresponding state of the state machine.

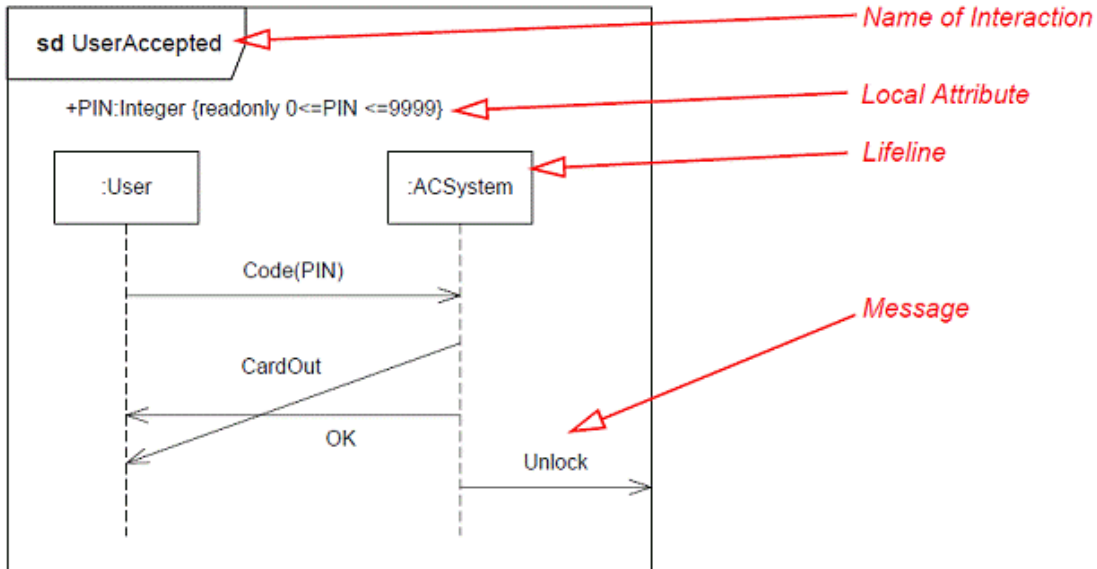
The regions represent the orthogonal regions of states. The identifier need only define the state partially. The value of the constraint is true if the specified state information is true.

The example in Figure 17.17 also shows this presentation option.

### Formal Gate

A formal Gate is just a point on the inside of the frame, as the end of a message. They may have an explicit name (see Figure 17.4).

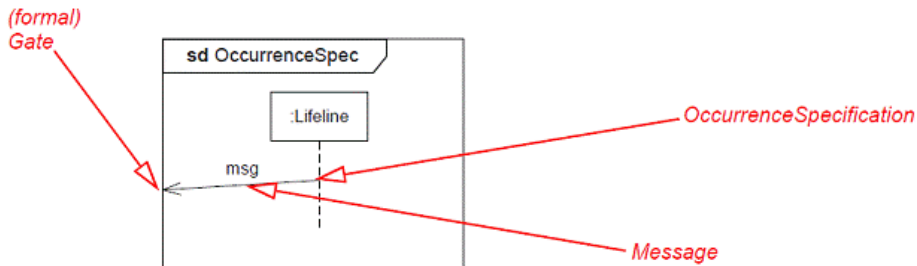
## 17.2.5 Examples



**Figure 17.3 - An example of an Interaction in the form of a Sequence Diagram**

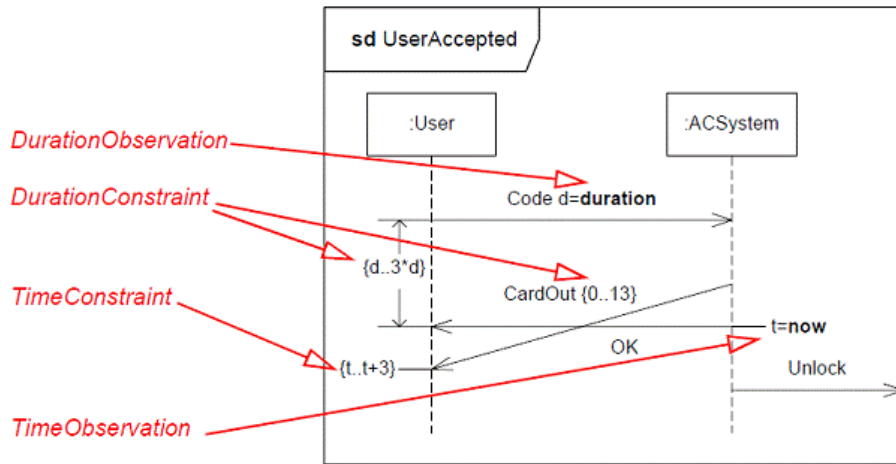
The example in Figure 17.3 shows three messages communicated between two (anonymous) lifelines of types User and ACSysyem. The message CardOut overtakes the message OK in the way that the receiving event occurrences are in the opposite order of the sending OccurrenceSpecifications. Such communication may occur when the messages are asynchronous. Finally a fourth message is sent from the ACSysyem to the environment through a gate with implicit name out\_Unlock. The local attribute PIN of UserAccepted is declared near the diagram top. It could have been declared in a Note somewhere else in the diagram.

An example showing OccurrenceSpecification is shown in Figure 17.4.



**Figure 17.4 - OccurrenceSpecification**

An example with a gate (labeled "Unlock") is shown in Figure 17.5.



**Figure 17.5 - Sequence Diagram with time and timing concepts**

The Sequence Diagram in Figure 17.5 shows how time and timing notation may be applied to describe time observation and timing constraints. The :User sends a message Code and its duration is measured. The :ACSystem will send two messages back to the :User. CardOut is constrained to last between 0 and 13 time units. Furthermore the interval between the sending of Code and the reception of OK is constrained to last between  $d$  and  $3*d$  where  $d$  is the measured duration of the Code signal. We also notice the observation of the time point  $t$  at the sending of OK and how this is used to constrain the time point of the reception of CardOut.

## 17.3 Lifelines

### 17.3.1 Summary

The Lifelines subclause specifies the abstract syntax, semantics, and notation for the following metaclass:

- Lifeline

## 17.3.2 Abstract Syntax

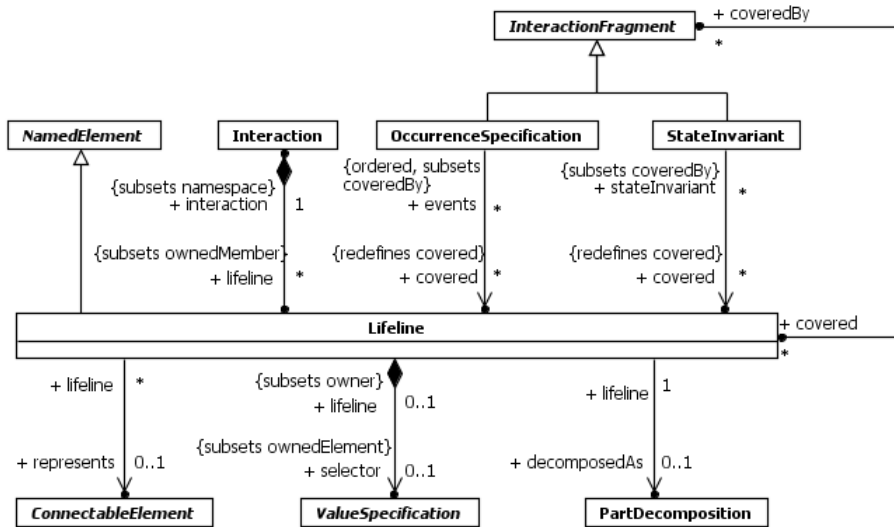


Figure 17.6 - Lifelines

## 17.3.3 Semantics

### Lifelines

In an interaction diagram a Lifeline describes the time-line for a process, where time increases down the page. The distance between two events on a time-line does not represent any literal measurement of time, only that non-zero time has passed.

Events on the same time-line are ordered linearly down the page, except where they occur within a parallel combined fragment, or along a lifeline within a “coregion”. See 17.6.3 (Parallel) and 17.6.4 (Parallel interactionOperator). Within a parallel combined fragment or a coregion, events are not locally ordered unless that is directly imposed by a general ordering construct.. See 17.5.3 (General Ordering).

The order of OccurrenceSpecifications along a Lifeline is significant denoting the order in which these OccurrenceSpecifications will occur. The absolute distances between the OccurrenceSpecifications on the Lifeline are, however, irrelevant for the semantics.

The semantics of the Lifeline (within an Interaction) is the semantics of the Interaction selecting only OccurrenceSpecifications of this Lifeline.

### 17.3.4 Notation

#### Lifeline

A Lifeline is shown using a symbol that consists of a rectangle forming its “head” followed by a vertical line (which may be dashed) that represents the lifetime of the participant. Information identifying the lifeline is displayed inside the rectangle in the following format:

```
<lifelineident> ::= ([<connectable-element-name>[‘<selector> ’]] [: <class_name>] [decomposition]) | ‘self’
```



<selector> ::= <expression>

<decomposition> ::= 'ref' <interactionident> ['strict']

where <class-name> is the type referenced by the represented ConnectableElement. Note that, although the syntax allows it, <lifelineident> cannot be empty.

The Lifeline head has a shape that is based on the classifier for the part that this lifeline represents. Often the head is a white rectangle containing the name.

If the name is the keyword self, then the Lifeline represents the object of the classifier that encloses the Interaction that owns the Lifeline. Ports of the enclosure may be shown separately even when self is included.

To depict method activations we apply a thin gray or white rectangle that covers the Lifeline line.

### 17.3.5 Examples

See Figure 17.3 where the Lifelines are pointed to.

See Figure 17.14 to see method activations.

## 17.4 Messages

### 17.4.1 Summary

The Messages subclause specifies the abstract syntax, semantics, and notation for the following metaclasses:

- Message
- MessageEnd
- MessageOccurrenceSpecification
- MessageSort
- MessageKind
- DestructionOccurrenceSpecification
- Gate



value for the corresponding parameter or attribute. Otherwise, the type of the argument must conform to the type of the corresponding parameter or attribute.

If the Message is a reply, then each of its arguments must be an Expression with at most one operand. If an operand is given, it is considered to be evaluated at the point of the send event of the Message, and its result provides the returned value for the out, inout or return parameter corresponding to the argument. The type of the output parameter must conform to the type of the operand. If no operand is given, then no returned value is modeled by the reply Message for that argument.

The symbol of the argument Expression of a reply Message represents the *assignment target* for the argument, to which the returned value for the argument is to be assigned. The following values for such an assignment-target symbol have standard interpretations:

- *Unknown*. An empty string, which represents an unknown assignment target. An argument with unknown assignment target and no modeled returned value is an output *wildcard*.
- *Interaction Parameter*. The unqualified name of an ownedParameter of the enclosing Interaction, which must be an out, inout or return Parameter. The type of the Operation Parameter corresponding to the argument must conform to the type of the target ownedParameter of the Interaction.
- *Attribute*. The (possibly qualified) name of an attribute of the context Behavior of the enclosing interaction or of the receiving Lifeline of the Message (which is the lifeline that sent the original message to which this is a reply). If an Interaction does not have a context Behavior, then the Interaction itself is considered to be the context. The type of the output parameter corresponding to the argument must conform to the type of the attribute. (Note that a qualified name may be used to distinguish attributes of the context and the Lifeline with the same names, or an attribute from an Interaction Parameter with the same name.)

Other values are allowed for an assignment-target symbol (e.g., for use in a profile), but their interpretation is not defined in this specification.

## Message Ends

Subclasses of MessageEnd define the specific semantics appropriate to the concept they represent.

## Message Occurrence Specifications

A MessageOccurrenceSpecification represents a send event or a receive event associated with a message between two Lifelines.

## Destruction Occurrence Specifications

A DestructionOccurrenceSpecification represents the destruction of the instance described by the lifeline that contains it. It may result in the subsequent destruction of other objects that this object owns by composition (see Clause [13](#)).

## Gates

A Gate is a MessageEnd which is used on the boundary of an Interaction, or an InteractionUse, or a CombinedFragment to establish the concrete sender and receiver for every Message.

There are four kinds of Gate, distinguished by their associations:

1. A formal Gate associated with an Interaction to provide a link point attached to the inside boundary of that Interaction, to convey a Message inside that Interaction to or from a Message in another Interaction which has an actual Gate as a MessageEnd attached to the outside of an InteractionUse of that Interaction.
2. An actual Gate is associated with an InteractionUse to provide a link attached to the outside boundary of an InteractionUse to convey a Message outside the InteractionUse to or from a Message inside the Interaction referred to by the InteractionUse.

3. An inner CombinedFragment Gate is associated with a CombinedFragment to provide a link point attached to the inside boundary of a CombinedFragment to convey a Message with a MessageEnd inside that CombinedFragment to or from a Message with a MessageEnd outside that CombinedFragment.
4. An outer CombinedFragment Gate is associated with a CombinedFragment to provide a link point attached to the outside boundary of a CombinedFragment to convey a Message with a MessageEnd outside that CombinedFragment to or from a Message with a MessageEnd inside that CombinedFragment.

The gates are named either explicitly or implicitly. Gates may be identified either by name (if specified), or by a constructed identifier formed by concatenating the direction of the message and the message name (e.g., out\_CardOut, in\_CardOut).

Gates are matched by name, with a formal Gate matched with a formal Gate having the same name, and with an inner CombinedFragment Gate matched with an outer CombinedFragment Gate having the same name.

The Messages for matched Gates must correspond. Messages correspond if they have identical name, messageSort, and signature property values, as well as being in the same direction.

## 17.4.4 Notation

### Message

A message is shown as a line from the sender MessageEnd to the receiver MessageEnd. The line must be such that every line fragment is either horizontal or downwards when traversed from send event to receive event. The send and receive events may both be on the same lifeline. The form of the line or arrowhead reflects properties of the message:

- An asynchronous Message (messageSort equals asynchCall or asynchSignal) has an open arrow head.
- A synchronous Message (messageSort equals synchCall) has a filled arrow head.
- A reply Message (messageSort equals reply) has a dashed line with a filled arrow head.
- An object creation Message (messageSort equals createMessage) has a dashed line with an open arrow head.
- An object deletion Message (messageSort equals deleteMessage) must end in a DestructionOccurrenceSpecification.
- A lost Message is denoted with a small black circle at the arrow end of the Message.
- A found Message is denoted with a small black circle at the starting end of the Message.
- On Communication Diagrams, the Messages are decorated by a small arrow in the direction of the Message close to the Message name and sequence number along the line between the lifelines (See Table 17.4 and Figure 17.26).

The syntax for the Message label in a diagram is the following:

```
<message-label> ::= <request-message-label> | <reply-message-label> | '*'
```

A message-label equaling '\*' is a shorthand for a more complex alternative CombinedFragment to represent a message of any type. This is to match asterisk triggers in State Machines.

A request-message-label is used for all sorts of Message other than a reply. It has the following form:

```
<request-message-label> ::= <message-name> ['(['<input-argument-list> ')]
```

```
<input-argument-list> ::= <input-argument> [','<input-argument>*]
```

```
<input-argument> ::= [<in-parameter-name> '='] <value-specification> | '-'
```

The message-name appearing in a request-message-label is the name property of the Message. If the Message has a signature, this will be the name of the Operation or Signal referenced by the signature. Otherwise the name is unconstrained.

If a request-message-label includes an input-argument-list, then either all input-arguments must have an in-parameter-name given or none may have one. If in-parameter-names are not given, then the input-arguments denote the arguments of the Message, in order, with a hyphen ('-') denoting a wildcard argument. If the Message has a signature, then the arguments are matched, by order, to the in and inout ownedParameters of an Operation or the attributes of a Signal. An argument must be provided for every such parameter or attribute.

A request-message-label may only have input-arguments with in-parameter-names if the Message has a signature. In this case, the input-arguments are matched by name to the in and inout ownedParameters of an Operation or the attributes of a Signal. Any such parameters or attributes that are not named are considered to have implicit wildcard arguments. The explicit wildcard notation ('-') is not used if in-parameter-names are given.

If a request-message-label does not include an input-argument-list and the Message has a signature, then this denotes that the Message has wildcard arguments corresponding to all in and inout ownedParameters of an Operation or attributes of a Signal (if any). Note that the parentheses are not considered part of the input-argument list, so a request-message-label without an input-argument-list may still optionally include an empty set of parentheses ("()") after the message-name.

A reply-message-label is used for reply Messages. It has the following form:

```
<reply-message-label> ::= [<assignment-target> '='] <message-name>
                        ['(' [<output-argument-list> ')'] [':' <value-specification>]
<output-argument-list> ::= <output-argument> [',' <output-argument>]*
<output-argument> ::= <out-parameter-name> ':' <value-specification> |
                    <assignment-target> '=' <out-parameter-name> [':' <value-specification>]
```

The message-name appearing in a reply-message-label is the name property of the Message. If the Message has a signature, this will be the name of the Operation referenced by the signature (which should be the Operation for whose call this is a reply). Otherwise the name is unconstrained.

A reply-message-label may optionally have an assignment-target given to the left of the message-name, with a corresponding returned value denoted by the optional value-specification given after a colon at the end of the reply-message-label. If the Message has a signature that is an Operation with a return parameter, then this assignment-target and/or value-specification corresponds to the argument for that parameter (if no assignment-target is given, it is considered to be unknown). If the Message has a signature without a return parameter, then no assignment-target or value-specification may be given for the reply-message-label as a whole.

If a reply Message does not have a signature, then the only argument that may be specified for it is a return argument as specified above. However, if the Message has a signature that is an Operation with out or inout ownedParameters, then output-arguments may be provided for these parameters. An output-argument always explicitly names the parameter to which it is to be matched. Any parameters that are not named are considered to have implicit wildcard arguments. (There is thus no need for an explicit wildcard notation for output-arguments.)

If a reply-message-label does not include an output-argument-list and the Message has a signature, then this denotes that the Message has wildcard arguments corresponding to all out and inout ownedParameters of the signature Operation (if any). Note that the parentheses are not considered part of the output-argument list, so a reply-message-label without an output-argument-list may still optionally include an empty set of parentheses ("()") after the message-name.

An output-argument with an explicit assignment-target given may also optionally include a value-specification. If a value-specification is given, then this denotes the returned value for the argument. Otherwise the argument has no modeled returned value. If an output-argument does not have an explicit assignment-target specified, it is considered to have an unknown assignment target. In this case, it is required to include a value-specification, which denotes the returned value for the argument.

If the identity of a reply Message is obvious (e.g., when its sendEvent is the only reply within the extent of an ExecutionOccurrence where there is only one receipt of an Operation call message), the label may be omitted to simplify the diagram. If the reply Message has a signature, then wildcard arguments are provided for all return, out and inout ownedParameters of the signature Operation. See Figure 17.2 for an example.

### **DestructionOccurrenceSpecification**

The DestructionOccurrenceSpecification is depicted by a cross in the form of an X at the bottom of a Lifeline. See Figure 17.8.



**Figure 17.8 - DestructionOccurrenceSpecification symbol**

### **Gate**

Gates are just points on the frame, the ends of the messages. They may have an explicit name (see Figure 17.4).

## **17.4.5 Examples**

In Figure 17.3 we see only asynchronous Messages. Such Messages may overtake each other.

In Figure 17.14 we see method calls that are synchronous accompanied by replies. We also see a Message that represents the creation of an object.

In Figure 17.26 we see how Messages are denoted in Communication Diagrams.

Examples of syntax:

```
mymessage(14, -, 3.14, "hello") // this is a request message; the second argument is a wildcard
mymsg(myint=16) // the is a request message; the input parameter 'myint' is given
// the argument value 16
v=mymsg(w=myout:16):96 // this is a reply message assigning the return value 69 to 'v' and
// the value 16 for the out parameter 'myout to 'w'.
```

See Figure 17.14 for a number of different applications of the textual syntax of message identification.

## **17.5 Occurrences**

### **17.5.1 Summary**

The Occurrences subclause specifies the abstract syntax, semantics, and notation for the following metaclasses:

- ActionExecutionSpecification
- BehaviorExecutionSpecification
- ExecutionOccurrenceSpecification
- GeneralOrdering

## 17.5.2 Abstract Syntax

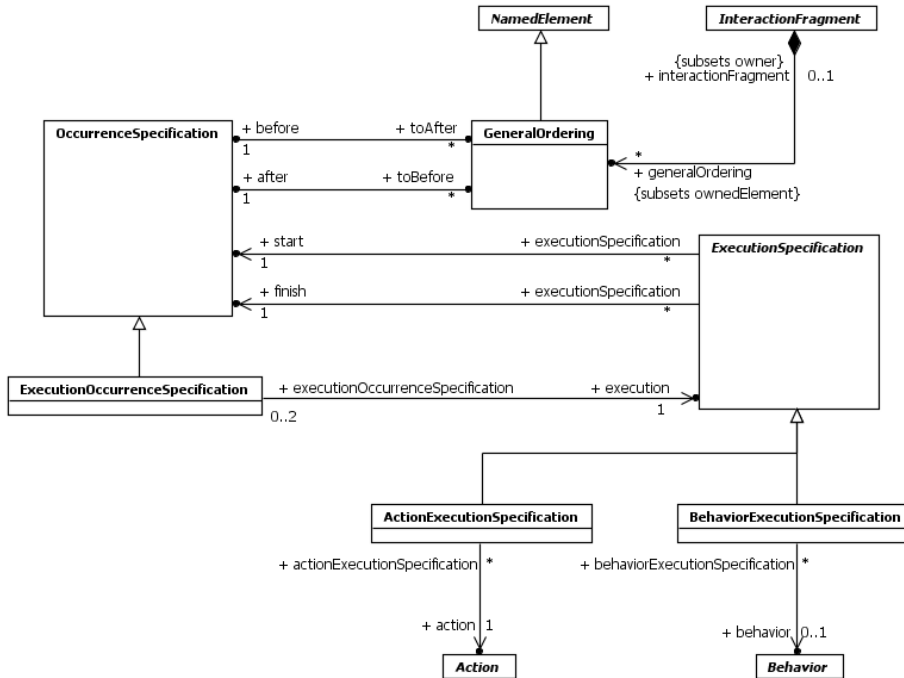


Figure 17.9 - Occurrences

## 17.5.3 Semantics

### Action Execution Specifications

See 17.2.3 (Execution Specification).

ActionExecutionSpecification is used for interactions specifying messages that result from actions, which may be actions owned by other behaviors.

### Behavior Execution Specifications

See 17.2.3 (Execution Specification).

BehaviorExecutionSpecification is used for interactions specifying messages that result from behaviors.

### Execution Occurrence Specifications

An ExecutionOccurrenceSpecification represents, on a lifeline, the start event or the end event of an ExecutionSpecification.

### General Orderings

A GeneralOrdering restricts the set of possible sequences. A partial order of OccurrenceSpecifications is constrained by a set of GeneralOrderings.

## 17.5.4 Notation

### ActionExecutionSpecification

See 17.2.4 (ExecutionSpecification).

### BehaviorExecutionSpecification

See “17.2.4 (ExecutionSpecification).

### ExecutionOccurrenceSpecification

An ExecutionOccurrenceSpecification is represented by the start or finish endpoint of the vertical box for an ExecutionSpecification on a lifeline. See Figure 17.2.

### GeneralOrdering

A GeneralOrdering is shown by a dotted line connecting the two OccurrenceSpecifications. The direction of the relation from the before to the after is given by an arrowhead placed somewhere in the middle of the dotted line (i.e., not at the endpoint).

## 17.5.5 Examples

An example showing a GeneralOrdering is shown in Figure 17.10

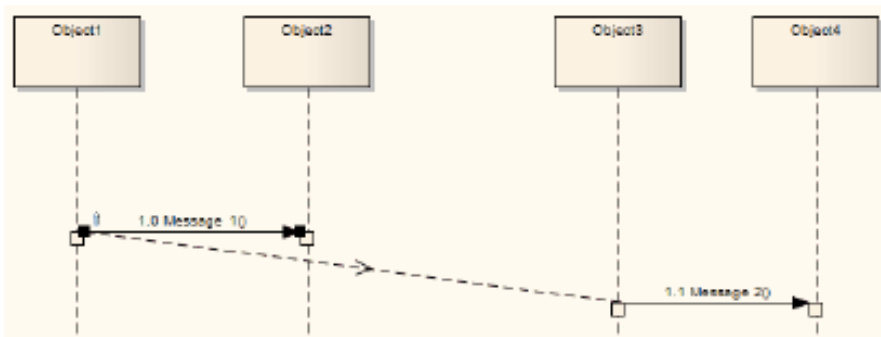


Figure 17.10 - Example showing GeneralOrdering in a sequence diagram

## 17.6 Fragments

### 17.6.1 Summary

The Fragments subclause specifies the abstract syntax, semantics, and notation for the following metaclasses:

- InteractionOperand
- InteractionConstraint
- CombinedFragment
- ConsiderIgnoreFragment
- Continuation



- InteractionOperatorKind

## 17.6.2 Abstract Syntax

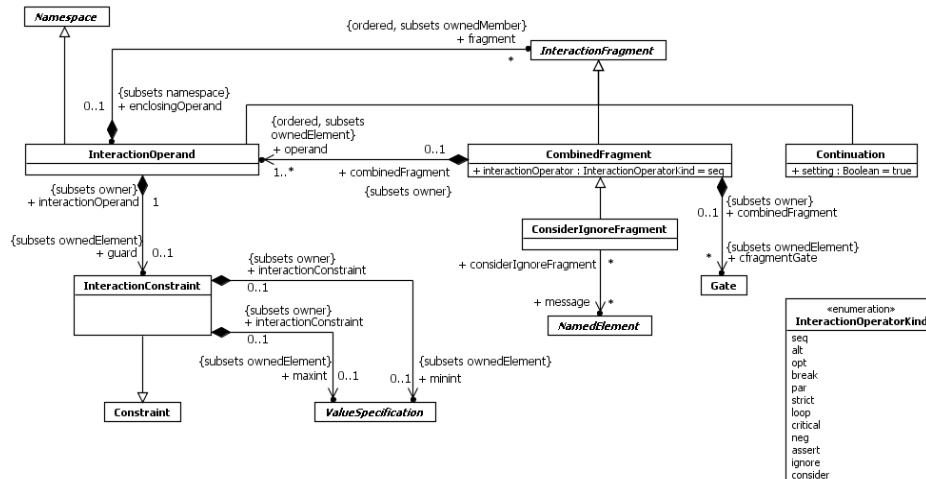


Figure 17.11 - Fragments

## 17.6.3 Semantics

### Interaction Operands

An InteractionOperand is a region within a CombinedFragment, see 17.6.3 (Combined Fragment). Only InteractionOperands with true guards are included in the calculation of the semantics. If no guard is present, this is taken to mean a true guard.

The semantics of an InteractionOperand is given by its constituent InteractionFragments combined by the implicit seq operation. The seq operator is described in “17.6.3 (Combined Fragment).

### Interaction Constraints

InteractionConstraints are always used in connection with CombinedFragments, see 17.6.3 (Combined Fragment).

### Combined Fragments

The semantics of a CombinedFragment is dependent upon the interactionOperator, as explained below for each kind of interactionOperator.

The Gates associated with a CombinedFragment represent the syntactic interface between the CombinedFragment and its surroundings, which means the interface towards other InteractionFragments.

### Consider Ignore Fragments

A ConsiderIgnoreFragment is a CombinedFragment with an Ignore or Consider interactionOperator value. See 17.6.3 (Ignore / Consider).

### Continuations

Continuations have semantics only in connection with Alternative CombinedFragments and (weak) sequencing.

If an InteractionOperand of an Alternative CombinedFragment ends in a Continuation with name (say) X, only InteractionFragments starting with the Continuation X (or no continuation at all) can be appended.

## Interaction Operator Kind Values

The value of the interactionOperator is significant for the semantics of CombinedFragment, as specified below for each interactionOperator enumeration value.

### Alternatives

The interactionOperator **alt** designates that the CombinedFragment represents a choice of behavior. At most one of the operands will be chosen. The chosen operand must have an explicit or implicit guard expression that evaluates to true at this point in the interaction. An implicit true guard is implied if the operand has no guard.

The set of traces that defines a choice is the union of the (guarded) traces of the operands.

An operand guarded by else designates a guard that is the negation of the disjunction of all other guards in the enclosing CombinedFragment.

If none of the operands has a guard that evaluates to true, none of the operands are executed and the remainder of the enclosing InteractionFragment is executed.

If an inner CombinedFragment Gate is used in any InteractionOperand of an alt CombinedFragment, a Gate with that same name must be used by every InteractionOperand of that alt CombinedFragment.

### Option

The interactionOperator **opt** designates that the CombinedFragment represents a choice of behavior where either the (sole) operand happens or nothing happens. An option is semantically equivalent to an alternative CombinedFragment where there is one operand with non-empty content and the second operand is empty.

### Break

The interactionOperator **break** designates that the CombinedFragment represents a breaking scenario in the sense that the operand is a scenario that is performed instead of the remainder of the enclosing InteractionFragment. A break operator with a guard is chosen when the guard is true and the rest of the enclosing Interaction Fragment is ignored. When the guard of the break operand is false, the break operand is ignored and the rest of the enclosing InteractionFragment is chosen. The choice between a break operand without a guard and the rest of the enclosing InteractionFragment is done non-deterministically.

A CombinedFragment with interactionOperator break should cover all Lifelines of the enclosing InteractionFragment.

### Parallel

The interactionOperator **par** designates that the CombinedFragment represents a parallel merge between the behaviors of the operands. The OccurrenceSpecifications of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved.

A parallel merge defines a set of traces that describes all the ways that OccurrenceSpecifications of the operands may be interleaved without obstructing the order of the OccurrenceSpecifications within the operand.

### Weak Sequencing

The interactionOperator **seq** designates that the CombinedFragment represents a weak sequencing between the behaviors of the operands.

Weak sequencing is defined by the set of traces with these properties:

1. The ordering of OccurrenceSpecifications within each of the operands are maintained in the result.
2. OccurrenceSpecifications on different lifelines from different operands may come in any order.
3. OccurrenceSpecifications on the same lifeline from different operands are ordered such that an OccurrenceSpecification of the first operand comes before that of the second operand.

Thus weak sequencing reduces to a parallel merge when the operands are on disjunct sets of participants. Weak sequencing reduces to strict sequencing when the operands work on only one participant.

### Strict Sequencing

The interactionOperator **strict** designates that the CombinedFragment represents a strict sequencing between the behaviors of the operands. The semantics of strict sequencing defines a strict ordering of the operands on the first level within the CombinedFragment with interactionOperator strict. Therefore OccurrenceSpecifications within contained CombinedFragment will not directly be compared with other OccurrenceSpecifications of the enclosing CombinedFragment.

### Negative

The interactionOperator **neg** designates that the CombinedFragment represents traces that are defined to be invalid.

The set of traces that defined a CombinedFragment with interactionOperator negative is equal to the set of traces given by its (sole) operand, only that this set is a set of invalid rather than valid traces. All InteractionFragments that are different from Negative are considered positive meaning that they describe traces that are valid and should be possible.

### Critical Region

The interactionOperator **critical** designates that the CombinedFragment represents a critical region. A critical region means that the traces of the region cannot be interleaved by other OccurrenceSpecifications (on those Lifelines covered by the region). This means that the region is treated atomically by the enclosing fragment when determining the set of valid traces. Even though enclosing CombinedFragments may imply that some OccurrenceSpecifications may interleave into the region, such as with par-operator, this is prevented by defining a region.

Thus the set of traces of enclosing constructs are restricted by critical regions.

### Ignore / Consider

The interactionOperator **ignore** designates that there are some message types that are not shown within this combined fragment. These message types can be considered insignificant and are implicitly ignored if they appear in a corresponding execution. Alternatively, one can understand ignore to mean that the message types that are ignored can appear anywhere in the traces.

Conversely, the interactionOperator **consider** designates which messages should be considered within this combined fragment. This is equivalent to defining every other message to be ignored.

### Assertion

The interactionOperator **assert** designates that the CombinedFragment represents an assertion. The sequences of the operand of the assertion are the only valid continuations. All other continuations result in an invalid trace. Assertions are often combined with Ignore or Consider as shown in Figure 17.17.

### Loop

The interactionOperator **loop** designates that the CombinedFragment represents a loop. The loop operand will be repeated a number of times.

The Guard may include a lower and an upper number of iterations of the loop as well as a Boolean expression. The semantics is such that a loop will iterate minimum the ‘minint’ number of times (given by the iteration expression in the guard) and at most the ‘maxint’ number of times. After the minimum number of iterations have executed and the Boolean expression is false the loop will terminate. The loop construct represents a recursive application of the seq operator where the loop operand is sequenced after the result of earlier iterations.

If the loop contains a separate InteractionConstraint with a specification, the loop will only continue if that specification evaluates to true during execution regardless of the minimum number of iterations specified in the loop.

## 17.6.4 Notation

### InteractionOperand

InteractionOperands are separated by a dashed horizontal line. The InteractionOperands together make up the framed CombinedFragment.

Within an InteractionOperand of a Sequence Diagram the order of the InteractionFragments are given simply by the topmost vertical position.

### InteractionConstraint

An InteractionConstraint is shown in square brackets covering the lifeline where the first event occurrence will occur, positioned above that event, in the containing Interaction or InteractionOperand.

`<interactionconstraint> ::= ‘[‘ (<Boolean-expression> | ‘else‘) ‘]’`

When the InteractionConstraint is omitted, true is assumed.

### CombinedFragment

The notation for a CombinedFragment in a Sequence Diagram is a solid-outline rectangle. The operator is shown in a pentagon in the upper left corner of the rectangle.

More than one operator may be shown in the pentagon descriptor. This is a shorthand for nesting CombinedFragments. This means that sd strict in the pentagon descriptor is the same as two CombinedFragments nested, the outermost with sd and the inner with strict.

The operands of a CombinedFragment are shown by tiling the graph region of the CombinedFragment using dashed horizontal lines to divide it into regions corresponding to the operands.

### ConsiderIgnoreFragment

The notation for ConsiderIgnoreFragment is the same as for all CombinedFragments with the keywords consider or ignore indicating the operator. The list of messages follows the operand enclosed in a pair of braces (curly brackets) according to the following format:

`(‘ignore’ | ‘consider’) ‘{‘ <message-name> [‘,’ <message-name>]* ‘}’`

Note that ignore and consider can be combined with other types of operations in a single rectangle (as a shorthand for nested rectangles), such as assert consider {msgA, msgB}.

### Continuation

Continuations are shown with the same symbol as States, but they may cover more than one Lifeline.

Continuations may also appear on flowlines of Interaction Overview Diagrams.

A continuation that is alone in an InteractionFragment is considered to be at the end of the enclosing InteractionFragment.

### InteractionOperatorKind

The value of the InteractionOperandKind is given as text in a small compartment in the upper left corner of the CombinedFragment frame. There is specialized notation for some of the interactionOperator values, as defined below.

#### Strict interactionOperator

Notationally, this means that the vertical coordinate of the contained fragments is significant throughout the whole scope of the CombinedFragment and not only on one Lifeline. The vertical position of an OccurrenceSpecification is given by the vertical position of the corresponding point. The vertical position of other InteractionFragments is given by the topmost vertical position of its bounding rectangle.

#### Ignore / Consider interactionOperator

See 17.6.4 (ConsiderIgnoreFragment).

#### Loop interactionOperator

Textual syntax of the loop operand:

```
'loop[( ' $\langle$ minint $\rangle$  [ ','  $\langle$ maxint $\rangle$  ] )']
```

$\langle$ minint $\rangle$  ::= non-negative natural

$\langle$ maxint $\rangle$  ::= non-negative natural (greater than or equal to  $\langle$ minint $\rangle$  | '\*'

'\*' means infinity.

If only  $\langle$ minint $\rangle$  is present, this means that  $\langle$ minint $\rangle$  =  $\langle$ maxint $\rangle$  =  $\langle$ integer $\rangle$ .

If only loop, then this means a loop with infinity upper bound and with 0 as lower bound.

#### Parallel interactionOperator

A conforming tool may use the shorthand notation of a “coregion area” within a single Lifeline.

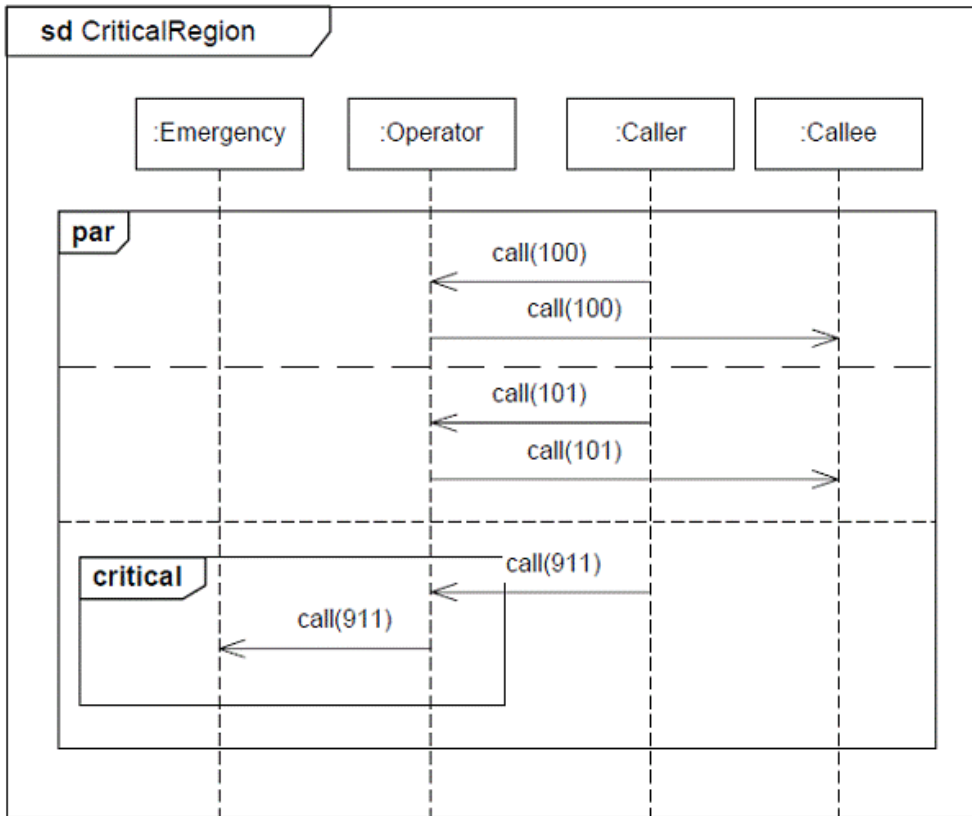
A “coregion” is a notational shorthand for parallel combined fragments, used for the common situation where the order of event occurrences (or other nested fragments) on one Lifeline is insignificant. This means that in a given “coregion” area of a Lifeline all the directly contained fragments are considered separate operands of a parallel combined fragment. See example in Figure 17.23.

## 17.6.5 Examples

See Figure 17.14 for examples of InteractionOperand.

See examples of InteractionConstraints in Figure 17.14 and Figure 17.27.

Examples of CombinedFragments with various interactionOperators are shown in Figure 17.12, Figure 17.13, and Figure 17.14.



**Figure 17.12 - Critical Region**

The example, Figure 17.12, shows that the handling of a 911-call must be contiguously handled. The operator must make sure to forward the 911-call before doing anything else. The normal calls, however, can be freely interleaved.

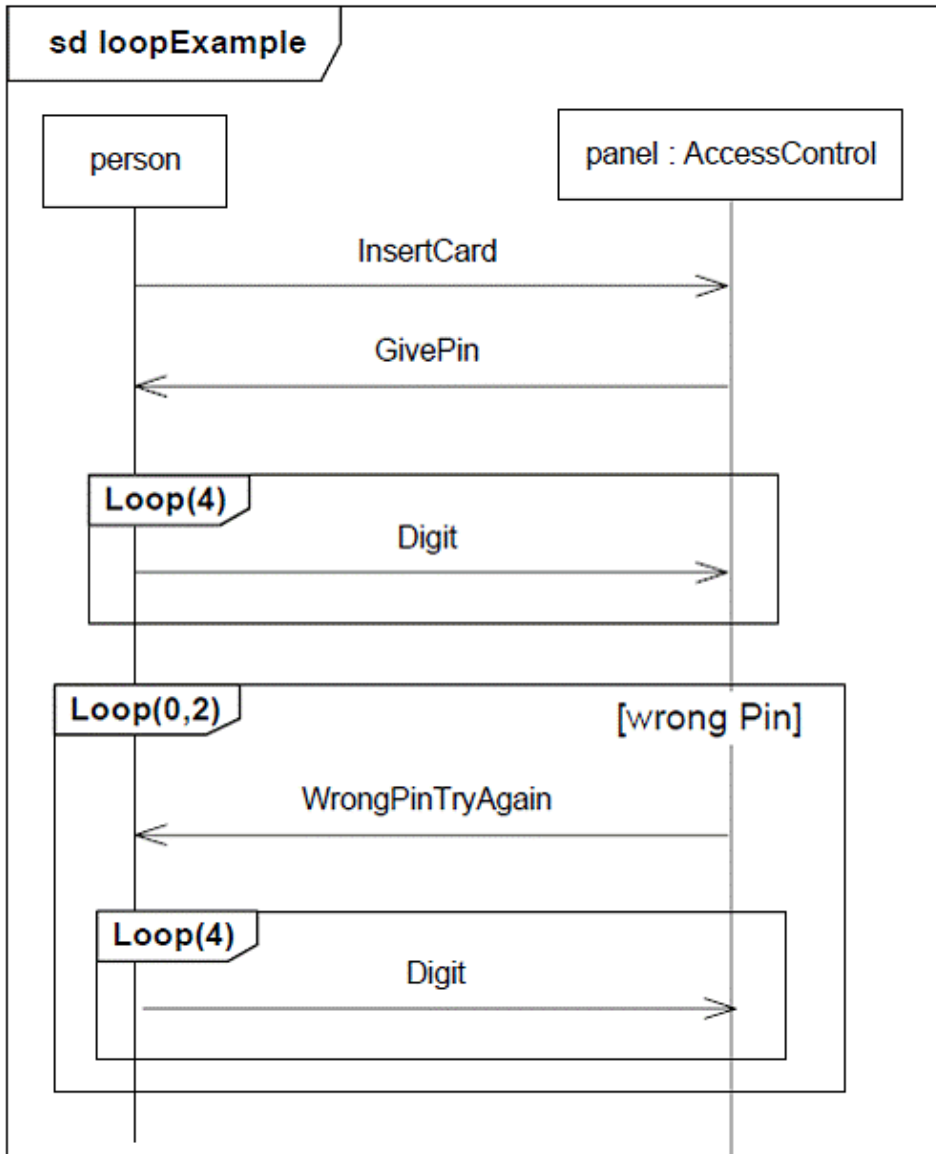
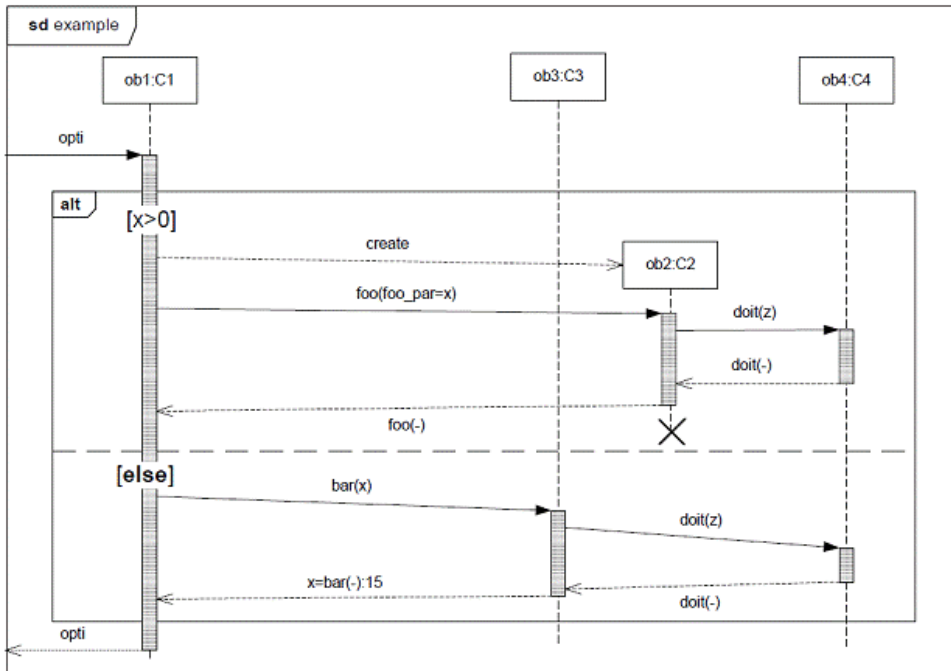


Figure 17.13 - Loop CombinedFragment



**Figure 17.14 - CombinedFragment**

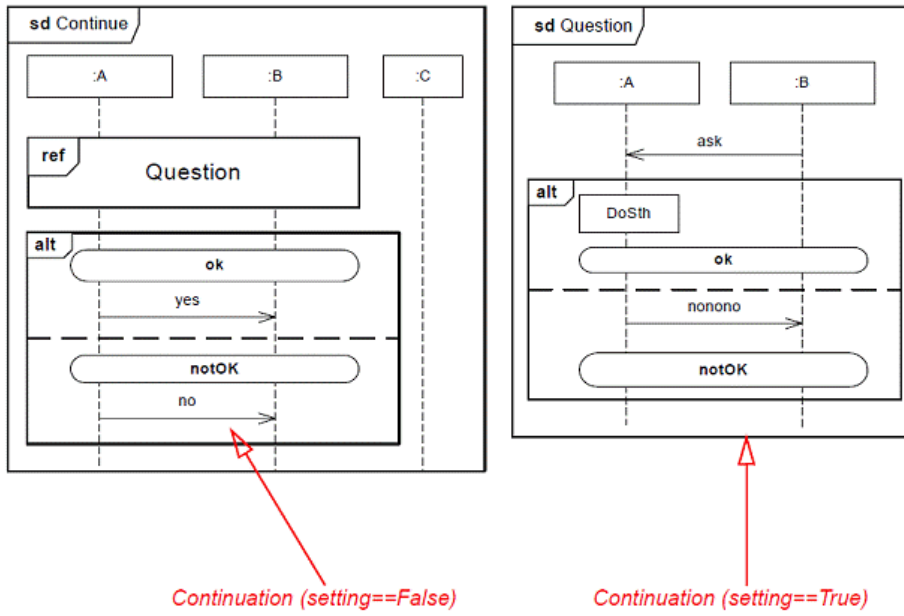
Figure 17.17 shows an example of ConsiderIgnoreFragments.

The following are other examples of operator notation for **consider** and **ignore** interactionOperators:

- consider {m, s}: showing that only m and s messages are considered significant.
- ignore {q,r}: showing that q and r messages are considered insignificant.
- Ignore and consider operations are typically combined with other operations such as “assert consider {m, s}.”

Figure 17.15 shows an example with a continuation.





**Figure 17.15 - Continuation**

The two diagrams in Figure 17.15 are together equivalent to the diagram in Figure 17.16.

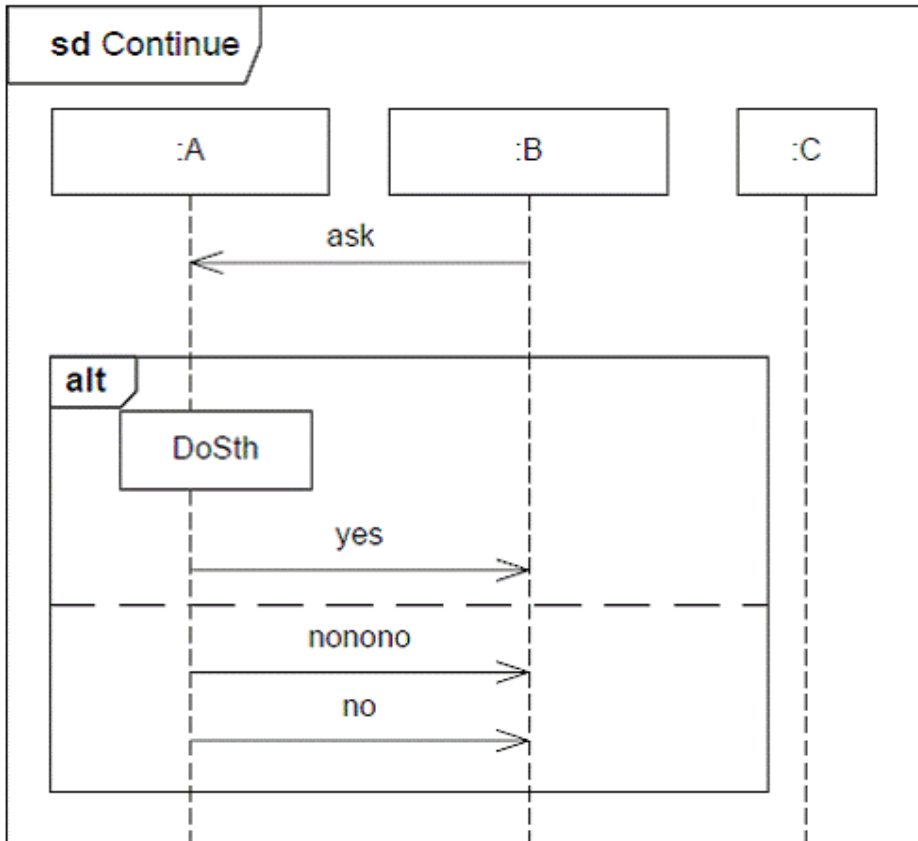


Figure 17.16 - Continuation interpretation

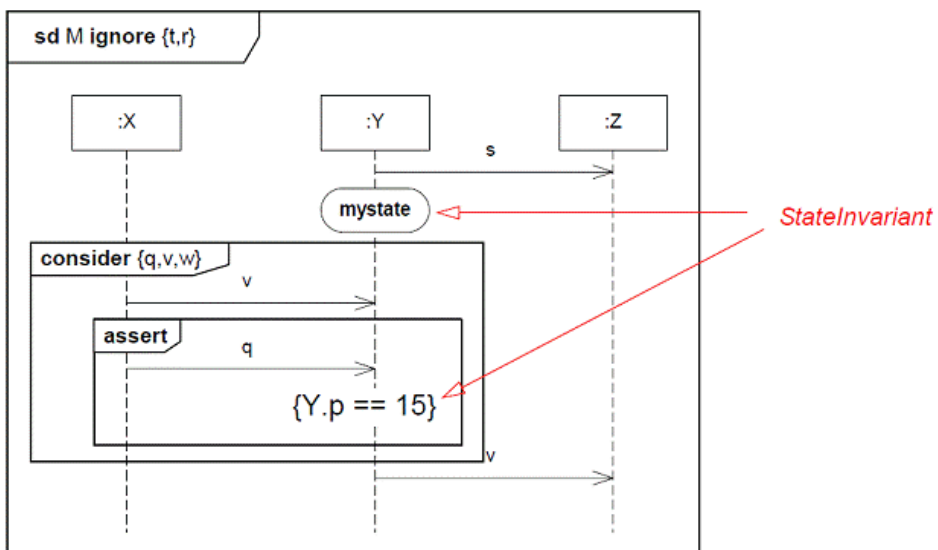


Figure 17.17 - Ignore, consider, assert with StateInvariants

In Figure 17.17 we have an Interaction M, which considers message types other than t and r. This means that if this Interaction is used to specify a test of an existing system and when running that system a t or an r occurs, these messages will be ignored by this specification. t and r will of course be handled in some manner by the running system, but how they are handled is irrelevant for our Interaction shown here.

The State invariant given as a state “mystate” will be evaluated at runtime directly prior to whatever event occurs on Y after “mystate.” This may be the reception of q as specified within the assert-fragment, or it may be an event that is specified to be insignificant by the filters.

The assert fragment is nested in a consider fragment to mean that we expect a q message to occur once a v has occurred here. Any occurrences of messages other than v, w, and q will be ignored in a test situation. Thus the appearance of a w message after the v is an invalid trace.

The state invariant given in curly brackets will be evaluated prior to the next event occurrence after that on Y.

## 17.7 Interaction Uses

### 17.7.1 Summary

The Interaction Uses subclause specifies the abstract syntax, semantics, and notation for the following metaclasses:

- InteractionUse
- PartDecomposition

### 17.7.2 Abstract Syntax

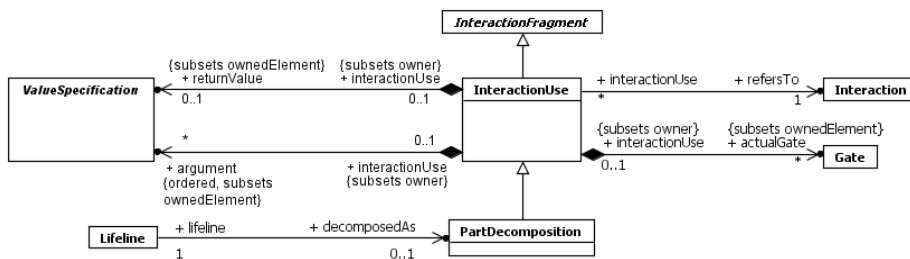


Figure 17.18 - InteractionUses

### 17.7.3 Semantics

#### Interaction Uses

The semantics of the InteractionUse is the set of traces of the semantics of the referred Interaction where the gates have been resolved as well as all generic parts having been bound such as the arguments substituting the parameters.

An actual Gate may be attached to the outer boundary of an InteractionUse to provide a link point to establish the concrete sender and receiver in the Interaction referred to by that InteractionUse.

## Part Decompositions

Decomposition of a lifeline within one Interaction by an Interaction (owned by the type of the Lifeline's associated ConnectableElement), is interpreted exactly as an InteractionUse. The messages that go into (or go out from) the decomposed lifeline are interpreted as actual gates that are matched by corresponding formal gates on the decomposition.

As the decomposed Lifeline is interpreted as an InteractionUse, the semantics of a PartDecomposition is the semantics of the Interaction referenced by the decomposition where the gates and parameters have been matched.

That a CombinedFragment is extra-global depicts that there is a CombinedFragment with the same operator covering the decomposed Lifeline in its Interaction. The full understanding of that (higher level) CombinedFragment must be acquired through combining the operands of the decompositions operand by operand.

### 17.7.4 Notation

#### InteractionUse

The InteractionUse is shown as a CombinedFragment symbol where the operator is called **ref**. The complete syntax of the name (situated in the InteractionUse area) is:

```
<name> ::= [ <attribute-name> '=' ] [ <collaboration-use> '.' ] <interaction-name>
          [ ( ' <io-argument> [ ' ' <io-argument> ] * ' ) ] [ ':' <return-value> ]
<io-argument> ::= <in-argument> | 'out' <out-argument>
```

The <attribute-name> refers to an attribute of one of the lifelines in the Interaction.

<collaboration-use> is an identification of a collaboration use that binds lifelines of a collaboration. The interaction name is in that case within that collaboration. See example of collaboration uses in Figure 17.24.

The io-arguments are most often arguments of IN-parameters. If there are OUT- or INOUT-parameters and the output value is to be described, this can be done following an out keyword.

The syntax of argument is explained in the notation sub clause of Messages 17.4.4 (Message).

If the InteractionUse returns a value, this may be described following a colon at the end of the clause.

#### PartDecomposition

PartDecomposition is designated by a referencing clause in the head of the Lifeline as can be seen in the notation sub clause 17.3.4 (Lifeline) (see also Figure 17.21).

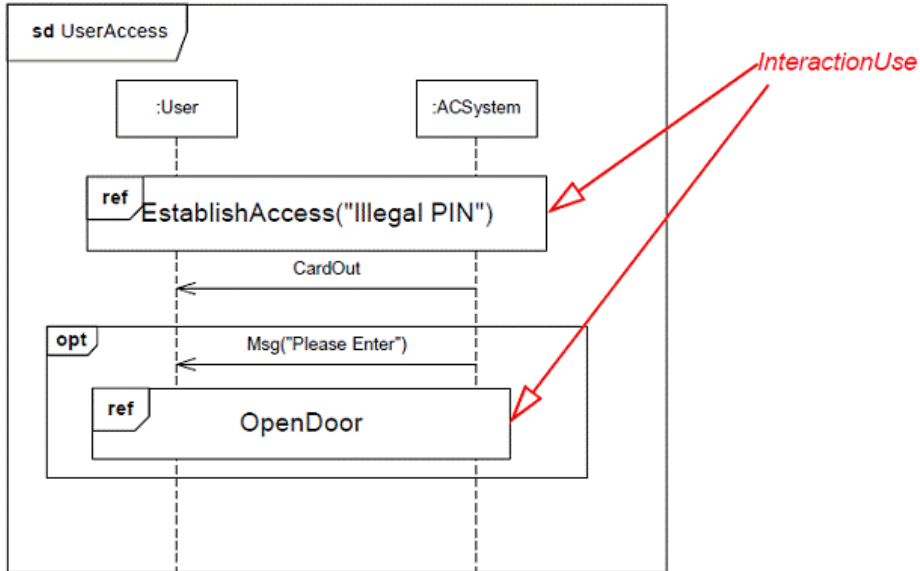
If the part decomposition is denoted inline under the decomposed lifeline and the decomposition clause is the keyword "strict," this indicates that the constructs on all sub lifelines within the inline decomposition are ordered in strict sequence (see 17.6.4 (Strict interactionOperator)).

Extra global CombinedFragments have their rectangular frame go outside the boundaries of the decomposition Interaction.

#### Style Guidelines

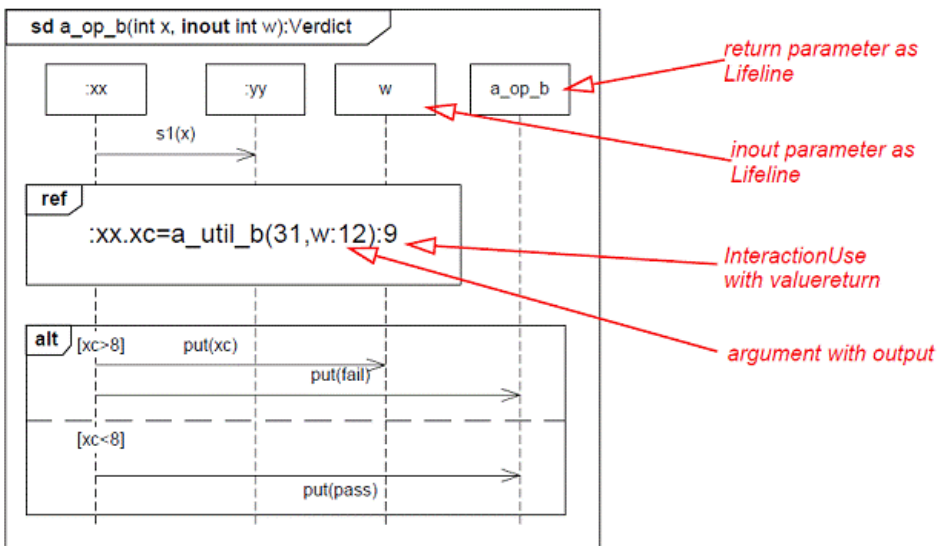
The name of an Interaction that is involved in decomposition would benefit from including in the name, the name of the type of the Part being decomposed and the name of the Interaction originating the decomposition. This is shown in Figure 17.21 where the decomposition is called AC\_UserAccess where 'AC' refers to ACSystem, which is the type of the Lifeline and UserAccess is the name of the Interaction where the decomposed lifeline is contained.

## 17.7.5 Examples



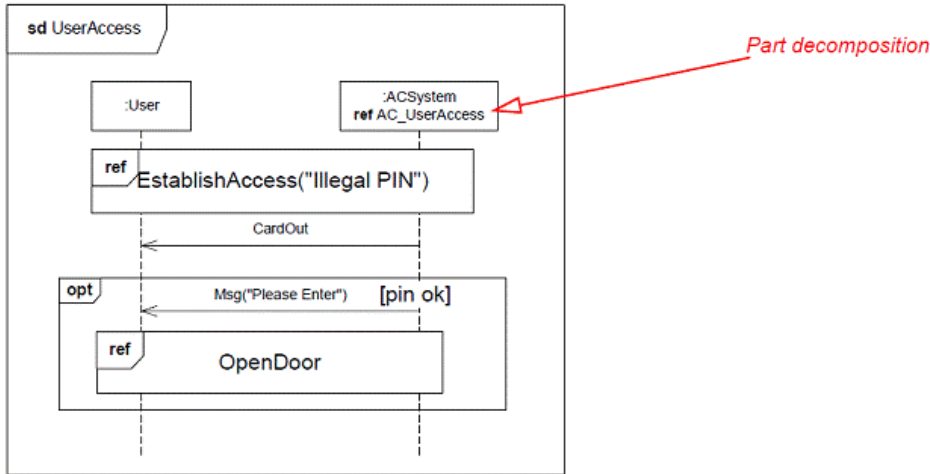
**Figure 17.19 - InteractionUse**

In Figure 17.19 we show an InteractionUse referring the Interaction EstablishAccess with (input) argument “Illegal PIN.” Within the optional CombinedFragment there is another InteractionUse without arguments referring OpenDoor.



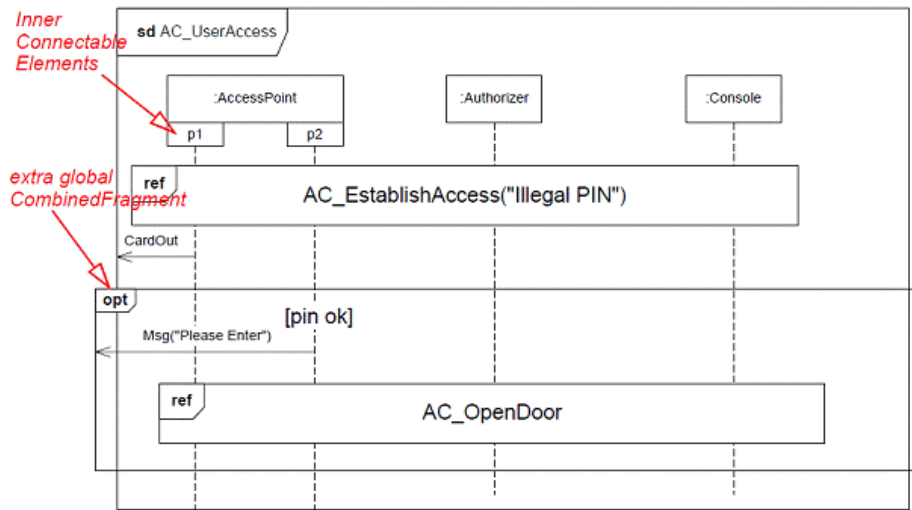
**Figure 17.20 - InteractionUse with value return**

In Figure 17.20 we have a more advanced Interaction that models a behavior returning a Verdict value. The return value from the Interaction is shown as a separate Lifeline a\_op\_b. Inside the Interaction there is an InteractionUse referring a\_util\_b with value return to the attribute xc of :xx with the value 9, and with inout parameter where the argument is w with returning out-value 12.



**Figure 17.21 - PartDecomposition - the decomposed part**

In Figure 17.21 we see how ACSystem within UserAccess is to be decomposed to AC\_UserAccess, which is an Interaction owned by class ACSystem.

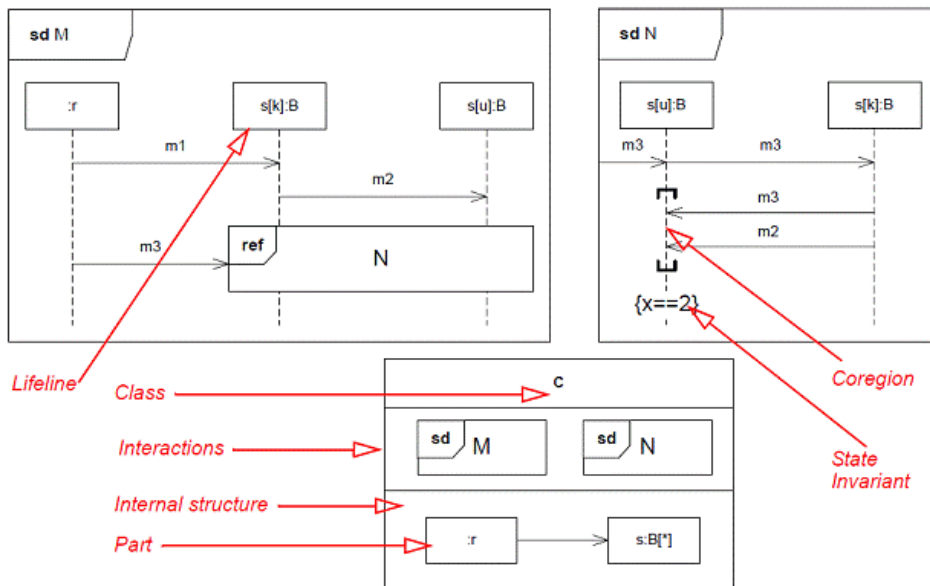


**Figure 17.22 - PartDecomposition - the decomposition**

In Figure 17.22 we see that AC\_UserAccess has global constructs that match the constructs of UserAccess covering ACSystem.

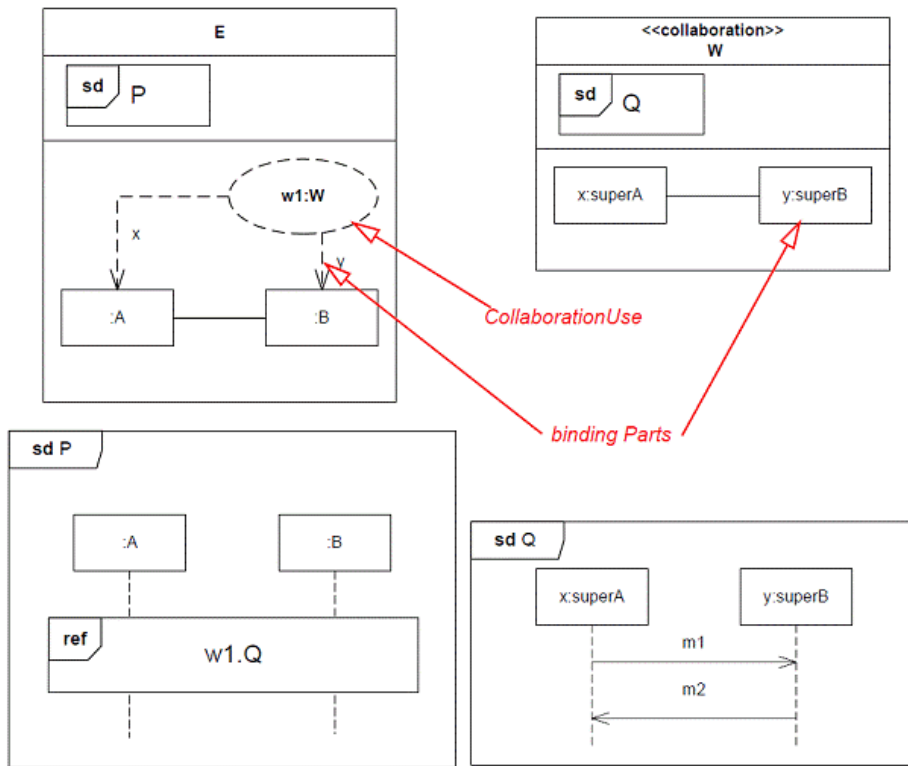
In particular we notice the “extra global interaction group” that goes beyond the frame of the Interaction. This construct corresponds to a CombinedFragment of UserAccess. However, we want to indicate that the operands of extra global interaction groups are combined one-to-one with similar extra global interaction groups of other decompositions of the same original CombinedFragment.

As a notational shorthand, decompositions can also be shown “inline.” In Figure 17.22 we see that the inner ConnectableElements of :AccessPoint (p1 and p2) are represented by Lifelines already on this level.



**Figure 17.23 - Sequence Diagrams where two Lifelines refer to the same set of Parts (and Internal Structure)**

The sequence diagrams shown in Figure 17.23 show a scenario where `r` sends `m1` to `s[k]` (which is of type `B`), and `s[k]` sends `m2` to `s[u]`. In the meantime independent of `s[k]` and `s[u]`, `r` may have sent `m3` towards the InteractionUse `N` through a gate. Following the `m3` message into `N` we see that `s[u]` then sends another `m3` message to `s[k]`. `s[k]` then sends `m3` and then `m2` towards `s[u]`. `s[u]` receives the two latter messages in any order (coregion). Having received these messages, we state an invariant on a variable `x` (most certainly owned by `s[u]`).



**Figure 17.24 - Describing Collaborations and their binding**

The example in Figure 17.24 shows how collaboration uses are employed to make Interactions of a Collaboration available in another classifier.

The collaboration W has two parts x and y that are of types (classes) superA and superB respectively. Classes A and B are specializations of superA and superB respectively. The Sequence Diagram Q shows a simple Interaction that we will reuse in another environment. The class E represents this other environment. There are two anonymous parts :A and :B and the CollaborationUse w1 of Collaboration W binds x and y to :A and :B respectively. This binding is legal as :A and :B are parts of types that are specializations of the types of x and y.

In the Sequence Diagram P (owned by class E) we use the Interaction Q made available via the CollaborationUse w1.

## 17.8 Sequence Diagrams

The most common kind of Interaction Diagram is the Sequence Diagram, which focuses on the Message interchange between a number of Lifelines.

A sequence diagram describes an Interaction by focusing on the sequence of Messages that are exchanged, along with their corresponding OccurrenceSpecifications on the Lifelines.

Interactions that are described by Sequence Diagrams form a basis for understanding the semantics of the meta classes in the Interactions package. Sequence Diagrams are used for the examples in subclauses for the Interaction sub packages.

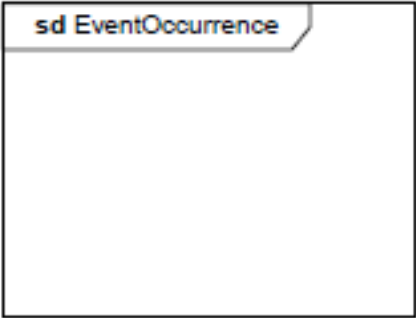

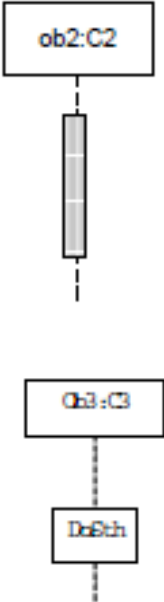


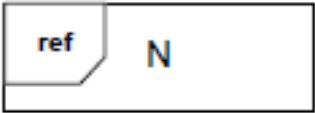
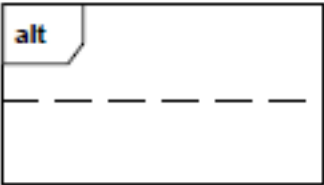
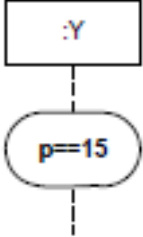
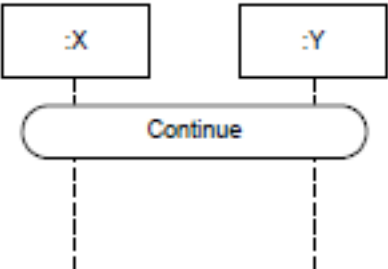
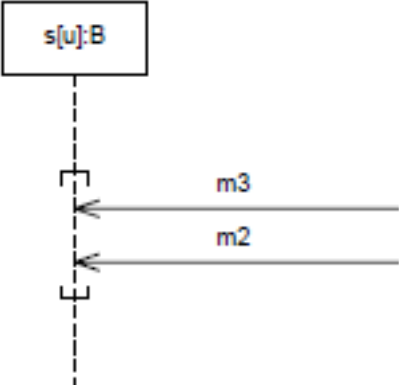

## 17.8.1 Sequence Diagram Notation

### Graphic Nodes

The graphic nodes that can be included in sequence diagrams are shown in Table 17.1.

**Table 17.1 Graphic Nodes Included in Sequence Diagrams**

Node Type	Notation	Reference
Frame (for Interaction)		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See 17.2.4 (Interaction)
Lifeline		See 17.3.4 (Lifeline)
ExecutionSpecification		See 17.2.4 (ExecutionSpecification)

InteractionUse		See 17.7.4 (InteractionUse).
CombinedFragment		See 17.6.4 (CombinedFragment)
StateInvariant		See 17.2.4 (StateInvariant)
Continuations		See 17.6.4 (Continuation)
Coregion		See 17.6.4 (Parallel interactionOperator)
DestructionOccurrenceSpecification		See 17.4.4 (DestructionOccurrenceSpecification) and example in Figure 17.14.

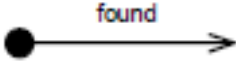

<p>DurationConstraint Duration Observation</p>		<p>See Figure 17.5.</p>
<p>TimeConstraint TimeObservation</p>		<p>See Figure 17.5.</p>

### Graphic Paths

The graphic paths between the graphic nodes are given in Table 17.2.

Table 17.2 - Graphic Paths Included in Sequence Diagrams

<p>Message</p>		<p>Messages come in different variants depending on what kind of Message they convey. Here we show an asynchronous message, a call and a reply. These are all <i>complete</i> messages. See 17.4.4 (Message)</p>
<p>LostMessage</p>		<p>Lost messages are messages with known sender, but the reception of the message does not happen. See 17.4.4 (Message)</p>

FoundMessage		Found messages are messages with known receiver, but the sending of the message is not described within the specification. See 17.4.4 (Message)
GeneralOrdering		See 17.5.4 (GeneralOrdering)

Interactions are units of behavior of an enclosing Classifier. Interactions focus on the passing of information with Messages between the ConnectableElements of the Classifier.

### 17.8.2 Example Sequence Diagram

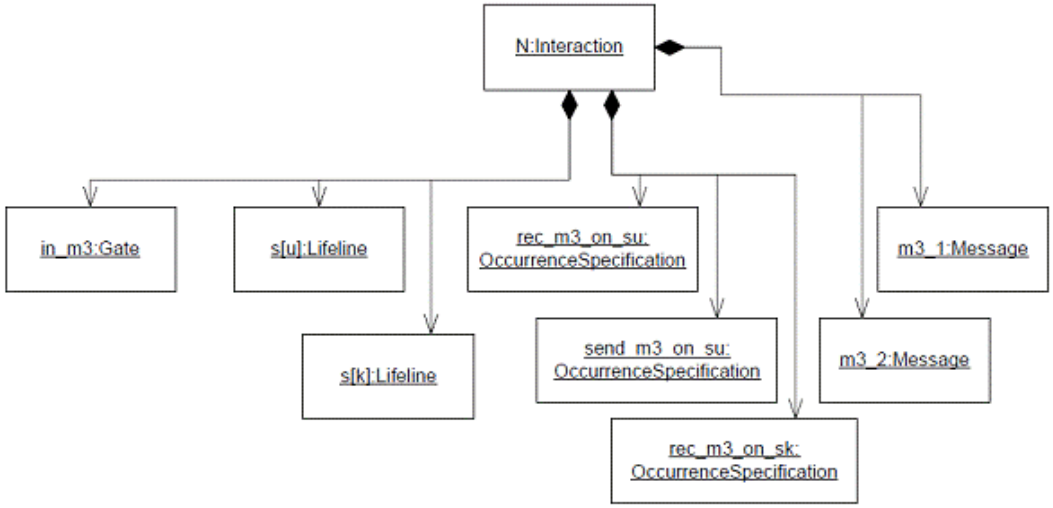
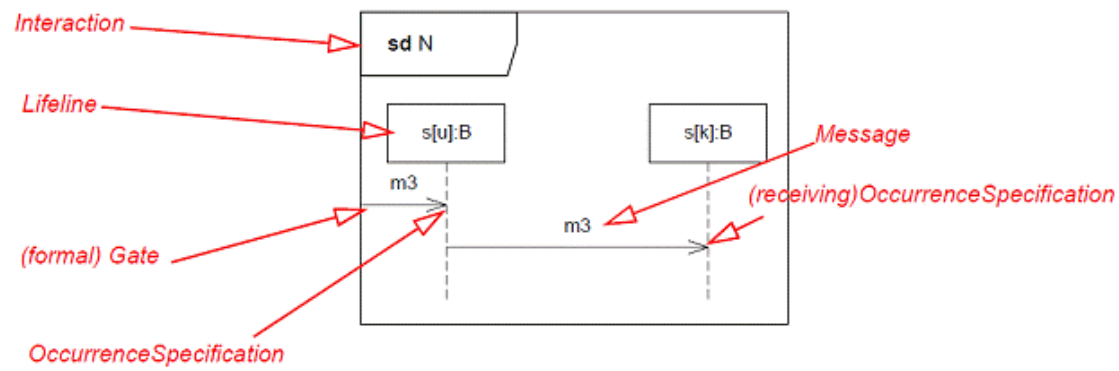


Figure 17.25 - Overview of Metamodel elements of a Sequence Diagram

In order to explain the mapping of the notation onto the metamodel we have pointed out areas and their corresponding metamodel concept in Figure 17.25. Let us go through the simple diagram and explain how the metamodel is built up. The whole diagram is

an Interaction (named N). There is a formal gate (with implicit name in\_m3) and two Lifelines (named s[u] and s[k] ) that are contained in the Interaction. Furthermore the two Messages (occurrences) both of the same type m3, implicitly named m3\_1 and m3\_2 here, are also owned by the Interaction. Finally there are the three OccurrenceSpecifications.

We have omitted in this metamodel the objects that are more peripheral to the Interaction model, such as the Part s and the class B and the connector referred by the Message.

## 17.9 Communication Diagrams

Communication Diagrams focus 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 numbering scheme.

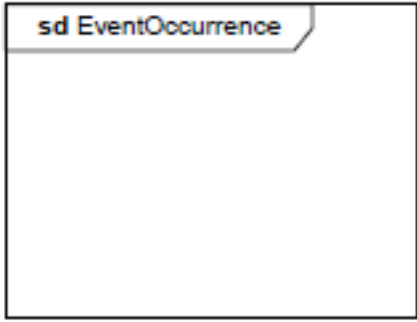
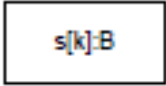
Communication Diagrams correspond to simple Sequence Diagrams that use none of the structuring mechanisms such as InteractionUses and CombinedFragments. It is also assumed that message overtaking (i.e., the order of the receptions are different from the order of sending of a given set of messages) will not take place or is irrelevant.

### 17.9.1 Communication Diagram Notation

#### Graphic Paths

Communication diagram nodes are shown in Table 17.3.

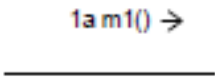
**Table 17.3 Graphic Nodes Included in Communication Diagrams**

Node Type	Notation	Reference
Frame (for Interaction)		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See 17.2.4 (Interaction)
Lifeline		See 17.3.4 (Lifeline)

#### Graphic Paths

Graphic paths of communication diagrams are given in Table 17.4

**Table 17.4 Graphic Paths Included in Communications Diagrams**

Message		See 17.4.4 (Message) and 17.9.1 (Sequence expression). The arrow
---------	---	--

		shown here indicates the communication direction.
--	--	---

### Sequence expression

The sequence-expression is a dot-separated list of sequence-terms followed by a colon (':').

sequence-term ‘.’ . . . ‘:’

Each term represents a level of procedural nesting within the overall interaction. If all the control is concurrent, then nesting does not occur. Each sequence-term has the following syntax:

[ integer | name ] [ recurrence ]

The integer represents the sequential order of the Message within the next higher level of procedural calling. Messages that differ in one integer term are sequentially related at that level of nesting. Example: Message 3.1.4 follows Message 3.1.3 within activation 3.1. The name represents a concurrent thread of control. Messages that differ in the final name are concurrent at that level of nesting. Example: Message 3.1a and Message 3.1b are concurrent within activation 3.1. All threads of control are equal within the nesting depth.

The recurrence represents conditional or iterative execution. This represents zero or more Messages that are executed depending on the conditions involved. The choices are:

‘\*’ ‘[’ iteration-clause ‘]’ an iteration

‘[’ guard ‘]’ a branch

An iteration represents a sequence of Messages at the given nesting depth. The iteration clause may be omitted (in which case the iteration conditions are unspecified). The iteration-clause is meant to be expressed in pseudocode or an actual programming language, UML does not prescribe its format. An example would be: \*[i := 1..n].

A guard represents a Message whose execution is contingent on the truth of the condition clause. The guard is meant to be expressed in pseudocode or an actual programming language; UML does not prescribe its format. An example would be: [x > y].

Note that a branch is notated the same as an iteration without a star. One might think of it as an iteration restricted to a single occurrence.

The iteration notation assumes that the Messages in the iteration will be executed sequentially. There is also the possibility of executing them concurrently. The notation for this is to follow the star by a double vertical line (for parallelism): \*||.

Note that in a nested control structure, the recurrence is not repeated at inner levels. Each level of structure specifies its own iteration within the enclosing context.

## 17.9.2 Example Communication Diagram

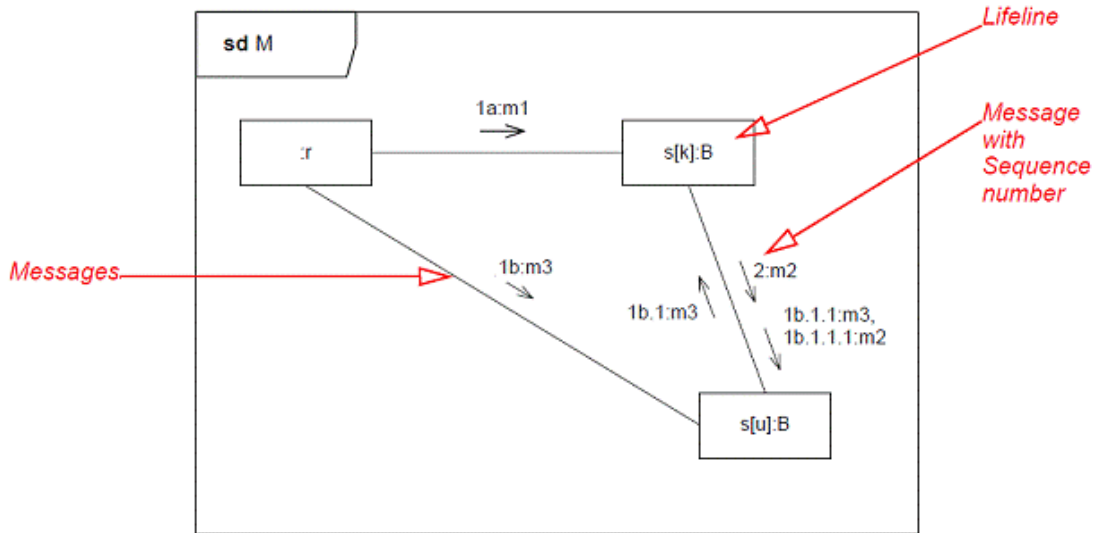


Figure 17.26 - Communication diagram

The Interaction described by a Communication Diagram in Figure 17.26 shows messages m1 and m3 being sent concurrently from :r towards two instances of the part s. The sequence numbers show how the other messages are sequenced. 1b.1 follows after 1b and 1b.1.1 thereafter etc. 2 follows after 1a and 1b.

## 17.10 Interaction Overview Diagrams

Interaction Overview Diagrams define Interactions through a variant of Activity Diagrams (described in Clause 12) in a way that promotes overview of the control flow.

Interaction Overview Diagrams focus on the overview of the flow of control where the nodes are Interactions or InteractionUses. The Lifelines and the Messages do not appear at this overview level.

### 17.10.1 Interaction Overview Diagram Notation

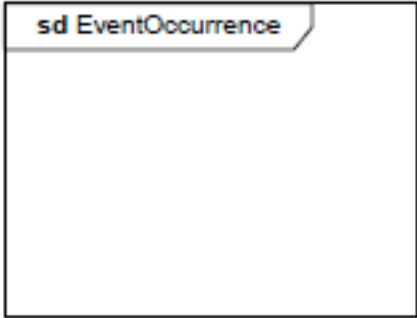
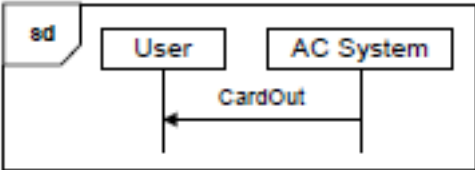
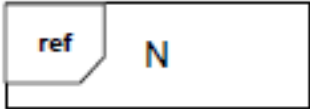
#### Graphic Nodes

Interaction Overview Diagrams are specialization of Activity Diagrams that represent Interactions. Interaction Overview Diagrams differ from Activity Diagrams in some respects.

- In place of ObjectNodes of Activity Diagrams, Interaction Overview Diagrams can only have either (inline) Interactions or InteractionUses. Inline Interaction diagrams and InteractionUses are considered special forms of CallBehaviorAction.
- Alternative Combined Fragments are represented by a Decision Node and a corresponding Merge Node.
- Parallel Combined Fragments are represented by a Fork Node and a corresponding Join Node.
- Loop Combined Fragments are represented by simple cycles.
- Branching and joining of branches must in Interaction Overview Diagrams be properly nested. This is more restrictive than in Activity Diagrams.

- Interaction Overview Diagrams are framed by the same kind of frame that encloses other forms of Interaction Diagrams. The heading text may also include a list of the contained Lifelines (that do not appear graphically).

**Table 17.5 Graphic nodes included in Interaction Overview Diagrams in addition to those borrowed from Activity Diagrams**

Node Type	Notation	Reference
Frame (for Interaction)		<p>The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See 17.2.4 (Interaction)</p>
Interaction		<p>An Interaction diagram of any kind may appear inline as an ActivityInvocation. See 17.2.4 (Interaction). The inline Interaction diagrams may be either anonymous (as here) or named.</p>
InteractionUse		<p>ActivityInvocation in the form of InteractionUse. See 17.7.4 (InteractionUse). The tools may choose to “explode” the view of an InteractionUse into an inline Interaction with the name of the Interaction referred by the occurrence. The inline Interaction will then replace the occurrence by a replica of the definition Interaction where arguments have replaced parameters.</p>

Interaction Overview Diagrams use Activity diagram notation where the nodes are either Interactions or InteractionUses. Interaction Overview Diagrams are a way to describe Interactions where Messages and Lifelines are abstracted away. In the purest form all Activities are InteractionUses and then there are no Messages or Lifelines shown in the diagram at all.



## 17.10.2 Examples of Interaction Overview Diagrams

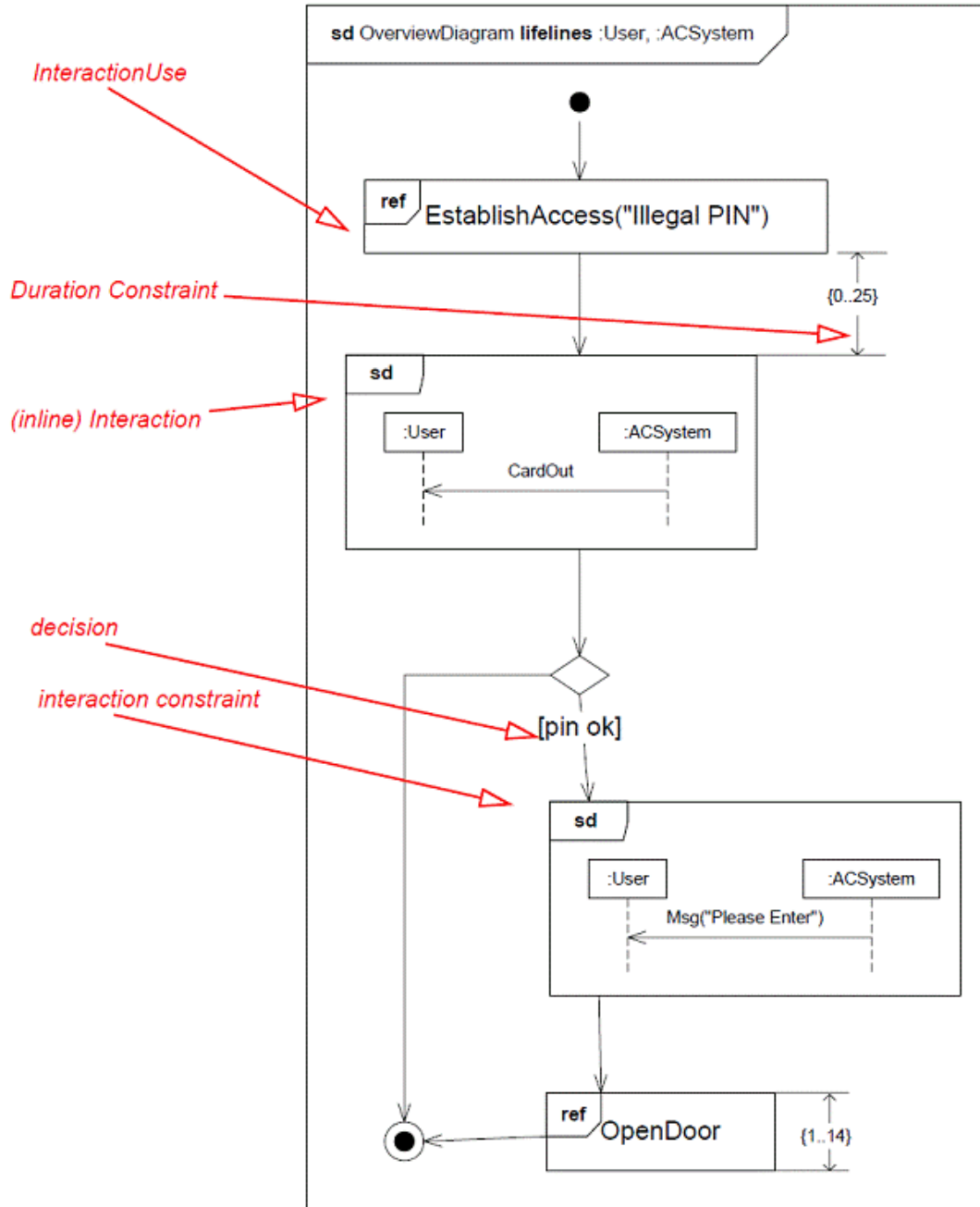


Figure 17.27 - Interaction Overview Diagram representing a High Level Interaction diagram

Figure 17.27 is another way to describe the behavior shown in Figure 17.19, with some added timing constraints. The Interaction EstablishAccess occurs first (with argument “Illegal PIN”) followed by weak sequencing with the message CardOut which is shown in an inline Interaction. Then there is an alternative as we find a decision node with an InteractionConstraint on one of the branches. Along that control flow we find another inline Interaction and an InteractionUse in (weak) sequence.

## 17.11 Timing Diagrams

Timing diagrams focus on conditions changing within and among Lifelines along a linear time axis.

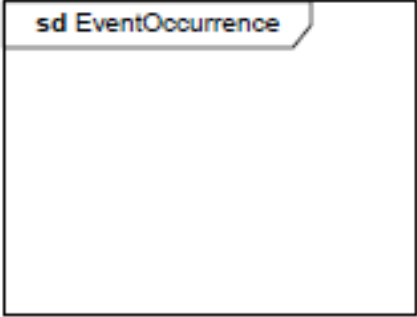

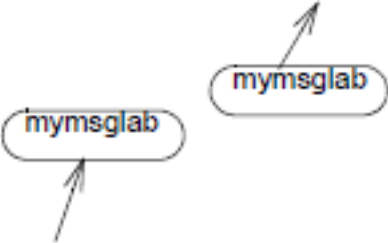
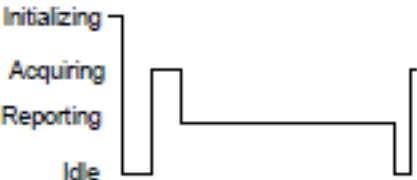
Timing diagrams describe behavior of both individual classifiers and interactions of classifiers, focusing attention on time of occurrence of events causing changes in the modeled conditions of the Lifelines.

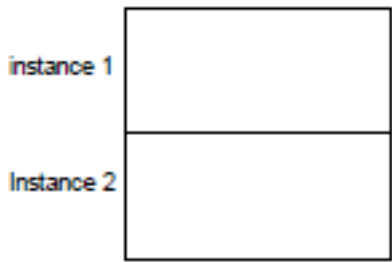


### 17.11.1 Timing Diagram Notation

#### Graphic Nodes and Paths

Timing Diagram graphic nodes and paths are shown in Table 17.6.

**Table 17.6 Graphic nodes and paths included in timing diagrams**

Node Type	Notation	Reference
Frame (for Interaction)		<p>The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See 17.2.4 (Interaction)</p>
Message		<p>Messages come in different variants depending on what kind of Message they convey. Here we show an asynchronous message, a call and a reply. See 17.4.4 (Message)</p>
MessageLabel		<p>Labels are only notational shorthands used to prevent cluttering of the diagrams with a number of messages crisscrossing the diagram between Lifelines that are far apart. The labels denote that a Message may be disrupted by introducing labels with the same name.</p>
State or condition timeline		<p>This is the state of the classifier or attribute, or some testable condition, such as a discrete enumerable value. See also 17.2.4 (StateInvariant). It is also permissible to let the state-dimension be continuous as well as discrete. This is</p>

		illustrative for scenarios where certain entities undergo continuous state changes, such as temperature or density.
General value lifeline		Shows the value of the connectable element as a function of time. Value is explicitly denoted as text. Crossing reflects the event where the value changed.
Lifeline		See 17.3.4 (Lifeline)
GeneralOrdering		17.5.4 (GeneralOrdering).
DestructionOccurrenceSpecification		See 17.4.4 (DestructionOccurrenceSpecification)

### 17.11.2 Examples of Timing Diagrams

Timing diagrams show change in state or other condition of a structural element over time. There are a few forms in use. We shall give examples of the simplest forms.

Sequence Diagrams as the primary form of Interactions may also depict time observation and timing constraints. We show in Figure 17.5 an example in Sequence Diagram that we will also give in Timing Diagrams.

The :User of the Sequence Diagram in Figure 17.5 is depicted with a simple Timing Diagram in Figure 17.28.

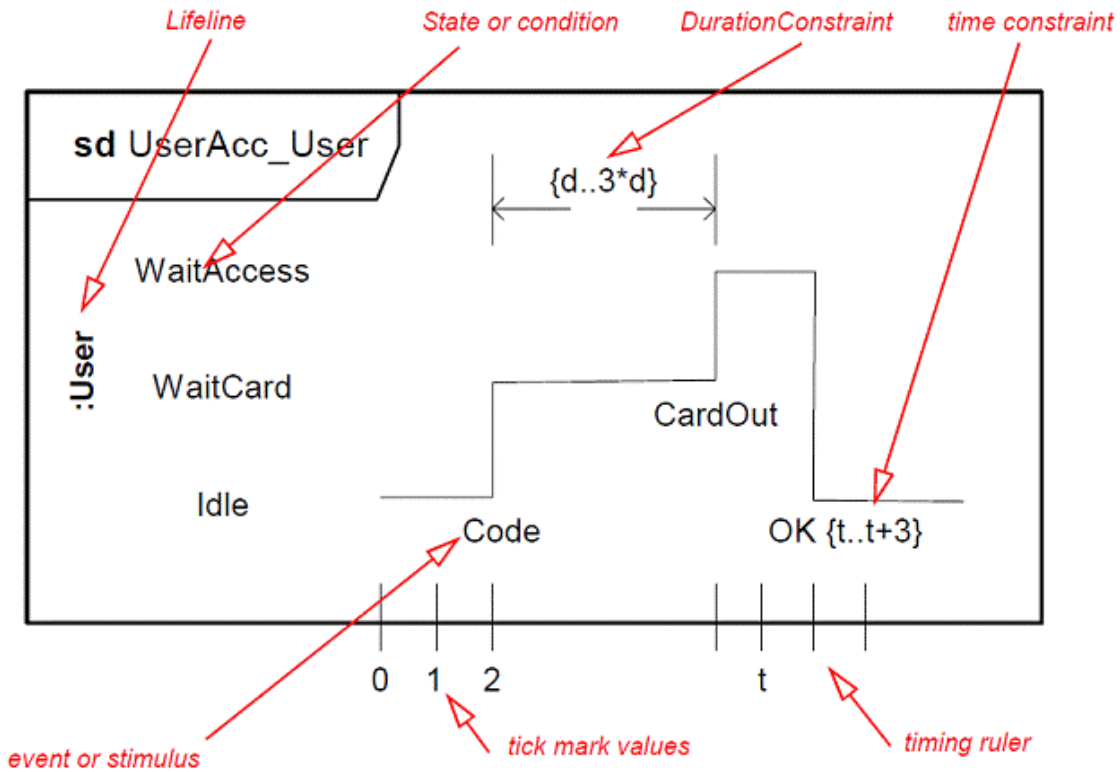


Figure 17.28 - A Lifeline for a discrete object

The primary purpose of the timing diagram is to show 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. The received events are annotated as shown when it is desirable to show the event causing the change in condition or state.

Sometimes it is more economical and compact to show the state or condition on the vertical Lifeline as shown in Figure 17.29.

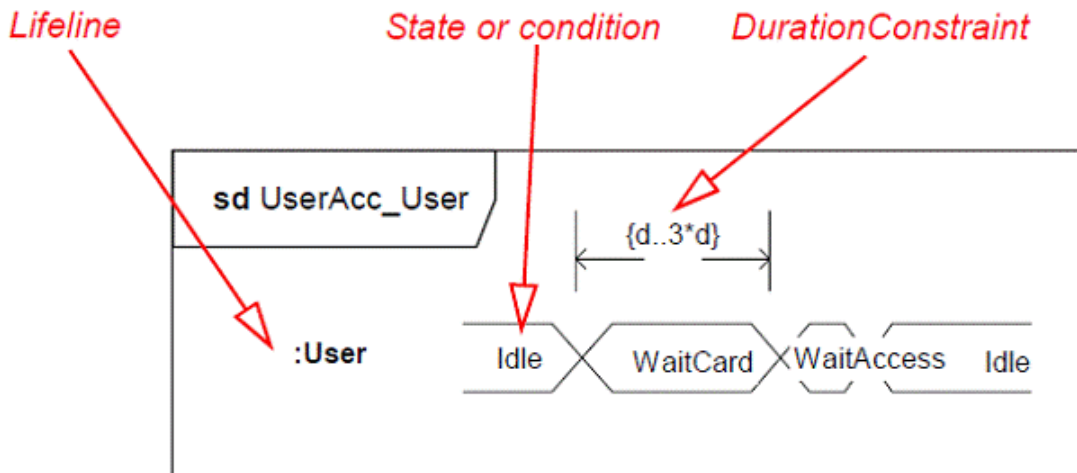


Figure 17.29 - Compact Lifeline with States

Finally we may have an elaborate form of TimingDiagrams where more than one Lifeline is shown and where the messages are also depicted. We show such a Timing Diagram in Figure 17.30 corresponding to the Sequence Diagram in Figure 17.5.

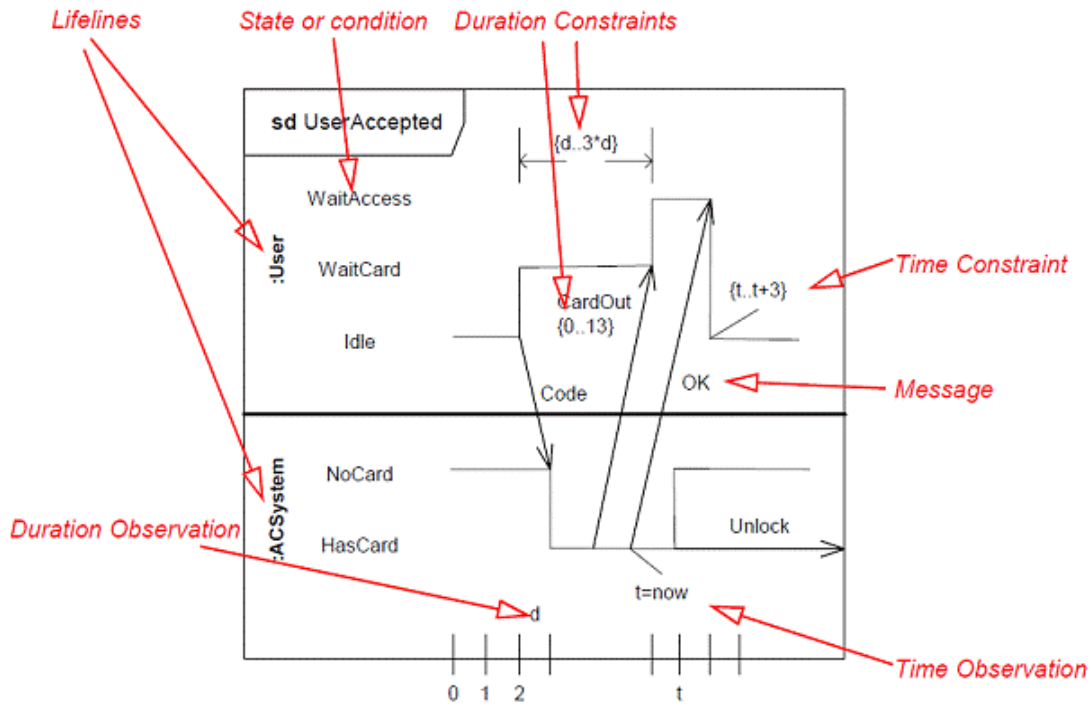


Figure 17.30 - Timing Diagram with more than one Lifeline and with Messages

## 17.12 Classifier Descriptions

### ActionExecutionSpecification [Class]

#### Description

An ActionExecutionSpecification is a kind of ExecutionSpecification representing the execution of an Action.

#### Diagrams

[Occurrences](#)

#### Generalizations

[ExecutionSpecification](#)

#### Association Ends

- action : [Action](#) [1..1] (opposite [A\\_action\\_actionExecutionSpecification::actionExecutionSpecification](#))  
Action whose execution is occurring.

#### Constraints

- action\_referenced  
The Action referenced by the ActionExecutionSpecification must be owned by the Interaction owning that ActionExecutionSpecification.

```
inv: (enclosingInteraction->notEmpty() or enclosingOperand.combinedFragment->notEmpty()) and  
let parentInteraction : Set(Interaction) = enclosingInteraction.oclAsType(Interaction)->asSet()-  
>union(  
enclosingOperand.combinedFragment->closure(enclosingOperand.combinedFragment)->  
collect(enclosingInteraction).oclAsType(Interaction)->asSet()) in  
(parentInteraction->size() = 1) and self.action.interaction->asSet() = parentInteraction
```

### BehaviorExecutionSpecification [Class]

#### Description

A BehaviorExecutionSpecification is a kind of ExecutionSpecification representing the execution of a Behavior.

#### Diagrams

[Occurrences](#)

#### Generalizations

[ExecutionSpecification](#)

## Association Ends

- behavior : [Behavior](#) [0..1] (opposite [A\\_behavior\\_behaviorExecutionSpecification::behaviorExecutionSpecification](#))  
Behavior whose execution is occurring.

## CombinedFragment [Class]

### Description

A CombinedFragment defines an expression of InteractionFragments. A CombinedFragment is defined by an interaction operator and corresponding InteractionOperands. Through the use of CombinedFragments the user will be able to describe a number of traces in a compact and concise manner.

### Diagrams

[Fragments](#)

### Generalizations

[InteractionFragment](#)

### Specializations

[ConsiderIgnoreFragment](#)

### Attributes

- interactionOperator : [InteractionOperatorKind](#) [1..1] = seq  
Specifies the operation which defines the semantics of this combination of InteractionFragments.

## Association Ends

- ♦ cfragmentGate : [Gate](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [A\\_cfragmentGate\\_combinedFragment::combinedFragment](#))  
Specifies the gates that form the interface between this CombinedFragment and its surroundings
- ♦ operand : [InteractionOperand](#) [1..\*]{ordered, subsets [Element::ownedElement](#)} (opposite [A\\_operand\\_combinedFragment::combinedFragment](#))  
The set of operands of the combined fragment.

## Constraints

- break  
If the interactionOperator is break, the corresponding InteractionOperand must cover all Lifelines covered by the enclosing InteractionFragment.

```
inv: interactionOperator=InteractionOperatorKind::break implies  
enclosingInteraction.ooclAsType(InteractionFragment)->asSet()->union(  
    enclosingOperand.ooclAsType(InteractionFragment)->asSet()).covered->asSet() = self.covered->asSet()
```

- **consider\_and\_ignore**  
The interaction operators 'consider' and 'ignore' can only be used for the ConsiderIgnoreFragment subtype of CombinedFragment

```
inv: ((interactionOperator = InteractionOperatorKind::consider) or (interactionOperator = InteractionOperatorKind::ignore)) implies oclIsKindOf(ConsiderIgnoreFragment)
```

- **opt\_loop\_break\_neg**  
If the interactionOperator is opt, loop, break, assert or neg, there must be exactly one operand.

```
inv: (interactionOperator = InteractionOperatorKind::opt or interactionOperator = InteractionOperatorKind::loop or interactionOperator = InteractionOperatorKind::break or interactionOperator = InteractionOperatorKind::assert or interactionOperator = InteractionOperatorKind::neg) implies operand->size()=1
```

## ConsiderIgnoreFragment [Class]

### Description

A ConsiderIgnoreFragment is a kind of CombinedFragment that is used for the consider and ignore cases, which require lists of pertinent Messages to be specified.

### Diagrams

[Fragments](#)

### Generalizations

[CombinedFragment](#)

### Association Ends

- message : [NamedElement](#) [0..\*] (opposite [A message considerIgnoreFragment::considerIgnoreFragment](#))  
The set of messages that apply to this fragment.

### Constraints

- **consider\_or\_ignore**  
The interaction operator of a ConsiderIgnoreFragment must be either 'consider' or 'ignore'.

```
inv: (interactionOperator = InteractionOperatorKind::consider) or (interactionOperator = InteractionOperatorKind::ignore)
```

- **type**  
The NamedElements must be of a type of element that can be a signature for a message (i.e., an Operation, or a Signal).

```
inv: message->forAll(m | m.ocIsKindOf(Operation) or m.ocIsKindOf(Signal))
```



## Continuation [Class]

### Description

A Continuation is a syntactic way to define continuations of different branches of an alternative CombinedFragment. Continuations are intuitively similar to labels representing intermediate points in a flow of control.

### Diagrams

[Fragments](#)

### Generalizations

[InteractionFragment](#)

### Attributes

- setting : [Boolean](#) [1..1] = true  
True: when the Continuation is at the end of the enclosing InteractionFragment and False when it is in the beginning.

### Constraints

- first\_or\_last\_interaction\_fragment  
Continuations always occur as the very first InteractionFragment or the very last InteractionFragment of the enclosing InteractionOperand.

```
inv: enclosingOperand->notEmpty() and
  let peerFragments : OrderedSet(InteractionFragment) = enclosingOperand.fragment in
  (peerFragments->notEmpty() and
   ((peerFragments->first() = self) or (peerFragments->last() = self)))
```

- same\_name  
Across all Interaction instances having the same context value, every Lifeline instance covered by a Continuation (self) must be common with one covered Lifeline instance of all other Continuation instances with the same name as self, and every Lifeline instance covered by a Continuation instance with the same name as self must be common with one covered Lifeline instance of self. Lifeline instances are common if they have the same selector and represents associationEnd values.

```
inv: enclosingOperand.combinedFragment->notEmpty() and
let parentInteraction : Set(Interaction) =
enclosingOperand.combinedFragment->closure(enclosingOperand.combinedFragment)->
collect(enclosingInteraction).oclAsType(Interaction)->asSet()
in
(parentInteraction->size() = 1)
and let peerInteractions : Set(Interaction) =
  (parentInteraction->union(parentInteraction->collect('_context')->collect(behavior)->
  select(oclIsKindOf(Interaction)).oclAsType(Interaction)->asSet()->asSet()) in
  (peerInteractions->notEmpty()) and
  let combinedFragments1 : Set(CombinedFragment) = peerInteractions.fragment->
  select(oclIsKindOf(CombinedFragment)).oclAsType(CombinedFragment)->asSet() in
  combinedFragments1->notEmpty() and combinedFragments1->closure(operand.fragment->
  select(oclIsKindOf(CombinedFragment)).oclAsType(CombinedFragment))->asSet().operand.fragment->
  select(oclIsKindOf(Continuation)).oclAsType(Continuation)->asSet()->
  forAll(c : Continuation | (c.name = self.name) implies
  (c.covered->asSet()->forAll(cl : Lifeline | -- cl must be common to one lifeline covered by self
```

```

self.covered->asSet()->
select(represents = cl.represents and selector = cl.selector)->asSet()->size()==1))
and
(self.covered->asSet()->forall(cl : Lifeline | -- cl must be common to one lifeline covered by c
c.covered->asSet()->
select(represents = cl.represents and selector = cl.selector)->asSet()->size()==1))
)

```

- **global**  
Continuations are always global in the enclosing InteractionFragment e.g., it always covers all Lifelines covered by the enclosing InteractionOperator.

```

inv: enclosingOperand->notEmpty() and
let operandLifelines : Set(Lifeline) = enclosingOperand.covered in
(operandLifelines->notEmpty() and
operandLifelines->forall(ol :Lifeline |self.covered->includes(ol)))

```

## **DestructionOccurrenceSpecification [Class]**

### **Description**

A DestructionOccurrenceSpecification models the destruction of an object.

### **Diagrams**

[Messages](#)

### **Generalizations**

[MessageOccurrenceSpecification](#)

### **Constraints**

- **no\_occurrence\_specifications\_below**  
No other OccurrenceSpecifications on a given Lifeline in an InteractionOperand may appear below a DestructionOccurrenceSpecification.

```

inv: let o : InteractionOperand = enclosingOperand in o->notEmpty() and
let peerEvents : OrderedSet(OccurrenceSpecification) = covered.events->select(enclosingOperand = o)
in peerEvents->last() = self

```

## **ExecutionOccurrenceSpecification [Class]**

### **Description**

An ExecutionOccurrenceSpecification represents moments in time at which Actions or Behaviors start or finish.

### **Diagrams**

[Occurrences](#)

### **Generalizations**

[OccurrenceSpecification](#)

## Association Ends

- execution : [ExecutionSpecification](#) [1..1] (opposite [A\\_execution\\_executionOccurrenceSpecification::executionOccurrenceSpecification](#))  
References the execution specification describing the execution that is started or finished at this execution event.

## ExecutionSpecification [Abstract Class]

### Description

An ExecutionSpecification is a specification of the execution of a unit of Behavior or Action within the Lifeline. The duration of an ExecutionSpecification is represented by two OccurrenceSpecifications, the start OccurrenceSpecification and the finish OccurrenceSpecification.

### Diagrams

[Interactions](#), [Occurrences](#)

### Generalizations

[InteractionFragment](#)

### Specializations

[ActionExecutionSpecification](#), [BehaviorExecutionSpecification](#)

## Association Ends

- finish : [OccurrenceSpecification](#) [1..1] (opposite [A\\_finish\\_executionSpecification::executionSpecification](#))  
References the OccurrenceSpecification that designates the finish of the Action or Behavior.
- start : [OccurrenceSpecification](#) [1..1] (opposite [A\\_start\\_executionSpecification::executionSpecification](#))  
References the OccurrenceSpecification that designates the start of the Action or Behavior.

## Constraints

- same\_lifeline  
The startEvent and the finishEvent must be on the same Lifeline.

```
inv: start.covered = finish.covered
```

## Gate [Class]

### Description

A Gate is a MessageEnd which serves as a connection point for relating a Message which has a MessageEnd (sendEvent / receiveEvent) outside an InteractionFragment with another Message which has a MessageEnd (receiveEvent / sendEvent) inside that InteractionFragment.

## Diagrams

[Interactions](#), [Messages](#), [Fragments](#), [Interaction Uses](#)

## Generalizations

[MessageEnd](#)

## Operations

- `isOutsideCF()` : [Boolean](#) [1..1]  
This query returns true if this Gate is attached to the boundary of a CombinedFragment, and its other end (if present) is outside of the same CombinedFragment.

```
body: self.oppositeEnd()-> notEmpty() and combinedFragment->notEmpty() implies
let oppEnd : MessageEnd = self.oppositeEnd()->asOrderedSet()->first() in
if oppEnd.ocIsKindOf(MessageOccurrenceSpecification)
then let oppMOS : MessageOccurrenceSpecification = oppEnd.ocAsType(MessageOccurrenceSpecification)
in self.combinedFragment.enclosingInteraction.ocAsType(InteractionFragment)->asSet()->
  union(self.combinedFragment.enclosingOperand.ocAsType(InteractionFragment)->asSet()->
  oppMOS.enclosingInteraction.ocAsType(InteractionFragment)->asSet()->
  union(oppMOS.enclosingOperand.ocAsType(InteractionFragment)->asSet()->
else let oppGate : Gate = oppEnd.ocAsType(Gate)
in self.combinedFragment.enclosingInteraction.ocAsType(InteractionFragment)->asSet()->
  union(self.combinedFragment.enclosingOperand.ocAsType(InteractionFragment)->asSet()->
  oppGate.combinedFragment.enclosingInteraction.ocAsType(InteractionFragment)->asSet()->
  union(oppGate.combinedFragment.enclosingOperand.ocAsType(InteractionFragment)->asSet()->
endif
```

- `isInsideCF()` : [Boolean](#)  
This query returns true if this Gate is attached to the boundary of a CombinedFragment, and its other end (if present) is inside of an InteractionOperator of the same CombinedFragment.

```
body: self.oppositeEnd()-> notEmpty() and combinedFragment->notEmpty() implies
let oppEnd : MessageEnd = self.oppositeEnd()->asOrderedSet()->first() in
if oppEnd.ocIsKindOf(MessageOccurrenceSpecification)
then let oppMOS : MessageOccurrenceSpecification = oppEnd.ocAsType(MessageOccurrenceSpecification)
in combinedFragment = oppMOS.enclosingOperand.combinedFragment
else let oppGate : Gate = oppEnd.ocAsType(Gate)
in combinedFragment = oppGate.combinedFragment
endif
```

- `isActual()` : [Boolean](#)  
This query returns true value if this Gate is an actualGate of an InteractionUse.

```
body: interactionUse->notEmpty()
```

- `isFormal()` : [Boolean](#)  
This query returns true if this Gate is a formalGate of an Interaction.

```
body: interaction->notEmpty()
```

- `getName()` : [String](#)  
This query returns the name of the gate, either the explicit name (.name) or the constructed name ('out\_' or 'in\_' concatenated in front of .message.name) if the explicit name is not present.

```

body: if name->notEmpty() then name->asOrderedSet()->first()
else if isActual() or isOutsideCF()
  then if isSend()
    then 'out_'.concat(self.message.name->asOrderedSet()->first())
    else 'in_'.concat(self.message.name->asOrderedSet()->first())
    endif
  else if isReceive()
    then 'in_'.concat(self.message.name->asOrderedSet()->first())
    else 'out_'.concat(self.message.name->asOrderedSet()->first())
    endif
  endif
endif
endif

```

- matches(gateToMatch : [Gate](#) [1..1]) : [Boolean](#)

This query returns true if the name of this Gate matches the name of the in parameter Gate, and the messages for the two Gates correspond. The Message for one Gate (say A) corresponds to the Message for another Gate (say B) if (A and B have the same name value) and (if A is a sendEvent then B is a receiveEvent) and (if A is a receiveEvent then B is a sendEvent) and (A and B have the same messageSort value) and (A and B have the same signature value).

```

body: self.getName() = gateToMatch.getName() and
self.message.messageSort = gateToMatch.message.messageSort and
self.message.name = gateToMatch.message.name and
self.message.sendEvent->includes(self) implies gateToMatch.message.receiveEvent-
>includes(gateToMatch) and
self.message.receiveEvent->includes(self) implies gateToMatch.message.sendEvent-
>includes(gateToMatch) and
self.message.signature = gateToMatch.message.signature

```

## Constraints

- actual\_gate\_matched

If this Gate is an actualGate, it must have exactly one matching formalGate within the referred Interaction.

```

inv: interactionUse->notEmpty() implies interactionUse.refersTo.formalGate->select(matches(self))-
>size()=1

```

- inside\_cf\_matched

If this Gate is inside a CombinedFragment, it must have exactly one matching Gate which is outside of that CombinedFragment.

```

inv: isInsideCF() implies combinedFragment.cfragmentGate->select(isOutsideCF() and matches(self))-
>size()=1

```

- outside\_cf\_matched

If this Gate is outside an 'alt' CombinedFragment, for every InteractionOperator inside that CombinedFragment there must be exactly one matching Gate inside the CombinedFragment with its opposing end enclosed by that InteractionOperator. If this Gate is outside CombinedFragment with operator other than 'alt', there must be exactly one matching Gate inside that CombinedFragment.

```

inv: isOutsideCF() implies
  if self.combinedFragment.interactionOperator->asOrderedSet()->first() = InteractionOperatorKind::alt
  then self.combinedFragment.operand->forall(op : InteractionOperator |
self.combinedFragment.cfragmentGate->select(isInsideCF() and
oppositeEnd().enclosingFragment()->includes(self.combinedFragment) and matches(self))->size()=1)
  else self.combinedFragment.cfragmentGate->select(isInsideCF() and matches(self))->size()=1
  endif

```

## GeneralOrdering [Class]

### Description

A GeneralOrdering represents a binary relation between two OccurrenceSpecifications, to describe that one OccurrenceSpecification must occur before the other in a valid trace. This mechanism provides the ability to define partial orders of OccurrenceSpecifications that may otherwise not have a specified order.

### Diagrams

[Occurrences](#)

### Generalizations

[NamedElement](#)

### Association Ends

- after : [OccurrenceSpecification](#) [1..1] (opposite [OccurrenceSpecification::toBefore](#))  
The OccurrenceSpecification referenced comes after the OccurrenceSpecification referenced by before.
- before : [OccurrenceSpecification](#) [1..1] (opposite [OccurrenceSpecification::toAfter](#))  
The OccurrenceSpecification referenced comes before the OccurrenceSpecification referenced by after.

### Constraints

- irreflexive\_transitive\_closure  
An occurrence specification must not be ordered relative to itself through a series of general orderings. (In other words, the transitive closure of the general orderings is irreflexive.)

```
inv: after->closure(toAfter.after)->excludes(before)
```

## Interaction [Class]

### Description

An Interaction is a unit of Behavior that focuses on the observable exchange of information between connectable elements.

### Diagrams

[Interactions](#), [Messages](#), [Lifelines](#), [Interaction Uses](#)

### Generalizations

[InteractionFragment](#), [Behavior](#)

### Association Ends

- ♦ action : [Action](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [A action interaction::interaction](#))  
Actions owned by the Interaction.

- ♦ formalGate : [Gate](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [A\\_formalGate\\_interaction::interaction](#))  
Specifies the gates that form the message interface between this Interaction and any InteractionUses which reference it.
- ♦ fragment : [InteractionFragment](#) [0..\*]{ordered, subsets [Namespace::ownedMember](#)} (opposite [InteractionFragment::enclosingInteraction](#))  
The ordered set of fragments in the Interaction.
- ♦ lifeline : [Lifeline](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [Lifeline::interaction](#))  
Specifies the participants in this Interaction.
- ♦ message : [Message](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [Message::interaction](#))  
The Messages contained in this Interaction.

### Constraints

- not\_contained  
An Interaction instance must not be contained within another Interaction instance.

```
inv: enclosingInteraction->isEmpty()
```

## InteractionConstraint [Class]

### Description

An InteractionConstraint is a Boolean expression that guards an operand in a CombinedFragment.

### Diagrams

[Fragments](#)

### Generalizations

[Constraint](#)

### Association Ends

- ♦ maxint : [ValueSpecification](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_maxint\\_interactionConstraint::interactionConstraint](#))  
The maximum number of iterations of a loop
- ♦ minint : [ValueSpecification](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_minint\\_interactionConstraint::interactionConstraint](#))  
The minimum number of iterations of a loop

### Constraints

- minint\_maxint  
Minint/maxint can only be present if the InteractionConstraint is associated with the operand of a loop CombinedFragment.

```
inv: maxint->notEmpty() or minint->notEmpty() implies
interactionOperand.combinedFragment.interactionOperator =
InteractionOperatorKind::loop
```

- **minint\_non\_negative**  
If minint is specified, then the expression must evaluate to a non-negative integer.

```
inv: minint->notEmpty() implies
minint->asSequence()->first().integerValue() >= 0
```

- **maxint\_positive**  
If maxint is specified, then the expression must evaluate to a positive integer.

```
inv: maxint->notEmpty() implies
maxint->asSequence()->first().integerValue() > 0
```

- **dynamic\_variables**  
The dynamic variables that take part in the constraint must be owned by the ConnectableElement corresponding to the covered Lifeline.

Cannot be expressed in OCL

- **global\_data**  
The constraint may contain references to global data or write-once data.

Cannot be expressed in OCL

- **maxint\_greater\_equal\_minint**  
If maxint is specified, then minint must be specified and the evaluation of maxint must be  $\geq$  the evaluation of minint.

```
inv: maxint->notEmpty() implies (minint->notEmpty() and
maxint->asSequence()->first().integerValue() >=
minint->asSequence()->first().integerValue() )
```

## InteractionFragment [Abstract Class]

### Description

InteractionFragment is an abstract notion of the most general interaction unit. An InteractionFragment is a piece of an Interaction. Each InteractionFragment is conceptually like an Interaction by itself.

### Diagrams

[Interactions](#), [Lifelines](#), [Occurrences](#), [Fragments](#), [Interaction Uses](#)

### Generalizations

[NamedElement](#)

### Specializations

[CombinedFragment](#), [Continuation](#), [ExecutionSpecification](#), [Interaction](#), [InteractionOperand](#), [InteractionUse](#), [OccurrenceSpecification](#), [StateInvariant](#)



## Association Ends

- covered : [Lifeline](#) [0..\*] (opposite [Lifeline::coveredBy](#))  
References the Lifelines that the InteractionFragment involves.
- enclosingInteraction : [Interaction](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [Interaction::fragment](#))  
The Interaction enclosing this InteractionFragment.
- enclosingOperand : [InteractionOperand](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [InteractionOperand::fragment](#))  
The operand enclosing this InteractionFragment (they may nest recursively).
- ♦ generalOrdering : [GeneralOrdering](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [A\\_generalOrdering\\_interactionFragment::interactionFragment](#))  
The general ordering relationships contained in this fragment.

## InteractionOperand [Class]

### Description

An InteractionOperand is contained in a CombinedFragment. An InteractionOperand represents one operand of the expression given by the enclosing CombinedFragment.

### Diagrams

[Fragments](#)

### Generalizations

[InteractionFragment](#), [Namespace](#)

### Association Ends

- ♦ fragment : [InteractionFragment](#) [0..\*]{ordered, subsets [Namespace::ownedMember](#)} (opposite [InteractionFragment::enclosingOperand](#))  
The fragments of the operand.
- ♦ guard : [InteractionConstraint](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_guard\\_interactionOperand::interactionOperand](#))  
Constraint of the operand.

### Constraints

- guard\_contain\_references  
The guard must contain only references to values local to the Lifeline on which it resides, or values global to the whole Interaction.

Cannot be expressed in OCL

- `guard_directly_prior`  
The guard must be placed directly prior to (above) the `OccurrenceSpecification` that will become the first `OccurrenceSpecification` within this `InteractionOperand`.

Cannot be expressed in OCL

## InteractionOperatorKind [Enumeration]

### Description

`InteractionOperatorKind` is an enumeration designating the different kinds of operators of `CombinedFragments`. The `InteractionOperand` defines the type of operator of a `CombinedFragment`.

### Diagrams

- [Fragments](#)

### Literals

- `seq`  
The `InteractionOperatorKind seq` designates that the `CombinedFragment` represents a weak sequencing between the behaviors of the operands.
- `alt`  
The `InteractionOperatorKind alt` designates that the `CombinedFragment` represents a choice of behavior. At most one of the operands will be chosen. The chosen operand must have an explicit or implicit guard expression that evaluates to true at this point in the interaction. An implicit true guard is implied if the operand has no guard.
- `opt`  
The `InteractionOperatorKind opt` designates that the `CombinedFragment` represents a choice of behavior where either the (sole) operand happens or nothing happens. An option is semantically equivalent to an alternative `CombinedFragment` where there is one operand with non-empty content and the second operand is empty.
- `break`  
The `InteractionOperatorKind break` designates that the `CombinedFragment` represents a breaking scenario in the sense that the operand is a scenario that is performed instead of the remainder of the enclosing `InteractionFragment`. A break operator with a guard is chosen when the guard is true and the rest of the enclosing `Interaction Fragment` is ignored. When the guard of the break operand is false, the break operand is ignored and the rest of the enclosing `InteractionFragment` is chosen. The choice between a break operand without a guard and the rest of the enclosing `InteractionFragment` is done non-deterministically.
- `par`  
The `InteractionOperatorKind par` designates that the `CombinedFragment` represents a parallel merge between the behaviors of the operands. The `OccurrenceSpecifications` of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved.
- `strict`  
The `InteractionOperatorKind strict` designates that the `CombinedFragment` represents a strict sequencing between the behaviors of the operands. The semantics of strict sequencing defines a strict ordering of the operands on the first level within the `CombinedFragment` with `interactionOperator strict`. Therefore `OccurrenceSpecifications` within contained `CombinedFragment` will not directly be compared with other `OccurrenceSpecifications` of the enclosing

CombinedFragment.

- loop  
The InteractionOperatorKind loop designates that the CombinedFragment represents a loop. The loop operand will be repeated a number of times.
- critical  
The InteractionOperatorKind critical designates that the CombinedFragment represents a critical region. A critical region means that the traces of the region cannot be interleaved by other OccurrenceSpecifications (on those Lifelines covered by the region). This means that the region is treated atomically by the enclosing fragment when determining the set of valid traces. Even though enclosing CombinedFragments may imply that some OccurrenceSpecifications may interleave into the region, such as with par-operator, this is prevented by defining a region.
- neg  
The InteractionOperatorKind neg designates that the CombinedFragment represents traces that are defined to be invalid.
- assert  
The InteractionOperatorKind assert designates that the CombinedFragment represents an assertion. The sequences of the operand of the assertion are the only valid continuations. All other continuations result in an invalid trace.
- ignore  
The InteractionOperatorKind ignore designates that there are some message types that are not shown within this combined fragment. These message types can be considered insignificant and are implicitly ignored if they appear in a corresponding execution. Alternatively, one can understand ignore to mean that the message types that are ignored can appear anywhere in the traces.
- consider  
The InteractionOperatorKind consider designates which messages should be considered within this combined fragment. This is equivalent to defining every other message to be ignored.

## InteractionUse [Class]

### Description

An InteractionUse refers to an Interaction. The InteractionUse is a shorthand for copying the contents of the referenced Interaction where the InteractionUse is. To be accurate the copying must take into account substituting parameters with arguments and connect the formal Gates with the actual ones.

### Diagrams

[Interaction Uses](#)

### Generalizations

[InteractionFragment](#)

### Specializations

[PartDecomposition](#)

## Association Ends

- ♦ actualGate : [Gate](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [A\\_actualGate\\_interactionUse::interactionUse](#))  
The actual gates of the InteractionUse.
- ♦ argument : [ValueSpecification](#) [0..\*]{ordered, subsets [Element::ownedElement](#)} (opposite [A\\_argument\\_interactionUse::interactionUse](#))  
The actual arguments of the Interaction.
- refersTo : [Interaction](#) [1..1] (opposite [A\\_refersTo\\_interactionUse::interactionUse](#))  
Refers to the Interaction that defines its meaning.
- ♦ returnValue : [ValueSpecification](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_returnValue\\_interactionUse::interactionUse](#))  
The value of the executed Interaction.
- returnValueRecipient : [Property](#) [0..1] (opposite [A\\_returnValueRecipient\\_interactionUse::interactionUse](#))  
The recipient of the return value.

## Constraints

- gates\_match  
Actual Gates of the InteractionUse must match Formal Gates of the referred Interaction. Gates match when their names are equal and their messages correspond.

```
inv: actualGate->notEmpty() implies  
refersTo.formalGate->forall( fg : Gate | self.actualGate->select(matches(fg))->size()==1) and  
self.actualGate->forall(ag : Gate | refersTo.formalGate->select(matches(ag))->size()==1)
```

- arguments\_are\_constants  
The arguments must only be constants, parameters of the enclosing Interaction or attributes of the classifier owning the enclosing Interaction.

Cannot be expressed in OCL

- returnValueRecipient\_coverage  
The returnValueRecipient must be a Property of a ConnectableElement that is represented by a Lifeline covered by this InteractionUse.

```
inv: returnValueRecipient->asSet()->notEmpty() implies  
let covCE : Set(ConnectableElement) = covered.represents->asSet() in  
covCE->notEmpty() and let classes:Set(Classifier) =  
covCE.type.ocIsKindOf(Classifier).oclAsType(Classifier)->asSet() in  
let allProps : Set(Property) = classes.attribute->union(classes.allParents().attribute)->asSet() in  
allProps->includes(returnValueRecipient)
```

- arguments\_correspond\_to\_parameters  
The arguments of the InteractionUse must correspond to parameters of the referred Interaction.

Cannot be expressed in OCL

- `returnValue_type_recipient_correspondence`  
The type of the `returnValue` must correspond to the type of the `returnValueRecipient`.

```
inv: returnValue.type->asSequence()->notEmpty() implies returnValue.type->asSequence()->first() =
returnValueRecipient.type->asSequence()->first()
```

- `all_lifelines`  
The `InteractionUse` must cover all `Lifelines` of the enclosing `Interaction` that are common with the lifelines covered by the referred `Interaction`. `Lifelines` are common if they have the same selector and represents associationEnd values.

```
inv: let parentInteraction : Set(Interaction) = enclosingInteraction->asSet()->
union(enclosingOperand.combinedFragment->closure(enclosingOperand.combinedFragment)->
collect(enclosingInteraction).oclAsType(Interaction)->asSet()) in
parentInteraction->size()=1 and let refInteraction : Interaction = refersTo in
parentInteraction.covered->forall(intLifeline : Lifeline | refInteraction.covered->
forall( refLifeline : Lifeline | refLifeline.represents = intLifeline.represents and
( refLifeline.selector.oclIsKindOf(LiteralString)->notEmpty() implies
intLifeline.selector.oclIsKindOf(LiteralString)->notEmpty() and
refLifeline.selector.oclAsType(LiteralString).value =
intLifeline.selector.oclAsType(LiteralString).value )
implies self.covered->asSet()->includes(intLifeline)))
```

## Lifeline [Class]

### Description

A `Lifeline` represents an individual participant in the `Interaction`. While parts and structural features may have multiplicity greater than 1, `Lifelines` represent only one interacting entity.

### Diagrams

[Lifelines](#), [Interaction Uses](#)

### Generalizations

[NamedElement](#)

### Association Ends

- `coveredBy` : [InteractionFragment](#) [0..\*] (opposite [InteractionFragment::covered](#))  
References the `InteractionFragments` in which this `Lifeline` takes part.
- `decomposedAs` : [PartDecomposition](#) [0..1] (opposite [A\\_decomposedAs\\_lifeline::lifeline](#))  
References the `Interaction` that represents the decomposition.
- `interaction` : [Interaction](#) [1..1]{subsets [NamedElement::namespace](#)} (opposite [Interaction::lifeline](#))  
References the `Interaction` enclosing this `Lifeline`.
- `represents` : [ConnectableElement](#) [0..1] (opposite [A\\_represents\\_lifeline::lifeline](#))  
References the `ConnectableElement` within the classifier that contains the enclosing `interaction`.
- ♦ `selector` : [ValueSpecification](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_selector\\_lifeline::lifeline](#))  
If the referenced `ConnectableElement` is multivalued, then this specifies the specific individual part within that set.

## Constraints

- **selector\_specified**

The selector for a Lifeline must only be specified if the referenced Part is multivalued.

```
inv: self.selector->notEmpty() = (self.represents.ocIsKindOf(MultiplicityElement) and
self.represents.ocIsType(MultiplicityElement).isMultivalued())
```

- **interaction\_uses\_share\_lifeline**

If a lifeline is in an Interaction referred to by an InteractionUse in an enclosing Interaction, and that lifeline is common with another lifeline in an Interaction referred to by another InteractionUse within that same enclosing Interaction, it must be common to a lifeline within that enclosing Interaction. By common Lifelines we mean Lifelines with the same selector and represents associations.

```
inv: let intUses : Set(InteractionUse) = interaction.interactionUse in intUses->notEmpty() implies
intUses->forall
( iuse : InteractionUse |
let usingInteraction : Set(Interaction) = iuse.enclosingInteraction->asSet()
->union(
iuse.enclosingOperand.combinedFragment->asSet()-
>closure(enclosingOperand.combinedFragment).enclosingInteraction->asSet()
)
in
let peerUses : Set(InteractionUse) = usingInteraction.fragment-
>select(ocIsKindOf(InteractionUse)).ocIsType(InteractionUse)->asSet()
->union(
usingInteraction.fragment->select(ocIsKindOf(CombinedFragment)).ocIsType(CombinedFragment)->asSet()
->closure(operand.fragment-
>select(ocIsKindOf(CombinedFragment)).ocIsType(CombinedFragment)).operand.fragment->
select(ocIsKindOf(InteractionUse)).ocIsType(InteractionUse)->asSet()
)->excluding(iuse)
in
peerUses->notEmpty() implies
peerUses->forall( peerUse : InteractionUse |
peerUse.refersTo.lifeline->forall( l : Lifeline | (l.represents = self.represents and
( self.selector.ocIsKindOf(LiteralString)->notEmpty() implies
l.selector.ocIsKindOf(LiteralString)->notEmpty() and
self.selector.ocIsType(LiteralString).value = l.selector.ocIsType(LiteralString).value )
) implies
usingInteraction.lifeline->select(represents = self.represents and
( self.selector.ocIsKindOf(LiteralString)->notEmpty() implies
selector.ocIsKindOf(LiteralString)->notEmpty() and
self.selector.ocIsType(LiteralString).value = selector.ocIsType(LiteralString).value )
)->notEmpty()
)
)
)
)
```

- **same\_classifier**

The classifier containing the referenced ConnectableElement must be the same classifier, or an ancestor, of the classifier that contains the interaction enclosing this lifeline.

```
inv: represents.namespace->closure(namespace)->includes(interaction._'context')
```

- **selector\_int\_or\_string**

The selector value, if present, must be a LiteralString or a LiteralInteger

```
inv: self.selector->notEmpty() implies
self.selector.ocIsKindOf(LiteralInteger)->notEmpty() or
```

```
self.selector.oclIsKindOf(LiteralInteger)->notEmpty()
```

## Message [Class]

### Description

A Message defines a particular communication between Lifelines of an Interaction.

### Diagrams

[Messages](#), [Information Flows](#)

### Generalizations

[NamedElement](#)

### Attributes

- /messageKind : [MessageKind](#) [1..1] = unknown  
The derived kind of the Message (complete, lost, found, or unknown).
- messageSort : [MessageSort](#) [1..1] = synchCall  
The sort of communication reflected by the Message.

### Association Ends

- ♦ argument : [ValueSpecification](#) [0..\*]{ordered, subsets [Element::ownedElement](#)} (opposite [A\\_argument\\_message::message](#))  
The arguments of the Message.
- connector : [Connector](#) [0..1] (opposite [A\\_connector\\_message::message](#))  
The Connector on which this Message is sent.
- interaction : [Interaction](#) [1..1]{subsets [NamedElement::namespace](#)} (opposite [Interaction::message](#))  
The enclosing Interaction owning the Message.
- receiveEvent : [MessageEnd](#) [0..1]{subsets [A\\_message\\_messageEnd::messageEnd](#)} (opposite [A\\_receiveEvent\\_endMessage::endMessage](#))  
References the Receiving of the Message.
- sendEvent : [MessageEnd](#) [0..1]{subsets [A\\_message\\_messageEnd::messageEnd](#)} (opposite [A\\_sendEvent\\_endMessage::endMessage](#))  
References the Sending of the Message.
- signature : [NamedElement](#) [0..1] (opposite [A\\_signature\\_message::message](#))  
The signature of the Message is the specification of its content. It refers either an Operation or a Signal.

## Operations

- `messageKind() : MessageKind`  
This query returns the `MessageKind` value for this `Message`.

```
body: messageKind
```

- `isDistinguishableFrom(n : NamedElement, ns : Namespace) : Boolean`  
The query `isDistinguishableFrom()` specifies that any two `Messages` may coexist in the same `Namespace`, regardless of their names.

```
body: true
```

## Constraints

- `sending_receiving_message_event`  
If the `sendEvent` and the `receiveEvent` of the same `Message` are on the same `Lifeline`, the `sendEvent` must be ordered before the `receiveEvent`.

```
inv: sendEvent.oclIsKindOf(MessageOccurrenceSpecification)->notEmpty() and
receiveEvent.oclIsKindOf(MessageOccurrenceSpecification)->notEmpty()
implies
let f : Lifeline =
sendEvent.oclIsKindOf(MessageOccurrenceSpecification).oclAsType(MessageOccurrenceSpecification)-
>asOrderedSet()->first().covered in
f =
receiveEvent.oclIsKindOf(MessageOccurrenceSpecification).oclAsType(MessageOccurrenceSpecification)-
>asOrderedSet()->first().covered implies
f.events->indexOf(sendEvent.oclAsType(MessageOccurrenceSpecification)->asOrderedSet()->first() ) <
f.events->indexOf(receiveEvent.oclAsType(MessageOccurrenceSpecification)->asOrderedSet()->first() )
```

- `arguments`  
Arguments of a `Message` must only be: i) attributes of the sending lifeline, ii) constants, iii) symbolic values (which are wildcard values representing any legal value), iv) explicit parameters of the enclosing `Interaction`, v) attributes of the class owning the `Interaction`.

```
Cannot be expressed in OCL
```

- `cannot_cross_boundaries`  
Messages cannot cross boundaries of `CombinedFragments` or their operands. This is true if and only if both `MessageEnds` are enclosed within the same `InteractionFragment` (i.e., an `InteractionOperand` or an `Interaction`).

```
inv: sendEvent->notEmpty() and receiveEvent->notEmpty() implies
let sendEnclosingFrag : Set(InteractionFragment) =
sendEvent->asOrderedSet()->first().enclosingFragment()
in
let receiveEnclosingFrag : Set(InteractionFragment) =
receiveEvent->asOrderedSet()->first().enclosingFragment()
in sendEnclosingFrag = receiveEnclosingFrag
```

- `signature_is_signal`  
In the case when the `Message` signature is a `Signal`, the arguments of the `Message` must correspond to the attributes of the `Signal`. A `Message` Argument corresponds to a `Signal` Attribute if the Argument is of the same Class or a specialization of that of the Attribute.



```

inv: (messageSort = MessageSort::asynchSignal ) and signature.ocIsKindOf(Signal)->notEmpty()
implies
  let signalAttributes : OrderedSet(Property) = signature.ocAsType(Signal).ownedAttribute-
  >asOrderedSet()
  in signalAttributes->size() = self.argument->size() and
  self.argument->forall( o: ValueSpecification |
  not (o.ocIsKindOf(Expression) and o.ocAsType(Expression).symbol->size()=0 and
  o.ocAsType(Expression).operand->isEmpty() ) implies
  let p : Property = signalAttributes->at(self.argument->indexOf(o)) in
  o.type.ocAsType(Classifier).conformsTo(p.type.ocAsType(Classifier))
  )

```

- **occurrence\_specifications**

If the MessageEnds are both OccurrenceSpecifications, then the connector must go between the Parts represented by the Lifelines of the two MessageEnds.

Cannot be expressed in OCL

- **signature\_refer\_to**

The signature must either refer an Operation (in which case messageSort is either synchCall or asynchCall or reply) or a Signal (in which case messageSort is asynchSignal). The name of the NamedElement referenced by signature must be the same as that of the Message.

```

inv: signature->notEmpty() implies
  ((signature.ocIsKindOf(Operation) and
  (messageSort = MessageSort::asynchCall or messageSort = MessageSort::synchCall or messageSort =
  MessageSort::reply)
  ) or (signature.ocIsKindOf(Signal) and messageSort = MessageSort::asynchSignal )
  ) and name = signature.name

```

- **signature\_is\_operation\_request**

In the case when a Message with messageSort synchCall or asynchCall has a non empty Operation signature, the arguments of the Message must correspond to the in and inout parameters of the Operation. A Parameter corresponds to an Argument if the Argument is of the same Class or a specialization of that of the Parameter.

```

inv: (messageSort = MessageSort::asynchCall or messageSort = MessageSort::synchCall) and
signature.ocIsKindOf(Operation)->notEmpty() implies
  let requestParms : OrderedSet(Parameter) = signature.ocAsType(Operation).ownedParameter->
  select(direction = ParameterDirectionKind::inout or direction = ParameterDirectionKind::_'in' )
  in requestParms->size() = self.argument->size() and
  self.argument->forall( o: ValueSpecification |
  not (o.ocIsKindOf(Expression) and o.ocAsType(Expression).symbol->size()=0 and
  o.ocAsType(Expression).operand->isEmpty() ) implies
  let p : Parameter = requestParms->at(self.argument->indexOf(o)) in
  o.type.ocAsType(Classifier).conformsTo(p.type.ocAsType(Classifier))
  )

```

- **signature\_is\_operation\_reply**

In the case when a Message with messageSort reply has a non empty Operation signature, the arguments of the Message must correspond to the out, inout, and return parameters of the Operation. A Parameter corresponds to an Argument if the Argument is of the same Class or a specialization of that of the Parameter.

```

inv: (messageSort = MessageSort::reply) and signature.ocIsKindOf(Operation)->notEmpty() implies
  let replyParms : OrderedSet(Parameter) = signature.ocAsType(Operation).ownedParameter->
  select(direction = ParameterDirectionKind::inout or direction = ParameterDirectionKind::out or
  direction = ParameterDirectionKind::return)
  in replyParms->size() = self.argument->size() and

```

```

self.argument->forAll( o: ValueSpecification | o.oclIsKindOf(Expression)->notEmpty() and let e :
Expression = o.oclAsType(Expression) in
e.operand->notEmpty() implies
let p : Parameter = replyParms->at(self.argument->indexOf(o)) in
e.operand->asSequence()->first().type.oclAsType(Classifier).conformsTo(p.type.oclAsType(Classifier))
)

```

## MessageEnd [Abstract Class]

### Description

MessageEnd is an abstract specialization of NamedElement that represents what can occur at the end of a Message.

### Diagrams

[Messages](#)

### Generalizations

[NamedElement](#)

### Specializations

[Gate](#), [MessageOccurrenceSpecification](#)

### Association Ends

- message : [Message](#) [0..1] (opposite [A\\_message\\_messageEnd::messageEnd](#))  
References a Message.

### Operations

- oppositeEnd() : [MessageEnd](#) [0..\*]  
This query returns a set including the MessageEnd (if exists) at the opposite end of the Message for this MessageEnd.

```
pre: message->notEmpty()
```

```
body: message->asSet().messageEnd->asSet()->excluding(self)
```

- isSend() : [Boolean](#)  
This query returns value true if this MessageEnd is a sendEvent.

```
pre: message->notEmpty()
```

```
body: message.sendEvent->asSet()->includes(self)
```

- isReceive() : [Boolean](#)  
This query returns value true if this MessageEnd is a receiveEvent.

```
pre: message->notEmpty()
```

```
body: message.receiveEvent->asSet()->includes(self)
```

- enclosingFragment() : [InteractionFragment](#) [0..\*]  
This query returns a set including the enclosing InteractionFragment this MessageEnd is enclosed within.

```

body: if self->select(oclIsKindOf(Gate))->notEmpty()
then -- it is a Gate
let endGate : Gate =
  self->select(oclIsKindOf(Gate)).oclAsType(Gate)->asOrderedSet()->first()
  in
  if endGate.isOutsideCF()
  then endGate.combinedFragment.enclosingInteraction.oclAsType(InteractionFragment)->asSet()->
    union(endGate.combinedFragment.enclosingOperand.oclAsType(InteractionFragment)->asSet())
  else if endGate.isInsideCF()
  then endGate.combinedFragment.oclAsType(InteractionFragment)->asSet()
  else if endGate.isFormal()
  then endGate.interaction.oclAsType(InteractionFragment)->asSet()
  else if endGate.isActual()
  then endGate.interactionUse.enclosingInteraction.oclAsType(InteractionFragment)->asSet()->
    union(endGate.interactionUse.enclosingOperand.oclAsType(InteractionFragment)->asSet())
  else null
  endif
  endif
  endif
endif
else -- it is a MessageOccurrenceSpecification
let endMOS : MessageOccurrenceSpecification =
  self-
  >select(oclIsKindOf(MessageOccurrenceSpecification)).oclAsType(MessageOccurrenceSpecification)-
  >asOrderedSet()->first()
  in
  if endMOS.enclosingInteraction->notEmpty()
  then endMOS.enclosingInteraction.oclAsType(InteractionFragment)->asSet()
  else endMOS.enclosingOperand.oclAsType(InteractionFragment)->asSet()
  endif
endif
endif

```

## MessageKind [Enumeration]

### Description

This is an enumerated type that identifies the type of Message.

### Diagrams

- [Messages](#)

### Literals

- complete  
sendEvent and receiveEvent are present
- lost  
sendEvent present and receiveEvent absent
- found  
sendEvent absent and receiveEvent present

- unknown  
sendEvent and receiveEvent absent (should not appear)

## MessageOccurrenceSpecification [Class]

### Description

A MessageOccurrenceSpecification specifies the occurrence of Message events, such as sending and receiving of Signals or invoking or receiving of Operation calls. A MessageOccurrenceSpecification is a kind of MessageEnd. Messages are generated either by synchronous Operation calls or asynchronous Signal sends. They are received by the execution of corresponding AcceptEventActions.

### Diagrams

[Messages](#)

### Generalizations

[MessageEnd](#), [OccurrenceSpecification](#)

### Specializations

[DestructionOccurrenceSpecification](#)

## MessageSort [Enumeration]

### Description

This is an enumerated type that identifies the type of communication action that was used to generate the Message.

### Diagrams

- [Messages](#)

### Literals

- synchCall  
The message was generated by a synchronous call to an operation.
- asynchCall  
The message was generated by an asynchronous call to an operation; i.e., a CallAction with isSynchronous = false.
- asynchSignal  
The message was generated by an asynchronous send action.
- createMessage  
The message designating the creation of another lifeline object.
- deleteMessage  
The message designating the termination of another lifeline.

- reply  
The message is a reply message to an operation call.

## OccurrenceSpecification [Class]

### Description

An OccurrenceSpecification is the basic semantic unit of Interactions. The sequences of occurrences specified by them are the meanings of Interactions.

### Diagrams

[Interactions](#), [Messages](#), [Lifelines](#), [Occurrences](#)

### Generalizations

[InteractionFragment](#)

### Specializations

[ExecutionOccurrenceSpecification](#), [MessageOccurrenceSpecification](#)

### Association Ends

- covered : [Lifeline](#) [1..1]{redefines [InteractionFragment::covered](#)} (opposite [A\\_covered\\_events::events](#))  
References the Lifeline on which the OccurrenceSpecification appears.
- toAfter : [GeneralOrdering](#) [0..\*] (opposite [GeneralOrdering::before](#))  
References the GeneralOrderings that specify EventOccurrences that must occur after this OccurrenceSpecification.
- toBefore : [GeneralOrdering](#) [0..\*] (opposite [GeneralOrdering::after](#))  
References the GeneralOrderings that specify EventOccurrences that must occur before this OccurrenceSpecification.

## PartDecomposition [Class]

### Description

A PartDecomposition is a description of the internal Interactions of one Lifeline relative to an Interaction.

### Diagrams

[Lifelines](#), [Interaction Uses](#)

### Generalizations

[InteractionUse](#)

### Constraints

- commutativity\_of\_decomposition  
Assume that within Interaction X, Lifeline L is of class C and decomposed to D. Assume also that there is within X an

InteractionUse (say) U that covers L. According to the constraint above U will have a counterpart CU within D. Within the Interaction referenced by U, L should also be decomposed, and the decomposition should reference CU. (This rule is called commutativity of decomposition.)

Cannot be expressed in OCL

- **assume**  
Assume that within Interaction X, Lifeline L is of class C and decomposed to D. Within X there is a sequence of constructs along L (such constructs are CombinedFragments, InteractionUse and (plain) OccurrenceSpecifications). Then a corresponding sequence of constructs must appear within D, matched one-to-one in the same order. i) CombinedFragment covering L are matched with an extra-global CombinedFragment in D. ii) An InteractionUse covering L is matched with a global (i.e., covering all Lifelines) InteractionUse in D. iii) A plain OccurrenceSpecification on L is considered an actualGate that must be matched by a formalGate of D.

Cannot be expressed in OCL

- **parts\_of\_internal\_structures**  
PartDecompositions apply only to Parts that are Parts of Internal Structures not to Parts of Collaborations.

Cannot be expressed in OCL

## StateInvariant [Class]

### Description

A StateInvariant is a runtime constraint on the participants of the Interaction. It may be used to specify a variety of different kinds of Constraints, such as values of Attributes or Variables, internal or external States, and so on. A StateInvariant is an InteractionFragment and it is placed on a Lifeline.

### Diagrams

[Interactions, Lifelines](#)

### Generalizations

[InteractionFragment](#)

### Association Ends

- covered : [Lifeline](#) [1..1]{redefines [InteractionFragment::covered](#)} (opposite [A\\_covered\\_stateInvariant::stateInvariant](#))  
References the Lifeline on which the StateInvariant appears.
- ♦ invariant : [Constraint](#) [1..1]{subsets [Element::ownedElement](#)} (opposite [A\\_invariant\\_stateInvariant::stateInvariant](#))  
A Constraint that should hold at runtime for this StateInvariant.

## 17.13 Association Descriptions

### A\_action\_actionExecutionSpecification [Association]

#### Diagrams

[Occurrences](#)

#### Owned Ends

- actionExecutionSpecification : [ActionExecutionSpecification](#) [0..\*] (opposite [ActionExecutionSpecification::action](#))

### A\_action\_interaction [Association]

#### Diagrams

[Interactions](#)

#### Owned Ends

- interaction : [Interaction](#) [0..1]{subsets [Element::owner](#)} (opposite [Interaction::action](#))

### A\_actualGate\_interactionUse [Association]

#### Diagrams

[Interaction Uses](#)

#### Owned Ends

- interactionUse : [InteractionUse](#) [0..1]{subsets [Element::owner](#)} (opposite [InteractionUse::actualGate](#))

### A\_argument\_interactionUse [Association]

#### Diagrams

[Interaction Uses](#)

#### Owned Ends

- interactionUse : [InteractionUse](#) [0..1]{subsets [Element::owner](#)} (opposite [InteractionUse::argument](#))

## A\_argument\_message [Association]

### Diagrams

[Messages](#)

### Owned Ends

- message : [Message](#) [0..1]{subsets [Element::owner](#)} (opposite [Message::argument](#))

## A\_before\_toAfter [Association]

### Diagrams

[Occurrences](#)

### Member Ends

- [GeneralOrdering::before](#)
- [OccurrenceSpecification::toAfter](#)

## A\_behavior\_behaviorExecutionSpecification [Association]

### Diagrams

[Occurrences](#)

### Owned Ends

- behaviorExecutionSpecification : [BehaviorExecutionSpecification](#) [0..\*] (opposite [BehaviorExecutionSpecification::behavior](#))

## A\_cfragmentGate\_combinedFragment [Association]

### Diagrams

[Fragments](#)

### Owned Ends

- combinedFragment : [CombinedFragment](#) [0..1]{subsets [Element::owner](#)} (opposite [CombinedFragment::cfragmentGate](#))



## A\_connector\_message [Association]

### Diagrams

[Messages](#)

### Owned Ends

- message : [Message](#) [0..\*] (opposite [Message::connector](#))

## A\_covered\_coveredBy [Association]

### Diagrams

[Lifelines](#)

### Member Ends

- [InteractionFragment::covered](#)
- [Lifeline::coveredBy](#)

## A\_covered\_events [Association]

### Diagrams

[Lifelines](#)

### Owned Ends

- events : [OccurrenceSpecification](#) [0..\*]{ordered, subsets [Lifeline::coveredBy](#)} (opposite [OccurrenceSpecification::covered](#))

## A\_covered\_stateInvariant [Association]

### Diagrams

[Lifelines](#)

### Owned Ends

- stateInvariant : [StateInvariant](#) [0..\*]{subsets [Lifeline::coveredBy](#)} (opposite [StateInvariant::covered](#))

## A\_decomposedAs\_lifeline [Association]

### Diagrams

[Lifelines](#), [Interaction Uses](#)

### Owned Ends

- lifeline : [Lifeline](#) [1..1] (opposite [Lifeline::decomposedAs](#))

## A\_execution\_executionOccurrenceSpecification [Association]

### Diagrams

[Occurrences](#)

### Owned Ends

- executionOccurrenceSpecification : [ExecutionOccurrenceSpecification](#) [0..2] (opposite [ExecutionOccurrenceSpecification::execution](#))

## A\_finish\_executionSpecification [Association]

### Diagrams

[Occurrences](#)

### Owned Ends

- executionSpecification : [ExecutionSpecification](#) [0..\*] (opposite [ExecutionSpecification::finish](#))

## A\_formalGate\_interaction [Association]

### Diagrams

[Interactions](#)

### Owned Ends

- interaction : [Interaction](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [Interaction::formalGate](#))

## A\_fragment\_enclosingInteraction [Association]

### Diagrams

[Interactions](#)

### Member Ends

- [Interaction::fragment](#)
- [InteractionFragment::enclosingInteraction](#)

## A\_fragment\_enclosingOperand [Association]

### Diagrams

[Fragments](#)

### Member Ends

- [InteractionOperand::fragment](#)
- [InteractionFragment::enclosingOperand](#)

## A\_generalOrdering\_interactionFragment [Association]

### Diagrams

[Occurrences](#)

### Owned Ends

- interactionFragment : [InteractionFragment](#) [0..1]{subsets [Element::owner](#)} (opposite [InteractionFragment::generalOrdering](#))

## A\_guard\_interactionOperand [Association]

### Diagrams

[Fragments](#)

### Owned Ends

- interactionOperand : [InteractionOperand](#) [1..1]{subsets [Element::owner](#)} (opposite [InteractionOperand::guard](#))

## A\_invariant\_stateInvariant [Association]

### Diagrams

[Interactions](#)

### Owned Ends

- stateInvariant : [StateInvariant](#) [0..1]{subsets [Element::owner](#)} (opposite [StateInvariant::invariant](#))

## A\_lifeline\_interaction [Association]

### Diagrams

[Lifelines](#)

### Member Ends

- [Interaction::lifeline](#)
- [Lifeline::interaction](#)

## A\_maxint\_interactionConstraint [Association]

### Diagrams

[Fragments](#)

### Owned Ends

- interactionConstraint : [InteractionConstraint](#) [0..1]{subsets [Element::owner](#)} (opposite [InteractionConstraint::maxint](#))

## A\_message\_considerIgnoreFragment [Association]

### Diagrams

[Fragments](#)

### Owned Ends

- considerIgnoreFragment : [ConsiderIgnoreFragment](#) [0..\*] (opposite [ConsiderIgnoreFragment::message](#))

## A\_message\_interaction [Association]

### Diagrams

[Messages](#)

### Member Ends

- [Interaction::message](#)
- [Message::interaction](#)

## A\_message\_messageEnd [Association]

### Diagrams

[Messages](#)

### Owned Ends

- messageEnd : [MessageEnd](#) [0..2] (opposite [MessageEnd::message](#))

## A\_minint\_interactionConstraint [Association]

### Diagrams

[Fragments](#)

### Owned Ends

- interactionConstraint : [InteractionConstraint](#) [0..1]{subsets [Element::owner](#)} (opposite [InteractionConstraint::minint](#))

## A\_operand\_combinedFragment [Association]

### Diagrams

[Fragments](#)

### Owned Ends

- combinedFragment : [CombinedFragment](#) [0..1]{subsets [Element::owner](#)} (opposite [CombinedFragment::operand](#))

## A\_receiveEvent\_endMessage [Association]

### Diagrams

[Messages](#)

### Owned Ends

- endMessage : [Message](#) [0..1]{subsets [MessageEnd::message](#)} (opposite [Message::receiveEvent](#))

## A\_refersTo\_interactionUse [Association]

### Diagrams

[Interaction Uses](#)

### Owned Ends

- interactionUse : [InteractionUse](#) [0..\*] (opposite [InteractionUse::refersTo](#))

## A\_represents\_lifeline [Association]

### Diagrams

[Lifelines](#)

### Owned Ends

- lifeline : [Lifeline](#) [0..\*] (opposite [Lifeline::represents](#))

## A\_returnValueRecipient\_interactionUse [Association]

### Diagrams

### Owned Ends

- interactionUse : [InteractionUse](#) [0..\*] (opposite [InteractionUse::returnValueRecipient](#))

## A\_returnValue\_interactionUse [Association]

### Diagrams

[Interaction Uses](#)

### Owned Ends

- interactionUse : [InteractionUse](#) [0..1]{subsets [Element::owner](#)} (opposite [InteractionUse::returnValue](#))

## A\_selector\_lifeline [Association]

### Diagrams

[Lifelines](#)

### Owned Ends

- lifeline : [Lifeline](#) [0..1]{subsets [Element::owner](#)} (opposite [Lifeline::selector](#))

## A\_sendEvent\_endMessage [Association]

### Diagrams

[Messages](#)

### Owned Ends

- endMessage : [Message](#) [0..1]{subsets [MessageEnd::message](#)} (opposite [Message::sendEvent](#))

## A\_signature\_message [Association]

### Diagrams

[Messages](#)

### Owned Ends

- message : [Message](#) [0..\*] (opposite [Message::signature](#))

## A\_start\_executionSpecification [Association]

### Diagrams

[Occurrences](#)

### Owned Ends

- executionSpecification : [ExecutionSpecification](#) [0..\*] (opposite [ExecutionSpecification::start](#))

## A\_toBefore\_after [Association]

### Diagrams

[Occurrences](#)

### Member Ends

- [OccurrenceSpecification::toBefore](#)
- [GeneralOrdering::after](#)





between the Actors and the subject, may result in changes to the state of the subject and communications with its environment. A UseCase can include possible variations of its basic behavior, including exceptional behavior and error handling.

The subject of a UseCase could be a system or any other element that may have behavior, such as a Component, subsystem, or Class. Each UseCase specifies a unit of useful functionality that the subject provides to its users (i.e., a specific way of interacting with the subject). This functionality, which is initiated by an Actor, must always be completed for the UseCase to complete. It is deemed complete if, after its execution, the subject will be in a state in which no further inputs or actions are expected and the UseCase can be initiated again, or in an error state.

UseCases can be used both for specification of the (external) requirements on a subject and for the specification of the functionality offered by a subject. Moreover, the UseCases may also state the requirements the specified subject poses on its environment by defining how the Actors should interact with the subject so that it will be able to perform its services.

The behaviors of a UseCase can be described by a set of Behaviors (through its ownedBehavior relationship), such as Interactions, Activities, and StateMachines, or by pre-conditions and post-conditions as well as by natural language text where appropriate. It may also be described indirectly through a Collaboration that uses the UseCase and its Actors as the Classifiers that type its parts. Which of these techniques to use depends on the nature of the UseCase behavior as well as on the intended reader. These descriptions can be combined. An example of a UseCase with an associated StateMachine is shown in Figure 18.12.

UseCases may have associated Actors, which describe how an instance of the Classifier realizing the UseCase and a user playing one of the roles of the Actor interact. Two UseCases specifying the same subject cannot be associated as each of them individually describes a complete usage of the subject.

When a UseCase has an association to an Actor with a multiplicity that is greater than one at the Actor end, it means that more than one Actor instance is involved in initiating the UseCase. The manner in which multiple Actors participate in the UseCase depends on the specific situation on hand and is not defined in this specification. For instance, a particular UseCase might require simultaneous (concurrent) action by two separate Actors (e.g., in launching a nuclear missile) or it might require complementary and successive actions by the Actors (e.g., one Actor starting something and the other one stopping it).

A UseCase may be owned either by a Package or by a Classifier. Although the owning Classifier typically represents the subject to which the owned UseCases apply, this is not necessarily the case. The same UseCase can be applied to multiple subjects.

An Actor models a type of role played by an entity that interacts with the subjects of its associated UseCases (e.g., by exchanging signals and data), but which is *external* in the sense that an instance of an Actor is not a part of the instance of a subject of an associated UseCase. Actors may represent roles played by human users, external hardware, or other systems.

**NOTE.** An Actor does not necessarily represent a specific physical entity but instead a particular role of some entity that is relevant to the specification of its associated UseCases. Thus, a single physical instance may play the role of several different Actors and, conversely, a given Actor may be played by multiple different instances.

**NOTE.** The term “role” is used informally here and does not imply any technical definition of that term found elsewhere in this specification.

When an Actor has an association to a UseCase with a multiplicity that is greater than one at the UseCase end, it means that a given Actor can be involved in multiple UseCases of that type. The specific nature of this multiple involvement depends on the case on hand and is not defined in this specification. Thus, an Actor may initiate multiple UseCases in parallel (concurrently) or at different points in time.

## Extends

An Extend is a relationship from an extending UseCase (the extension) to an extended UseCase (the extendedCase) that specifies how and when the behavior defined in the extending UseCase can be inserted into the behavior defined in the extended UseCase. The extension takes place at one or more specific extension points defined in the extended UseCase.

Extend is intended to be used when there is some additional behavior that should be added, possibly conditionally, to the behavior defined in a UseCase.

The extended UseCase is defined independently of the extending UseCase and is meaningful independently of the extending UseCase. On the other hand, the extending UseCase typically defines behavior that may not necessarily be meaningful by itself. Instead, the extending UseCase defines a set of modular behavior increments that augment an execution of the extended UseCase under specific conditions.

**NOTE.** The same extending UseCase can extend more than one UseCase. Furthermore, an extending UseCase may itself be extended.

Extend is a kind of DirectedRelationship, such that the source is the extending UseCase and the target is the extended UseCase. It is also a kind of NamedElement so that it can have a name in the context of its owning UseCase. The Extend relationship itself is owned by the extension.

An ExtensionPoint identifies a point in the behavior of a UseCase where that behavior can be extended by an Extend relationship. Each ExtensionPoint has a unique name within a UseCase.

The specific manner in which the location of an ExtensionPoint is defined is intentionally unspecified. This is because UseCases may be specified in various formats such as natural language, tables, trees, etc. Therefore, it is not easy to capture its structure accurately or generally by a formal model. The intuition behind the notion of extensionLocation is best explained through the example of a textually described UseCase: Usually, a UseCase with ExtensionPoints consists of a set of finer-grained behavioral fragment descriptions, which are most often executed in sequence. This segmented structuring of the UseCase text allows the original behavioral description to be extended by merging in supplementary behavioral fragment descriptions at the appropriate insertion points between the original fragments (extension points). Thus, an extending UseCase typically consists of one or more behavior fragment descriptions that are to be inserted into the appropriate spots of the extended UseCase. An extensionLocation, therefore, is a specification of all the various ExtensionPoints in a UseCase where supplementary behavioral increments can be merged.

If the condition of the Extend is true at the time the first ExtensionPoint is reached during the execution of the extended UseCase, then all of the appropriate behavior fragments of the extending UseCase will also be executed. If the condition is false, this does not happen. The individual fragments are executed as the corresponding ExtensionPoints of the extending UseCase are reached. Once a given fragment is completed, execution continues with the behavior of the extended UseCase following the ExtensionPoint. Note that even though there are multiple UseCases involved, there is just a single behavior execution.

## **Includes**

Include is a DirectedRelationship between two UseCases, indicating that the behavior of the included UseCase (the addition) is inserted into the behavior of the including UseCase (the includingCase). It is also a kind of NamedElement so that it can have a name in the context of its owning UseCase. The including UseCase may depend on the changes produced by executing the included UseCase. The included UseCase must be available for the behavior of the including UseCase to be completely described.

The Include relationship is intended to be used when there are common parts of the behavior of two or more UseCases. This common part is then extracted to a separate UseCase, to be included by all the base UseCases having this part in common. As the primary use of the Include relationship is for reuse of common parts, what is left in a base UseCase is usually not complete in itself but dependent on the included parts to be meaningful. This is reflected in the direction of the relationship, indicating that the base UseCase depends on the addition but not vice versa.

All of the behavior of the included UseCase is executed at a single location in the included UseCase before execution of the including UseCase is resumed.

The Include relationship allows hierarchical composition of UseCases as well as reuse of UseCases.

### **18.1.4 Notation**

A UseCase is shown as an ellipse, either containing the name of the UseCase or with the name of the UseCase placed below the ellipse. An optional stereotype keyword may be placed above the name.

The subject for a set of UseCases (sometimes called a *system boundary*) may be shown as a rectangle with its name (and associated keywords and stereotypes) in the top-left corner, with the UseCase ellipses visually located inside this rectangle.

Note that this notation for the subject classifier differs from the normal Classifier notation – it has no header or compartments.

Note also that the subject rectangle does not imply that the subject classifier owns the contained UseCases, but merely that the UseCases apply to that classifier. In particular, there is scope for confusion between a UseCase appearing visually contained in a boundary rectangle representing a Classifier that is its subject, and appearing visually contained in a compartment of a Classifier that is its owner (see Figure 18.9).

Attributes and operations may be shown in compartments within the UseCase oval, with the same content as though they were in a normal Classifier rectangle.

ExtensionPoints may be listed in a compartment of the UseCase with the heading **extension points**. Each ExtensionPoint is denoted by a text string within the UseCase oval symbol according to the syntax below:

*<extension point>* ::= *<name>* [*: <explanation>*]

Note that *explanation*, which is optional, may be any informal text or a more precise definition of the location in the behavior of the UseCase where the extension point occurs, such as the name of a State in a StateMachine, an Activity in an activity diagram, a precondition, or a postcondition.

UseCases may have other Associations and Dependencies to other Classifiers (e.g., to denote input/output, events, and behaviors).

The detailed behaviors defined by a UseCase are notated according to the chosen description technique, in a separate diagram or textual document.

A UseCase may also be shown using the standard rectangle notation for Classifiers with an ellipse icon in the upper-right-hand corner of the rectangle. In this case, “extension points” is an optional compartment. This rendering is more suitable when there are a large number of extension points or features.

An Actor is represented by “stick man” icon with the name of the Actor in the vicinity (usually above or below) the icon.

An Actor may also be shown as a Classifier rectangle with the keyword «actor», with the usual notation for all compartments.

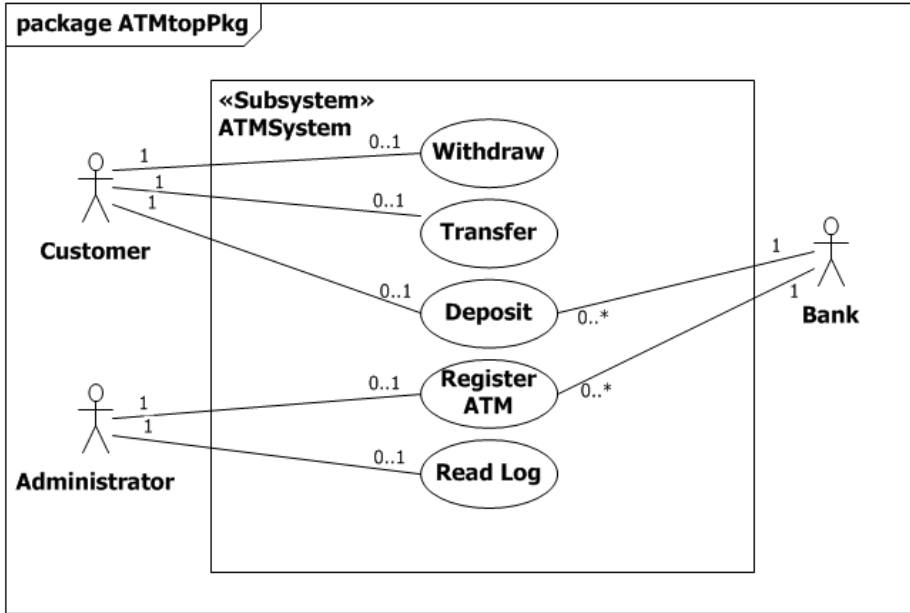
Other icons that convey the kind of Actor may also be used to denote an Actor, such as using a separate icon for non-human Actors.

The nesting (owning) of a UseCase by a Classifier may optionally be represented by nesting the UseCase ellipse inside the Classifier rectangle in a separate compartment. This is a case of the optional compartment for ownedMembers described in [9.2.4](#).

An Extend relationship between UseCases is shown by a dashed arrow with an open arrowhead pointing from the extending UseCase towards the extended UseCase. The arrow is labeled with the keyword «extend». The condition of the Extend as well as references to the ExtensionPoints are optionally shown in a note symbol (see [7.2.4](#)) attached to the corresponding arrow.

An Include relationship between UseCases is shown by a dashed arrow with an open arrowhead pointing from the base UseCase to the included UseCase. The arrow is labeled with the keyword «include».

## 18.1.5 Examples

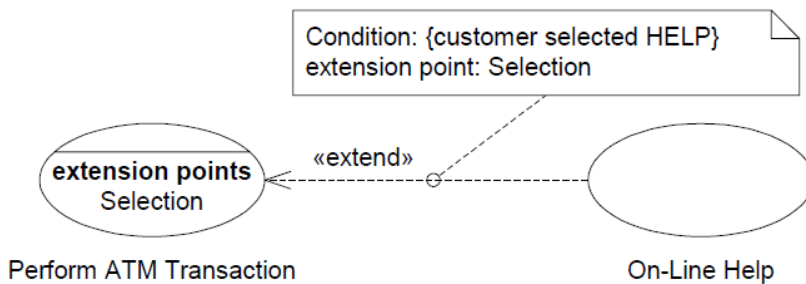


**Figure 18.2** Class diagram of a Package owning a set of UseCases, Actors, and a Subsystem

Figure 18.2 illustrates a class diagram corresponding to the Package ATMtopPkg which owns a set of UseCases, Actors, and a Subsystem that is the subject of the UseCases.

In the UseCase diagram in Figure 18.3 below, the UseCase “Perform ATM Transaction” has an ExtensionPoint “Selection.” This UseCase is extended via that ExtensionPoint by the UseCase “On-Line Help” whenever execution of the “Perform ATM Transaction” UseCase occurrence is at the location referenced by the “Selection” extension point and the customer selects the HELP key.

**NOTE.** The “Perform ATM Transaction” UseCase is defined independently of the “On-Line Help” UseCase.



**Figure 18.3** Example Extend

In Figure 18.4 below a UseCase “Withdraw” includes an independently defined UseCase “Card Identification.”

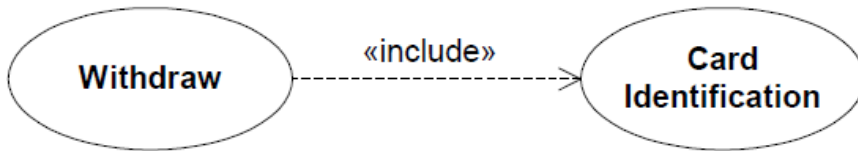


Figure 18.4 Example Include

Figure 18.5 shows a UseCase using the standard rectangle notation for Classifiers with an ellipse icon, and the optional “extension points” compartment.

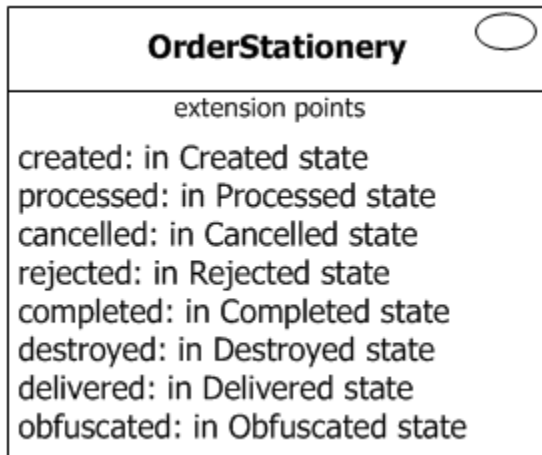


Figure 18.5 UseCase using Classifier rectangle notation

Figure 18.6, Figure 18.7 and Figure 18.8 exemplify the three different notations for Actors.



Figure 18.6 Actor notation using stick-man

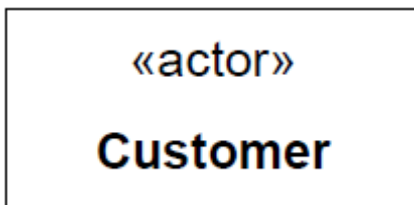


Figure 18.7 Actor notation using Class rectangle



Figure 18.8 Actor notation using icon

Figure 18.9 illustrates an ownedUseCase of a Class using an optional ownedMember compartment.



Figure 18.9 Notation for UseCase owned by Classifier

UseCases need not be owned by their subject. For example, the UseCases shown in Figure 18.10 below apply to the “ATMSystem” subsystem but are owned by various packages as shown in Figure 18.11.

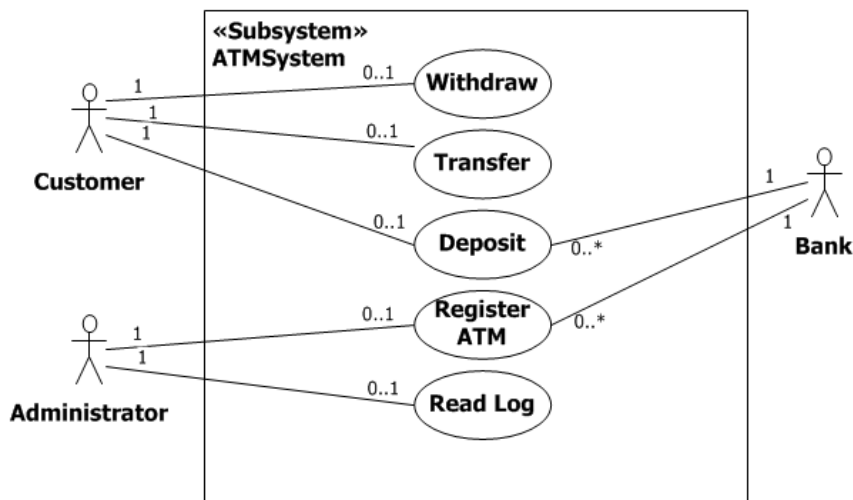


Figure 18.10 Example ATM system with UseCases and Actors

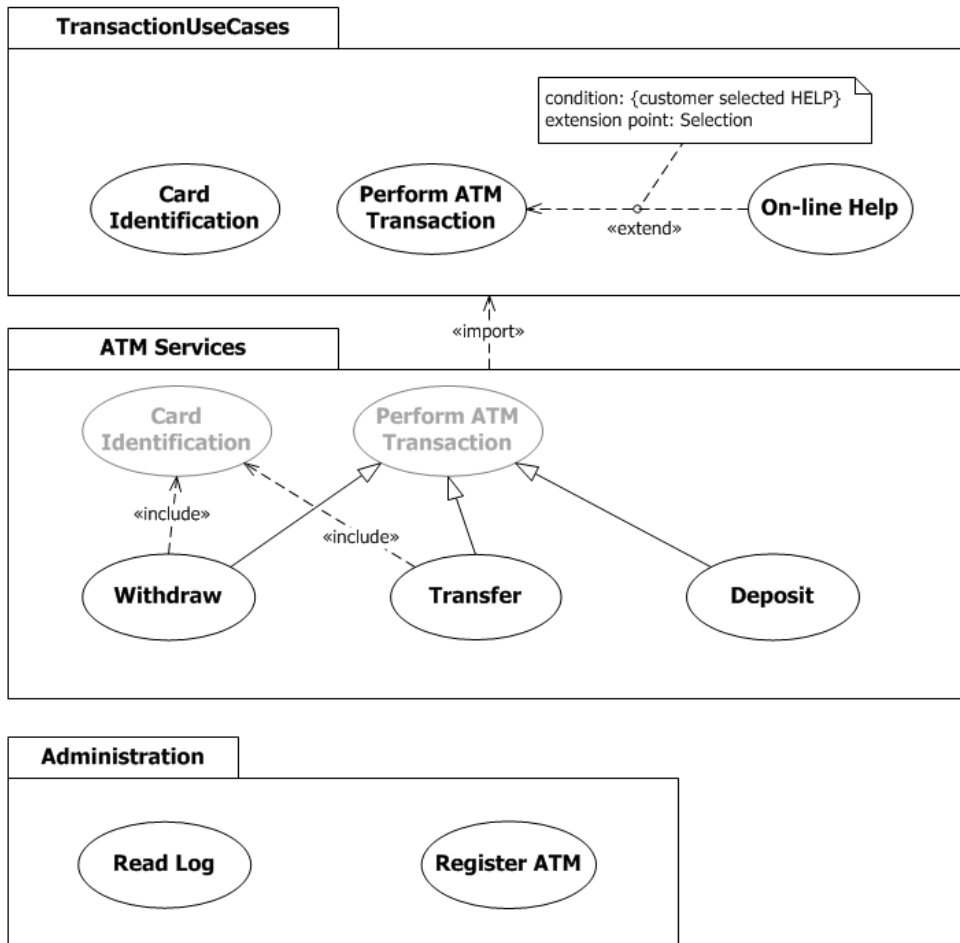


Figure 18.11 Example UseCases owned by Packages

Figure 18.12 shows a UseCase which has a StateMachine as one of its ownedBehaviors. The Classifier symbol for the StateMachine may be shown in the optional “owned behaviors” compartment, and the internal details of the StateMachine are shown in the state machine diagram on the right.



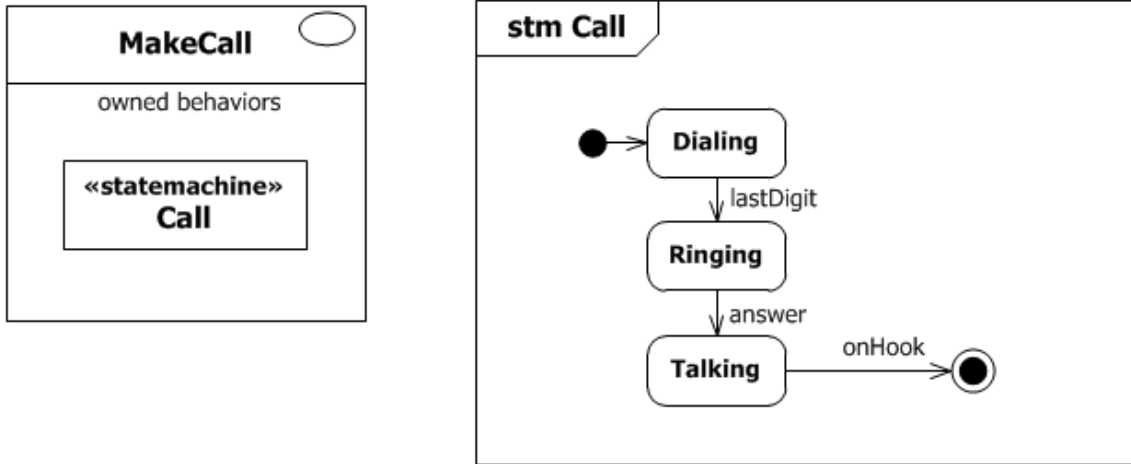


Figure 18.12 Example UseCase with associated StateMachine

## 18.2 Classifier Descriptions

### Actor [Class]

#### Description

An Actor specifies a role played by a user or any other system that interacts with the subject.

#### Diagrams

[Use Cases](#)

#### Generalizations

[BehavioeredClassifier](#)

#### Constraints

- associations  
An Actor can only have Associations to UseCases, Components, and Classes. Furthermore these Associations must be binary.

```
inv: self.attribute->forAll ( a |  
    (a.association->notEmpty()) implies  
    ((a.association.memberEnd->size() = 2) and  
    (a.opposite.class.ocIsKindOf(UseCase) or  
    (a.opposite.class.ocIsKindOf(Class) and not a.opposite.class.ocIsKindOf(Behavior))))))
```

- must\_have\_name  
An Actor must have a name.

```
inv: name->notEmpty()
```

### Extend [Class]

#### Description

A relationship from an extending UseCase to an extended UseCase that specifies how and when the behavior defined in the extending UseCase can be inserted into the behavior defined in the extended UseCase.

#### Diagrams

[Use Cases](#)

#### Generalizations

[NamedElement](#), [DirectedRelationship](#)

#### Association Ends

- ♦ condition : [Constraint](#) [0..1]{subsets [Element::ownedElement](#)} (opposite [A\\_condition\\_extend::extend](#))  
References the condition that must hold when the first ExtensionPoint is reached for the extension to take place. If no

constraint is associated with the Extend relationship, the extension is unconditional.

- extendedCase : [UseCase](#) [1..1]{subsets [DirectedRelationship::target](#)} (opposite [A\\_extendedCase\\_extend::extend](#))  
The UseCase that is being extended.
- extension : [UseCase](#) [1..1]{subsets [NamedElement::namespace](#), subsets [DirectedRelationship::source](#)} (opposite [UseCase::extend](#))  
The UseCase that represents the extension and owns the Extend relationship.
- extensionLocation : [ExtensionPoint](#) [1..\*]{ordered} (opposite [A\\_extensionLocation\\_extension::extension](#))  
An ordered list of ExtensionPoints belonging to the extended UseCase, specifying where the respective behavioral fragments of the extending UseCase are to be inserted. The first fragment in the extending UseCase is associated with the first extension point in the list, the second fragment with the second point, and so on. Note that, in most practical cases, the extending UseCase has just a single behavior fragment, so that the list of ExtensionPoints is trivial.

### Constraints

- extension\_points  
The ExtensionPoints referenced by the Extend relationship must belong to the UseCase that is being extended.

```
inv: extensionLocation->forAll (xp | extendedCase.extensionPoint->includes(xp))
```

## ExtensionPoint [Class]

### Description

An ExtensionPoint identifies a point in the behavior of a UseCase where that behavior can be extended by the behavior of some other (extending) UseCase, as specified by an Extend relationship.

### Diagrams

[Use Cases](#)

### Generalizations

[RedefinableElement](#)

### Association Ends

- useCase : [UseCase](#) [1..1]{subsets [NamedElement::namespace](#)} (opposite [UseCase::extensionPoint](#))  
The UseCase that owns this ExtensionPoint.

### Constraints

- must\_have\_name  
An ExtensionPoint must have a name.

```
inv: name->notEmpty ()
```

## Include [Class]

### Description

An Include relationship specifies that a UseCase contains the behavior defined in another UseCase.

### Diagrams

[Use Cases](#)

### Generalizations

[DirectedRelationship](#), [NamedElement](#)

### Association Ends

- addition : [UseCase](#) [1..1]{subsets [DirectedRelationship::target](#)} (opposite [A\\_addition\\_include::include](#))  
The UseCase that is to be included.
- includingCase : [UseCase](#) [1..1]{subsets [NamedElement::namespace](#), subsets [DirectedRelationship::source](#)} (opposite [UseCase::include](#))  
The UseCase which includes the addition and owns the Include relationship.

## UseCase [Class]

### Description

A UseCase specifies a set of actions performed by its subject, which yields an observable result that is of value for one or more Actors or other stakeholders of the subject.

### Diagrams

[Use Cases](#), [Classifiers](#)

### Generalizations

[BehavioredClassifier](#)

### Association Ends

- ♦ extend : [Extend](#) [0..\*]{subsets [A\\_source\\_directedRelationship::directedRelationship](#), subsets [Namespace::ownedMember](#)} (opposite [Extend::extension](#))  
The Extend relationships owned by this UseCase.
- ♦ extensionPoint : [ExtensionPoint](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [ExtensionPoint::useCase](#))  
The ExtensionPoints owned by this UseCase.
- ♦ include : [Include](#) [0..\*]{subsets [A\\_source\\_directedRelationship::directedRelationship](#), subsets [Namespace::ownedMember](#)} (opposite [Include::includingCase](#))  
The Include relationships owned by this UseCase.

- subject : [Classifier](#) [0..\*] (opposite [Classifier::useCase](#))  
The subjects to which this UseCase applies. The subject or its parts realize all the UseCases that apply to this subject. UseCases need not be attached to any specific subject, however. The subject may, but need not, own the UseCases that apply to it.

## Operations

- allIncludedUseCases() : [UseCase](#) [0..\*]  
The query allIncludedUseCases() returns the transitive closure of all UseCases (directly or indirectly) included by this UseCase.

```
body: self.include.addition->union(self.include.addition->collect(uc | uc.allIncludedUseCases()))->asSet()
```

## Constraints

- binary\_associations  
UseCases can only be involved in binary Associations.

```
inv: Association.allInstances()->forall(a | a.memberEnd.type->includes(self) implies a.memberEnd->size() = 2)
```

- no\_association\_to\_use\_case  
UseCases cannot have Associations to UseCases specifying the same subject.

```
inv: Association.allInstances()->forall(a | a.memberEnd.type->includes(self) implies
(
  let usecases: Set(UseCase) = a.memberEnd.type->select(oclIsKindOf(UseCase))-
  >collect(oclAsType(UseCase))->asSet() in
  usecases->size() > 1 implies usecases->collect(subject)->size() > 1
)
)
```

- cannot\_include\_self  
A UseCase cannot include UseCases that directly or indirectly include it.

```
inv: not allIncludedUseCases()->includes(self)
```

- must\_have\_name  
A UseCase must have a name.

```
inv: name -> notEmpty ()
```

## 18.3 Association Descriptions

### A\_addition\_include [Association]

#### Diagrams

[Use Cases](#)

## Owned Ends

- include : [Include](#) [0..\*]{subsets [A\\_target\\_directedRelationship::directedRelationship](#)} (opposite [Include::addition](#))

## A\_condition\_extend [Association]

### Diagrams

[Use Cases](#)

## Owned Ends

- extend : [Extend](#) [0..1]{subsets [Element::owner](#)} (opposite [Extend::condition](#))

## A\_extend\_extension [Association]

### Diagrams

[Use Cases](#)

## Member Ends

- [UseCase::extend](#)
- [Extend::extension](#)

## A\_extendedCase\_extend [Association]

### Diagrams

[Use Cases](#)

## Owned Ends

- extend : [Extend](#) [0..\*]{subsets [A\\_target\\_directedRelationship::directedRelationship](#)} (opposite [Extend::extendedCase](#))

## A\_extensionLocation\_extension [Association]

### Diagrams

[Use Cases](#)

## Owned Ends

- extension : [Extend](#) [0..\*] (opposite [Extend::extensionLocation](#))

## A\_extensionPoint\_useCase [Association]

### Diagrams

[Use Cases](#)

### Member Ends

- [UseCase::extensionPoint](#)
- [ExtensionPoint::useCase](#)

## A\_include\_includingCase [Association]

### Diagrams

[Use Cases](#)

### Member Ends

- [UseCase::include](#)
- [Include::includingCase](#)

## A\_subject\_useCase [Association]

### Diagrams

[Use Cases](#), [Classifiers](#)

### Member Ends

- [UseCase::subject](#)
- [Classifier::useCase](#)

# 19 Deployments

## 19.1 Summary

The Deployments package specifies constructs that can be used to define the execution architecture of systems and the assignment of software artifacts to system elements. A streamlined model of deployment, sufficient for the majority of modern applications, is provided. Where more elaborate deployment models are required, the package can be extended through profiles or metamodels to represent specific hardware and/or software environments.

## 19.2 Deployments

### 19.2.1 Summary

Deployments capture relationships between logical and/or physical elements of systems and information technology assets assigned to them.

### 19.2.2 Abstract Syntax

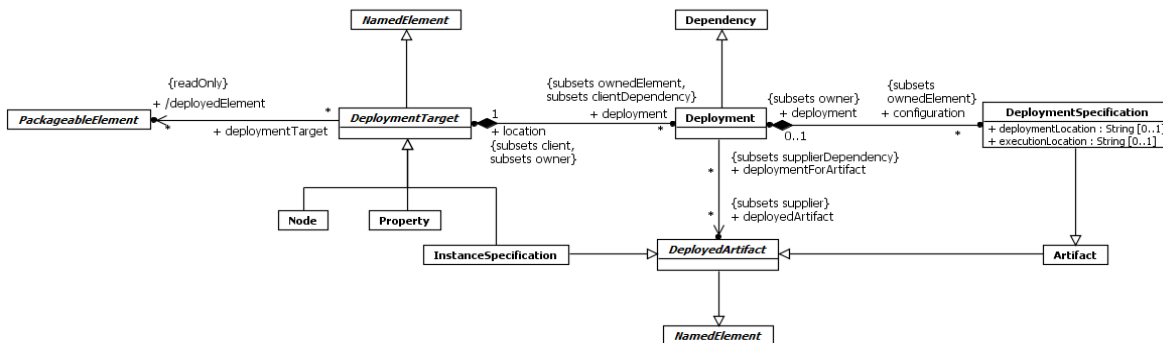


Figure 19.1 Deployments

### 19.2.3 Semantics

A Deployment captures the relationship between a particular conceptual or physical element of a modeled system and the information assets assigned to it. System elements are represented as DeploymentTargets, and information assets, as DeploymentArtifacts. DeploymentTargets and DeploymentArtifacts are abstract classes that cannot be directly instantiated. They are, however, elaborated as concrete classes as described in the Artifacts and Nodes subclasses that follow.

Individual Deployment relationships can be tailored for specific uses by adding DeploymentSpecifications containing configurational and/or parametric information and may be extended in specific component profiles. For example, standard, non-normative stereotypes that a profile might add to DeploymentSpecification include «concurrencyMode», with tagged values {thread, process, none}, and «transactionMode», with tagged values {transaction, nestedTransaction, none}.

DeploymentSpecification information becomes execution-time input to components associated with DeploymentTargets via their deployedElements links. Using these links, DeploymentSpecifications can be targeted at specific container types, as long as the containers are kinds of Components. As shown in Figure 19.1, DeploymentSpecifications can be captured as elements of the Deployment relationships because they are Artifacts (described in the following subclause). Furthermore,



DeploymentSpecifications can only be associated with DeploymentTargets that are ExecutionEnvironments (described in the Nodes subclause, below).

The Deployment relationship between a DeployedArtifact and a DeploymentTarget can be defined at the “type” level and at the “instance” level. At the “type” level, the Deployment connects kinds of DeploymentTargets to kinds of DeployedArtifacts. Whereas, at the “instance” level, the Deployment connects particular DeploymentTargets instances to particular DeployedArtifacts instances. For example, a “type” level Deployment might connect an “application server” with an “order entry request handler.” In contrast, at the “instance” level, three specific application services (say, “app-server1”, “app-server-2” and “app-server3”) may be the DeploymentTargets for six different “request handler” instances.

For modeling complex models consisting of composite structures, a Property, functioning as a part (i.e., owned by a composition), may be the target of a Deployment. Likewise, InstanceSpecifications can be DeploymentTargets in a Deployment relationship, if they represent a Node that functions as a part within an encompassing Node composition hierarchy, or if they represent an Artifact.

### 19.2.4 Notation

DeployedTargets are shown as a perspective view of cube labeled with the name of the DeployedTarget shown prepended by a colon. System elements deployed on a DeployedTarget, and Deployments that connect them, may be drawn inside the perspective cube. Alternately, deployed system elements can be shown as a textual list of element names.

Deployments are depicted using the same dashed line notation as Dependencies. When Deployment relationships are shown within a DeploymentTarget graphic, no labeling is required. Alternately, Deployment relationships can be decorated with the «deploy» keyword when not contained inside a DeployedTarget graphic. Deployment arrows are generally drawn from DeployedArtifacts to the DeployedTargets.

DeploymentSpecifications are graphically displayed as classifier rectangles and may be decorated with the «deployment spec» keyword. They may be attached to a component deployed on a container using a regular dependency arrow.

### 19.2.5 Examples

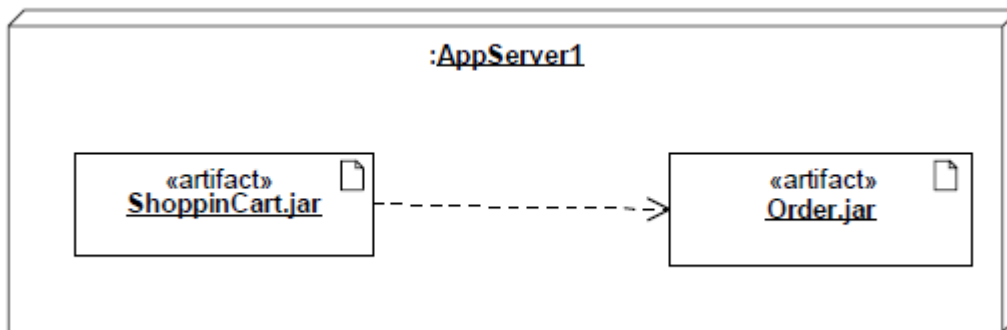


Figure 19.2 A visual representation of the deployment location of artifacts, including a dependency between them, inside a DeployedTarget graphic.

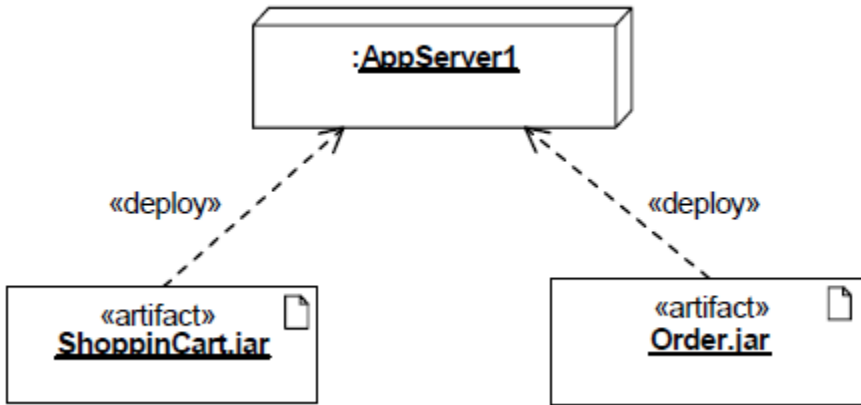


Figure 19.3 Alternative deployment representation of using a dependency called «deploy» used when DeployedArtifacts are visually outside their DeployedTarget graphics.

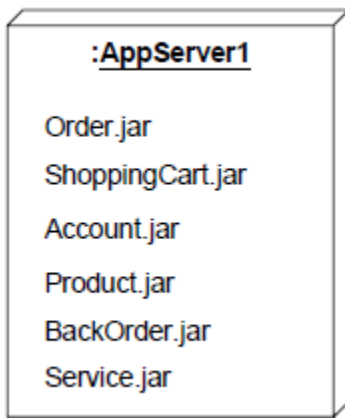


Figure 19.4 Textual list based representation of DeployedArtifacts.



Figure 19.5 DeploymentSpecification for an artifact. On the left, a type-level specification, and on the right, an instance-level specification.

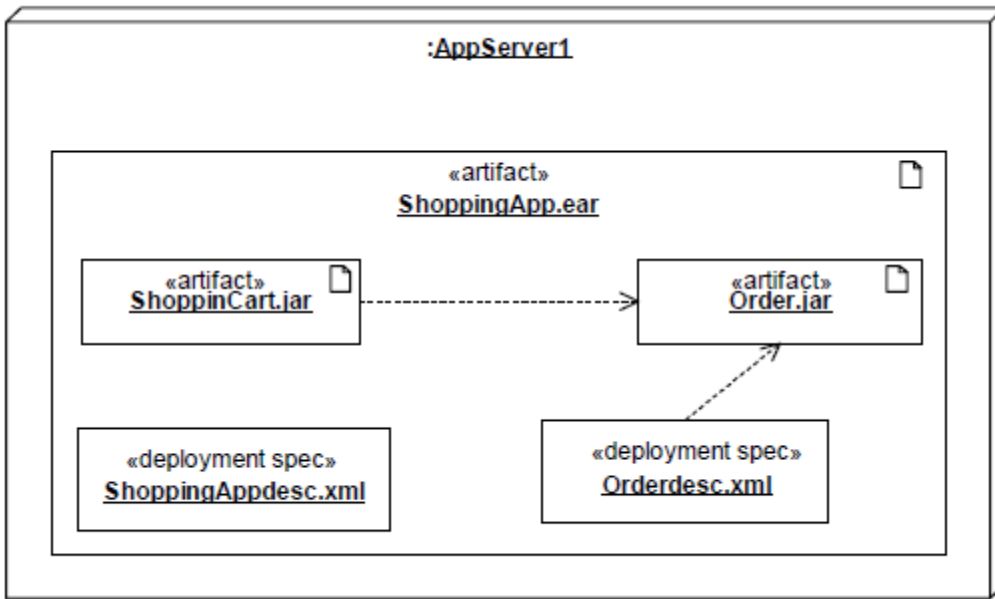


Figure 19.6 DeploymentSpecifications related to the DeployedArtifacts that they parameterize.

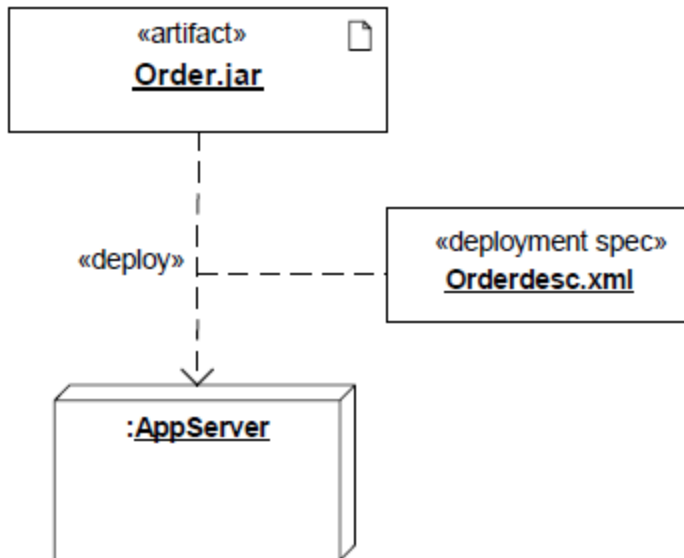


Figure 19.7 A DeploymentSpecification for a DeployedArtifact.

## 19.3 Artifacts

### 19.3.1 Summary

An Artifact represents some (usually reifiable) item of information that is used or produced by a software development process or by operation of a system. Examples of Artifacts include model files, source files, scripts, executable files, database tables, development deliverables, word-processing documents, and mail messages.

### 19.3.2 Abstract Syntax

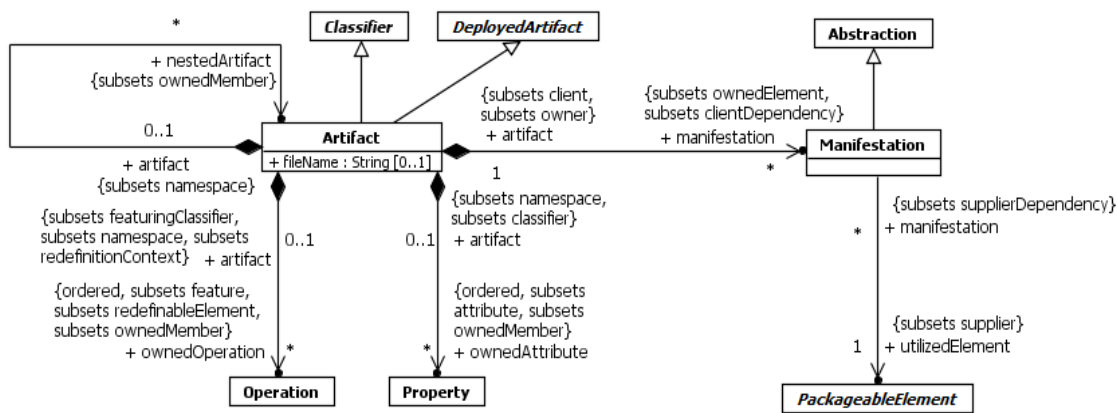


Figure 19.8 Artifacts

### 19.3.3 Semantics

Artifacts elaborate and reify the abstract notion of DeployedArtifact. They represent concrete elements in the physical world, may have Properties representing their features and Operations that can be performed their instances, and may be multiply-instantiated so that different instances may be deployed to various DeploymentTargets, each with separate property values.

More complex Artifacts can be created by organizing them into composition hierarchies. In this way, a DeploymentSpecification for a component may be contained within an Artifact, allowing a component and its parameters to be deployed as a single Artifact instance.

Artifacts may be extended to better represent the needs of specific information items. For example, profiles may extend Artifact to model sets of files. UML defines several standard stereotypes for Artifacts, including «source» and «executable», that may be further specialized as needed. For example, an EJB profile might define «jar» as a subclass of «executable» for executable Java archives.

An Artifact may embody, or manifest, a number of model elements. The Artifact owns the Manifestations, each representing the utilization of some PackageableElement. Profiles may extend the Manifestation relationship to indicate particular forms of embodiment. For example, «tool generated» and «custom code» might be two Manifestations for different Classes embodied in an Artifact.

### 19.3.4 Notation

An Artifact is presented using an ordinary Class rectangle with the keyword «artifact». Alternatively, it may be depicted by an icon (such as the document icon shown in Figure 19.9). Optionally, the underlining of the name of an artifact instance may be omitted, as the context is assumed to be known to users.

A Manifestation is notated in the same way as an Abstraction, that is, as a dashed line with an open arrow-head, labeled with the keyword «manifest».

### 19.3.5 Examples



Figure 19.9 An Artifact instance

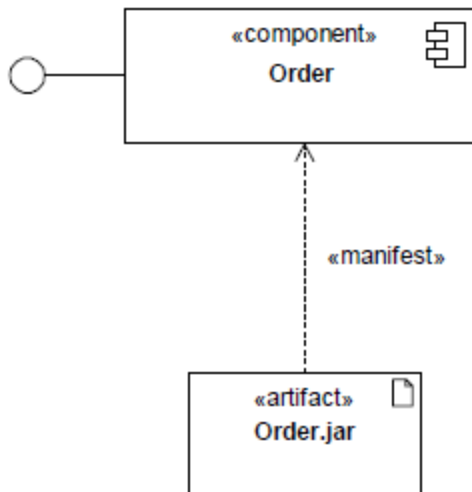


Figure 19.10 A Manifestation relationship between an Artifact and a Component

## 19.4 Nodes

### 19.4.1 Summary

Nodes elaborate and reify the abstract notion of DeploymentTargets. They can be nested and can be connected into systems of arbitrary complexity using communication paths. Typically, Nodes represent either hardware devices or software execution environments.

## 19.4.2 Abstract Syntax

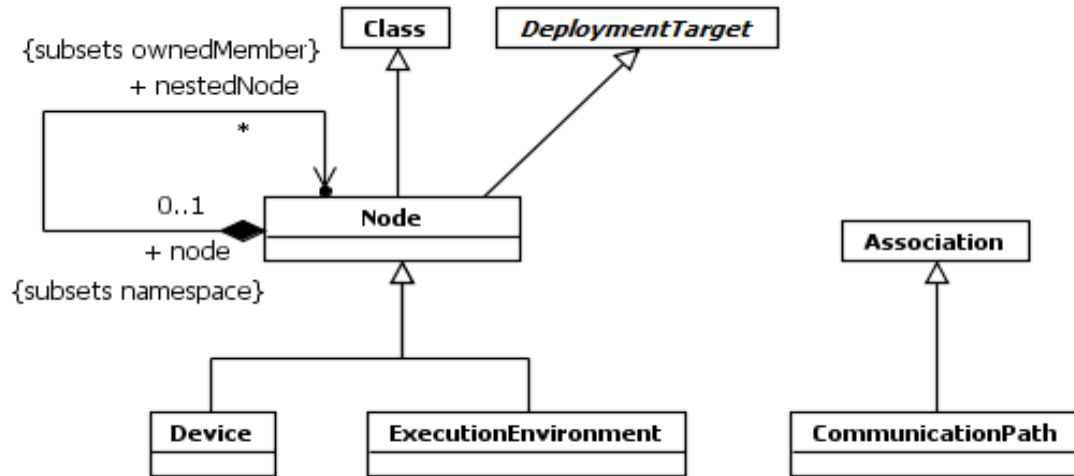


Figure 19.11 Nodes

## 19.4.3 Semantics

A Node is computational resource upon which Artifacts may be deployed, via Deployment relationships, for execution. For advanced modeling applications, Nodes may have complex internal structure defined by nesting and may be interconnected to represent specific situations. The internal structure of Nodes can only consist of other Nodes. Besides participating in Deployments, Nodes acquire a set of associated elements derived from the Manifestation relationships of the Artifacts deployed on them.

Nodes may be further sub-typed as Devices and ExecutionEnvironments. Devices represent physical machine components. ExecutionEnvironments represent standard software systems that application components may require at execution time. Specific profiles might, for example, define stereotypes for ExecutionEnvironments such as «OS», «workflow engine», «database system», and «J2EE container».

A Device is a physical computational resource with processing capability upon which Artifacts may be deployed for execution. Devices may be complex (i.e., they may consist of other devices) either through namespace ownership or through attributes that are themselves typed by Devices. Entire physical computing systems may be decomposed into their constituents in this way. Examples of Devices might include «application server», «client workstation», «mobile device», and «embedded device».

Typically, ExecutionEnvironments are assigned to some, often higher level, Device or general system Node via the composition relationship defined on Node. ExecutionEnvironments can be nested (for example, a database ExecutionEnvironment might be nested in an operating system ExecutionEnvironment). ExecutionEnvironment may have explicit interfaces for system level services that can be called by the deployed elements. In such cases, software ExecutionEnvironment services should be explicitly modeled. Application Components of the appropriate type are then deployed, with a Deployment relationship, to specific ExecutionEnvironment nodes or the Manifestations relationships of DeployedArtifacts. For each component Deployment, aspects of these services may be determined by properties in a DeploymentSpecification for a particular kind of ExecutionEnvironment.

Nodes can be connected to represent specific network topologies using CommunicationPaths defining specific connections between Node instances. A CommunicationPath is an Association between two DeploymentTargets, through which they may exchange Signals and Messages.

## 19.4.4 Notation

A Node is depicted using the same graphic representation as a DeployedTarget.

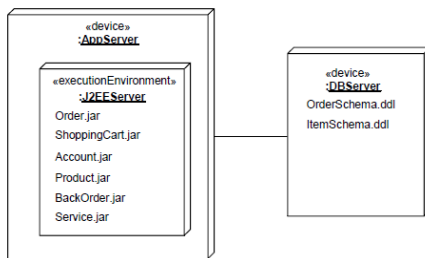
A Device is shown as a Node graphic with the keyword «device».

An ExecutionEnvironment is notated by a Node annotated with the keyword «executionEnvironment».

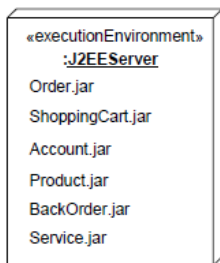
CommunicationPaths between Nodes are depicted using the same as normal Association links.

Dashed arrows with the keyword «deploy» show the capability of a Node to support an externally depicted DeployedArtifact. Alternatively, this may be shown by nesting DeployedArtifact graphics inside Node symbols.

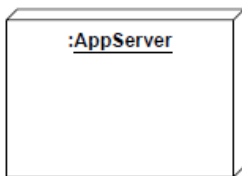
## 19.4.5 Examples



**Figure 19.12** Notation for a Device containing an ExecutionEnvironment and connected to another Device by a CommunicationPath link.



**Figure 19.13** Notation for a ExecutionEnvironment.



**Figure 19.14** An instance of a Node

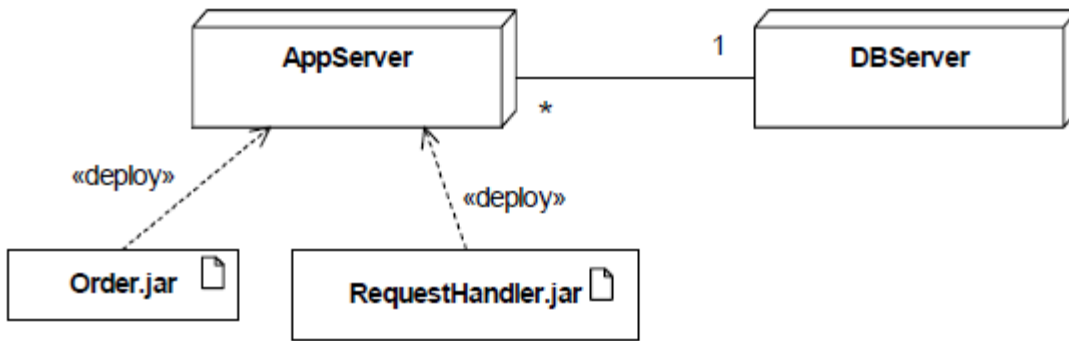


Figure 19.15 CommunicationPath between AppServer with deployed Artifacts and a DBServer.

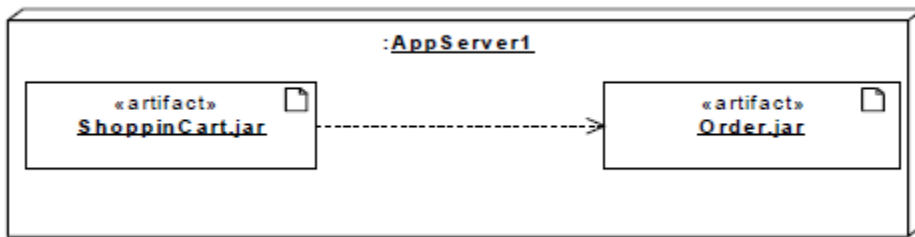


Figure 19.16 Deployed component Artifacts on a Node.



## 19.5 Classifier Descriptions

### Artifact [Class]

#### Description

An artifact is the specification of a physical piece of information that is used or produced by a software development process, or by deployment and operation of a system. Examples of artifacts include model files, source files, scripts, and binary executable files, a table in a database system, a development deliverable, or a word-processing document, a mail message. An artifact is the source of a deployment to a node.

#### Diagrams

[Deployments](#), [Artifacts](#)

#### Generalizations

[Classifier](#), [DeployedArtifact](#)

#### Specializations

[DeploymentSpecification](#)

#### Attributes

- fileName : [String](#) [0..1]  
A concrete name that is used to refer to the Artifact in a physical context. Example: file system name, universal resource locator.

#### Association Ends

- ♦ manifestation : [Manifestation](#) [0..\*]{subsets [Element::ownedElement](#), subsets [NamedElement::clientDependency](#)} (opposite [A\\_manifestation\\_artifact::artifact](#))  
The set of model elements that are manifested in the Artifact. That is, these model elements are utilized in the construction (or generation) of the artifact.
- ♦ nestedArtifact : [Artifact](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [A\\_nestedArtifact\\_artifact::artifact](#))  
The Artifacts that are defined (nested) within the Artifact. The association is a specialization of the ownedMember association from Namespace to NamedElement.
- ♦ ownedAttribute : [Property](#) [0..\*]{ordered, subsets [Classifier::attribute](#), subsets [Namespace::ownedMember](#)} (opposite [A\\_ownedAttribute\\_artifact::artifact](#))  
The attributes or association ends defined for the Artifact. The association is a specialization of the ownedMember association.
- ♦ ownedOperation : [Operation](#) [0..\*]{ordered, subsets [Classifier::feature](#), subsets [A\\_redefinitionContext\\_redefinableElement::redefinableElement](#), subsets [Namespace::ownedMember](#)} (opposite [A\\_ownedOperation\\_artifact::artifact](#))  
The Operations defined for the Artifact. The association is a specialization of the ownedMember association.

## CommunicationPath [Class]

### Description

A communication path is an association between two deployment targets, through which they are able to exchange signals and messages.

### Diagrams

[Nodes](#)

### Generalizations

[Association](#)

### Constraints

- `association_ends`  
The association ends of a CommunicationPath are typed by DeploymentTargets.

```
inv: endType->forAll (oclIsKindOf(DeploymentTarget))
```

## DeployedArtifact [Abstract Class]

### Description

A deployed artifact is an artifact or artifact instance that has been deployed to a deployment target.

### Diagrams

[Deployments](#), [Artifacts](#), [Instances](#)

### Generalizations

[NamedElement](#)

### Specializations

[Artifact](#), [InstanceSpecification](#)

## Deployment [Class]

### Description

A deployment is the allocation of an artifact or artifact instance to a deployment target. A component deployment is the deployment of one or more artifacts or artifact instances to a deployment target, optionally parameterized by a deployment specification. Examples are executables and configuration files.

### Diagrams

[Deployments](#)

## Generalizations

[Dependency](#)

## Association Ends

- ♦ configuration : [DeploymentSpecification](#) [0..\*]{subsets [Element::ownedElement](#)} (opposite [DeploymentSpecification::deployment](#))  
The specification of properties that parameterize the deployment and execution of one or more Artifacts.
- deployedArtifact : [DeployedArtifact](#) [0..\*]{subsets [Dependency::supplier](#)} (opposite [A\\_deployedArtifact\\_deploymentForArtifact::deploymentForArtifact](#))  
The Artifacts that are deployed onto a Node. This association specializes the supplier association.
- location : [DeploymentTarget](#) [1..1]{subsets [Dependency::client](#), subsets [Element::owner](#)} (opposite [DeploymentTarget::deployment](#))  
The DeployedTarget which is the target of a Deployment.

## DeploymentSpecification [Class]

### Description

A deployment specification specifies a set of properties that determine execution parameters of a component artifact that is deployed on a node. A deployment specification can be aimed at a specific type of container. An artifact that reifies or implements deployment specification properties is a deployment descriptor.

### Diagrams

[Deployments](#)

## Generalizations

[Artifact](#)

## Attributes

- deploymentLocation : [String](#) [0..1]  
The location where an Artifact is deployed onto a Node. This is typically a 'directory' or 'memory address.'
- executionLocation : [String](#) [0..1]  
The location where a component Artifact executes. This may be a local or remote location.

## Association Ends

- deployment : [Deployment](#) [0..1]{subsets [Element::owner](#)} (opposite [Deployment::configuration](#))  
The deployment with which the DeploymentSpecification is associated.

## Constraints

- `deployment_target`  
The `DeploymentTarget` of a `DeploymentSpecification` is a kind of `ExecutionEnvironment`.

```
inv: deployment->forall (location.ocIsKindOf(ExecutionEnvironment))
```

- `deployed_elements`  
The `deployedElements` of a `DeploymentTarget` that are involved in a `Deployment` that has an associated `DeploymentSpecification` is a kind of `Component` (i.e., the configured components).

```
inv: deployment->forall (location.deployedElement->forall (ocIsKindOf(Component)))
```

## DeploymentTarget [Abstract Class]

### Description

A deployment target is the location for a deployed artifact.

### Diagrams

[Deployments](#), [Nodes](#), [Properties](#), [Instances](#)

### Generalizations

[NamedElement](#)

### Specializations

[Node](#), [InstanceSpecification](#), [Property](#)

### Association Ends

- `/deployedElement` : [PackageableElement](#) [0..\*]{ } (opposite [A\\_deployedElement\\_deploymentTarget::deploymentTarget](#))  
The set of elements that are manifested in an Artifact that is involved in Deployment to a DeploymentTarget.
- ♦ `deployment` : [Deployment](#) [0..\*]{ subsets [Element::ownedElement](#), subsets [NamedElement::clientDependency](#) } (opposite [Deployment::location](#))  
The set of Deployments for a DeploymentTarget.

### Operations

- `deployedElement()` : [PackageableElement](#) [0..\*]  
Derivation for `DeploymentTarget::/deployedElement`

```
body: deployment.deployedArtifact->select(ocIsKindOf(Artifact))->collect(oclAsType(Artifact).manifestation)->collect(utilizedElement)->asSet()
```

## Device [Class]

### Description

A device is a physical computational resource with processing capability upon which artifacts may be deployed for execution. Devices may be complex (i.e., they may consist of other devices).

### Diagrams

[Nodes](#)

### Generalizations

[Node](#)

## ExecutionEnvironment [Class]

### Description

An execution environment is a node that offers an execution environment for specific types of components that are deployed on it in the form of executable artifacts.

### Diagrams

[Nodes](#)

### Generalizations

[Node](#)

## Manifestation [Class]

### Description

A manifestation is the concrete physical rendering of one or more model elements by an artifact.

### Diagrams

[Artifacts](#)

### Generalizations

[Abstraction](#)

### Association Ends

- utilizedElement : [PackageableElement](#) [1..1]{subsets [Dependency::supplier](#)} (opposite [A\\_utilizedElement\\_manifestation::manifestation](#))  
The model element that is utilized in the manifestation in an Artifact.

## Node [Class]

### Description

A Node is computational resource upon which artifacts may be deployed for execution. Nodes can be interconnected through communication paths to define network structures.

### Diagrams

[Deployments](#), [Nodes](#)

### Generalizations

[Class](#), [DeploymentTarget](#)

### Specializations

[Device](#), [ExecutionEnvironment](#)

### Association Ends

- ◆ nestedNode : [Node](#) [0..\*]{subsets [Namespace::ownedMember](#)} (opposite [A\\_nestedNode\\_node::node](#))  
The Nodes that are defined (nested) within the Node.

### Constraints

- internal\_structure  
The internal structure of a Node (if defined) consists solely of parts of type Node.

```
inv: part->forAll(oclIsKindOf(Node))
```

## 19.6 Association Descriptions

### A\_configuration\_deployment [Association]

#### Diagrams

[Deployments](#)

#### Member Ends

- [Deployment::configuration](#)
- [DeploymentSpecification::deployment](#)

### A\_deployedArtifact\_deploymentForArtifact [Association]

#### Diagrams

[Deployments](#)

## Owned Ends

- deploymentForArtifact : [Deployment](#) [0..\*]{subsets [A\\_supplier\\_supplierDependency::supplierDependency](#)} (opposite [Deployment::deployedArtifact](#))

## A\_deployedElement\_deploymentTarget [Association]

### Diagrams

[Deployments](#)

## Owned Ends

- deploymentTarget : [DeploymentTarget](#) [0..\*] (opposite [DeploymentTarget::deployedElement](#))

## A\_deployment\_location [Association]

### Diagrams

[Deployments](#)

## Member Ends

- [DeploymentTarget::deployment](#)
- [Deployment::location](#)

## A\_manifestation\_artifact [Association]

### Diagrams

[Artifacts](#)

## Owned Ends

- artifact : [Artifact](#) [1..1]{subsets [Dependency::client](#), subsets [Element::owner](#)} (opposite [Artifact::manifestation](#))

## A\_nestedArtifact\_artifact [Association]

### Diagrams

[Artifacts](#)

## Owned Ends

- artifact : [Artifact](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [Artifact::nestedArtifact](#))

## A\_nestedNode\_node [Association]

### Diagrams

[Nodes](#)

## Owned Ends

- node : [Node](#) [0..1]{subsets [NamedElement::namespace](#)} (opposite [Node::nestedNode](#))

## A\_ownedAttribute\_artifact [Association]

### Diagrams

[Artifacts](#)

## Owned Ends

- artifact : [Artifact](#) [0..1]{subsets [NamedElement::namespace](#), subsets [A\\_attribute\\_classifier::classifier](#)} (opposite [Artifact::ownedAttribute](#))

## A\_ownedOperation\_artifact [Association]

### Diagrams

[Artifacts](#)

## Owned Ends

- artifact : [Artifact](#) [0..1]{subsets [Feature::featuringClassifier](#), subsets [NamedElement::namespace](#), subsets [RedefinableElement::redefinitionContext](#)} (opposite [Artifact::ownedOperation](#))

## A\_utilizedElement\_manifestation [Association]

### Diagrams

[Artifacts](#)



## Owned Ends

- manifestation : [Manifestation](#) [0..\*]{subsets [A\\_supplier\\_supplierDependency::supplierDependency](#)} (opposite [Manifestation::utilizedElement](#))

# 20 InformationFlows

## 20.1 Information Flows

### 20.1.1 Summary

The InformationFlows package supports exchange of information between system entities at high levels of abstraction. InformationFlows may be useful during top-down model development, representing aspects of models not yet fully specified, and for recording less detailed, heuristic representations of more complex model areas. In these ways, InformationFlows can help to clarify and document overall understanding of the intent of large or complicated models.

InformationFlows describe circulation of information through a system in a general manner. They do not specify the nature of the information, mechanisms by which it is conveyed, sequences of exchange, or any control conditions. During more detailed modeling, representation and realization links may be added to specify which model elements implement an InformationFlow and to show how information is conveyed. Similarly, InformationItems can be used to represent the information that flows along InformationFlows even before details of their realization have been designed.

The contents of the InformationFlows package are shown in Figure 20.1.

### 20.1.2 Abstract Syntax

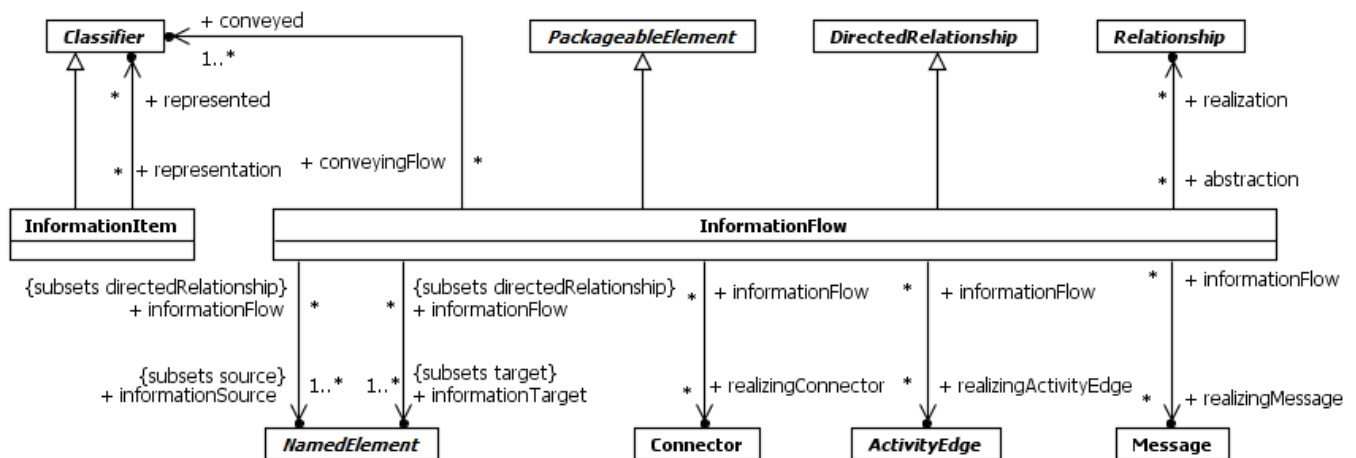


Figure 20.1 Information Flows

### 20.1.3 Semantics

InformationFlows require some kind of “information channel” for unidirectional transmission of information items from sources to targets. They specify the information channel’s realizations, if any, and identify the information that flows along them. Information moving along the information channel may be represented by abstract InformationItems and by concrete Classifiers.

The sources and targets of an InformationFlow designate sets of objects that can send (sources) or receive (targets) conveyed InformationItems or Classifiers. In Figure 20.1, sources and targets are shown as NamedElements. In practice, a constraint on InformationFlow requires that sources and targets must of one of the following types: Actor, Node, UseCase, Artifact, Class, Component, Port, Property, Interface, Package, ActivityNode, ActivityPartition, and InstanceSpecification. Furthermore, when a

source or target is an InstanceSpecification, it cannot be a link (i.e., the InstanceSpecification's typing Classifier cannot represent a Relationship).

An InformationFlow's sources and targets represent all potential instances typed or contained (i.e., owned) by them. For example, if a source or target is a

- Classifier, it represents all potential instances of the Classifier.
- Part, it represents all instances that can play the role specified by the Part.
- Package, it represents all potential instances of the directly or indirectly owned Classifiers in the Package.

The information channel can be realized in various ways depending on the nature of sources and targets. As shown in Figure 20.1, information channels can be realized by Relationships, Connectors, ActivityEdges, and Messages. The types of sources and targets must be compatible with the types of information that flow along the information channel. For example, if the source and target are parts of some composite structure such as a Collaboration, the information channel is likely to be represented by a Connector between them. Or, if the source and target are objects (which are kinds of InstanceSpecifications), the information channel may be represented by a link that joins them. Multiple sources and targets are allowed, but they must have compatible types.

Typically, InformationFlows identify InformationItems flowing from sources to targets. However, concrete Classifiers, such as Class, may be conveyed as well.

InformationItems represent many kinds of information that can flow from sources to targets in very abstract ways. They represent the kinds of information that may move within a system, but do not elaborate details of the transferred information. Details of transferred information are the province of other Classifiers that may ultimately define InformationItems. Consequently, InformationItems cannot be instantiated and do not themselves have features, generalizations, or associations. In this respect, InformationItems are similar to Interfaces -- a constraint in the metamodel enforces the inability to instantiate them.

An important use of InformationItems is to represent information during early design stages, possibly before the detailed modeling decisions that will ultimately define them have been made. Another purpose of InformationItems is to abstract portions of complex models in less precise, but perhaps more general and communicable, ways.

InformationItems may be decomposed into more specific InformationItems or Classifiers. Representation links between them express the idea that, in the context a particular InformationFlow, specific information is exchanged.

InformationItems may only be realized by Classes (of all kinds, including Components, AssociationClasses, Behaviors, etc.), Interfaces, Signals, and other InformationItems.

The principal goal of InformationFlows is to convey that information moves from one area of a model to another. Consequently, the metamodel is intentionally very permissive about the realization of information channels and the types of information that can flow along them. InformationFlows can be simultaneously realized by multiple types of information channels, and InformationItems can represent many different varieties of Classifiers.

#### 20.1.4 Notation

An InformationFlow is represented using the same notation as Dependency, with the keyword «flow» adorning its dashed line.

Representation of InformationItems is determined by the context in which they are displayed:

- When attached to the dashed «flow» lines of an InformationFlow, the InformationItem's name is displayed close to the appropriate «flow» line.
- When displayed independently of their InformationFlows, InformationItems may be represented, because they are Classifiers, as names inside rectangles. The rectangle is adorned with the «information» keyword or with a solid, black-

filled isosceles triangle; in this usage, the triangle has no directional significance. Because they have neither attributes nor operations, InformationItem rectangles are shown without visible subcompartments.

- When attached to a realization of an InformationFlow’s information channel, a black-filled isosceles triangle on the information channel indicates the direction of information flow. The InformationItem’s name is placed close to the triangle. When representing several InformationItems having the same direction, only one triangle is shown, with a nearby list of InformationItem names separated by commas.
- When an InformationItem represents other InformationItems or Classifiers, they are connected with dashed line arrows adorned with the keyword «representation».

### 20.1.5 Examples

Figure 20.2 shows information about products and wages (InformationItems) flowing from a Company to its Customers and Employees along two separate InformationFlows. The information channels realizing these flows are not depicted.

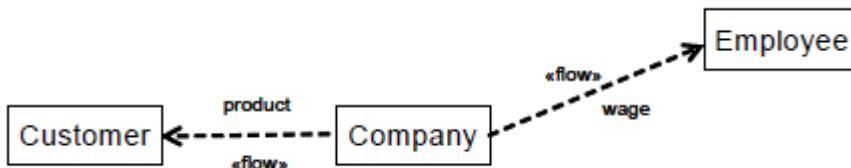


Figure 20.2 Example of InformationFlows conveying InformationItems

The wage InformationItem is represented independently of any InformationFlow in Figure 20.3. The two representations are equivalent. When used in this way, the black triangle merely identifies the box as an InformationItem; it has no directional flow meaning.



Figure 20.3 Information Item represented as a classifier

InformationItems can represent other InformationItems or concrete Classifiers. The travel document InformationItem represents both passports and plane tickets (themselves InformationItems), whereas the Wage InformationItem acts as a “stand-in” for both the concrete Classes Salary and Bonus (Figure 20.4).



Figure 20.4 Examples of «representation» notation

When InformationItems are displayed on an information channel realizing an InformationFlow, triangles indicate the directional flow of information. In Figure 20.5, InformationItems adorn a Connector between m1:myC1 and m2:myC2. InformationItem “a”

moves from source m2:myC to target m1:myC1, whereas InformationItems “b” and “d” move from source m1:myC1 to target m2:myC. In the latter, only one triangle is used to indicate flow of the multiple, named InformationItems. At least two InformationFlows are required to describe this situation – one for “a” and one for “b” and “d”. Alternately, “b” and “d” might be described by separate InformationFlows, bringing the total to three.



Figure 20.5 InformationItems attached to Connectors

In Figure 20.6, the InformationItems product and wage are attached to Associations realizing the information channels of their respective InformationFlows. Here, again, the triangles indicate direction of information flow.

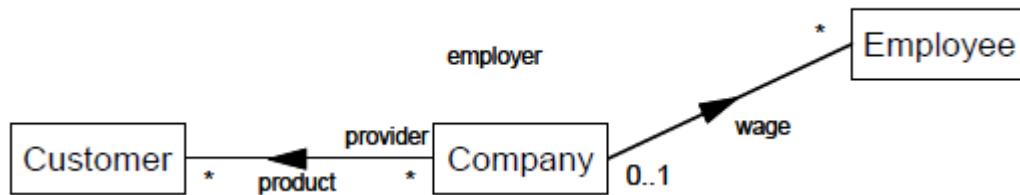


Figure 20.6 InformationItems attached to Associations

## 20.2 Classifier Descriptions

### InformationFlow [Class]

#### Description

InformationFlows describe circulation of information through a system in a general manner. They do not specify the nature of the information, mechanisms by which it is conveyed, sequences of exchange or any control conditions. During more detailed modeling, representation and realization links may be added to specify which model elements implement an InformationFlow and to show how information is conveyed. InformationFlows require some kind of “information channel” for unidirectional transmission of information items from sources to targets. They specify the information channel’s realizations, if any, and identify the information that flows along them. Information moving along the information channel may be represented by abstract InformationItems and by concrete Classifiers.

#### Diagrams

[Information Flows](#)

#### Generalizations

[DirectedRelationship](#), [PackageableElement](#)

#### Association Ends

- conveyed : [Classifier](#) [1..\*] (opposite [A\\_conveyed\\_conveyingFlow::conveyingFlow](#))  
Specifies the information items that may circulate on this information flow.
- informationSource : [NamedElement](#) [1..\*]{subsets [DirectedRelationship::source](#)} (opposite [A\\_informationSource\\_informationFlow::informationFlow](#))  
Defines from which source the conveyed InformationItems are initiated.
- informationTarget : [NamedElement](#) [1..\*]{subsets [DirectedRelationship::target](#)} (opposite [A\\_informationTarget\\_informationFlow::informationFlow](#))  
Defines to which target the conveyed InformationItems are directed.
- realization : [Relationship](#) [0..\*] (opposite [A\\_realization\\_abstraction\\_flow::abstraction](#))  
Determines which Relationship will realize the specified flow.
- realizingActivityEdge : [ActivityEdge](#) [0..\*] (opposite [A\\_realizingActivityEdge\\_informationFlow::informationFlow](#))  
Determines which ActivityEdges will realize the specified flow.
- realizingConnector : [Connector](#) [0..\*] (opposite [A\\_realizingConnector\\_informationFlow::informationFlow](#))  
Determines which Connectors will realize the specified flow.
- realizingMessage : [Message](#) [0..\*] (opposite [A\\_realizingMessage\\_informationFlow::informationFlow](#))  
Determines which Messages will realize the specified flow.

## Constraints

- **must\_conform**  
The sources and targets of the information flow must conform to the sources and targets or conversely the targets and sources of the realization relationships.

Cannot be expressed in OCL

- **sources\_and\_targets\_kind**  
The sources and targets of the information flow can only be one of the following kind: Actor, Node, UseCase, Artifact, Class, Component, Port, Property, Interface, Package, ActivityNode, ActivityPartition and InstanceSpecification except when its classifier is a relationship (i.e., it represents a link).

```
inv: (self.informationSource->forAll(oclIsKindOf(Actor) or oclIsKindOf(Node) or
oclIsKindOf(UseCase) or oclIsKindOf(Artifact) or oclIsKindOf(Class) or
oclIsKindOf(Component) or oclIsKindOf(Port) or oclIsKindOf(Property) or
oclIsKindOf(Interface) or oclIsKindOf(Package) or oclIsKindOf(ActivityNode) or
oclIsKindOf(ActivityPartition) or oclIsKindOf(InstanceSpecification))) and
(self.informationTarget->forAll(oclIsKindOf(Actor) or oclIsKindOf(Node) or
oclIsKindOf(UseCase) or oclIsKindOf(Artifact) or oclIsKindOf(Class) or
oclIsKindOf(Component) or oclIsKindOf(Port) or oclIsKindOf(Property) or
oclIsKindOf(Interface) or oclIsKindOf(Package) or oclIsKindOf(ActivityNode) or
oclIsKindOf(ActivityPartition) or oclIsKindOf(InstanceSpecification)))
```

- **convey\_classifiers**  
An information flow can only convey classifiers that are allowed to represent an information item.

```
inv: self.conveyed->forAll(oclIsKindOf(Class) or oclIsKindOf(Interface)
or oclIsKindOf(InformationItem) or oclIsKindOf(Signal) or oclIsKindOf(Component))
```

## InformationItem [Class]

### Description

InformationItems represent many kinds of information that can flow from sources to targets in very abstract ways. They represent the kinds of information that may move within a system, but do not elaborate details of the transferred information. Details of transferred information are the province of other Classifiers that may ultimately define InformationItems. Consequently, InformationItems cannot be instantiated and do not themselves have features, generalizations, or associations. An important use of InformationItems is to represent information during early design stages, possibly before the detailed modeling decisions that will ultimately define them have been made. Another purpose of InformationItems is to abstract portions of complex models in less precise, but perhaps more general and communicable, ways.

### Diagrams

[Information Flows](#)

### Generalizations

[Classifier](#)

### Association Ends

- **represented** : [Classifier](#) [0..\*] (opposite [A represented representation::representation](#))  
Determines the classifiers that will specify the structure and nature of the information. An information item represents all

its represented classifiers.

## Constraints

- **sources\_and\_targets**  
The sources and targets of an information item (its related information flows) must designate subsets of the sources and targets of the representation information item, if any. The Classifiers that can realize an information item can only be of the following kind: Class, Interface, InformationItem, Signal, Component.

```
inv: (self.represented->select(oclIsKindOf(InformationItem))->forall(p |
  p.conveyingFlow.source->forall(q | self.conveyingFlow.source->includes(q)) and
  p.conveyingFlow.target->forall(q | self.conveyingFlow.target->includes(q)))) and
  (self.represented->forall(oclIsKindOf(Class) or oclIsKindOf(Interface) or
  oclIsKindOf(InformationItem) or oclIsKindOf(Signal) or oclIsKindOf(Component)))
```

- **has\_no**  
An informationItem has no feature, no generalization, and no associations.

```
inv: self.generalization->isEmpty() and self.feature->isEmpty()
```

- **not\_instantiable**  
It is not instantiable.

```
inv: isAbstract
```

## 20.3 Association Descriptions

### A\_conveyed\_conveyingFlow [Association]

#### Diagrams

[Information Flows](#)

#### Owned Ends

- conveyingFlow : [InformationFlow](#) [0..\*] (opposite [InformationFlow::conveyed](#))

### A\_informationSource\_informationFlow [Association]

#### Diagrams

[Information Flows](#)

#### Owned Ends

- informationFlow : [InformationFlow](#) [0..\*]{subsets [A\\_source\\_directedRelationship::directedRelationship](#)} (opposite [InformationFlow::informationSource](#))



## A\_informationTarget\_informationFlow [Association]

### Diagrams

[Information Flows](#)

### Owned Ends

- informationFlow : [InformationFlow](#) [0..\*]{subsets [A\\_target\\_directedRelationship::directedRelationship](#)} (opposite [InformationFlow::informationTarget](#))

## A\_realization\_abstraction\_flow [Association]

### Diagrams

[Information Flows](#)

### Owned Ends

- abstraction : [InformationFlow](#) [0..\*] (opposite [InformationFlow::realization](#))

## A\_realizingActivityEdge\_informationFlow [Association]

### Diagrams

[Information Flows](#)

### Owned Ends

- informationFlow : [InformationFlow](#) [0..\*] (opposite [InformationFlow::realizingActivityEdge](#))

## A\_realizingConnector\_informationFlow [Association]

### Diagrams

[Information Flows](#)

### Owned Ends

- informationFlow : [InformationFlow](#) [0..\*] (opposite [InformationFlow::realizingConnector](#))

### A\_realizingMessage\_informationFlow [Association]

#### Diagrams

[Information Flows](#)

### Owned Ends

- informationFlow : [InformationFlow](#) [0..\*] (opposite [InformationFlow::realizingMessage](#))

### A\_represented\_representation [Association]

#### Diagrams

[Information Flows](#)

### Owned Ends

- representation : [InformationItem](#) [0..\*] (opposite [InformationItem::represented](#))

# 21 Primitive Types

## 21.1 Summary

The PrimitiveTypes package is an independent package that defines a set of reusable PrimitiveTypes that are commonly used in the definition of metamodels. The UML metamodel uses the PrimitiveTypes package.



Figure 21.1 Primitive Types

## 21.2 Semantics

Table 21.1 PrimitiveType semantics

Integer	An instance of Integer is a value in the (infinite) set of integers (...-2, -1, 0, 1, 2...).
Boolean	An instance of Boolean is one of the predefined values <i>true</i> and <i>false</i> .
String	An instance of String defines a sequence of characters. Character sets may include non-Roman alphabets. The semantics of the string itself depends on its purpose; it can be a comment, computational language expression, OCL expression, etc.
UnlimitedNatural	An instance of UnlimitedNatural is a value in the (infinite) set of natural numbers (0, 1, 2...) plus <i>unbounded</i> . The value of <i>unbounded</i> is shown using an asterisk (*). UnlimitedNatural values are typically used to denote the upper bound of a range, such as a multiplicity; <i>unbounded</i> is used whenever the range is specified to have no upper bound.
Real	An instance of Real is a value in the (infinite) set of real numbers. Typically an implementation will represent Real numbers using a floating point standard such as IEEE 754.

## 21.3 Notation

There is no notation for PrimitiveTypes. There is notation for literal values of PrimitiveTypes; this notation is covered in [Clause 8.2](#).

## 21.4 Examples

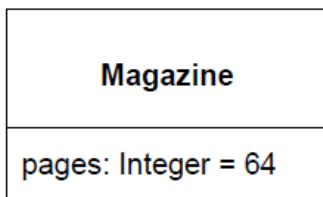


Figure 21.2 An Integer used as a type for an attribute, with a default value

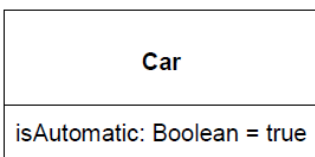


Figure 21.3 A Boolean used as a type for an attribute, with a default value

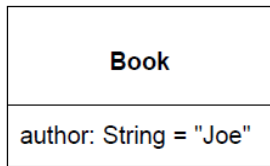


Figure 21.4 A String used as a type for an attribute, with a default value

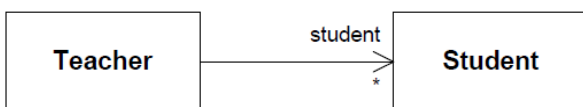


Figure 21.5 An UnlimitedNatural used as an upper bound for a multiplicity

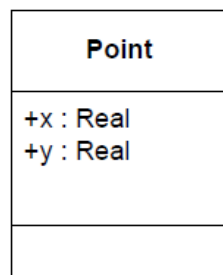


Figure 21.6 Two attributes with type Real

# 22 Standard Profile

## 22.1 Summary

The Standard Profile specifies a set of predefined standard stereotypes. A conforming tool shall support all of the stereotypes in the Standard Profile.

## 22.2 Model

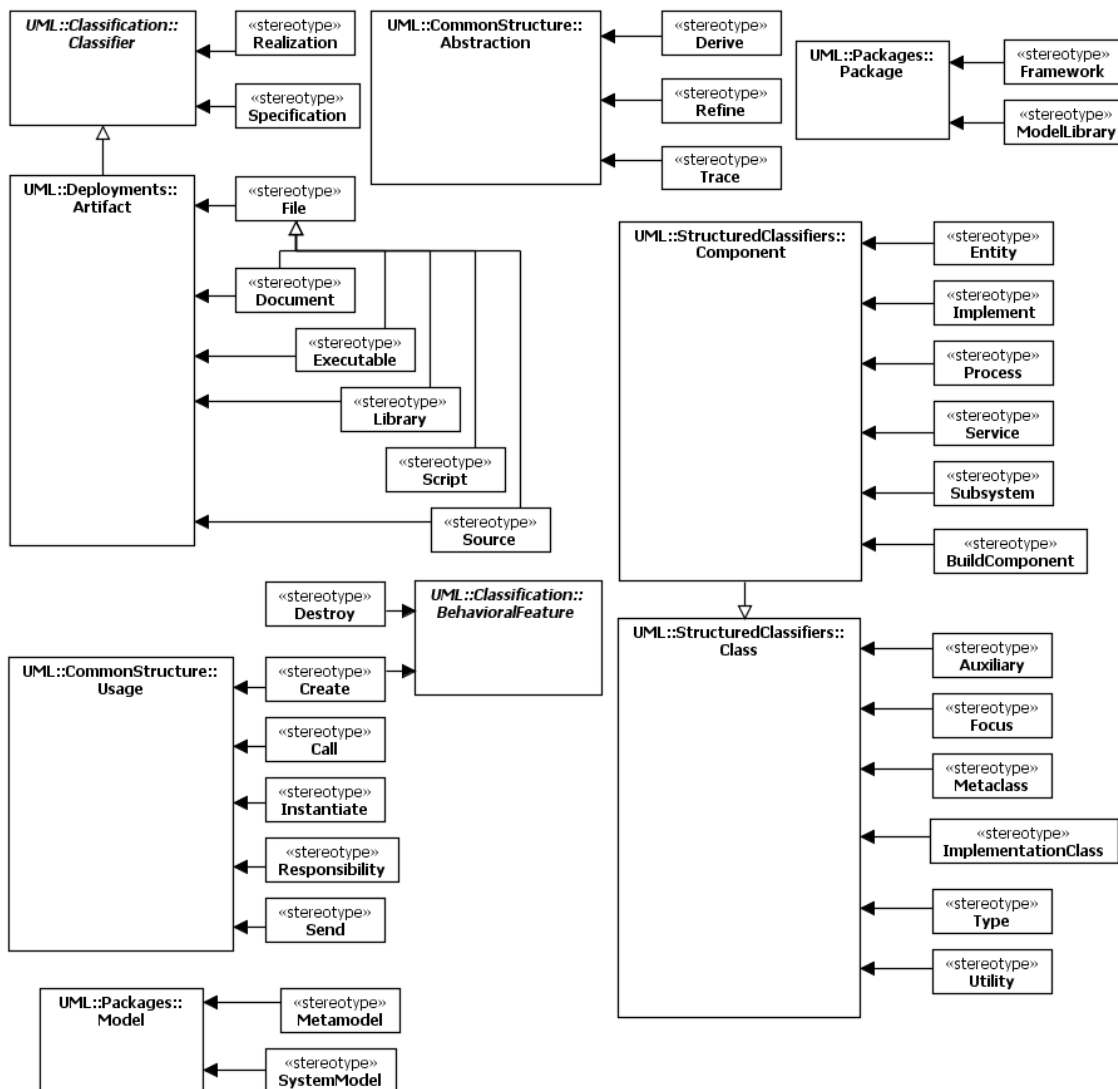


Figure 22.1 Model of StandardProfile

## 22.3 Standard Stereotypes

The stereotypes belonging to the profile are described using a tabular form. The first column gives the name of the stereotype label corresponding to the stereotype. The actual name of the stereotype is the same as the stereotype label. The second column identifies the metaclass to which the stereotype applies and the last column provides a description of the meaning of the stereotype.

Name	Applies to	Description
«Auxiliary»	Class	A class that supports another more central or fundamental class, typically by implementing secondary logic or control flow. The class that the auxiliary supports may be defined explicitly using a Focus class or implicitly by a dependency relationship. 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».
«BuildComponent»	Component	A collection of elements defined for the purpose of system level development activities, such as compilation and versioning.
«Call»	Usage	A usage dependency whose source is an operation and whose target is an operation. The relationship may also be subsumed to the class containing an operation, with the meaning that there exists an operation in the class to which the dependency applies. A call dependency specifies that the source operation or an operation in the source class invokes the target operation or an operation in the target class. A call dependency may connect a source operation to any target operation that is within scope including, but not limited to, operations of the enclosing classifier and operations of other visible classifiers.
«Create»	Usage	A usage dependency denoting that the client classifier creates instances of the supplier classifier.
«Create»	BehavioralFeature	Specifies that the designated feature creates an instance of the classifier to which the feature is attached. May be promoted to the Classifier containing the feature.
«Derive»	Abstraction	Specifies a derivation relationship among model elements that are usually, but not necessarily, of the same type. A derived dependency specifies that the client may be computed from the supplier. The mapping specifies the computation. The client may be implemented for design reasons, such as efficiency, even though it is logically redundant.
«Destroy»	BehavioralFeature	Specifies that the designated feature destroys an instance of the classifier to which the feature is attached. May be promoted to the classifier containing the feature.
«Document»	Artifact	A generic file that is not a «Source» file or «Executable». Subclass of «File».
«Entity»	Component	A persistent information component representing a business concept.
«Executable»	Artifact	A program file that can be executed on a computer system. Subclass of «File».
«File»	Artifact	A physical file in the context of the system developed.
«Focus»	Class	A class that defines the core logic or control flow for one or more auxiliary classes that support it. Support classes may be defined explicitly using Auxiliary classes or implicitly by dependency relationships. 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».
«Framework»	Package	A package that contains model elements that 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.

«Implement»	Component	A component definition that is not intended to have a specification itself. Rather, it is an implementation for a separate «Specification» to which it has a Dependency.
«Implementation Class»	Class	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. This is in contrast to Class, for which an instance may have multiple classes at one time and may gain or lose classes over time, and an object (a child of instance) may dynamically have multiple classes. An Implementation class is said to realize a Classifier if it provides all of the operations defined for the Classifier with the same behavior as specified for the Classifier's operations. An Implementation Class may realize a number of different Types. The physical attributes and associations of the Implementation class do not have to be the same as those of any Classifier it realizes and the Implementation Class may provide methods for its operations in terms of its physical attributes and associations. See also: «Type».
«Instantiate»	Usage	A usage dependency among classifiers indicating that operations on the client create instances of the supplier.
«Library»	Artifact	A static or dynamic library file. Subclass of «File».
«Metaclass»	Class	A class whose instances are also classes.
«Metamodel»	Model	A model that specifies the modeling concepts of some modeling language (e.g., a MOF model). See «Metaclass».
«ModelLibrary»	Package	A package that contains model elements that are intended to be reused by other packages. Model libraries are frequently used in conjunction with applied profiles. This is expressed by defining a dependency between a profile and a model library package, or by defining a model library as contained in a profile package. The classes in a model library are not stereotypes and tagged definitions extending the metamodel. A model library is analogous to a class library in some programming languages. When a model library is defined as a part of a profile, it is imported or deleted with the application or removal of the profile. The profile is implicitly applied to its model library. In the other case, when the model library is defined as an external package imported by a profile, the profile requires that the model library be there in the model at the stage of the profile application. The application or the removal of the profile does not affect the presence of the model library elements.
«Process»	Component	A transaction based component.
«Realization»	Classifier	A classifier that specifies a domain of objects and that also defines the physical implementation of those objects. For example, a Component stereotyped by «Realization» will only have realizing Classifiers that implement behavior specified by a separate «Specification» Component. See «Specification». This differs from «Implementation class» because an «Implementation class» is a realization of a Class that can have features such as attributes and methods that are useful to system designers.
«Refine»	Abstraction	Specifies a refinement relationship between model elements at different semantic levels, such as analysis and design. The mapping specifies the relationship between the two elements or sets of elements. The mapping may or may not be computable, and it may be unidirectional or bidirectional. Refinement can be used to model transformations from analysis to design and other such changes.
«Responsibility»	Usage	A contract or an obligation of an element in its relationship to other elements.
«Script»	Artifact	A script file that can be interpreted by a computer system. Subclass of «File».
«Send»	Usage	A usage dependency whose source is an operation and whose target is a signal, specifying that the source sends the target signal.
«Service»	Component	A stateless, functional component (computes a value).

«Source»	Artifact	A source file that can be compiled into an executable file. Subclass of «File».
«Specification»	Classifier	A classifier that specifies a domain of objects without defining the physical implementation of those objects. For example, a Component stereotyped by «Specification» will only have provided and required interfaces, and is not intended to have any realizingClassifiers as part of its definition. This differs from «Type» because a «Type» can have features such as attributes and methods that are useful to analysts modeling systems. Also see: «Realization»
«Subsystem»	Component	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. See also: «Specification» and «Realization».
«SystemModel»	Model	A SystemModel is a stereotyped model that contains a collection of models of the same system. A SystemModel also contains all relationships and constraints between model elements contained in different models.
«Trace»	Abstraction	Specifies a trace relationship between model elements or sets of model elements that represent the same concept in different models. Traces are mainly used for tracking requirements and changes across models. As model changes can occur in both directions, the directionality of the dependency can often be ignored. The mapping specifies the relationship between the two, but it is rarely computable and is usually informal.
«Type»	Class	A class that specifies a domain of objects together with the operations applicable to the objects, without defining the physical implementation of those objects. However, it may have attributes and associations. Behavioral specifications for type operations may be expressed using, for example, activity diagrams. An object may have at most one implementation class, however it may conform to multiple different types. See also: «ImplementationClass».
«Utility»	Class	A class that has no instances, but rather denotes a named collection of non-member attributes and operations, all of which are class-scoped.



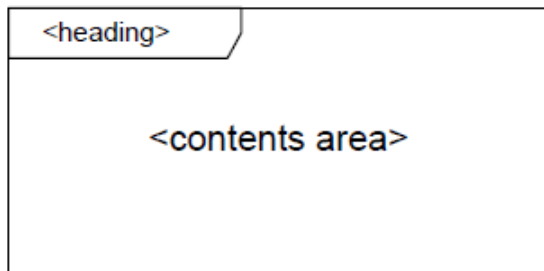
# Annex A: Diagrams

(normative)

This annex describes the general properties of UML diagrams and how they relate to a UML (repository) model and to elements of this. It also introduces the different diagram types of UML.

A UML model consists of elements such as packages, classes, and associations. The corresponding UML diagrams are graphical representations of parts of the UML model. UML diagrams contain graphical elements (nodes connected by paths) that represent elements in the UML model. As an example, two associated classes defined in a package will, in a diagram for the package, be represented by two class symbols and an association path connecting these two class symbols.

Each diagram has a *contents area*. As an option, it may have a *frame* and a *heading* as shown in Figure A. 1.



**Figure A. 1**

The frame is a rectangle. The frame is primarily used in cases where the diagrammed element has graphical border elements, like ports for classes and components (in connection with composite structures), and entry/exit points on statemachines. In cases where not needed, the frame may be omitted and implied by the border of the diagram area provided by a tool. In case the frame is omitted, the heading is also omitted.

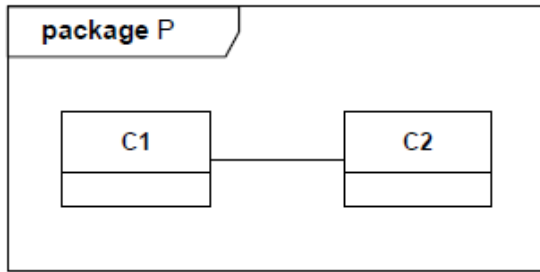
The diagram contents area contains the graphical symbols; the primary graphical symbols define the type of the diagram (e.g., a class diagram is a diagram where the primary symbols in the contents area are class symbols).

The heading is a string contained in a name tag (rectangle with cutoff corner) in the upper leftmost corner of the rectangle, with the following syntax:

`[<kind>]<name>[<parameters>]`

The heading of a diagram represents the kind, name, and parameters of the namespace enclosing or the model element owning elements that are represented by symbols in the contents area. Most elements of a diagram contents area represent model elements that are defined in the namespace or are owned by another model element.

As an example, Figure A. 2 is a class diagram of a package P: classes C1 and C2 are defined in the namespace of the package P.



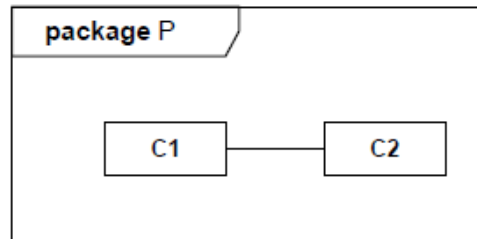
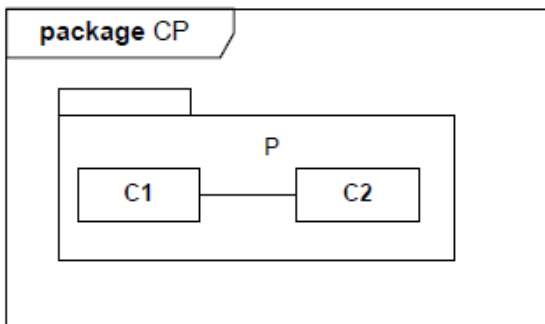
**Figure A. 2 Class diagram of package P**

Figure A. 3 illustrates that a package symbol for package P (in some containing package CP) may show the same contents as the class diagram for the package. i) is a diagram for package CP with graphical symbols representing the fact that the CP package contains a package P. ii) is a class diagram for this package P.

**NOTE.** The package symbol in i) does not have to contain the class symbols and the association symbol; for more realistic models, the package symbols will typically just have the package names, while the class diagrams for the packages will have class symbols for the classes defined in the packages.

i) Package symbol (as part of a larger diagram diagram)

ii) Class diagram for the same package

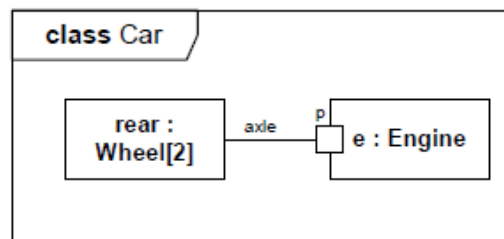
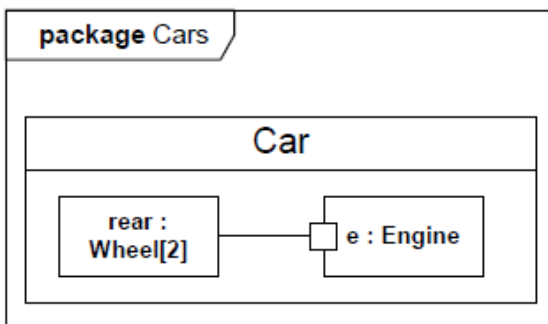


**Figure A. 3 Two diagrams of packages**

In Figure A. 4 i) is a class diagram for the package Cars, with a class symbol representing the fact that the Cars package contains a class Car. ii) is a composite structure diagram for this class Car. The class symbol in i) does not have to contain the structure of the class in a compartment; for more realistic models, the class symbols will typically just have the class names, while the composite structure diagrams for the classes will have symbols for the composite structures.

i) Class symbol for class Car (as part of a larger diagram)

ii) Composite structure diagram for the same class Car



**Figure A. 4 A class diagram and a composite structure diagram**

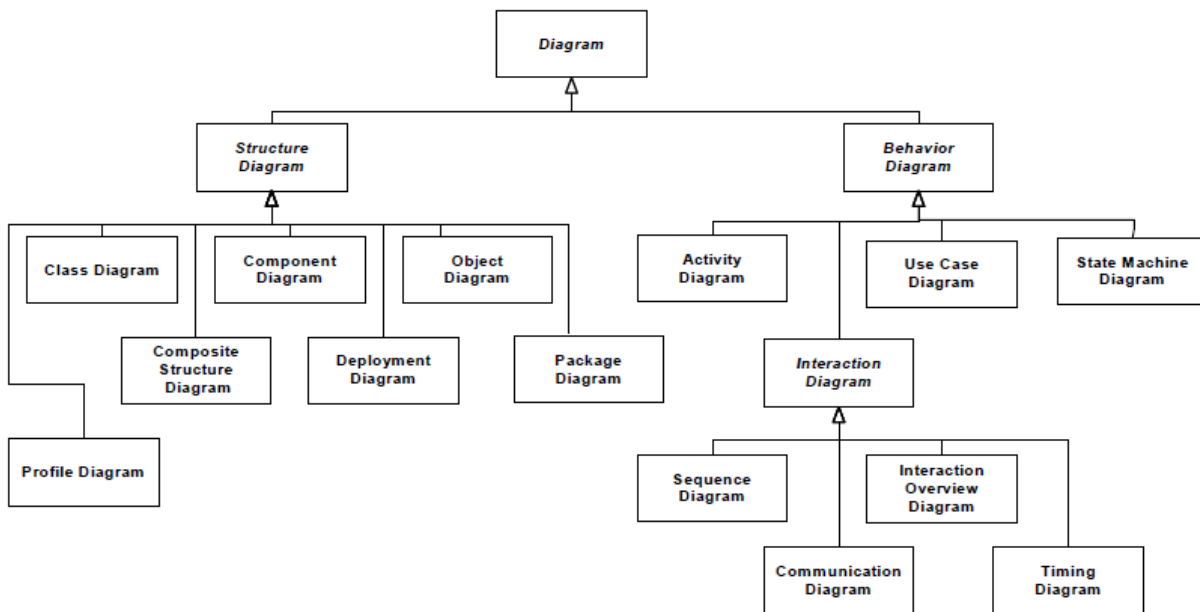
UML diagrams may have the following kinds of frame names as part of the heading:

- activity
- class
- component
- deployment
- interaction
- package
- state machine
- use case

In addition to the long form names for diagram heading types, the following abbreviated forms can also be used:

- **act** (for activity frames)
- **cmp** (for component frames)
- **dep** (for deployment frames)
- **sd** (for interaction frames)
- **pkg** (for package frames)
- **stm** (for state machine frames)
- **uc** (for use case frames)

As is shown in Figure A. 5, there are two major kinds of diagram types: structure diagrams and behavior diagrams.



**Figure A. 5 The taxonomy of structure and behavior diagrams**

Structure diagrams show the static structure of the objects in a system. That is, they depict those elements in a specification that are irrespective of time. The elements in a structure diagram represent the meaningful concepts of an application, and may include abstract, real-world and implementation concepts. For example, a structure diagram for an airline reservation system might include classifiers that represent seat assignment algorithms, tickets, and a credit authorization service. Structure diagrams do not show the details of dynamic behavior, which are illustrated by behavioral diagrams. However, they may show relationships to the behaviors of the classifiers exhibited in the structure diagrams.

Behavior diagrams show the dynamic behavior of the objects in a system, including their methods, collaborations, activities, and state histories. The dynamic behavior of a system can be described as a series of changes to the system over time. Behavior diagrams can be further classified into several other kinds as illustrated in Figure A. 5.

**NOTE.** This taxonomy provides a logical organization for the various major kinds of diagrams. However, it does not preclude mixing different kinds of diagram types, as one might do when one combines structural and behavioral elements (e.g., showing a state machine nested inside an internal structure). Consequently, the boundaries between the various kinds of diagram types are not strictly enforced.

The constructs contained in each of the UML diagrams are described in the clauses indicated below.

- Activity Diagram - Activities
- Class Diagram - Structured Classifiers
- Communication Diagram - Interactions
- Component Diagram - Structured Classifiers
- Composite Structure Diagram - Structured Classifiers
- Deployment diagram - Deployments
- Interaction Overview Diagram - Interactions
- Object Diagram - Classification
- Package Diagram - Packages
- Profile Diagram - Packages
- State Machine Diagram - State Machines
- Sequence Diagram - Interactions
- Timing Diagram - Interactions
- Use Case Diagram - Use Cases

# Annex B: UML Diagram Interchange

(normative)

## B.1 Summary

This Annex enables interchange of the purely graphical aspects of UML models that modelers have control over, such as the position of shapes on a diagram and line routing points (UML DI). This information must be interchanged between tools to reproduce UML diagrams reliably. This annex does not address the graphical aspects of UML models that are determined solely by UML, rather than the modeler, such as most geometric shapes and line styles for the various kinds of model elements. This information is the same for all diagrams conforming to UML, and does not need to be interchanged. Tools can determine which part of UML specifies the uninterchanged graphical aspects by links provided between the interchanged diagram information and the interchanged instances of abstract syntax.

UML DI is based on Diagram Definition (DD, <http://www.omg.org/spec/DD>), specifically its Diagram Interchange (DI) and Diagram Common (DC) packages. As illustrated informally in Figure B.2, UML DI specializes classes and associations from DI, and references elements of the UML abstract syntax to specify diagram interchange for those elements. UML diagrams are interchanged by instantiating classes and associations in UML abstract syntax and in UML DI, then linking them together by reference. Figure B.2 shows an instance of UseCase from UML's abstract syntax on the lower left and an instance of UMLDI's UMLShape referencing it. The shape instance specifies the location of the graphic on the diagram and a label it contains. Note that there are no runtime instances due to UML DI, because diagrams are design time elements only.

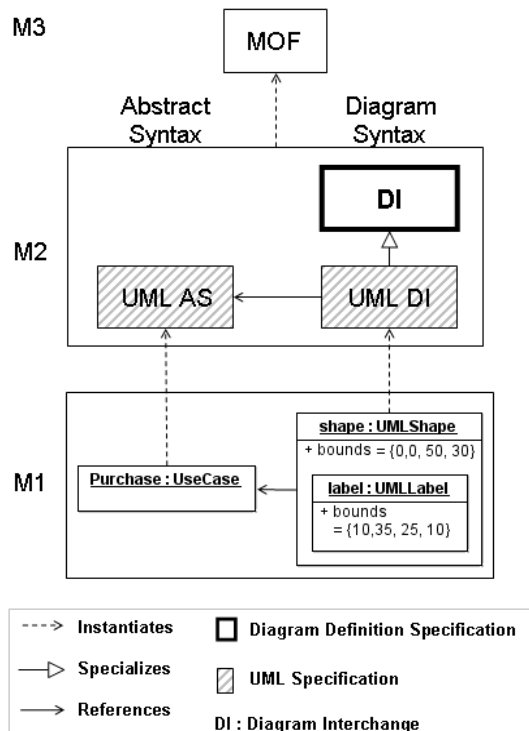


Figure B.2 UML Diagram Interchange Architecture

Receivers of interchange files containing instances of UML DI elements *render* the instances visually as specified by UML. In most cases, the rendering can be determined using only the location and size given by the UML DI instances, the referenced instances of the UML abstract syntax, and the UML specification of notation. In cases where UML gives multiple notations to choose from for the same abstract syntax, UML DI includes additional information to indicate which notation shall be used.

## B.2 Generic

### B.2.1 Summary

The Generic portion of UML DI captures graphical aspects that are common across multiple kinds of UML diagrams.

### B.2.2 UML Diagrams and Diagram Elements

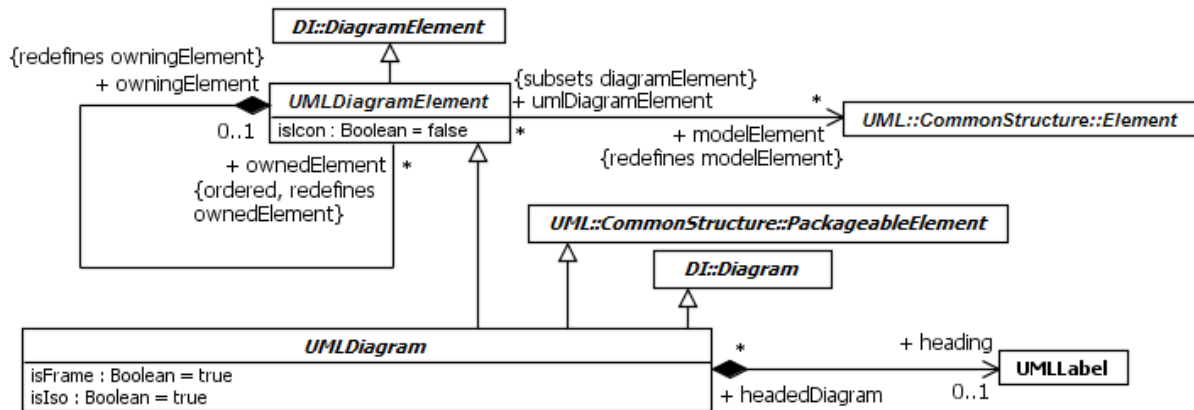


Figure B.3 UML Diagrams and Diagram Elements

The model in Figure B.3 specializes DI's Diagram and DiagramElement into UMLDiagram and UMLDiagramElement, respectively, to make UMLDiagram concrete, add properties and associations, and redefine inherited properties.

UMLDiagramElement is the most general class for UML DI. It redefines modelElement from DI's DiagramElement to restrict UMLDiagramElements to be notation for UML Elements, rather than other language elements. It also redefines ownedElement from DI's DiagramElement to restrict UMLDiagramElements to own only UMLDiagramElements, and to be ordered (for specifying how diagram elements hide each other, see the DD specification). It introduces isIcon for modelElements that have an option to be shown with geometric figures other than rectangles, such as Actors, or that have an option to be shown with lines other than solid with open arrow heads, such as Realization. A value of true for isIcon indicates the alternative notation shall be rendered. UMLShapes may use both notations at the same time, with the small alternative notation rendered inside the regular notation. This is interchanged as two UMLShapes with the same modelElement, one having a value of false for isIcon and the other true, with the first having the second as an ownedElement, and the first having much larger bounds than the second.

Some things to note about UMLDiagramElement:

- The ownedElements of UMLDiagramElements may have bounds that do not intersect (are visually outside) the bounds of their owningElements.
- The modelElements of ownedElements of UMLDiagramElements may have different owners than the modelElements of owningElements.

UMLDiagram is the most general class for UML diagrams, and is the root of the diagram taxonomy defined in this Annex (also see in Figure A.5 in Annex A). It is specialized from PackageableElement for ownership by Packages. UMLDiagram introduces isFrame to indicate whether diagram frames shall be shown (see Annex A). If isFrame has the value true, heading shall give a string to be shown in the pentagonal header of the frame (see sub clause B.2.4 about UMLLabel). The heading shall have the same modelElement as its headedDiagram, if any. The diagram kind in the heading shall be rendered in boldface. UMLDiagram also introduces isIso to indicate whether ISO notation rules shall be followed.



## B.2.4 Labels

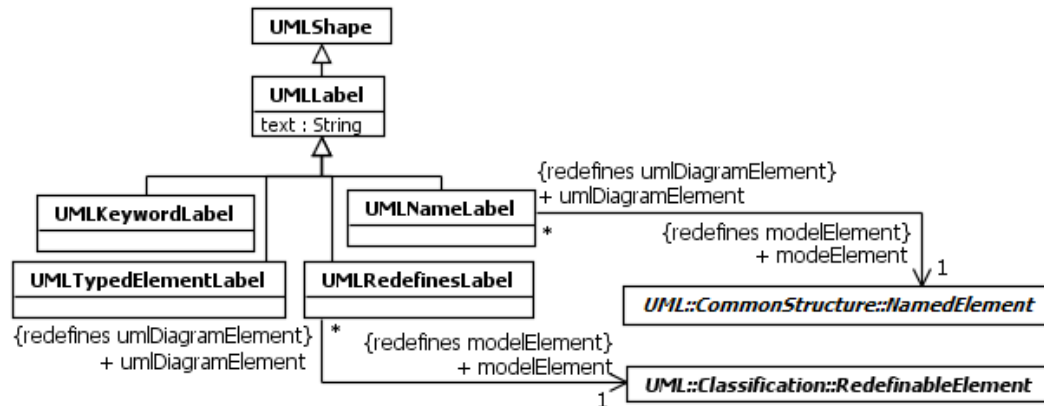


Figure B.5 Labels

The model in Figure B.5 specializes **UMLShape** into the most general class for shapes that shall be rendered only as character strings (**UMLLabel**), and specializes it into various kinds of labels used in multiple kinds of **UMLDiagrams**. Specializations of **UMLLabel** enable receivers of interchange files to determine which portions of the **modelElements** are intended to be shown, enabling them to update renderings when portions of the **modelElements** change. For example, receivers need to know when a **UMLLabel** is intended to show the name of its **modelElement**, to update the string rendered when the name of the **modelElement** changes. **UMLNameLabel** shall be used in this case to indicate the intent. Specializations of **UMLLabel** are introduced only when it is ambiguous or cumbersome to determine which portions of the **modelElements** are intended to be shown by parsing text as UML specifies. Many strings are interchanged with instances of **UMLLabel** that are not instances of its specializations.

All **UMLLabels** shall have at least one **modelElement**, except those which are classified only by **UMLLabel** and none of its specializations, which might have no **modelElements** (most **UMLLabels** are constrained to have exactly one **modelElement**). The **modelElements** shall be same as those of the **UMLDiagramElement** being labeled, if any, except where noted in this Annex, and shall be ownedElements of that **UMLDiagramElement**, including when the **UMLLabel** is positioned outside the **UMLDiagramElement** (not all **UMLLabels** are positioned near **UMLDiagramElements** having the same **modelElements**, for example **UMLLabels** in **UMLCompartments** showing Properties of Classifiers). This enables receivers of interchange files to determine where to position renderings of **UMLLabels** as positions change for renderings of the **UMLDiagramElement** being labeled, in particular when multiple **UMLDiagramElements** in the same **UMLDiagram** have the same **modelElement** as the **UMLLabel**. All **UMLLabels** shall have values of false for `iscon`. **UMLLabels** shall always specify text to be rendered, which shall conform to UML specifications for labeling each kind of **modelElement**, including BNF, if any.

The specializations of **UMLLabel** are for showing information about the portions of their **modelElements** defined by the type of the **modelElement** Property. For example, **UMLNameLabel**'s **modelElement** is restricted to **NamedElements**, which have Properties for name and visibility, and these are the portions of the **modelElement** **UMLNameLabel** shows. If no specialization shows all the information needed, use **UMLLabel** directly. For example, a wide range of information about Properties is shown textually in **UMLCompartments**, including redefinition and default values. Text for these Properties shall be interchanged with **UMLLabel** directly (see sub clause B.3.3), even when it only includes name and visibility.

**UMLNameLabels** are for showing information about **NamedElements**, which always includes their name, or `nameExpression`, or `ElementImport` alias if the **NamedElement** is imported, and may also include package containment (qualified names) and visibility.



UMLKeywordLabels are for showing various kinds of information about modelElements using wording and punctuation specified by UML between guillimets (see Annex C). UMLKeywords shall have exactly one modelElement. The text shall include the guillimets and the keyword only.

UMLTypedElementLabel is for showing name, type, and role information about exactly one Slot, InstanceSpecification, InstanceValue, or element with a type, such as TypedElement or Connector. Some other things to note about interchanging UMLTypedElementLabels are:

- If modelElement is a Slot, the text may include the name of the definingFeature, the name of the type of the definingFeature, and the value of the modelElement.
- If modelElement is an InstanceSpecification or an InstanceValue, the text may include the name of the InstanceSpecification or of the instance of the InstanceValue, and the names of the classifiers of the InstanceSpecification or of the instance of the InstanceValue.
- If modelElement is an InstanceSpecification that has a specification, the text may include the specification.
- If modelElement is an InstanceValue that is a value of a slot of an InstanceSpecification that has a StructuredClassifier as classifier, and the UMLTypedElementLabel is appearing in a shape with the InstanceValue as modelElement, in a UMLCompositeStructureDiagram that has the InstanceSpecification as a modelElement (see sub clause 11.2.4), the text may include the name of the Property that is the defining definingFeature of the Slot owning the InstanceValue that is a modelElement of the UMLTypedElementLabel.
- If modelElement has a type, as in TypedElements or Connectors, the text may include the name of the modelElement, and the name of the type of the modelElement.

UMLRedefinesLabel is for showing information about exactly one RedefinableElement, which may include the name of the redefinedElement.

## B.2.5 Compartmentable Shapes

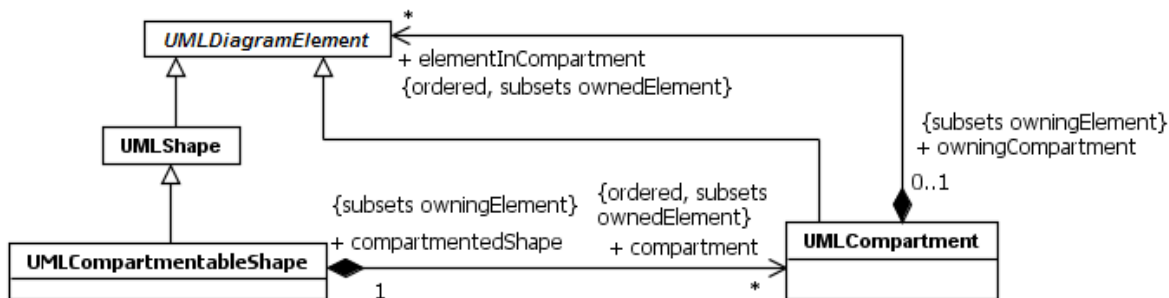


Figure B.6 UML Compartmentable Shapes

The model in Figure B.6 specializes UMLDiagramElement and UMLShape into UMLCompartmentableShape and UMLCompartment, respectively, to make them concrete, add properties, and redefine inherited properties for shapes with segregated contents.

UMLCompartmentableShape is the most general class for UML elements that may have information shown in separated portions inside their shapes, usually arranged linearly and separated by solid lines (*compartments*). It subsets ownedElement from UMLDiagramElement to specify compartments that are to appear vertically ordered (first in order is shown at the top), where are

captured with UMLCompartment. UMLCompartment subsets ownedElement from UMLDiagramElement to specify contents of compartments that are to appear vertically ordered (first in order is shown at the top). UMLCompartments have no modelElements.

Compartment titles shall be interchanged as UMLLabels with no modelElements, and as the first orderedElement of UMLCompartments.

## B.2.6 Stereotype Applications

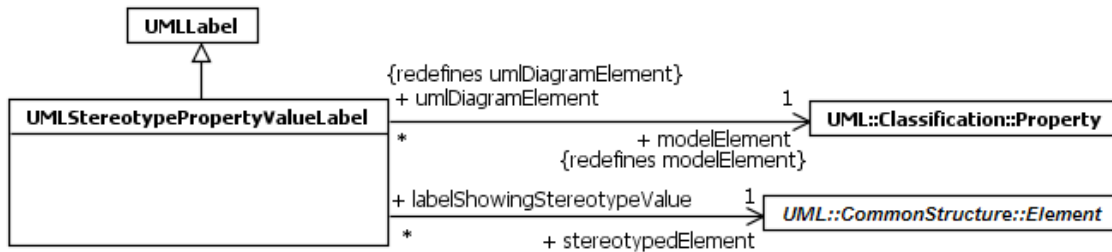


Figure B.7 Stereotype Application Labels

The model in Figure B.7 specializes UMLLabel into UMLStereoTypePropertyValueLabel to introduce associations for showing Property values of Stereotypes applied to UML abstract syntax elements. See sub clauses B.3.2 and B.3.4 about interchanging notation for Stereotype definitions.

UMLStereoTypePropertyValueLabels have attributes of Stereotypes as modelElements, and give the Element to which the Stereotype is applied. UMLStereoTypePropertyValueLabels shall be rendered as strings showing the values of applied Stereotypes, in the syntax specified by UML (see sub clause 12.3.4). UMLStereoTypePropertyValueLabels are ownedElements of

- UMLShapes with no modelElements (rendered as note symbols, see sub clause B.2.3) that are the source or target of UMLEdges with no modelElements, where the other end of the UMLEdges are UMLShapes with modelElements that are the stereoTypeElements.
- UMLCompartments where all the UMLStereoTypePropertyValueLabels have
  - as their stereoTypeElement the modelElement of the owningElement of the UMLCompartment, and
  - as their modelElements Properties of the same Stereotype, which is applied to the modelElement of the owningElement of the UMLCompartment.

Some other things to note about interchanging Stereotype applications are:

- Stereotypes applied to UML abstract syntax elements may be shown:
  - textually, interchanged with UMLLabels that have modeler-defined Stereotypes as modelElements, and that are ownedElements of UMLDiagramElements that have modelElements with the Stereotypes applied. These UMLLabels shall be rendered as strings giving the names of the Stereotype between guillemets. The UMLLabels may have multiple Stereotypes as modelElements, which shall be rendered as a comma-delimited list of Stereotype names between one set of guillemets.
  - graphically with modeler-defined icons. The icons are specified on Stereotypes, and are interchanged with UML abstract syntax, rather than UML DI. The rules for rendering the icon are specified in sub clause 12.3.4, and below in terms of UML DI.
- UMLShapes that

- have a modelElement with exactly one Stereotype applied that have a value for icon, and
- have a value of true for isIcon, and
- are rendered with rectangles when they have a value of false for isIcon,

shall be rendered as the icon. Any icons rendered when no Stereotypes are applied are not rendered in this case.

- UMLShapes that
  - have a Stereotype as modelElement that has a value for icon, and
  - are ownedElements of UMLDiagramElements that have modelElements with the modelElement Stereotype applied, and
  - have a value of true for isIcon,

shall be rendered as the Stereotype icon. These UMLShapes shall be rendered either

- Inside geometrical figures rendered for UMLShapes that are their owningElements, which shall have a value of false for isIcon. Or,
- Near lines rendered for UMLEdges that are their owningElements.
- To the left of strings rendered for UMLLabels that are their owningElements.

## B.2.7 UML Styles

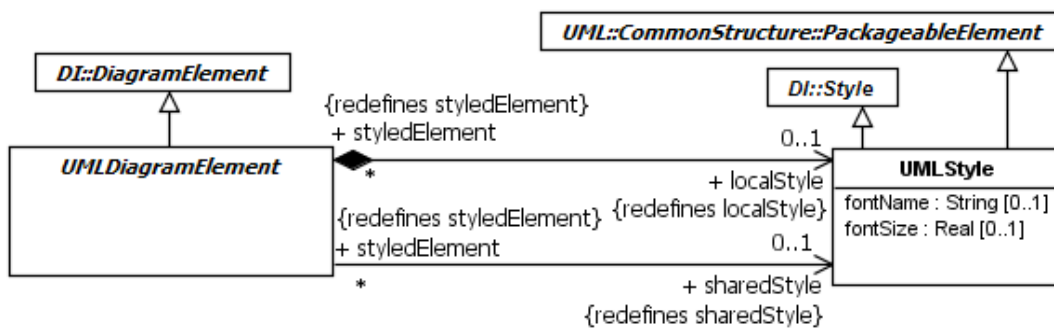


Figure B.8 UML Styles

The model in Figure B.8 specializes DI Style into the most general class for styles in UML (UMLStyle) to make it concrete, add properties, and redefine inherited style-related properties on UMLDiagramElement. UMLStyle introduces properties to specify the name and size of fonts used in rendering UMLDiagramElements. The fontSize is given in typographical points.

## B.3 Structure

### B.3.1 Summary

The Structure portion of UML DI captures graphical aspects of structure diagrams and their contents.

## B.3.2 Structure Diagrams

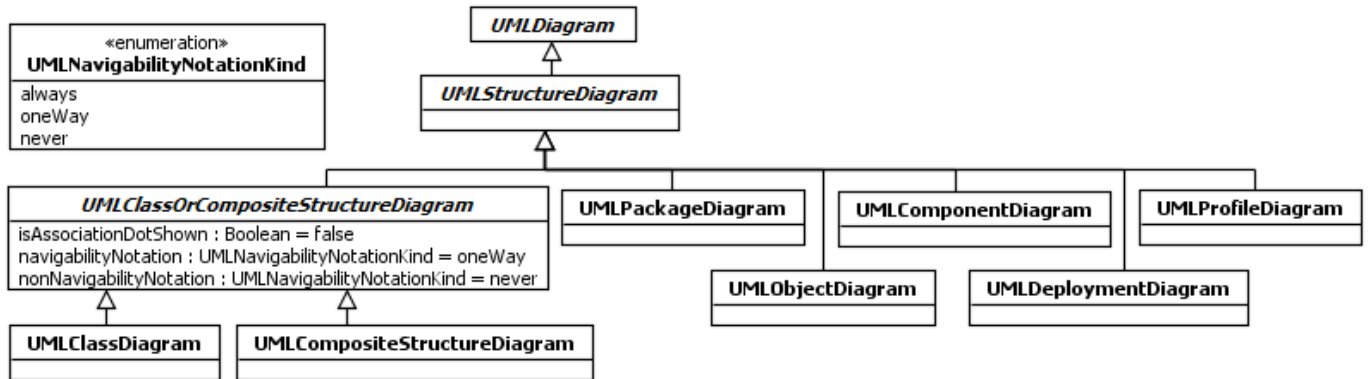


Figure B.9 Structure Diagrams

The model in Figure B.9 specializes `UMLDiagram` into the most general class for diagrams depicting structural elements (`UMLStructureDiagram`), and specializes it into various kinds of structure diagrams (see Annex A). All `UMLStructureDiagrams` have no `modelElements`, except for `UMLCompositeStructureDiagrams`, which have exactly one.

`UMLClassDiagrams` and `UMLCompositeStructureDiagrams` have some common graphical options. They may use the dot notation for associations, or not, as indicated by `isAssociationDotShown`. They may show navigability (by an open arrowhead) or nonnavigability (by an X) either always, only for unidirectional associations (`oneWay`), or never, as indicated by `navigabilityNotation` and `nonNavigabilityNotation`, respectively.

`UMLCompositeStructureDiagrams` have `modelElements` that are `StructuredClassifiers` or `InstanceSpecifications` with a classifier that is a `StructuredClassifier` (see `UMLTypedElementLabel` in sub clause B.2.4, next to last bullet).

Some notes about the contents of `UMLClassDiagrams` and `UMLCompositeStructureDiagrams` are:

- Navigability notation as determined by `navigabilityNotation` and `nonNavigabilityNotation` applies to `UMLEdges` that have as `modelElements` `Associations`, `InstanceSpecifications` that have an `Association` as classifier, `Properties`, or `Slots`.
- `UMLEdges` between `UMLShapes` that have `InstanceSpecifications` as `modelElements` and that are intended to show links have `InstanceSpecifications` as `modelElements` that have an `Association` classifying the link as classifier. `UMLLabels` showing the names of the `Association` memberEnds have `Slots` as `modelElements`.
- Ball and socket notations for required and provided Interfaces are interchanged as `UMLClassifierShapes` with a value of true for `iscon`. The lines between these notations and port rectangles are interchanged as `UMLEdges` with `InterfaceRealization` or `Usage Dependencies` as `modelElements` with the Interfaces as suppliers, which determine whether the `UMLClassifierShapes` shall be rendered as balls or sockets (used Interfaces are required and realized Interfaces are provided).

Some notes about the contents of `UMLClassDiagrams` specifically are:

- Properties or Slots shown with lines are interchanged with `UML Edges` with Properties or Slots as `modelElements`, respectively.
- Generalization arrows between association lines are interchanged as `UMLEdges` with `Generalizations` as `modelElements`, and sources and targets that are `UMLEdges` with `Associations` (including `AssociationClasses` shown as lines) as `modelElements`.

- UMLShapes with Packages as modelElements that do not have UMLShapes rendered inside them show the name of the Package in the large rectangle, rather than the tab. Otherwise the name of the Package is shown in the tab. The same applies to keywords and icon adornments.
- UMLEdges with no modelElements between UMLShapes that have a Namespace and a NamedElement as modelElement are shown with cross hairs in a circle on the Namespace end.
- The notation for qualified Association memberEnds is interchanged as UMLShapes with the memberEnd as modelElement, owning UMLLabels with qualifiers (UMLProperties) as modelElements.
- UMLClassifierShapes with Interfaces as modelElements, and rendered with ball or socket notations (see notes on UMLClassDiagrams and UMLCompositeStructureDiagrams above), may be the source or target of UMLEdges that have Dependencies as modelElements between:
  - Ports requiring or providing the Interfaces.
  - InterfaceRealization or Usage Dependencies with the Interfaces as suppliers.

Some notes about the contents of UMLCompositeStructureDiagrams specifically are:

- UMLClassifierShapes with Interfaces as modelElements, and rendered with ball or socket notations (see notes on UMLClassDiagrams and UMLCompositeStructureDiagrams above), may be the source or target of UMLEdges that have Connectors as modelElements that have ConnectorEnds with roles that are Ports requiring or providing the Interfaces.
- UMLEdges with Connectors as modelElements may have as source and/or target UMLClassifierShapes with Interfaces as modelElements, and rendered as ball or socket notation (see sub clause B.3.3), when the Connector end roles are Ports with a value for partWithPort and requiring or providing exactly one Interface (simple ports).
- UMLClassifierShapes with Interfaces as modelElements, and rendered as ball or socket notation (see sub clause B.3.3), may be rendered near each other (with the ball “in” the socket, and no line between them) to show Connectors with end roles that are the Ports with a value for partWithPort and requiring or providing exactly one Interface (simple ports). These UMLClassifierShapes may be the source or target of multiple UMLEdges that have all InterfaceRealization or all Usage Dependencies as modelElements when the Connector has more than two ends.

Some notes about UMLProfileDiagrams are:

- Stereotype definitions shall be shown with UMLClassifierShapes (see sub clause B.3.3, Stereotypes are Classes and are shown the same way).
- UMLEdges with Extensions as modelElements shall be rendered as solid lines with filled arrowheads, without navigability arrows or other association adornments. See sub clause B.3.4 about ExtensionEnd labels on these UMLEdges.
- UMLEdges with ProfileApplications as modelElements shall be rendered as dashed lines with open arrowheads.

### B.3.3 Classifier Shapes

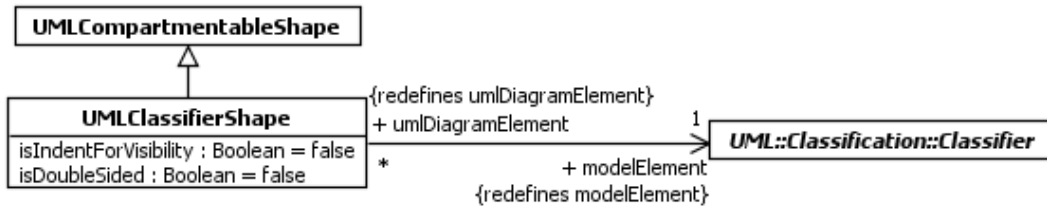


Figure B.10 Classifier Shapes

The model in Figure B.10 specializes UMLCompartmentableShape into UMLClassifierShape, to add properties, and restrict them to show exactly one Classifier. UMLClassifierShape introduces `isIndentForVisibility` for modelElements that are shown with feature compartments, to indicate features are shown indented under visibility headings. It also introduces `isDoubleSided` for modelElements that are Classes with true as a value for `isActive` that are shown as rectangles, to indicate whether the vertical sides shall be rendered as double lines.

Some notes about UMLClassifierShapes are:

- Classifier features shown in order textually in UMLClassifierShape compartments shall be interchanged with UMLLabels having a single Feature each as modelElements.
- Ellipsis appearing below Classifier features shown in order textually in UMLClassifierShape compartments shall be interchanged as UMLLabels with no modelElements, and as the last orderedElement of UMLCompartments.
- UMLClassifierShapes with Components as modelElements may have UMLCompartments as ownedElements that have UMLLabels as ownedElements with required and provided Interfaces of the Components as modelElements, where the UMLLabels show names of the Interfaces. The UMLLabels shall have InterfaceRealization or Usage Dependencies as modelElements with Interfaces as suppliers, which determine whether the Interfaces are required and/or provided (used Interfaces are required and realized Interfaces are provided).
- See sub clause B.3.2 for more notes on interchanging UMLClassifierShapes.

### B.3.4 Multiplicity and Association End Labels

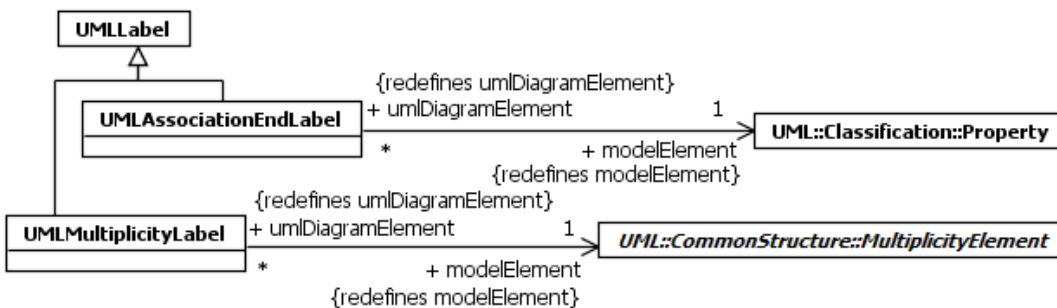


Figure B.11 Multiplicity and Association End Labels

The model in Figure B.11 specializes UMLLabel into UMLMultiplicityLabel and UMLAssociationEndLabel to restrict them to show information about exactly one MultiplicityElement or exactly one Association memberEnd, respectively.

Some notes about UMLMultiplicityLabels and UMLAssociationEndLabels are:

- Information about Properties shown textually in UMLCompartments shall be interchanged with UMLLabel directly (see sub clause B.3.3), rather than UMLMultiplicityLabels or UMLAssociationEndLabel, even when the Property is an Association memberEnd or when Multiplicity is shown (see sub clause B.2.4 about when to use specializations of UMLLabel generally).
- UMLMultiplicityLabel shall be used to show information about the portions of their modelElements defined by MultiplicityElement, rather than other specializations of MultiplicityElement that might classify the modelElements, such as Property (see sub clause B.2.4 about when to use specializations of UMLLabel generally).
- ConnectorEnds shall be shown with UMLMultiplicityLabel (the only underived Property introduced by ConnectorEnds, role, is shown by the targets of UMLEdges with Connectors as modelElements, and the only generalization of ConnectorEnd is MultiplicityElement). Information about definingEnds of ConnectorEnds is shown with UMLAssociationEndLabels (definingEnds are Properties).
- UMLAssociationEndLabels shall be used to show information about Association memberEnds, other than that shown with UMLMultiplicityLabel (Properties are MultiplicityElements). Multiple UMLAssociationEndLabels can have the same modelElement, each showing its own aspect of the modelElement.
- UMLAssociationEndLabels with ExtensionEnds as modelElements may only be used when the modelElements have an value of 1 for lower, in which case UMLAssociationEndLabels shall have the value “{required}” for text (ExtensionEnds are Extension ownedEnds, a kind of memberEnd, and Extensions are Associations).

### B.3.5 Association, Connector, and Link Shapes

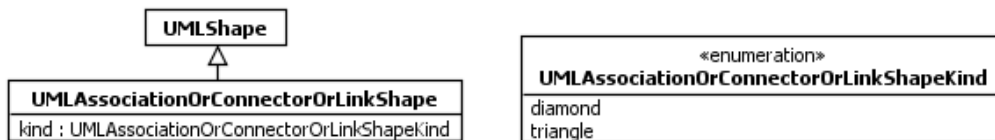


Figure B.12 Association, Connector, and Link Shapes

The model in Figure B.12 specializes UMLShape into UMLAssociationOrConnectorOrLinkShape, to add a property, and restrict them to show exactly one Association, Connector, or InstanceSpecification with an Association classifier. It introduces kind to indicate to how the modelElement shall be shown.

UMLAssociationOrConnectorOrLinkShapes with a value of diamond for kind shall be rendered as rhombuses, and shall have a modelElement that is either an Association with two memberEnds, a Connector with two ends, or an InstanceSpecification with an Association classifier that has two memberEnds (when there are more than two ends these elements are always shown this way, and the notation shall be interchanged with UMLShapes). Regardless of the number of ends, exactly two of the lines from the rhombus shall be interchanged with UMLEdges with modelElements that are either Properties (the memberEnds of the Association modelElement of the UMLAssociationOrConnectorOrLinkShape, or memberEnds of an Association that is a classifier of the InstanceSpecification modelElement UMLAssociationOrConnectorOrLinkShape) or ConnectorEnds (the ends of a Connector modelElement of UMLAssociationOrConnectorOrLinkShape).

UMLAssociationOrConnectorOrLinkShape with a value of triangle for kind shall be rendered as a filled triangle annotating a UMLEdge that has the same modelElement as the UMLAssociationOrConnectorOrLinkShape.

## B.4 Behavior

### B.4.1 Summary

The Behavior portion of UML DI captures graphical aspects of behavior diagrams and their contents.

### B.4.2 Behavior Diagrams

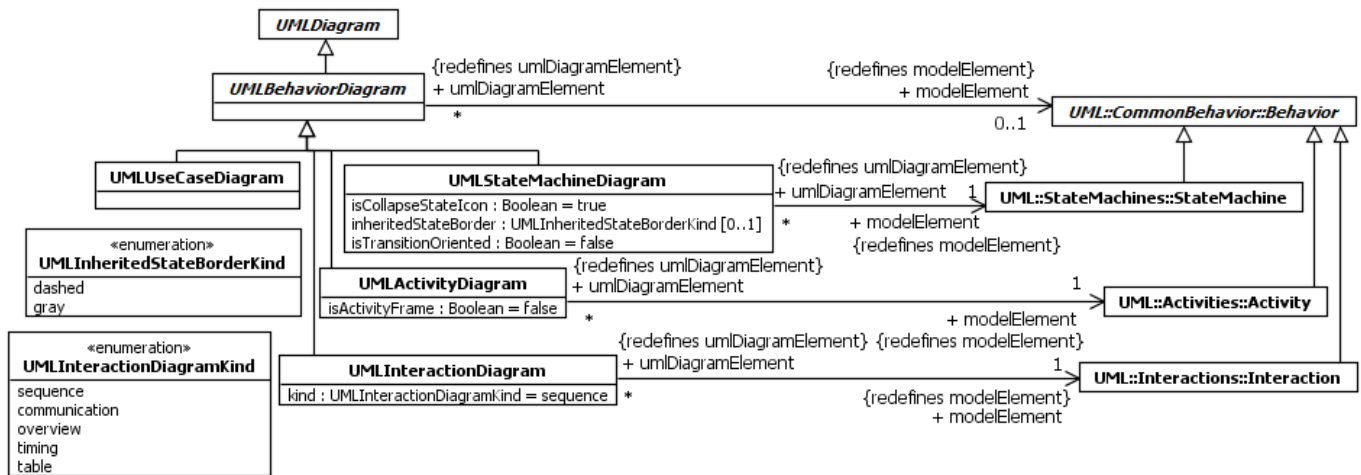


Figure B.13 Behavior Diagrams

The model in Figure B.13 specializes **UMLDiagram** into the most general class for diagrams depicting behavioral elements (**UMLBehaviorDiagram**), and specializes it into various kinds of behavior diagrams (see Annex A). **UMLBehaviorDiagrams** are restricted to have no more than one **modelElement**, which must be a **Behavior**.

**UMLUseCaseDiagrams** have no **modelElements**. Some notes about the contents of **UMLUseCaseDiagrams** are:

- UseCase notations shall be interchanged with **UMLClassifierShapes**, rendered as ovals when true is a value of **isIcon** and as rectangles when false is a value.
- UseCase notations shall be interchanged with **UMLClassifierShapes**, rendered as stick figures when true is a value of **isIcon** and as a rectangles when false is a value.
- Modeler-defined icons showing Actors may be specified with Stereotype icons.
- Extends and Includes notations shall be interchanged with **UMLEdges** with Extends and Includes as **modelElements**.
- **UMLShapes** with Extends as **modelElements** shall be rendered as note symbols, and have **UMLLabels** (not specializations) as the only two ownedElements. The **UMLLabels** shall have the condition or extensionLocations of the Extend as **modelElements**, and text giving the condition specification or extensionLocation names, respectively, in the syntax specified by UML. **UMLEdges** between such **UMLShapes** and other **UMLDiagramElements** shall have no **modelElements** and be rendered with dashed lines.

**UMLStateMachineDiagrams** are restricted to have exactly one **modelElement**, which must be a **StateMachine**.

**UMLStateMachineDiagram** introduces **isCollapseStateIcon** to indicate whether **UMLShapes** for composite States shall contain a small icon distinguishing them from non-composite States. It also introduces **inheritedStateBorder** to indicate how borders shall be



rendered on UMLShapes that have an inherited State as modelElement, which must have a value for UMLStateMachineDiagrams that show inherited States.

StateMachines may be shown in a “flow-chart” style (StateMachineDiagrams with a value of true for isTransitionOriented), where Transition triggers are shown as concave pentagons, and executableNodes in SequenceNodes in Transition effects are shown as round-cornered rectangles, except when they are SendSignalActions, which are shown as convex pentagons (this information about Transitions is shown only textually in StateMachineDiagrams with a value of false for isTransitionOriented). Transition-oriented UMLStateMachineDiagrams may have UMLShapes as ownedElements that have Triggers or ExecutableNodes as modelElements (other UMLStateMachineDiagrams shall not have these UMLShapes). UMLEdges that have these UMLShapes as source or target shall have Transitions as modelElements, and the modelElements shall be the same for UMLEdges that share the same UMLShape as source or target. In transition-oriented UMLStateMachineDiagrams, UMLEdges that have Transitions as modelElements, and have as source or target UMLShapes with States as modelElements with a value true for isSubmachineState, where the Transition’s corresponding source or target is a ConnectionPointReference, the ConnectionPointReference shall be shown textually overlaying the line rendered by the UMLEdge, with semicircles at both ends of the text. The text shall be interchanged with a UMLLabel that has the ConnectionPointReference as modelElement.

Some notes about the contents of UMLStateMachineDiagrams are:

- UMLShapes with Regions as modelElements that are adjacent to each other shall be rendered with a dashed line between them.
- Regions that are the only ones in their StateMachines or States are not shown.

UMLActivityDiagrams are restricted to have exactly one modelElement, which must be an Activity. UMLActivityDiagram introduces isActivityFrame to indicate whether the diagram frame shall be a round-cornered rectangle without a pentagonal header (isFrame shall have a value of false when isActivityFrame has a value of true). When true is a value of isActivityFrame, the name of the Activity is shown by a UMLNameLabel in the upper left of the diagram without showing the diagram kind.

Some notes about the contents of UMLActivityDiagrams are:

- UMLShapes with CallBehaviorActions as modelElements and Activities as behaviors shall be rendered with a rake symbol inside it when collapsed (when the UMLShape has no UMLShapes as ownedElements that have ActivityNodes as modelElements). When expanded, the behavior shall be shown with activity frame notation, including UMLEdges that have ActivityEdges as modelElements and UMLShapes as sources or targets that have the Activity or its ActivityParameterNodes as modelElements.
- Some information about InputPins and OutputPins may be shown graphically, as follows:
  - UMLShapes with a single InputPin or OutputPin as modelElement and a true value for isIcon shall be rendered with a small arrows in the rectangle directed towards or the round rectangle they are next to for InputPins, or away for OutputPins. If the InputPin or OutputPin corresponds to a Parameter that has a value of true for isStream the UMLShape shall be rendered as filled, inverting the arrow, if any.
  - UMLShapes with a single InputPin or OutputPin as modelElement corresponding to a Parameter that has a value of true for isStream, and with a true value for isIcon, shall be rendered as filled, inverting the arrow inside it.
- ObjectFlows between OutputPins and InputPins may be shown with shorthand notations, as follows:
  - UMLEdges shall have the same ActivityEdge as modelElement when their source or target are the same UMLShape with multiple Pins as modelElements (specifically an InputPin and OutputPin that are the source and target of the ActivityEdge, respectively). The UMLShape may be rendered as a small square, with a UMLLabel rendered near the small square. These UMLEdges shall be shown with filled arrowheads when they have ObjectFlows as modelElements, a true value for isIcon, and as source or target UMLShapes with multiple Pins as modelElements (specifically an InputPin and OutputPin), and all the Pins correspond to Parameters that have a value of true for isStream.

- UMLEdges with ObjectFlows as modelElements where the UMLEdges have as source and target the same UMLShapes with Actions as modelElements, shall be rendered with a small square near the line. The small square shall be interchanged with a UMLShape with the same ObjectFlows as modelElement as the UML Edge (a single UMLEdge may have multiple ObjectFlows as modelElement).
- A single UMLShape may be used as a shorthand for two ControlNodes that are the target and source an ActivityEdge as follows:
  - A single UMLShape may have as modelElements a MergeNode, ActivityEdge, and DecisionNode, with the MergeNode and DecisionNode as source and target of the ActivityEdge, respectively. UMLEdges that have such UMLShapes as target have an ActivityEdge as modelElement with the MergeNode as target. UMLEdges that have such UMLShape as source have an ActivityEdge as modelElement with the DecisionNode as source.
  - A single UMLShape may have as modelElements a JoinNode, ActivityEdge, and ForkNode, with the JoinNode and ForkNode as source and target of the ActivityEdge, respectively. UMLEdges that have such UMLShapes as target have an ActivityEdge as modelElement with the JoinNode as target. UMLEdges that have such UMLShape as source have an ActivityEdge as modelElement with the ForkNode as source.
- UMLEdges may have as targets UMLShapes that have as modelElement a handlerBody of an ExceptionHandler. These UMLEdges with a value of true for islcon shall be rendered as zigzag lines, and with a value of false shall be rendered as a zigzag graphic near the line.
- ExpansionRegions that have exactly one inputElement, outputElement, and Action node, and two ObjectFlow edges, (one with the inputElement as source and an inputValue of the node as target, the other with an outputValue of the node as source and the outputElement as target) may be shown with one of two kinds of shorthand notation, both of which include a UMLShape with the ExpansionRegion as modelElement, and containing a UMLLabel with the Action node as modelElement, rendered in the center of the UMLShape:
  - One notation shall have on the border of the UMLShape two other UMLShapes with the ExpansionRegion's inputElement and outputElement as modelElements, where these two other UMLShapes shall be ownedElements of the first UMLShape, and shall be the target and source of two UMLEdges with ObjectFlows as modelElements, respectively, where the ObjectFlows shall have the inputElement and outputElement as target and source, respectively. The UMLShape with ExpansionRegion as modelElement shall have an additional ownedElement that is a UMLKeywordLabel with the ExpansionRegion as modelElement showing the mode of the ExpansionRegion in the upper left.
  - The other notation is for parallel mode ExpansionRegions only. It may not show the inputElement and outputElement. The UMLShape with ExpansionRegion as modelElement shall be the target and source of the two UMLEdges with ObjectFlows as modelElements described above, respectively, and shall have an additional ownedElement in the upper right that is a UMLLabel with the ExpansionRegion as modelElement, and the value “\*” for text.
- Interrupting activity edges are identified by interruptible regions. UMLEdges with ActivityEdges as modelElements that are interruptingEdges of an InterruptibleRegion, and have a true value for islcon, are shown as zigzag lines. When these UMLEdges have and have a false value for islcon, UMLShapes with the ActivityEdge as modelElement shall be rendered as zigzag graphics near the lines.
- The outside line segments of UMLShapes with ActivityPartitions as modelElements that are ownedElements of an Activity that is the modelElement of a UMLActivityDiagram with a value of true for isActivityFrame may be merged with the activity frame.
- ActivityEdge connector notations shall be interchanged with UMLShapes that have the same ActivityEdge as modelElements, and have as ownedElements UMLLabels with that ActivityEdge as modelElement.
- See sub clause B.4.3 about UMLLabels in UMLActivityDiagrams.

UMLInteractionDiagrams are restricted to have exactly one modelElement, which must be an Interaction. UMLInteractionDiagram introduced kind, which affects the rendering of diagram contents in some cases. For example, UMLShapes with alt CombinedFragments as modelElements in UMLInteractionDiagrams with the value overview for kind shall be rendered as rhombuses. When the value for kind is table, the ownedElements that are UMLShapes and not UMLLabels shall be rendered as rectangles (see sub clause B.4.5).

Some notes about the contents of UMLInteractionDiagrams are:

- Properties of Interactions (as Classes) may be shown as text below the upper left corner of diagram (below the heading pentagon), interchanged with UMLLabels in the same way as notation for Properties in Classes. These UMLLabels may appear in other UMLShapes that are the source or target of UMLEdges. These UMLShapes shall be rendered as notes, and the UMLEdges as dashed lines.
- UMLEdges with GeneralOrderings as modelElements shall be rendered with filled arrowheads in the middle of the lines. The arrowheads shall not be interchanged as separate UMLShapes
- UMLEdges with Messages as modelElements that have lost or found as the value of messageKind shall be rendered with small filled circles at the target or source ends, respectively. The circles shall not be interchanged as separate UMLShapes.
- UMLShapes that have CombinedFragments as modelElements with a value of par for interactionOperand, and that have a value of true for isIcon, shall be rendered as brackets at the top and bottom of the shape (coregion notation).
- All CombinedFragments, regardless of the interactionOperand, shall be shown with text in pentagons in the upper left corners of their UMLShapes, where the text is interchanged with UMLLabels that have CombinedFragments as modelElements, possibly multiple labels per pentagon. The extra labels have immediately nested combined fragments as modelElements.
- UMLShapes with ExecutionSpecifications as modelElements a value of true for isIcon shall be rendered with shaded fill patterns.

### B.4.3 Activity Diagram Labels

Some notes about UMLLabels in UMLActivityDiagrams are:

- UMLLabels that are ownedElements of UMLActivityDiagrams, and have Activities as modelElements, and are not classified by any specialization of UMLLabel, shall have text beginning with keywords specified by UML in guillemets indicating the kind of information they show about the Activity.
- UMLLabels that are ownedElements of UMLObjectNodes with ActivityParameterNodes, and have Parameters as modelElements, and are not classified by any specialization of UMLLabel, shall have text giving information about the Parameter, in the syntax specified by UML.
- UMLLabels that are ownedElements of UMLShapes with CallOperationAction as modelElements, and have Classes or Operations as modelElements, and are not classified by any specialization of UMLLabel, shall have text giving the name of the Class or Operation, or both, in the syntax specified by UML.
- UMLLabels that are ownedElements of UMLShapes with Actions as modelElements, and have ActivityPartitions as modelElements, and are not classified by any specialization of UMLLabel, shall have text giving the names of the ActivityPartitions in the syntax specified by UML.
- UMLLabels that are ownedElements of UMLShapes with no modelElements, and have ActivityNodes as modelElements, and are not classified by any specialization of UMLLabel, shall have text beginning with keywords specified by UML in guillemets indicating the kind of information they show about the ActivityNode (these UMLShapes are rendered as note symbols, see sub clause B.2.3).

- UMLLabels that are ownedElements of UMLShapes with ObjectNodes as modelElements, and have States as modelElements, and are not classified by any specialization of UMLLabel, shall be rendered as comma delimited lists of the names of the UMLLabel's modelElements, enclosed in square brackets.
- UMLLabels that have text beginning and ending with braces (curly brackets), and are not classified by any specialization of UMLLabel, shall have ActivityNodes or ActivityEdges as modelElements, and the text shall give values for properties of the modelElements in a syntax specified by UML.
- UMLLabels that have text beginning and ending with square brackets, and are not classified by any specialization of UMLLabel, shall have ActivityEdges as modelElements, and the text between the square brackets shall show the guard of the ActivityEdge.
- UMLLabels may have Parameters as modelElements and be placed near UMLShapes that have InputPins or OutputPins of CallActions as modelElements (or near where the Pins would be if they are elided) to show Parameters of behaviors or operations of InvocationActions.
- UMLLabels near UMLShapes showing InputPins or OutputPins may be used to interchange shorthand notations as follows:
  - Actions that have an InputPin as the target of an ObjectFlow with an ActivityParameterNode as source, may be shown in a shorthand notation that shows only the Action, its InputPin, and the name of the parameter of the ActivityParameterNode interchanged as a UMLLabel with the parameter as modelElement (see Figure 15-52 in sub clause [15.4.4](#)).
  - Actions that have an OutputPin as the source of an ObjectFlow with an ActivityParameterNode as target, may be shown in a shorthand notation that shows only the Action, its OutputPin, and nearby the name of the parameter of the ActivityParameterNode interchanged as a UMLLabel with the parameter as modelElement (see Figure 15-52 in sub clause [15.4.4](#)).
  - AddVariableValueActions that have an InputPin as the target of an ObjectFlow with an OutputPin of another Action as source, and that have an OutputPin that is the source of an ObjectFlow having a FlowFinal as target, may be shown in a shorthand notation that shows only the other Action, its OutputPin, and nearby the name of the AddVariableValueAction's variable shown by a UMLLabel with the variable as modelElement (see Figure 16-38 in sub clause [16.9.4](#)).
  - ActionInputPins with fromActions that are ReadVariableActions may be shown in a shorthand notation that i only the ActionInputPin and nearby the name of the variable of the ReadVariableAction interchanged as a UMLNameLabel with the variable as modelElement.
  - ActionInputPins with fromActions that are ReadSelfObjectActions may be shown in a shorthand notation that shows only the ActionInputPin and nearby the string "self" interchanged as a UMLLabel with the ReadSelfObjectAction as modelElement.
  - ActionInputPins with fromActions that are ValueSpecificationActions may be shown in a shorthand notation that shows only the ActionInputPin and nearby the value of the ValueSpecificationAction interchanged as a UMLLabel with the value as modelElement.
- ActivityPartitions headings shall be interchanged with UMLLabels with their ActivityPartition as modelElement.

## B.4.4 State Shapes

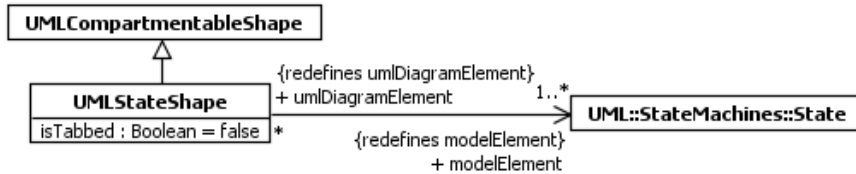


Figure B.14 State Shapes

The model in Figure B.14 specializes `UMLCompartmentableShape` into `UMLStateShape`, to add a property, and restrict them to show one or more States. It introduces `isTabbed` to indicate whether a tab shall be added to the top of the shape to show the name of the State. `UMLStateShapes` may have multiple `modelElements` only when their outgoing `Transitions` have no triggers or effects and target a single `Pseudostate` with a value of `junction` for `kind` and exactly one outgoing `Transition` (“state list” notation). In this case, the names of the `modelElements` are shown by a `UMLLabel` with the same `modelElements`, with the `UMLStateShape` as `owningElement`, and the `UMLLabel` shall be rendered as a comma delimited list of the names of its `modelElements`.

Some notes about the contents of `UMLStateShapes` are:

- `UMLLabels` showing entry, do, exit `Behaviors` shall have the `Behaviors` as `modelElements`, and text in the syntax specified by UML.
- `UMLLabels` showing internal or local `Transitions` have the `Transitions` as `modelElements`, and text in the syntax specified by UML.

## B.4.5 Interaction Tables

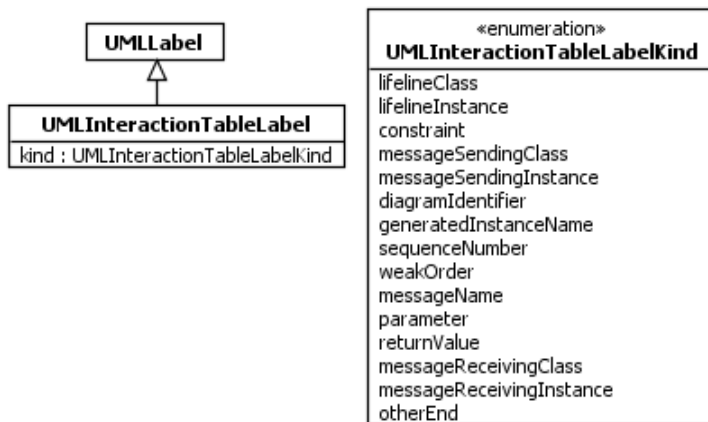


Figure B.15 Interaction Shapes

The model in Figure B.15 specializes `UMLLabel` into `UMLInteractionTableLabel` to add a property for labels in the tabular format of `UMLInteractionDiagrams`.

`UMLInteractionTableLabels` shall be `ownedElements` of `UMLShapes` that are `ownedElements` of a single `UMLShape` that is an `ownedElement` of a `UMLInteractionDiagram` with a value of `table` for `kind`. All these `UMLShapes` shall be rendered as rectangles, one for the entire table (an `ownedElement` of the `UMLInteractionDiagram`), and one for each cell in the table (the second layer of `ownedElements`). `UMLShapes` for the whole table shall have an `InteractionDiagram` as `modelElement` that is the same as the `modelElement` of their `owningElement`. `UMLShapes` for cells shall be `ownedElements` in the `UMLShape` for the table, and have no

more than one ownedElement, which shall be a UMLInteractionTableLabel. UMLShapes for cells in the top row shall have one ownedElement that has no modelElement, while all the UMLShapes for cells and their ownedElements shall have exactly one modelElement., which shall be the same for each UMLShape and its ownedElement. UMLInteractionTableLabels for cells in the same column shall all have the same value for kind.

## B.5 Information Flows

Some notes about UMLDiagramElements related to InformationFlows are:

- UMLLabels with InformationFlows as modelElements show the names of the kinds of things conveyed. Multiple conveyed names for the InformationFlows shall be shown with one UMLLabel rendered as a comma-delimited list of the names, which may be due to multiple modelElements for the same UMLLabel, or multiple conveyed names for the same modelElement, or both.
- UMLShapes with InformationFlows as modelElements and a value of true for isIcon shall be rendered as filled triangles overlaying and aligned with UMLEdges with the same InformationFlows as modelElements, pointing towards the informationTarget. There shall be one triangle for all InformationFlows going in the same direction realized by the same element (a maximum of two triangles for the same realizing element).

## B.6 Classifier Descriptions

### UMLActivityDiagram [Class]

#### Description

Shows an Activity and its elements. Also see Annex A.

#### Generalizations

[UMLBehaviorDiagram](#)

#### Attributes

- `isActivityFrame` : [Boolean](#) [1..1] = false  
Indicates whether the UMLActivityDiagram shall be shown with a frame that is a round-cornered rectangle without a pentagonal header.

#### Association Ends

- `modelElement` : [Activity](#) [1..1]{redefines [UMLBehaviorDiagram::modelElement](#)} (opposite [A UMLActivityDiagram modelElement umlDiagramElement::umlDiagramElement](#))  
Restricts UMLActivityDiagrams to show Activities.

#### Constraints

- `frame`  
`isFrame` and `isActivityFrame` must not be true at the same time.

```
inv: not (isActivityFrame and isFrame)
```

### UMLAssociationEndLabel [Class]

#### Description

Shows text about Properties that are Association ends.

#### Generalizations

[UMLLabel](#)

#### Association Ends

- `modelElement` : [Property](#) [1..1]{redefines [UMLDiagramElement::modelElement](#)} (opposite [A UMLAssociationEndLabel modelElement umlDiagramElement::umlDiagramElement](#))  
Restricts UMLAssociationEndLabels to show only Properties (that are AssociationEnds, see constraint).

## Constraints

- `modelElement_type`  
modelElement must be an Association end.

```
inv: not modelElement->forall(association->isEmpty())
```

## UMLAssociationOrConnectorOrLinkShape [Class]

### Description

Shows shapes for binary relationships.

### Generalizations

[UMLShape](#)

### Attributes

- `kind` : [UMLAssociationOrConnectorOrLinkShapeKind](#) [1..1]

## Constraints

- `modelElement_mult`  
UMLAssociationOrConnectorOrLinkShapes must have exactly one modelElement.

```
inv: modelElement->size()=1
```

- `modelElement_type`  
modelElement must be an Association, Connector, or InstanceSpecification with an Association classifier.

```
inv: modelElement->asSequence()->first().oclIsKindOf(Association) or modelElement->asSequence()->first().oclIsKindOf(Connector)
or (
  modelElement->asSequence()->first().oclIsKindOf(InstanceSpecification)
  and (modelElement->asSequence()->first().oclAsType(InstanceSpecification).classifier->select(oclIsKindOf(Association))->size() > 0)
```

- `edge_association`  
For diamond kind and an Association modelElement, the Association has exactly two memberEnds, and exactly two of the UMLEdges linked to the shape have those memberEnds as modelElements.

```
inv: (kind=UMLAssociationOrConnectorOrLinkShapeKind::diamond and modelElement->forall(oclIsKindOf(Association))) implies
let association : Association = modelElement->any(true).oclAsType(Association) in
  ((association.memberEnd->size() = 2) and
   (sourceEdge.modelElement->union(targetEdge.modelElement)->select(em | association.memberEnd->includes(em))->size()=2))
```

- `edge_instancespec`  
For diamond kind and an InstanceSpecification modelElement, exactly one Association classifier of the



InstanceSpecification has exactly two memberEnds and exactly two of the UMLEdges linked to the shape have those memberEnds as modelElements.

```
inv: (kind=UMLAssociationOrConnectorOrLinkShapeKind::diamond and modelElement-
>forall(oclIsKindOf(InstanceSpecification))) implies
let instanceSpecification : InstanceSpecification = modelElement-
>any(true).oclAsType(InstanceSpecification) in
  (instanceSpecification.classifier->select(a | a.oclIsKindOf(Association) and
(a.oclAsType(Association).memberEnd-
>size() = 2) and
  (sourceEdge.modelElement-
>union(targetEdge.modelElement)
  ->select(e |
a.oclAsType(Association).memberEnd->includes(e.modelElement))
  ->size()=2))
  ->size()=1)
```

- edge\_connector  
For diamond kind and a Connector modelElement, the Connector has exactly two ends, and exactly two of the UMLEdges linked to the shape have definingEnds of those ends as modelElements.

```
inv: (kind=UMLAssociationOrConnectorOrLinkShapeKind::diamond and modelElement-
>forall(oclIsKindOf(Connector))) implies
let connector : Connector = modelElement->any(true).oclAsType(Connector) in
  ((connector.end->size() = 2) and
  (sourceEdge.modelElement->union(targetEdge.modelElement)->select(em |
connector.end.definingEnd->includes(em))->size()=2))
```

## UMLAssociationOrConnectorOrLinkShapeKind [Enumeration]

### Description

### Literals

- diamond
- triangle

## UMLBehaviorDiagram [Abstract Class]

### Description

The most general class for UMLDiagrams depicting behavioral elements.

### Generalizations

[UMLDiagram](#)

### Specializations

[UMLActivityDiagram](#), [UMLInteractionDiagram](#), [UMLStateMachineDiagram](#), [UMLUseCaseDiagram](#)

## Association Ends

- modelElement : [Behavior](#) [0..1]{redefines [UMLDiagramElement::modelElement](#)} (opposite [A UMLBehaviorDiagram modelElement umlDiagramElement::umlDiagramElement](#))  
Restricts UMLBehaviorDiagrams to showing Behaviors.

## UMLClassDiagram [Class]

### Description

See Annex A.

### Generalizations

[UMLClassOrCompositeStructureDiagram](#)

### Constraints

- no\_modelElement  
UMLClassDiagrams must have no modelElements.

inv: modelElement->isEmpty()

## UMLClassOrCompositeStructureDiagram [Abstract Class]

### Description

Specifies the common aspects of UMLClassDiagrams and UMLCompositeStructureDiagrams.

### Generalizations

[UMLStructureDiagram](#)

### Specializations

[UMLClassDiagram](#), [UMLCompositeStructureDiagram](#)

### Attributes

- isAssociationDotShown : [Boolean](#) [1..1] = false  
Indicates whether dot notation for associations shall be used.
- navigabilityNotation : [UMLNavigabilityNotationKind](#) [1..1] = oneWay  
Indicates when to show navigability of associations or connectors typed by associations.
- nonNavigabilityNotation : [UMLNavigabilityNotationKind](#) [1..1] = never  
Indicates when to show non-navigability of associations or connectors typed by associations.

## UMLClassifierShape [Class]

### Description

Shows Classifiers with shapes that may have compartments.

### Generalizations

[UMLCompartmentableShape](#)

### Attributes

- isIndentForVisibility : [Boolean](#) [1..1] = false  
For modelElements that are shown with feature compartments, indicates whether features are shown indented under visibility headings.
- isDoubleSided : [Boolean](#) [1..1] = false  
For modelElements that are Classes with true as a value for isActive that are shown as rectangles, indicates whether the vertical sides shall be rendered as double lines.

### Association Ends

- modelElement : [Classifier](#) [1..1]{redefines [UMLDiagramElement::modelElement](#)} (opposite [A UMLClassifierShape modelElement umlDiagramElement::umlDiagramElement](#))  
Restricts UMLClassifierShapes to showing exactly one Classifier.

### Constraints

- doubleSided\_isActive  
isDoubleSided may be true only when modelElement is an active Class.

```
inv: isDoubleSided implies (modelElement->forall(oclIsKindOf(Class)) and  
modelElement.oclAsType(Class)->forall(isActive))
```

## UMLCompartment [Class]

### Description

A separated portion of a UMLCompartmentableShape.

### Generalizations

[UMLDiagramElement](#)

### Association Ends

- ♦ elementInCompartment : [UMLDiagramElement](#) [0..\*]{ordered, subsets [DiagramElement::ownedElement](#)} (opposite [A UMLCompartment elementInCompartment owningCompartment::owningCompartment](#))

Contents of the compartment.

### Constraints

- no\_modelElement  
UMLCompartments have no modelElements.

inv: modelElement->isEmpty()

## UMLCompartmentableShape [Class]

### Description

The most general class for UML elements that may have contents shown in separated portions inside the shape.

### Generalizations

[UMLShape](#)

### Specializations

[UMLClassifierShape](#), [UMLStateShape](#)

### Association Ends

- ♦ compartment : [UMLCompartment](#) [0..\*]{ordered, subsets [DiagramElement::ownedElement](#)} (opposite [A\\_UMLCompartmentableShape compartmentedShape shape::compartmentedShape](#))  
Separated portions of the shape.

## UMLComponentDiagram [Class]

### Description

See Annex A.

### Generalizations

[UMLStructureDiagram](#)

### Constraints

- no\_modelElement

inv: modelElement->isEmpty()

## UMLCompositeStructureDiagram [Class]

### Description

Shows the internal structure of a StructuredClassifier. Also see Annex A.

## Generalizations

[UMLClassOrCompositeStructureDiagram](#)

## Constraints

- `modelElement_mult`  
UMLCompositeStructureDiagrams must have exactly one `modelElement`.

```
inv: modelElement->size() = 1
```

- `modelElement_type`  
`modelElement` must be a `StructuredClassifier` or an `InstanceSpecification` with a classifier that is a `StructuredClassifier`.

```
inv: modelElement->asSequence()->first().oclIsKindOf(StructuredClassifier)
or (
    modelElement->asSequence()->first().oclIsKindOf(InstanceSpecification)
    and (modelElement->asSequence()->first().oclAsType(InstanceSpecification).classifier->select(c
| c.oclIsKindOf(StructuredClassifier))->size() > 0))
```

## UMLDeploymentDiagram [Class]

### Description

See Annex A.

### Generalizations

[UMLStructureDiagram](#)

### Constraints

- `no_modelElement`  
UMLDeploymentDiagrams must have no `modelElement`.

```
inv: modelElement->isEmpty()
```

## UMLDiagram [Abstract Class]

### Description

The most general class for UML diagrams.

### Generalizations

[Diagram](#), [PackageableElement](#), [UMLDiagramElement](#)

### Specializations

[UMLStructureDiagram](#), [UMLBehaviorDiagram](#)

## Attributes

- isFrame : [Boolean](#) [1..1] = true  
Indicates when diagram frames shall be shown.
- isIso : [Boolean](#) [1..1] = true  
Indicate when ISO notation rules shall be followed.

## Association Ends

- ♦ heading : [UMLLabel](#) [0..1] (opposite [A\\_UMLDiagram\\_heading\\_headedDiagram::headedDiagram](#))

## Constraints

- no-frame-no-heading  
UMLDiagrams cannot have headings without frames, or vice-versa.  
  
`inv: (isFrame = false) = (heading->isEmpty())`
- heading\_modelElement  
The modelElement of the heading is the same as the modelElement of the diagram it heads.  
  
`inv: (heading->isEmpty()) or (heading.modelElement = modelElement)`

## UMLDiagramElement [Abstract Class]

### Description

The most general class for UML diagram interchange.

### Generalizations

[DiagramElement](#)

### Specializations

[UMLCompartment](#), [UMLDiagram](#), [UMLEdge](#), [UMLShape](#)

## Attributes

- isIcon : [Boolean](#) [1..1] = false  
For modelElements that have an option to be shown with shapes other than rectangles, such as Actors, or with other identifying shapes inside them, such as arrows distinguishing InputPins and OutputPins, or edges that have an option to be shown with lines other than solid with open arrow heads, such as Realization. A value of true for isIcon indicates the alternative notation shall be shown.

## Association Ends

- `modelElement` : [Element](#) [0..\*]{redefines [DiagramElement::modelElement](#)} (opposite [A UMLDiagramElement modelElement umlDiagramElement::umlDiagramElement](#))  
Restricts UMLDiagramElements to show UML Elements, rather than other language elements.
- `sharedStyle` : [UMLStyle](#) [0..1]{redefines [DiagramElement::sharedStyle](#)} (opposite [A UMLDiagramElement sharedStyle styledElement::styledElement](#))  
Restricts shared styles to UMLStyles.
- `♦ localStyle` : [UMLStyle](#) [0..1]{redefines [DiagramElement::localStyle](#)} (opposite [A UMLDiagramElement localStyle styledElement::styledElement](#))  
Restricts owned styles to UMLStyles.
- `owningElement` : [UMLDiagramElement](#) [0..1]{redefines [DiagramElement::owningElement](#)} (opposite [UMLDiagramElement::ownedElement](#))  
Restricts UMLDiagramElements to be owned by only UMLDiagramElements.
- `♦ ownedElement` : [UMLDiagramElement](#) [0..\*]{ordered, redefines [DiagramElement::ownedElement](#)} (opposite [UMLDiagramElement::owningElement](#))  
Restricts UMLDiagramElements to own only UMLDiagramElements.

## UMLEdge [Class]

### Description

The most general class for UML diagram elements that are rendered as lines.

### Generalizations

[Edge](#), [UMLDiagramElement](#)

### Association Ends

- `source` : [UMLDiagramElement](#) [1..1]{redefines [Edge::source](#)} (opposite [A UMLEdge source sourceEdge::sourceEdge](#))  
Restricts the sources of UMLEdges to UMLDiagramElements.
- `target` : [UMLDiagramElement](#) [1..1]{redefines [Edge::target](#)} (opposite [A UMLEdge target targetEdge::targetEdge](#))  
Restricts the targets of UMLEdges to UMLDiagramElements.

## UMLInheritedStateBorderKind [Enumeration]

### Description

Alternatives for rendering borders on UMLShapes that have an inherited State as modelElement.

## Literals

- dashed  
Dashed borders.
- gray  
Gray borders.

## UMLInteractionDiagram [Class]

### Description

Shows an Interaction and its elements. Also see Annex A.

### Generalizations

[UMLBehaviorDiagram](#)

### Attributes

- kind : [UMLInteractionDiagramKind](#) [1..1] =  
Indicates how an Interaction shall be shown.

### Association Ends

- modelElement : [Interaction](#) [1..1]{redefines [UMLBehaviorDiagram::modelElement](#)} (opposite [A UMLInteractionDiagram modelElement umlDiagramElement::umlDiagramElement](#))  
Restricts UMLInteractionDiagrams to showing Interactions.

## UMLInteractionDiagramKind [Enumeration]

### Description

Alternatives for diagramming Interactions.

### Literals

- sequence  
See Subclause 17.8.
- communication  
See Subclause 17.9.
- overview  
See Subclause 17.10.



- timing  
See Subclause 17.11.
- table  
See Annex D.

## UMLInteractionTableLabel [Class]

### Description

### Generalizations

[UMLLabel](#)

### Attributes

- kind : [UMLInteractionTableLabelKind](#) [1..1]

## UMLInteractionTableLabelKind [Enumeration]

### Description

### Literals

- lifelineClass
- lifelineInstance
- constraint
- messageSendingClass
- messageSendingInstance
- diagramIdentifier
- generatedInstanceName
- sequenceNumber

- weakOrder
- messageName
- parameter
- returnValue
- messageReceivingClass
- messageReceivingInstance
- otherEnd

## UMLKeywordLabel [Class]

### Description

For showing the keywords of the modelElement.

### Generalizations

[UMLLabel](#)

### Constraints

- modeElement\_mult  
UMLKeywords must have exactly one modelElement.

```
inv: modelElement->size() = 1
```

## UMLLabel [Class]

### Description

The most general class for UML shapes that are rendered only as text.

### Generalizations

[UMLShape](#)

## Specializations

[UMLAssociationEndLabel](#), [UMLMultiplicityLabel](#), [UMLKeywordLabel](#), [UMLNameLabel](#), [UMLRedefinesLabel](#), [UMLStereotypePropertyValueLabel](#), [UMLTypedElementLabel](#), [UMLInteractionTableLabel](#)

## Attributes

- text : [String](#) [1..1]  
String to be rendered.

## Constraints

- modelElement\_mult  
UMLLabels must have no more than one modelElement.

```
inv: modelElement->size() <= 1
```

- no\_icon  
UMLLabels must have the value of false for isIcon.

```
inv: isIcon=false
```

## UMLMultiplicityLabel [Class]

### Description

Shows text about MultiplicityElements.

### Generalizations

[UMLLabel](#)

### Association Ends

- modelElement : [MultiplicityElement](#) [1..1]{redefines [UMLDiagramElement::modelElement](#)} (opposite [A UMLMultiplicityElement modelElement umlDiagramElement::umlDiagramElement](#))  
Restricts UMLMultiplicityLabels to show only MultiplicityElements.

## UMLNameLabel [Class]

### Description

For showing text about NamedElements.

### Generalizations

[UMLLabel](#)

## Association Ends

- modelElement : [NamedElement](#) [1..1]{redefines [UMLDiagramElement::modelElement](#)} (opposite [A UMLNameLabel modelElement umlDiagramElement::umlDiagramElement](#))  
Restricts UMLNameLabels to be notation for NamedElements.

## UMLNavigabilityNotationKind [Enumeration]

### Description

Alternatives for showing navigability or non-navigability of associations and connectors typed associations.

### Literals

- always  
Always show navigability or non-navigability.
- oneWay  
Show navigability or non-navigability only for unidirectional associations and connectors typed by unidirectional associations.
- never  
Never show navigability or non-navigability.

## UMLObjectDiagram [Class]

### Description

See Annex A.

### Generalizations

[UMLStructureDiagram](#)

### Constraints

- no\_modelElement  
UMLObjectDiagrams must have no modelElement.

inv: modelElement->isEmpty()

## UMLPackageDiagram [Class]

### Description

See Annex A.

## Generalizations

[UMLStructureDiagram](#)

## Constraints

- no\_modelElement  
UMLPackageDiagrams must have no modelElement.

```
inv: modelElement->isEmpty()
```

## UMLProfileDiagram [Class]

### Description

See Annex A.

## Generalizations

[UMLStructureDiagram](#)

## Constraints

- no\_modelElement  
UMLProfileDiagrams must have no modelElement.

```
inv: modelElement->isEmpty()
```

## UMLRedefinesLabel [Class]

### Description

For showing redefinition.

## Generalizations

[UMLLabel](#)

## Association Ends

- modelElement : [RedefinableElement](#) [1..1]{redefines [UMLDiagramElement::modelElement](#)} (opposite [A UMLRedefines modelElement umlDiagramElement::umlDiagramElement](#))  
Restricts UMLRedefinesLabels to be notation for RedefinableElements.

## UMLShape [Class]

### Description

The most general class for UML diagram elements that are not rendered as lines.

## Generalizations

[Shape](#), [UMLDiagramElement](#)

## Specializations

[UMLAssociationOrConnectorOrLinkShape](#), [UMLCompartmentableShape](#), [UMLLabel](#)

## UMLStateMachineDiagram [Class]

### Description

Shows a StateMachine and its elements. Also see Annex A.

### Generalizations

[UMLBehaviorDiagram](#)

### Attributes

- `isCollapseStateIcon` : [Boolean](#) [1..1] = true  
Indicates whether UMLShapes for composite States shall contain a small icon distinguishing them from non-composite States.
- `inheritedStateBorder` : [UMLInheritedStateBorderKind](#) [0..1]  
Indicates how borders shall be rendered on UMLShapes that have an inherited State as modelElement.
- `isTransitionOriented` : [Boolean](#) [1..1] = false  
Indicates whether properties of Transitions shall be shown graphically.

### Association Ends

- `modelElement` : [StateMachine](#) [1..1]{redefines [UMLBehaviorDiagram::modelElement](#)} (opposite [A UMLStateMachine modelElement umlDiagramElement::umlDiagramElement](#))

### Constraints

- `isb_mult`  
`inheritedStateBorder` must have a value if the diagram shows any inherited states.

`inv:`

## UMLStateShape [Class]

### Description

### Generalizations

[UMLCompartmentableShape](#)

### Attributes

- isTabbed : [Boolean](#) [1..1] = false

### Association Ends

- modelElement : [State](#) [1..\*]{redefines [UMLDiagramElement::modelElement](#)} (opposite [A UMLStateShape\\_modelElement\\_umlDiagramElement::umlDiagramElement](#))

### Constraints

- state\_list  
UMLStateShapes may have multiple modelElements only when their outgoing Transitions have no triggers or effects, and target the same junction State that has one outgoing Transition.

```
inv: (modelElement->size() > 1) implies  
(  
    modelElement->forall(outgoing->forall(trigger->isEmpty() and  
effect->isEmpty() and  
target.oclIsKindOf(Pseudostate) and  
target.oclAsType(Pseudostate).kind = PseudostateKind::junction and  
1))  
    and modelElement.outgoing.target->asSet()->size()=1)
```

## UMLStereoTypePropertyValueLabel [Class]

### Description

For showing Property values of Stereotypes applied to UML abstract syntax elements.

### Generalizations

[UMLLabel](#)

### Association Ends

- modelElement : [Property](#) [1..1]{redefines [UMLDiagramElement::modelElement](#)} (opposite [A UMLStereoTypePropertyValueLabel\\_modelElement\\_umlDiagramElement::umlDiagramElement](#))

A Property of a Stereotype applied to the stereotypedElement.

- stereotypedElement : [Element](#) [1..1] (opposite [A UMLStereotypePropertyValueLabel stereotypedElement labelShowingStereotypeValue::labelShowingStereotypeValue](#))  
Element to which a Stereotype having the modelElement (Property) is applied.

### Constraints

- prop\_on\_stereotype  
modelElement is a Property of a Stereotype.

```
inv: modelElement->forall(classifier.ocIsKindOf(Stereotype))
```

- stereotypedElement  
Property must be on Stereotype applied to stereotypedElement.

```
inv:
```

## UMLStructureDiagram [Abstract Class]

### Description

The most general class for UMLDiagrams depicting structural elements.

### Generalizations

[UMLDiagram](#)

### Specializations

[UMLClassOrCompositeStructureDiagram](#), [UMLComponentDiagram](#), [UMLDeploymentDiagram](#), [UMLObjectDiagram](#), [UMLPackageDiagram](#), [UMLProfileDiagram](#)

## UMLStyle [Class]

### Description

The most general class for Styles in UML.

### Generalizations

[Style](#), [PackageableElement](#)

### Attributes

- fontName : [String](#) [0..1]  
Name of a font used to render strings.



- `fontSize` : [Real](#) [0..1]  
Size of a font for rendering strings, given in typographical points.

### Constraints

- `fontSize_positive`  
`fontSize` must be greater than zero.

```
inv: fontSize > 0
```

## UMLTypedElementLabel [Class]

### Description

For showing text about Slots, InstanceSpecifications, InstanceValues, or elements with a type, such as TypedElements or Connectors.

### Generalizations

[UMLLabel](#)

### Constraints

- `modelElement_mult`  
UMLTypedElementLabels must have exactly one modelElement.

```
inv: modelElement->size() = 1
```

- `modelElement_type`  
`modelElement` must be a Slot, InstanceSpecification, InstanceValue, or an element with a type, such as a TypedElement or Connector.

```
Cannot be expressed in OCL
```

## UMLUseCaseDiagram [Class]

### Description

See Annex A.

### Generalizations

[UMLBehaviorDiagram](#)

### Constraints

- `no_modelElement`  
UMLUseCaseDiagrams must have no modelElements.

```
inv: modelElement->isEmpty()
```

## B.7 Association Descriptions

### A\_UMLActivityDiagram\_modelElement\_umDiagramElement [Association]

#### Owned Ends

- umDiagramElement : [UMLActivityDiagram](#) [0..\*]{redefines [A\\_UMLBehaviorDiagram\\_modelElement\\_umDiagramElement::umDiagramElement](#)} (opposite [UMLActivityDiagram::modelElement](#))

### A\_UMLAssociationEndLabel\_modelElement\_umDiagramElement [Association]

#### Owned Ends

- umDiagramElement : [UMLAssociationEndLabel](#) [0..\*]{redefines [A\\_UMLDiagramElement\\_modelElement\\_umDiagramElement::umDiagramElement](#)} (opposite [UMLAssociationEndLabel::modelElement](#))

### A\_UMLBehaviorDiagram\_modelElement\_umDiagramElement [Association]

#### Owned Ends

- umDiagramElement : [UMLBehaviorDiagram](#) [0..\*]{redefines [A\\_UMLDiagramElement\\_modelElement\\_umDiagramElement::umDiagramElement](#)} (opposite [UMLBehaviorDiagram::modelElement](#))

### A\_UMLClassifierShape\_modelElement\_umDiagramElement [Association]

#### Owned Ends

- umDiagramElement : [UMLClassifierShape](#) [0..\*]{redefines [A\\_UMLDiagramElement\\_modelElement\\_umDiagramElement::umDiagramElement](#)} (opposite [UMLClassifierShape::modelElement](#))

### A\_UMLCompartment\_elementInCompartment\_owningCompartment [Association]

#### Owned Ends

- owningCompartment : [UMLCompartment](#) [0..1]{subsets [DiagramElement::owningElement](#)} (opposite [UMLCompartment::elementInCompartment](#))

## A\_UMLCompartmentableShape\_compartmentedShape\_shape [Association]

### Owned Ends

- compartmentedShape : [UMLCompartmentableShape](#) [1..1]{subsets [DiagramElement::owningElement](#)} (opposite [UMLCompartmentableShape::compartment](#))

## A\_UMLDiagramElement\_localStyle\_styledElement [Association]

### Owned Ends

- styledElement : [UMLDiagramElement](#) [0..\*]{redefines [A\\_sharedStyle\\_styledElement::styledElement](#)} (opposite [UMLDiagramElement::localStyle](#))

## A\_UMLDiagramElement\_modelElement\_umlDiagramElement [Association]

### Owned Ends

- umlDiagramElement : [UMLDiagramElement](#) [0..\*]{subsets [A\\_modelElement\\_diagramElement::diagramElement](#)} (opposite [UMLDiagramElement::modelElement](#))

## A\_UMLDiagramElement\_ownedElement\_owningElement [Association]

### Member Ends

- [UMLDiagramElement::ownedElement](#)
- [UMLDiagramElement::owningElement](#)

## A\_UMLDiagramElement\_sharedStyle\_styledElement [Association]

### Owned Ends

- styledElement : [UMLDiagramElement](#) [0..\*]{redefines [A\\_sharedStyle\\_styledElement::styledElement](#)} (opposite [UMLDiagramElement::sharedStyle](#))

## A\_UMLDiagram\_heading\_headedDiagram [Association]

### Owned Ends

- headedDiagram : [UMLDiagram](#) [0..\*] (opposite [UMLDiagram::heading](#))

## A\_UMLEdge\_source\_sourceEdge [Association]

### Owned Ends

- sourceEdge : [UMLEdge](#) [0..\*]{redefines [A\\_source\\_sourceEdge::sourceEdge](#)} (opposite [UMLEdge::source](#))

## A\_UMLEdge\_target\_targetEdge [Association]

### Owned Ends

- targetEdge : [UMLEdge](#) [0..\*]{redefines [A\\_target\\_targetEdge::targetEdge](#)} (opposite [UMLEdge::target](#))

## A\_UMLInteractionDiagram\_modelElement\_umDiagramElement [Association]

### Owned Ends

- umlDiagramElement : [UMLInteractionDiagram](#) [0..\*]{redefines [A\\_UMLBehaviorDiagram\\_modelElement\\_umDiagramElement::umlDiagramElement](#)} (opposite [UMLInteractionDiagram::modelElement](#))

## A\_UMLMultiplicityElement\_modelElement\_umDiagramElement [Association]

### Owned Ends

- umlDiagramElement : [UMLMultiplicityLabel](#) [0..\*]{redefines [A\\_UMLDiagramElement\\_modelElement\\_umDiagramElement::umlDiagramElement](#)} (opposite [UMLMultiplicityLabel::modelElement](#))

## A\_UMLNameLabel\_modelElement\_umlDiagramElement [Association]

### Owned Ends

- umlDiagramElement : [UMLNameLabel](#) [0..\*]{redefines [A\\_UMLDiagramElement\\_modelElement\\_umlDiagramElement::umlDiagramElement](#)} (opposite [UMLNameLabel::modelElement](#))

## A\_UMLRedefines\_modelElement\_umlDiagramElement [Association]

### Owned Ends

- umlDiagramElement : [UMLRedefinesLabel](#) [0..\*]{redefines [A\\_UMLDiagramElement\\_modelElement\\_umlDiagramElement::umlDiagramElement](#)} (opposite [UMLRedefinesLabel::modelElement](#))

## A\_UMLStateMachine\_modelElement\_umlDiagramElement [Association]

### Owned Ends

- umlDiagramElement : [UMLStateMachineDiagram](#) [0..\*]{redefines [A\\_UMLBehaviorDiagram\\_modelElement\\_umlDiagramElement::umlDiagramElement](#)} (opposite [UMLStateMachineDiagram::modelElement](#))

## A\_UMLStateShape\_modelElement\_umlDiagramElement [Association]

### Owned Ends

- umlDiagramElement : [UMLStateShape](#) [0..\*]{redefines [A\\_UMLDiagramElement\\_modelElement\\_umlDiagramElement::umlDiagramElement](#)} (opposite [UMLStateShape::modelElement](#))

## A\_UMLStereotypePropertyValueLabel\_modelElement\_umlDiagramElement [Association]

### Owned Ends

- umlDiagramElement : [UMLStereotypePropertyValueLabel](#) [0..\*]{redefines [A\\_UMLDiagramElement\\_modelElement\\_umlDiagramElement::umlDiagramElement](#)} (opposite [UMLStereotypePropertyValueLabel::modelElement](#))

## **A\_UMLStereotypePropertyValueLabel\_stereotypedElement\_labelShowingStereotypeValue [Association]**

### **Owned Ends**

- labelShowingStereotypeValue : [UMLStereotypePropertyValueLabel](#) [0..\*] (opposite [UMLStereotypePropertyValueLabel::stereotypedElement](#))

# Annex C: Keywords

(normative)

UML keywords are reserved words that are an integral part of the UML notation and normally appear as text annotations attached to a UML graphic element or as part of a text line in a UML diagram. These words have special significance in the context in which they are defined and, therefore, cannot be used to name user-defined model elements where such naming would result in ambiguous interpretation of the model. For example, the keyword “trace” is a system-defined stereotype of Abstraction and, therefore, cannot be used to define any user-defined stereotype.

In UML, keywords are used for four different purposes:

1. To distinguish a particular UML concept (metaclass) from others sharing the same general graphical form. For instance, the «interface» keyword in the header box of a classifier rectangle is used to distinguish an Interface from other kinds of Classifiers.
2. To distinguish a particular kind of relationship between UML concepts (meta-association) from other relationships sharing the same general graphical form. For example, dashed lines between elements are used for a number of different relationships, including Dependencies, relationships between UseCases and an extending UseCases, and so on.
3. To specify the value of some modifier attached to a UML concept (meta-attribute value). Thus, the keyword «singleExecution» appearing within an Activity signifies that the “isSingleExecution” attribute of that Activity is true.
4. To indicate a Standard Stereotype (see Clause 22). For example, the «ModelLibrary» keyword attached to a package identifies that the package contains a set of model elements intended to be shared by multiple models.

Keywords are always enclosed in guillemets («keyword»), which serve as visual cues to more readily distinguish when a keyword is being used.

**NOTE.** Guillemets are a special kind of quotation marks and should not be confused with or replaced by duplicated “greater than” (>>) or “less than” (<<) symbols, except in situations where the available character set may not include guillemets.

In addition to identifying keywords, guillemets are also used to distinguish the usage of stereotypes defined in user profiles. This means that:

- Not all words appearing between guillemets are necessarily keywords (i.e., reserved words), and
- words appearing in guillemets do not necessarily represent stereotypes.

If multiple keywords and/or stereotype names apply to the same model element, each stereotype may be enclosed in a separate pair of guillemets and listed one after the other. Alternatively they all appear between the same pair of guillemets, separated by commas:

“«” <label> [“,” <label>]\* “»”

where:

<label> ::= <keyword> | <stereotype-label>

Keywords are context sensitive and, in a few cases, the same keyword is used for different purposes in different contexts. For instance, the «create» keyword can appear next to an operation name to indicate that it is a constructor operation, and it can also be used to label a Usage dependency between two Classes to indicate that one Class creates instances of the other. This means

that it is possible in principle to use a keyword for a user-defined stereotype (provided that it is used in a context that does not conflict with the keyword context). However, such practices are discouraged as they are likely to lead to confusion.

The keywords currently defined as part of standard UML are specified in Table C.1, sorted in alphabetical order. The following is the interpretation of the individual columns in this table:

- *Keyword* provides the exact spelling of the keyword (without the guillemets).
- *Metamodel Element* specifies the element of the UML metamodel (either a metaclass or a metaclass feature) that the keyword denotes.
- *Semantics* gives a brief description of the semantics of the keyword (see further explanations below); more detailed explanations are provided in the Notation clauses of the corresponding metaclass description. The following formats are used:
  1. If the entry contains the name of a UML metaclass, this indicates that the keyword is simply used to identify the corresponding metaclass.
  2. If the entry is a constraint (usually but not necessarily an OCL expression), it specifies a constraint that applies to metamodel elements that are tagged with that keyword.
  3. If the entry is in the form “standard stereotype” it means that the keyword represents a stereotype. In those cases, the more detailed description of the semantics can be found in clause 22, “Standard Profile.”
- *Notation Placement* indicates where the keyword appears (see further explanations below). The following conventions are used to specify the notation placement:
  1. “box header” means that the keyword appears in the name compartment of a classifier rectangle.
  2. “list-box header” means that the keyword is used as a header on a list box appearing as part of a classifier specification.
  3. “dashed-line label” means that the keyword is used as a label on some dashed line, such as a Dependency.
  4. “line label” means the keyword is used as a label on some line.
  5. “inline label” means that the keyword appears as part of a text line (usually at the front), such as an attribute definition.
  6. “between braces” means that the keyword appears between “curly” brackets (similar to the constraint notation) and is used to select the value of some property of a metaclass.
  7. “swimlane header” means that the keyword appears as the header of a swimlane in an activity diagram.
  8. “note label” means that the keyword is used in a note symbol.

**Table C.1**

<b>Keyword</b>	<b>Metamodel Element</b>	<b>Semantics</b>	<b>Notation Placement</b>
abstraction	Abstraction	Abstraction	box header
access	PackageImport	visibility <> #public	dashed-line label
activity	Activity	Activity	box header
actor	Actor	Actor	box header
after	TimeEvent	isRelative = true	inline label
all	AnyReceiveEvent	Any event	inline label
apply	ProfileApplication	Package::appliedProfile->collect(importedProfile)	dashed-line label
artifact	Artifact	Artifact	box header
artifacts	Component	manifesting Artifacts	list-box header
at	TimeEvent	isRelative = false	inline label
attribute	ActivityPartition::represents	ActivityPartition::represents->forAll( r   r.ocllsKindOf(Property)	swimlane header



Auxiliary	Classifier	Standard stereotype	box header
BuildComponent	Component	Standard stereotype	box header
Call	Usage	Standard stereotype	dashed-line label
centralBuffer	CentralBufferNode	CentralBufferNode	box header
class	ActivityPartition::represents	ActivityPartition::represents->forAll( r   r.oclIsKindOf(Class))	swimlane header
component	Component	Component	box header
Create	Usage	Standard stereotype	dashed-line label
Create	BehavioralFeature	Standard stereotype	inline label
create	Dependency	Applies only to dependencies between an InstanceSpec and Parameter of an operation, indicating a return parameter of an Operation that is a constructor.	dashed-line label
datastore	DataStoreNode	DataStoreNode	box header
dataType	DataType	DataType	box header
decisionInput	Comment attached to Decision	annotatedElement.decisionInput.name = body, or annotatedElement.decisionInput.body = body (for opaque behaviors)	note label
decisionInputFlow	ActivityEdge	target.decisionInputFlow = self	line label
delegate	Connector	Connector::kind = #delegation	connector label
deploy	Connector	Deployment	dashed-line label
deployment spec	DeploymentSpecification	DeploymentSpecification	box header
Derive	Abstraction	Standard stereotype	dashed-line label
Destroy	BehavioralFeature	Standard stereotype	inline label
device	Device	Device	box header
Document	Artifact	Standard stereotype	box header
element access	ElementImport	not (visibility = #public)	dashed-line label
element import	ElementImport	visibility = #public	dashed-line label
Entity	Component	Standard stereotype	box header
enumeration	Enumeration	Enumeration	box header
Executable	Artifact	Standard stereotype	box header
executionEnvironment	ExecutionEnvironment	ExecutionEnvironment	box header
extend	Extend	Extend	dashed-line label
extended	Region	Region::extendedRegion->notEmpty()	between braces
extended	StateMachine	StateMachine::redefinedBehavior->notEmpty()	between braces
external	ActivityPartition	ActivityPartition::isExternal = true	swimlane header
file	Artifact	Standard stereotype	box header
Focus	Class	Standard stereotype	box header
Framework	Package	Standard stereotype	box header
from	Trigger	to show port name	inline label
Implement	Component	Standard stereotype	box header
ImplementationClass	Class	Standard stereotype	box header
import	PackageImport	visibility = #public	dashed-line label
include	Include	Include	dashed-line label
information	InformationItem	InformationItem	box header
Instantiate	Usage	Standard stereotype	dashed-line label
interface	Interface	Interface	box header
Library	Artifact	Standard stereotype	box header
localPostcondition	Constraint	Action::localPostcondition	box header
localPrecondition	Constraint	Action::localPrecondition	box header
manifest	Manifestation	Manifestation	dashed-line label
merge	PackageMerge	PackageMerge	dashed-line label
Metaclass	Class	Standard stereotype	box header
Metamodel	Model	Standard stereotype	box header
model	Model	Model	box header
ModelLibrary	Package	Standard stereotype	box header

multicast	ObjectFlow	ObjectFlow::isMulticast	flow label
multireceive	ObjectFlow	ObjectFlow::isMultireceive	flow label
occurrence	Collaboration	(Behavior::context) from a Collaboration to the owning BehavioredClassifier, indicating that the collaboration represents the use of a classifier.	dashed-line label
postcondition	Constraint	Behavior::postcondition	box header
precondition	Constraint	Behavior::precondition	box header
primitive	PrimitiveType	PrimitiveType	box header
Process	Component	Standard stereotype	box header
profile	Profile	Profile	box header
provided interfaces	Component	Component::provided	list-box header
Realization	Classifier	Standard stereotype	box header
realizations	Component	Component::realizations->collect(realizingClassifier)	list-box header
reference	ElementImport	Profile::metaclassReference	dashed-line label
reference	PackageImport	Profile::metamodelReference	dashed-line label
Refine	Abstraction	Standard stereotype	dashed-line label
representation	Classifier	InformationFlow::conveyed	dashed-line label
represents	Collaboration	(Behavior::context) from a Collaboration to the owning BehavioredClassifier; collaboration is USED in the classifier	dashed-line label
required interfaces	Component	Component::required	list-box header
Responsibility	usage	Standard stereotype	dashed-line label
Script	Artifact	Standard stereotype	box header
selection	Behavior	ObjectFlow::selection	box header
Send	Usage	Standard stereotype	dashed-line label
Service	Component	Standard stereotype	box header
signal	Signal	Signal	box header
singleExecution	Activity	Activity::isSingleExecution = true	inside box
Source	Artifact	Standard stereotype	box header
Specification	Classifier	Standard stereotype	box header
statemachine	BehavioredClassifier	BehavioredClassifier::self.ocIsKindOf(StateMachine)	box header
Stereotype	Stereotype	Stereotype	box header
strict	ProfileApplication	isStrict = true	dashed-line label
structured	StructuredActivityNode	StructuredActivityNode	box header
substitute	Substitution	Substitution	dashed-line label
Subsystem	Component	Standard stereotype	box header
SystemModel	Model	Standard stereotype	box header
Trace	Abstraction	Standard stereotype	dashed-line label
transformation	Behavior	ObjectFlow::transformation	box header
Type	Class	Standard stereotype	box header
use	Usage	Usage	dashed-line label
Utility	Class	Standard stereotype	box header
when	ChangeEvent	ChangeEvent::changeExpression	inline label

# Annex D: Tabular Notations

(normative)

This annex describes optional tabular notations for UML behavioral diagrams, that some vendors or users may want to use as alternatives to UML's graphic notation. Although this annex describes tabular notations for sequence diagrams, the approach may also be applied to other kinds of behavioral diagrams.

This annex describes an optional tabular notation for sequence diagrams. The table row descriptions for this notation follow:

1. **Lifeline Class:** Designates Class name of Lifeline. If there is no Class name on the Lifeline symbol, this class name is omitted.
2. **Lifeline Instance:** Designates Instance name of Lifeline. If there is no Instance name on the Lifeline symbol, this instance name is omitted.
3. **Constraint:** Designates some kind of constraint. For example, indication of oblique line is denoted as “{delay}.” To represent CombinedFragments, those operators are denoted with an index adorned by square bracket. In a case of InteractionUse, it is shown as parenthesized “Diagram ID,” which designates referred Interaction Diagram, with “ref” tag, like “ref(M.sq).”
4. **Message Sending Class:** Designates the message sending class name for each incoming arrow.
5. **Message Sending Instance:** Designates the message sending instance name for each incoming arrow. In a case of Gate message that is outgoing message from InteractionUse, it is shown as parenthesized “Diagram ID,” which designates referred Interaction Diagram, with underscore, like “\_(M.sq).”
6. **Diagram ID:** Identifies the document that describes the corresponding sequence/communication diagram and can be the name of the file that contains the corresponding sequence or communication diagram.
7. **Generated instance name:** An identifier name that is given to each instance symbol in the sequence/communication diagram. The identifier name is unique in each document.
8. **Sequence Number:** The corresponding message number on the sequence/communication diagram.
9. **Weak Order:** Designates partial (relative) orders of events, as ordered on individual lifelines and across lifelines, given a message receive event has to occur after its message send event. Events are shown as “e” + event order + event direction (incoming or outgoing).
10. **Message name:** The corresponding message name on the sequence/communication diagram.
11. **Parameter:** A set of parameter variable names and parameter types of the corresponding message on the sequence/communication diagrams.
12. **Return value:** The return value type of the corresponding message on the sequence/communication diagram.
13. **Message Receiving Class:** Designates the message receiving class name for each outgoing arrow.
14. **Message Receiving Instance:** Designates the message receiving instance name for each outgoing arrow. In a case of Gate message that is outgoing message from ordinary instance symbol, it is shown as parenthesized message name with “out\_” tag, like “(out\_s).”
15. **Other End:** Designates event order of another end on the each message.

## Examples

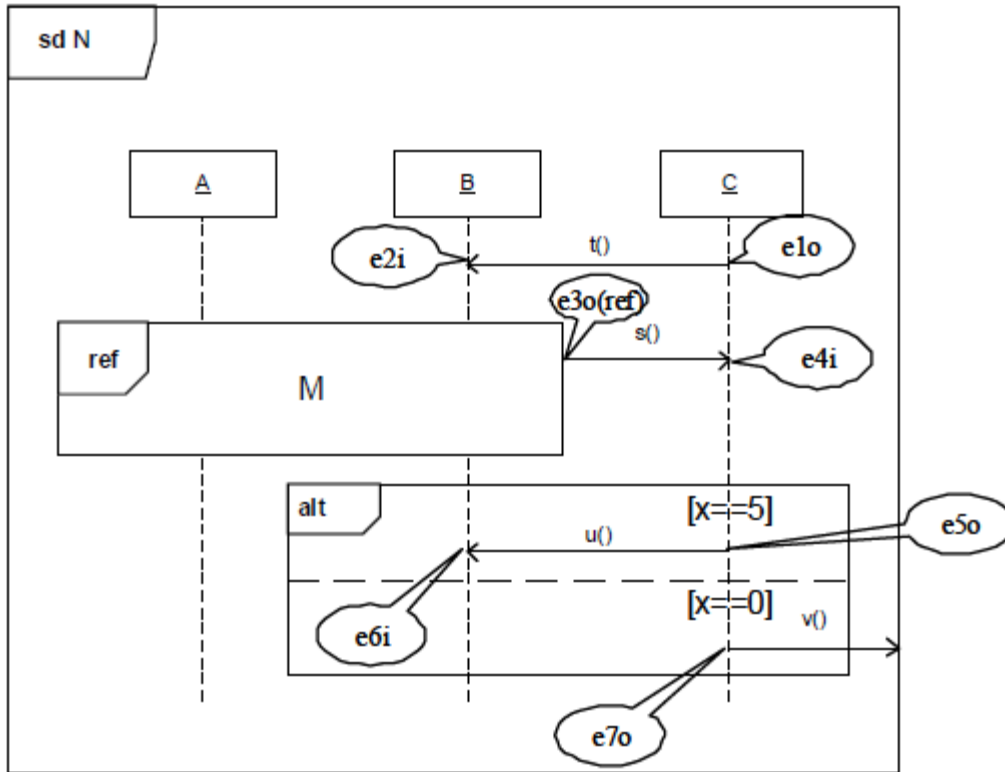


Figure D.1 Sequence diagram enhanced with identification of the Event occurrences

Table D.1 Interaction Table describing Figure D.1

Lifeline Class	Lifeline Instance	Constraint	Message Sending Class	Message Sending Instance	Diagram ID	Generated Instance Name	Sequence Number	Weak Order	Message Name	Parameter	Return Value	Message Receiving Class	Message Receiving Instance	Other End
	C				N.sq			e1o	t				B	e2i
	B			C	N.sq			e2i	t					e1o
	A	Ref(M.sq)			N.sq			e3o(ref)	s				C	e4i
	B	Ref(M.sq)			N.sq			e3o(ref)	s				C	e4i
	C			_(M.sq)	N.sq			e4i	s					e3o(ref)
	C	alt[1]x==5			N.sq			e5o	u				B	e6i
	B	alt[1]x==5			N.sq			e6i	u					e5o
	C	alt[2]x==0			N.sq			e7o	v				out_v	

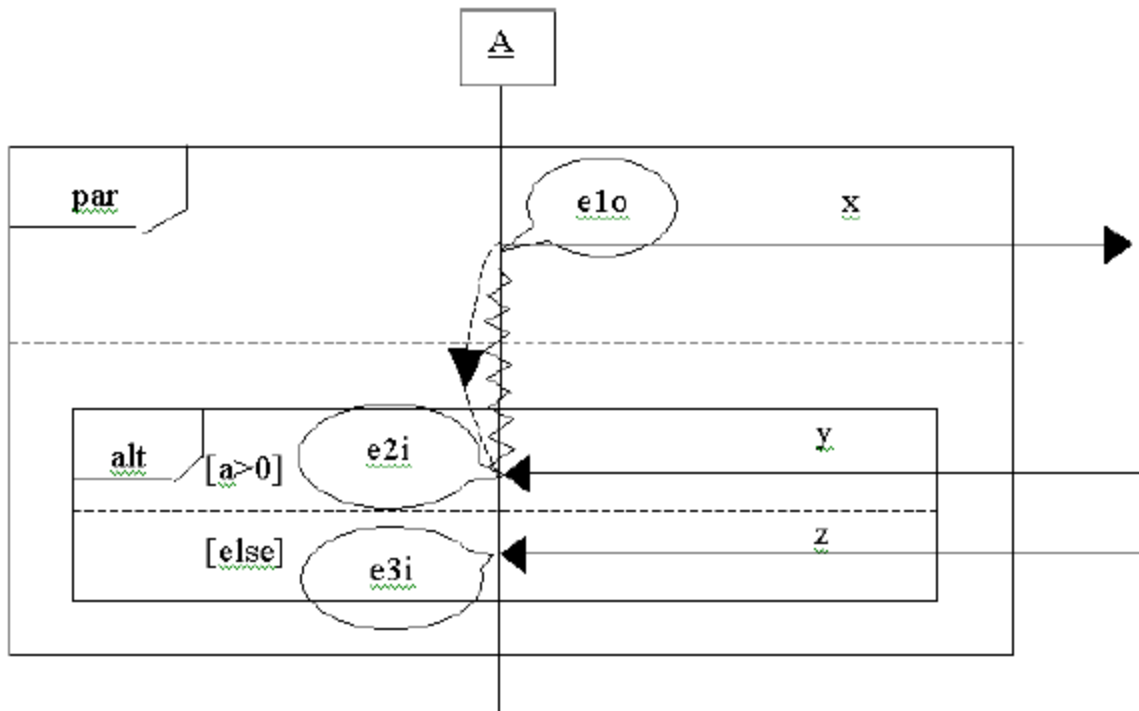


Figure D.2 Sequence diagram with guards, parallel composition and alternatives

Table D.2 Interaction Table for Figure D.2

Lifeline Class	Lifeline Instance	Const-aint	Message Sending Classs	Message Sending Instance	Diagram ID	Generated Instance Name	Sequence Number	Weak Order	Message Name	Para-meter	Return Value	Message Receiving Class	Message Receiving Instance	Other End
	A	par[1]			para.sq			e1o	x					
	A	par[2].alt[1] {after e1o} a > 0		in_y	para.sq			e2i	y				out_x	
	A	par[2].alt[2] else		in_z	para.sq			e3i	z					

# Annex E: XMI Serialization and Schema

(normative)

UML 2 models are serialized in XMI 2 according to the rules specified by the MOF 2 XMI Mapping Specification.

As is common policy for OMG, the normative representation of MOF 2 and UML 2 models is an XMI file. The normative XMI document for UML 2 itself consists of a single XMI document. There are related XMI documents to specify the StandardProfile and the UML Diagram Interchange model. PrimitiveTypes, on which UML 2 depends and other specifications may depend, is specified in a separate XMI document.

XMI allows the use of tags to tailor the schemas that are produced and the documents that are produced using XMI. The following are the tag settings that appear in the UML 2 metamodel for the XMI interchange of UML 2 models:

- tag “org.omg.xmi.nsPrefix” set to “uml”

No other tags are explicitly set, which means they assume their default values as documented in the MOF 2 XMI Mappings Specification.

The xmi:ids for metamodel elements in UML 2.5 are the same as for equivalent elements in UML 2.4.1, except for those ids corresponding to associations and properties whose name changed between UML 2.4.1 and UML 2.5. The metaclasses and associations in UML 2.5 are organized in a package structure that corresponds to the specification clause structure. All of these packages are imported into the top-level UML package, so that all metamodel elements can be referred to unqualified in the top-level package.

Changes that have been made to the metamodel from 2.4.1 include: the package organization described above; the relaxation (to 0) of some lower multiplicities in order to represent default values; the introduction of {ordered} on some properties in order to clarify the semantics; the renaming of some association-owned properties and corresponding associations; and identifying LoopNodes as the owner of their loopVariables. As a consequence, UML 2.5 and UML 2.4.1 model files are interchangeable by substituting version numbers, apart from loopVariables of LoopNodes, which could not be interchanged in a standard way in UML 2.4.1. Apart from such LoopNodes, a UML 2.4.1 tool should be able to load a UML 2.5 model if the version numbers are edited back to 20110701. A conforming UML 2.5 tool shall be able to load and save either UML 2.5 or UML 2.4.1 models.

**NOTE.** Because UML 2.5 has many more OCL constraints formalized, it may be the case that a model that validates in a UML 2.4.1 tool may not validate in a UML 2.5 tool.

## Index

A_action_actionExecutionSpecification.....	669	A_connectableElement_templateParameter_para	
A_action_interaction.....	669	meteredElement .....	245
A_actual_templateParameterSubstitution.....	53	A_connection_state.....	380
A_actualGate_interactionUse .....	669	A_connectionPoint_state .....	379
A_addition_include.....	691	A_connectionPoint_stateMachine .....	379
A_annotatedElement_comment.....	53	A_connector_message .....	671
A_appliedProfile_profileApplication .....	291	A_constrainedElement_constraint.....	53
A_argument_interactionUse .....	669	A_constrainingClassifier_classifierTemplatePara	
A_argument_invocationAction.....	569	meter .....	160
A_argument_message.....	670	A_containedEdge_inGroup.....	452
A_association_clearAssociationAction .....	569	A_containedNode_inGroup.....	453
A_attribute_classifier.....	159	A_context_action.....	572
A_before_toAfter.....	670	A_context_behavior.....	312
A_behavior_behaviorExecutionSpecification .	670	A_contract_connector.....	245
A_behavior_callBehaviorAction .....	570	A_contract_interfaceRealization.....	186
A_behavior_opaqueExpression .....	89	A_contract_substitution .....	161
A_body_clause.....	571	A_conveyed_conveyingFlow .....	718
A_bodyCondition_bodyContext.....	159	A_covered_coveredBy.....	671
A_bodyOutput_clause.....	570	A_covered_events.....	671
A_bodyOutput_loopNode.....	570	A_covered_stateInvariant .....	671
A_bodyPart_loopNode .....	570	A_decider_clause.....	572
A_cfragmentGate_combinedFragment.....	670	A_decider_loopNode .....	573
A_changeExpression_changeEvent.....	312	A_decisionInput_decisionNode.....	453
A_classifier_createObjectAction .....	571	A_decisionInputFlow_decisionNode.....	453
A_classifier_enumerationLiteral.....	186	A_decomposedAs_lifeline.....	672
A_classifier_instanceSpecification .....	159	A_default_templateParameter.....	54
A_classifier_readExtentAction.....	571	A_defaultValue_owningParameter.....	161
A_classifier_readIsClassifiedObjectAction.....	571	A_defaultValue_owningProperty .....	161
A_classifier_templateParameter_parameteredEle		A_deferrableTrigger_state .....	380
ment.....	160	A_definingEnd_connectorEnd.....	245
A_classifierBehavior_behavoredClassifier ....	185	A_definingFeature_slot.....	161
A_clause_conditionalNode .....	572	A_deployedArtifact_deploymentForArtifact...	708
A_clientDependency_client.....	53	A_deployedElement_deploymentTarget .....	709
A_collaborationRole_collaboration.....	244	A_deployment_location.....	709
A_collaborationUse_classifier.....	160	A_destroyAt_linkEndDestructionData .....	573
A_collection_reduceAction .....	572	A_doActivity_state .....	380
A_condition_extend.....	692	A_edge_activity .....	453
A_condition_parameterSet .....	160	A_edge_inPartition .....	454
A_configuration_deployment .....	708	A_edge_inStructuredNode.....	573
A_conformance_specificMachine .....	379	A_effect_transition .....	380
		A_elementImport_importingNamespace.....	54

A_end_connector .....	246	A_generalMachine_protocolConformance .....	382
A_end_linkEndData .....	574	A_generalOrdering_interactionFragment .....	673
A_end_readLinkObjectEndAction .....	574	A_group_inActivity .....	454
A_end_role .....	246	A_guard_activityEdge .....	455
A_endData_createLinkAction .....	573	A_guard_interactionOperand .....	673
A_endData_destroyLinkAction .....	574	A_guard_transition .....	382
A_endData_linkAction .....	574	A_handler_protectedNode .....	455
A_endType_association .....	245	A_handlerBody_exceptionHandler .....	455
A_entry_connectionPointReference .....	381	A_icon_stereotype .....	291
A_entry_state .....	381	A_importedElement_import .....	55
A_event_durationObservation .....	89	A_importedMember_namespace .....	55
A_event_timeObservation .....	89	A_importedPackage_packageImport .....	55
A_event_trigger .....	312	A_include_includingCase .....	693
A_exception_raiseExceptionAction .....	575	A_incoming_target_node .....	456
A_exceptionInput_exceptionHandler .....	454	A_incoming_target_vertex .....	383
A_exceptionType_exceptionHandler .....	454	A_informationSource_informationFlow .....	718
A_executableNode_sequenceNode .....	575	A_informationTarget_informationFlow .....	719
A_execution_executionOccurrenceSpecification .....	672	A_inheritedMember_inheritingClassifier .....	163
A_exit_connectionPointReference .....	381	A_inheritedParameter_redefinableTemplateSignature .....	163
A_exit_state .....	381	A_inInterruptibleRegion_node .....	455
A_expr_duration .....	89	A_inPartition_node .....	456
A_expr_timeExpression .....	90	A_input_action .....	576
A_extend_extension .....	692	A_inputElement_regionAsInput .....	576
A_extendedCase_extend .....	692	A_inputValue_linkAction .....	576
A_extendedRegion_region .....	382	A_inputValue_opaqueAction .....	576
A_extendedSignature_redefinableTemplateSignature .....	162	A_insertAt_addStructuralFeatureValueAction .....	577
A_extendedStateMachine_stateMachine .....	382	A_insertAt_addVariableValueAction .....	577
A_extension_metaclass .....	246	A_insertAt_linkEndCreationData .....	577
A_extensionLocation_extension .....	692	A_instance_instanceValue .....	164
A_extensionPoint_useCase .....	693	A_inState_objectNode .....	456
A_feature_featuringClassifier .....	162	A_interfaceRealization_implementingClassifier .....	186
A_finish_executionSpecification .....	672	A_interruptingEdge_interrupts .....	456
A_first_testIdentityAction .....	575	A_invariant_stateInvariant .....	674
A_formal_templateParameterSubstitution .....	54	A_joinSpec_joinNode .....	456
A_formalGate_interaction .....	672	A_lifeline_interaction .....	674
A_fragment_enclosingInteraction .....	673	A_localPostcondition_action .....	577
A_fragment_enclosingOperand .....	673	A_localPrecondition_action .....	578
A_fromAction_actionInputPin .....	575	A_loopVariable_loopNode .....	578
A_general_classifier .....	162	A_loopVariableInput_loopNode .....	578
A_general_generalization .....	162	A_lowerValue_owningLower .....	55
A_generalization_specific .....	163	A_manifestation_artifact .....	709
A_generalizationSet_generalization .....	163	A_mapping_abstraction .....	56



A_max_durationInterval.....	90	A_operation_templateParameter_parameteredElement.....	164
A_max_interval.....	90	A_opposite_property.....	164
A_max_timeInterval.....	91	A_outgoing_source_node.....	457
A_maxint_interactionConstraint.....	674	A_outgoing_source_vertex.....	383
A_member_memberNamespace.....	56	A_output_action.....	583
A_memberEnd_association.....	246	A_outputElement_regionAsOutput.....	582
A_mergedPackage_packageMerge.....	291	A_outputValue_opaqueAction.....	582
A_message_considerIgnoreFragment.....	674	A_ownedActual_owningTemplateParameterSubstitution.....	56
A_message_interaction.....	675	A_ownedAttribute_artifact.....	710
A_message_messageEnd.....	675	A_ownedAttribute_class.....	247
A_metaclassReference_profile.....	291	A_ownedAttribute_datatype.....	187
A_metamodelReference_profile.....	292	A_ownedAttribute_interface.....	187
A_method_specification.....	164	A_ownedAttribute_owningSignal.....	187
A_min_durationInterval.....	91	A_ownedAttribute_structuredClassifier.....	247
A_min_interval.....	91	A_ownedBehavior_behavoredClassifier.....	187
A_min_timeInterval.....	92	A_ownedComment_owningElement.....	57
A_minint_interactionConstraint.....	675	A_ownedConnector_structuredClassifier.....	248
A_nameExpression_namedElement.....	56	A_ownedDefault_templateParameter.....	57
A_navigableOwnedEnd_association.....	246	A_ownedElement_owner.....	57
A_nestedArtifact_artifact.....	709	A_ownedEnd_extension.....	292
A_nestedClassifier_interface.....	186	A_ownedEnd_owningAssociation.....	248
A_nestedClassifier_nestingClass.....	247	A_ownedLiteral_enumeration.....	188
A_nestedNode_node.....	710	A_ownedMember_namespace.....	57
A_nestedPackage_nestingPackage.....	292	A_ownedOperation_artifact.....	710
A_newClassifier_reclassifyObjectAction.....	578	A_ownedOperation_class.....	248
A_node_activity.....	457	A_ownedOperation_datatype.....	188
A_node_inStructuredNode.....	579	A_ownedOperation_interface.....	188
A_object_clearAssociationAction.....	579	A_ownedParameter_behavior.....	313
A_object_readIsClassifiedObjectAction.....	579	A_ownedParameter_operation.....	165
A_object_readLinkObjectEndAction.....	579	A_ownedParameter_ownerFormalParam.....	165
A_object_readLinkObjectEndQualifierAction.....	580	A_ownedParameter_signature.....	58
A_object_reclassifyObjectAction.....	580	A_ownedParameteredElement_owningTemplateParameter.....	58
A_object_startClassifierBehaviorAction.....	580	A_ownedParameterSet_behavior.....	313
A_object_startObjectBehaviorAction.....	581	A_ownedParameterSet_behavioralFeature.....	165
A_object_structuralFeatureAction.....	581	A_ownedPort_encapsulatedClassifier.....	248
A_object_unmarshallAction.....	581	A_ownedReception_class.....	249
A_observation_duration.....	92	A_ownedReception_interface.....	188
A_observation_timeExpression.....	92	A_ownedRule_context.....	58
A_oldClassifier_reclassifyObjectAction.....	581	A_ownedStereotype_owningPackage.....	292
A_onPort_invocationAction.....	582	A_ownedTemplateSignature_classifier.....	165
A_operand_combinedFragment.....	675	A_ownedTemplateSignature_template.....	58
A_operand_expression.....	92		
A_operation_callEvent.....	313		
A_operation_callOperationAction.....	582		

A_ownedType_package.....	293	A_redefinedClassifier_classifier.....	168
A_ownedUseCase_classifier.....	166	A_redefinedConnector_connector.....	251
A_packagedElement_component.....	249	A_redefinedEdge_activityEdge.....	458
A_packagedElement_owningPackage.....	293	A_redefinedElement_redefinableElement.....	168
A_packageImport_importingNamespace.....	59	A_redefinedInterface_interface.....	189
A_packageMerge_receivingPackage.....	293	A_redefinedNode_activityNode.....	458
A_parameter_activityParameterNode.....	457	A_redefinedOperation_operation.....	168
A_parameter_templateSignature.....	59	A_redefinedPort_port.....	251
A_parameteredElement_templateParameter.....	59	A_redefinedProperty_property.....	168
A_parameterSet_parameter.....	166	A_redefinedState_state.....	384
A_parameterSubstitution_templateBinding.....	59	A_redefinedTransition_transition.....	384
A_part_structuredClassifier.....	249	A_redefinitionContext_redefinableElement....	169
A_partition_activity.....	457	A_redefinitionContext_region.....	384
A_partWithPort_connectorEnd.....	249	A_redefinitionContext_state.....	385
A_port_trigger.....	313	A_redefinitionContext_transition.....	385
A_postcondition_behavior.....	314	A_reducer_reduceAction.....	584
A_postCondition_owningTransition.....	383	A_referred_protocolTransition.....	385
A_postcondition_postContext.....	166	A_refersTo_interactionUse.....	676
A_powertypeExtent_powertype.....	166	A_region_state.....	385
A_precondition_behavior.....	314	A_region_stateMachine.....	386
A_precondition_preContext.....	167	A_relatedElement_relationship.....	59
A_preCondition_protocolTransition.....	383	A_removeAt_removeStructuralFeatureValueActi on.....	584
A_predecessorClause_successorClause.....	583	A_removeAt_removeVariableValueAction....	584
A_profile_stereotype.....	294	A_replyToCall_replyAction.....	585
A_profileApplication_applyingPackage.....	293	A_replyValue_replyAction.....	585
A_protocol_interface.....	189	A_representation_classifier.....	169
A_protocol_port.....	250	A_represented_representation.....	720
A_provided_component.....	250	A_represents_activityPartition.....	458
A_provided_port.....	250	A_represents_lifeline.....	676
A_qualifier_associationEnd.....	167	A_request_sendObjectAction.....	585
A_qualifier_linkEndData.....	583	A_required_component.....	251
A_qualifier_qualifierValue.....	583	A_required_port.....	252
A_qualifier_readLinkObjectEndQualifierAction .....	584	A_result_acceptEventAction.....	585
A_raisedException_behavioralFeature.....	167	A_result_callAction.....	586
A_raisedException_operation.....	167	A_result_clearStructuralFeatureAction.....	586
A_realization_abstraction_component.....	250	A_result_conditionalNode.....	586
A_realization_abstraction_flow.....	719	A_result_createLinkObjectAction.....	586
A_realizingActivityEdge_informationFlow....	719	A_result_createObjectAction.....	587
A_realizingClassifier_componentRealization.	251	A_result_loopNode.....	587
A_realizingConnector_informationFlow.....	719	A_result_opaqueExpression.....	93
A_realizingMessage_informationFlow.....	720	A_result_readExtentAction.....	587
A_receiveEvent_endMessage.....	676	A_result_readIsClassifiedObjectAction.....	587
A_redefinedBehavior_behavior.....	314	A_result_readLinkAction.....	588

A_result_readLinkObjectEndAction .....	588	A_subgroup_superGroup .....	459
A_result_readLinkObjectEndQualifierAction .....	588	A_subject_useCase .....	693
A_result_readSelfAction .....	588	A_submachineState_submachine .....	386
A_result_readStructuralFeatureAction .....	589	A_subpartition_superPartition .....	459
A_result_readVariableAction .....	589	A_subsettedProperty_property .....	170
A_result_reduceAction .....	589	A_substitution_substitutingClassifier .....	170
A_result_testIdentityAction .....	589	A_subvertex_container .....	386
A_result_unmarshallAction .....	590	A_superClass_class .....	252
A_result_valueSpecificationAction .....	590	A_supplier_supplierDependency .....	61
A_result_writeStructuralFeatureAction .....	590	A_target_callOperationAction .....	593
A_returnInformation_acceptCallAction .....	590	A_target_destroyObjectAction .....	593
A_returnInformation_replyAction .....	591	A_target_directedRelationship .....	61
A_returnValue_interactionUse .....	677	A_target_sendObjectAction .....	593
A_returnValueRecipient_interactionUse .....	676	A_target_sendSignalAction .....	593
A_role_structuredClassifier .....	252	A_templateBinding_boundElement .....	61
A_roleBinding_collaborationUse .....	252	A_test_clause .....	594
A_second_testIdentityAction .....	591	A_test_loopNode .....	594
A_selection_objectFlow .....	458	A_toBefore_after .....	678
A_selection_objectNode .....	459	A_transformation_objectFlow .....	460
A_selector_lifeline .....	677	A_transition_container .....	387
A_sendEvent_endMessage .....	677	A_trigger_acceptEventAction .....	594
A_setupPart_loopNode .....	591	A_trigger_transition .....	387
A_signal_broadcastSignalAction .....	591	A_type_collaborationUse .....	253
A_signal_reception .....	189	A_type_connector .....	253
A_signal_sendSignalAction .....	592	A_type_extensionEnd .....	294
A_signal_signalEvent .....	314	A_type_operation .....	170
A_signature_message .....	677	A_type_typedElement .....	61
A_signature_templateBinding .....	60	A_UMLActivityDiagram_modelElement_umlDiagramElement .....	768
A_slot_owningInstance .....	169	A_UMLAssociationEndLabel_modelElement_umlDiagramElement .....	768
A_source_directedRelationship .....	60	A_UMLBehaviorDiagram_modelElement_umlDiagramElement .....	768
A_specification_durationConstraint .....	93	A_UMLClassifierShape_modelElement_umlDiagramElement .....	768
A_specification_intervalConstraint .....	93	A_UMLCompartment_elementInCompartment_owningCompartment .....	768
A_specification_owningConstraint .....	60	A_UMLCompartmentableShape_compartmentedShape_shape .....	769
A_specification_owningInstanceSpec .....	169	A_UMLDiagram_heading_headedDiagram .....	770
A_specification_timeConstraint .....	94	A_UMLDiagramElement_localStyle_styledElement .....	769
A_start_executionSpecification .....	678	A_UMLDiagramElement_modelElement_umlDiagramElement .....	769
A_stateInvariant_owningState .....	386		
A_structuralFeature_structuralFeatureAction .....	592		
A_structuredNode_activity .....	459		
A_structuredNodeInput_structuredActivityNode .....	592		
A_structuredNodeOutput_structuredActivityNode .....	592		
A_subExpression_owningExpression .....	94		

A\_UMLDiagramElement\_ownedElement\_ownin  
   gElement ..... 769  
 A\_UMLDiagramElement\_sharedStyle\_styledEle  
   ment..... 769  
 A\_UMLEdge\_source\_sourceEdge ..... 770  
 A\_UMLEdge\_target\_targetEdge ..... 770  
 A\_UMLInteractionDiagram\_modelElement\_uml  
   DiagramElement ..... 770  
 A\_UMLMultiplicityElement\_modelElement\_uml  
   DiagramElement ..... 770  
 A\_UMLNameLabel\_modelElement\_umlDiagram  
   Element ..... 771  
 A\_UMLRedefines\_modelElement\_umlDiagramEl  
   ement..... 771  
 A\_UMLStateMachine\_modelElement\_umlDiagra  
   mElement ..... 771  
 A\_UMLStateShape\_modelElement\_umlDiagram  
   Element ..... 771  
 A\_UMLStereotypePropertyValueLabel\_modelEle  
   ment\_umlDiagramElement ..... 771  
 A\_UMLStereotypePropertyValueLabel\_stereotyp  
   edElement\_labelShowingStereotypeValue.. 772  
 A\_unmarshallType\_unmarshallAction ..... 594  
 A\_upperBound\_objectNode ..... 460  
 A\_upperValue\_owningUpper ..... 62  
 A\_utilizedElement\_manifestation..... 710  
 A\_value\_linkEndData..... 595  
 A\_value\_owningSlot..... 170  
 A\_value\_qualifierValue ..... 595  
 A\_value\_valuePin ..... 595  
 A\_value\_valueSpecificationAction ..... 595  
 A\_value\_writeStructuralFeatureAction ..... 596  
 A\_value\_writeVariableAction ..... 596  
 A\_variable\_activityScope ..... 460  
 A\_variable\_scope..... 596  
 A\_variable\_variableAction ..... 596  
 A\_weight\_activityEdge..... 460  
 A\_when\_timeEvent..... 315  
 abstraction ..... 56, 236, 719  
 Abstraction ..... 31  
 acceptCallAction ..... 591  
 AcceptCallAction..... 511  
 acceptEventAction ..... 586, 594  
 AcceptEventAction ..... 512

action..... 572, 577, 578, 583, 644, 652  
 Action..... 513  
 actionExecutionSpecification ..... 669  
 ActionExecutionSpecification ..... 644  
 actionInputPin ..... 576  
 ActionInputPin..... 515  
 activity..... 432, 435, 458, 561  
 Activity ..... 431  
 activityEdge ..... 455, 458, 460  
 ActivityEdge ..... 432  
 ActivityFinalNode..... 433  
 ActivityGroup ..... 434  
 activityNode ..... 458  
 ActivityNode..... 435  
 activityParameterNode..... 457  
 ActivityParameterNode..... 436  
 activityPartition..... 458  
 ActivityPartition..... 437  
 activityScope..... 452  
 Actor ..... 688  
 actual ..... 48  
 actualGate ..... 658  
 addition ..... 690  
 addStructuralFeatureValueAction..... 577  
 AddStructuralFeatureValueAction ..... 515  
 addVariableValueAction..... 577  
 AddVariableValueAction ..... 516  
 after ..... 652  
 aggregation..... 151  
 AggregationKind..... 131  
 alias ..... 35  
 allActions ..... 514, 525, 538, 562  
 allApplicableStereotypes ..... 286  
 allAttributes..... 137  
 allFeatures ..... 135  
 allIncludedUseCases ..... 691  
 allNamespaces..... 40  
 allOwnedElements ..... 34  
 allOwnedNodes..... 514, 562  
 allOwningPackages..... 40  
 allowSubstitutable..... 138  
 allParents..... 135  
 allPins..... 534, 535, 536  
 allRealizedInterfaces..... 136

allUsedInterfaces.....	136	callBehaviorAction .....	570
alt656		CallBehaviorAction .....	520
always .....	762	CallConcurrencyKind .....	133
ancestor .....	373	callEvent .....	313
annotatedElement.....	32	CallEvent.....	307
AnyReceiveEvent .....	305	callOperationAction.....	582, 593
appliedProfile.....	289	CallOperationAction .....	520
applyingPackage .....	289	CentralBufferNode.....	439
argument .....	532, 658, 661	cfragmentGate.....	645
artifact .....	709, 710	changeEvent.....	312
Artifact .....	703	ChangeEvent .....	308
assembly.....	240	changeExpression .....	308
assert .....	657	choice .....	365
association.....	151, 245, 247, 523, 533	class.....	144, 151, 249, 253
Association.....	229	Class.....	231
AssociationClass .....	230	classifier .. 142, 157, 159, 160, 162, 166, 168, 169,	
associationEnd .....	151	182, 528, 543, 544	
asynchCall.....	666	Classifier .....	133
asynchSignal .....	666	classifierBehavior .....	180
attribute .....	134	classifierTemplateParameter.....	161
basicProvided.....	243	ClassifierTemplateParameter .....	137
basicRequired.....	243	clause.....	525, 570, 571, 572, 594
before .....	652	Clause.....	521
behavior.....	83, 312, 313, 314, 520, 559, 645	clearAssociationAction .....	569, 579
Behavior.....	305	ClearAssociationAction .....	523
behavioralFeature.....	165, 167	clearStructuralFeatureAction .....	586
BehavioralFeature .....	131	ClearStructuralFeatureAction .....	523
behavioredClassifier .....	185, 188, 306	ClearVariableAction .....	524
BehavioredClassifier.....	180	client.....	33
behaviorExecutionSpecification .....	670	clientDependency.....	39
BehaviorExecutionSpecification.....	644	collaboration .....	244
belongsToPSM.....	367	Collaboration.....	232
body.....	31, 83, 310, 522, 540	collaborationRole.....	233
bodyCondition.....	144	collaborationUse .....	134, 252, 253
bodyContext.....	159	CollaborationUse .....	233
bodyOutput .....	522, 537	collection.....	552
bodyPart .....	537	combinedFragment .....	670, 675
booleanValue .....	78, 88	CombinedFragment.....	645
boundElement .....	46	comment.....	53
break.....	656	Comment.....	31
broadcastSignalAction .....	592	communication.....	758
BroadcastSignalAction .....	517	CommunicationPath.....	704
callAction .....	586	compartment .....	754
CallAction.....	518	compartmentedShape.....	769

compatibleWith.....	37	ControlFlow .....	439
complete .....	665	ControlNode.....	440
component.....	249, 250, 251	conveyed .....	716
Component.....	234	conveyingFlow.....	718
componentRealization.....	251	covered.....	655, 667, 668
ComponentRealization.....	236	coveredBy .....	659
composite .....	131	create .....	149
concurrency.....	131	createLinkAction.....	573
concurrent .....	133	CreateLinkAction.....	526
condition .....	150, 688	createLinkObjectAction .....	587
conditionalNode.....	572, 586	CreateLinkObjectAction .....	527
ConditionalNode .....	524	createMessage .....	666
configuration .....	705	createObjectAction .....	571, 587
conformance.....	360	CreateObjectAction.....	527
conformsTo .....	51, 135	critical .....	657
ConnectableElement .....	236	dashed .....	758
ConnectableElementTemplateParameter .....	237	DataStoreNode.....	440
connection .....	369	datatype .....	144, 152
connectionPoint.....	369, 372	DataType.....	180
connectionPointReference .....	381	decider.....	522, 537
ConnectionPointReference .....	358	decisionInput.....	441
connector.....	245, 246, 251, 253, 661	decisionInputFlow.....	441
Connector .....	237	decisionNode.....	453
connectorEnd .....	245, 249	DecisionNode.....	440
ConnectorEnd .....	239	decomposedAs .....	659
ConnectorKind.....	240	deepHistory .....	364
consider .....	657	default .....	47, 147, 148, 151, 152
considerIgnoreFragment .....	674	defaultValue.....	147, 152
ConsiderIgnoreFragment .....	646	deferrableTrigger .....	369
constrainedElement.....	32	definingEnd.....	239
constrainingClassifier .....	138	definingFeature .....	158
constraint.....	54, 759	delegation .....	240
Constraint.....	32	delete .....	149
containedEdge.....	434	deleteMessage .....	666
containedNode .....	434	Dependency.....	33
container.....	374, 378	deployedArtifact .....	705
containingActivity.....	434, 436, 562	DeployedArtifact.....	704
containingBehavior .....	515	deployedElement.....	706
containingProfile.....	286, 290	deployment.....	705, 706
containingStateMachine.....	367, 370, 375, 378	Deployment.....	704
content.....	284	deploymentForArtifact.....	709
context.....	32, 305, 306, 514	deploymentLocation .....	705
Continuation.....	647	DeploymentSpecification.....	705
contract.....	159, 183, 238	deploymentTarget .....	709

DeploymentTarget .....	706	EnumerationLiteral .....	182
destroyAt .....	536	event .....	76, 87, 312
destroyLinkAction .....	574	Event .....	308
DestroyLinkAction .....	528	events .....	671
destroyObjectAction .....	593	exception .....	542
DestroyObjectAction .....	529	exceptionHandler .....	454, 455
DestructionOccurrenceSpecification .....	648	ExceptionHandler .....	442
Device .....	707	exceptionInput .....	442
diagramIdentifier .....	759	exceptionType .....	442
diamond .....	751	excludeCollisions .....	42
directedRelationship .....	60, 61	executableNode .....	557
DirectedRelationship .....	33	ExecutableNode .....	444
direction .....	147	execution .....	649
directlyRealizedInterfaces .....	136	ExecutionEnvironment .....	707
directlyUsedInterfaces .....	136	executionLocation .....	705
doActivity .....	369	executionOccurrenceSpecification .....	672
duration .....	90, 92	ExecutionOccurrenceSpecification .....	648
Duration .....	74	executionSpecification .....	672, 678
durationConstraint .....	93	ExecutionSpecification .....	649
DurationConstraint .....	74	exit .....	358, 369
durationInterval .....	90, 91	exitPoint .....	366
DurationInterval .....	75	ExpansionKind .....	529
durationObservation .....	89	ExpansionNode .....	530
DurationObservation .....	76	ExpansionRegion .....	531
edge .....	431, 438, 561	expr .....	74, 86
effect .....	147, 374	expression .....	92
Element .....	34	Expression .....	76
elementImport .....	41	extend .....	690, 692
ElementImport .....	35	Extend .....	688
elementInCompartment .....	753	extendedCase .....	689
encapsulatedClassifier .....	248	extendedRegion .....	366
EncapsulatedClassifier .....	241	extendedSignature .....	157
enclosingFragment .....	665	extendedStateMachine .....	372
enclosingInteraction .....	655	extension .....	231, 232, 292, 689, 693
enclosingOperand .....	655	Extension .....	282
end .....	237, 238, 535, 546	extensionEnd .....	294
endData .....	526, 528, 533	ExtensionEnd .....	283
endMessage .....	676, 677	extensionLocation .....	689
endType .....	229	extensionPoint .....	690
entry .....	358, 369	ExtensionPoint .....	689
entryPoint .....	365	external .....	377
enumeration .....	182	feature .....	134
Enumeration .....	181	Feature .....	139
enumerationLiteral .....	186	featuringClassifier .....	139

FIFO .....	451	implementingClassifier .....	184
fileName .....	703	import .....	55
FinalNode .....	444	importedElement .....	35
FinalState .....	358	importedMember .....	41, 42
finish .....	649	importedPackage .....	43
first .....	563	importingNamespace .....	36, 43
firstEvent .....	74, 76, 85, 87	importMembers .....	42
FlowFinalNode .....	445	in 149	
fontName .....	766	inActivity .....	434
fontSize .....	767	include .....	690, 692
fork .....	365	Include .....	690
ForkNode .....	445	includesMultiplicity .....	37
formal .....	48	includingCase .....	690
formalGate .....	653	incoming .....	378, 436
format .....	284	informationFlow .....	718, 719, 720
found .....	665	InformationFlow .....	716
fragment .....	653, 655	InformationItem .....	717
fromAction .....	515	informationSource .....	716
FunctionBehavior .....	309	informationTarget .....	716
Gate .....	649	inGroup .....	433, 435
general .....	134, 135, 140	inherit .....	135, 181, 232
generalization .....	134, 141, 163	inheritableMembers .....	136
Generalization .....	140	inheritedMember .....	134, 136
generalizationSet .....	140	inheritedParameter .....	157
GeneralizationSet .....	140	inheritedStateBorder .....	764
generalMachine .....	360	inheritingClassifier .....	163
generalOrdering .....	655	inInterruptibleRegion .....	435
GeneralOrdering .....	652	initial .....	364
generatedInstanceName .....	759	InitialNode .....	445
getName .....	36, 650	inout .....	149
getNamesOfMember .....	42	inPartition .....	433, 435
gray .....	758	input .....	514
group .....	431	inputElement .....	531
guard .....	375, 432, 655	inputParameters .....	132, 307, 519, 520, 521, 559
guarded .....	133	InputPin .....	531
handler .....	444	inputValue .....	533, 540
handlerBody .....	443	insertAt .....	516, 517, 534
hasAllDataTypeAttributes .....	309	instance .....	143
hasVisibilityOf .....	135	instanceSpecification .....	160
headedDiagram .....	770	InstanceSpecification .....	142
heading .....	756	instanceValue .....	164
icon .....	289	InstanceValue .....	143
ignore .....	657	inState .....	450
Image .....	284	inStructuredNode .....	433, 435



integerValue .....	79, 88	isConsistentWith .....	145, 153, 156, 157, 367, 370, 373, 375, 433, 436
interaction .....	659, 661, 669, 672	isContainedInRegion .....	379
Interaction .....	652	isContainedInState .....	378
interactionConstraint .....	674, 675	isControl .....	541
InteractionConstraint .....	653	isControlType .....	450
interactionFragment .....	673	isCovering .....	141
InteractionFragment .....	654	isDerived .....	151, 229
interactionOperand .....	673	isDerivedUnion .....	151
InteractionOperand .....	655	isDestroyDuplicates .....	536
interactionOperator .....	645	isDestroyLinks .....	529
InteractionOperatorKind .....	656	isDestroyOwnedObjects .....	529
interactionUse .....	669, 676, 677	isDeterminate .....	525
InteractionUse .....	657	isDimension .....	437
interface .....	144, 152, 186, 188, 189	isDirect .....	544
Interface .....	182	isDisjoint .....	141
interfaceRealization .....	180, 186	isDistinguishableFrom .....	40, 132, 662
InterfaceRealization .....	183	isDoubleSided .....	753
internal .....	377	isException .....	147
InterruptibleActivityRegion .....	446	isExternal .....	437
interruptingEdge .....	446	isFinalSpecialization .....	134
interrupts .....	433	isFormal .....	650
interval .....	90, 91	isFrame .....	756
Interval .....	77	isIcon .....	756
intervalConstraint .....	93	isID .....	151
IntervalConstraint .....	78	isIndentForVisibility .....	753
invariant .....	668	isIndirectlyInstantiated .....	234
invocationAction .....	569, 582	isInsideCF .....	650
InvocationAction .....	532	isIntegral .....	83
is 37		isIso .....	756
isAbstract .....	132, 133, 231	isLeaf .....	155
isAccessibleBy .....	452	isLocallyReentrant .....	514
isActive .....	231	isMulticast .....	448
isActivityFrame .....	749	isMultireceive .....	448
isActual .....	650	isMultivalued .....	38
isAssociationDotShown .....	752	isNavigable .....	153
isAssured .....	525	isNonNegative .....	83
isAttribute .....	152	isNull .....	80, 88
isBehavior .....	242	isOrdered .....	37, 143, 145, 552
isCollapseStateIcon .....	764	isOrthogonal .....	369, 370
isCombineDuplicate .....	447	isOutsideCF .....	650
isCompatibleWith .....	45, 88, 152	isPositive .....	83
isComposite .....	151, 153, 368, 370	isQuery .....	144
isComputable .....	79, 80, 81, 82, 88	isReadOnly .....	151, 158, 431
isConjugated .....	242		

isReceive	664	LinkEndCreationData	533
isRedefinitionContextValid	156, 367, 370, 373	linkEndData	574, 583, 595
isReentrant	305	LinkEndData	534
isRelative	311	linkEndDestructionData	573
isRemoveDuplicates	553, 554	LinkEndDestructionData	536
isReplaceAll	516, 517, 534, 551	LiteralBoolean	78
isRequired	282	LiteralInteger	79
isSend	664	LiteralNull	79
isService	242	LiteralReal	80
isSimple	369, 371	LiteralSpecification	80
isSingleExecution	431	LiteralString	81
isStatic	139	LiteralUnlimitedNatural	81
isStream	147	local	377
isStrict	289	localPostcondition	514
isSubmachineState	369, 371	localPrecondition	514
isSubstitutable	140	localStyle	757
isSubstitutableFor	137	location	284, 705
isSynchronous	518	loop	657
isTabbed	765	loopNode	570, 573, 578, 587, 591, 594
isTemplate	50, 136	LoopNode	537
isTemplateParameter	45	loopVariable	537
isTestedFirst	537	loopVariableInput	537
isTransitionOriented	764	lost	665
isUnique	37, 144, 145	lower	37, 38, 144, 145, 283
isUnmarshall	512	lowerBound	38, 283
iterative	530	lowerValue	37
join	365	makesVisible	286
joinNode	457	manifestation	703, 711
JoinNode	447	Manifestation	707
joinSpec	447	mapping	31
junction	365	matches	651
kind	238, 362, 374, 750, 758, 759	max	75, 77, 87
labelShowingStereotypeValue	772	maxint	653
language	83, 310, 540	maySpecializeType	136
LCA	372	member	41
LCAState	373	memberEnd	229
lifeline	653, 672, 676, 677	memberNamespace	56
Lifeline	659	membersAreDistinguishable	42
lifelineClass	759	mergedPackage	287
lifelineInstance	759	MergeNode	447
LIFO	451	message	646, 653, 664, 670, 671, 677
linkAction	574, 576	Message	661
LinkAction	532	messageEnd	675
linkEndCreationData	577	MessageEnd	664

MessageEvent .....	309	Node.....	708
messageKind .....	661, 662	none.....	131
MessageKind.....	665	nonNavigabilityNotation.....	752
messageName .....	760	object.....	523, 544, 546, 547, 551, 558, 560, 564
MessageOccurrenceSpecification .....	666	objectFlow.....	459, 460
messageReceivingClass .....	760	ObjectFlow.....	448
messageReceivingInstance .....	760	objectNode .....	456, 459, 460
messageSendingClass .....	759	ObjectNode .....	450
messageSendingInstance.....	759	ObjectNodeOrderingKind.....	451
messageSort .....	661	observation.....	74, 86
MessageSort.....	666	Observation .....	82
metaclass .....	282	OccurrenceSpecification .....	667
metaclassEnd.....	282	oldClassifier .....	551
metaclassReference.....	288	oneWay .....	762
metamodelReference.....	288	onPort.....	532
method.....	132	opaqueAction .....	576, 582
min .....	75, 77, 87	OpaqueAction .....	539
minint .....	653	OpaqueBehavior .....	310
mode.....	531	opaqueExpression .....	89, 93
modeElement .....	762	OpaqueExpression .....	82
Model .....	284	openEnd .....	545
modelElement . 749, 752, 753, 757, 758, 761, 763, 764, 765		operand.....	77, 645
MultiplicityElement .....	36	operation .....	147, 167, 168, 170, 308, 521
mustBeOwned.....	35, 286	Operation.....	143
mustIsolate .....	561	OperationTemplateParameter .....	146
name .....	39	opposite.....	152, 153
namedElement.....	56	oppositeEnd.....	664
NamedElement.....	39	opt .....	656
nameExpression .....	39	ordered .....	451
namespace .....	39, 55	ordering .....	450
Namespace .....	41	otherEnd.....	760
navigabilityNotation .....	752	out .....	149
navigableOwnedEnd.....	229	outgoing .....	378, 436
neg.....	657	output .....	514
nestedArtifact .....	703	outputElement.....	531
nestedClassifier.....	182, 231	outputParameters.....	132, 307, 519, 520, 521, 558
nestedNode.....	708	OutputPin .....	540
nestedPackage .....	285, 287	outputValue.....	540
nestingClass .....	247	overview.....	758
nestingPackage.....	285	ownedActual .....	48
never.....	762	ownedAttribute .....	181, 183, 185, 231, 244, 703
newClassifier.....	551	ownedBehavior .....	180
node.....	431, 438, 446, 561, 710	ownedComment .....	34
		ownedConnector .....	244

ownedDefault.....	47	par .....	656
ownedElement.....	34, 757	parallel.....	530
ownedEnd .....	229, 282	parameter.....	49, 150, 436, 760
ownedLiteral .....	181	Parameter .....	147
ownedMember .....	41	ParameterableElement .....	44
ownedOperation.....	181, 183, 231, 703	parameterableElements .....	50
ownedParameter.....	49, 132, 144, 306	ParameterDirectionKind .....	148
ownedParameteredElement.....	47	parameteredElement .....	48, 138, 146, 237
ownedParameterSet.....	132, 306	ParameterEffectKind.....	149
ownedPort .....	241	parameterSet .....	147, 160
ownedReception.....	183, 232	ParameterSet .....	149
ownedRule .....	42	parameterSubstitution .....	46
ownedStereotype.....	286, 287	parents .....	136
ownedTemplateSignature .....	50, 134	part .....	244
ownedType.....	286, 287	PartDecomposition.....	667
ownedUseCase.....	134	partition .....	431
owner.....	34	partWithPort.....	239
ownerFormalParam.....	165	Pin .....	541
owningAssociation.....	152	port .....	250, 251, 252, 312
owningCompartment.....	768	Port.....	241
owningConstraint .....	60	postcondition.....	144, 306
owningElement .....	57, 757	postCondition.....	361
owningExpression.....	84	postContext .....	166
owningInstance .....	158	powertype.....	141
owningInstanceSpec .....	170	powertypeExtent .....	134
owningLower .....	55	precondition .....	144, 306
owningPackage .....	293	preCondition .....	361
owningParameter .....	161	preContext.....	167
owningProperty.....	161	predecessorClause.....	522
owningSignal .....	187	PrimitiveType .....	184
owningSlot .....	171	private .....	52
owningState.....	386	profile.....	290, 292
owningTemplateParameter .....	45	Profile.....	288
owningTemplateParameterSubstitution.....	56	profileApplication .....	286, 291
owningTransition .....	383	ProfileApplication.....	289
owningUpper.....	62	property .....	165, 168, 170
package .....	51, 52	Property.....	150
Package .....	285	protected.....	52
PackageableElement .....	44	protectedNode .....	443
packagedElement .....	234, 286	protocol .....	183, 242
packageImport.....	42, 55	protocolConformance .....	382
PackageImport .....	43	ProtocolConformance .....	359
packageMerge.....	286, 291	ProtocolStateMachine .....	360
PackageMerge.....	287	protocolTransition.....	384, 385

ProtocolTransition.....	361	redefinableTemplateSignature .....	162, 164
provided .....	235, 242	RedefinableTemplateSignature .....	156
Pseudostate.....	362	redefinedBehavior .....	306
PseudostateKind.....	364	redefinedClassifier .....	134
public.....	52	redefinedConnector .....	238
qualifiedName .....	39, 40	redefinedEdge .....	433
qualifier .....	152, 535, 542, 547	redefinedElement .....	155
qualifierValue .....	583, 595	redefinedInterface .....	183
QualifierValue.....	541	redefinedNode.....	436
raisedException.....	132, 144	redefinedOperation .....	145
raiseExceptionAction .....	575	redefinedPort.....	242
RaiseExceptionAction.....	542	redefinedProperty.....	152
read.....	149	redefinedState .....	369
readExtentAction .....	571, 587	redefinedTransition .....	375
ReadExtentAction .....	543	redefinitionContext ..	155, 366, 367, 369, 371, 375
readIsClassifiedObjectAction .....	571, 579, 588	reduceAction .....	572, 584, 589
ReadIsClassifiedObjectAction .....	543	ReduceAction.....	552
readLinkAction .....	588	reducer.....	552
ReadLinkAction .....	544	referred .....	361, 362
readLinkObjectEndAction .....	575, 580, 588	refersTo .....	658
ReadLinkObjectEndAction .....	546	region .....	370, 372, 382, 384
readLinkObjectEndQualifierAction .....	580, 584, 588	Region .....	366
ReadLinkObjectEndQualifierAction .....	547	regionAsInput .....	530
readSelfAction .....	589	regionAsOutput.....	530
ReadSelfAction .....	548	relatedElement .....	46
readStructuralFeatureAction .....	589	relationship.....	60
ReadStructuralFeatureAction.....	549	Relationship .....	46
readVariableAction .....	589	removeAt.....	553, 554
ReadVariableAction.....	550	removeStructuralFeatureValueAction .....	584
realization.....	235, 716	RemoveStructuralFeatureValueAction .....	553
Realization .....	45	removeVariableValueAction .....	584
realizingActivityEdge .....	716	RemoveVariableValueAction .....	554
realizingClassifier .....	236	reply .....	667
realizingConnector .....	716	replyAction .....	585, 591
realizingMessage.....	716	ReplyAction .....	554
realValue .....	80, 88	replyToCall .....	555
receiveEvent.....	661	replyValue.....	555
receivingPackage .....	288	representation .....	134, 720
reception.....	189	represented .....	717
Reception .....	184	represents .....	438, 659
reclassifyObjectAction.....	579, 580, 581	request .....	556
ReclassifyObjectAction .....	551	required .....	235, 242, 243
redefinableElement .....	168, 169		
RedefinableElement.....	155		

result..	83, 512, 519, 524, 525, 527, 528, 537, 543, 544, 545, 546, 547, 549, 550, 552, 563, 564, 566, 567	start.....	649
return .....	149	startClassifierBehaviorAction.....	580
returnInformation .....	511, 555	StartClassifierBehaviorAction .....	557
returnResult.....	145	startObjectBehaviorAction .....	581
returnValue .....	658, 760	StartObjectBehaviorAction.....	558
returnValueRecipient .....	658	state .....	358, 363, 366, 380, 381, 384, 385
role .....	239, 244	State.....	368
roleBinding .....	233	stateInvariant.....	370, 671, 674
scope .....	452	StateInvariant .....	668
second .....	563	stateMachine .....	363, 366, 382
selection .....	448, 450	StateMachine.....	372
selector .....	659	stereotype .....	291, 294
sendEvent.....	661	Stereotype .....	289
sendObjectAction.....	585, 593	stereotypedElement.....	766
SendObjectAction.....	555	stream.....	530
sendSignalAction .....	592, 594	strict.....	656
SendSignalAction .....	556	StringExpression .....	84
separator .....	40	stringValue.....	81, 85, 88
seq .....	656	structuralFeature .....	560
sequence.....	758	StructuralFeature.....	158
sequenceNode .....	575	structuralFeatureAction.....	581, 592
SequenceNode.....	557	StructuralFeatureAction .....	559
sequenceNumber.....	759	structuredActivityNode.....	592, 593
sequential .....	133	StructuredActivityNode .....	560
setting.....	647	structuredClassifier .....	247, 248, 249, 252
setupPart.....	537	StructuredClassifier.....	243
shallowHistory .....	365	structuredNode.....	431
shared .....	131	structuredNodeInput .....	561
sharedStyle .....	757	structuredNodeOutput.....	561
signal .....	184, 310, 518, 556	styledElement.....	769
Signal .....	185	subExpression .....	84
signalEvent.....	315	subgroup.....	434
SignalEvent.....	310	subject .....	691
signature.....	47, 48, 661	submachine .....	370
slot.....	142, 162	submachineState .....	372
Slot .....	157	subpartition .....	438
source .....	34, 375, 433, 757	subsettingProperty.....	152
sourceEdge.....	770	subsettingContext.....	153
sourceNodes .....	538, 562	substitutingClassifier.....	159
specific .....	140	substitution.....	135, 161
specification .....	32, 75, 78, 85, 142, 306	Substitution .....	158
specificMachine .....	360	subvertex .....	366
		successorClause .....	522
		superClass .....	232

superGroup.....	434	Trigger.....	311
superPartition .....	438	type.....	51, 145, 146, 233, 238, 283
supplier.....	33	Type .....	50
supplierDependency.....	61	typedElement .....	61
symbol.....	77	TypedElement .....	51
synchCall.....	666	UMLActivityDiagram.....	749
table.....	759	UMLAssociationEndLabel .....	749
target .....	34, 375, 433, 521, 529, 556, 757	UMLAssociationOrConnectorOrLinkShape ...	750
targetEdge .....	770	UMLAssociationOrConnectorOrLinkShapeKind	
targetNodes .....	562	.....	751
template.....	49	UMLBehaviorDiagram .....	751
TemplateableElement .....	50	UMLClassDiagram .....	752
templateBinding.....	48, 50, 60	UMLClassifierShape.....	753
TemplateBinding.....	46	UMLClassOrCompositeStructureDiagram.....	752
templateParameter.....	45, 54, 57, 135, 145, 237	UMLCompartment.....	753
TemplateParameter .....	47	UMLCompartmentableShape .....	754
templateParameterSubstitution .....	53, 54	UMLComponentDiagram .....	754
TemplateParameterSubstitution .....	48	UMLCompositeStructureDiagram.....	754
templateSignature .....	59	UMLDeploymentDiagram .....	755
TemplateSignature .....	49	UMLDiagram.....	755
terminate .....	366	umlDiagramElement .....	768, 769, 770, 771
test.....	522, 538	UMLDiagramElement .....	756
testIdentityAction.....	575, 590, 591	UMLEdge .....	757
TestIdentityAction .....	563	UMLInheritedStateBorderKind .....	757
text.....	761	UMLInteractionDiagram .....	758
timeConstraint.....	94	UMLInteractionDiagramKind .....	758
TimeConstraint .....	85	UMLInteractionTableLabel .....	759
timeEvent .....	315	UMLInteractionTableLabelKind .....	759
TimeEvent.....	311	UMLKeywordLabel.....	760
timeExpression.....	90, 92	UMLLabel.....	760
TimeExpression .....	86	UMLMultiplicityLabel .....	761
timeInterval .....	91, 92	UMLNameLabel .....	761
TimeInterval.....	86	UMLNavigabilityNotationKind.....	762
timeObservation .....	89	UMLObjectDiagram .....	762
TimeObservation.....	87	UMLPackageDiagram .....	762
timing .....	759	UMLProfileDiagram.....	763
toAfter .....	667	UMLRedefinesLabel.....	763
toBefore.....	667	UMLShape.....	763
transformation .....	449	UMLStateMachineDiagram.....	764
transition .....	366, 380, 383, 384, 385, 387	UMLStateShape .....	765
Transition .....	374	UMLStereotypePropertyValueLabel .....	765
TransitionKind .....	377	UMLStructureDiagram .....	766
triangle .....	751	UMLStyle .....	766
trigger.....	313, 314, 375, 512	UMLTypedElementLabel .....	767

UMLUseCaseDiagram.....	767	ValueSpecification.....	87
unknown.....	666	valueSpecificationAction.....	590, 596
unlimitedValue.....	82, 89	ValueSpecificationAction .....	565
unmarshallAction.....	581, 590, 595	variable.....	431, 561, 566
UnmarshallAction.....	564	Variable.....	452
unmarshallType.....	564	variableAction.....	597
unordered .....	451	VariableAction.....	566
update .....	149	Vertex.....	377
upper .....	37, 38, 144, 146	viewpoint.....	285
upperBound.....	38, 450	visibility .....	35, 39, 43, 44
upperValue .....	37	VisibilityKind .....	52
URI.....	285	visibleMembers.....	287
Usage.....	52	weakOrder.....	760
useCase .....	135, 689	weight.....	433
UseCase.....	690	when.....	311
utilizedElement .....	707	WriteLinkAction .....	567
value ....	78, 79, 80, 81, 82, 84, 158, 535, 542, 565, 566, 568, 569	writeStructuralFeatureAction.....	590, 596
valuePin.....	595	WriteStructuralFeatureAction.....	567
ValuePin.....	565	writeVariableAction.....	596
		WriteVariableAction.....	568



