Date: February 2009

OMG Unified Modeling Language™ (OMG UML), Superstructure

Version 2.2
with change bars

OMG Document Number: formal/2009-02-03
Standard document URL: http://www.omg.org/spec/UML/2.2/Superstructure
Associated Files:

http://www.omg.org/spec/UML/20080501/Superstructure.xmi
http://www.omg.org/spec/UML/20080501/uml-L0-model.xmi
http://www.omg.org/spec/UML/20080501/uml-LM-model.xmi
http://www.omg.org/spec/UML/20080501/uml-L1-model.xmi
http://www.omg.org/spec/UML/20080501/uml-L2-model.xmi
http://www.omg.org/spec/UML/20080501/uml-L3-model.xmi

Normative machine-readable files: ptc/08-05-07, ptc/08-05-08, ptc/08-05-09, ptc/08-05-10, ptc/08-05-11, ptc/08-05-12.

Version 2.2 is a minor revision to the UML 2.1.2 specification. It supersedes formal/2007-11-01.

OMG's Issue Reporting Procedure

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

Table of Contents

1 Scope

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

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

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

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

2 Conformance

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

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

2.1 Language Units

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

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

2.2 Compliance Levels

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

For ease of model interchange, there are just four compliance levels defined for the whole of UML:

- *Level 0 (L0)*. This compliance level is formally defined in the UML Infrastructure. It contains a single language unit that provides for modeling the kinds of class-based structures encountered in most popular object-oriented programming languages. As such, it provides an entry-level modeling capability. More importantly, it represents a low-cost common denominator that can serve as a basis for interoperability between different categories of modeling tools.
- *Level 1 (L1)*. This level adds new language units and extends the capabilities provided by Level 0. Specifically, it adds language units for use cases, interactions, structures, actions, and activities.
- *Level 2 (L2)*. This level extends the language units already provided in Level 1and adds language units for deployment, state machine modeling, and profiles.
- *Level 3 (L3)*. This level represents the complete UML. It extends the language units provided by Level 2 and adds new language units for modeling information flows, templates, and model packaging.

The contents of language units are defined by corresponding top-tier packages of the UML metamodel, while the contents of their various increments are defined by second-tier packages within language unit packages. Therefore, the contents of a compliance level are defined by the set of metamodel packages that belong to that level.

As noted, compliance levels build on supporting compliance levels. The principal mechanism used in this specification for achieving this is *package merge* (see "PackageMerge (from Kernel)" on page 112). Package merge allows modeling concepts defined at one level to be extended with new features. Most importantly, this is achieved *in the context of the same namespace*, which enables interchange of models at different levels of compliance as described in "Meaning and Types of Compliance" on page 6.

For this reason, all compliance levels are ultimately merged into a single core "UML" model package that defines the common namespace shared by all the compliance levels. Level 0 is defined by the top-level metamodel shown in Figure 2.1. In this model, "L0" is originally an empty package that simply merges in the contents of the Basic package from the UML Infrastructure. This package is then merged into the UML model. Package L0 contains elementary concepts such as Class, Package, DataType, Operation, etc. merged in from Basic and Primitive Types (see the *Unified Modeling Language: Infrastructure* specification for the complete list of contents of these two packages).

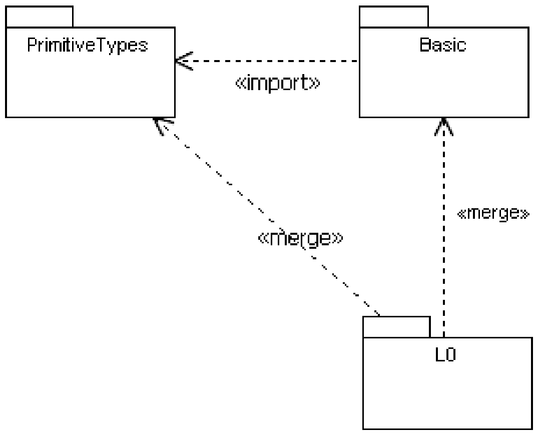


Figure 2.1 - Level 0 package diagram

At the next level (Level 1) the packages merged into Level 0 and their contents are extended with additional packages as shown in Figure 2.2 on page 4. Note that each of the four packages shown in the figure merges in additional packages that are not shown in the diagram. They are defined in the corresponding package diagrams in this specification. Consequently, the set of language units that results from this model is more than is indicated by the top-level model in the diagram. The specific packages included at this level are listed in Table 2.3 on page 8.

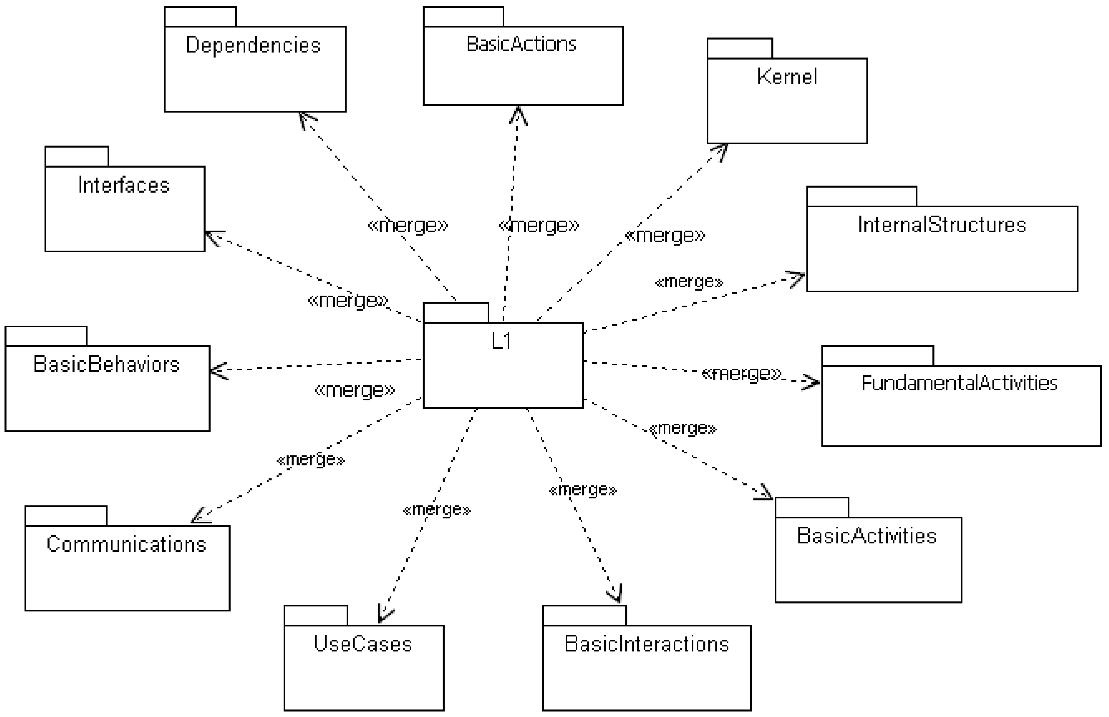


Figure 2.2 - Level 1 top-level package merges

Level 2 adds further language units and extensions to those provided by the Level 1. The actual language units and packages included at this level of compliance are listed in Table 2.4 on page 9.

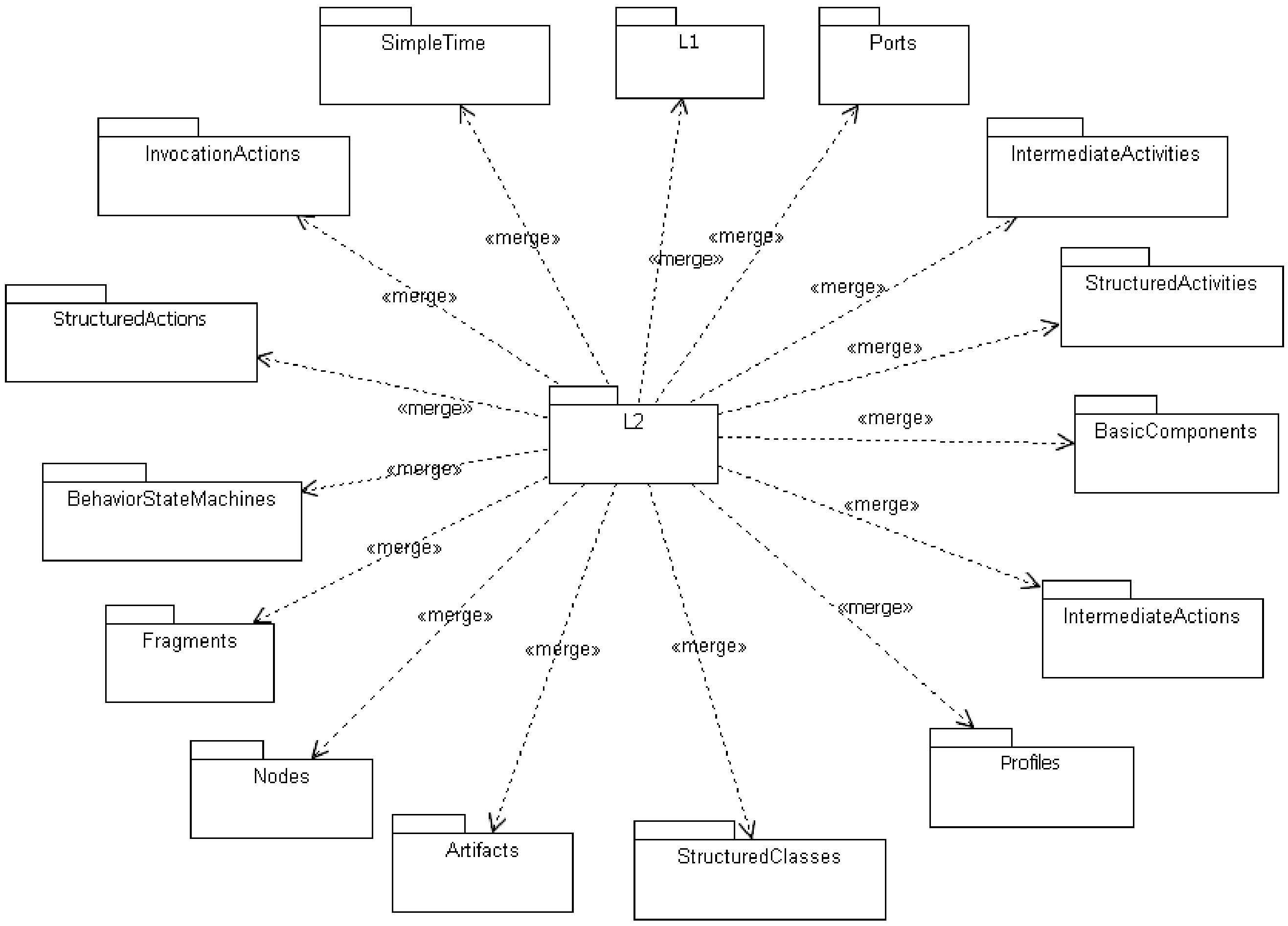


Figure 2.3 - Level 2 top-level package merges

Finally, Level3, incorporating the full UML definition, is shown in Figure 2.4 on page 6. Its contents are described in Table 2.5 on page 9.

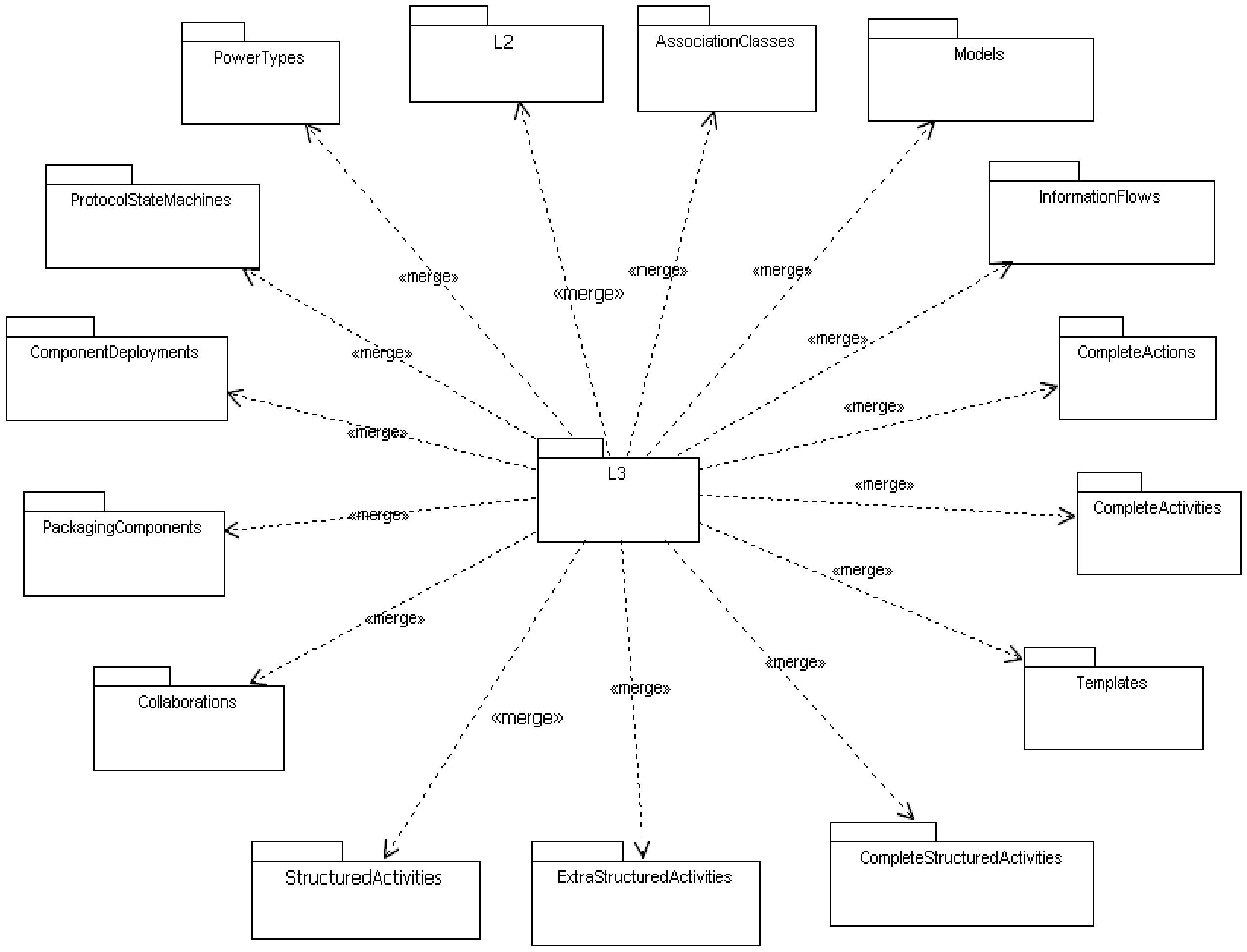


Figure 2.4 - Level 3 top-level package merges

2.3 Meaning and Types of Compliance

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

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

There are two distinct types of compliance. They are:

1. *Abstract syntax compliance*. For a given compliance level, this entails:
   - compliance with the metaclasses, their structural relationships, and any constraints defined as part of the merged UML metamodel for that compliance level and,

- the ability to output models and to read in models based on the XMI schema corresponding to that compliance level.

2. *Concrete syntax compliance*. For a given compliance level, this entails:
   - Compliance to the notation defined in the “Notation” sub clauses in this specification for those metamodel elements that are defined as part of the merged metamodel for that compliance level and, by implication, the diagram types in which those elements may appear. And, optionally:
   - the ability to output diagrams and to read in diagrams based on the XMI schema defined by the Diagram Interchange specification for notation at that level. This option requires abstract syntax and concrete syntax compliance.

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

Compliance for a given level can be expressed as:

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

Table 2.1 Example compliance statement

Compliance Summary			
Compliance level	**Abstract Syntax**	**Concrete Syntax**	**Diagram Interchange Option**
Level 0	YES	YES	YES
Level 1	YES	YES	NO
Level 2	YES	NO	NO

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

A similar situation exists with presentation options, since different implementors may make different choices on which ones to support. Finally, it is recognized that some implementors and profile designers may want to support only a subset of features from levels that are above their formal compliance level. (Note, however, that they can only claim compliance to the level that they fully support, even if they implement significant parts of the capabilities of higher levels.) Given this potential variability, it is useful to be able to specify clearly and efficiently, which capabilities are supported by a given implementation. To this end, in addition to a formal statement of compliance, implementors and profile designers may also provide informal *feature support statements*. These statements identify support for additional features in terms of language units and/or individual metamodel packages, as well as for less precisely defined dimensions such as presentation options and semantic variation points.

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

Table 2.2 Example feature support statement

Feature Support Statement					
Language Unit	**Packages**	**Abstract Syntax**	**Concrete Syntax**	**Semantics**	**Presentation Options**
Deployments	Deployments::Artifacts (L2) Deployments::Nodes (L2)	**YES**	**YES**	**Note (4)**	**Note (5)**
State Machines	StateMachines::BehaviorStateMachines (L2) StateMachines::ProtocolStateMachines (L3)	**Note (1)**	**YES**	**Note (2)**	**Note (3)**

Note (1):	States and state machines are limited to a single region Shallow history pseudostates not supported
Note (2):	FIFO queueing in event pool
Note (3):	Inherited elements indicated using grey-toned lines, etc.

2.4 Compliance Level Contents

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

Table 2.3 Metamodel packages added in Level 1

Language Unit	Metamodel Packages
Actions	Actions::BasicActions
Activities	Activities::FundamentalActivities
	Activities::BasicActivities
Classes	Classes::Kernel
	Classes::Dependencies
	Classes::Interfaces
General Behavior	CommonBehaviors::BasicBehaviors
	CommonBehaviors::Communications
Structures	CompositeStructure::InternalStructures
Interactions	Interactions::BasicInteractions
UseCases	UseCases

Table 2.4 Metamodel packages added in Level 2

Language Unit	Metamodel Packages
Actions	Actions::StructuredActions
	Actions::IntermediateActions
Activities	Activities::IntermediateActivities
	Activities::StructuredActivities
Components	Components::BasicComponents
Deployments	Deployments::Artifacts
	Deployments::Nodes
General Behavior	CommonBehaviors::SimpleTime
Interactions	Interactions::Fragments
Profiles	AuxilliaryConstructs::Profiles
Structures	CompositeStructures::InvocationActions
	CompositeStructures::Ports
	CompositeStructures::StructuredClasses
State Machines	StateMachines::BehaviorStateMachines

Table 2.5 Metamodel packages added in Level 3

Language Unit	Metamodel Packages
Action	Actions::CompleteActions
Activities	Activities::CompleteActivities
	Activities::CompleteStructuredActivities
	Activities::ExtraStructuredActivities
Classes	Classes::AssociationClasses
	Classes::PowerTypes
Components	Components::PackagingComponents
Deployments	Deployments::ComponentDeployments
Information Flows	AuxilliaryConstructs::InformationFlows
Models	AuxilliaryConstructs::Models
State Machines	StateMachines::ProtocolStateMachines
Structures	CompositeStructures::Collaborations
	CompositeStructures::StructuredActivities
Templates	AuxilliaryConstructs::Templates

3 Normative References

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

- UML 2.0 Superstructure RFP
- UML 2. Infrastructure Specification
- MOF 2.0 Specification

4 Terms and Definitions

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

5 Symbols

There are no symbols defined in this specification.

6 Additional Information

6.1 Architectural Alignment and MDA Support

Clause 1, "Language Architecture" of the *Unified Modeling Language: Infrastructure* explains how the *Unified Modeling Language: Infrastructure* is architecturally aligned with the *Unified Modeling Language: Superstructure* that complements it. It also explains how the InfrastructureLibrary defined in the *Unified Modeling Language: Infrastructure* can be strictly reused by MOF 2.0 specifications.

It is the intent that the unified MOF 2.0 Core specification must be architecturally aligned with the *Unified Modeling Language: Infrastructure* part of this specification. Similarly, the unified UML 2.0 Diagram Interchange specification must be architecturally aligned with the *Unified Modeling Language: Superstructure* part of this specification.

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

6.2 On the Run-Time Semantics of UML

The purpose of this sub clause of the document is to provide a very high-level view of the *run-time semantics* of UML and to point out where the various elements of that view are covered in the specification. The term "run-time" is used to refer to the execution environment. Run-time semantics, therefore, are specified as a mapping of modeling concepts into corresponding program execution phenomena. There are, of course, other semantics relevant to UML specifications, such as the *repository semantics*, that is, how a UML model behaves in a model repository. However, those semantics are really part of the definition of the MOF. Still, it is worth remarking that not every concept in UML models a run-time phenomenon (e.g., the “package” concept).

6.2.1 The Basic Premises

There are two fundamental premises regarding the nature of UML semantics. The first is the assumption that all behavior in a modeled system is ultimately caused by actions executed by so-called “active” objects (see “Class (from Communications)” on page 437). This includes behaviors, which are objects in UML 2, which can be active and coordinate other behaviors. The second is that UML behavioral semantics only deal with *event-driven*, or discrete, behaviors. However, UML does not dictate the amount of time between events, which can be as small as needed by the application, for example, when simulating continuous behaviors.

6.2.2 The Semantics Architecture

Figure 6.1 identifies the key semantic areas covered by the current standard and how they relate to each other. The items in the upper layers depend on the items in the lower layers but not the other way around. (Note that the structure of metamodel package dependencies is somewhat similar to the dependency structure indicated here. However, they are not the same and should be distinguished. This is because package dependencies specify repository dependencies not necessarily run-time dependencies.)

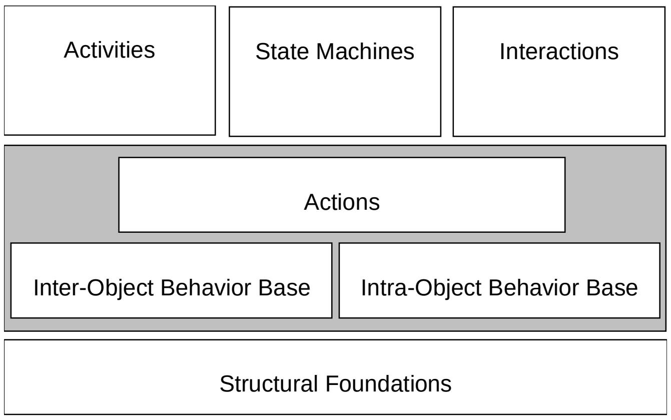


Figure 6.1 - A schematic of the UML semantic areas and their dependencies

At the highest level of abstraction, it is possible to distinguish three distinct composite layers of semantic definitions. The foundational layer is structural. This reflects the premise that there is no disembodied behavior in UML – all behavior is the consequence of the actions of structural entities. The next layer is behavioral and provides the foundation for the semantic description of all the higher-level behavioral formalisms (the term “behavioral formalism” refers to a formalized framework for describing behavior, such as state machines, Petri nets, data flow graphs, etc.). This layer, represented by the shaded box in Figure 6.1, is the behavioral semantic base and consists of three separate sub areas arranged into two sub layers. The bottom sub layer consists of the *inter-object behavior base*, which deals with how structural entities communicate with each other, and the *intra-object behavior base*, which addresses the behavior occurring within structural entities. The *actions* sub layer is placed on top of these two. It defines the semantics of individual actions. Actions are the fundamental units of behavior in UML and are used to define fine-grained behaviors. Their resolution and expressive power are comparable to the executable instructions in traditional programming languages. Actions in this sub layer are available to any of the higher-level formalisms to be used for describing detailed behaviors. The topmost layer in the semantics hierarchy defines the semantics of the higher-level behavioral formalisms of UML: *activities*, *state machines*, and *interactions*. Other behavioral formalisms may be added to this layer in the future.

6.2.3 The Basic Causality Model

The "causality model" is a specification of how things happen at run time and is described in detail in the Common Behaviors clause on page 421. It is briefly summarized here for convenience, using the example depicted in the communication diagram in Figure 6.2. The example shows two independent and possibly concurrent threads of causally chained interactions. The first, identified by the thread prefix 'A' consists of a sequence of events that commence with activeObject-1 sending signal s1 to activeObject-2. In turn, activeObject-2 responds by invoking operation op1( ) on passiveObject-1 after which it sends signal s2 to activeObject-3. The second thread, distinguished by the thread prefix 'B,' starts with activeObject-4 invoking operation op2( ) on passiveObject-1. The latter responds by executing the method that realizes this operation in which it sends signal s3 to activeObject-2.

The causality model is quite straightforward: Objects respond to messages that are generated by objects executing communication actions. When these messages arrive, the receiving objects eventually respond by executing the behavior that is matched to that message. The dispatching method by which a particular behavior is associated with a given message depends on the higher-level formalism used and is not defined in the UML specification (i.e., it is a semantic variation point).

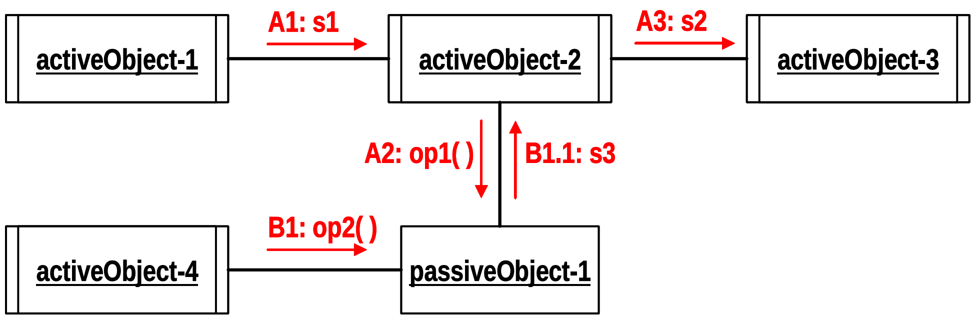


Figure 6.2 - Example illustrating the basic causality model of UML

The causality model also subsumes behaviors invoking each other and passing information to each other through arguments to parameters of the invoked behavior, as enabled by CallBehaviorAction (see "CallBehaviorAction (from BasicActions)" on page 243). This purely "procedural" or "process" model can be used by itself or in conjunction with the object-oriented model of the previous example.

6.2.4 Semantics Descriptions in the Specification

The general causality model is described in the introductory part of Clause 13 (CommonBehaviors) and also, in part, in the introduction to Clause 14 (Interactions) and the sub clause on Interaction (14.3.13) and Message (14.3.20).

The structural foundations are mostly covered in two clauses. The elementary level is mostly covered in Clause 7, where the root concepts of UML are specified. In particular, the sub clauses on InstanceSpecifications (7.3.22), Classes (7.3.7), Associations (7.3.3), and Features (7.3.19). The composites level is described primarily in Clause 9 (Composite Structures), with most of the information related to semantics contained in sub clauses 9.3.12 (Property concept) and 9.3.13 (StructuredClassifier). In addition, the introduction to this clause contains a high-level view of some aspects of composite structures.

The relationship between structure and behavior and the general properties of the Behavior concept, which are at the core of the behavioral base are described in CommonBehaviors (in the introduction to Clause 13 and in sub clause 13.3.2 in particular).

Inter-object behavior is covered in three separate clauses. The basic semantics of communications actions are described in the introduction to Clause 11 (Actions) and, in more detail, in the clauses describing the specific actions. These can potentially be used by an object on itself, so can be inter- or intra-object. The read/write actions can also be used by one object to access other objects, so are potentially inter- or intra-object. These actions can be used by any of the behavior formalisms in UML, so all are potentially inter-object behaviors. However, the interactions diagram is designed specifically to highlight inter-object behavior, under its concept of message. These are defined in the Interactions clause (sub clauses 14.3.20 and 14.3.21), while the concepts of events and triggers are defined in the Communications package of CommonBehaviors (Clause 13). Occurrence specifications are defined in sub clause 14.3.25 of the Interactions clause. The other two behavior formalisms can be translated to interactions when they use inter-object actions.

All the behavior formalisms are potentially intra-object, if they are specified to be executed by and access only one object. However, state machines are designed specifically to model the state of a single object and respond to events arriving at that object. Activities can be used in a similar way, but also highlight input and output dependency between behaviors, which may reside in multiple objects. Interactions are potentially intra-object, but generally not designed for that purpose.

The various shared actions and their semantics are described in Clause 13.

Finally, the higher-level behavioral formalisms are each described in their own clauses: Activities in Clause 12, Interactions in Clause 14, and State Machines in Clause 15.

6.3 The UML Metamodel

6.3.1 Models and What They Model

A model contains three major categories of elements: Classifiers, events, and behaviors. Each major category models individuals in an incarnation of the system being modeled. A classifier describes a set of objects; an object is an individual thing with a state and relationships to other objects. An event describes a set of possible occurrences; an occurrence is something that happens that has some consequence within the system. A behavior describes a set of possible executions; an execution is the performance of an algorithm according to a set of rules. Models do not contain objects, occurrences, and executions, because those things are the subject of models, not their content. Classes, events, and behaviors model sets of objects, occurrences, and executions with similar properties. Value specifications, occurrence specifications, and execution specifications model individual objects, occurrences, and executions within a particular context. The distinction between objects and models of objects, for example, may appear subtle, but it is important. Objects (and occurrences and executions) are the domain of a model and, as such, are always complete, precise, and concrete. Models of objects (such as value specifications) can be incomplete, imprecise, and abstract according to their purpose in the model.

6.3.2 Semantic Levels and Naming

A large number of UML metaclasses can be arranged into 4 levels with metasemantic relationships among the metaclasses in the different levels that transcend different semantic categories (e.g., classifiers, events, behaviors). We have tried (with incomplete success) to provide a consistent naming pattern across the various categories to place elements into levels and emphasize metarelationships among related elements in different levels. The following 4 levels are important:

Type level – Represents generic types of entities in models, such as classes, states, activities, events, etc. These are the most common constituents of models because models are primarily about making generic specifications.

Instance level – These are the things that models represent at runtime. They don't appear in models directly (except very occasionally as detailed examples), but they are necessary to explain the semantics of what models mean. These classes do not appear at all in the UML2 metamodel or in UML models, but they underlie the meaning of models. We provide a brief runtime metamodel in the Common Behavior clause, but we do not formally define the semantics of UML using the runtime metamodel. Such a formal definition would be a major amount of work.

Value specifications – A realization of UML2, compared to UML, is that values can be specified at various levels of precision. The specification of a value is not necessarily an instance; it might be a large set of possible instances consistent with certain conditions. What appears in models is usually not instances (individual values) but specifications of values that may or may not be limited to a single value. In any case, models contain specifications of values, not values themselves, which are runtime entities.

Individual appearances of a type within a context – These are roles within a generic, reusable context. When their context is instantiated, they are also bound to contained instances, but as model elements they are reusable structural parts of their context; they are not instances themselves. A realization of UML2 was that the things called instances in UML1 were mostly roles: they map to instances in an instance of their container, but they are model elements, not instances, because they are generic and can be used many times to generate many different instances.

We have established the following naming patterns:

> Types : Instances : Values : Uses
>
> Classifier, Class : Instance, Object : InstanceSpecification : Part, Role, Attribute, XXXUse (e.g., CollaborationUse)
>
> Event : Occurrence : OccurrenceSpecification : various (e.g., Trigger)
>
> Behavior : Execution : ExecutionSpecification : various (e.g., ActivityNode, State), XXXUse (e.g., InteractionUse)

The appearances category has too wide a variety of elements to reduce to a single pattern, although the form XXXUse is suggested for simple cases where an appearance of an element is contained in a definition of the same kind of element.

In particular, the word "event" has been used inconsistently in the past to mean both type and instance. The word "event" now means the type and the word "occurrence" means the instance. When necessary, the phrases "event type" (for event) and "event occurrence" (for occurrence) may be used. Note that this is consistent with the frequent English usage "an event occurs" = the occurrence of an event of a given type; so to describe a runtime situation, one could say "event X occurs" or "an occurrence of event X" depending on which form is more convenient in a sentence. It is redundant and incorrect to say "an event occurrence occurs."

6.4 How to Read this Specification

The rest of this document contains the technical content of this specification. As background for this specification, readers are encouraged to first read the *UML: Infrastructure* specification that complements this specification. Part I, "Introduction" of *UML: Infrastructure* explains the language architecture structure and the formal approach used for its specification. Afterwards the reader may choose to either explore the InfrastructureLibrary, described in Part II, "Infrastructure Library," or the Classes::Kernel package that reuses it, described in Clause 1, "Classes." The former specifies the flexible metamodel library that is reused by the latter; the latter defines the basic constructs used to define the UML metamodel.

With that background the reader should be well prepared to explore the user level constructs defined in this *UML: Superstructure* specification. These concepts are organized into three parts: Part I - “Structure,” Part II - “Behavior,” and Part III - “Supplement.” “Part I - Structure” defines the static, structural constructs (e.g., classes, components, nodes artifacts) used in various structural diagrams, such as class diagrams, component diagrams, and deployment diagrams. “Part II - Behavior” specifies the dynamic, behavioral constructs (e.g., activities, interactions, state machines) used in various behavioral diagrams, such as activity diagrams, sequence diagrams, and state machine diagrams. “Part III - Supplement” defines auxiliary constructs (e.g., information flows, models, templates, primitive types) and the profiles used to customize UML for various domains, platforms, and methods.

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.

6.4.1 Specification format

The concepts of UML are grouped into three major parts:

- Part I: Concepts related to the modeling of structure
- Part II: Concepts related to the modeling of behavior
- Part III: Supplementary concepts

Within each part, the concepts are grouped into clauses according to modeling *capability.* A capability typically covers a specific modeling formalism. For instance, all concepts related to the state machine modeling capability are gathered in the State Machines clause and all concepts related to the activities modeling capability are in the Activities clause. The Capability clauses in each part are presented in alphabetical order.

Within each clause, there is first a brief informal description of the capability described in that clause. This is followed by a sub clause describing the *abstract syntax* for that capability. The abstract syntax is defined by a CMOF model (i.e., the UML metamodel) with each modeling concept represented by an instance of a MOF class or association. The model is decomposed into packages according to capabilities. In the specification, this model is described by a set of UML class and package diagrams showing the concepts and their relationships. The diagrams were designed to provide comprehensive information about a related set of concepts, but it should be noted that, in many cases, the representation of a concept in a given diagram displays only a subset of its features (the subset that is relevant in that context). The same concept may appear in multiple diagrams with different feature subsets. For a complete specification of the features of a concept, readers should refer to its formal concept description (explained below). When the concepts in the capability are grouped into sub packages, the diagrams are also grouped accordingly with a heading identifying the sub package preceding each group of diagrams. In addition, the name of the owning package is included in each figure caption.

The “Concept Definitions” clause follows the abstract syntax clause. This clause includes formal specifications of all concepts belonging to that capability, listed in alphabetical order. Each concept is described separately according to the format explained below.

The final sub clause in most clauses gives an overview of the diagrams, diagram elements, and notational rules and conventions that are specific to that capability.

The formal concept descriptions of individual concepts are broken down into sub clauses corresponding to different aspects. In cases where a given aspect does not apply, its sub clause may be omitted entirely from the class description. The following sub clauses and conventions are used to specify a concept:

- The *heading* gives the formal name of the concept and indicates, in parentheses, the sub package in which the concept is defined. In some cases, there may be more than one sub package name listed. This occurs when a concept is defined in multiple package merge increments – one per package. In a few instances, there is no package name, but the phrase "as specialized" appears in parentheses. This indicates a "semantic" increment, which does not involve a new increment in the metamodel and which, therefore, does not change the abstract syntax, but which adds new semantics to previous increments (e.g., additional constraints).

- In some cases, following the heading is a brief, one- or two-sentence informal description of the meaning of a concept. This is intended as a quick reference for those who want only the basic information about a concept.
- All the direct generalizations of a concept are listed, alphabetically, in the "Generalizations" sub clause. A "direct" generalization of a concept is a concept (e.g., a class) that is immediately above it in the hierarchy of its ancestors (i.e., its "parent"). Note that these items are hyperlinked in electronic versions of the document to facilitate navigation through the metamodel class hierarchy. Readers of hardcopy versions can use the page numbers listed with the names to rapidly locate the description of the superclass. This sub clause is omitted for enumerations.
- A more detailed description of the purpose, nature, and potential usage of the concept may be provided in the "Description" sub clause. This too is informal. If a concept is defined in multiple increments, then the first part of the description covers the top-level package and is followed, in turn, by successive description increments for each sub package. The individual increments are identified by a sub package heading such as

  Package PowerTypes

  This indicates that the text that follows the heading describes the increment that was added in the PowerTypes sub package. The description continues either until the end of the sub clause or until the next sub package increment heading is encountered.
- This convention for describing sub package increments is applied to all other sub clauses related to the concept.
- The "Attributes" sub clause of a concept description lists each of the attributes that are defined for that metaclass. Each attribute is specified by its formal name, its type, and multiplicity. If no multiplicity is listed, it defaults to 0..*. 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. For example:

  •body: String[1]   Specifies a string that is the comment

  specifies an attribute called "body" whose type is "String" and whose multiplicity is 1.
- If an attribute is derived, where possible, the definition will also include a specification (usually expressed as an OCL constraint) specifying how that attribute is derived. For instance:

  •/isComposite : Boolean   A state with isComposite = true is said to be a *composite state*. A composite state is a state that contains at least one region>

  ```
  isComposite = (region > 1)
  ```
- The "Associations" sub clause lists all the association ends owned by the concept. The format for these is the same as the one for attributes described above. Association ends that are specializations or redefinitions of other association ends in superclasses are flagged appropriately. For example:

  •lowerValue: ValueSpecification[0..1]   {subsets *Element::ownedElement*} The specification of the lower bound for this multiplicity.

  specifies an association end called "lowerValue" that is connected to the "ValueSpecification" class and whose multiplicity is 0..1. Furthermore, it is a specialization of the "ownedElement" association end of the class "Element."
- As with derived attributes, if an association end is derived, where possible, the definition will also include a specification (usually expressed as an OCL constraint) specifying how that association end is derived.

- The “Constraints” sub clause contains a numerical list of all the constraints that define additional well-formedness rules that apply to this concept. Each constraint consists of a textual description and may be followed by a formal constraint expressed in OCL. Note that in a few cases, it may not be possible to express the constraint in OCL, in which case the formal expression is omitted.
- “Additional Operations” contains a numerical list of operations that are applicable to the concept. These may be queries or utility operations that are used to define constraints or other operations. Where possible, operations are specified using OCL.
- The “Semantics” sub clause describes the meaning of the concept in terms of its concrete manifestation. This is a specification of the set of things that the concept models (represents) including, where appropriate, a description of the behavior of those things (i.e., the dynamic semantics of the concept).
- “Semantic Variation Points” explicitly identifies the areas where the semantics are intentionally under specified to provide leeway for domain-specific refinements of the general UML semantics (e.g., by using stereotypes and profiles).
- The “Notation” sub clause gives the basic notational forms used to represent a concept and its features in diagrams. Only concepts that can appear in diagrams will have a notation specified. This typically includes a simple example illustrating the basic notation. For textual notations a variant of the Backus-Naur Form (BNF) is often used to specify the legal formats. The conventions of this BNF are:
    - All non-terminals are in italics and enclosed between angle brackets (e.g., <*non-terminal*>).
    - All terminals (keywords, strings, etc.), are enclosed between single quotes (e.g., ‘*or*’).
    - Non-terminal production rule definitions are signified with the ‘::=’ operator.
    - Repetition of an item is signified by an asterisk placed after that item: ‘*’.
    - Alternative choices in a production are separated by the ‘|’ symbol (e.g., <*alternative-A*> | <*alternative-B*>*)*.
    - Items that are optional are enclosed in square brackets (e.g., *[<item-x>]*).
    - Where items need to be grouped they are enclosed in simple parenthesis; for example:

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

      signifies a sequence of one or more items, each of which is <*item-1*> or <*item-2*>.
- The “Presentation Options” sub clause supplements the “Notation” clause by providing alternative representations for the concept or its parts. Users have the choice to use either the forms described in this sub clause or the forms described in the “Notation” sub clause.
- “Style Guidelines” identifies notational conventions recommended by the specification. These are not normative but, if applied consistently, will facilitate communication and understanding. For example, there is a style guideline that suggests that the names of classes should be capitalized and another one that recommends that the names of abstract classes be written out in italic font. (Note that these specific recommendations only make sense in certain writing systems, which is why they cannot be normative.)
- The “Examples” sub clause, if present, includes additional illustrations of the application of the concept and its notation.
- “Changes from previous UML” identifies the main differences in the specification of the concept relative to UML versions 1.5 and earlier.

6.4.2 Diagram format

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

- An association with one end marked by a navigability arrow means that:
  - the association is navigable in the direction of that end,
  - the marked association end is owned by the classifier, and
  - the opposite (unmarked) association end is owned by the association.

Note – This convention was inherited from UML 1.x and was used in the initial versions of the specification because there was no explicit notation for indicating association end ownership. Such a notation was introduced in revision 2.1.1 (see the notation sub clause of the Association metaclass on page 39) but was not applied to the diagrams in the specification due to lack of tool support. In accord with the new notation, the ownership of an association end by the association would continue to be shown by leaving the end unmarked, but the ownership of an end by the classifier would be shown by marking that classifier-owned end with a dot.

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

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

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

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

6.5 Acknowledgements

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

- 7irene
- 88solutions
- Adaptive
- Advanced Concepts Center LLC
- Alcatel
- Artisan
- Borland
- Ceira Technologies
- Commissariat à L'Energie Atomique
- Computer Associates
- Compuware
- DaimlerChrysler
- Domain Architects
- Embarcadero Technologies
- Enea Business Software
- Ericsson
- France Telecom
- Fraunhofer FOKUS
- Fujitsu
- Gentleware
- Intellicorp
- Hewlett-Packard
- I-Logix
- International Business Machines
- IONA
- Jaczone
- Kabira Technologies
- Kennedy Carter
- Klasse Objecten
- KLOCwork
- Lockheed Martin
- MEGA International
- Mercury Computer
- Motorola
- MSC.Software

- Northeastern University
- oose Innovative Informatik GmbH
- Oracle
- Popkin Software
- Proforma
- Project Technology
- Sims Associates
- SOFTEAM
- Sun Microsystems
- Syntropy Ltd.
- Telelogic
- Thales Group
- TNI-Valiosys
- Unisys
- University of Kaiserslautern
- University of Kent
- VERIMAG
- WebGain
- X-Change Technologies

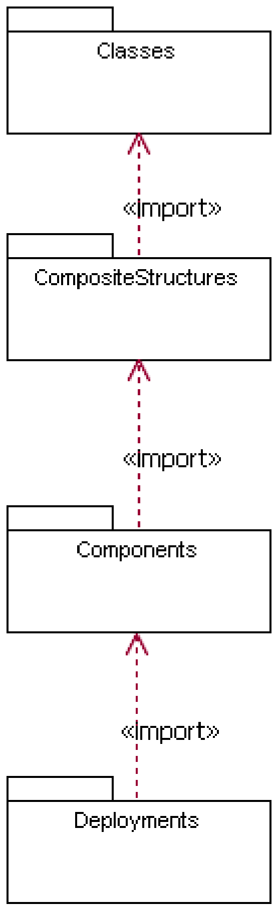
The following persons were members of the core team that designed and wrote this specification: Don Baisley, Morgan Björkander, Conrad Bock, Steve Cook, Philippe Desfray, Nathan Dykman, Anders Ek, David Frankel, Eran Gery, Øystein Haugen, Sridhar Iyengar, Cris Kobryn, Birger Møller-Pedersen, James Odell,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

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

The authors are grateful to Pavel Hruby for his drawing tool stencil for UML, which was used to create many of the UML diagrams in this document.

Part I - Structure

This part defines the static, structural constructs (e.g., classes, components, nodes artifacts) used in various structural diagrams, such as class diagrams, component diagrams, and deployment diagrams. The UML packages that support structural modeling are shown in the figure below.



Part I, Figure 1 - UML packages that support structural modeling

The function and contents of these packages are described in following clauses, which are organized by major subject areas.

7 Classes

7.1 Overview

The Classes package contains sub packages that deal with the basic modeling concepts of UML, and in particular classes and their relationships.

Reusing packages from UML 2 Infrastructure

The *Kernel* package represents the core modeling concepts of the UML, including classes, associations, and packages. This part is mostly reused from the infrastructure library, since many of these concepts are the same as those that are used in, for example, MOF. The *Kernel* package is the central part of the UML, and reuses the *Constructs* and *PrimitiveTypes* packages of the InfrastructureLibrary.

In many cases, the reused classes are extended in the *Kernel* with additional features, associations, or superclasses. In subsequent diagrams showing abstract syntax, the subclassing of elements from the infrastructure library is always elided since this information only adds to the complexity without increasing understandability. Each metaclass is completely described as part of this clause; the text from the infrastructure library is repeated here.

It should also be noted that *Kernel* is a flat structure that like *Constructs* only contains metaclasses and no sub-packages. The reason for this distinction is that parts of the infrastructure library have been designed for flexibility and reuse, while the *Kernel* in reusing the infrastructure library has to bring together the different aspects of the reused metaclasses.

The packages that are explicitly merged from the InfrastructureLibrary are the following:

- PrimitiveTypes
- Constructs

All other packages of the InfrastructureLibrary::Core are implicitly merged through the ones that are explicitly merged.

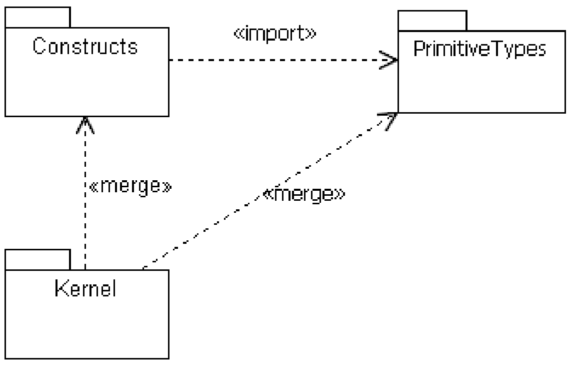


Figure 7.1 - InfrastructureLibrary packages that are merged by Kernel (all dependencies in the picture represent package merges)

7.2 Abstract Syntax

Figure 7.2 shows the package dependencies of the Kernel packages.

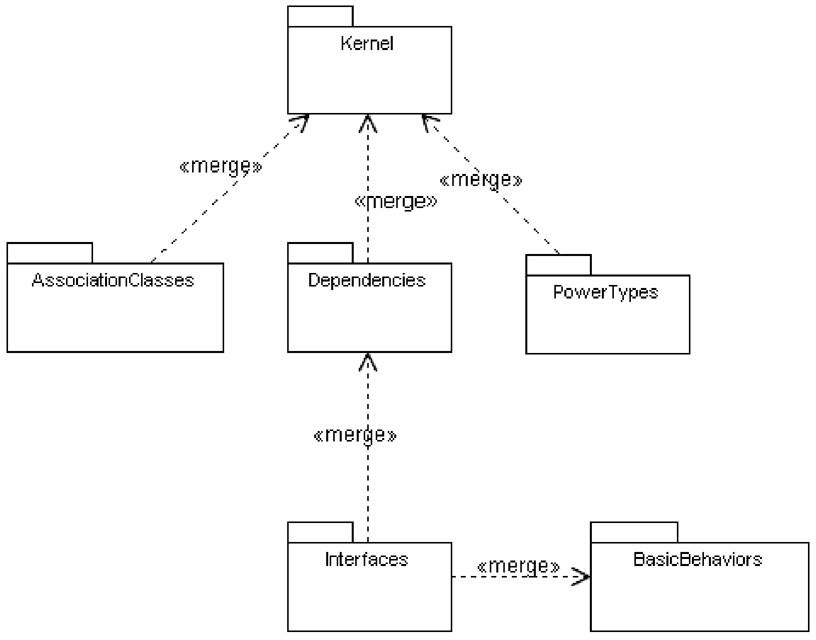


Figure 7.2 - Subpackages of the Classes package and their dependencies

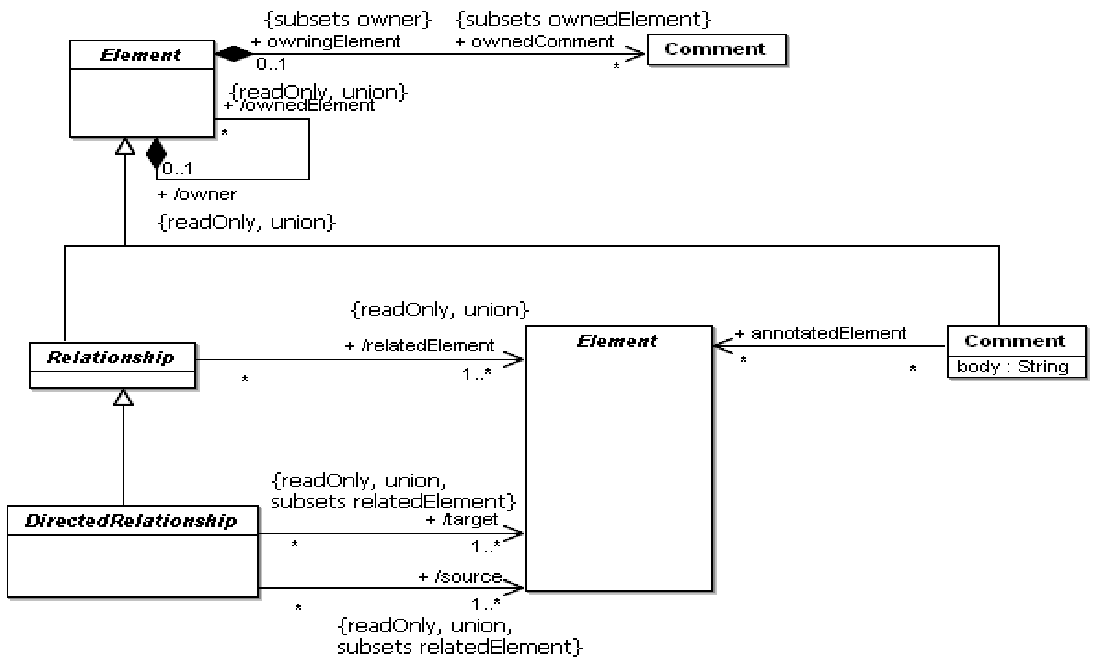


Figure 7.3 - Root diagram of the Kernel package

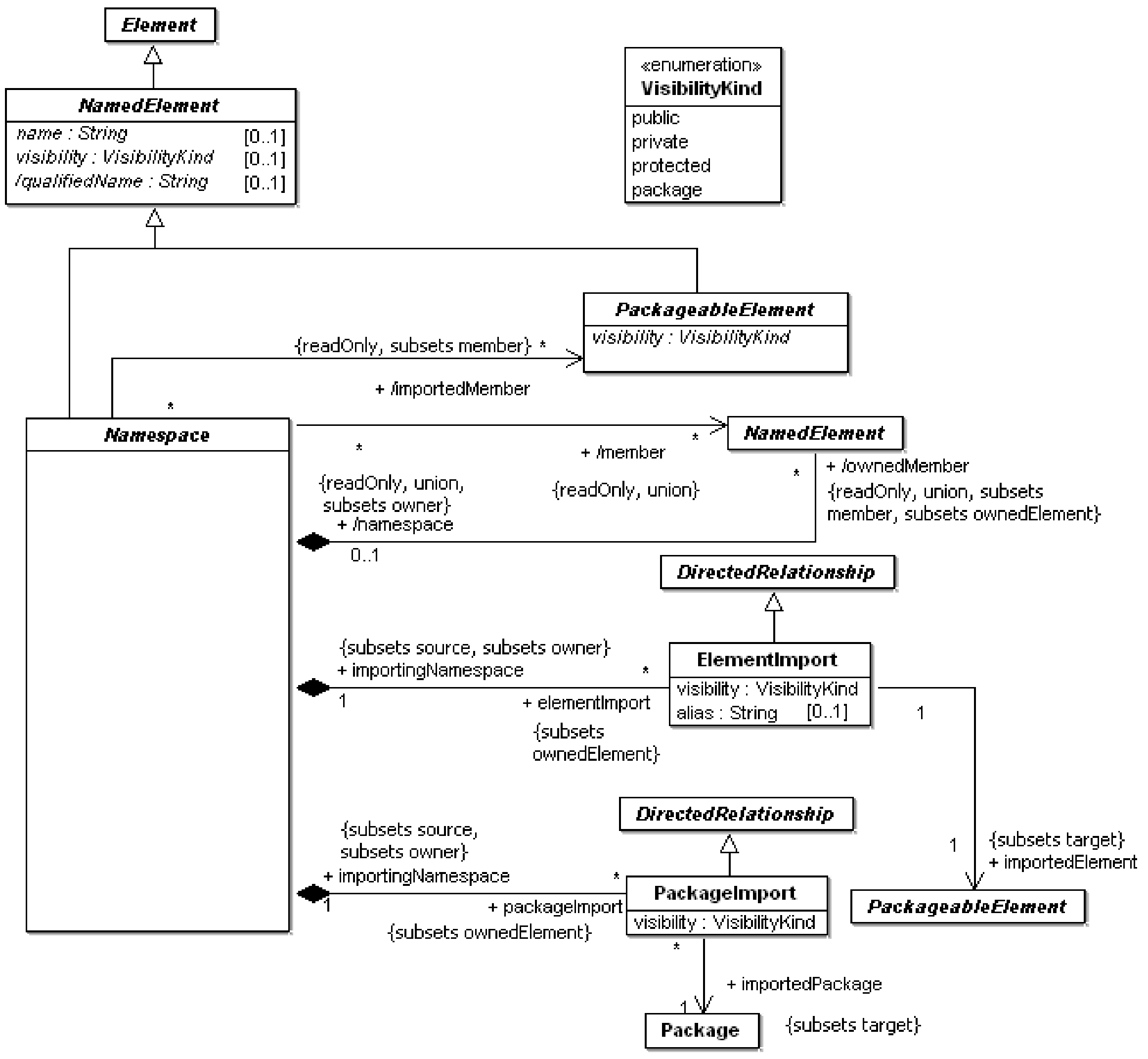


Figure 7.4 - Namespaces diagram of the Kernel package

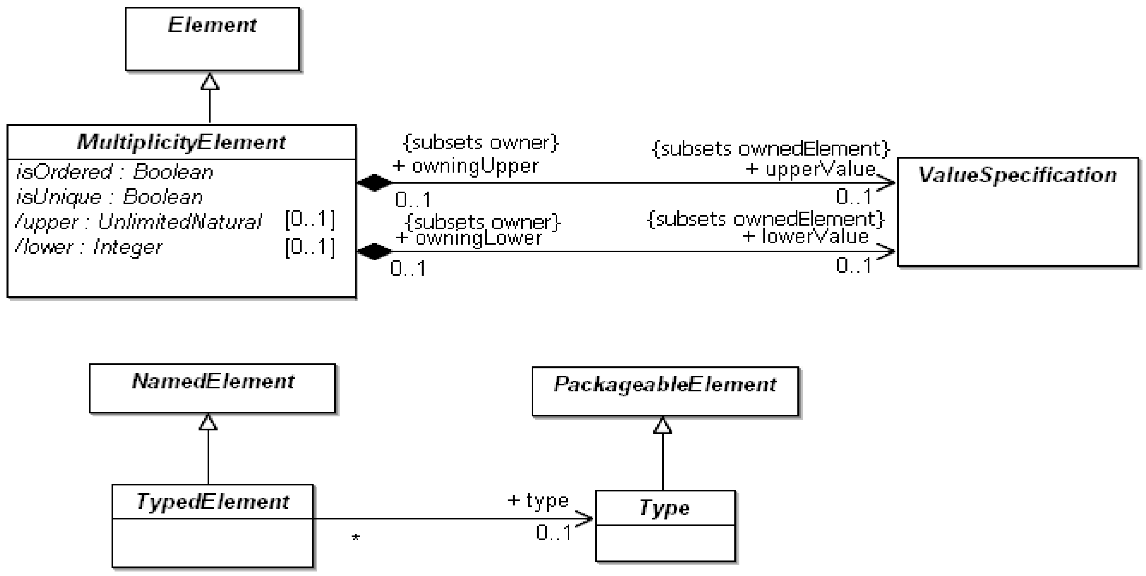


Figure 7.5 - Multiplicities diagram of the Kernel package

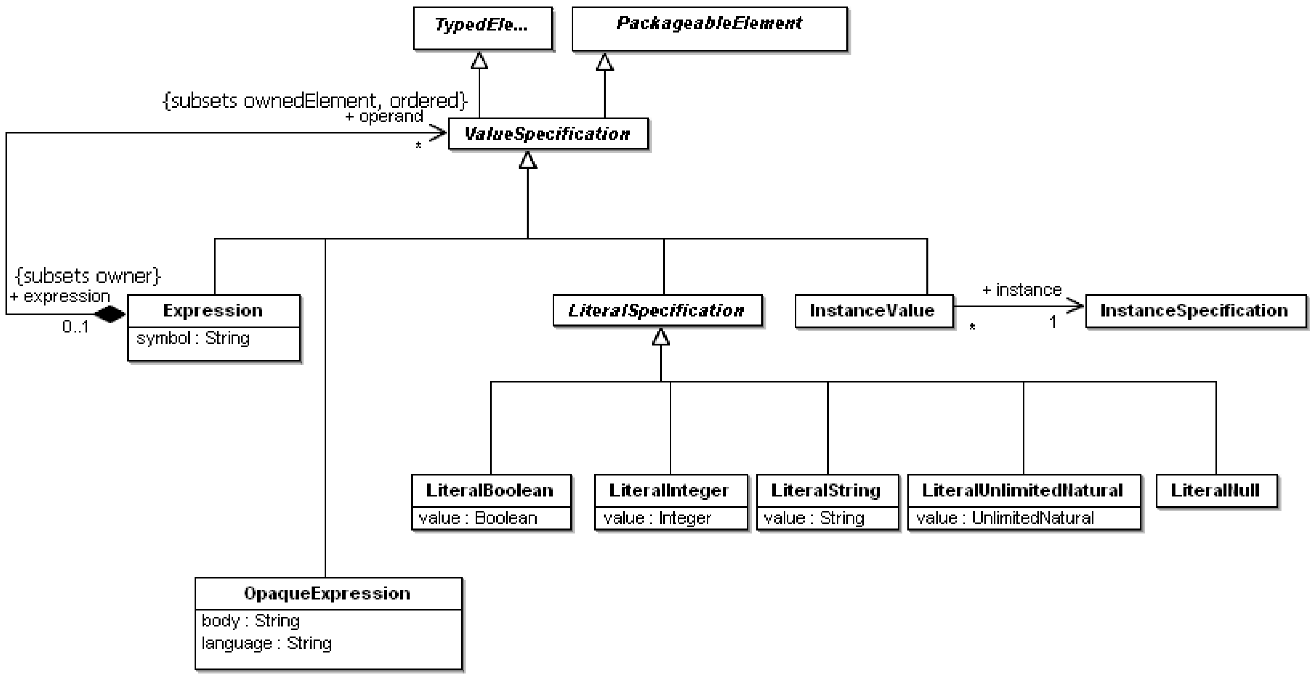


Figure 7.6 - Expressions diagram of the Kernel package

.

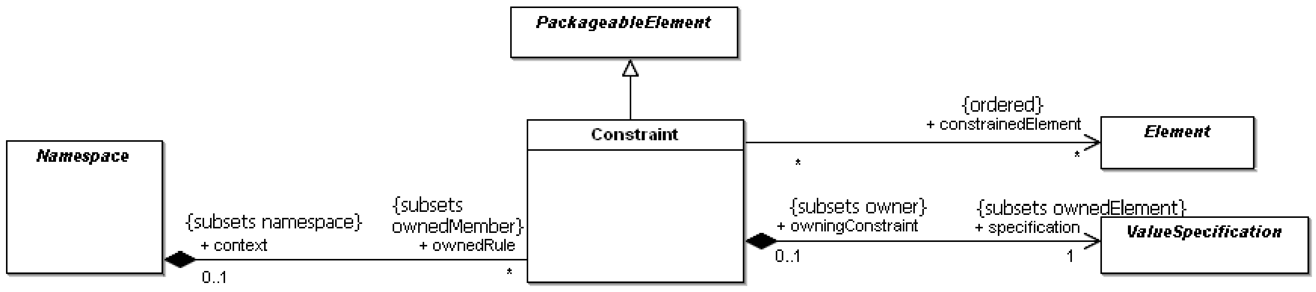


Figure 7.7 - Constraints diagram of the Kernel package

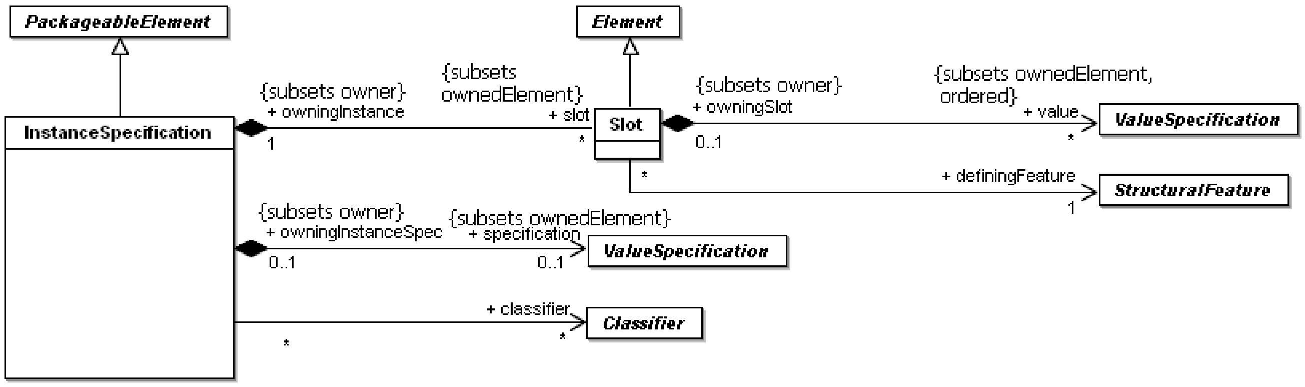


Figure 7.8 - Instances diagram of the Kernel package

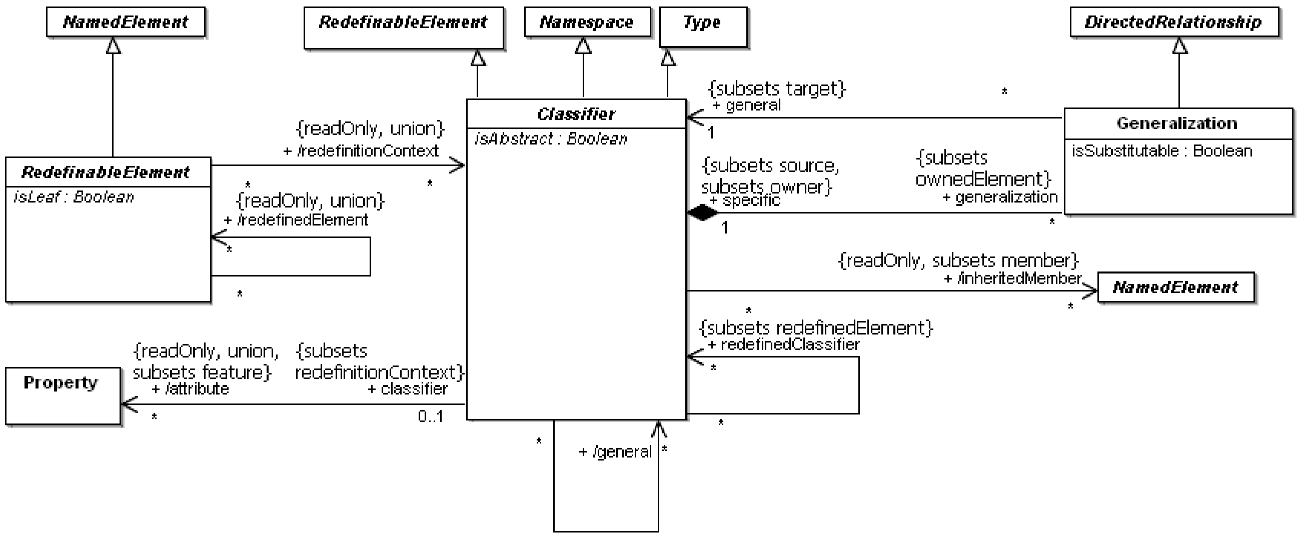


Figure 7.9 - Classifiers diagram of the Kernel package

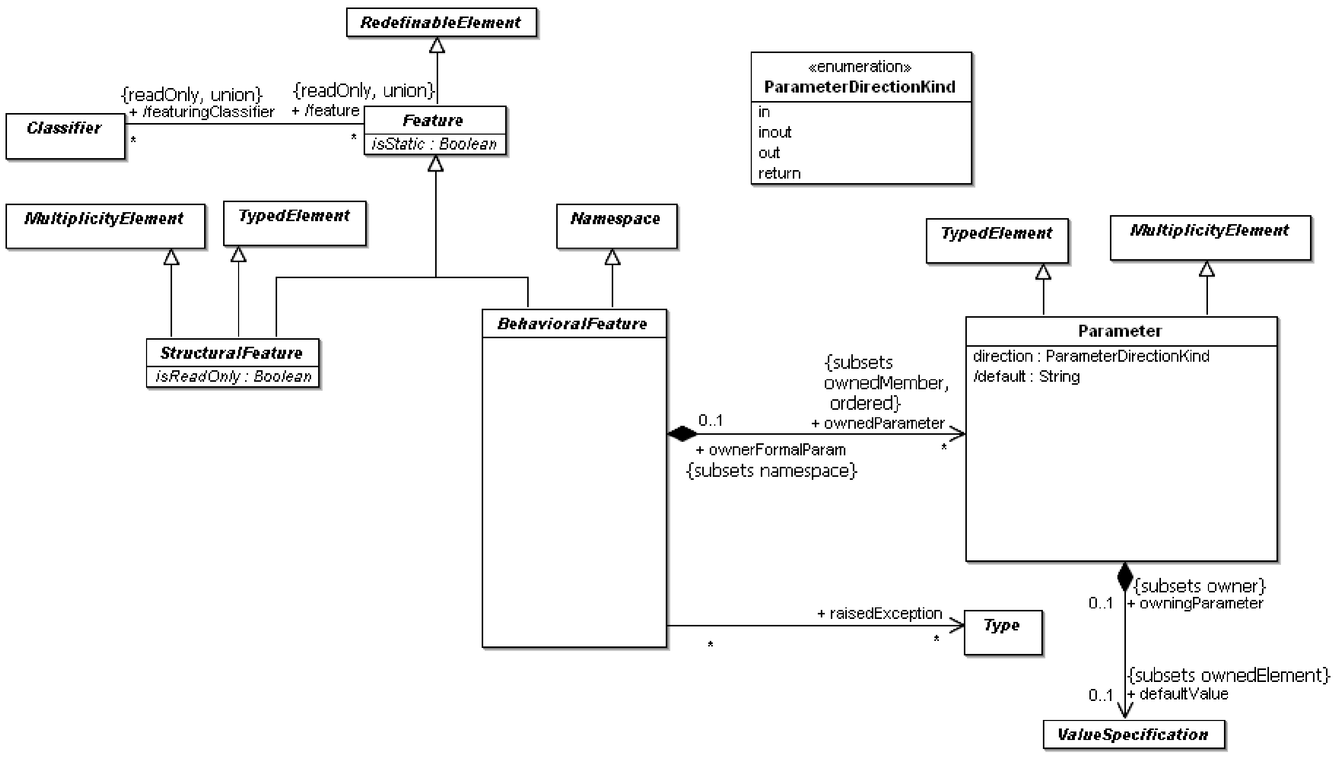


Figure 7.10 - Features diagram of the Kernel package

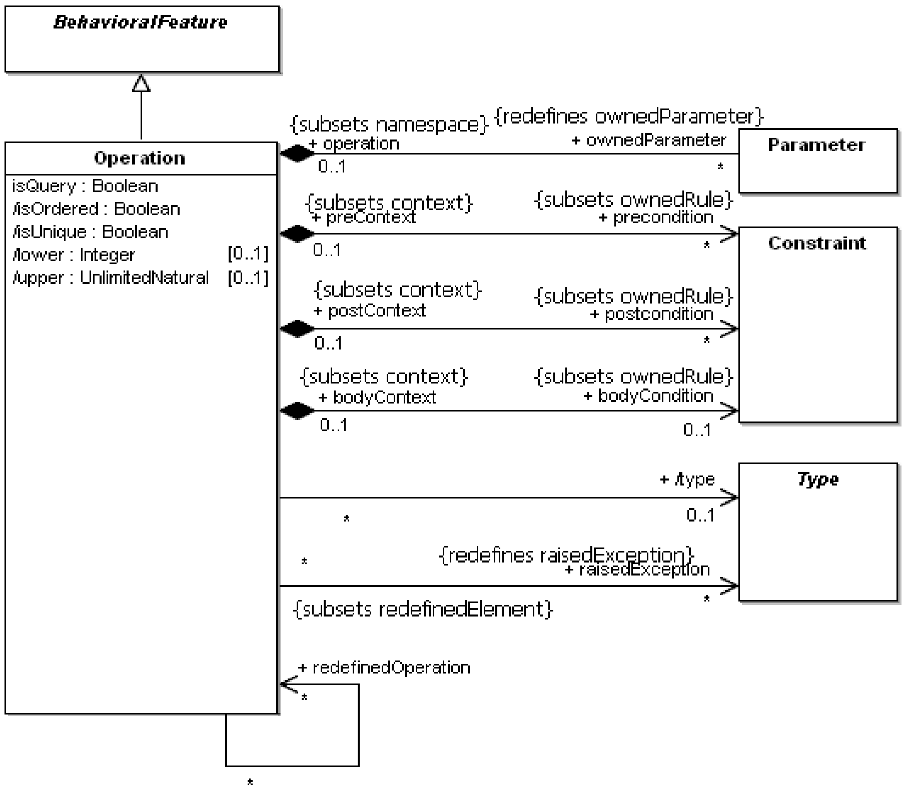


Figure 7.11 - Operations diagram of the Kernel package

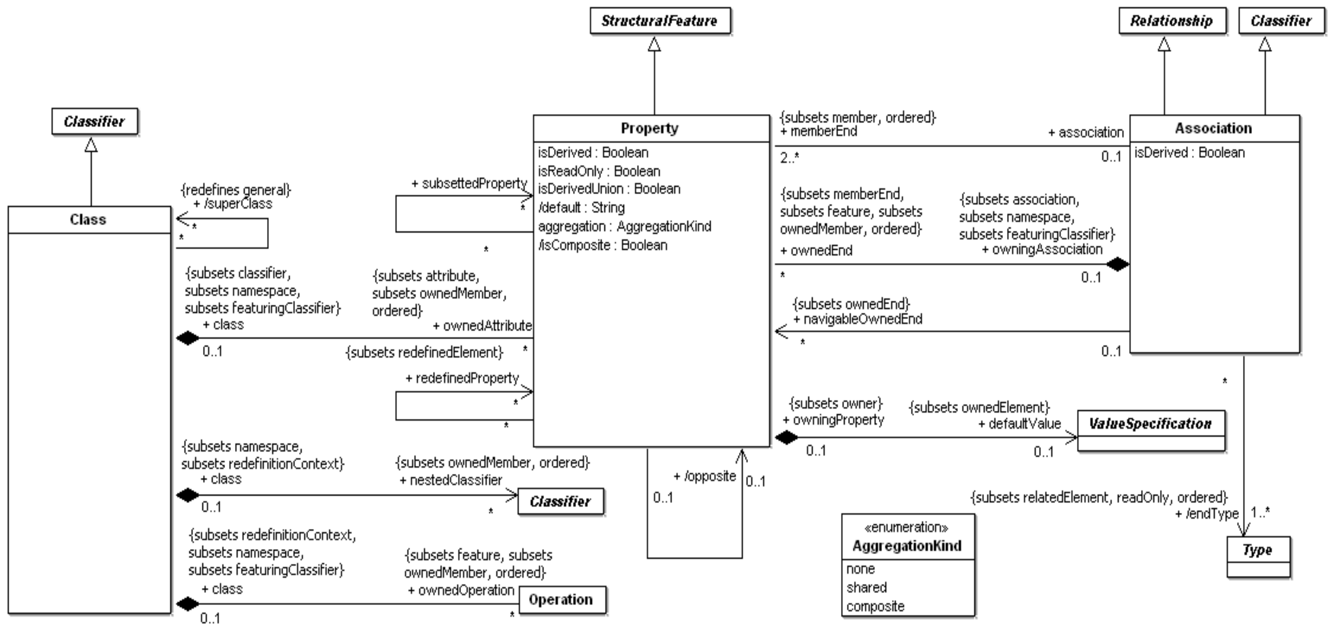


Figure 7.12 - Classes diagram of the Kernel package

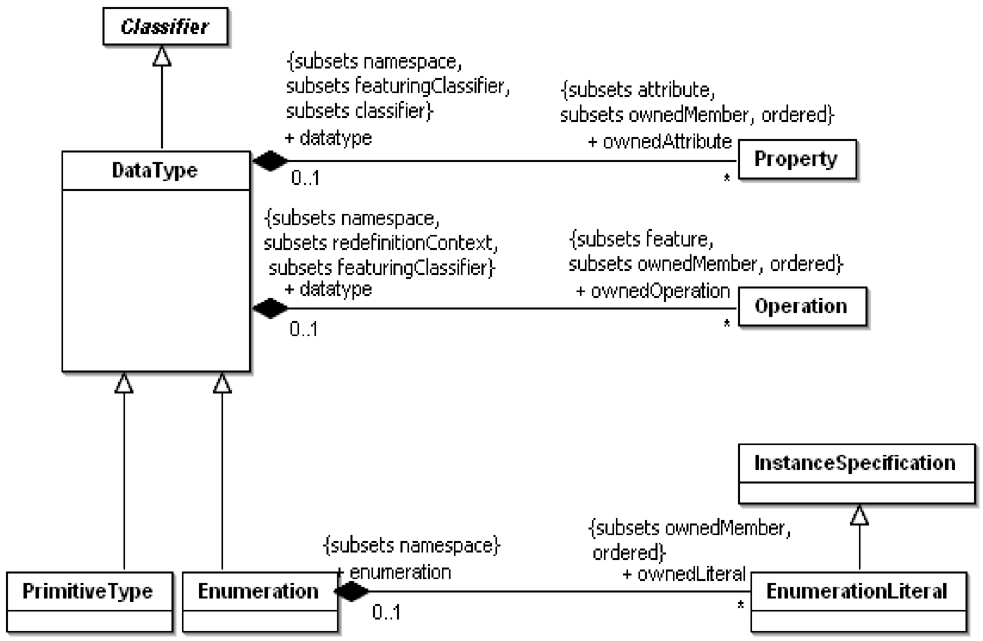


Figure 7.13 - DataTypes diagram of the Kernel package

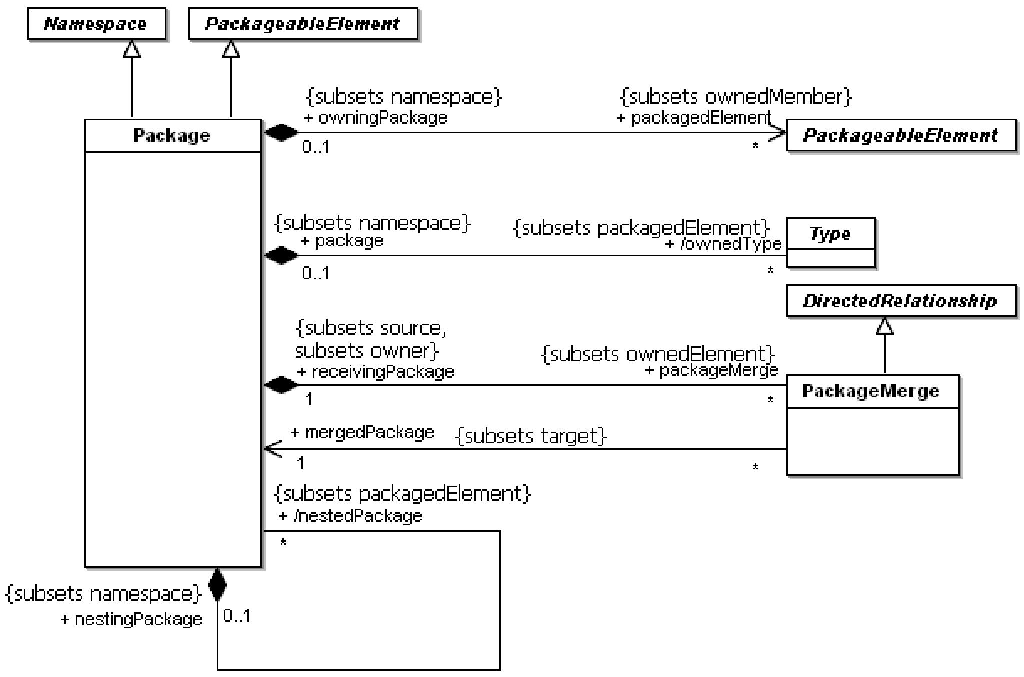


Figure 7.14 - The Packages diagram of the Kernel package

Package Dependencies

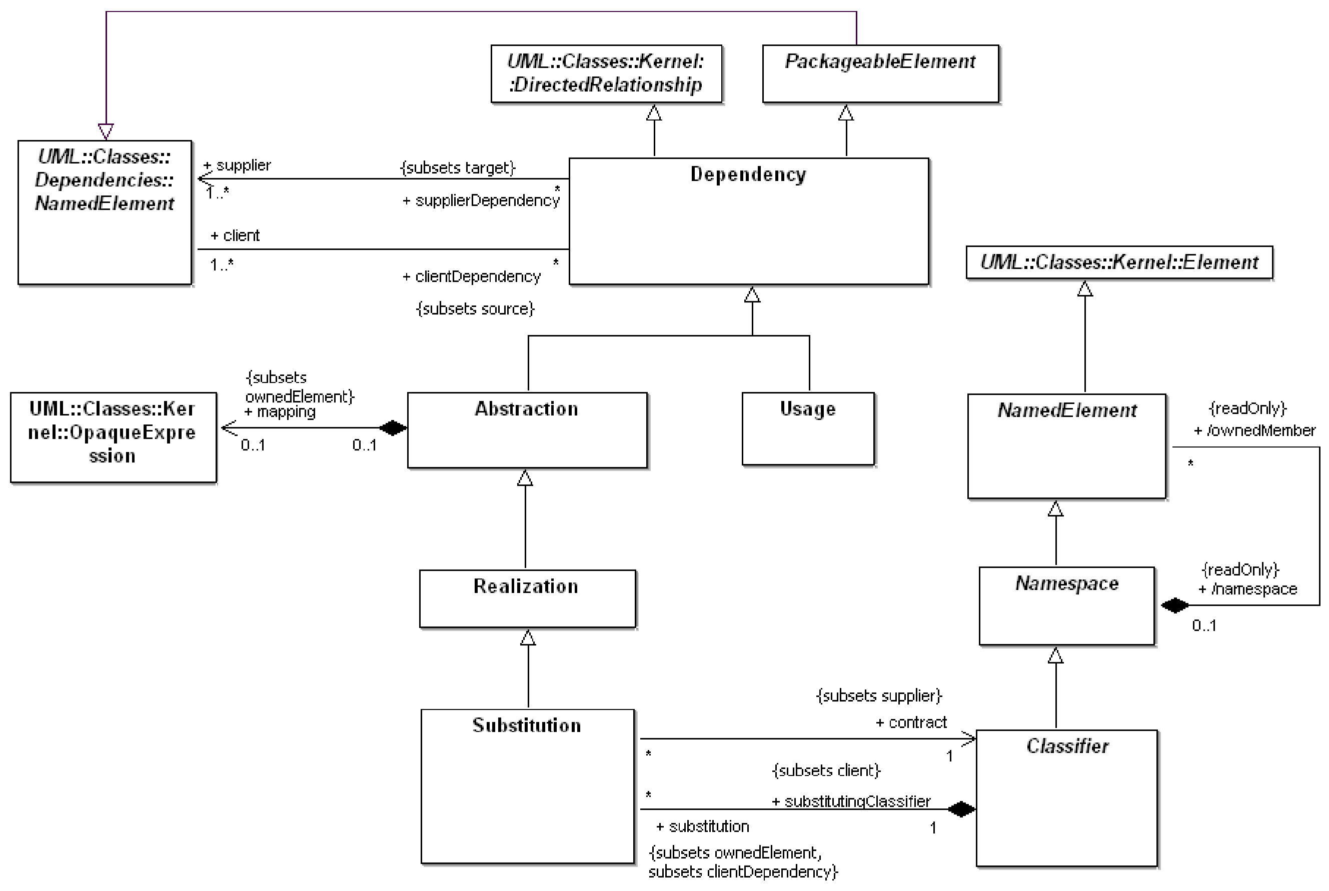
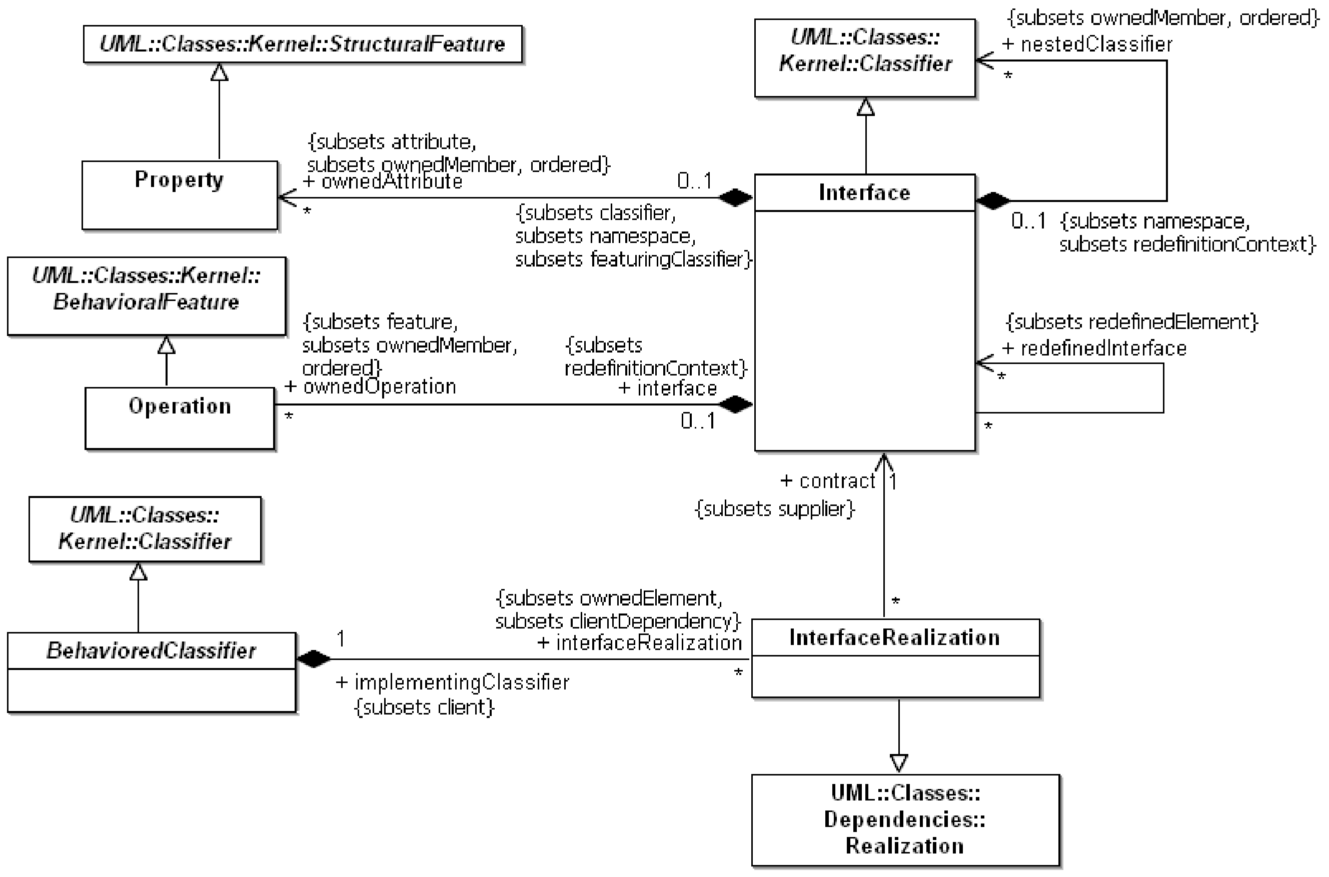


Figure 7.15 - Contents of Dependencies package

Package Interfaces



UML::Classes::Kernel::StructuralFeature
Property
{subsets attribute, subsets ownedMember, ordered} + ownedAttribute
*
0..1
{subsets classifier, subsets namespace, subsets featuringClassifier}
UML::Classes:: Kernel::Classifier
{subsets ownedMember, ordered} + nestedClassifier
*
Interface
0..1 {subsets namespace, subsets redefinitionContext}
UML::Classes::Kernel:: BehavioralFeature
Operation
{subsets feature, subsets ownedMember, ordered} + ownedOperation
*
{subsets redefinitionContext} + interface
0..1
{subsets redefinedElement} + redefinedInterface
*
*
+ contract
1
{subsets supplier}
UML::Classes:: Kernel::Classifier
BehavioredClassifier
1
{subsets ownedElement, subsets clientDependency} + interfaceRealization
*
+ implementingClassifier {subsets client}
*
InterfaceRealization
UML::Classes:: Dependencies:: Realization

Figure 7.16 - Contents of Interfaces package

Package AssociationClasses

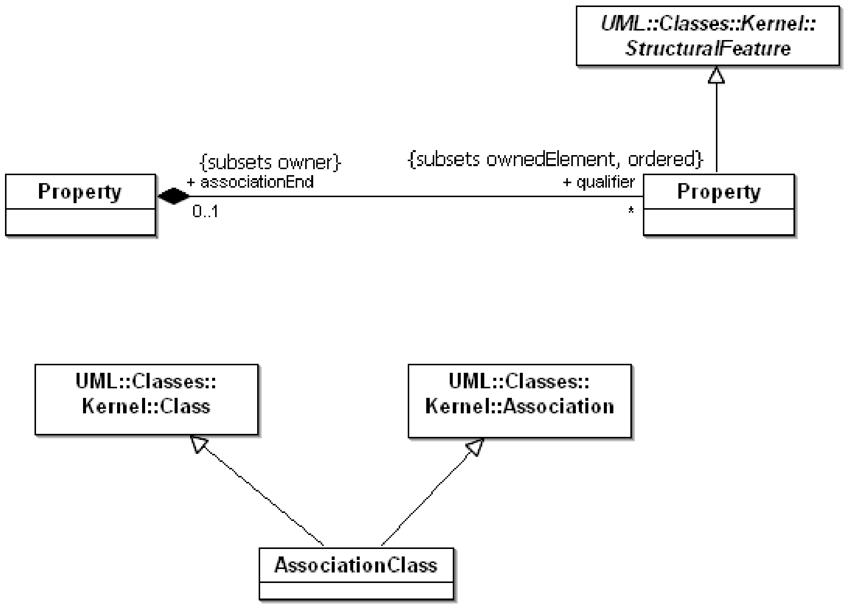


Figure 7.17 - Contents of AssociationClasses package

Package PowerTypes

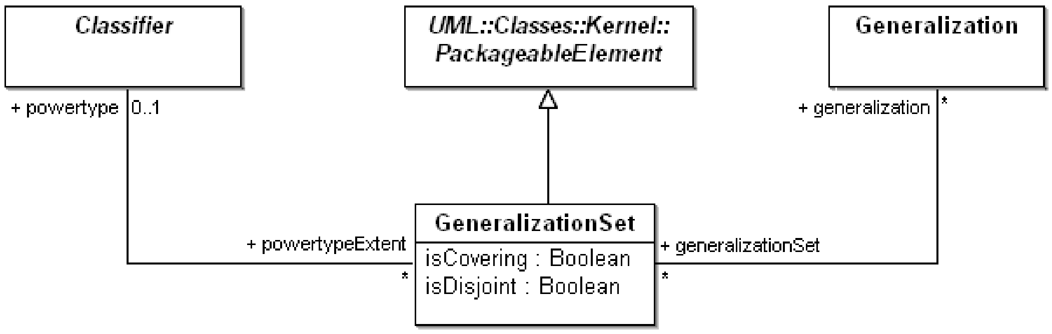


Figure 7.18 - Contents of PowerTypes package

7.3 Class Descriptions

7.3.1 Abstraction (from Dependencies)

Generalizations

- “Dependency (from Dependencies)” on page 62

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. In the metamodel, an Abstraction is a Dependency in which there is a mapping between the supplier and the client.

Attributes

No additional attributes

Associations

- mapping: Expression[0..1]
  A composition of an Expression that states the abstraction relationship between the supplier and the client. 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.

Constraints

No additional constraints

Semantics

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 Profiles clause. If an Abstraction element has more than one client element, the supplier element maps into the set of client elements 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 element.

Notation

An abstraction relationship is shown as a dependency with an «abstraction» keyword attached to it or the specific predefined stereotype name.

7.3.2 AggregationKind (from Kernel)

AggregationKind is an enumeration type that specifies the literals for defining the kind of aggregation of a property.

Generalizations

None

Description

AggregationKind is an enumeration of the following literal values:

- none
  Indicates that the property has no aggregation.
- shared
  Indicates that the property has a 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).

Semantic Variation Points

Precise semantics of shared aggregation varies by application area and modeler.

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

7.3.3 Association (from Kernel)

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

Generalizations

- “Classifier (from Kernel, Dependencies, PowerTypes)” on page 52
- “Relationship (from Kernel)” on page 131

Description

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

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

Attributes

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

Associations

- memberEnd : Property [2..*]
  Each end represents participation of instances of the classifier connected to the end in links of the association. This is an ordered association. Subsets *Namespace::member.*
- ownedEnd : Property [*]
  The ends that are owned by the association itself. This is an ordered association. Subsets *Association::memberEnd, Classifier::feature,* and *Namespace::ownedMember.*

- navigableOwnedEnd : Property [*]
  The navigable ends that are owned by the association itself. Subsets *Association::ownedEnd*.
- / endType: Type [1..*]
  References the classifiers that are used as types of the ends of the association. Subsets *Relationship::relatedElement.*

Constraints

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

```
self.parents()->forAll(p | p.memberEnd.size() = self.memberEnd.size())
```

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

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

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

[4] Only binary associations can be aggregations.

```
self.memberEnd->exists(aggregation <> Aggregation::none) implies self.memberEnd->size() = 2
```

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

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

Semantics

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

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

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

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

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

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

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

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.

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

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

Semantic Variation Points

- The order and way in which part instances in a composite are created is not defined.
- The logical relationship between the derivation of an association and the derivation of its ends is not defined.
- The interaction of association specialization with association end redefinition and subsetting is not defined.

Notation

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

A binary 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 7.21). 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 property string enclosed in curly braces. The following property strings can be applied to an association end:
    - {subsets <property-name>} to show that the end is a subset of the property called <property-name>.
    - {redefines <end-name>} to show that the end redefines the one named <end-name>.
    - {union} to show that the end is derived by being the union of its subsets.
    - {ordered} to show that the end represents an ordered set.
    - {nonunique} to show that the end represents a collection that permits the same element to appear more than once.
    - {sequence} or {seq} to show that the end represents a sequence (an ordered bag).
    - If the end is navigable, any property strings that apply to an attribute.

Note that by default an association end represents a set.

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. A visibility symbol can be added as an adornment on a navigable end to show the end's visibility as an attribute of the featuring classifier.

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

The notation for an attribute can be applied to a navigable end name as specified in the Notation sub clause of "Property (from Kernel, AssociationClasses)" on page 122.

An association with *aggregationKind = shared* differs in notation from binary associations in adding a hollow diamond as a terminal adornment at the aggregate end of the association line. The diamond shall be noticeably smaller than the diamond notation for associations. An association with *aggregationKind = composite* likewise has a diamond at the aggregate 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.

This standard does not mandate the use of explicit end-ownership notation, but defines a notation which shall apply in models where such use is elected. The dot notation must be applied at the level of complete associations or higher, 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.

This notation may only be used on association ends which may, consistent with the metamodel, be owned by classifiers. Users may conceptualize the dot as showing 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.

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.

The dot is illustrated in Figure 7.19, at the maximum allowed size. The diagram shows endA to be owned by classifier B, and because of the rule requiring the notation be applied at the level of complete associations (or above), this diagram also shows unambiguously that end B is owned by BinaryAssociationAB.

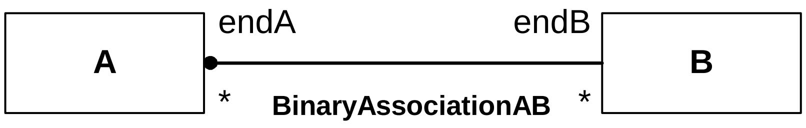


Figure 7.19 - Graphic notation indicating exactly one association end owned by the association

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 orthogonal concepts, each with their own explicit notation. The notational standard now provides for combining these notations as shown in Figure 7.20, where the associated nodes use the default rectangular notation for Classifiers. The dot is outside the perimeter of the rectangle. If non-rectangular notations represent the associated Classifiers, the rule is to put the dot just outside the boundary of the node.

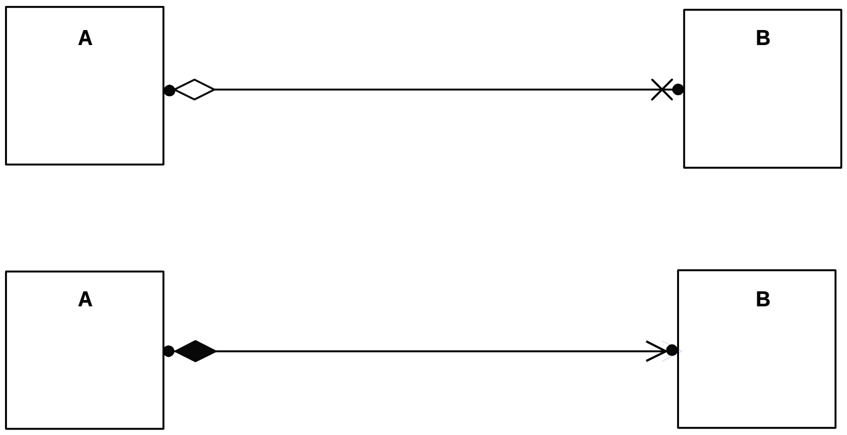


Figure 7.20 - Combining line path graphics

Presentation Options

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

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

- Show all arrows and x’s. Navigation and its absence are made completely explicit.
- Suppress all arrows and x’s. No inference can be drawn about navigation. This is similar to any situation in which information is suppressed from a view.

- Suppress arrows for associations with navigability in both directions, and show arrows only for associations with one-way navigability. In this case, the two-way navigability cannot be distinguished from situations where there is no navigation at all; however, the latter case occurs rarely in practice.

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

Style Guidelines

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

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

Examples

Figure 7.21 shows a binary association from *Player* to *Year* named *PlayedInYear.*

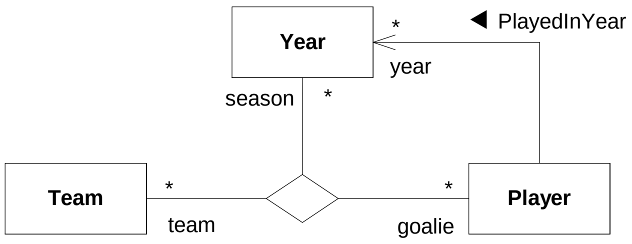


Figure 7.21 - Binary and ternary associations

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.

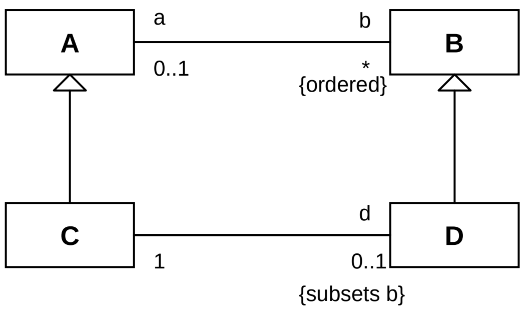


Figure 7.22 - Association ends with various adornments

The following adornments are shown on the four association ends in Figure 7.22.

- Names *a, b,* and *d* on three of the ends.

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

  context C inv: b->includesAll(d)

The following examples show notation for navigable ends.

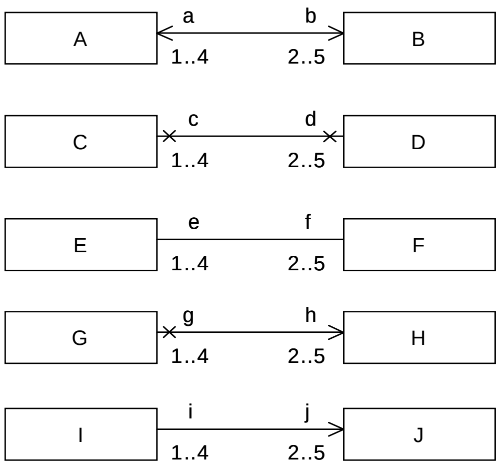


Figure 7.23 - Examples of navigable ends

In Figure 7.23:

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

Figure 7.24 shows that the attribute notation can be used for an association end owned by a class, because an association end owned by a class is also an attribute. This notation may be used in conjunction with the line-arrow notation to make it perfectly clear that the attribute is also an association end.

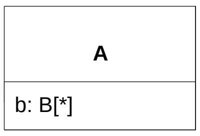


Figure 7.24 - Example of attribute notation for navigable end owned by an end class

Figure 7.25 shows the notation for a derived union. The attribute A::b is derived by being the strict union of all of the attributes that subset it. In this case there is just one of these, A1::b1. So for an instance of the class A1, b1 is a subset of b, and b is derived from b1.

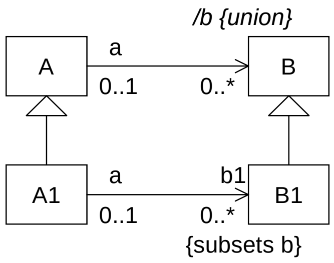


Figure 7.25 - Derived supersets (union)

Figure 7.26 shows the black diamond notation for composite aggregation.

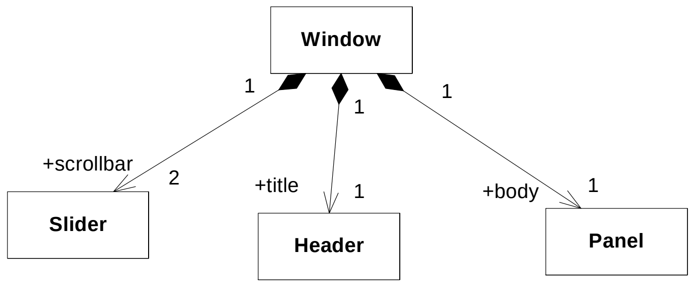


Figure 7.26 - Composite aggregation is depicted as a black diamond

Changes from previous UML

AssociationEnd was a metaclass in prior UML, now demoted to a member of Association. The metaatribute *targetScope* that characterized AssociationEnd in prior UML is no longer supported. Fundamental changes in the abstract syntax make it impossible to continue *targetScope* or replace it by a new metaattribute, or even a standard tag, there being no appropriate model element to tag. In UML 2, the type of the property determines the nature of the values represented by the members of an Association.

7.3.4 AssociationClass (from AssociationClasses)

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 relationship itself and not to any of the classifiers.

Generalizations

- “Association (from Kernel)” on page 39
- “Class (from Kernel)” on page 49

Description

In the metamodel, an AssociationClass is a declaration of a semantic relationship between Classifiers, which has a set of features of its own. AssociationClass is both an Association and a Class.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] An AssociationClass cannot be defined between itself and something else.

```
self.endType->excludes(self) and self.endType>collect(et|et.allparents()->excludes(self))
```

Additional Operations

[1] The operation allConnections results in the set of all AssociationEnds of the Association.

```
AssociationClass::allConnections ( ) : Set ( Property );
allConnections = memberEnd->union ( self.parents ()->collect (p | p.allConnections () )
```

Semantics

An association may be refined to have its own set of features; that is, features that do not belong to any of the connected classifiers but rather to the association itself. Such an association is called an association class. It will be both an association, connecting a set of classifiers and a class, and as such have features and be included in other associations. The semantics of an association class is a combination of the semantics of an ordinary association and of a class.

An association class is both a kind of association and kind of a class. Both of these constructs are classifiers and hence have a set of common properties, like being able to have features, having a name, etc. As these properties are inherited from the same construct (Classifier), they will not be duplicated. Therefore, an association class 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 association class, which implies for example that the attributes of the association class, the ends of the association class, and the opposite ends of associations connected to the association class must all have distinct names. Moreover, the specialization and refinement rules defined for class and association are also applicable to association class.

Note – It should be noted that in an instance of an association class, there is only one instance of the associated classifiers at each end, i.e., from the instance point of view, the multiplicity of the associations ends are '1.'

Notation

An association class is shown as a class symbol attached to the association path by a dashed line. The association path and the association class 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.

Logically, the association class and the association are the same semantic entity; however, they are graphically distinct. The association class symbol can be dragged away from the line, but the dashed line must remain attached to both the path and the class symbol.

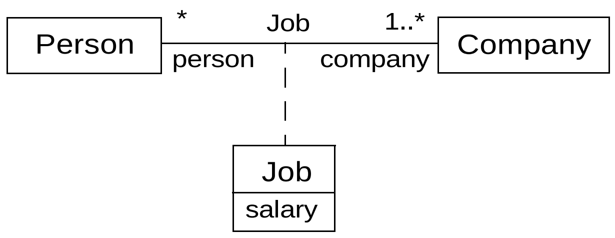


Figure 7.27 - An AssociationClass is depicted by an association symbol (a line) and a class symbol (a box) connected with a dashed line. The diagram shows the association class Job, which is defined between the two classes Person and Company.

7.3.5 BehavioralFeature (from Kernel)

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

Generalizations

- "Feature (from Kernel)" on page 70
- "Namespace (from Kernel)" on page 99

Description

A behavioral feature specifies that an instance of a classifier will respond to a designated request by invoking a behavior. BehavioralFeature is an abstract metaclass specializing Feature and Namespace. Kinds of behavioral aspects are modeled by subclasses of BehavioralFeature.

Attributes

No additional attributes

Associations

- ownedParameter: Parameter[*]
  Specifies the ordered set of formal parameters owned by this BehavioralFeature. The parameter direction can be 'in,' 'inout,' 'out,' or 'return' to specify input, output, or return parameters. Subsets *Namespace::ownedMember*
- raisedException: Type[*]
  References the Types representing exceptions that may be raised during an invocation of this operation.

Constraints

No additional constraints

Additional Operations

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

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

Semantics

The list of owned parameters describes the order, type, and direction of arguments that can be given when the BehavioralFeature is invoked or which are returned when the BehavioralFeature terminates.

The owned parameters with direction in or inout define the type, and number of arguments that must be provided when invoking the BehavioralFeature. An owned parameter with direction out, inout, or return defines the type of the argument that will be returned from a successful invocation. A BehavioralFeature may raise an exception during its invocation.

Notation

No additional notation

7.3.6 BehavioredClassifier (from Interfaces)

Generalizations

- “BehavioredClassifier (from BasicBehaviors, Communications)” on page 434 *(merge increment)*

Description

A BehavioredClassifier may have an interface realization.

Associations

- interfaceRealization: InterfaceRealization [*]
  (Subsets *Element::ownedElement* and *Realization::clientDependency.*)

7.3.7 Class (from Kernel)

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

Generalizations

- “Classifier (from Kernel, Dependencies, PowerTypes)” on page 52

Description

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

Attributes

No additional attributes

Associations

- nestedClassifier: Classifier [*]
  References all the Classifiers that are defined (nested) within the Class. Subsets *Element::ownedMember*
- ownedAttribute : Property [*]
  The attributes (i.e., the properties) owned by the class. The association is ordered. Subsets *Classifier::attribute* and *Namespace::ownedMember*
- ownedOperation : Operation [*]
  The operations owned by the class. The association is ordered. Subsets *Classifier::feature* and *Namespace::ownedMember*
- / superClass : Class [*]
  This gives the superclasses of a class. It redefines *Classifier::general*. This is derived.

Constraints

No additional constraints

Additional Operations

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

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

Semantics

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

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

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

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

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

Notation

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 classifier symbol without a metaclass shown in guillemets indicates a class.

Presentation Options

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

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

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

Style Guidelines

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

Examples

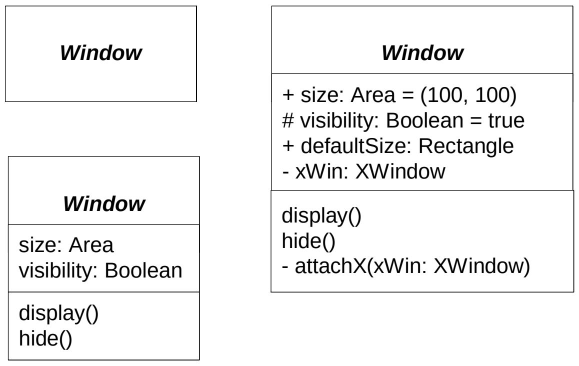


Figure 7.28 - Class notation: details suppressed, analysis-level details, implementation-level details

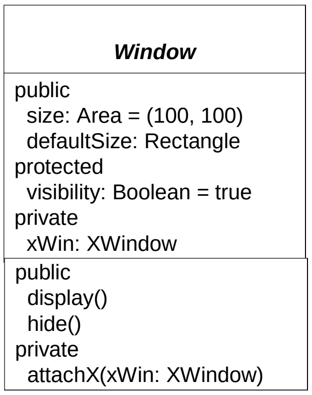


Figure 7.29 - Class notation: attributes and operations grouped according to visibility

7.3.8 Classifier (from Kernel, Dependencies, PowerTypes)

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

Generalizations

- "Namespace (from Kernel)" on page 99
- "RedefinableElement (from Kernel)" on page 130
- "Type (from Kernel)" on page 135

Description

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

A classifier is a type and can own generalizations, thereby making it possible to define generalization relationships to other classifiers. A classifier can specify a generalization hierarchy by referencing its general classifiers.

A classifier is a redefinable element, meaning that it is possible to redefine nested classifiers.

Attributes

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

Associations

- /attribute: Property [*]
  Refers to all of the Properties that are direct (i.e., not inherited or imported) attributes of the classifier. Subsets *Classifier::feature* and is a derived union.
- / feature : Feature [*]
  Specifies each feature defined in the classifier. Subsets *Namespace::member.* This is a derived union.
- / general : Classifier[*]
  Specifies the general Classifiers for this Classifier. This is derived.

- generalization: Generalization[*]
  Specifies the Generalization relationships for this Classifier. These Generalizations navigate to more general classifiers in the generalization hierarchy. Subsets *Element::ownedElement*
- / inheritedMember: NamedElement[*]
  Specifies all elements inherited by this classifier from the general classifiers. Subsets *Namespace::member*. This is derived.
- redefinedClassifier: Classifier [*]
  References the Classifiers that are redefined by this Classifier. Subsets *RedefinableElement::redefinedElement*

Package Dependencies

- substitution : Substitution
  References the substitutions that are owned by this Classifier. Subsets *Element::ownedElement* and *NamedElement::clientDependency*.)

Package PowerTypes

- powertypeExtent : GeneralizationSet
  Designates the GeneralizationSet of which the associated Classifier is a power type.

Constraints

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

general = self.parents()

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

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

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

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

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

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

Package PowerTypes

[5] 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.

Additional Operations

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

Classifier::allFeatures(): Set(Feature);

allFeatures = member->select(oclIsKindOf(Feature))

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

Classifier::parents(): Set(Classifier);

parents = generalization.general

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

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

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

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

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

```
Classifier::hasVisibilityOf(n: NamedElement) : Boolean;
pre: self.allParents()->collect(c | c.member)->includes(n)
    if (self.inheritedMember->includes(n)) then
        hasVisibilityOf = (n.visibility <> #private)
    else
        hasVisibilityOf = true
```

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

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

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

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

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

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

Semantics

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

A Classifier may participate in generalization relationships with other Classifiers. An instance of a specific Classifier is also an (indirect) instance of each of the general Classifiers. Therefore, features specified for instances of the general classifier are implicitly specified for instances of the specific classifier. Any constraint applying to instances of the general classifier also applies to instances of the specific classifier.

The specific semantics of how generalization affects each concrete subtype of Classifier varies. All instances of a classifier have values corresponding to the classifier’s attributes.

A Classifier defines a type. Type conformance between generalizable Classifiers is defined so that a Classifier conforms to itself and to all of its ancestors in the generalization hierarchy.

Package PowerTypes

The notion of power type was inspired by the notion of power set. A power set is defined as a set whose instances are subsets. In essence, then, a power type is a class whose instances are subclasses. The powertypeExtent association relates a Classifier with a set of generalizations that a) have a common specific Classifier, and b) represent a collection of subsets for that class.

Semantic Variation Points

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

Notation

Classifier is an abstract model element, and so properly speaking has no notation. It is nevertheless convenient to define in one place a default notation available for any concrete subclass of Classifier for which this notation is suitable. The default notation for a classifier is a solid-outline rectangle containing the classifier's name, and optionally with compartments separated by horizontal lines containing features or other members of the classifier. The specific type of classifier can be shown in guillemets above the name. Some specializations of Classifier have their own distinct notations.

The name of an abstract Classifier is shown in italics.

An attribute can be shown as a text string. The format of this string is specified in the Notation sub clause of "Property (from Kernel, AssociationClasses)" on page 122.

Presentation Options

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

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

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

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

Style Guidelines

- Attribute names typically begin with a lowercase letter. Multi-word names are often formed by concatenating the words and using lowercase for all letters except for upcasing the first letter of each word but the first.
- Center the name of the classifier in boldface.
- Center keyword (including stereotype names) in plain face within guillemets above the classifier name.
- For those languages that distinguish between uppercase and lowercase characters, capitalize names (i.e, begin them with an uppercase character).
- Left justify attributes and operations in plain face.
- Begin attribute and operation names with a lowercase letter.
- Show full attributes and operations when needed and suppress them in other contexts or references.

Examples

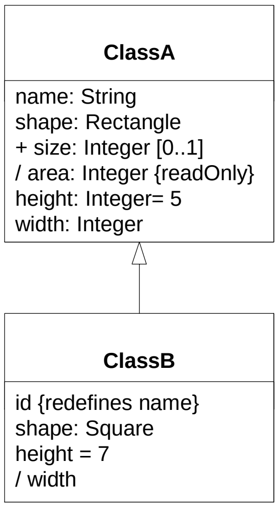


Figure 7.30 - Examples of attributes

The attributes in Figure 7.30 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.

An attribute may also be shown using association notation, with no adornments at the tail of the arrow as shown in Figure 7.31.

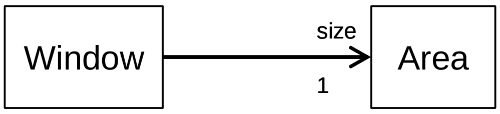


Figure 7.31 - Association-like notation for attribute

Package PowerTypes

For example, a Bank Account Type classifier could have a powertype association with a GeneralizationSet. This GeneralizationSet could then associate with two Generalizations where the class (i.e., general Classifier) Bank Account has two specific subclasses (i.e., Classifiers): Checking Account and Savings Account. Checking Account and Savings Account, then, are instances of the power type: Bank Account Type. In other words, Checking Account and Savings Account are *both:* instances of Bank Account Type, as well as subclasses of Bank Account. (For more explanation and examples, see Examples in the GeneralizationSet sub clause, below.)

7.3.9 Comment (from Kernel)

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

Generalizations

- "Element (from Kernel)" on page 64.

Description

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

A comment can be owned by any element.

Attributes

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

Associations

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

Constraints

No additional constraints

Semantics

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

Notation

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

Presentation Options

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

Examples

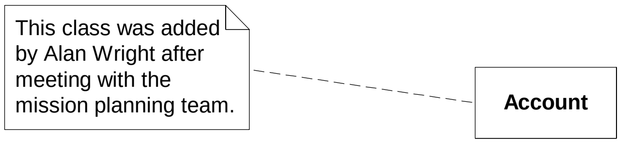

This class was added by Alan Wright after meeting with the mission planning team.
Account

Figure 7.32 - Comment notation

7.3.10 Constraint (from Kernel)

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.

Generalizations

- "PackageableElement (from Kernel)" on page 109

Description

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

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

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

Attributes

No additional attributes

Associations

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

Constraints

[1] The value specification for a constraint must evaluate to a Boolean value.
Cannot be expressed in OCL.

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

Cannot be expressed in OCL.

[3] A constraint cannot be applied to itself.

not constrainedElement->includes(self)

Semantics

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

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

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

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

Notation

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

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

For an element whose notation is a text string (such as an attribute, etc.), the constraint string may follow the element text string in braces. Figure 7.33 shows a constraint string that follows an attribute within a class symbol.

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

For a Constraint that applies to two elements (such as two classes or two associations), the constraint may be shown as a dashed line between the elements labeled by the constraint string (in braces). Figure 7.34 shows an {xor} constraint between two associations.

Presentation Options

The constraint string may be placed in a note symbol and attached to each of the symbols for the constrained elements by a dashed line. Figure 7.35 shows an example of a constraint in a note symbol.

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

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

Examples

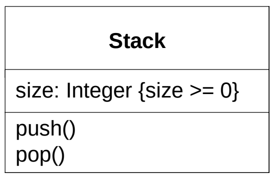


Figure 7.33 - Constraint attached to an attribute

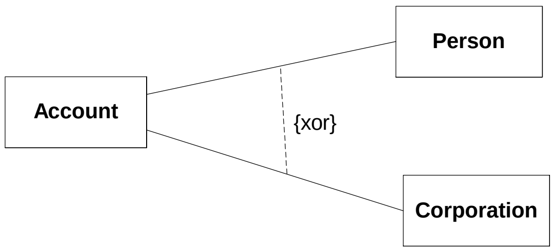


Figure 7.34 - {xor} constraint

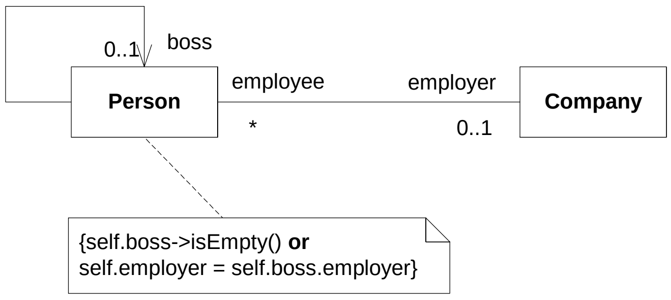

0..1
boss
Person
employee
employer
Company
*
0..1
{self.boss->isEmpty() **or**
self.employer = self.boss.employer}

Figure 7.35 - Constraint in a note symbol

7.3.11 DataType (from Kernel)

Generalizations

- "Classifier (from Kernel, Dependencies, PowerTypes)" on page 52.

Description

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

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

Attributes

No additional attributes

Associations

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

Constraints

No additional constraints

Semantics

A data type is a special kind of classifier, similar to a class. It differs from a class in that instances of a data type are identified only by their value.

All copies of an instance of a data type and any instances of that data type with the same value are considered to be the same instance. Instances of a data type that have attributes (i.e., is a structured data type) are considered to be the same if the structure is the same and the values of the corresponding attributes are the same. If a data type has attributes, then instances of that data type will contain attribute values matching the attributes.

Semantic Variation Points

Any restrictions on the capabilities of data types, such as constraining the types of their attributes, is a semantic variation point.

Notation

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

Examples

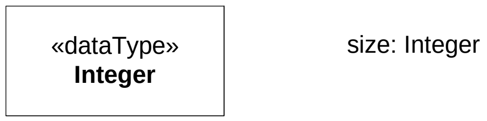


Figure 7.36 - Notation of data type: to the left is an icon denoting a data type and to the right is a reference to a data type that is used in an attribute.

7.3.12 Dependency (from Dependencies)

Generalizations

- “DirectedRelationship (from Kernel)” on page 63
- “PackageableElement (from Kernel)” on page 109

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 depending elements is either semantically or structurally dependent on the definition of the supplier element(s).

Attributes

No additional attributes

Associations

- client: NamedElement [1..*]
  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. Subsets *DirectedRelationship::source.*

- supplier: NamedElement [1..*]
  The element(s) independent of the client element(s), in the same respect and the same dependency relationship. In some directed dependency relationships (such as Refinement Abstractions), a common convention in the domain of class-based OO software is to put the more abstract element in this role. Despite this convention, users of UML may stipulate a sense of dependency suitable for their domain, which makes a more abstract element dependent on that which is more specific. Subsets *DirectedRelationship::target*.

Constraints

No additional constraints

Semantics

A dependency signifies a supplier/client relationship between model elements where the modification of the supplier may impact the client model elements. A dependency implies the semantics of the client is not complete without the supplier. The presence of dependency relationships in a model does not have any runtime semantics implications, it is all given in terms of the model-elements that participate in the relationship, not in terms of their instances.

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 stereotype and an optional name. 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.

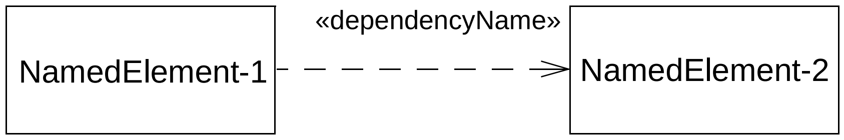


Figure 7.37 - Notation for a dependency between two elements

Examples

In the example below, the Car class has a dependency on the CarFactory class. In this case, the dependency is an instantiate dependency, where the Car class is an instance of the CarFactory class.

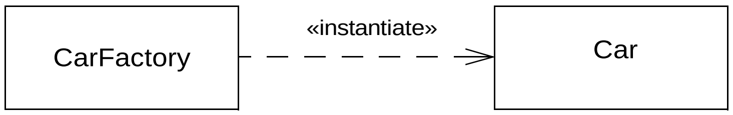


Figure 7.38 - An example of an instantiate dependency

7.3.13 DirectedRelationship (from Kernel)

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

Generalizations

- “Relationship (from Kernel)” on page 131

Description

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

Attributes

No additional attributes

Associations

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

Constraints

No additional constraints

Semantics

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

Notation

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

7.3.14 Element (from Kernel)

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

Generalizations

None

Description

Element is an abstract metaclass with no superclass. It is used as the common superclass for all metaclasses in the infrastructure library. Element has a derived composition association to itself to support the general capability for elements to own other elements.

Attributes

No additional attributes

Associations

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

Constraints

[1] An element may not directly or indirectly own itself.

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

[2] Elements that must be owned must have an owner.

self.mustBeOwned() **implies** owner->notEmpty()

Additional Operations

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

Element::allOwnedElements(): Set(Element);
allOwnedElements = ownedElement->union(ownedElement->collect(e | e.allOwnedElements()))

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

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

Semantics

Subclasses of Element provide semantics appropriate to the concept they represent. The comments for an Element add no semantics but may represent information useful to the reader of the model.

Notation

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

7.3.15 ElementImport (from Kernel)

An element import identifies an element in another package, and allows the element to be referenced using its name without a qualifier.

Generalizations

- “DirectedRelationship (from Kernel)” on page 63

Description

An element import is defined as a directed relationship between an importing namespace and a packageable element. The name of the packageable element or its alias is to be added to the namespace of the importing namespace. It is also possible to control whether the imported element can be further imported.

Attributes

- visibility: VisibilityKind
  Specifies the visibility of the imported PackageableElement within the importing Package. The default visibility is the same as that of the imported element. If the imported element does not have a visibility, it is possible to add visibility to the element import. Default value is *public*.

- alias: String [0..1]
  Specifies the name that should be added to the namespace of the importing Package in lieu of the name of the imported PackagableElement. The aliased name must not clash with any other member name in the importing Package. By default, no alias is used.

Associations

- importedElement: PackageableElement [1]
  Specifies the PackageableElement whose name is to be added to a Namespace. Subsets *DirectedRelationship::target*.

- importingNamespace: Namespace [1]
  Specifies the Namespace that imports a PackageableElement from another Package. Subsets *DirectedRelationship::source* and *Element::owner.*

Constraints

[1] The visibility of an ElementImport is either public or private.

self.visibility = #public **or** self.visibility = #private

[2] An importedElement has either public visibility or no visibility at all.

self.importedElement.visibility.notEmpty() **implies** self.importedElement.visibility = #public

Additional Operations

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

```
ElementImport::getName(): String;
getName =
    if self.alias->notEmpty() then
        self.alias
    else
        self.importedElement.name
    endif
```

Semantics

An element import adds the name of a packageable element from a package to the importing namespace. It works by reference, which means that it is not possible to add features to the element import itself, but it is possible to modify the referenced element in the namespace from which it was imported. An element import is used to selectively import individual elements without relying on a package import.

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

If more than one element with the same name would be imported to a namespace as a consequence of element imports or package imports, the elements are not added to the importing namespace and the names of those elements must be qualified in order to be used in that namespace. If the name of an imported element is the same as the name of an element owned by the importing namespace, that element is not added to the importing namespace and the name of that element must be qualified in order to be used.

An imported element can be further imported by other namespaces using either element or package imports.

The visibility of the ElementImport may be either the same or more restricted than that of the imported element.

Notation

An element import is shown using a dashed arrow with an open arrowhead from the importing namespace to the imported element. The keyword «import» is shown near the dashed arrow if the visibility is public; otherwise, the keyword «access» is shown to indicate private visibility.

If an element import has an alias, this is used in lieu of the name of the imported element. The aliased name may be shown after or below the keyword «import».

Presentation options

If the imported element is a package, the keyword may optionally be preceded by element, i.e., «element import».

As an alternative to the dashed arrow, it is possible to show an element import by having a text that uniquely identifies the imported element within curly brackets either below or after the name of the namespace. The textual syntax is then:

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

Optionally, the aliased name may be shown as well:

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

Examples

The element import that is shown in Figure 7.39 allows elements in the package Program to refer to the type Time in Types without qualification. However, they still need to refer explicitly to Types::Integer, since this element is not imported. The Type string can be used in the Program package but cannot be further imported from that package.

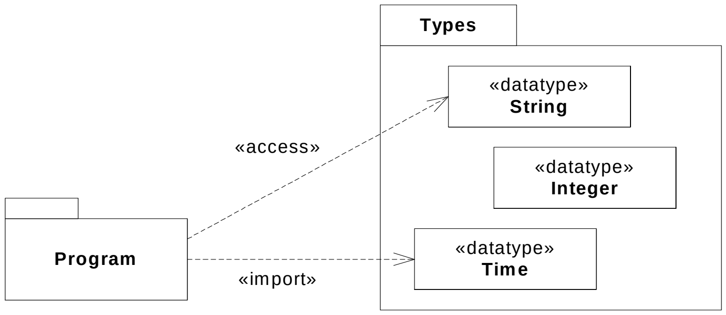


Figure 7.39 - Example of element import

In Figure 7.40, the element import is combined with aliasing, meaning that the type Types::Real will be referred to as Double in the package Shapes.

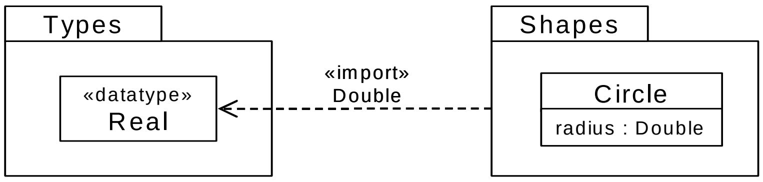


Figure 7.40 - Example of element import with aliasing

7.3.16 Enumeration (from Kernel)

An enumeration is a data type whose values are enumerated in the model as enumeration literals.

Generalizations

- “DataType (from Kernel)” on page 60

Description

Enumeration is a kind of data type, whose instances may be any of a number of user-defined enumeration literals.

It is possible to extend the set of applicable enumeration literals in other packages or profiles.

Attributes

No additional attributes

Associations

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

Constraints

No additional constraints

Semantics

The run-time instances of an Enumeration are data values. Each such value corresponds to exactly one EnumerationLiteral.

Notation

An enumeration may be shown using the classifier notation (a rectangle) with the keyword «enumeration». The name of the enumeration is placed in the upper compartment. A compartment listing the attributes for the enumeration is placed below the name compartment. A compartment listing the operations for the enumeration is placed below the attribute compartment. A list of enumeration literals may be placed, one to a line, in the bottom compartment. The attributes and operations compartments may be suppressed, and typically are suppressed if they would be empty.

Examples

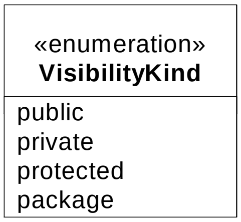


Figure 7.41 - Example of an enumeration

7.3.17 EnumerationLiteral (from Kernel)

An enumeration literal is a user-defined data value for an enumeration.

Generalizations

- "InstanceSpecification (from Kernel)" on page 82

Description

An enumeration literal is a user-defined data value for an enumeration.

Attributes

No additional attributes

Associations

- enumeration: Enumeration[0..1]
  The Enumeration that this EnumerationLiteral is a member of. Subsets *NamedElement::namespace*

Constraints

No additional constraints

Semantics

An EnumerationLiteral defines an element of the run-time extension of an enumeration data type.

An EnumerationLiteral has a name that can be used to identify it within its enumeration datatype. The enumeration literal name is scoped within and must be unique within its enumeration. Enumeration literal names are not global and must be qualified for general use.

The run-time values corresponding to enumeration literals can be compared for equality.

Notation

An EnumerationLiteral is typically shown as a name, one to a line, in the compartment of the enumeration notation.

7.3.18 Expression (from Kernel)

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

Generalizations

- "ValueSpecification (from Kernel)" on page 137

Description

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

Attributes

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

Associations

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

Constraints

No additional constraints

Semantics

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

Notation

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

Examples

xor
else
plus(x,1)
x+1

7.3.19 Feature (from Kernel)

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

Generalizations

- "RedefinableElement (from Kernel)" on page 130

Description

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

Attributes

- isStatic: Boolean
  Specifies whether this feature characterizes individual instances classified by the classifier (*false*) or the classifier itself (*true*). Default value is *false*.

Associations

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

Constraints

No additional constraints

Semantics

A feature represents some characteristic for its featuring classifiers; this characteristic may be of the classifier's instances considered individually (*not static*), or of the classifier itself (*static*). A Feature can be a feature of multiple classifiers. The same feature cannot be static in one context but not another.

Semantic Variation Points

With regard to static features, two alternative semantics are recognized. A static feature may have different values for different featuring classifiers, or the same value for all featuring classifiers.

In accord with this semantic variation point, inheritance of values for static features is permitted but not required by UML 2. Such inheritance is encouraged when modeling systems will be coded in languages, such as C++, Java, and C#, which stipulate inheritance of values for static features.

Notation

No general notation. Subclasses define their specific notation.

Static features are underlined.

Presentation Options

Only the names of static features are underlined.

An ellipsis (...) as the final element of a list of features indicates that additional features exist but are not shown in that list.

Changes from previous UML

The property *isStatic* in UML 2 serves in place of the metaattribute *ownerScope* of Feature in UML 1. The enumerated data type *ScopeKind* with two values, *instance* and *classifier*, provided in UML 1 as the type for *ownerScope* is no longer needed because *isStatic* is Boolean.

7.3.20 Generalization (from Kernel, PowerTypes)

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 indirect instance of the general classifier. Thus, the specific classifier inherits the features of the more general classifier.

Generalizations

- "DirectedRelationship (from Kernel)" on page 63

Description

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

Package PowerTypes

A generalization can be designated as being a member of a particular generalization set.

Attributes

- isSubstitutable: Boolean [0..1]
  Indicates whether the specific classifier can be used wherever the general classifier can be used. If *true*, the execution traces of the specific classifier will be a superset of the execution traces of the general classifier. The default value is *true.*

Associations

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

Package PowerTypes

- generalizationSet
  Designates a set in which instances of Generalization are considered members.

Constraints

No additional constraints

Package PowerTypes

[1] Every Generalization associated with a given GeneralizationSet must have the same general Classifier. That is, all Generalizations for a particular GeneralizationSet must have the same superclass.

Semantics

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

Package PowerTypes

Each Generalization is a binary relationship that relates a specific Classifier to a more general Classifier (i.e., a subclass). Each GeneralizationSet contains a particular set of Generalization relationships that *collectively* describe the way in which a specific Classifier (or class) may be divided into subclasses. The generalizationSet associates those instances of a Generalization with a particular GeneralizationSet.

For example, one Generalization could relate Person as a general Classifier with a Female Person as the specific Classifier. Another Generalization could also relate Person as a general Classifier, but have Male Person as the specific Classifier. These two Generalizations could be associated with the same GeneralizationSet, because they specify one way of partitioning the Person class.

Notation

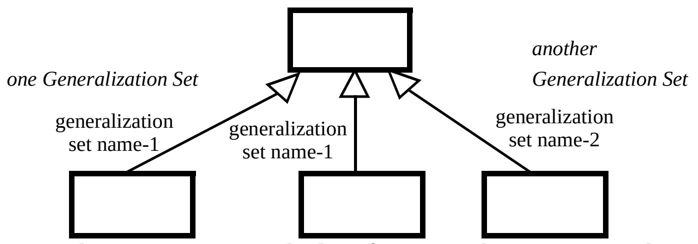
A Generalization is shown as a line with a hollow triangle as an arrowhead between the symbols representing the involved classifiers. The arrowhead points to the symbol representing the general classifier. This notation is referred to as the “separate target style.” See the example sub clause below.

Package PowerTypes

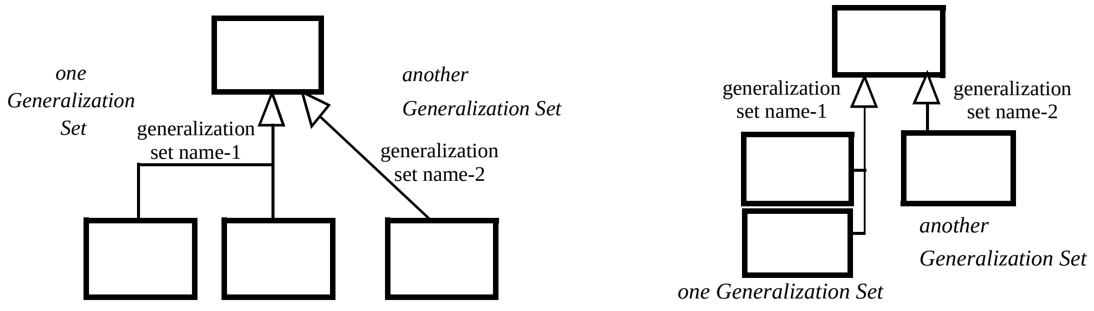
A generalization is shown as a line with a hollow triangle as an arrowhead between the symbols representing the involved classifiers. The arrowhead points to the symbol representing the general classifier. When these relationships are named, that name designates the GeneralizationSet to which the Generalization belongs. Each GeneralizationSet has a name (which it inherits since it is a subclass of PackageableElement). Therefore, all Generalization relationships with the same GeneralizationSet name are part of the same GeneralizationSet. This notation form is depicted in a), Figure 7.42.

When two or more lines are drawn to the same arrowhead, as illustrated in b), Figure 7.42, the specific Classifiers are part of the same GeneralizationSet. When diagrammed in this way, the lines do not need to be labeled separately; instead the generalization set need only be labeled once. The labels are optional because the GeneralizationSet is clearly designated.

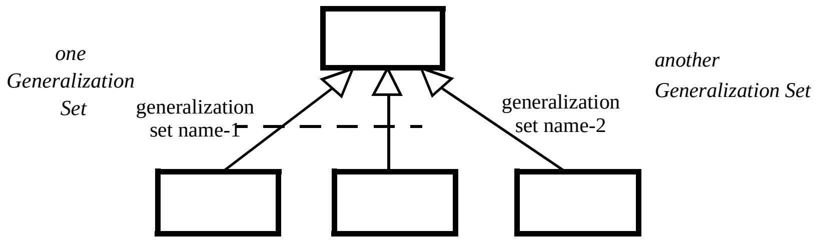
Lastly in c), Figure 7.42, a GeneralizationSet can be designated by drawing a dashed line across those lines with separate arrowheads that are meant to be part of the same set, as illustrated at the bottom of Figure 7.42. Here, as with b), the GeneralizationSet may be labeled with a single name, instead of each line labeled separately. However, such labels are optional because the GeneralizationSet is clearly designated.


one Generalization Set
another Generalization Set
generalization set name-1
generalization set name-1
generalization set name-2

a) GeneralizationSet sharing same general Classifier using the same generalization relationship names.


one Generalization Set
another Generalization Set
generalization set name-1
generalization set name-2
generalization set name-1
generalization set name-2
another Generalization Set
one Generalization Set

b) GeneralizationSet designation by subtypes sharing a common generalization arrowhead.


one Generalization Set
another Generalization Set
generalization set name-1
generalization set name-2

c) GeneralizationSet sharing same general Classifier using the dashed-line notation.

Figure 7.42 - GeneralizationSet designation notations

Presentation Options

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

Examples

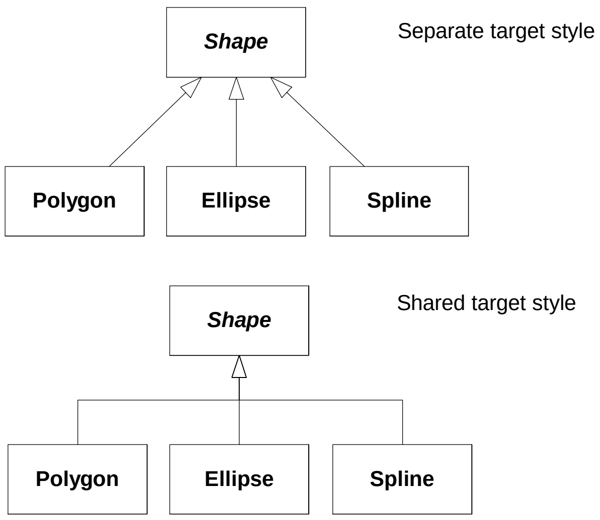


Figure 7.43 - Examples of generalizations between classes

Package PowerTypes

In Figure 7.44, the Person class can be specialized as either a Female Person or a Male Person. Furthermore, Person's can be specialized as an Employee. Here, Female Person or a Male Person of Person constitute one GeneralizationSet and Employee another. This illustration employs the notation forms depicted in the diagram above.

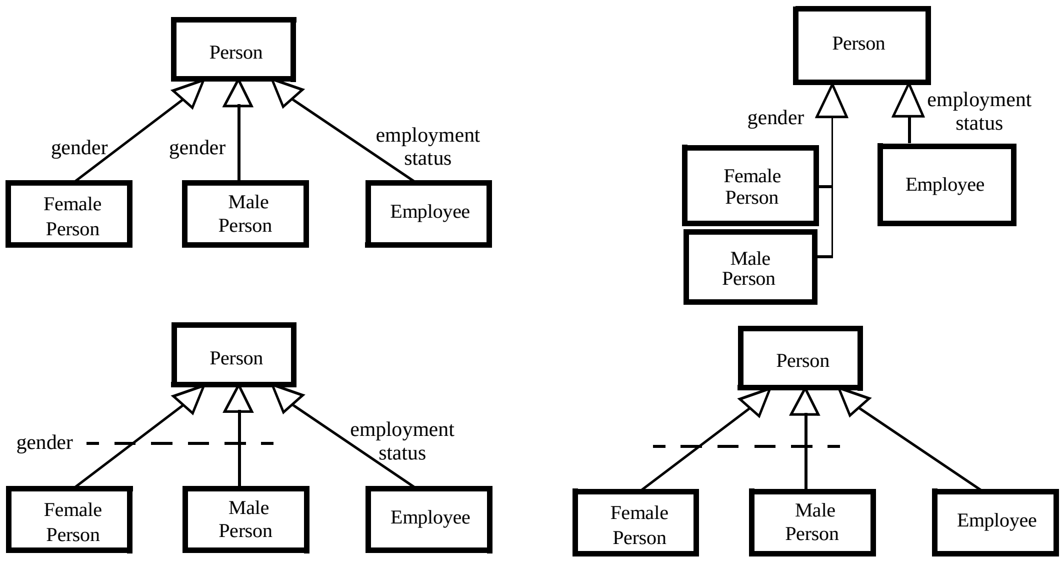


Figure 7.44 - Multiple subtype partitions (GeneralizationSets) example

7.3.21 GeneralizationSet (from PowerTypes)

A GeneralizationSet is a PackageableElement (from Kernel) whose instances define collections of subsets of Generalization relationships.

Generalizations

- “PackageableElement (from Kernel)” on page 109

Description

Each Generalization is a binary relationship that relates a specific Classifier to a more general Classifier (i.e., from a class to its superclasses). Each GeneralizationSet defines a particular set of Generalization relationships that describe the way in which a general Classifier (or superclass) may be divided using specific subtypes. For example, a GeneralizationSet could define a partitioning of the class Person into two subclasses: Male Person and Female Person. Here, the GeneralizationSet would associate two instances of Generalization. Both instances would have Person as the general classifier; however, one Generalization would involve Male Person as the specific Classifier and the other would involve Female Person as the specific classifier. In other words, the class Person can here be said to be *partitioned* into two subclasses: Male Person and Female Person. Person could also be divided into North American Person, Asian Person, European Person, or something else. This collection of subsets would define a different GeneralizationSet that would associate with three other Generalization relationships. All three would have Person as the general Classifier; only the specific classifiers would differ (i.e., North American Person, Asian Person, and European Person).

Attributes

- isCovering : Boolean
  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. For example, Person could have two Generalization relationships each with a different specific Classifier: Male Person and Female Person. This GeneralizationSet would be covering because every instance of Person would be an instance of Male Person or Female Person. In contrast, Person could have a three Generalization relationship involving three specific Classifiers: North American Person, Asian Person, and European Person. This GeneralizationSet would not be covering because there are instances of Person for which these three specific Classifiers do not apply. The first example, then, could be read: any Person would be specialized as either being a Male Person or a Female Person— and *nothing else*; the second could be read: any Person would be specialized as being North American Person, Asian Person, European Person, or something else. Default value is *false*.

- isDisjoint : Boolean
  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. For example, Person could have two Generalization relationships, each with the different specific Classifier: Manager or Staff. This would be disjoint because every instance of Person must either be a Manager or Staff. In contrast, Person could have two Generalization relationships involving two specific (and non- covering) Classifiers: Sales Person and Manager. This GeneralizationSet would not be disjoint because there are instances of Person that can be a Sales Person *and* a Manager. Default value is *false*.

Associations

- generalization : Generalization [*]
  Designates the instances of Generalization that are members of a given GeneralizationSet (see constraint [1] below).
- powertype : Classifier [0..1]
  Designates the Classifier that is defined as the power type for the associated GeneralizationSet (see constraint [2] below).

Constraints

[1] Every Generalization associated with a particular GeneralizationSet must have the same general Classifier.

```
generalization->collect(g | g.general)->asSet()->size() <= 1
```

[2] 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.

Semantics

The generalizationSet association designates the collection of subsets to which the Generalization link belongs. All of the Generalization links that share a given general Classifier are divided into subsets (e.g., partitions or overlapping subset groups) using the generalizationSet association. Each collection of subsets represents an orthogonal dimension of specialization of the general Classifier.

As mentioned above, in essence, a power type is a class whose instances are subclasses of another class. Power types, then, are metaclasses with an extra twist: the instances can also be subclasses. The powertype association relates a classifier to the instances of that classifier, which are the specific classifiers identified for a GeneralizationSet. For example, the Bank Account Type classifier could associate with a GeneralizationSet that has Generalizations with specific classifiers of Checking Account and Savings Account. Here, then, Checking Account and Savings Account are instances of Bank Account Type. Furthermore, if the Generalization relationship has a general classifier of Bank Account, then Checking Account and Savings Account are also subclasses of Bank Account. Therefore, Checking Account and Savings Account are both instances of Bank Account Type and subclasses of Bank Account. (For more explanation and examples see “Examples” on page 78.)

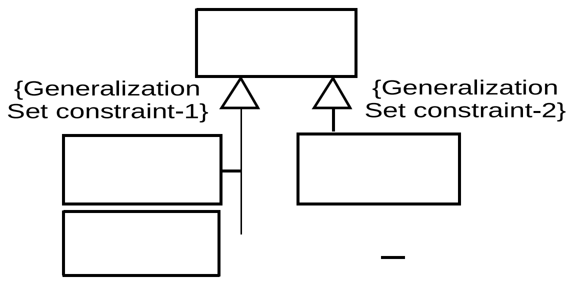
Notation

The notation to express the grouping of Generalizations into GeneralizationSets was presented in the Notation sub clause of Generalization, above. To indicate whether or not a generalization set is covering and disjoint, each set should be labeled with one of the constraints indicated below.

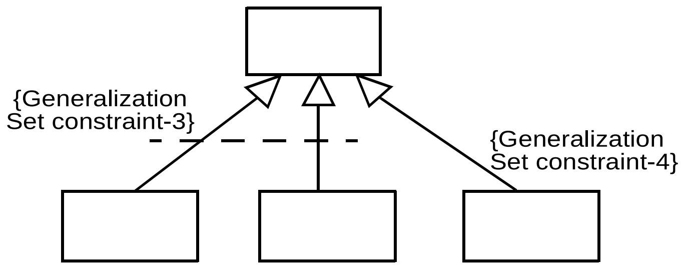
{complete, disjoint} -	Indicates the generalization set is covering and its specific Classifiers have no common instances.
{incomplete, disjoint} -	Indicates the generalization set is not covering and its specific Classifiers have no common instances*.
{complete, overlapping} -	Indicates the generalization 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.
* default is {incomplete, disjoint}	

Figure 7.45 - Generalization set constraint notation

Graphically, the GeneralizationSet constraints are placed next to the sets, whether the common arrowhead notation is employed of the dashed line, as illustrated below..



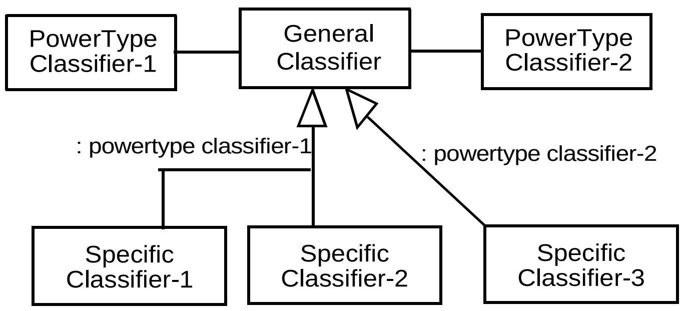
(a) GeneralizationSet constraint when sharing common generalization arrowhead.



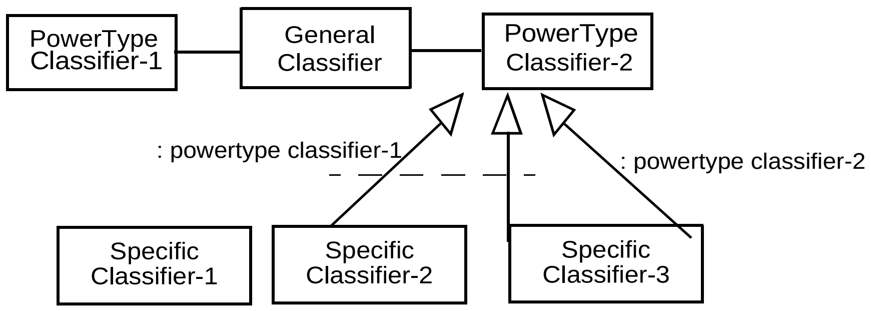
(b) GeneralizationSet constraint using dashed-line notation.

Figure 7.46 - GeneralizationSet constraint notation

Power type specification is indicated by placing the name of the powertype Classifier—preceded by a colon—next to the GeneralizationSet graphically containing the specific classifiers that are the instances of the power type. The illustration below indicates how this would appear for both the “shared arrowhead” and the “dashed-line” notation for GeneralizationSets.



(a) Power type specification when sharing common generalization arrowhead



(b) Power type specification using dashed-line notation

Figure 7.47 - Power type notation

Examples

In the illustration below, the Person class can be specialized as either a Female Person or a Male Person. Because this GeneralizationSet is partitioned (i.e., is constrained to be complete and disjoint), each instance of Person must *either* be a Female Person or a Male Person; that is, it must 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 Female Person or a Male Person.) Furthermore, a Person object can be specialized as an Employee. The generalization set here is expressed as {incomplete, disjoint}, which means that instances of Persons can be subset as Employees or some other unnamed collection that consists of all non-Employee instances. In other words, Persons can *either* be an Employee or in the complement of Employee, and not both. Taken together, the diagram indicates that a Person may be 1) either a Male Person or Female Person, *and* 2) an Employee or not. When expressed in this manner, it is possible to partition the instances of a classifier using a disjunctive normal form (DNF).

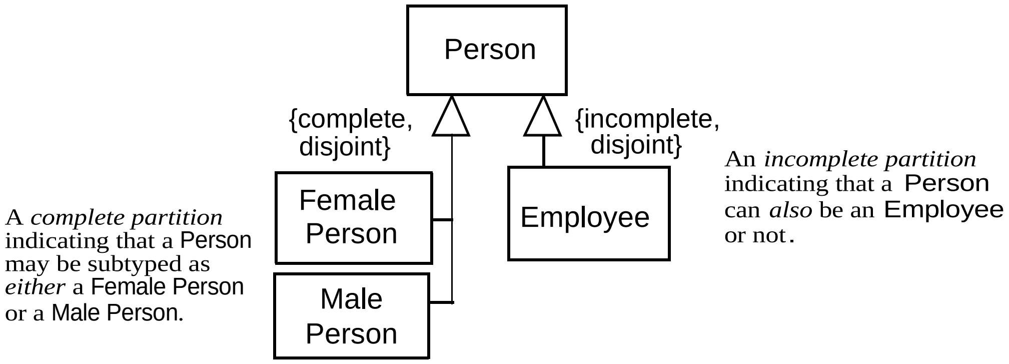

Person
{complete, disjoint}
{incomplete, disjoint}
Female Person
Male Person
Employee
A *complete partition* indicating that a Person may be subtyped as *either* a Female Person or a Male Person.
An *incomplete partition* indicating that a Person can *also* be an Employee or not.

Figure 7.48 - Multiple ways of dividing subtypes (generalization sets) and constraint examples

Grouping the objects in our world by categories, or classes, is an important technique for organizations. For instance, one of the ways botanists organize trees is by species. In this way, each tree we see can 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. Instances of Tree Species, such as sugar maple and apricot. 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 expresses that each Tree Species classifies zero or more instances of Tree, and each Tree is classified as exactly one Tree Species. It also 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.

Lastly, this diagram illustrates that Tree is subtyped as American Elm, Sugar Maple, Apricot, or Saguaro—or something else. Each subtype, then, can 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. At first glance, it would seem that a modeler should only use either the Tree Species class or the subclasses of Tree—since the instances of Tree Species are the same as the subclasses of tree. In other words, it *seems* redundant to represent both on the same diagram. Furthermore, having both would seem to cause potential diagram maintenance issues. For instance, if botanists got together and decided that the American elm should no longer be a species of tree, the American Elm object would then be removed as an instance of Tree Species. To maintain the integrity of our model in such a situation, the American Elm subtype of Tree must also be removed. Additionally, if a new species were added as a subtype of Tree, that new species would have to be added as an instance of Tree Species. The same kind of situation exists if the name of a tree species were changed—both the subtype of Tree and the instance of Tree Species would have to be modified accordingly.

As it turns out, this apparent redundancy is not a redundancy semantically (although it may be implemented that way). Different modeling approaches depicted above are not really all that different. In reality, the subtypes of Tree and the instances of Tree Species *are* the same objects. In other words, the subtypes of Tree are instances of Tree Species. Furthermore, the instances of Tree Species are the subtypes of Tree. The fact that an instance of Tree Species is called sugar maple and a subtype of Tree is called Sugar Maple is no coincidence. The sugar maple instance and Sugar Maple subtype are the same object. The instances of Tree Species are—as the name implies—types of trees. The subtypes of Tree are—by definition—types of trees. While Tree may be divided into various collections of subsets (based on size or age, for example), in this example it is divided on the basis of species. Therefore, the integrity issue mentioned above is not really an issue here. Deleting the American Elm subtype from the collection of Tree subtypes does not require also deleting the corresponding Tree Species instance, because the American Elm subtype and the corresponding Tree Species instance are the same object.

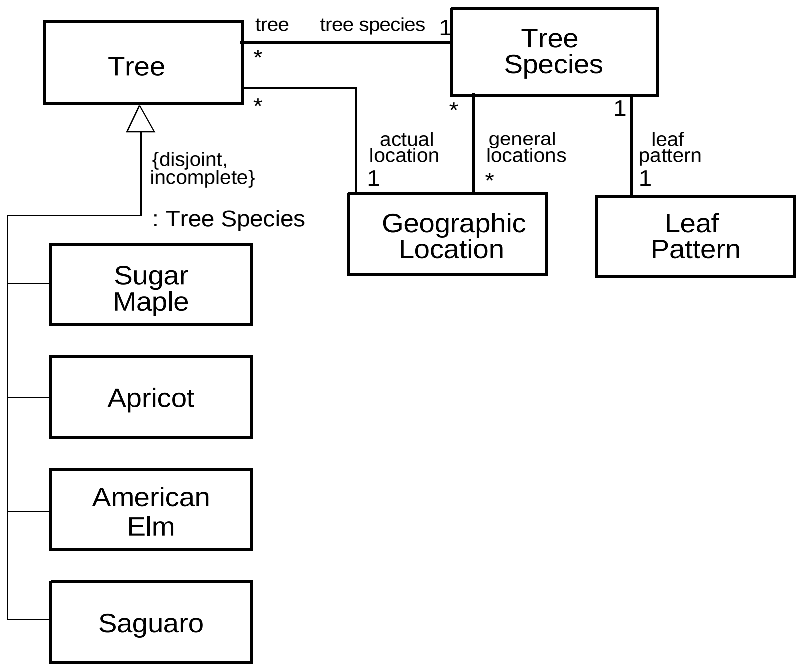
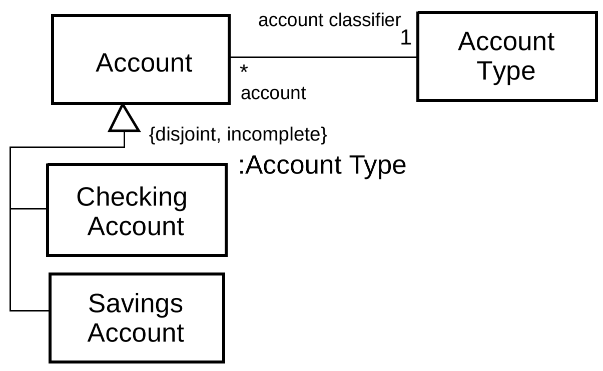


Figure 7.49 - Power type example and notation

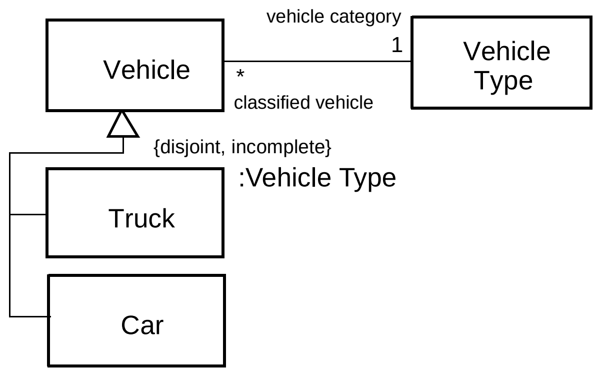
As established above, the instances of Classifiers can also be Classifiers. (This is the stuff that metamodels are made of.) These same instances, however, can also be specific classifiers (i.e., subclasses) of another classifier. When this occurs, we have what is called a *power type*. Formally, a power type is a classifier whose instances are also subclasses of another classifier.

In the examples above, Tree Species is a power type on the Tree type. Therefore, the instances of Tree Species are subtypes of Tree. This concept applies to many situations within many lines of business. Figure 7.50 depicts other examples of power types. The name on the generalization set beginning with a colon indicates the power type. In other words, this name is the name of the type of which the subtypes are instances.

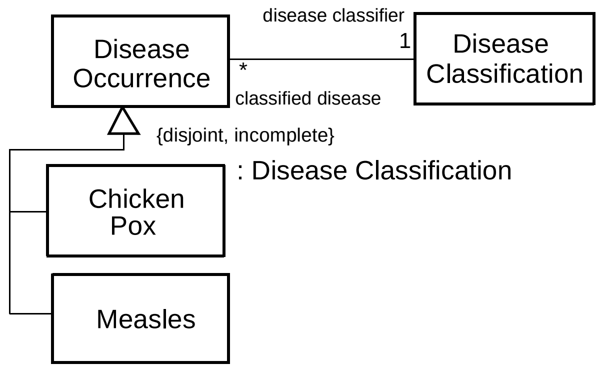
Diagram (a) in the figure below, then, can be interpreted as: each instance of Account is classified with exactly one instance of Account Type. It can also be interpreted as: the subtypes of Account are instances of Account Type. This means that each instance of Checking Account can have its own attributes (based on those defined for Checking Account and those inherited from Account), such as account number and balance. Additionally, it means that Checking Account *as an object in its own right* can have attributes, such as interest rate and maximum delay for withdrawal. (Such attributes are sometimes referred to as class variables, rather than instance variables.) The example (b) depicts a vehicle-modeling example. Here, each Vehicle can be subclassed as either a Truck or a Car or something else. Furthermore, Truck and Car are 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 classes such as Chicken Pox and Measles.



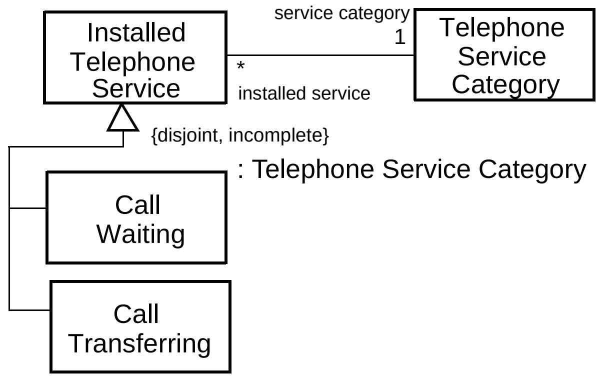
(a) Bank account/account type example



(b) Vehicle/vehicle type example



(c) Disease Occurrence/Disease Classification example



(d) Telephone service example

Figure 7.50 - Other power type examples

Labeling collections of subtypes with the power type becomes increasingly important when a type has more than one power type. The figure below is one such example. Without knowing which subtype collection contains Policy Coverage Types and which Insurance Lines, clarity is compromised. This figure depicts an even more complex situation. Here, a power type is expressed with multiple collections of subtypes. For instance, a Policy can be subtyped as either a Life, Health, Property/Casualty, or some other Insurance Line. Furthermore, a Property/Casualty policy can be further subtyped as Automobile, Equipment, Inland Marine, or some other Property/Casualty line of insurance. In other words, the subtypes in the collection labeled Insurance Line are all instances of the Insurance Line power type.

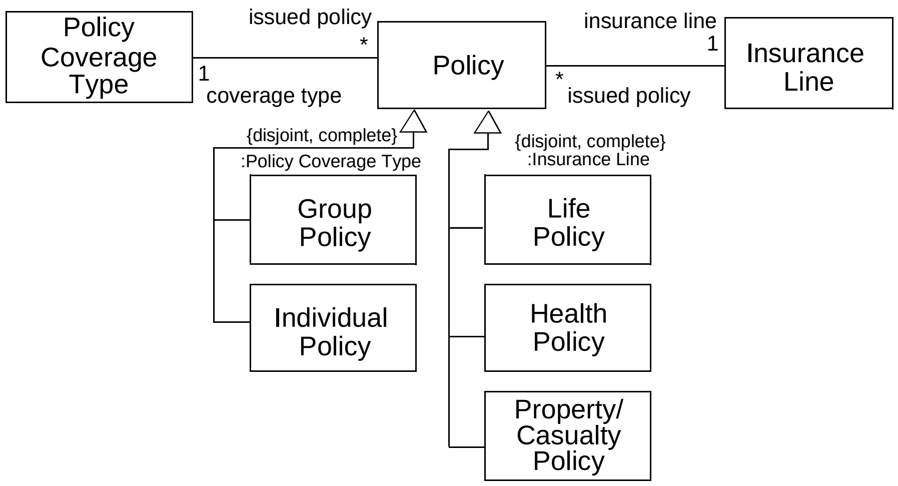


Figure 7.51 - Other power type examples

Power types are a conceptual, or analysis, notion. They express a real-world situation; however, implementing them may not be easy and efficient. To implement power types with a relational database would mean that the instances of a relation could also be relations in their own right. In object-oriented implementations, the instances of a class could also be classes. However, if the software implementation cannot directly support classes being objects and vice versa, redundant structures must be defined. In other words, unless you're programming in Smalltalk or CLOS, the designer must be aware of the integrity problem of keeping the list of power type instances in sync with the existing subclasses. Without the power type designation, implementors would not be aware that they need to consider keeping the subclasses in sync with the instances of the power type; with the power type indication, the implementor knows that a) a data integrity situation exists, and b) how to manage the integrity situation. For example, if the Life Policy instance of Insurance Line were deleted, the subclass called Life Policy can no longer exist. Or, if a new subclass of Policy were added, a new instance must also be added to the appropriate power type.

7.3.22 InstanceSpecification (from Kernel)

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

Generalizations

- "PackageableElement (from Kernel)" on page 109

Description

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

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

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

InstanceSpecification is a concrete class.

Attributes

No additional attributes

Associations

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

Constraints

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

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

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

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

Semantics

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

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

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

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

Notation

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

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

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

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

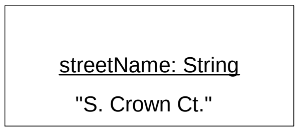


Figure 7.52 - Specification of an instance of String

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

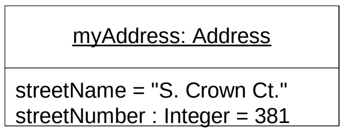


Figure 7.53 - Slots with values

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

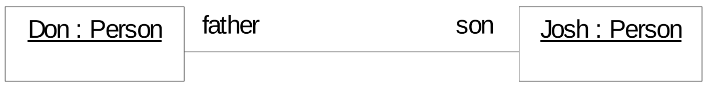


Figure 7.54 - Instance specifications representing two objects connected by a link

Presentation Options

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

7.3.23 InstanceValue (from Kernel)

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

Generalizations

- “ValueSpecification (from Kernel)” on page 137

Description

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

Attributes

No additional attributes

Associations

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

Constraints

No additional constraints

Semantics

The instance specification is the specified value.

Notation

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

7.3.24 Interface (from Interfaces)

Generalizations

- "Classifier (from Kernel, Dependencies, PowerTypes)" on page 52

Description

An interface is a kind of classifier that represents a declaration of a set of coherent public features and obligations. An interface specifies a contract; any instance of a classifier that realizes the interface must fulfill that contract. The obligations that may be associated with an interface are in the form of various kinds of constraints (such as pre- and post-conditions) or protocol specifications, which may impose ordering restrictions on interactions through the interface.

Since interfaces are declarations, they are not instantiable. Instead, an interface specification is *implemented* by an instance of an instantiable classifier, which means that the instantiable classifier presents a public facade that conforms to the interface specification. Note that a given classifier may implement more than one interface and that an interface may be implemented by a number of different classifiers (see "InterfaceRealization (from Interfaces)" on page 89).

Attributes

No additional attributes

Associations

- ownedAttribute: Property
  References all the properties owned by the Interface. (Subsets *Namespace::ownedMember* and *Classifier::feature*)
- ownedOperation: Operation
  References all the operations owned by the Interface. (Subsets *Namespace::ownedMember* and *Classifier::feature*)
- nestedClassifier: Classifier
  (References all the Classifiers owned by the Interface. (Subsets *Namespace::ownedMember*)
- redefinedInterface: Interface
  (References all the Interfaces redefined by this Interface. (Subsets *Element::redefinedElement*)

Constraints

[1] The visibility of all features owned by an interface must be public.

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

Semantics

An interface declares a set of public features and obligations that constitute a coherent service offered by a classifier. Interfaces provide a way to partition and characterize groups of properties that realizing classifier instances must possess. An interface does not specify how it is to be implemented, but merely what needs to be supported by realizing instances. That is, such instances must provide a public facade (attributes, operations, externally observable behavior) that conforms to the interface. Thus, if an interface declares an attribute, this does not necessarily mean that the realizing instance will necessarily have such an attribute in its implementation, only that it will appear so to external observers.

Because an interface is merely a declaration it is not an instantiable model element; that is, there are no instances of interfaces at run time.

The set of interfaces realized by a classifier are its *provided* interfaces, which represent the obligations that instances of that classifier have to their clients. They describe the services that the instances of that classifier offer to their clients. Interfaces may also be used to specify *required* interfaces, which are specified by a usage dependency between the classifier and the corresponding interfaces. Required interfaces specify services that a classifier needs in order to perform its function and fulfill its own obligations to its clients.

Properties owned by interfaces are abstract and imply that the conforming instance 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 classifier realizing that Interface (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 classifier.

An association between an interface and any other classifier implies that a conforming association must exist between any implementation of that interface and that other classifier. In particular, an association between interfaces implies that a conforming association must exist between implementations of the interfaces.

An interface cannot be directly instantiated. Instantiable classifiers, such as classes, must implement an interface (see "InterfaceRealization (from Interfaces)").

Notation

As a classifier, an interface may be shown using a rectangle symbol with the keyword «interface» preceding the name.

The interface realization dependency from a classifier to an interface is shown by representing the interface by a circle or *ball*, labeled with the name of the interface, attached by a solid line to the classifier that realizes this interface (see Figure 7.55).

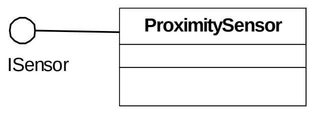


Figure 7.55 - Isensor is the provided interface of ProximitySensor

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 (see Figure 7.56).

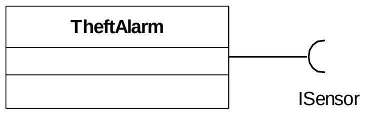


Figure 7.56 - Isensor is the required interface of TheftAlarm

Presentation Options

Alternatively, in cases where interfaces are represented using the rectangle notation, interface realization and usage dependencies are denoted with appropriate dependency arrows (see Figure 7.57). The classifier at the tail of the arrow implements the interface at the head of the arrow or uses that interface, respectively.

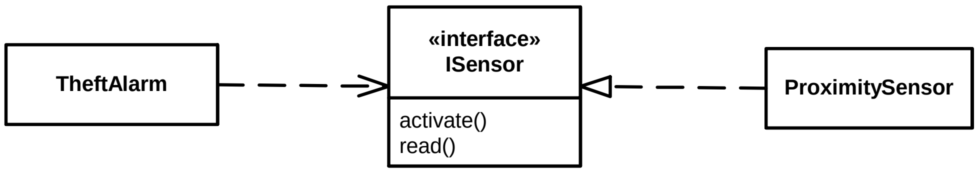


Figure 7.57 - Alternative notation for the situation depicted in Figure 7.55 and Figure 7.56

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 can be captured by associations between interfaces as shown in the example in Figure 7.58.

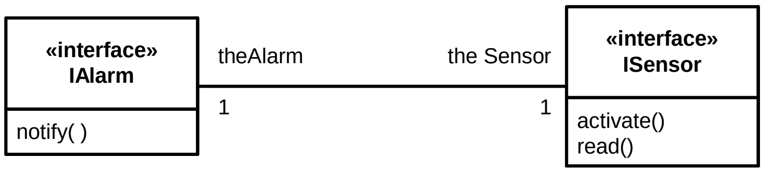


Figure 7.58 - Alarm is the required interface for any classifier implementing Isensor; conversely, Isensor is the required interface for any classifier implementing IAlarm.

Examples

The following example shows a set of associated interfaces that specify an alarm system. (These interfaces may be defined independently or as part of a collaboration.) Figure 7.59 shows the specification of three interfaces, *IAlarm*, *ISensor*, and *IBuzzer*. *IAlarm* and *Isensor* are shown as engaged in a bidirectional protocol; *IBuzzer* describes the required interface for instances of classifiers implementing *IAlarm*, as depicted by their respective associations.

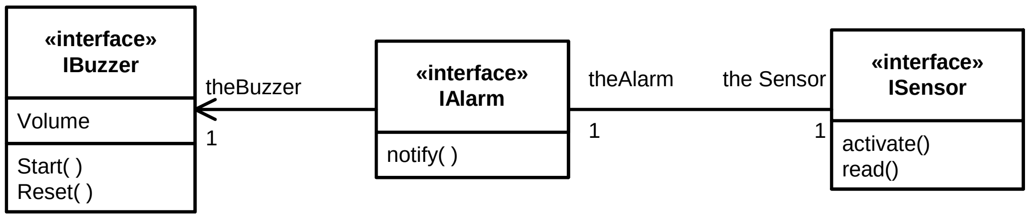


Figure 7.59 - A set of collaborating interfaces

Three classes: *DoorSensor*, *DoorAlarm*, and *DoorBell* implement the above interfaces (see Figure 7.60). These classifiers are completely decoupled. Nevertheless, instances of these classifiers are able to interact by virtue of the conforming associations declared by the associations between the interfaces that they realize.

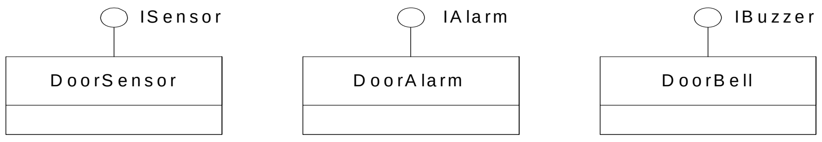


Figure 7.60 - Classifiers implementing the above interfaces

7.3.25 InterfaceRealization (from Interfaces)

Generalizations

- "Realization (from Dependencies)" on page 129

Description

An InterfaceRealization is a specialized Realization relationship between a Classifier and an Interface. This relationship signifies that the realizing classifier conforms to the contract specified by the Interface.

Attributes

No additional attributes

Associations

- contract: Interface [1]
  References the Interface specifying the conformance contract. (Subsets *Dependency::supplier*).
- implementingClassifier: BehavioredClassifier [1]
  References the BehavioredClassifier that owns this Interfacerealization (i.e., the classifier that realizes the Interface to which it points). (Subsets *Dependency::client, Element::owner.*)

Constraints

No additional constraints

Semantics

A classifier that implements an interface specifies instances that are conforming to the interface and to any of its ancestors. A classifier may implement a number of interfaces. The set of interfaces implemented by the classifier are its *provided* interfaces and signify the set of services the classifier offers to its clients. A classifier implementing an interface supports the set of features owned by the interface. In addition to supporting the features, a classifier must comply with the constraints owned by the interface.

An interface realization relationship between a classifier and an interface implies that the classifier supports the set of features owned by the interface, and any of its parent interfaces. For behavioral features, the implementing classifier will have an operation or reception for every operation or reception, respectively, defined by the interface. For properties, the realizing classifier 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 classifier, it may also be supported by the state machine of the classifier or by a pair of operations that support the retrieval of the state information and an operation that changes the state information.

Notation

See "Interface (from Interfaces)"

7.3.26 LiteralBoolean (from Kernel)

A literal Boolean is a specification of a Boolean value.

Generalizations

- "LiteralSpecification (from Kernel)" on page 92

Description

A literal Boolean contains a Boolean-valued attribute. Default value is *false*.

Attributes

- value: Boolean
  The specified Boolean value.

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

[1] The query isComputable() is redefined to be true.
```
LiteralBoolean::isComputable(): Boolean;
isComputable = true
```
[2] The query booleanValue() gives the value.
```
LiteralBoolean::booleanValue() : [Boolean];
booleanValue = value
```

Semantics

A LiteralBoolean specifies a constant Boolean value.

Notation

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

7.3.27 LiteralInteger (from Kernel)

A literal integer is a specification of an integer value.

Generalizations

- "LiteralSpecification (from Kernel)" on page 92

Description

A literal integer contains an Integer-valued attribute.

Attributes

- value: Integer
  The specified Integer value. Default value is 0.

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

[1] The query isComputable() is redefined to be true.
```
LiteralInteger::isComputable(): Boolean;
isComputable = true
```
[2] The query integerValue() gives the value.
```
LiteralInteger::integerValue() : [Integer];
integerValue = value
```

Semantics

A LiteralInteger specifies a constant Integer value.

Notation

A LiteralInteger is shown as a sequence of digits.

7.3.28 LiteralNull (from Kernel)

A literal null specifies the lack of a value.

Generalizations

- "LiteralSpecification (from Kernel)" on page 92

Description

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

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

[1] The query isComputable() is redefined to be true.
```
LiteralNull::isComputable(): Boolean;
isComputable = true
```

[2] The query isNull() returns true.

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

Semantics

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

Notation

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

7.3.29 LiteralSpecification (from Kernel)

A literal specification identifies a literal constant being modeled.

Generalizations

- “ValueSpecification (from Kernel)” on page 137

Description

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

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

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

Notation

No specific notation

7.3.30 LiteralString (from Kernel)

A literal string is a specification of a string value.

Generalizations

- “LiteralSpecification (from Kernel)” on page 92.

Description

A literal string contains a String-valued attribute.

Attributes

- value: String [0..1]
  The specified String value

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

[1] The query isComputable() is redefined to be true.
```
LiteralString::isComputable(): Boolean;
isComputable = true
```
[2] The query stringValue() gives the value.
```
LiteralString::stringValue() : [String];
stringValue = value
```

Semantics

A LiteralString specifies a constant String value.

Notation

A LiteralString is shown as a sequence of characters within double quotes.

The character set used is unspecified.

7.3.31 LiteralUnlimitedNatural (from Kernel)

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

Generalizations

- “LiteralSpecification (from Kernel)” on page 92

Description

A literal unlimited natural contains an UnlimitedNatural-valued attribute.

Attributes

- value: UnlimitedNatural
  The specified UnlimitedNatural value. Default value is 0.

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

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

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

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

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

Semantics

A LiteralUnlimitedNatural specifies a constant UnlimitedNatural value.

Notation

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

7.3.32 MultiplicityElement (from Kernel)

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

Generalizations

- "Element (from Kernel)" on page 64

Description

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

Attributes

- isOrdered : Boolean
  For a multivalued multiplicity, this attribute specifies whether the values in an instantiation of this element are sequentially ordered. Default is *false*.
- isUnique : Boolean
  For a multivalued multiplicity, this attributes specifies whether the values in an instantiation of this element are unique. Default is *true*.

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

Associations

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

Constraints

These constraints must handle situations where the upper bound may be specified by an expression not computable in the model.

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

[1] The lower bound must be a non-negative integer literal.
lowerBound()->notEmpty() **implies** lowerBound() >= 0

[2] The upper bound must be greater than or equal to the lower bound.
(upperBound()->notEmpty() **and** lowerBound()->notEmpty()) **implies** upperBound() >= lowerBound()

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

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

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

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

Additional Operations

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

[2] The query includesCardinality() checks whether the specified cardinality is valid for this multiplicity.
MultiplicityElement::includesCardinality(C : Integer) : Boolean;
pre: upperBound()->notEmpty() **and** lowerBound()->notEmpty()
includesCardinality = (lowerBound() <= C) **and** (upperBound() >= C)

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

MultiplicityElement::includesMultiplicity(M : MultiplicityElement) : Boolean;

pre: self.upperBound()->notEmpty() **and** self.lowerBound()->notEmpty()
 and M.upperBound()->notEmpty() **and** M.lowerBound()->notEmpty()

includesMultiplicity = (self.lowerBound() <= M.lowerBound()) **and** (self.upperBound() >= M.upperBound())

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

MultiplicityElement::lowerBound() : [Integer];

lowerBound = **if** lowerValue->isEmpty() **then** 1 **else** lowerValue.integerValue() **endif**

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

MultiplicityElement::upperBound() : [UnlimitedNatural];

upperBound = **if** upperValue->isEmpty() **then** 1 **else** upperValue.unlimitedValue() **endif**

Semantics

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

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

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

If the MultiplicityElement is specified as ordered (i.e., isOrdered is true), then the 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. If a MultiplicityElement is not multivalued, then the value for isUnique has no semantic effect.

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

Notation

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

The multiplicity bounds are typically shown in the format:

<lower-bound> ‘..’ *<upper-bound>*

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

If the Multiplicity is associated with an element whose notation is a text string (such as an attribute, etc.), the multiplicity string will be placed within square brackets ([ ]) as part of that text string. Figure 7.61 shows two multiplicity strings as part of attribute specifications within a class symbol.

If the Multiplicity is associated with an element that appears as a symbol (such as an association end), the multiplicity string is displayed without square brackets and may be placed near the symbol for the element. Figure 7.62 shows two multiplicity strings as part of the specification of two association ends.

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

Presentation Options

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

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

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

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

Examples

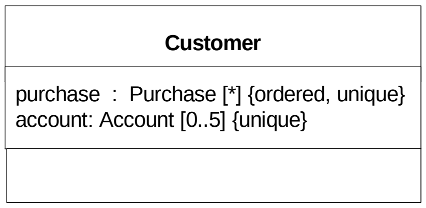

Customer
purchase : Purchase [*] {ordered, unique}
account: Account [0..5] {unique}

Figure 7.61 - Multiplicity within a textual specification

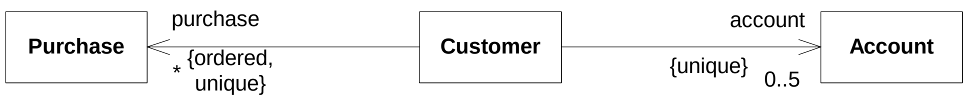


Figure 7.62 - Multiplicity as an adornment to a symbol

7.3.33 NamedElement (from Kernel, Dependencies)

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

Generalizations

- "Element (from Kernel)" on page 64

Description

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

Attributes

- name: String [0..1]
  The name of the NamedElement.
- / qualifiedName: String [0..1]
  A name 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. This is a derived attribute.
- visibility: VisibilityKind [0..1]
  Determines where the NamedElement appears within different Namespaces within the overall model, and its accessibility..

Package Dependencies

- clientDependency: Dependency[*]
  Indicates the dependencies that reference the client.

Associations

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

Constraints

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

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

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

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

[3] If a NamedElement is not owned by a Namespace, it does not have a visibility.

```
namespace->isEmpty() implies visibility->isEmpty()
```

Additional Operations

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

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

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

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

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

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

Semantics

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

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

Notation

No additional notation

7.3.34 Namespace (from Kernel)

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

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 98

Description

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

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

A namespace has the ability to import either individual members or all members of a package, thereby making it possible to refer to those named elements without qualification in the importing namespace. In the case of conflicts, it is necessary to use qualified names or aliases to disambiguate the referenced elements.

Attributes

No additional attributes

Associations

- elementImport: ElementImport [*]
  References the ElementImports owned by the Namespace. Subsets *Element::ownedElement*
- / importedMember: PackageableElement [*]
  References the PackageableElements that are members of this Namespace as a result of either PackageImports or ElementImports. Subsets *Namespace::member*
- / member: NamedElement [*]
  A collection of NamedElements identifiable within the Namespace, either by being owned or by being introduced by importing or inheritance. This is a derived union.
- / ownedMember: NamedElement [*]
  A collection of NamedElements owned by the Namespace. Subsets *Element::ownedElement* and *Namespace::member.* This is a derived union.
- ownedRule: Constraint[*]
  Specifies a set of Constraints owned by this Namespace. Subsets *Namespace::ownedMember*
- packageImport: PackageImport [*]
  References the PackageImports owned by the Namespace. Subsets *Element::ownedElement*

Constraints

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

```
membersAreDistinguishable()
```

[2] The importedMember property is derived from the ElementImports and the PackageImports.

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

Additional Operations

[1] The query getNamesOfMember() gives a set of all of the names that a member would have in a Namespace. In general a member can have multiple names in a Namespace if it is imported more than once with different aliases. The query takes account of importing. It gives back the set of names that an element would have in an importing namespace, either because it is owned; or if not owned, then imported individually; or if not individually, then from a package.

```
Namespace::getNamesOfMember(element: NamedElement): Set(String);
getNamesOfMember =
    if self.ownedMember ->includes(element)
        then Set{}->include(element.name)
    else let elementImports: ElementImport = self.elementImport->select(ei | ei.importedElement = element) in
```

```
        if elementImports->notEmpty()
            then elementImports->collect(el | el.getName())
        else
            self.packageImport->select(pi | pi.importedPackage.visibleMembers()->includes(element))->
                collect(pi | pi.importedPackage.getNamesOfMember(element))
        endif
    endif
```

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

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

[3] The query importMembers() defines which of a set of PackageableElements are actually imported into the namespace. This excludes hidden ones, i.e., those that have names that conflict with names of owned members, and also excludes elements that would have the same name when imported.

```
Namespace::importMembers(imps: Set(PackageableElement)): Set(PackageableElement);
importMembers = self.excludeCollisions(imps)->select(imp | self.ownedMember->forAll(mem |
mem.imp.isDistinguishableFrom(mem, self)))
```

[4] The query excludeCollisions() excludes from a set of PackageableElements any that would not be distinguishable from each other in this namespace.

```
Namespace::excludeCollisions(imps: Set(PackageableElements)): Set(PackageableElements);
excludeCollisions = imps->reject(imp1 | imps.exists(imp2 | not imp1.isDistinguishableFrom(imp2, self)))
```

Semantics

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

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

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

Notation

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

7.3.35 OpaqueExpression (from Kernel)

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

Generalizations

- "ValueSpecification (from Kernel)" on page 137

Description

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

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

Attributes

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

Associations

No additional associations

Constraints

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

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

Additional Operations

These operations are not defined within the specification of UML. They should be defined within an implementation that implements constraints so that constraints that use these operations can be evaluated.

[1] The query value() gives an integer value for an expression intended to produce one.

```
Expression::value(): Integer;
pre: self.isIntegral()
```

[2] The query isIntegral() tells whether an expression is intended to produce an integer.

```
Expression::isIntegral(): Boolean;
```

[3] The query isPositive() tells whether an integer expression has a positive value.

```
Expression::isPositive(): Boolean;
pre: self.isIntegral()
```

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

```
Expression::isNonNegative(): Boolean;
pre: self.isIntegral()
```

Semantics

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

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

Notation

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

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

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

Style Guidelines

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

Examples

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

7.3.36 Operation (from Kernel, Interfaces)

An operation is a behavioral feature of a classifier that specifies the name, type, parameters, and constraints for invoking an associated behavior.

Generalizations

- "BehavioralFeature (from Kernel)" on page 48

Description

An operation is a behavioral feature of a classifier that specifies the name, type, parameters, and constraints for invoking an associated behavior.

Attributes

- /isOrdered : Boolean
  Specifies whether the return parameter is ordered or not, if present. This is derived.

- isQuery : Boolean
  Specifies whether an execution of the BehavioralFeature leaves the state of the system unchanged (isQuery=true) or whether side effects may occur (isQuery=false). The default value is false.
- /isUnique : Boolean
  Specifies whether the return parameter is unique or not, if present. This is derived.
- /lower : Integer[0..1]
  Specifies the lower multiplicity of the return parameter, if present. This is derived.
- /upper : UnlimitedNatural[0..1]
  Specifies the upper multiplicity of the return parameter, if present. This is derived.

Associations

- class : Class [0..1]
  The class that owns this operation. Subsets *RedefinableElement::redefinitionContext*, *NamedElement::namespace* and *Feature::featuringClassifier*
- bodyCondition: Constraint[0..1]
  An optional Constraint on the result values of an invocation of this Operation. Subsets *Namespace::ownedRule*
- ownedParameter: Parameter[*] {ordered}
  Specifies the parameters owned by this Operation. Redefines *BehavioralFeature::ownedParameter.*
- postcondition: Constraint[*]
  An optional set of Constraints specifying the state of the system when the Operation is completed. Subsets *Namespace::ownedRule*.
- precondition: Constraint[*]
  An optional set of Constraints on the state of the system when the Operation is invoked. Subsets *Namespace::ownedRule*
- raisedException: Type[*]
  References the Types representing exceptions that may be raised during an invocation of this operation. Redefines *Basic::Operation.raisedException* and *BehavioralFeature::raisedException*.
- redefinedOperation: Operation[*]
  References the Operations that are redefined by this Operation. Subsets *RedefinableElement::redefinedElement*
- /type: Type[0..1]
  Specifies the return result of the operation, if present. This is a derived value.

Package Interfaces

- interface: Interface [0..1]
  The Interface that owns this Operation. (Subsets *RedefinableElement::redefinitionContext, NamedElement::namespace* and *Feature::featuringClassifier*)

Constraints

[1] An operation can have at most one return parameter (i.e., an owned parameter with the direction set to 'return').

```
ownedParameter->select(par | par.direction = #return)->size() <= 1
```

[2] If this operation has a return parameter, isOrdered equals the value of isOrdered for that parameter; otherwise, isOrdered is false.

isOrdered = **if** returnResult()->notEmpty() **then** returnResult()->any().isOrdered **else** false **endif**

[3] If this operation has a return parameter, isUnique equals the value of isUnique for that parameter; otherwise, isUnique is true.

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

[4] If this operation has a return parameter, lower equals the value of lower for that parameter; otherwise, lower is not defined.

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

[5] If this operation has a return parameter, upper equals the value of upper for that parameter; otherwise, upper is not defined.

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

[6] If this operation has a return parameter, type equals the value of type for that parameter; otherwise, type is not defined.

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

[7] A bodyCondition can only be specified for a query operation.

bodyCondition->notEmpty() **implies** isQuery

Additional Operations

[1] The query isConsistentWith() specifies, for any two Operations in a context in which redefinition is possible, whether redefinition would be logically consistent. A redefining operation is consistent with a redefined operation if it has the same number of owned parameters, and the type of each owned parameter conforms to the type of the corresponding redefined parameter.

A redefining operation is consistent with a redefined operation if it has the same number of formal parameters, the same number of return results, and the type of each formal parameter and return result conforms to the type of the corresponding redefined parameter or return result.

```
Operation::isConsistentWith(redefinee: RedefinableElement): Boolean;
pre: redefinee.isRedefinitionContextValid(self)
isConsistentWith = (redefinee.oclIsKindOf(Operation) and
                let op: Operation = redefinee.oclAsType(Operation) in
                self.ownedParameter.size() = op.ownedParameter.size() and
                forAll(i | op.ownedParameter[i].type.conformsTo(self.ownedParameter[i].type))
                )
```

[2] The query returnResult() returns the set containing the return parameter of the Operation if one exists; otherwise, it returns an empty set.

```
Operation::returnResult() : Set(Parameter);
returnResult = ownedParameter->select (par | par.direction = #return)
```

Semantics

An operation is invoked on an instance of the classifier for which the operation is a feature.

The preconditions for an operation define conditions that must be true when the operation is invoked. These preconditions may be assumed by an implementation of this operation.

The postconditions for an operation define conditions that will be true when the invocation of the operation completes successfully, assuming the preconditions were satisfied. These postconditions must be satisfied by any implementation of the operation.

The bodyCondition for an operation constrains the return result. The bodyCondition differs from postconditions in that the bodyCondition may be overridden when an operation is redefined, whereas postconditions can only be added during redefinition.

An operation may raise an exception during its invocation. When an exception is raised, it should not be assumed that the postconditions or bodyCondition of the operation are satisfied.

An operation may be redefined in a specialization of the featured classifier. This redefinition may specialize the types of the owned parameters, add new preconditions or postconditions, add new raised exceptions, or otherwise refine the specification of the operation.

Each operation states whether or not its application will modify the state of the instance or any other element in the model (isQuery).

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

Semantic Variation Points

The behavior of an invocation of an operation when a precondition is not satisfied is a semantic variation point. When operations are redefined in a specialization, rules regarding invariance, covariance, or contravariance of types and preconditions determine whether the specialized classifier is substitutable for its more general parent. Such rules constitute semantic variation points with respect to redefinition of operations.

Notation

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

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

where:

- *<visibility>* is the visibility of the operation (See "VisibilityKind (from Kernel)" on page 138).

  <visibility> ::= '+' | '-' | '#' | '~'
- *<name>* is the name of the operation.
- *<return-type>* is the type of the return result parameter if the operation has one defined.
- *<multiplicity>* is the multiplicity of the return type. (See "MultiplicityElement (from Kernel)" on page 94).
- *<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 (from Kernel)” on page 94).
- <*default*> is an expression that defines the value specification for the default value of the parameter.
- <*parm-property*> indicates additional property values that apply to the parameter.

Presentation Options

The parameter list can be suppressed. The return result of the operation can be expressed as a return parameter, or as the type of the operation. For example:

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

means the same thing as

```
toString() : String
```

Style Guidelines

An operation name typically begins with a lowercase letter.

Examples

display ()

-hide ()

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

+toString (): String

7.3.37 Package (from Kernel)

A package is used to group elements, and provides a namespace for the grouped elements.

Generalizations

- “Namespace (from Kernel)” on page 99
- “PackageableElement (from Kernel)” on page 109

Description

A package is a namespace for its members, and may contain other packages. Only packageable elements can be owned members of a package. By virtue of being a namespace, a package can import either individual members of other packages, or all the members of other packages.

In addition a package can be merged with other packages.

Attributes

No additional attributes

Associations

- /nestedPackage: Package [*]
  References the owned members that are Packages. Subsets *Package::packagedElement*
- /packagedElement: PackageableElement [*]
  Specifies the packageable elements that are owned by this Package. Subsets *Namespace::ownedMember.*
- /ownedType: Type [*]
  References the packaged elements that are Types. Subsets *Package::packagedElement*
- packageMerge: Package [*]
  References the PackageMerges that are owned by this Package. Subsets *Element::ownedElement*
- nestingPackage: Package [0..1]
  References the Package that owns this Package. Subsets *NamedElement::namespace*

Constraints

[1] If an element that is owned by a package has visibility, it is public or private.

self.ownedElements->forAll(e | e.visibility->notEmpty() **implies** e.visbility = #public **or** e.visibility = #private)

Additional Operations

[1] The query mustBeOwned() indicates whether elements of this type must have an owner.

Package::mustBeOwned() : Boolean

mustBeOwned = false

[2] The query visibleMembers() defines which members of a Package can be accessed outside it.

Package::visibleMembers() : Set(PackageableElement);

visibleMembers = member->select(m | self.makesVisible(m))

[3] The query makesVisible() defines whether a Package makes an element visible outside itself. Elements with no visibility and elements with public visibility are made visible.

Package::makesVisible(el: Namespaces::NamedElement) : Boolean;

pre: self.member->includes(el)

makesVisible =

```
    -- case: the element is in the package itself
    (ownedMember->includes(el)) or
    -- case: it is imported individually with public visibility
    (elementImport->select(ei|ei.importedElement = #public)->collect(ei|ei.importedElement)->includes(el)) or
    -- case: it is imported in a package with public visibility
    (packageImport->select(pi|pi.visibility = #public)->collect(pi|pi.importedPackage.member->includes(el))->notEmpty())
```

Semantics

A package is a namespace and is also a packageable element that can be contained in other packages.

The elements that can be referred to using non-qualified names within a package are owned elements, imported elements, and elements in enclosing (outer) namespaces. Owned and imported elements may each have a visibility that determines whether they are available outside the package.

A package owns its owned members, with the implication that if a package is removed from a model, so are the elements owned by the package.

The public contents of a package are always accessible outside the package through the use of qualified names.

Notation

A package is shown as a large rectangle with a small rectangle (a “tab”) attached to the left side of the top of the large rectangle. The members of the package may be shown within the large rectangle. Members may also be shown by branching lines to member elements, drawn outside the package. A plus sign (+) within a circle is drawn at the end attached to the namespace (package).

- If the members of the package are not shown within the large rectangle, then the name of the package should be placed within the large rectangle.
- If the members of the package are shown within the large rectangle, then the name of the package should be placed within the tab.

The visibility of a package element may be indicated by preceding the name of the element by a visibility symbol (‘+’ for public and ‘-’ for private). Package elements with defined visibility may not have protected or package visibility.

Presentation Options

A tool may show visibility by a graphic marker, such as color or font. A tool may also show visibility by selectively displaying those elements that meet a given visibility level (e.g., only public elements). A diagram showing a package with contents must not necessarily show all its contents; it may show a subset of the contained elements according to some criterion.

Elements that become available for use in an importing package through a package import or an element import may have a distinct color or be dimmed to indicate that they cannot be modified.

Examples

There are three representations of the same package Types in Figure 7.63. The one on the left just shows the package without revealing any of its members. The middle one shows some of the members within the borders of the package, and the one to the right shows some of the members using the alternative membership notation.

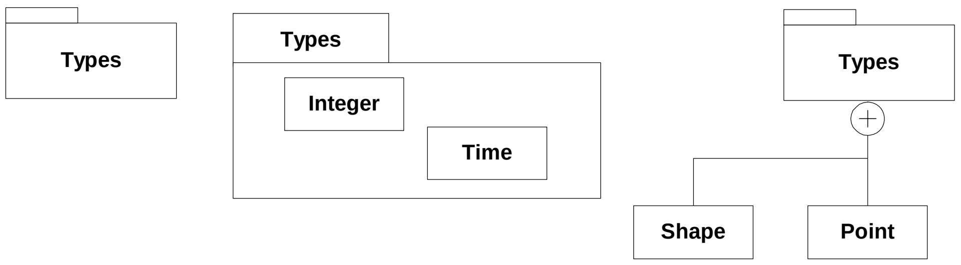


Figure 7.63 - Examples of a package with members

7.3.38 PackageableElement (from Kernel)

A packageable element indicates a named element that may be owned directly by a package.

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 98

Description

A packageable element indicates a named element that may be owned directly by a package.

Attributes

- visibility: VisibilityKind [1]
  Indicates that packageable elements must always have a visibility (i.e., visibility is not optional). Redefines N*amedElement::visibility*. Default value is *false*.

Associations

No additional associations

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

7.3.39 PackageImport (from Kernel)

A package import is a relationship that allows the use of unqualified names to refer to package members from other namespaces.

Generalizations

- "DirectedRelationship (from Kernel)" on page 63

Description

A package import is defined as a directed relationship that identifies a package whose members are to be imported by a namespace.

Attributes

- visibility: VisibilityKind
  Specifies the visibility of the imported PackageableElements within the importing Namespace, i.e., whether imported elements will in turn be visible to other packages that use that importingPackage as an importedPackage. If the PackageImport is public, the imported elements will be visible outside the package, while if it is private they will not. By default, the value of visibility is *public*.

Associations

- importedPackage: Package [1]
  Specifies the Package whose members are imported into a Namespace. Subsets *DirectedRelationship::target*
- importingNamespace: Namespace [1]
  Specifies the Namespace that imports the members from a Package. Subsets *DirectedRelationship::source* and *Element::owner*

Constraints

[1] The visibility of a PackageImport is either public or private.

```
self.visibility = #public or self.visibility = #private
```

Semantics

A package import is a relationship between an importing namespace and a package, indicating that the importing namespace adds the names of the members of the package to its own namespace. Conceptually, a package import is equivalent to having an element import to each individual member of the imported namespace, unless there is already a separately-defined element import.

Notation

A package import is shown using a dashed arrow with an open arrowhead from the importing namespace to the imported package. A keyword is shown near the dashed arrow to identify which kind of package import is intended. The predefined keywords are «import» for a public package import, and «access» for a private package import.

Presentation options

As an alternative to the dashed arrow, it is possible to show an element import by having a text that uniquely identifies the imported element within curly brackets either below or after the name of the namespace. The textual syntax is then:

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

Examples

In Figure 7.64, a number of package imports are shown. The elements in Types are imported to ShoppingCart, and then further imported to WebShop. However, the elements of Auxiliary are only accessed from ShoppingCart, and cannot be referenced using unqualified names from WebShop.

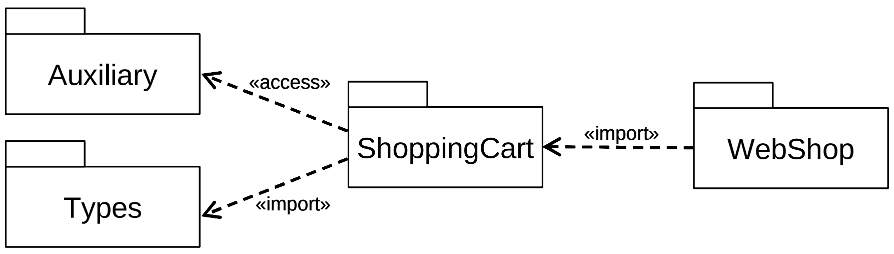


Figure 7.64 - Examples of public and private package imports

7.3.40 PackageMerge (from Kernel)

A package merge defines how the contents of one package are extended by the contents of another package.

Generalizations

- "DirectedRelationship (from Kernel)" on page 63

Description

A package merge is a directed relationship between two packages that indicates that the contents of the two packages are to be combined. It is very similar to Generalization in the sense that the source element conceptually adds the characteristics of the target element to its own characteristics resulting in an element that combines the characteristics of both.

This mechanism should be used when elements defined in different packages have the same name and are intended to represent the same concept. Most often it is used to provide different definitions of a given concept for different purposes, starting from a common base definition. A given base concept is extended in increments, with each increment defined in a separate merged package. By selecting which increments to merge, it is possible to obtain a custom definition of a concept for a specific end. Package merge is particularly useful in meta-modeling and is extensively used in the definition of the UML metamodel.

Conceptually, a package merge can be viewed as an operation that takes the contents of two packages and produces a new package that combines the contents of the packages involved in the merge. In terms of model semantics, there is no difference between a model with explicit package merges, and a model in which all the merges have been performed.

Attributes

No additional attributes

Associations

- mergedPackage: Package [1]
  References the Package that is to be merged with the receiving package of the PackageMerge. Subsets *DirectedRelationship::target*
- receivingPackage: Package [1]
  References the Package that is being extended with the contents of the merged package of the PackageMerge. Subsets *Element::owner* and *DirectedRelationship::source*

Constraints

No additional constraints

Semantics

A package merge between two packages *implies* a set of transformations, whereby the contents of the package to be merged are combined with the contents of the receiving package. In cases in which certain elements in the two packages represent the same entity, their contents are (conceptually) merged into a single resulting element according to the formal rules of package merge specified below.

As with Generalization, a package merge between two packages in a model merely implies these transformations, but the results are not themselves included in the model. Nevertheless, the receiving package and its contents are deemed to represent the result of the merge, in the same way that a subclass of a class represents the aggregation of features of all of its superclasses (and not merely the increment added by the class). Thus, within a model, any reference to a model element contained in the receiving package implies a reference to the results of the merge rather than to the increment that is physically contained in that package. This is illustrated by the example in Figure 7.65 in which package P1 and package P2 both define different increments of the same class A (identified as P1::A and P2::A respectively). Package P2 merges the contents of package P1, which implies the merging of increment P1::A into increment P2::A. Package P3 imports the contents of P2 so that it can define a subclass of A called SubA. In this case, element A in package P3 (P3::A) represents the *result* of the merge of P1::A into P2::A and not just the increment P2::A. Note that if another package were to *import* P1, then a reference to A in the importing package would represent the increment P1::A rather than the A resulting from merge.

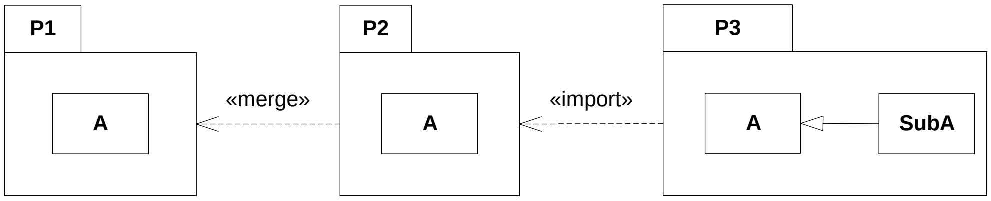


Figure 7.65 - Illustration of the meaning of package merge

To understand the rules of package merge, it is necessary to clearly distinguish between three distinct entities: the merged increment (e.g., P1::A in Figure 7.65), the receiving increment (e.g., P2::A), and the result of the merge transformations. The main difficulty comes from the fact that the receiving package and its contents represents both the operand and the results of the package merge, depending on the context in which they are considered. For example, in Figure 7.65, with respect to the package merge operation, P2 represents the increment that is an operand for the merge. However, with respect to the import operation, P2 represents the result of the merge. This dual interpretation of the same model element can be confusing, so it is useful to introduce the following terminology that aids understanding:

- *merged package* - the first operand of the merge, that is, the package that is to be merged into the receiving package (this is the package that is the target of the merge arrow in the diagrams).
- *receiving package* - the second operand of the merge, that is, the package that, conceptually, contains the results of the merge (and which is the source of the merge arrow in the diagrams). However, this term is used to refer to the package and its contents *before* the merge transformations have been performed.
- *resulting package* - the package that, conceptually, contains the results of the merge. In the model, this is, of course, the same package as the receiving package, but this particular term is used to refer to the package and its contents *after* the merge has been performed.
- *merged element* - refers to a model element that exists in the merged package.
- *receiving element* - is a model element in the receiving package. If the element has a *matching* merged element, the two are combined to produce the resulting element (see below). This term is used to refer to the element *before* the merge has been performed (i.e., the increment itself rather than the result).
- *resulting element* - is a model element in the resulting package *after* the merge was performed. For receiving elements that have a matching merged element, this is the same element as the receiving element, but in the state *after* the merge was performed. For merged elements that have no matching receiving element, this is the merged element. For receiving elements that have no matching merged element, this is the same as the receiving element.
- *element type* - refers to the type of any kind of TypedElement, such as the type of a Parameter or StructuralFeature.
- *element metatype* - is the MOF type of a model element (e.g., Classifier, Association, Feature).

This terminology is based on a conceptual view of package merge that is represented by the schematic diagram in Figure 7.66 (NB: this is not a UML diagram). The owned elements of packages A and B are all incorporated into the namespace of package B. However, it is important to emphasize that this view is merely a convenience for describing the semantics of package merge and is not reflected in the repository model, that is, the *physical* model itself is not transformed in any way by the presence of package merges.

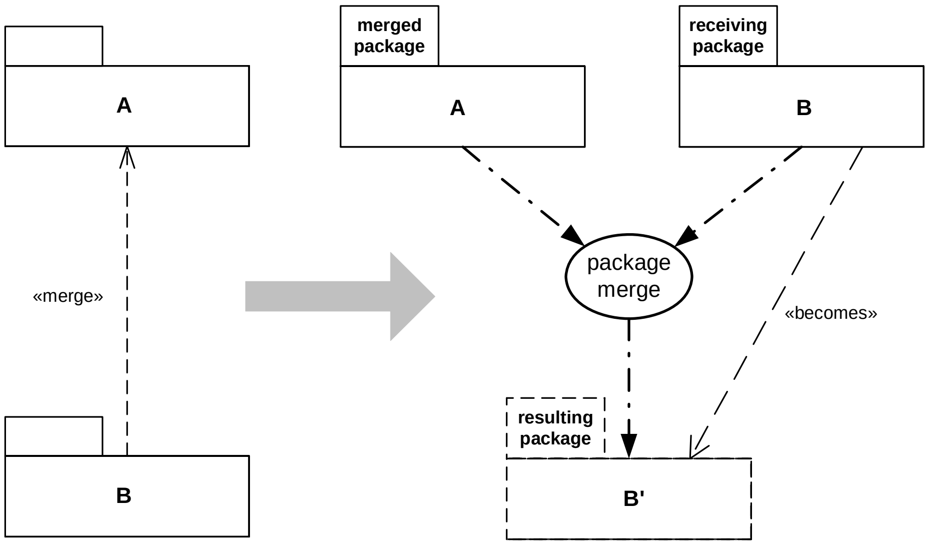


Figure 7.66 - Conceptual view of the package merge semantics

The semantics of package merge are defined by a set of constraints and transformations. The constraints specify the preconditions for a valid package merge, while the transformations describe its semantic effects (i.e., postconditions). If any constraints are violated, the package merge is ill formed and the resulting model that contains it is invalid. Different metatypes have different semantics, but the general principle is always the same: a resulting element will not be any less capable than it was prior to the merge. This means, for instance, that the resulting navigability, multiplicity, visibility, etc. of a receiving model element will not be reduced as a result of a package merge. One of the key consequences of this is that model elements in the resulting package are compatible extensions of the corresponding elements in the (unmerged) receiving package *in the same namespace*. This capability is particularly useful in defining metamodel compliance levels such that each successive level is compatible with the previous level, including their corresponding XMI representations.

In this specification, explicit merge transformations are only defined for certain general metatypes found mostly in metamodels (Packages, Classes, Associations, Properties, etc.), since the semantics of merging other kinds of metatypes (e.g., state machines, interactions) are complex and domain specific. Elements of all other kinds of metatypes are transformed according to the default rule: they are simply deep copied into the resulting package. (This rule can be superseded for specific metatypes through profiles or other kinds of language extensions.)

General package merge rules

A merged element and a receiving element *match* if they satisfy the matching rules for their metatype.

CONSTRAINTS:

1. There can be no cycles in the «merge» dependency graph.
2. A package cannot merge a package in which it is contained.

3. A package cannot merge a package that it contains.
4. A merged element whose metatype is not a kind of Package, Class, DataType, Property, Association, Operation, Constraint, Enumeration, or EnumerationLiteral cannot have a receiving element with the same name and metatype unless that receiving element is an exact copy of the merged element (i.e., they are the same).
5. A package merge is valid if and only if all the constraints required to perform the merge are satisfied.
6. Matching typed elements (e.g., Properties, Parameters) must have conforming types. For types that are classes or data types, a conforming type is either the same type or a common supertype. For all other cases, conformance means that the types must be the same.
7. A receiving element cannot have explicit references to any merged element.

TRANSFORMATIONS:

1. (*The default rule*) Merged or receiving elements for which there is no matching element are deep copied into the resulting package.
2. The result of merging two elements with matching names and metatypes that are exact copies of each other is the receiving element.
3. Matching elements are combined according to the transformation rules specific to their metatype and the results included in the resulting package.
4. All type references to typed elements that end up in the resulting package are transformed into references to the corresponding resulting typed elements (i.e., not to their respective increments).
5. For all matching elements: if both matching elements have private visibility, the resulting element will have private visibility; otherwise, the resulting element will have public visibility.
6. For all matching classifier elements: if both matching elements are abstract, the resulting element is abstract; otherwise, the resulting element is non-abstract.
7. For all matching elements: if both matching elements are not derived, the resulting element is also not derived; otherwise, the resulting element is derived.
8. For all matching multiplicity elements: the lower bound of the resulting multiplicity is the lesser of the lower bounds of the multiplicities of the matching elements.
9. For all matching multiplicity elements: the upper bound of the resulting multiplicity is the greater of the upper bounds of the multiplicities of the matching elements.
10. Any stereotypes applied to a model element in either a merged or receiving element are also applied to the corresponding resulting element.

Package rules

Elements that are a kind of Package match by name and metatype (e.g., profiles match with profiles and regular packages with regular packages).

TRANSFORMATIONS:

1. A nested package from the merged package is transformed into a nested package with the same name in the resulting package, unless the receiving package already contains a matching nested package. In the latter case, the merged nested package is recursively merged with the matching receiving nested package.

2. An element import owned by the receiving package is transformed into a corresponding element import in the resulting package. Imported elements are not merged (unless there is also a package merge to the package owning the imported element or its alias).

Class and DataType rules

Elements that are kinds of Class or DataType match by name and metatype.

TRANSFORMATIONS:

1. All properties from the merged classifier are merged with the receiving classifier to produce the resulting classifier according to the property transformation rules specified below.
2. Nested classifiers are merged recursively according to the same rules.

Property rules

Elements that are kinds of Property match by name and metatype.

CONSTRAINTS:

1. The static (or non-static) characteristic of matching properties must be the same.
2. The uniqueness characteristic of matching properties must be the same.
3. Any constraints associated with matching properties must not be conflicting.
4. Any redefinitions associated with matching properties must not be conflicting.

TRANSFORMATIONS:

1. For merged properties that do not have a matching receiving property, the resulting property is a newly created property in the resulting classifier that is the same as the merged property.
2. For merged properties that have a matching receiving property, the resulting property is a property with the same name and characteristics except where these characteristics are different. Where these characteristics are different, the resulting property characteristics are determined by application of the appropriate transformation rules.
3. For matching properties: if both properties are designated read-only, the resulting property is also designated read-only; otherwise, the resulting property is designated as not read-only.
4. For matching properties: if both properties are unordered, then the resulting property is also unordered; otherwise, the resulting property is ordered.
5. For matching properties: if neither property is designated as a subset of some derived union, then the resulting property will not be designated as a subset; otherwise, the resulting property will be designated as a subset of that derived union.
6. For matching properties: different redefinitions of matching properties are combined conjunctively.
7. For matching properties: different constraints of matching properties are combined conjunctively.
8. For matching properties: if either the merged and/or receiving elements are non-unique, the resulting element is non-unique; otherwise, the resulting element is designated as unique.
9. The resulting property type is transformed to refer to the corresponding type in the resulting package.

Association rules

Elements that are a kind of Association match by name (including if they have no name) and by their association ends where those match by name and type (i.e., the same rule as properties). These rules are in addition to regular property rules described above.

CONSTRAINTS:

1. These rules only apply to binary associations. (The default rule is used for merging n-ary associations.)
2. The receiving association end must be a composite if the matching merged association end is a composite.
3. The receiving association end must be owned by the association if the matching merged association end is owned by the association.

TRANSFORMATIONS:

1. A merge of matching associations is accomplished by merging the Association classifiers (using the merge rules for classifiers) and merging their corresponding owned end properties according to the rules for properties and association ends.
2. For matching association ends: if neither association end is navigable, then the resulting association end is also not navigable. In all other cases, the resulting association end is navigable.

Operation rules

Elements that are a kind of Operation match by name, parameter order, and parameter types, not including any return type.

CONSTRAINTS:

1. Operation parameters and types must conform to the same rules for type and multiplicity as were defined for properties.
2. The receiving operation must be a query if the matching merged operation is a query.

TRANSFORMATIONS:

1. For merged operations that do not have a matching receiving operation, the resulting operation is an operation with the same name and signature in the resulting classifier.
2. For merged operations that have a matching receiving operation, the resulting operation is the outcome of a merge of the matching merged and receiving operations, with parameter transformations performed according to the property transformations defined above.

Enumeration rules

Elements that are a kind of EnumerationLiteral match by owning enumeration and literal name.

CONSTRAINTS:

1. Matching enumeration literals must be in the same order.

TRANSFORMATIONS:

1. Non-matching enumeration literals from the merged enumeration are concatenated to the receiving enumeration.

Constraint Rules

CONSTRAINTS:

1. Constraints must be mutually non-contradictory.

TRANSFORMATIONS:

1. The constraints of the merged model elements are conjunctively added to the constraints of the matching receiving model elements.

Notation

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

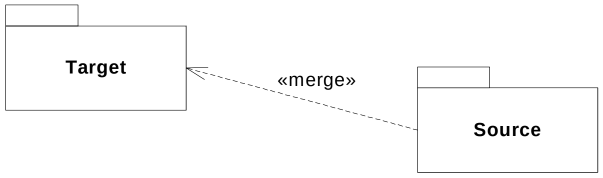


Figure 7.67 - Notation for package merge

Examples

In Figure 7.68, packages P and Q are being merged by package R, while package S merges only package Q.

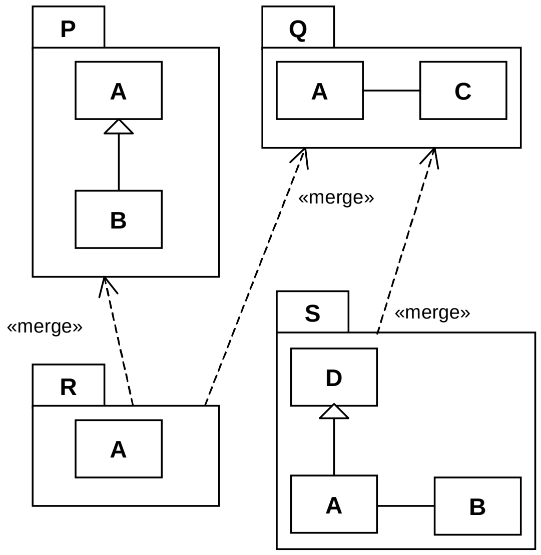


Figure 7.68 - Simple example of package merges

The transformed packages R and S are shown in Figure 7.69. The expressions in square brackets indicating which individual increments were merged into produce the final result, with the “@” character denoting the merge operator (note that these expressions are not part of the standard notation, but are included here for explanatory purposes).

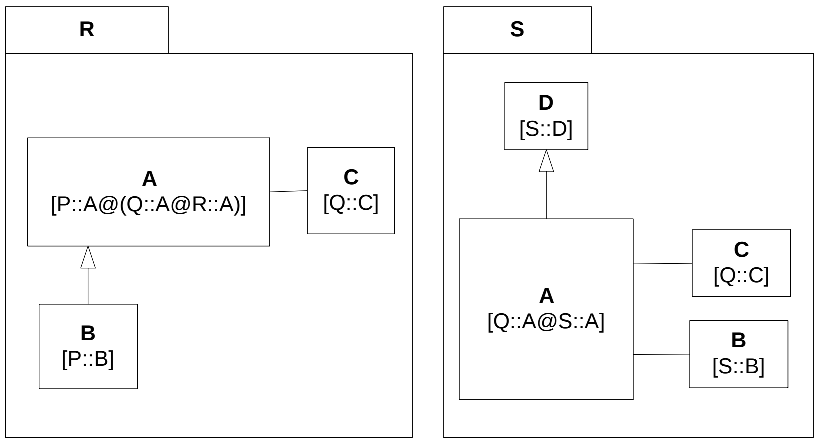


Figure 7.69 - Simple example of transformed packages following the merges in Figure 7.68

In Figure 7.70, additional package merges are introduced by having package T, which is empty prior to execution of the merge operation, merge packages R and S defined previously.

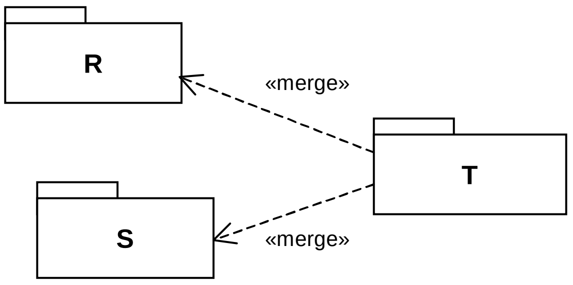


Figure 7.70 - Introducing additional package merges

In Figure 7.71, the transformed version of package T is depicted. In this package, the partial definitions of A, B, C, and D have all been brought together. Note that the types of the ends of the associations that were originally in the packages Q and S have all been updated to refer to the appropriate elements in package T.

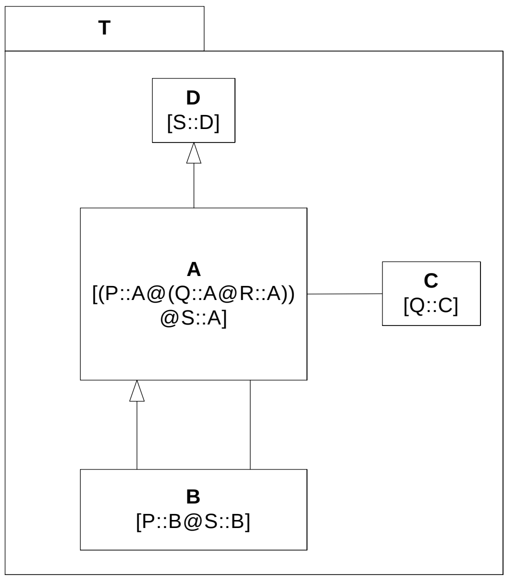


Figure 7.71 - The result of the additional package merges in Figure 7.70

7.3.41 Parameter (from Kernel)

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

Generalizations

- "MultiplicityElement (from Kernel)" on page 94.
- "TypedElement (from Kernel)" on page 136.

Description

A parameter is a specification of an argument used to pass information into or out of an invocation of a behavioral feature. It has a type, and may have a multiplicity and an optional default value.

Attributes

- / default: String [0..1]
  Specifies a String that represents a value to be used when no argument is supplied for the Parameter. This is a derived value.
- direction: ParameterDirectionKind [1]
  Indicates whether a parameter is being sent into or out of a behavioral element. The default value is *in*.

Associations

- /operation: Operation[0..1]
  References the Operation owning this parameter. Subsets *NamedElement::namespace*
- defaultValue: ValueSpecification [0..1]
  Specifies a ValueSpecification that represents a value to be used when no argument is supplied for the Parameter. Subsets *Element::ownedElement*

Constraints

No additional constraints

Semantics

A parameter specifies how arguments are passed into or out of an invocation of a behavioral feature like an operation. The type and multiplicity of a parameter restrict what values can be passed, how many, and whether the values are ordered.

If a default is specified for a parameter, then it is evaluated at invocation time and used as the argument for this parameter if and only if no argument is supplied at invocation of the behavioral feature.

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

The parameter direction specifies whether its value is passed into, out of, or both into and out of the owning behavioral feature. A single parameter may be distinguished as a return parameter. If the behavioral feature is an operation, then the type and multiplicity of this parameter is the same as the type and multiplicity of the operation itself.

Notation

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

Style Guidelines

A parameter name typically starts with a lowercase letter.

7.3.42 ParameterDirectionKind (from Kernel)

Parameter direction kind is an enumeration type that defines literals used to specify direction of parameters.

Generalizations

None

Description

ParameterDirectionKind is an enumeration of the following literal values:

- in Indicates that parameter values are passed into the behavioral element by the caller.
- inout Indicates that parameter values are passed into a behavioral element by the caller and then back out to the caller from the behavioral element.
- out Indicates that parameter values are passed from a behavioral element out to the caller.

- return Indicates that parameter values are passed as return values from a behavioral element back to the caller.

7.3.43 PrimitiveType (from Kernel)

A primitive type defines a predefined data type, without any relevant substructure (i.e., it has no parts in the context of UML). A primitive datatype may have an algebra and operations defined outside of UML, for example, mathematically.

Generalizations

- “DataType (from Kernel)” on page 60.

Description

The instances of primitive type used in UML itself include Boolean, Integer, UnlimitedNatural, and String.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

The run-time instances of a primitive type are data values. The values are in many-to-one correspondence to mathematical elements defined outside of UML (for example, the various integers).

Instances of primitive types do not have identity. If two instances have the same representation, then they are indistinguishable.

Notation

A primitive type has the keyword «primitive» above or before the name of the primitive type.

Instances of the predefined primitive types may be denoted with the same notation as provided for references to such instances (see the subtypes of “ValueSpecification (from Kernel)”).

7.3.44 Property (from Kernel, AssociationClasses)

A property is a structural feature.

A property related to a classifier by ownedAttribute represents an attribute, and it may also represent an association end. It relates an instance of the class to a value or collection of values of the type of the attribute.

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

Generalizations

- "StructuralFeature (from Kernel)" on page 133

Description

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

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

Package AssociationClasses

A property may have other properties (attributes) that serve as qualifiers.

Attributes

- aggregation: AggregationKind [1]
  Specifies the kind of aggregation that applies to the Property. The default value is *none*.
- / 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. This is a derived value.
- / isComposite: Boolean [1]
  This is a derived value, indicating whether the aggregation of the Property is composite or not.
- isDerived: Boolean [1]
  Specifies whether the Property is derived, i.e., whether its value or values can be computed from other information. The default value is *false*.
- isDerivedUnion : Boolean
  Specifies whether the property is derived as the union of all of the properties that are constrained to subset it. The default value is *false*.
- isReadOnly : Boolean
  If true, the attribute may only be read, and not written. The default value is *false*.

Associations

- association: Association [0..1]
  References the association of which this property is a member, if any.
- owningAssociation: Association [0..1]
  References the owning association of this property. Subsets *Property::association*, *NamedElement::namespace*, *Feature::featuringClassifier*, and *RedefinableElement::redefinitionContext*.
- datatype : DataType [0..1]
  The DataType that owns this Property. Subsets *NamedElement::namespace*, *Feature::featuringClassifier*, and *Property::classifier.*
- defaultValue: ValueSpecification [0..1]
  A ValueSpecification that is evaluated to give a default value for the Property when an object of the owning Classifier is instantiated. Subsets *Element::ownedElement*.

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

Package AssociationClasses

- associationEnd : Property [0..1]
  Designates the optional association end that owns a qualifier attribute. Subsets *Element::owner*
- qualifier : Property [*]
  An optional list of ordered qualifier attributes for the end. If the list is empty, then the Association is not qualified. Subsets *Element::ownedElement*

Constraints

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

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

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

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

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

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

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

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

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

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

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

```
isReadOnly implies isNavigable()
```

[7] A derived union is derived.

```
isDerivedUnion implies isDerived
```

[8] A derived union is read only.

```
isDerivedUnion implies isReadOnly
```

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

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

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

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

Additional Operations

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

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

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

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

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

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

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

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

Semantics

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

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

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

Table 7.1 - Collection types for properties

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

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

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

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

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

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

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

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

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

A property may be owned by and in the namespace of a datatype.

Package AssociationClasses

A qualifier 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 instance comprises one value for each qualifier attribute. Given a qualified object and a qualifier instance, the number of objects at the other end of the association is constrained by the declared multiplicity. In the common case in which the multiplicity is 0..1, the qualifier value is unique with respect to the qualified object, and designates at most one associated object. In the general case of multiplicity 0..*, the set of associated instances is partitioned into subsets, each selected by a given qualifier instance. 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 – The multiplicity of a qualifier is given assuming that the qualifier value is supplied. The “raw” multiplicity without the qualifier is assumed to be 0..*. This is not fully general but it is almost always adequate, as a situation in which the raw multiplicity is 1 would best be modeled without a qualifier.

Note – A qualified multiplicity whose lower bound is zero indicates that a given qualifier value may be absent, while a lower bound of 1 indicates that any possible qualifier value must be present. The latter is reasonable only for qualifiers with a finite number of values (such as enumerated values or integer ranges) that represent full tables indexed by some finite range of values.

Notation

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

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

[‘{‘ <prop-modifier > [‘,’ <prop-modifier >] ’}’]*

where:

- *<visibility>* is the visibility of the property. (See “VisibilityKind (from Kernel)” on page 138.)

  <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 (from Kernel)” on page 94.)
- *<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’ | <prop-constraint>

where:

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

Feature redefinitions can either be explicit with the use of a {redefines <x>} property string on the feature or implicit 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 must conform to the compatibility constraint on the redefinitions (e.g., the type of the feature must be a subtype of the feature's type in the more general context).

Package AssociationClasses

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 is part of the association path, not part of the classifier. The qualifier is attached to the source end of the association.

The multiplicity attached to the target end denotes the possible cardinalities of the set of target instances selected by the pairing of a source 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 each end of a single association.

A qualifier may not be suppressed.

Style Guidelines

Package AssociationClasses

The qualifier rectangle should be smaller than the attached class rectangle, although this is not always practical.

Examples

Package AssociationClasses

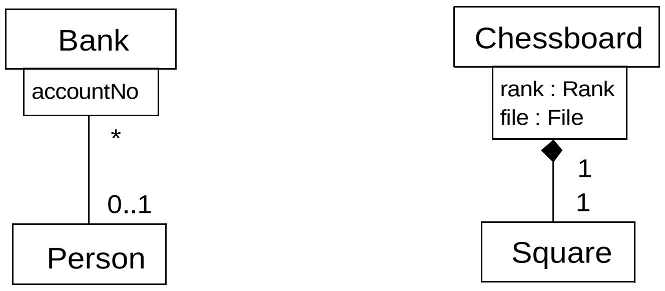


Figure 7.72 - Qualified associations

7.3.45 Realization (from Dependencies)

Generalizations

- "Abstraction (from Dependencies)" on page 38

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.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

A Realization signifies that the client set of elements are an implementation of the supplier set, 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 it is not necessarily computable.

Notation

A Realization dependency is shown as a dashed line with a triangular arrowhead at the end that corresponds to the realized element. Figure 7.73 illustrates an example in which the Business class is realized by a combination of Owner and Employee classes.

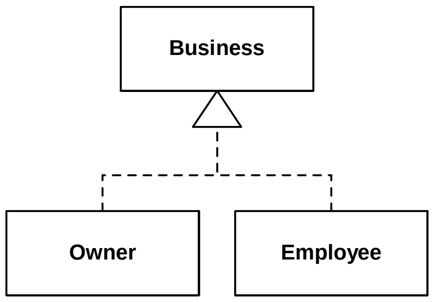


Figure 7.73 - An example of a realization dependency

7.3.46 RedefinableElement (from Kernel)

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

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 98

Description

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

Attributes

- isLeaf: Boolean
  Indicates whether it is possible to further specialize a RedefinableElement. If the value is true, then it is not possible to further specialize the RedefinableElement. Default value is *false*.

Associations

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

Constraints

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

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

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

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

Additional Operations

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

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

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

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

Semantics

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

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

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

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

Semantic Variation Points

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

Notation

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

7.3.47 Relationship (from Kernel)

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

Generalizations

- “Element (from Kernel)” on page 64

Description

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

Attributes

No additional attributes

Associations

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

Constraints

No additional constraints

Semantics

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

Notation

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

7.3.48 Slot (from Kernel)

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

Generalizations

- “Element (from Kernel)” on page 64

Description

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

Attributes

No additional attributes

Associations

- definingFeature : StructuralFeature [1]
  The structural feature that specifies the values that may be held by the slot.
- owningInstance : InstanceSpecification [1]
  The instance specification that owns this slot. Subsets *Element::owner*

- value : ValueSpecification [*]
  The value or values corresponding to the defining feature for the owning instance specification. This is an ordered association. Subsets *Element::ownedElement*

Constraints

No additional constraints

Semantics

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

Notation

See "InstanceSpecification (from Kernel)."

7.3.49 StructuralFeature (from Kernel)

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

Generalizations

- "Feature (from Kernel)" on page 70
- "MultiplicityElement (from Kernel)" on page 94
- "TypedElement (from Kernel)" on page 136

Description

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

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

Attributes

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

Associations

No additional associations

Constraints

No additional constraints

Semantics

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

Notation

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

Presentation Options

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

Changes from previous UML

The meta-attribute *targetScope*, which characterized StructuralFeature and AssociationEnd in prior UML is no longer supported.

7.3.50 Substitution (from Dependencies)

Generalizations

- "Realization (from Dependencies)" on page 129

Description

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.

Associations

- contract: Classifier [1]
  (Subsets *Dependency::target*.).
- substitutingClassifier: Classifier [1]
  (Subsets *Dependency::client*).

Attributes

None

Constraints

No additional constraints

Semantics

The substitution relationship 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. A substitution like relationship is instrumental to specify runtime substitutability for domains that do not support specialization such as certain component technologies. It requires that (1) interfaces implemented by the contract classifier are also implemented by the substituting classifier, or else the substituting classifier implements a more specialized interface type. And, (2) the any port owned by the contract classifier has a matching port (see ports) owned by the substituting classifier.

Notation

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

Examples

In the example below, a generic Window class is substituted in a particular environment by the Resizable Window class.

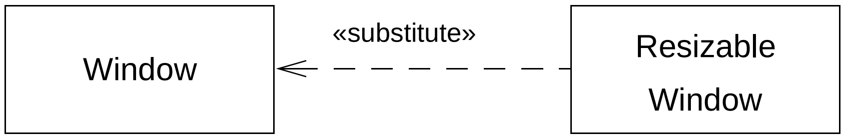


Figure 7.74 - An example of a substitute dependency

7.3.51 Type (from Kernel)

A type constrains the values represented by a typed element.

Generalizations

- "PackageableElement (from Kernel)" on page 109

Description

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

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

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

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

Semantics

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

Notation

No general notation

7.3.52 TypedElement (from Kernel)

A typed element has a type.

Generalizations

- “NamedElement (from Kernel, Dependencies)” on page 98

Description

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

Attributes

No additional attributes

Associations

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

Constraints

No additional constraints

Semantics

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

Notation

No general notation

7.3.53 Usage (from Dependencies)

Generalizations

- “Dependency (from Dependencies)” on page 62

Description

A usage is a relationship in which one element requires another element (or set of elements) for its full implementation or operation. In the metamodel, a Usage is a Dependency in which the client requires the presence of the supplier.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

The usage dependency 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.

Notation

A usage dependency is shown as a dependency with a «use» keyword attached to it.

Examples

In the example below, an Order class requires the Line Item class for its full implementation.

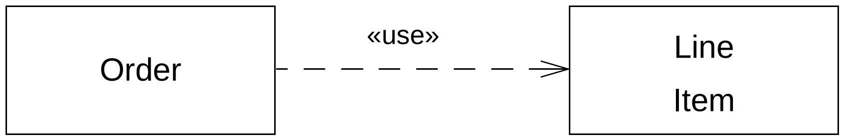


Figure 7.75 - An example of a use dependency

7.3.54 ValueSpecification (from Kernel)

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

Generalizations

- "PackageableElement (from Kernel)" on page 109
- "TypedElement (from Kernel)" on page 136

Description

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

Attributes

No additional attributes.

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

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

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

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

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

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

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

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

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

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

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

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

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

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

Semantics

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

Notation

No general notation

7.3.55 VisibilityKind (from Kernel)

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

Generalizations

None

Description

VisibilityKind is an enumeration of the following literal values:

- public
- private
- protected
- package

Additional Operations

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

```
VisibilityKind::bestVisibility(vis: Set(VisibilityKind)) : VisibilityKind;
pre: not vis->includes(#protected) and not vis->includes(#package)
bestVisibility = if vis->includes(#public) then #public else #private endif
```

Semantics

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

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

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

Notation

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

- ‘+’ public
- ‘-’ private
- ‘#’ protected
- ‘~’ package

7.4 Diagrams

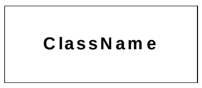
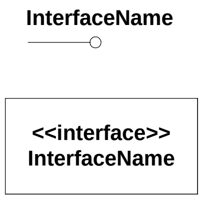
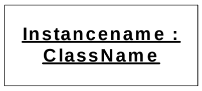
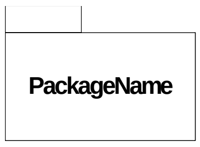
Structure diagram

This sub clause outlines the graphic elements that may be shown in structure 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.

Graphical nodes

The graphic nodes that can be included in structure diagrams are shown in Table 7.2.

Table 7.2 - Graphic nodes included in structure diagrams

NODE TYPE	NOTATION	REFERENCE
Class		See "Class (from Kernel)" on page 49.
Interface		See "Interface (from Interfaces)" on page 86.
InstanceSpecification		See "InstanceSpecification (from Kernel)" on page 82. (Note that instances of any classifier can be shown by prefixing the classifier name by the instance name followed by a colon and underlining the complete name string.)
Package		See "Package (from Kernel)" on page 107.

Graphical paths

The graphic paths that can be included in structure diagrams are shown in Table 7.3.

Table 7.3 - Graphic paths included in structure diagrams

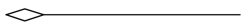
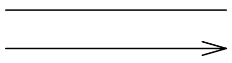
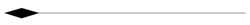
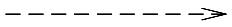
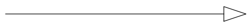
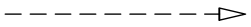
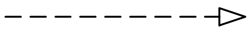
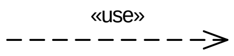
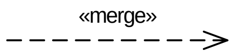
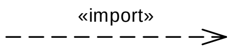
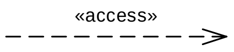
PATH TYPE	NOTATION	REFERENCE
Aggregation		See "AggregationKind (from Kernel)" on page 38.
Association		See "Association (from Kernel)" on page 39.
Composition		See "AggregationKind (from Kernel)" on page 38.
Dependency		See "Dependency (from Dependencies)" on page 62.
Generalization		See "Generalization (from Kernel, PowerTypes)" on page 71.
InterfaceRealization		See "InterfaceRealization (from Interfaces)" on page 89.
Realization		See "Realization (from Dependencies)" on page 129.
Usage	 «use»	See "Usage (from Dependencies)" on page 136.
Package Merge	 «merge»	See "PackageMerge (from Kernel)" on page 112.

Table 7.3 - Graphic paths included in structure diagrams

PATH TYPE	NOTATION	REFERENCE
PackageImport (public)		See "PackageImport (from Kernel)" on page 110.
PackageImport (private)		See "PackageImport (from Kernel)" on page 110.

Variations

Variations of structure diagrams often focus on particular structural aspects, such as relationships between packages, showing instance specifications, or relationships between classes. There are no strict boundaries between different variations; it is possible to display any element you normally display in a structure diagram in any variation.

Class diagram

The following nodes and edges are typically drawn in a class diagram:

- Association
- Aggregation
- Class
- Composition
- Dependency
- Generalization
- Interface
- InterfaceRealization
- Realization

Package diagram

The following nodes and edges are typically drawn in a package diagram:

- Dependency
- Package
- PackageExtension
- PackageImport

Object diagram

The following nodes and edges are typically drawn in an object diagram:

- InstanceSpecification
- Link (i.e., Association)

8 Components

8.1 Overview

The Components package specifies a set of constructs that can be used to define software systems of arbitrary size and complexity. In particular, the package 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 via their provided and required interfaces. 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.

Basic Components

The BasicComponents package focuses on defining a component as an executable element in a system. It defines the concept of a component as a specialized class that has an external specification in the form of one or more provided and required interfaces, and an internal implementation consisting of one or more classifiers that realize its behavior. In addition, the BasicComponents package defines specialized connectors for ‘wiring’ components together based on interface compatibility.

Packaging Components

The PackagingComponents package focuses on defining a component as a coherent group of elements as part of the development process. It extends the concept of a basic component to formalize the aspects of a component as a ‘building block’ that may own and import a (potentially large) set of model elements.

8.2 Abstract Syntax

Figure 8.1 shows the dependencies of the Component packages.

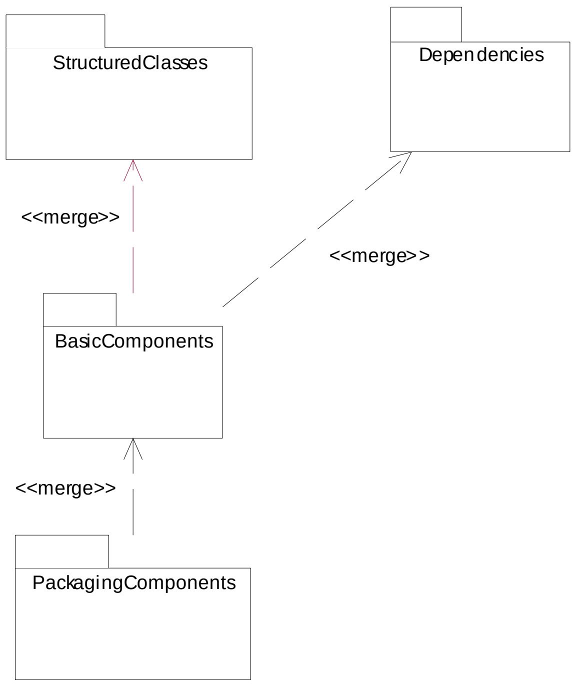


Figure 8.1 - Dependencies between packages described in this clause (transitive dependencies to Kernel and Interfaces packages are not shown).

Package BasicComponents

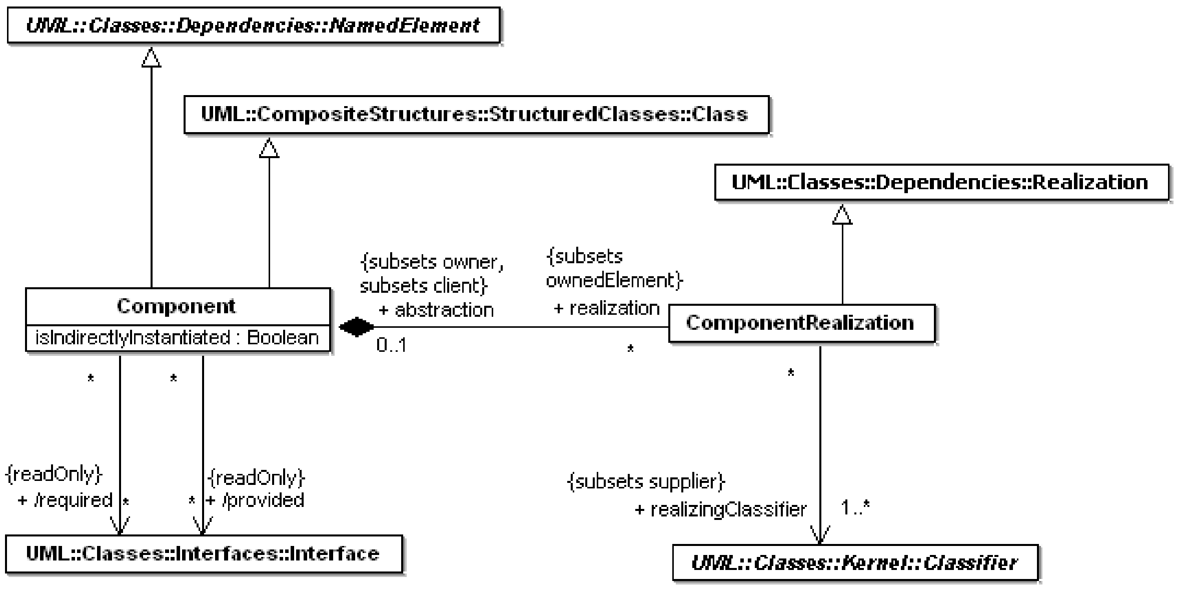

UML::Classes::Dependencies::NamedElement
UML::CompositeStructures::StructuredClasses::Class
UML::Classes::Dependencies::Realization
Component
isIndirectlyInstantiated : Boolean
{subsets owner, subsets client}
+ abstraction
0..1
{subsets ownedElement}
+ realization
*
ComponentRealization
*
*
*
{readOnly}
+ /required
*
*
{readOnly}
+ /provided
UML::Classes::Interfaces::Interface
{subsets supplier}
+ realizingClassifier
1..*
UML::Classes::Kernel::Classifier

Figure 8.2 - The metaclasses that define the basic Component construct

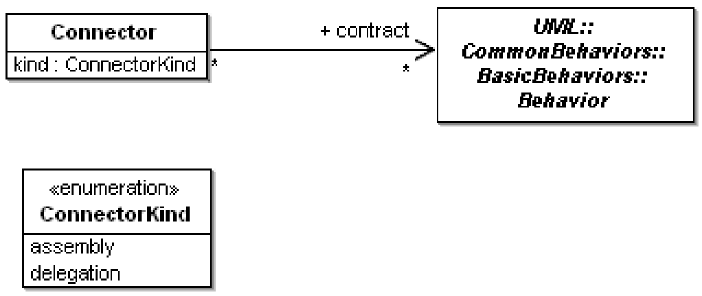


Figure 8.3 - The metaclasses that define the component wiring constructs

Package PackagingComponents

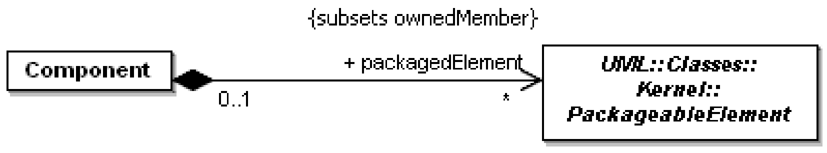


Figure 8.4 - The packaging capabilities of Components

8.3 Class Descriptions

8.3.1 Component (from BasicComponents, PackagingComponents)

A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.

A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics). One component may therefore be substituted by another only if the two are type conformant. 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 together their required and provided interfaces.

A component is modeled throughout the development life cycle and successively refined into deployment and run-time. A component may be manifest by one or more artifacts, and in turn, that artifact may be deployed to its execution environment. A deployment specification may define values that parameterize the component's execution. (See Deployment clause).

Generalizations

- "Class (from StructuredClasses)" on page 160
- "NamedElement (from Kernel, Dependencies)" on page 98

Description

BasicComponents

A component is a subtype of Class that provides for a Component having attributes and operations, and being able to participate in Associations and Generalizations. A Component may form the abstraction for a set of realizingClassifiers that realize its behavior. In addition, because a Class itself is a subtype of an EncapsulatedClassifier, a Component may optionally have an internal structure and own a set of Ports that formalize its interaction points.

A component has a number of provided and required Interfaces, that form the basis for wiring components together, either using Dependencies, or by using Connectors. A provided Interface is one that is either implemented directly by the component or one of its realizingClassifiers, or it is the type of a provided Port of the Component. A required interface is designated by a Usage Dependency from the Component or one of its realizingClassifiers, or it is the type of a required Port.

PackagingComponents

A component is extended to define the grouping aspects of packaging components. This defines the Namespace aspects of a Component through its inherited ownedMember and elementImport associations. In the namespace of a component, all model elements that are involved in or related to its definition are either owned or imported explicitly. This may include, for example, UseCases and Dependencies (e.g., mappings), Packages, Components, and Artifacts.

Attributes

Package BasicComponents

- isIndirectlyInstantiated : Boolean {default = true}
  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. Several standard stereotypes use this meta attribute (e.g., «specification», «focus», «subsystem»).

Associations

Package BasicComponents

- /provided: Interface [*]
  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. The provided interfaces association is a derived association:

```
context Component::provided derive:
        let implementedInterfaces : Set(Interface) = self.implementation->collect(impl|impl.contract),
        realizedInterfaces : Set(Interface) = RealizedInterfaces(self) ,
        realizingClassifiers : Set(Classifier) = Set{self.realizingClassifier}->union(self.allParents().realizingClassifier),
        allRealizingClassifiers : Set(Classifier) = realizingClassifiers->union(realizingClassifiers.allParents()) ,
        realizingClassifierInterfaces : Set(Interface) = allRealizingClassifiers->
                        iterate(c; rci : Set(Interface) = Set{} | rci->union(RealizedInterfaces(c))) ,
        ports : Set(Port) = self.ownedPort->union(allParents.oclAsType(Set(EncapsulatedClassifier)).ownedPort) ,
        providedByPorts : Set(Interface) = ports.provided in
                        ( (implementedInterfaces->union(realizedInterfaces)->
                        union(realizingClassifierInterfaces) )->union(providedByPorts) )->asSet()
```

- /required: Interface [*]
  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. The required interfaces association is a derived association:

```
context Component::required derive:
        let usingInterfaces : Set(Interface) = self.implementation->collect(impl|impl.contract),
        usedInterfaces : Set(Interface) = UsedInterfaces(self) ,
        realizingClassifiers : Set(Classifier) = Set{self.realizingClassifier}->union(self.allParents().realizingClassifier) ,
        allRealizingClassifiers : Set(Classifier) = realizingClassifiers->union(realizingClassifiers.allParents()) ,
        realizingClassifierInterfaces : Set(Interface) = allRealizingClassifiers->iterate(c; rci : Set(Interface) = Set{} |
                        rci->union(UsedInterfaces(c))) ,
        ports : Set(Port) = self.ownedPort->union(allParents.oclAsType(Set(EncapsulatedClassifier)).ownedPort) ,
        usedByPorts : Set(Interface) = ports.provided in
                        ( (usingInterfaces->union(usedInterfaces)->
                        union(realizingClassifierInterfaces) )->union(usedByPorts) )->asSet()
```

- realization: ComponentRealization [*]
  The set of Realizations owned by the Component. These realizations reference the Classifiers of which the Component is an abstraction (i.e., those that realize its behavior).

PackagingComponents

- packagedElement: PackageableElement [*]
  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, Use cases, Dependencies (e.g., mappings), and Artifacts. Subsets *Namespace::ownedMember.*

Constraints

No further constraints

Additional Operations

[1] Utility returning the set of realized interfaces of a component:

```
def: RealizedInterfaces : (classifier : Classifier) : Interface = (classifier.clientDependency->
    select(dependency|dependency.oclIsKindOf(Realization) and dependency.supplier.oclIsKindOf(Interface)))->
      collect(dependency|dependency.client)
```

[2] Utility returning the set of required interfaces of a component:

```
def: UsedInterfaces : (classifier : Classifier) : Interface = (classifier.supplierDependency->
   select(dependency|dependency.oclIsKindOf(Usage) and dependency.supplier.oclIsKindOf(interface)))->
     collect(dependency|dependency.supplier)
```

Semantics

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 obeys the constraints expressed by 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.

The required and provided interfaces of a component allow for the specification of structural features such as attributes and association ends, as well as behavioral features such as operations and events. 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 protocol state machine 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. Other behaviors may also be associated with interfaces or connectors to define the 'contract' between participants in a collaboration (e.g., in terms of use case, activity, or interaction specifications).

The wiring between components in a system or other context can be structurally defined by using dependencies between component interfaces (typically on structure diagrams). Optionally, a more detailed specification of the structural collaboration can be made using parts and connectors in composite structures, to specify the role or instance level collaboration between components (See Clause Composite Structures).

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. The mapping between external and internal view is by means of dependencies (on structure diagrams), or delegation connectors to internal parts (on composite structure diagrams). Again, more detailed behavior specifications such as interactions and activities may be used to detail the mapping from external to internal behavior.

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 UML Standard Elements Annex).

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.

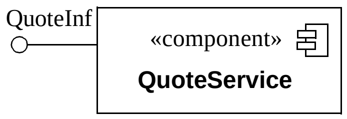


Figure 8.5 - A Component with one provided interface

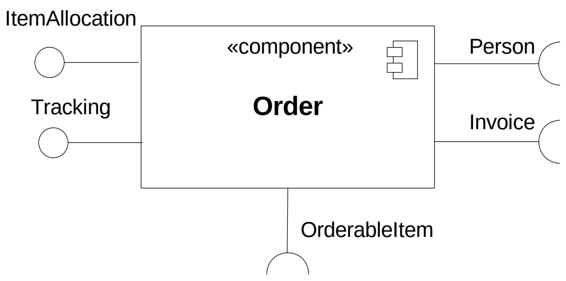


Figure 8.6 - A Component with two provided and three required interfaces

An external view of a Component is by means of Interface symbols sticking out of the Component box (external, or black-box view). Alternatively, the interfaces and/or individual operations and attributes can be listed in the compartments of a component box (for scalability, tools may offer way of listing and abbreviating component properties and behavior).

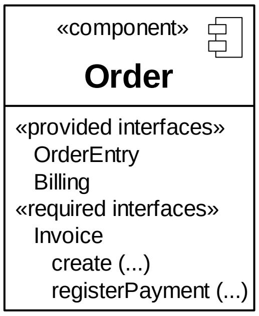


Figure 8.7 - Black box notation showing a listing of the properties of a component

For displaying the full signature of an interface of a component, the interfaces can also be displayed as typical classifier rectangles that can be expanded to show details of operations and events.

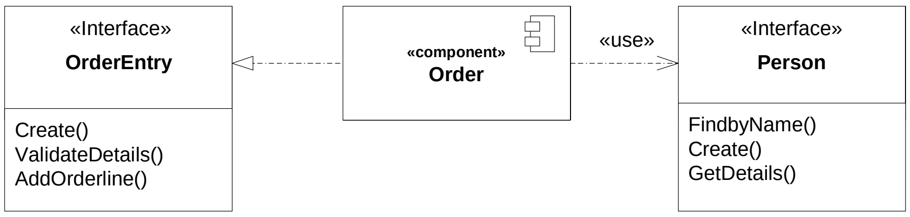


Figure 8.8 - Explicit representation of the provided and required interfaces, allowing interface details such as operation to be displayed (when desired).

An internal, or white box view of a Component is where the realizing classifiers are listed in an additional compartment. Compartments may also be used to display a listing of any parts and connectors, or any implementing artifacts.

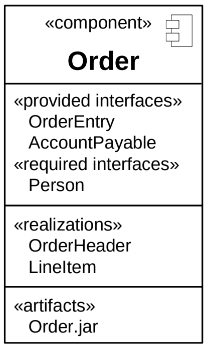


Figure 8.9 - A white-box representation of a component

The internal classifiers that realize the behavior of a component may be displayed by means of general dependencies. Alternatively, they may be nested within the component shape.

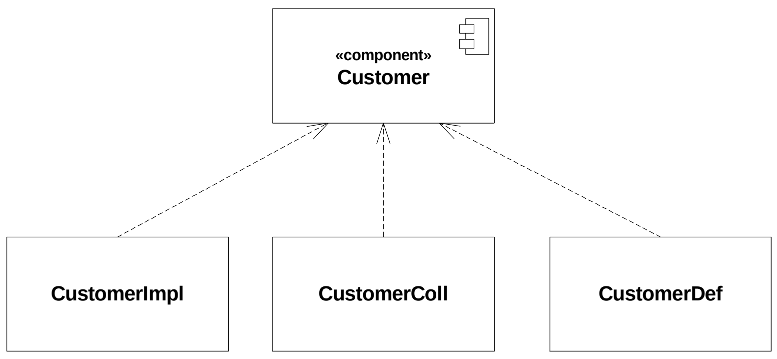


Figure 8.10 - A representation of the realization of a complex component

Alternatively, the internal classifiers that realize the behavior of a component may be displayed nested within the component shape.

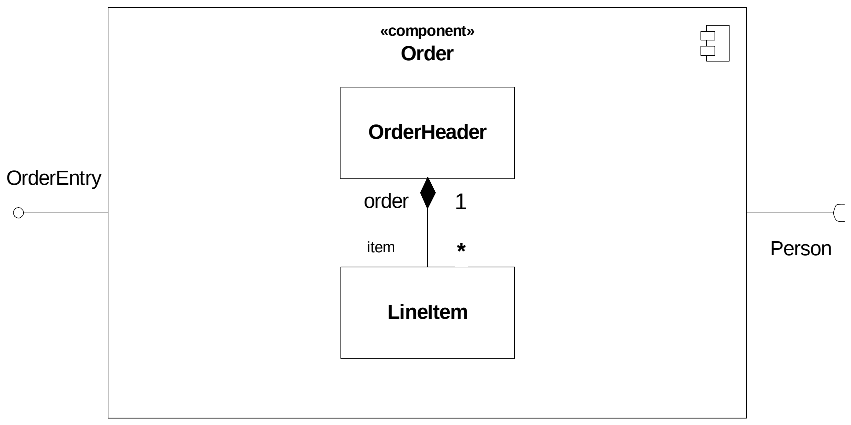


Figure 8.11 - An alternative nested representation of a complex component

If more detail is required of the role or instance level containment of a component, then an internal structure consisting of parts and connectors can be defined for that component. This allows, for example, explicit part names or connector names to be shown in situations where the same Classifier (Association) is the type of more than one Part (Connector). That is, the Classifier is instantiated more than once inside the component, playing different roles in its realization. Optionally, specific instances (InstanceSpecifications) can also be referred to as in this notation.

Interfaces that are exposed by a Component and notated on a diagram, either directly or through a port definition, may be inherited from a supertype component. These interfaces are indicated on the diagram by preceding the name of the interface by a forward slash. An example of this can be found in Figure 8.14, where “/orderedItem” is an interface that is implemented by a supertype of the Product component.

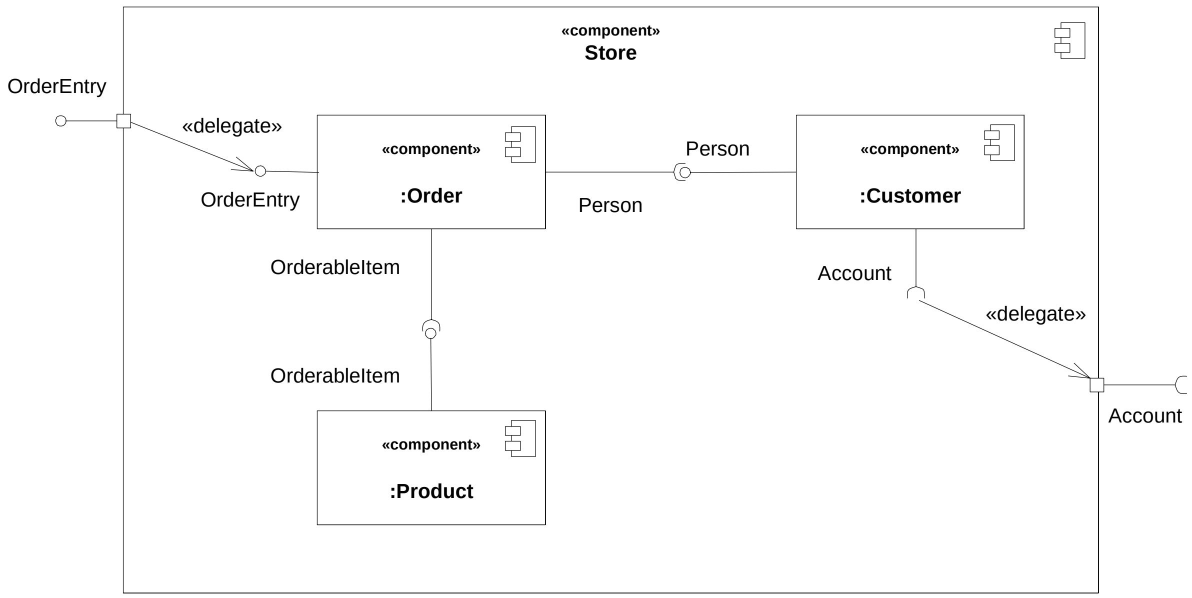


Figure 8.12 - An internal or white-box view of the internal structure of a component that contains other components as parts of its internal assembly.

Artifacts that implement components can be connected to them by physical containment or by an «implement» relationship, which is an instance of the meta association between Component and Artifact.

Examples

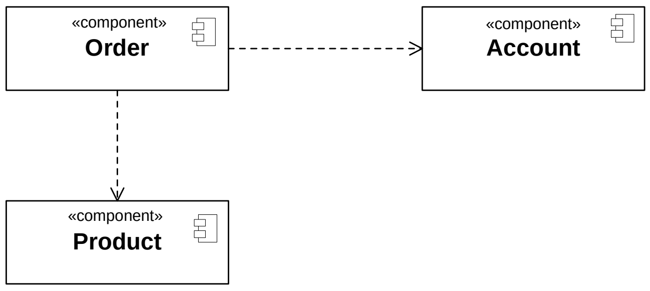


Figure 8.13 - Example of an overview diagram showing components and their general dependencies

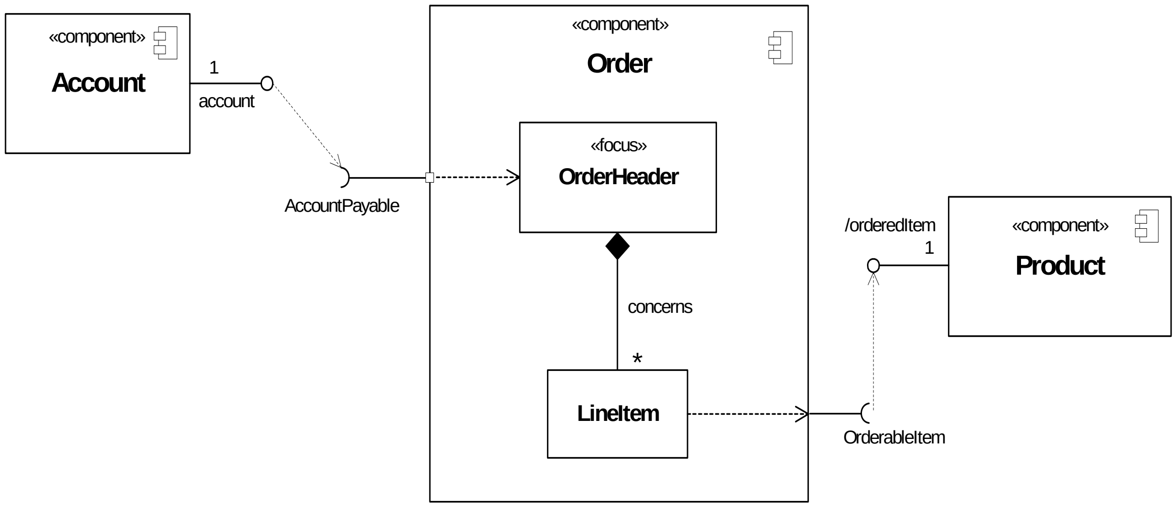


Figure 8.14 - Example of a platform independent model of a component, its provided and required interfaces, and wiring through dependencies on a structure diagram.

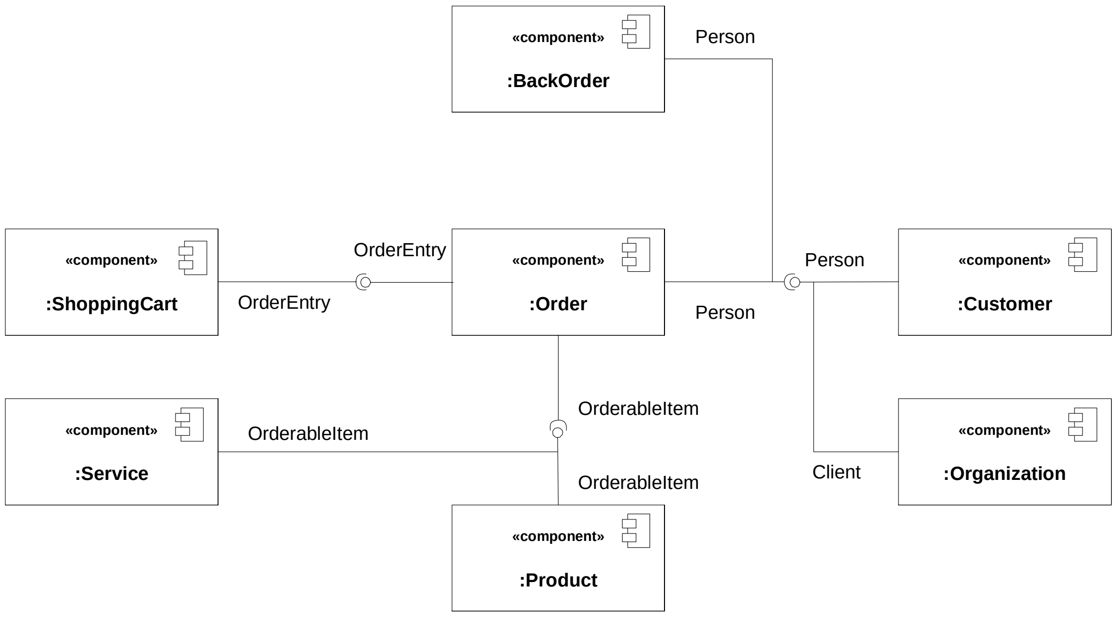


Figure 8.15 -Example of a composite structure of components, with connector wiring between provided and required interfaces of parts (Note: “Client” interface is a subtype of “Person”).

The wiring of components can be represented on structure diagrams by means of classifiers and dependencies between them (Note: the ball-and-socket notation from Figure 8.15 may be used as a notation option for dependency based wiring). On composite structure diagrams, detailed wiring can be performed at the role or instance level by defining parts and connectors.

Changes from previous UML

The following changes from UML 1.x have been made.

The component model has made a number of implicit concepts from the UML 1.x model explicit, and made the concept more applicable throughout the modeling life cycle (rather than the implementation focus of UML 1.x). In particular, the “resides” relationship from 1.x relied on namespace aspects to define both namespace aspects as well as ‘residence’ aspects. These two aspects have been separately modeled in the UML metamodel in 2.0. The basic residence relationship in 1.x maps to the realizingClassifiers relationship in 2.0. The namespace aspects are defined through the basic namespace aspects of Classifiers in UML 2.0, and extended in the PackagingComponents metamodel for optional namespace relationships to elements other than classifiers.

In addition, the Component construct gains the capabilities from the general improvements in CompositeStructures (around Parts, Ports, and Connectors).

In UML 2.0, a Component is notated by a classifier symbol that no longer has two protruding rectangles. These were cumbersome to draw and did not scale well in all circumstances. Also, they interfered with any interface symbols on the edge of the Component. Instead, a «component» keyword notation is used in UML 2.0. Optionally, a component icon that is similar to the UML 1.4 icon can still be used in the upper right-hand corner of the component symbol. For backward compatibility reasons, the UML 1.4 notation with protruding rectangles can still be used.

8.3.2 ComponentRealization (from BasicComponents)

The ComponentRealization concept is specialized in the Components package 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.

Generalizations

- “Realization (from Dependencies)” on page 129 *(merge increment)*

Description

In the metamodel, a ComponentRealization is a subtype of Dependencies::Realization.

Attributes

No additional attributes

Associations

- abstraction : Component [0..1]
  The Component that own this Realization and which is implemented by its realizing classifiers.{Subsets *Element::owner, DirectedRelationship::source, Dependency::client*}
- realizingClassifier : Classifier [1..*]
  A classifier that is involved in the implementation of the Component that owns this Realization. {Subsets *Dependency::supplier, DirectedRelationship::target*}

Constraints

No additional constraints

Semantics

A component's behavior may typically be realized (or implemented) by a number of Classifiers. In effect, it forms an abstraction for a collection of model elements. In that case, a component owns a set of Component Realization Dependencies to these Classifiers. In effect, it forms an abstraction for a collection of model elements. In that case, a component owns a set of Realization Dependencies to these Classifiers.

It should be noted that for the purpose of applications that require multiple different sets of realizations for a single component specification, a set of standard stereotypes are defined in the UML Standard Profile. In particular, «specification» and «realization» are defined there for this purpose.

Notation

A component realization is notated in the same way as the realization dependency (i.e., as a general dashed line with a hollow triangle as an arrowhead).

Changes from previous UML

The following changes from UML 1.x have been made: Realization is defined in UML 1.4 as a 'free standing' general dependency - it is not extended to cover component realization specifically. These semantics have been made explicit in UML 2.0.

8.3.3 Connector (from BasicComponents)

The connector concept is extended in the Components package to include interface based constraints and notation.

A *delegation* connector is a connector that links the external contract of a component (as specified by its ports) to the internal realization of that behavior by the component's parts. It represents the forwarding of signals (operation requests and events): a signal that arrives at a port that has a delegation connector to a part or to another port will be passed on to that target for handling.

An *assembly* connector is a connector between two components that defines that one component provides the services that another component requires. An assembly connector is a connector that is defined from a required interface or port to a provided interface or port.

Generalizations

- "Connector (from InternalStructures)" on page 174 (*merge increment*)

Description

In the metamodel, a connector kind attribute is added to the Connector metaclass. Its value is an enumeration type with valid values "assembly" or "delegation."

Attributes

Package BasicComponents

- kind : ConnectorKind
  Indicates the kind of connector.

Associations

- contract : Behavior [0..*]
  The set of Behaviors that specify the valid interaction patterns across the connector.

Constraints

[1] A delegation connector must only be defined between used Interfaces or Ports of the same kind (e.g., between two provided Ports or between two required Ports).

[2] If a delegation connector is defined between a used Interface or Port and an internal Part Classifier, then that Classifier must have an “implements” relationship to the Interface type of that Port.

[3] If a delegation connector is defined between a source Interface or Port and a target Interface or Port, then the target Interface must support a signature compatible subset of Operations of the source Interface or Port.

[4] In a complete model, if a source Port has delegation connectors to a set of delegated target Ports, then the union of the Interfaces of these target Ports must be signature compatible with the Interface that types the source Port.

[5] An assembly connector must only be defined from a required Interface or Ports to a provided Interface or Port.

Semantics

A delegation connector is a declaration that behavior that is available on a component instance is not actually realized by that component itself, but by another instance that has “compatible” capabilities. This may be another Component or a (simple) Class. The latter situation is modeled through a delegation connector from a Component Interface or Port to a contained Class that functions as a Part. In that case, the Class must have an implements relationship to the Interface of the Port.

Delegation connectors are used to model the hierarchical decomposition of behavior, where services provided by a component may ultimately be realized by one that is nested multiple levels deep within it. The word delegation suggests that concrete message and signal flow will occur between the connected ports, possibly over multiple levels. It should be noted that such signal flow is not always realized in all system environments or implementations (i.e., it may be design time only).

A port may delegate to a set of ports on subordinate components. In that case, these subordinate ports must collectively offer the delegated functionality of the delegating port. At execution time, signals will be delivered to the appropriate port. In the cases where multiple target ports support the handling of the same signal, the signal will be delivered to all these subordinate ports.

The execution time semantics for an assembly connector are that signals travel along an instance of a connector, originating in a required port and delivered to a provided port. Multiple connectors directed from a single required interface or port to provided interfaces on different components indicates that the instance that will handle the signal will be determined at execution time. Similarly, multiple required ports that are connected to a single provided port indicates that the request may originate from instances of different component types.

The interface compatibility between provided and required ports that are connected enables an existing component in a system to be replaced by one that (minimally) offers the same set of services. Also, in contexts where components are used to extend a system by offering existing services, but also adding new functionality, assembly connectors can be used to link in the new component definition. That is, by adding the new component type that offers the same set of services as existing types, and defining new assembly connectors to link up its provided and required ports to existing ports in an assembly.

Notation

A delegation connector is notated as a Connector from the delegating source Port to the handling target Part, and vice versa for required Interfaces or Ports.

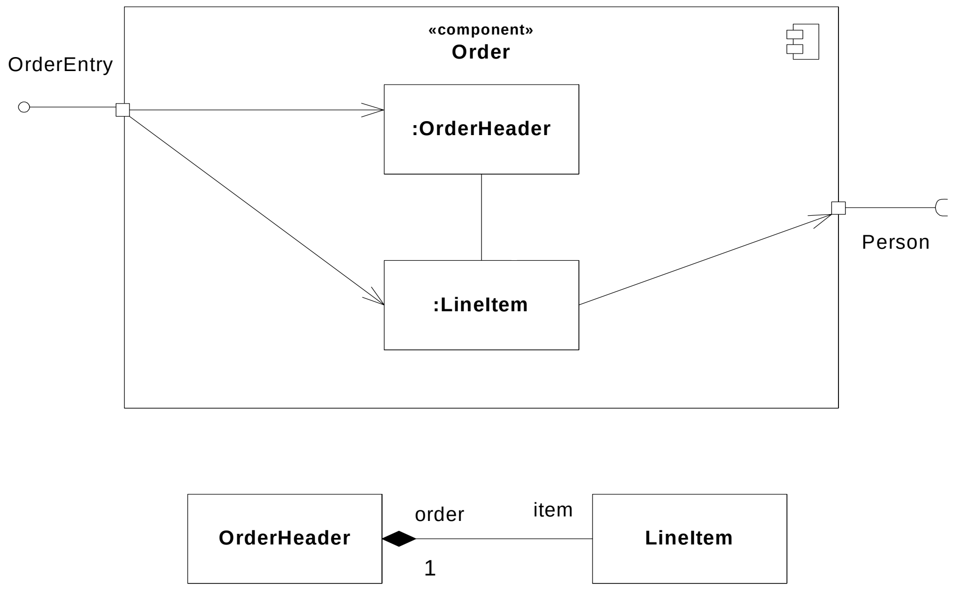


Figure 8.16 - Delegation connectors connect the externally provided interfaces of a component to the parts that realize or require them.

An assembly connector is notated by a “ball-and-socket” connection between a provided interface and a required interface. This notation allows for succinct graphical wiring of components, a requirement for scaling in complex systems.

When this notation is used to connect “complex” ports that are typed by multiple provided and/or required interfaces, the various interfaces are listed as an ordered set, designated with {provided} or {required} if needed.

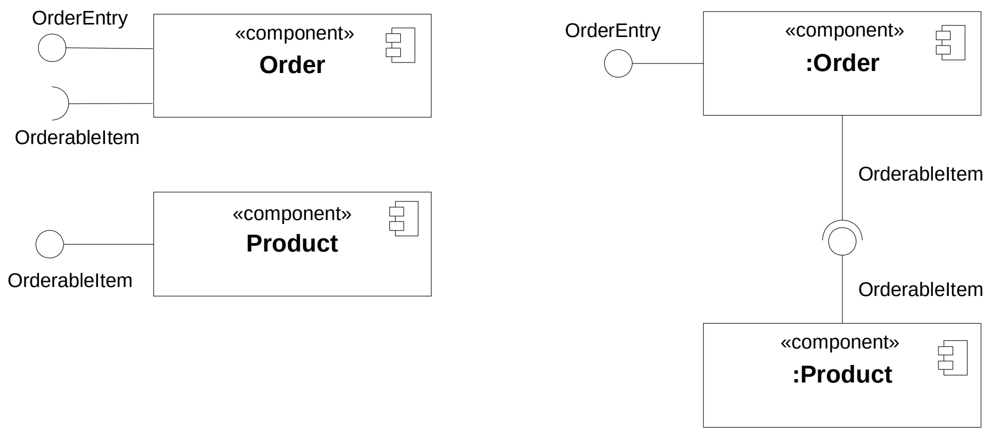


Figure 8.17 - An assembly connector maps a required interface of a component to a provided interface of another component in a certain context (definition of components, e.g., in a library on the left, an assembly of those components on the right).

Where multiple components provide or require the same interface, a single symbol representing the interface can be shown, and lines from the components can be drawn to that symbol, indicating that this interface is either a required or provided interface for the components. This presentation option is applicable whether the interface is shown using "ball-and-socket" notation, as in Figure 8.18, or just using a required or provided interface symbol.

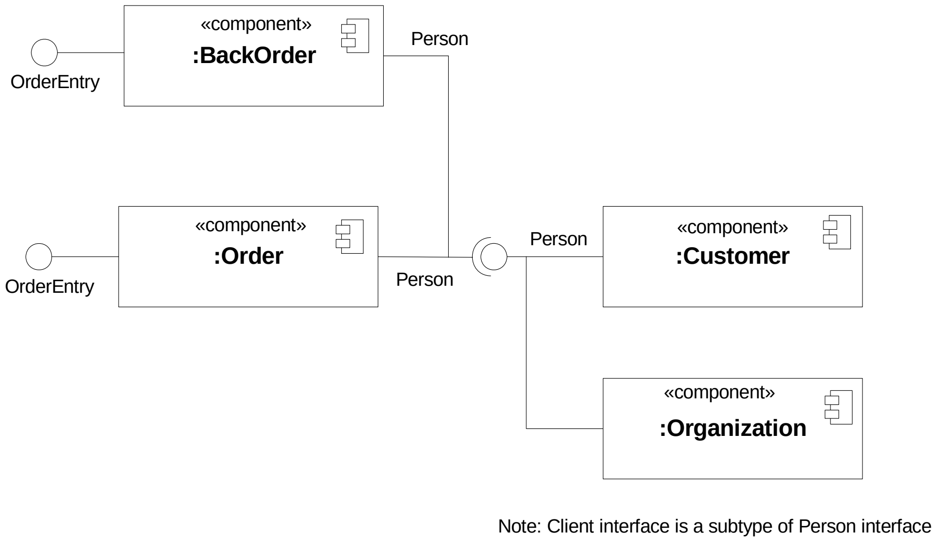


Figure 8.18 - As a notation abstraction, multiple wiring relationships can be visually grouped together in a component assembly.

Changes from previous UML

The following changes from UML 1.x have been made — Connector is not defined in UML 1.4.

8.3.4 ConnectorKind (from BasicComponents)

Generalizations

None

Description

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.

8.4 Diagrams

Structure diagram

Graphical nodes

The graphic nodes that can be included in structure diagrams are shown in Table 8.1.

Table 8.1 - Graphic nodes included in structure diagrams

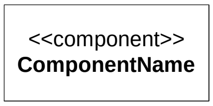
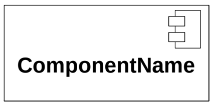
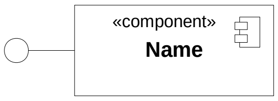
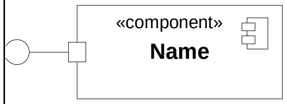
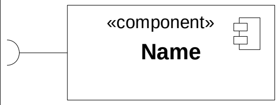
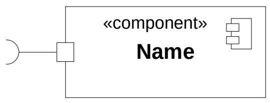
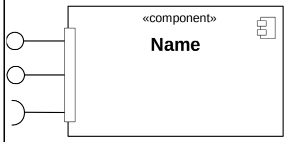
NODE TYPE	NOTATION	REFERENCE
Component	 	See "Component"
Component implements Interface		See "Interface"
Component has provided Port (typed by Interface)		See "Port"
Component uses Interface		See "Interface"

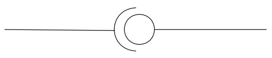
Table 8.1 - Graphic nodes included in structure diagrams

NODE TYPE	NOTATION	REFERENCE
Component has required Port (typed by Interface)		See “Port”
Component has complex Port (typed by provided and required Interfaces)		See “Port”

Graphical paths

The graphic paths that can be included in structure diagrams are shown in Table 8.2.

Table 8.2 - Graphic nodes included in structure diagrams

PATH TYPE	NOTATION	REFERENCE
Assembly connector		See “assembly connector.” Also used as notation option for wiring between interfaces using Dependencies.

Variations

Variations of structure diagrams often focus on particular structural aspects, such as relationships between packages, showing instance specifications, or relationships between classes. There are no strict boundaries between different variations; it is possible to display any element you normally display in a structure diagram in any variation.

Component diagram

The following nodes and edges are typically drawn in a component diagram:

- Component
- Interface
- ComponentRealization, Interface Realization, Usage Dependencies
- Class
- Artifact
- Port

9 Composite Structures

9.1 Overview

The term “structure” in this clause refers to a composition of interconnected elements, representing run-time instances collaborating over communications links to achieve some common objectives.

Internal Structures

The InternalStructure subpackage provides mechanisms for specifying structures of interconnected elements that are created within an instance of a containing classifier. A structure of this type represents a decomposition of that classifier and is referred to as its “internal structure.”

Ports

The Ports subpackage provides mechanisms for isolating a classifier from its environment. This is achieved by providing a point for conducting interactions between the internals of the classifier and its environment. This interaction point is referred to as a “port.” Multiple ports can be defined for a classifier, enabling different interactions to be distinguished based on the port through which they occur. By decoupling the internals of the classifier from its environment, ports allow a classifier to be defined independently of its environment, making that classifier reusable in any environment that conforms to the interaction constraints imposed by its ports.

Collaborations

Objects in a system typically cooperate with each other to produce the behavior of a system. The behavior is the functionality that the system is required to implement.

A behavior of a collaboration will eventually be exhibited by a set of cooperating instances (specified by classifiers) that communicate with each other by sending signals or invoking operations. However, to understand the mechanisms used in a design, it may be important to describe only those aspects of these classifiers and their interactions that are involved in accomplishing a task or a related set of tasks, projected from these classifiers. *Collaborations* allow us to describe only the relevant aspects of the cooperation of a set of instances by identifying the specific roles that the instances will play. *Interfaces* allow the externally observable properties of an instance to be specified without determining the classifier that will eventually be used to specify this instance. Consequentially, the roles in a collaboration will often be typed by interfaces and will then prescribe properties that the participating instances must exhibit, but will not determine what class will specify the participating instances.

StructuredClasses

The StructuredClasses subpackage supports the representation of classes that may have ports as well as internal structure.

Actions

The Actions subpackage adds actions that are specific to the features introduced by composite structures (e.g., the sending of messages via ports).

9.2 Abstract Syntax

Figure 9.1 shows the dependencies of the CompositeStructures packages.

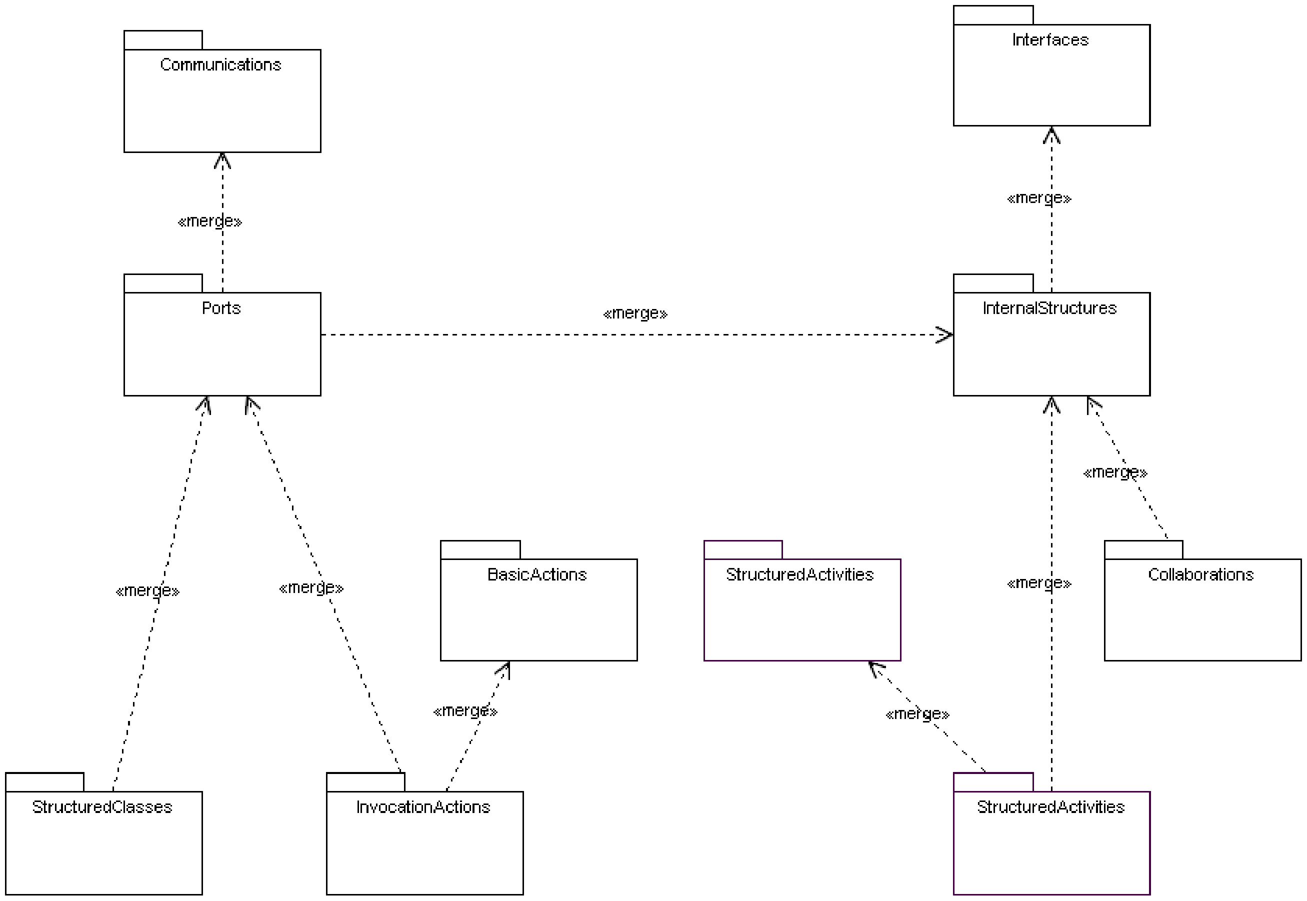


Figure 9.1 - Dependencies between packages described in this clause

Package InternalStructures

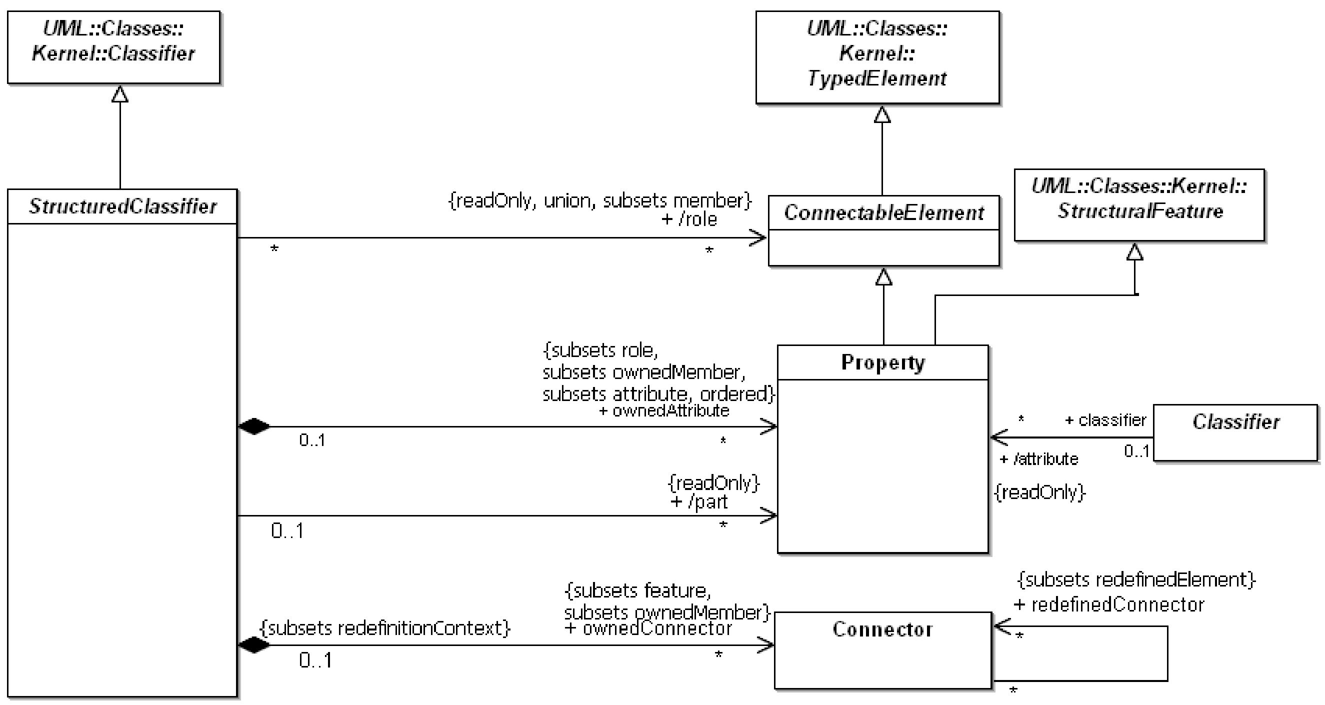

UML::Classes::Kernel::Classifier
UML::Classes::Kernel::TypedElement
StructuredClassifier
ConnectableElement
UML::Classes::Kernel::StructuralFeature
{readOnly, union, subsets member}
+ /role
*
*
Property
{subsets role, subsets ownedMember, subsets attribute, ordered}
+ ownedAttribute
0..1
*
Classifier
+ classifier
*
0..1
+ /attribute
{readOnly}
{readOnly}
+ /part
0..1
*
{subsets feature, subsets ownedMember}
+ ownedConnector
{subsets redefinitionContext}
0..1
*
Connector
{subsets redefinedElement}
+ redefinedConnector
*
*

Figure 9.2 - Structured classifier

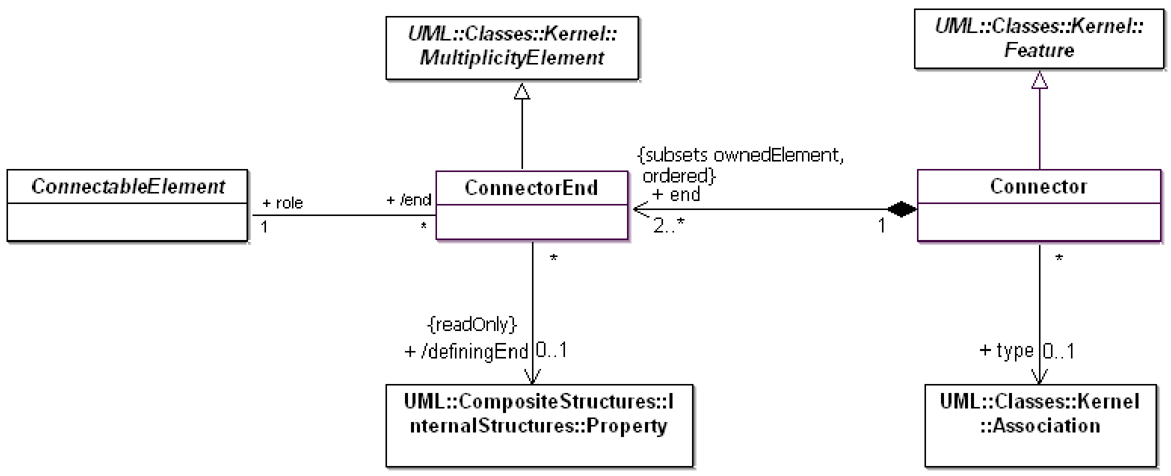

UML::Classes::Kernel::MultiplicityElement
UML::Classes::Kernel::Feature
ConnectableElement
+ role
1
+ /end
*
ConnectorEnd
{subsets ownedElement, ordered}
+ end
2..*
1
Connector
*
{readOnly}
+ /definingEnd
0..1
*
+ type
0..1
UML::CompositeStructures::InternalStructures::Property
UML::Classes::Kernel::Association

Figure 9.3 - Connectors

Package Ports

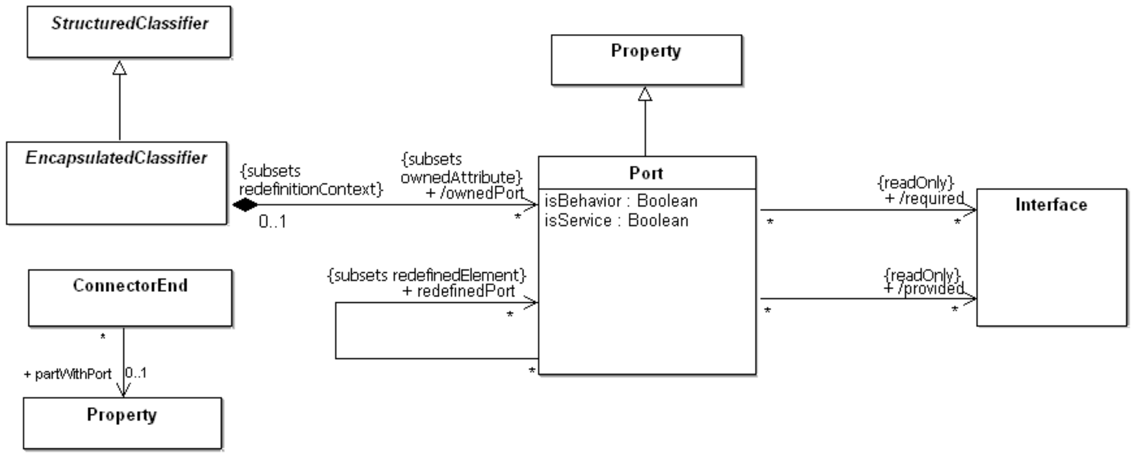


Figure 9.4 - The Port metaclass

Package StructuredClasses

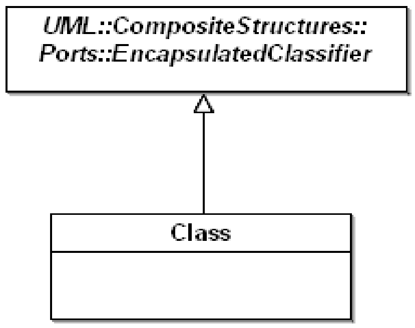


Figure 9.5 - Classes with internal structure

Package Collaborations

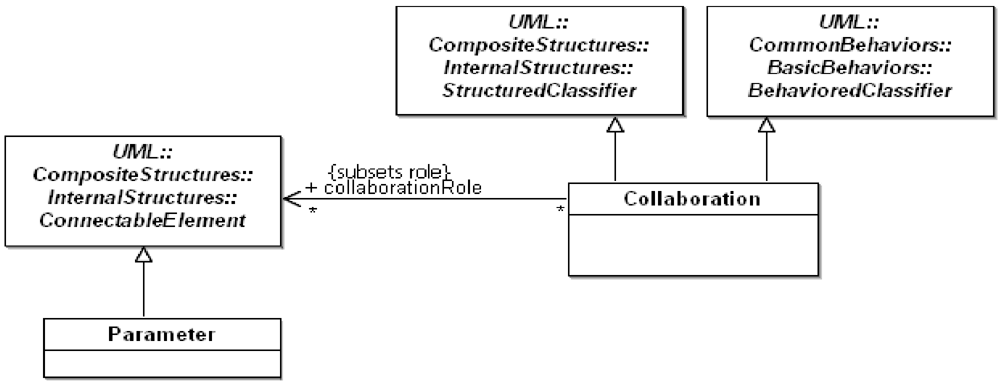


Figure 9.6 - Collaboration

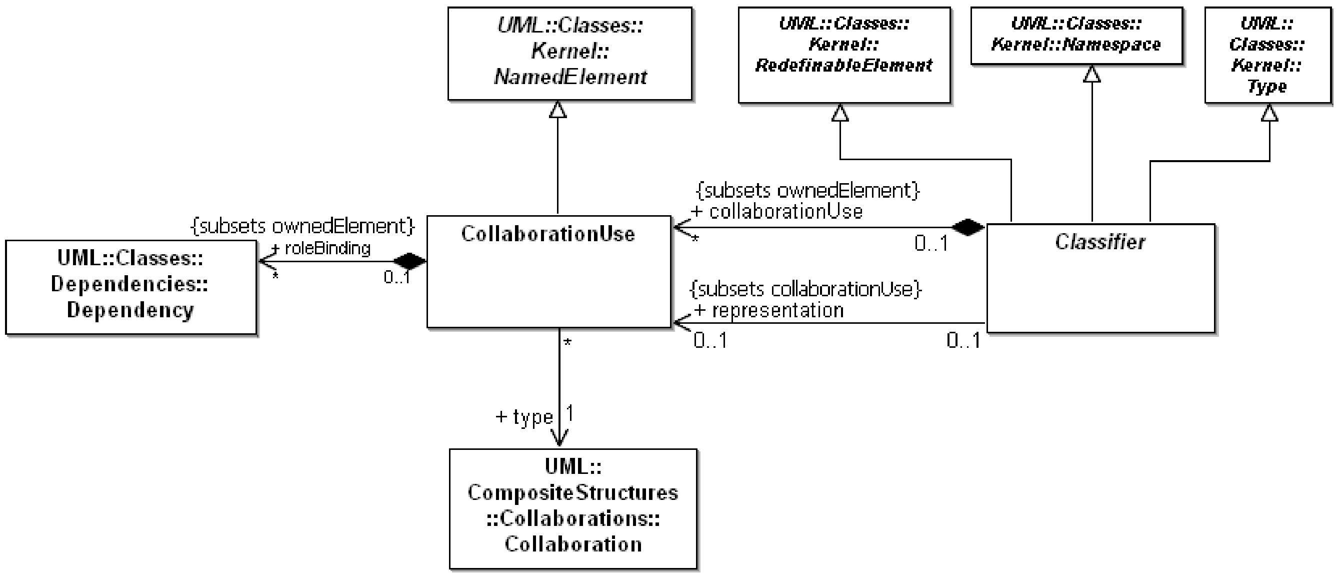


Figure 9.7 - Collaboration.use and role binding

Package InvocationActions

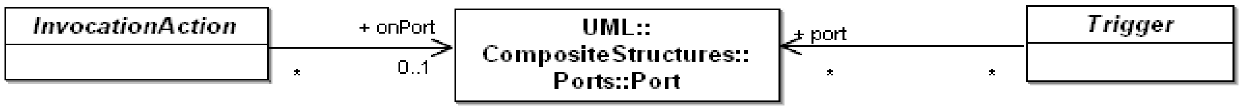


Figure 9.8 - Actions specific to composite structures

Package StructuredActivities

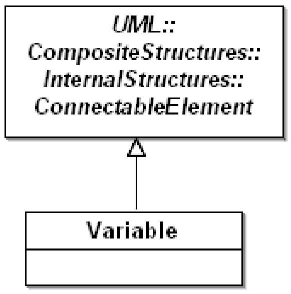


Figure 9.9 - Extension to Variable

9.3 Class Descriptions

9.3.1 Class (from StructuredClasses)

Generalizations

- "EncapsulatedClassifier (from Ports)" on page 178.

Description

Extends the metaclass Class with the capability to have an internal structure and ports.

Semantics

See "Property (from InternalStructures)" on page 183, "Connector (from InternalStructures)" on page 174, and "Port (from Ports)" on page 179 for the semantics of the features of Class. Initialization of the internal structure of a class is discussed in sub clause "StructuredClassifier (from InternalStructures)" on page 186.

A class acts as the namespace for various kinds of classifiers defined within its scope, including classes. Nesting of classifiers limits the visibility of the classifier to within the scope of the namespace of the containing class and is used for reasons of information hiding. Nested classifiers are used like any other classifier in the containing class.

Notation

See “Class (from Kernel)” on page 49, “StructuredClassifier” on page 183, and “Port” on page 179.

Presentation Options

A usage dependency may relate an instance value 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 instance value 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. The instance value that is the supplier of the usage dependency represents the default value of the single return result parameter of a constructor operation. (The constructor operation is typically denoted by the stereotype “create,” as shown in Figure 9.10.)

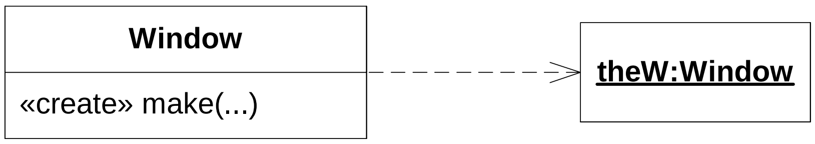


Figure 9.10 - Instance specification describes the return value of an operation

Changes from previous UML

Class has been extended with internal structure and ports.

9.3.2 Classifier (from Collaborations)

Generalizations

- 7.3.8, “Classifier (from Kernel, Dependencies, PowerTypes),” on page 52

Description

Classifier is extended with the capability to own collaboration uses. These collaboration uses link a collaboration with the classifier to give a description of the workings of the classifier.

Associations

- collaborationUse: CollaborationUse
  References the collaboration uses owned by the classifier. (Subsets *Element::ownedElement*)
- representation: CollaborationUse [0..1]
  References a collaboration use which indicates the collaboration that represents this classifier. (Subsets *Classifier::collaborationUse*)

Semantics

A classifier can own collaboration uses that relate (aspects of) this classifier to a collaboration. The collaboration describes those aspects of this classifier.

One of the collaboration uses owned by a classifier may be singled out as representing the behavior of the classifier as a whole. The collaboration that is related to the classifier by this collaboration use shows how the instances corresponding to the structural features 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 collaboration use.

Notation

See "CollaborationUse (from Collaborations)" on page 171

Changes from previous UML

Replaces and widens the applicability of *Collaboration.usedCollaboration*. Together with the newly introduced internal structure concept replaces Collaboration.representedClassifier.

9.3.3 Collaboration (from Collaborations)

A collaboration describes a structure of collaborating elements (roles), each performing a specialized function, which collectively accomplish some desired functionality. Its primary purpose is to explain how a system works and, therefore, it typically only incorporates those aspects of reality that are deemed relevant to the explanation. Thus, details, such as the identity or precise class of the actual participating instances are suppressed.

Generalizations

- "BehavioredClassifier (from BasicBehaviors, Communications)" on page 434
- "StructuredClassifier (from InternalStructures)" on page 186

Description

A collaboration is represented as a kind of classifier and defines a set of cooperating entities to be played by instances (its roles), as well as a set of connectors that define communication paths between the participating instances. The cooperating entities are the properties of the collaboration (see "Property (from InternalStructures)" on page 183).

A collaboration specifies a view (or projection) of a set of cooperating classifiers. It describes the required links between instances that play the roles of the collaboration, as well as the features required of the classifiers that specify the participating instances. Several collaborations may describe different projections of the same set of classifiers.

Attributes

No additional attributes

Associations

- collaborationRole: ConnectableElement [*]
  References connectable elements (possibly owned by other classifiers), which represent roles that instances may play in this collaboration. (Subsets *StructuredClassifier.role*)

Constraints

No additional constraints

Semantics

Collaborations are generally 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. A role specifies (through its type) the required set of features a participating instance must have. 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 (see "Interface (from Interfaces)" on page 86). An interface is a description of a set of properties (externally observable features) required or provided by an instance. An interface can be viewed as a projection of the externally observable features of a classifier realizing the interface. Instances of different classifiers can play a role defined by a given interface, as long as these classifiers realize the interface (i.e., have all the required properties). Several interfaces may be realized by the same classifier, even in the same context, but their features may be different subsets of the features of the realizing classifier.

Collaborations may be specialized from other collaborations. If a role is extended in the specialization, the type of a role in the specialized collaboration must conform to the type of the role 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 may be attached to an operation or a classifier through a CollaborationUse. A collaboration used in this way describes how this operation or this classifier is realized by a set of cooperating instances. The connectors defined within the collaboration specify links between the instances when they perform the behavior specified in the classifier. The collaboration specifies the context in which behavior is performed. Such a collaboration may constrain the set of valid interactions that may occur between the instances that are connected by a link.

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 (the collaboration is a selective view of that situation).

Notation

A collaboration is shown as a dashed ellipse icon 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 icon. Alternatively, a composite structure diagram can be used.

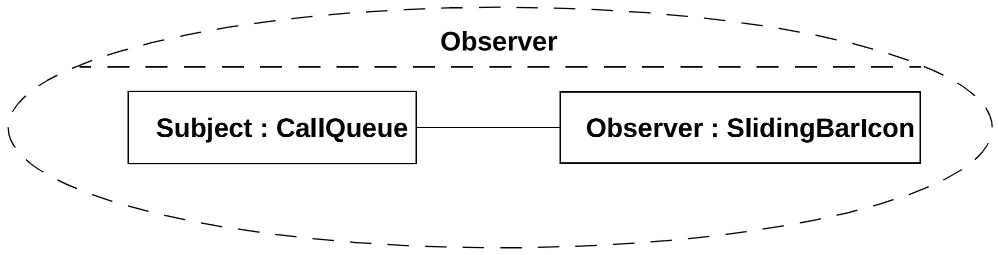


Figure 9.11 - The internal structure of the Observer collaboration shown inside the collaboration icon (a connection is shown between the Subject and the Observer role).

Using an alternative notation for properties, a line may be drawn from the collaboration icon to each of the symbols 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 collaboration icon can show the use of a collaboration together with the actual classifiers that occur in that particular use of the collaboration (see Figure 9.12).

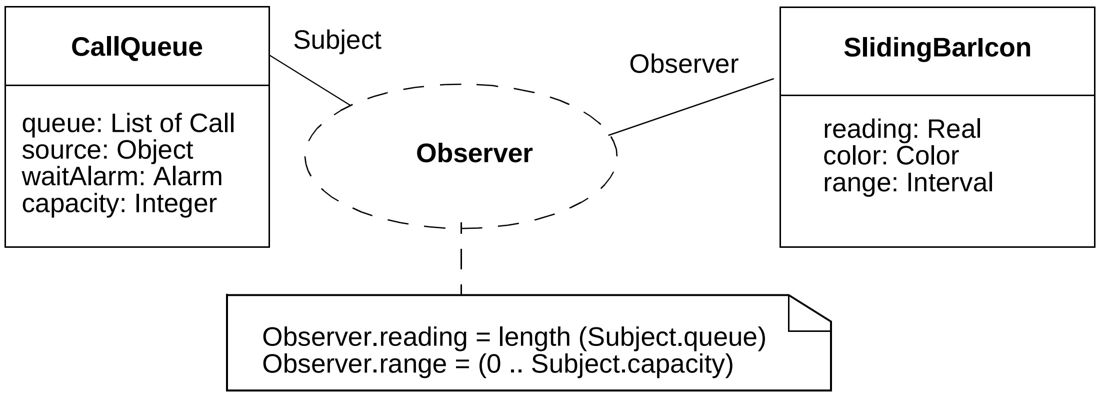


Figure 9.12 - In the Observer collaboration two roles, a Subject and an Observer, collaborate to produce the desired behavior. Any instance playing the Subject role must possess the properties specified by CallQueue, and similarly for the Observer role.

Rationale

The primary purpose of collaborations is to explain how a system of communicating entities collectively accomplish a specific task or set of tasks without necessarily having to incorporate detail that is irrelevant to the explanation. It is particularly useful as a means for capturing standard design patterns.

Changes from previous UML

The contents of a collaboration is specified as its internal structure relying on roles and connectors; the concepts of ClassifierRole, AssociationRole, and AssociationEndRole have been superseded. A collaboration in UML 2.0 is a kind of classifier, and can have any kind of behavioral descriptions associated. There is no loss in modeling capabilities.

9.3.4 CollaborationUse (from Collaborations)

A collaboration use represents the application of the pattern described by a collaboration to a specific situation involving specific classes or instances playing the roles of the collaboration.

Generalizations

- “NamedElement (from Kernel, Dependencies)” on page 98

Description

A collaboration use represents one particular use of a collaboration to explain the relationships between the properties of a classifier. A collaboration use shows how the pattern described by a collaboration is applied in a given context, by binding specific entities from that context to the roles of the collaboration. Depending on the context, these entities could be structural features of a classifier, instance specifications, or even roles in some containing collaboration. There may be multiple occurrences of a given collaboration within a classifier, each involving a different set of roles and connectors. A given role or connector may be involved in multiple occurrences of the same or different collaborations.

Associated dependencies map features of the collaboration type to features in the classifier. These dependencies indicate which role in the classifier plays which role in the collaboration.

Attributes

No additional attributes

Associations

- type: Collaboration [1]
  The collaboration that is used in this occurrence. The collaboration defines the cooperation between its roles that are mapped to properties of the classifier owning the collaboration use.
- roleBinding: Dependency [*]
  A mapping between features of the collaboration type and features of the classifier or operation. This mapping indicates which connectable element of the classifier or operation plays which role(s) in the collaboration. A connectable element may be bound to multiple roles in the same collaboration use (that is, it may play multiple roles).

Constraints

[1] All the client elements of a *roleBinding* are in one classifier and all supplier elements of a *roleBinding* are in one collaboration and they are compatible.

[2] Every role in the collaboration is bound within the collaboration use to a connectable element within the classifier or operation.

[3] The connectors in the classifier connect according to the connectors in the collaboration.

Semantics

A collaboration use relates a feature in its collaboration type to a connectable element in the classifier or operation that owns the collaboration use.

Any behavior attached to the collaboration type applies to the set of roles and connectors bound within a given collaboration use. For example, an interaction among parts of a collaboration applies to the classifier parts bound to a single collaboration use. If the same connectable element is used in both the collaboration and the represented element, no role binding is required.

Semantic Variation Points

It is a semantic variation when client and supplier elements in role bindings are compatible.

Notation

A collaboration use is shown by a dashed ellipse containing the name of the occurrence, a colon, and the name of the collaboration type. For every role binding, 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.

Examples

This example shows the definition of two collaborations, *Sale* (Figure 9.13) and *BrokeredSale* (Figure 9.14). *Sale* is used twice as part of the definition of *BrokeredSale*. *Sale* is a collaboration among two roles, 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*.

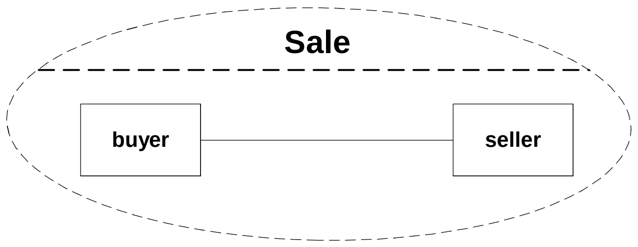


Figure 9.13 - The Sale collaboration

BrokeredSale is a collaboration among three roles, a *producer*, a *broker*, and a *consumer.* The specification of *BrokeredSale* shows that it consists of two occurrences 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 are implicit in the *BrokeredSale* collaboration in virtue of them being comprised of *Sale*. The *BrokeredSale* collaboration could itself be used as part of a larger collaboration.

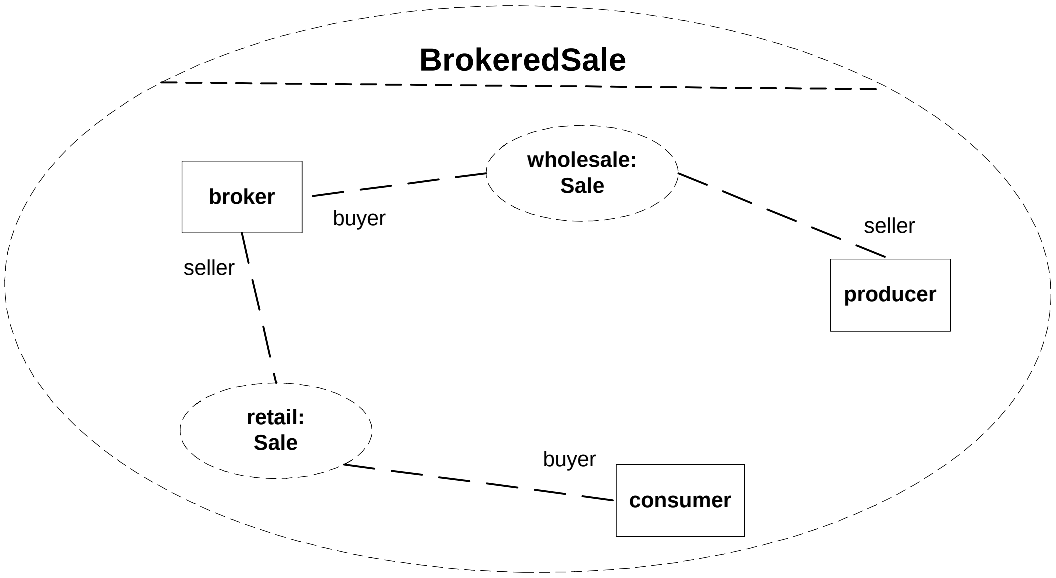


Figure 9.14 - The BrokeredSale collaboration

Figure 9.15 shows part of the *BrokeredSale* collaboration in a presentation option.

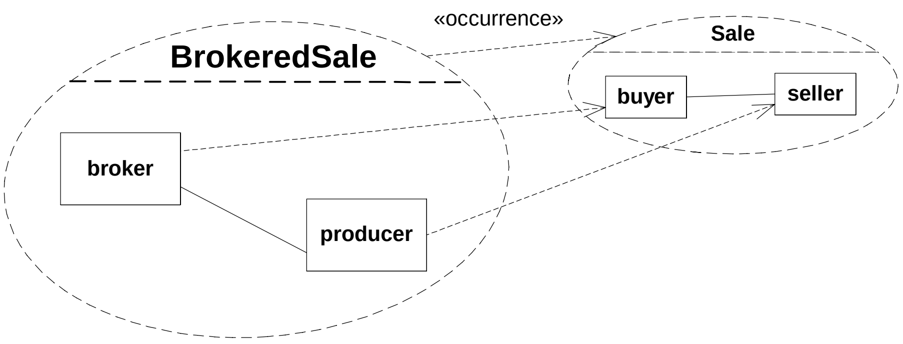


Figure 9.15 - A subset of the BrokeredSale collaboration

Rationale

A collaboration use is used to specify the application of a pattern specified by a collaboration to a specific situation. In that regard, it acts as the invocation of a macro with specific values used for the parameters (roles).

Changes from previous UML

This metaclass has been added.

9.3.5 ConnectableElement (from InternalStructures)

Generalizations

- "TypedElement (from Kernel)" on page 136

Description

A ConnectableElement is an abstract metaclass representing a set of instances that play roles of a classifier. Connectable elements may be joined by attached connectors and specify configurations of linked instances to be created within an instance of the containing classifier.

Attributes

No additional attributes

Associations

- /end: ConnectorEnd
  Denotes a connector that attaches to this connectable element. It is derived in the following way:

```
context ConnectableElement::end derive:
        ConnectorEnd.allInstances() -> select (e | e.role = self)
```

Constraints

No additional constraints

Semantics

The semantics of ConnectableElement is given by its concrete subtypes.

Notation

None

Rationale

This metaclass supports factoring out the ability of a model element to be linked by a connector.

Changes from previous UML

This metaclass generalizes the concept of classifier role from 1.x.

9.3.6 Connector (from InternalStructures)

Specifies a link that enables communication between two or more instances. This link may be an instance of an association, or it may represent the possibility of the instances being able to communicate because their identities are known by virtue of being passed in as parameters, held in variables or slots, or because the communicating instances are the same instance. The link may be realized by something as simple as a pointer or by something as complex as a network connection. In contrast to associations, which specify links between any instance of the associated classifiers, connectors specify links between instances playing the connected parts only.

Generalizations

- "Feature (from Kernel)" on page 70

Description

Each connector may be attached to two or more connectable elements, each representing a set of instances. Each connector end is distinct in the sense that it plays a distinct role in the communication realized over a connector. The communications realized over a connector may be constrained by various constraints (including type constraints) that apply to the attached connectable elements.

Attributes

No additional attributes

Associations

- end: ConnectorEnd [2..*]
  A connector consists of at least two connector ends, each representing the participation of instances of the classifiers typing the connectable elements attached to this end. The set of connector ends is ordered. (Subsets*Element::ownedElement*)
- type: Association [0..1]
  An optional association that specifies the link corresponding to this connector.
- redefinedConnector: Connector [0..*]
  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 connector ends of the redefining connector may specialize the types of the connector ends of the redefined connector. The properties of the connector ends of the redefining connector may be replaced. (Subsets *Element::redefinedElement*)

Constraints

[1] The types of the connectable elements that the ends of a connector are attached to must conform to the types of the association ends of the association that types the connector, if any.

[2] The connectable elements attached to the ends of a connector must be compatible.

[3] 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.

Semantics

If a connector between two roles of a classifier is a feature of an instantiable classifier, it declares that a link may exist within an instance of that classifier. If a connector between two roles of a classifier is a feature of an uninstantiable classifier, it declares that links may exist within an instance of the classifier that realizes the original classifier. These links will connect instances corresponding to the parts joined by the connector.

Links corresponding to connectors may be created upon the creation of the instance of the containing classifier (see "StructuredClassifier" on page 183). All such links corresponding to connectors are destroyed, when the containing classifier instance is destroyed.

If the type of the connector is omitted, the type is inferred based on the connector, as follows: If the type of a role (i.e, the connectable element attached to a connector end) realizes an interface that has a unique association to another interface which is realized by the type of another role (or an interface compatible to that interface is realized by the type of another role), then that association is the type of the connector between these parts. If the connector realizes a collaboration (that is, a collaboration use maps the connector to a connector in an associated collaboration through role bindings), then the type of the connector is an anonymous association with association ends corresponding to each connector end. The type of each association end is the classifier that realizes the parts connected to the matching connector in the collaboration. Any adornments on the connector ends (either the original connector or the connector in the collaboration) specify adornments of the ends of the inferred association; otherwise, the type of the connector is an anonymously named association with association ends corresponding to each connector end. The type of each association end is the type of the part that each corresponding connector end is attached to. Any adornments on the connector ends specify adornments of the ends of the inferred association. Any inferred associations are always bidirectionally navigable and are owned by the containing classifier.

Semantic Variation Points

What makes connectable elements compatible is a semantic variation point.

Notation

A connector is drawn using the notation for association (see "Association (from Kernel)" on page 39). The optional name string of the connector obeys the following syntax:

([name] ':' <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.

Examples

Examples are shown in "StructuredClassifier" on page 183.

Changes from previous UML

Connector has been added in UML 2.0. The UML 1.4 concept of association roles is subsumed by connectors.

9.3.7 ConnectorEnd (from InternalStructures, Ports)

Generalizations

- "MultiplicityElement (from Kernel)" on page 94

Description

A connector end is an endpoint of a connector, which attaches the connector to a connectable element. Each connector end is part of one connector.

Attributes

No additional attributes

Associations

InternalStructures

- role: ConnectableElement [1]
  The connectable element attached at this connector end. 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 connectable element.
- definingEnd: Property [0..1]
  A derived association referencing the corresponding association end on the association that types the connector owing this connector end. This association is derived by selecting the association end at the same place in the ordering of association ends as this connector end.

Ports

- partWithPort: Property [0..1]
  Indicates the role of the internal structure of a classifier with the port to which the connector end is attached.

Constraints

[1] If a connector end is attached to a port of the containing classifier, *partWithPort* will be empty.

[2] If a connector end references both a *role* and a *partWithPort*, then the *role* must be a port that is defined by the type of the *partWithPort*.

[3] The property held in self.partWithPort must not be a Port.

[4] The multiplicity of the connector end may not be more general than the multiplicity of the association typing the owning connector.

Semantics

InternalStructures

A connector end describes which connectable element is attached to the connector owning that end. Its multiplicity indicates the number of instances that may be linked to each instance of the property connected on the other end.

Notation

InternalStructures

Adornments may be shown on the connector end corresponding to adornments on association ends (see “Association (from Kernel)” on page 39). In cases where there is no explicit association in the model typing the connector, these adornments specify the multiplicities of an implicit association; otherwise, they show properties of that association, or specializations of these on the connector.. The multiplicity indicates the number of instances that may be connected to each instance of the role on the other end. If no multiplicity is specified, the multiplicity matches the multiplicity of the role the end is attached to.

Ports

If the end is attached to a port on a part of the internal structure and no multiplicity is specified, the multiplicity matches the multiplicity of the port multiplied by the multiplicity of the part (if any).

Changes from previous UML

Connector end has been added in UML 2.0. The UML 1.4 concept of association end roles is subsumed by connector ends.

9.3.8 EncapsulatedClassifier (from Ports)

Generalizations

- "StructuredClassifier (from InternalStructures)" on page 186

Description

Extends a classifier with the ability to own ports as specific and type checked interaction points.

Attributes

No additional attributes

Associations

- /ownedPort: Port [0..*]
  The set of port attributes owned by EncapsulatedClassifier. (Subsets *Class::ownedAttribute*)

Constraints

No additional constraints

Semantics

See "Port" on page 179.

Notation

See "Port" on page 179.

Changes from previous UML

This metaclass has been added to UML.

9.3.9 InvocationAction (from InvocationActions)

Generalizations

- "InvocationAction (from BasicActions)" on page 256 *(merge increment)*

Description

In addition to targeting an object, invocation actions can also invoke behavioral features on ports from where the invocation requests are routed onwards on links deriving from attached connectors. Invocation actions may also be sent to a target via a given port, either on the sending object or on another object.

Associations

- onPort: Port [0..1]
  An optional port of the receiver object on which the behavioral feature is invoked.

Constraints

[1] The *onPort* must be a port on the receiver object.

Semantics

The target value of an invocation action may also be a port. In this case, the invocation request is sent to the object owning this port as identified by the port identity, and is, upon arrival, handled as described in "Port" on page 179.

Notation

The optional port is identified by the phrase "via <port>" in the name string of the icon denoting the particular invocation action.

9.3.10 Parameter (from Collaborations)

Generalizations

- "ConnectableElement (from InternalStructures)" on page 174
- "Parameter (from Kernel)" on page 120 *(merge increment)*

Description

Parameters are allowed to be treated as connectable elements.

Constraints

[1] A parameter may only be associated with a connector end within the context of a collaboration.

self.end->notEmpty() **implies** self.collaboration->notEmpty()

9.3.11 Port (from Ports)

A port is a property of a classifier that specifies a distinct interaction point between that classifier and its environment or between the (behavior of the) classifier and its internal parts. Ports are connected to properties of the classifier by connectors through which requests can be made to invoke the behavioral features of a classifier. A Port may specify the services a classifier provides (offers) to its environment as well as the services that a classifier expects (requires) of its environment.

Generalizations

- "Property (from InternalStructures)" on page 183

Description

Ports represent interaction points between a classifier and its environment. The interfaces associated with a port specify the nature of the interactions that may occur over a port. The required interfaces of a port characterize the requests that may be made from the classifier to its environment through this port. The provided interfaces of a port characterize requests to the classifier that its environment may make through this port.

A port has the ability to specify that any requests arriving at this port are handled by the behavior of the instance of the owning classifier, rather than being forwarded to any contained instances, if any.

Attributes

- isService: Boolean
  If *true*, indicates that this port is used to provide the published functionality of a classifier. If *false*, this port is used to implement the classifier but is not part of the essential externally-visible functionality of the classifier and can, therefore, be altered or deleted along with the internal implementation of the classifier and other properties that are considered part of its implementation. The default value for this attribute is *true*.
- isBehavior: Boolean
  Specifies whether requests arriving at this port are sent to the classifier behavior of this classifier (see "BehavioredClassifier (from BasicBehaviors, Communications)" on page 434). Such ports are referred to as *behavior port*. Any invocation of a behavioral feature targeted at a behavior port will be handled by the instance of the owning classifier itself, rather than by any instances that this classifier may contain. The default value is *false*.

Associations

- required: Interface
  References the interfaces specifying the set of operations and receptions that the classifier expects its environment to handle. This association is derived as the set of interfaces required by the type of the port or its supertypes.
- provided: Interface
  References the interfaces specifying the set of operations and receptions that the classifier offers to its environment, and which it will handle either directly or by forwarding it to a part of its internal structure. This association is derived from the interfaces realized by the type of the port or by the type of the port, if the port was typed by an interface.
- redefinedPort: Port
  A port may be redefined when its containing classifier 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. (Subsets *Element::redefinedElement*)

Constraints

[1] The required interfaces of a port must be provided by elements to which the port is connected.

[2] Port.aggregation must be *composite*.

[3] When a port is destroyed, all connectors attached to this port will be destroyed also.

[4] A defaultValue for port cannot be specified when the type of the Port is an Interface.

Semantics

A port represents an interaction point between a classifier instance and its environment or between a classifier instance and instances it may contain. A port by default has public visibility.

The required interfaces characterize services that the owning classifier expects from its environment and that it may access through this interaction point: Instances of this classifier expect that the features owned by its required interfaces will be offered by one or more instances in its environment. The provided interfaces characterize the behavioral features that the owning classifier offers to its environment at this interaction point. The owning classifier must offer the features owned by the provided interfaces.

The provided and required interfaces completely characterize any interaction that may occur between a classifier and its environment at a port with respect to the data communicated at this port and the behaviors that may be invoked through this port. The interfaces do not necessarily establish the exact sequences of interactions across the port. When an instance of a classifier is created, instances corresponding to each of its ports are created and held in the slots specified by the ports, in accordance with its multiplicity. These instances are referred to as "interaction points" and provide unique references. A link from that instance to the instance of the owning classifier is created through which communication is forwarded to the instance of the owning classifier or through which the owning classifier communicates with its environment. It is, therefore, possible for an instance to differentiate between requests for the invocation of a behavioral feature 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. (In the following, "requests arriving at a port" shall mean "request occurrences arriving at the interaction point of this instance corresponding to this port.")

The interaction point object must be an instance of a classifier that realizes the provided interfaces of the port. If the port was typed by an interface, the classifier typing the interaction point object realizes that interface. If the port was typed by a class, the interaction point object will be an instance of that class. The latter case allows elaborate specification of the communication over a port. For example, it may describe that communication is filtered, modified in some way, or routed to other parts depending on its contents as specified by the classifier that types the port.

If connectors are attached to both the port when used on a property within the internal structure of a classifier and the port on the container of an internal structure, the instance of the owning classifier will forward any requests arriving at this port along the link specified by those connectors. If there is a connector attached to only one side of a port, any requests arriving at this port will terminate at this port.

For a behavior port, the instance of the owning classifier will handle requests arriving at this port (as specified in the behavior of the classifier, see Clause 13, "Common Behaviors"), if this classifier has any behavior. If there is no behavior defined for this classifier, any communication arriving at a behavior port is lost.

Semantic Variation Points

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 a semantic variation point whether these requests will be forwarded on all links, or on only one of those links. In the latter case, one possibility is that the link at which this request will be forwarded will be arbitrarily selected among those links leading to an instance that had been specified as being able to handle this request (i.e., this request is specified in a provided interface of the part corresponding to this instance).

Notation

A port of a classifier 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 classifier or it may be shown inside the rectangle symbol.

A port of a classifier may also be shown as a small square symbol overlapping the boundary of the rectangle symbol denoting a part typed by that classifier (see Figure 9.16). The name of the port is shown near the port; the multiplicity follows the name surrounded by brackets. Name and multiplicity may be elided.

The type of a port may be shown following the port name, separated by colon (":"). A provided interface may be shown using the "lollipop" notation (see "Interface (from Interfaces)" on page 86) attached to the port. A required interface may be shown by the "socket" notation attached to the port. The presentation options shown there are also applicable to interfaces of ports. Figure 9.16 shows the notation for ports: *p* is a port on the *Engine* class. The provided interface (also its type) of port *p* is *powertrain*. The multiplicity of *p* is "1." In addition, a required interface, *power*, is shown also. The figure on the left shows the provided interface using the "lollipop" notation, while the figure on the right shows the interface as the type of the port.

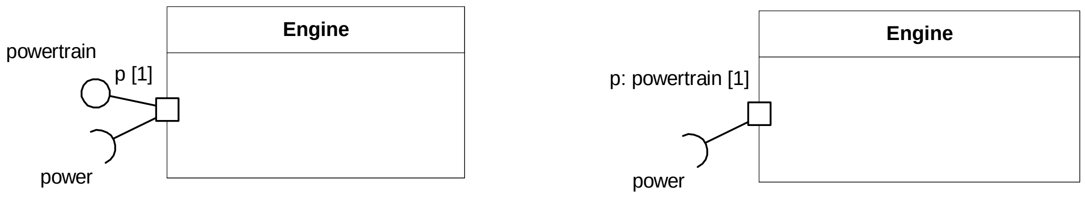


Figure 9.16 - Port notation

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 classifier. (The small state symbol indicates the behavior of the containing classifier.) Figure 9.17 shows the behavior port *p*, as indicated by its connection to the state symbol representing the behavior of the *Engine* class. Its provided interface is *powertrain*. In addition, a required interface, *power*, is shown also.

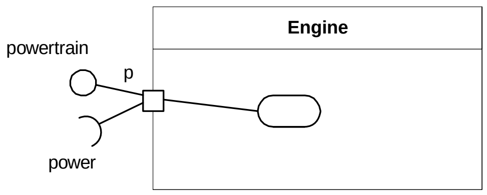


Figure 9.17 - Behavior port notation

Presentation Options

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 with the interface icon, separated by commas. Figure 9.18 below shows a port *OnlineServices* on the *OrderProcess* class with two provided interfaces, *OrderEntry* and *Tracking,* as well as a required interface *Payment*.

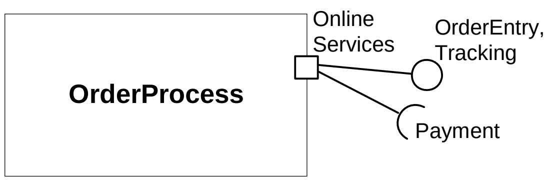


Figure 9.18 - Port notation showing multiple provided interfaces

Examples

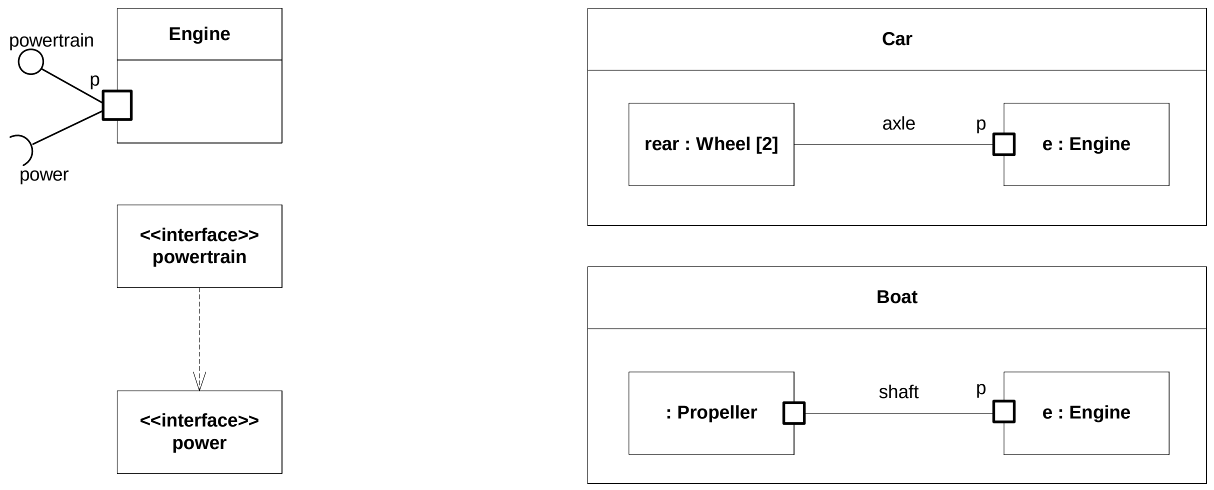


Figure 9.19 - Port examples

Figure 9.19 shows a class *Engine* with a port *p* with a provided interface *powertrain*. 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). The interface *power* is the required interface of the engine. The required interface specifies the services that the engine expects its environment to provide. At port *p*, the *Engine* class is completely encapsulated; it can be specified without any knowledge of the environment the engine will be embedded in. As long as the environment obeys the constraints expressed by the provided and required interfaces of the engine, the engine will function properly.

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 set 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 the provided and required interfaces, the engine will function as specified, whether it is an engine of a car or an engine of a boat. (This example also shows that connectors need not necessarily attach to parts via ports (as shown in the *Car* class.)

Rationale

The required and provided interfaces of a port specify everything that is necessary for interactions through that interaction point. If all interactions of a classifier with its environment are achieved through ports, then the internals of the classifier are fully isolated from the environment. This allows such a classifier to be used in any context that satisfies the constraints specified by its ports.

Changes from previous UML

This metaclass has been added to UML.

9.3.12 Property (from InternalStructures)

Generalizations

- "Property (from Kernel, AssociationClasses)" on page 122 *(merge increment)*
- "ConnectableElement (from InternalStructures)" on page 174

Description

A property represents a set of instances that are owned by a containing classifier instance.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

When an instance of the containing classifier is created, a set of instances corresponding to its properties may be created either immediately or at some later time. These instances are instances of the classifier typing the property. A property specifies that a set of instances may exist; this set of instances is a subset of the total set of instances specified by the classifier typing the property.

A part declares that an instance of this classifier may contain a set of instances by composition. All such instances are destroyed when the containing classifier instance is destroyed. Figure 9.20 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).

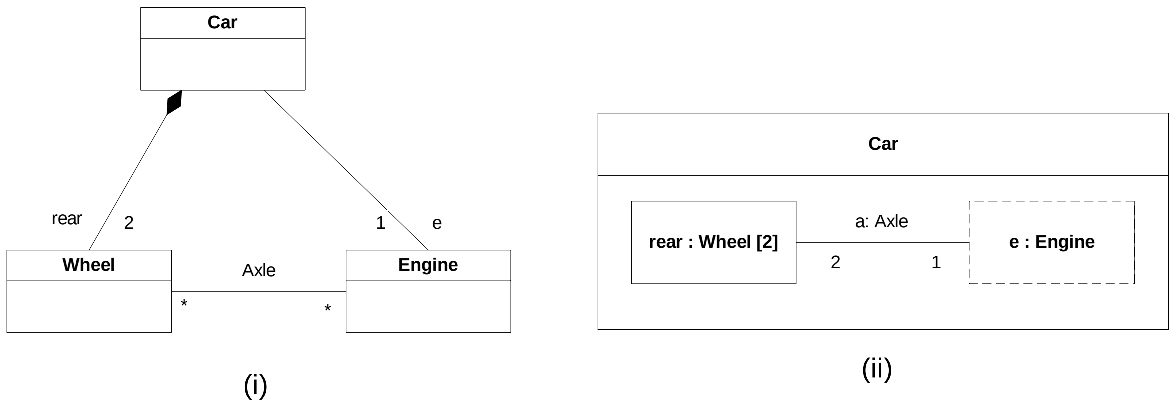


Figure 9.20 - Properties

Notation

A part is shown by graphical nesting of a box symbol with a solid outline representing the part within the symbol representing the containing classifier in a separate compartment. A property specifying an instance that is not owned by composition by the instance of the containing classifier is shown by graphical nesting of a box symbol with a dashed outline.

The contained box symbol has only a name compartment, which contains a string according to the syntax defined in the Notation sub clause of "Property (from Kernel, AssociationClasses)" on page 122. Detail may be shown within the box symbol indicating specific values for properties of the type classifier when instances corresponding to the property symbol are created.

Presentation Options

The multiplicity for a property may also be shown as a multiplicity mark in the top right corner of the part box.

A property symbol 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 class. The part has this anonymous class as its type. Every occurrence of an anonymous class is different from any other occurrence. The anonymously defined class has the properties specified with the part symbol. It is allowed to show compartments defining attributes and operations of the anonymously named class.

Examples

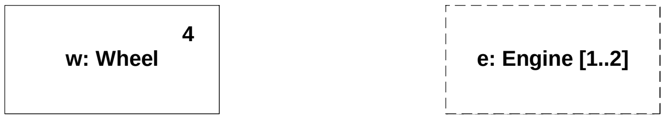


Figure 9.21 - Property examples

Figure 9.21 shows examples of properties. On the left, the property denotes that the containing instance will own four instances of the *Wheel* class by composition. The multiplicity is shown using the presentation option discussed above. The property on the right denotes that the containing instance will reference one or two instances of the *Engine* class. For additional examples, see 9.3.13, “StructuredClassifier (from InternalStructures),” on page 186.

Changes from previous UML

A connectable element used in a collaboration subsumes the concept of ClassifierRole.

9.3.13 StructuredClassifier (from InternalStructures)

Generalizations

- “Classifier (from Kernel, Dependencies, PowerTypes)” on page 52

Description

A structured classifier is an abstract metaclass that represents any classifier whose behavior can be fully or partly described by the collaboration of owned or referenced instances.

Attributes

No additional attributes

Associations

- /role: ConnectableElement [0..*]
  References the roles that instances may play in this classifier. (Abstract union; subsets *Classifier::feature*)
- ownedAttribute: Property [0..*]
  References the properties owned by the classifier. (Subsets *StructuredClassifier::role, Classifier.attribute,* and *Namespace::ownedMember*)
- /part: Property [0..*]
  References the properties specifying instances that the classifier owns by composition. This association is derived, selecting those owned properties where *isComposite* is *true*.
- ownedConnector: Connector [0..*]
  References the connectors owned by the classifier. (Subsets *Classifier::feature* and *Namespace::ownedMember*)

Constraints

[1] The multiplicities on connected elements must be consistent.

Semantics

The multiplicities on the structural features and connector ends indicate the number of instances (objects and links) that may be created within an instance of the containing classifier, either when the instance of the containing classifier is created, or in the case of links, when an object is added as the value of a role, or at a later time. The lower bound of the multiplicity range indicates the number of instances that are created (unless indicated differently by an associated instance specification or an invoked constructor function); the upper bound of the multiplicity range indicates the maximum number of instances that may be created. The slots corresponding to the structural features are initialized with these instances.

The manner of creation of the containing classifier may override the default instantiation. When an instance specification is used to specify the initial instance to be created for a classifier (see "Class" on page 166), the multiplicities of its parts determine the number of initial instances that will be created within that classifier. Initially, there will be as many instances held in slots as indicated by the corresponding multiplicity. Multiplicity ranges on such instance specifications may not contain upper bounds.

All instances corresponding to parts of a structured classifier are destroyed recursively, when an instance of that structured classifier is deleted. The instance is removed from the extent of its classifier, and is itself destroyed.

When an instance is removed from a role of a composite object, links that exist due to connectors between that role and others are destroyed.

Semantic Variation Points

The rules for matching the multiplicities of connector ends and those of parts and ports they interconnect are a semantic variation point. Also, the specific topology that results from such multi-connectors will differ from system to system. One possible approach to this is illustrated in Figure 9.22 and Figure 9.23.

For each instance playing a role in an internal structure, there will initially be as many links as indicated by the multiplicity of the opposite ends of connectors attached to that role (see "ConnectorEnd" on page 176 for the semantics where no multiplicities are given for an end). If the multiplicities of the ends match the multiplicities of the roles they are attached to (see Figure 9.22 i), the initial configuration that will be created when an instance of the containing classifier 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, Figure 9.22 ii).

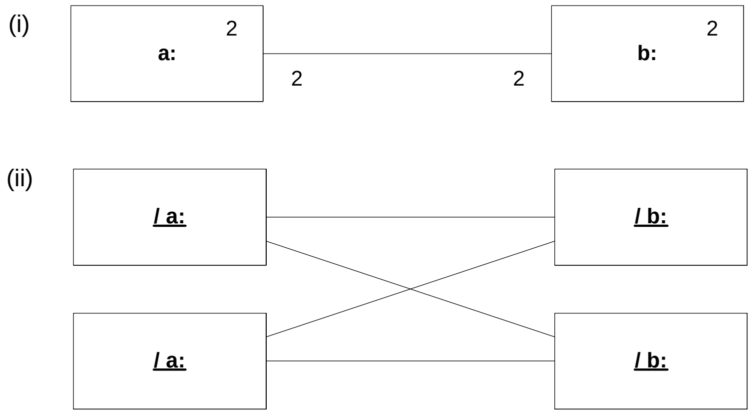


Figure 9.22 - "Star" connector pattern

Multiplicities on connector ends serve to restrict the number of initial links created. Links will be created for each instance playing the connected roles according to their ordering until the minimum connector end multiplicity is reached for both ends of the connector (see the resultant instance, Figure 9.23 ii). In this example, only two links are created, resulting in an array pattern.

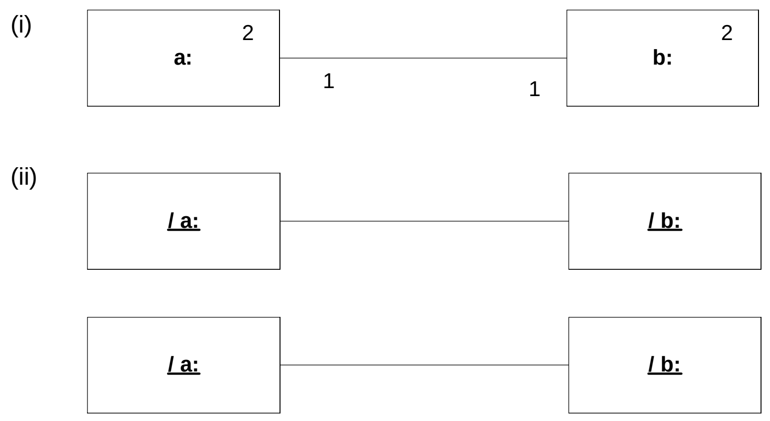


Figure 9.23 - "Array" connector pattern

Notation

The namestring of a role in an instance specification obeys the following syntax:

{<name> ['/' <rolename>] | '/' <rolename>} [':' <classifiername> [',' <classifiername>]]*

The name of the instance specification 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.

Examples

The following example shows 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 a connector representing the front and rear axle, respectively. An implicit association is defined as the type of each axle with each end typed by the *Wheel* class. Figure 9.24 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.

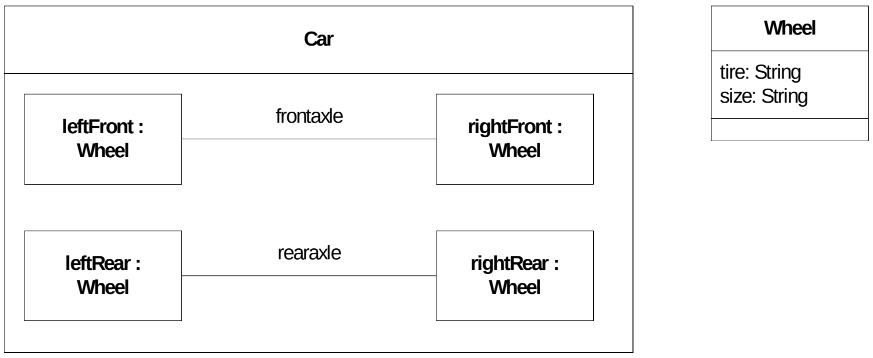


Figure 9.24 - Connectors and parts in a structure diagram

Figure 9.25 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 two instances of the left wheel and two instances of the right wheel (as no multiplicity is specified for the connector at the right wheel, the multiplicity is taken from the attached role), with each matching instance connected by a link deriving from the connector representing the axle. As specified by the multiplicities, no additional instances of the *Wheel* class can be added as left or right parts for a *Car* instance.

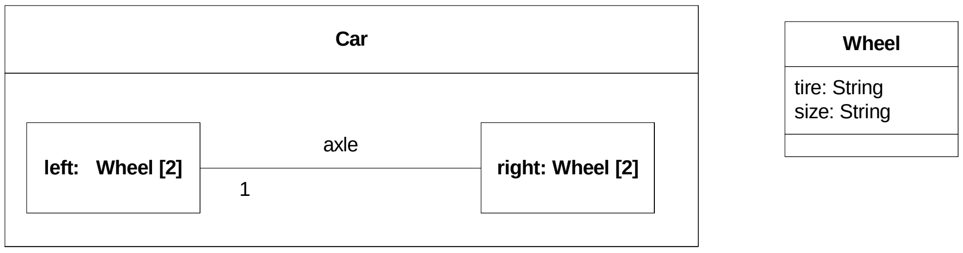


Figure 9.25 - Connectors and parts in a structure diagram using multiplicities

Figure 9.26 shows an instance of the *Car* class (as specified in Figure 9.24). 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.

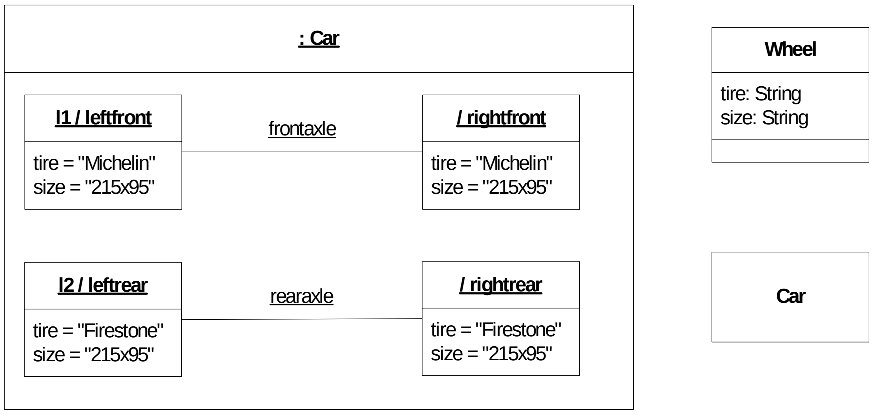


Figure 9.26 - An instance of the Car class

Finally, Figure 9.27 shows a constructor for the *Car* class (see “Class” on page 166). 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.

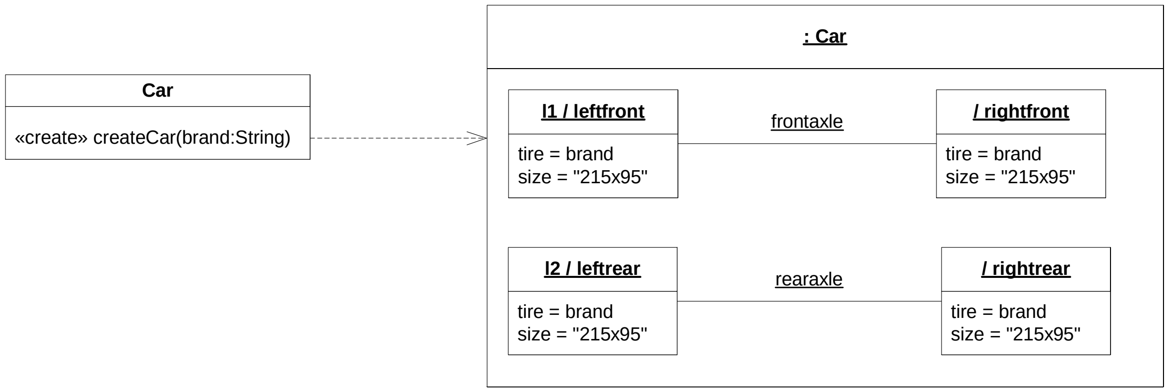


Figure 9.27 - A constructor for the Car class

9.3.14 Trigger (from InvocationActions)

Generalizations

- “Trigger (from Communications)” on page 456 *(merge increment)*

Description

A trigger specification may be qualified by the port on which the event occurred.

Associations

- port: Port [*]
  Specifies the ports at which a communication that caused an event may have arrived.

Semantics

Specifying one or more ports for an event implies that the event triggers the execution of an associated behavior only if the event was received via one of the specified ports.

Notation

The ports of a trigger are specified following a trigger signature by a list of port names separated by comma, preceded by the keyword «from»:

*‘«from»’ <port-name> [‘,’ <port-name>]**

9.3.15 Variable (from StructuredActivities)

Generalizations

- “Variable (from StructuredActivities)” on page 414 *(merge increment)*

Description

A variable is considered a connectable element.

Semantics

Extends variable to specialize connectable element.

9.4 Diagrams

Composite structure diagram

A composite structure diagram depicts the internal structure of a classifier, as well as the use of a collaboration in a collaboration use.

Graphical nodes

Additional graphical nodes that can be included in composite structure diagrams are shown in Table 9.1.

Table 9.1 - Graphic nodes included in composite structure diagrams

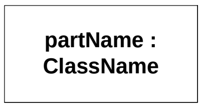
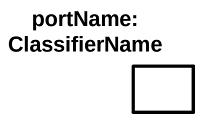
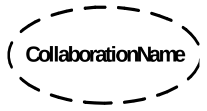
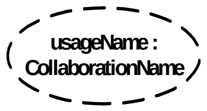
Node Type	Notation	Reference
Part		See “Property (from InternalStructures)” on page 183.

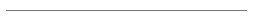
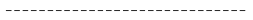
Table 9.1 - Graphic nodes included in composite structure diagrams

Node Type	Notation	Reference
Port		See "Ports" on page 179. A port may appear either on a contained part representing a port on that part, or on the boundary of the class diagram, representing a port on the represented classifier itself. The optional *ClassifierName* is only used if it is desired to specify a class of an object that implements the port.
Collaboration		See "Collaboration" on page 168.
CollaborationUse		See "CollaborationUse (from Collaborations)" on page 171.

Graphical paths

Additional graphical paths that can be included in composite structure diagrams are shown in Table 9.2.

Table 9.2 - Graphic nodes included in composite structure diagrams

Path Type	Notation	Reference
Connector		See "Connector" on page 174.
Role binding		See "CollaborationUse (from Collaborations)" on page 171.

Structure diagram

All graphical nodes and paths shown on composite structure diagrams can also be shown on other structure diagrams.

10 Deployments

10.1 Overview

The Deployments package specifies a set of constructs that can be used to define the execution architecture of systems that represent the assignment of software artifacts to nodes. Nodes are connected through communication paths to create network systems of arbitrary complexity. Nodes are typically defined in a nested manner, and represent either hardware devices or software execution environments. Artifacts represent concrete elements in the physical world that are the result of a development process.

The Deployment package supports a streamlined model of deployment that is deemed sufficient for the majority of modern applications. Where more elaborate deployment models are required, it can be extended through profiles or meta models to model specific hardware and software environments.

Artifacts

The Artifacts package defines the basic Artifact construct as a special kind of Classifier.

Nodes

The Nodes package defines the concept of Node, as well as the basic deployment relationship between Artifacts and Nodes.

Component Deployments

The ComponentDeployments package extends the basic deployment model with capabilities to support deployment mechanisms found in several common component technologies.

10.2 Abstract Syntax

Figure 10.1 shows the dependencies of the Deployments packages.

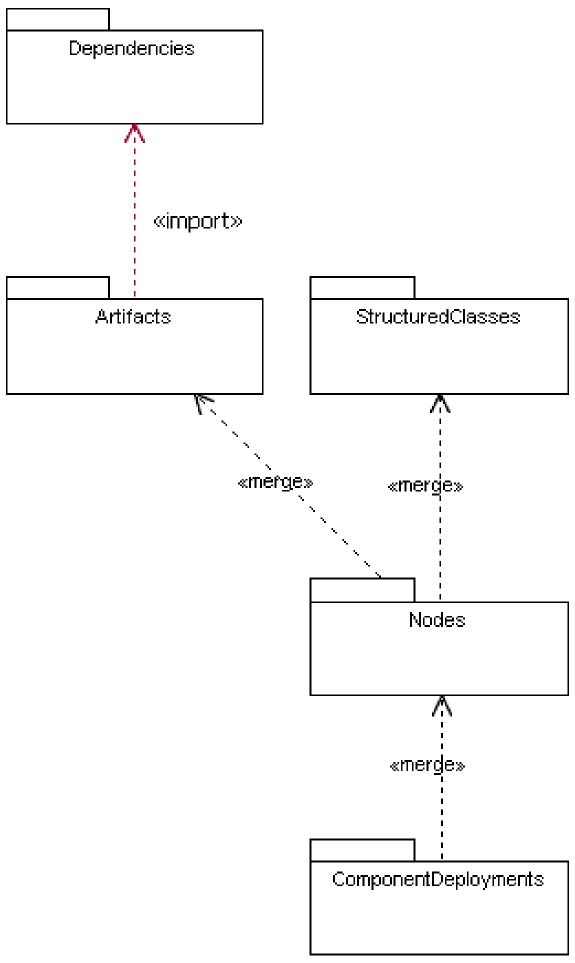


Figure 10.1 - Dependencies between packages described in this clause

Package Artifacts

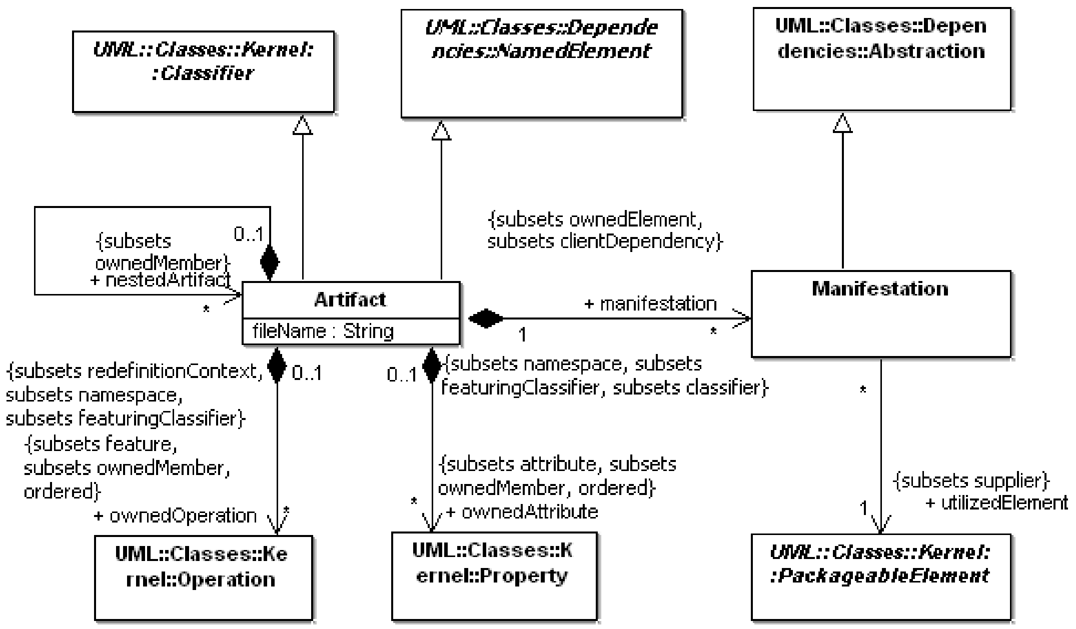

UML::Classes::Kernel::Classifier
UML::Classes::Dependencies::NamedElement
UML::Classes::Dependencies::Abstraction
{subsets ownedMember}
+ nestedArtifact
0..1
*
Artifact
fileName : String
{subsets ownedElement, subsets clientDependency}
+ manifestation
1
*
Manifestation
{subsets redefinitionContext, subsets namespace, subsets featuringClassifier}
0..1
{subsets feature, subsets ownedMember, ordered}
+ ownedOperation
*
0..1
{subsets namespace, subsets featuringClassifier, subsets classifier}
{subsets attribute, subsets ownedMember, ordered}
+ ownedAttribute
*
{subsets supplier}
+ utilizedElement
1
UML::Classes::Kernel::Operation
UML::Classes::Kernel::Property
UML::Classes::Kernel::PackageableElement

Figure 10.2 - The elements defined in the Artifacts package

Package Nodes

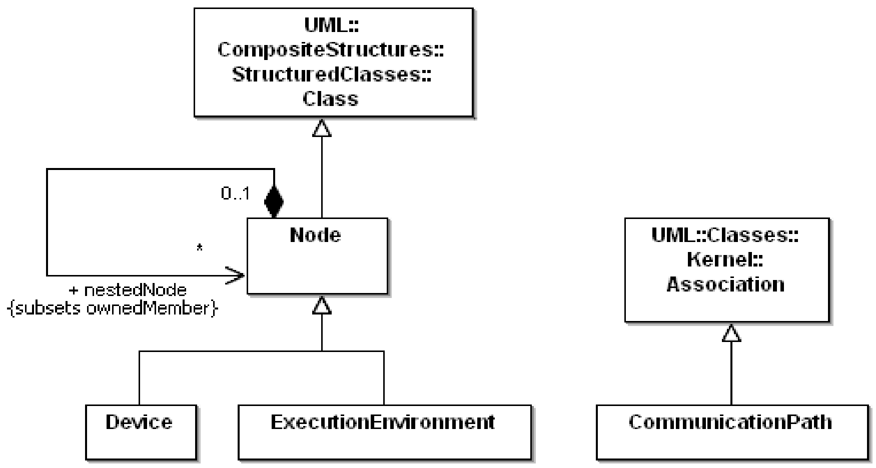


Figure 10.3 - The definition of the Node concept

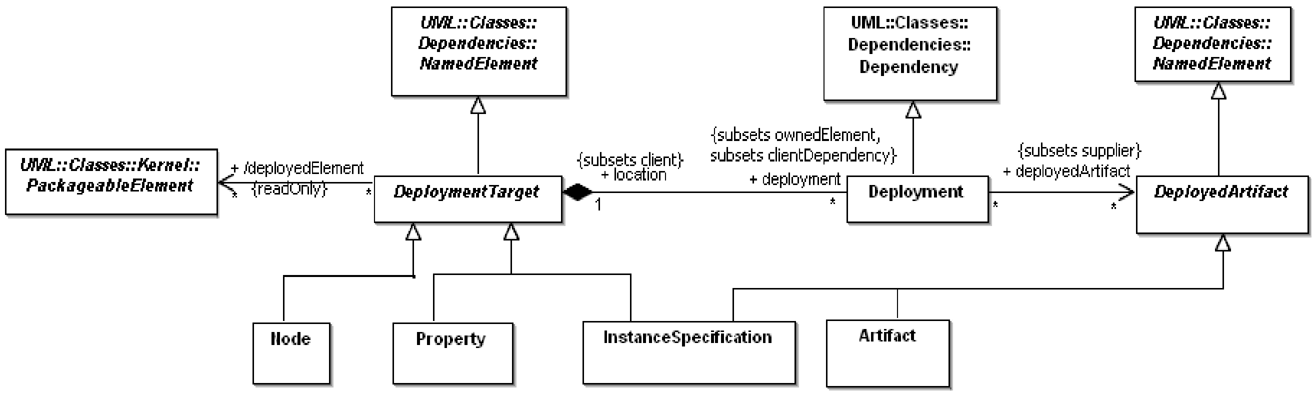


Figure 10.4 - Definition of the Deployment relationship between DeploymentTargets and DeployedArtifacts

Package ComponentDeployments

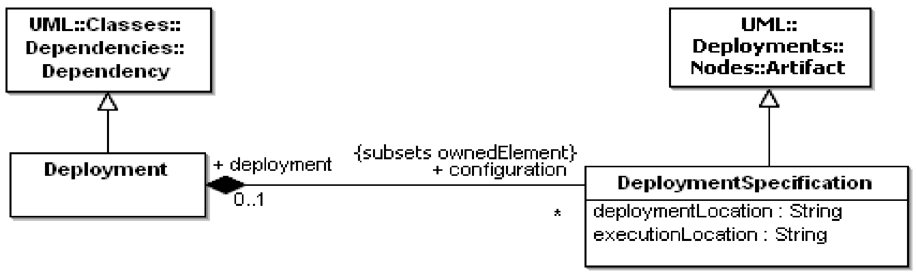


Figure 10.5 - Metaclasses that define component Deployment

10.3 Class Descriptions

10.3.1 Artifact (from Artifacts, Nodes)

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.

Generalizations

- "Classifier (from Kernel, Dependencies, PowerTypes)" on page 52
- "DeployedArtifact (from Nodes)" on page 200
- "NamedElement (from Kernel, Dependencies)" on page 98

Description

Package Artifacts

In the metamodel, an Artifact is a Classifier that represents a physical entity. Artifacts may have Properties that represent features of the Artifact, and Operations that can be performed on its instances. Artifacts can be involved in Associations to other Artifacts (e.g., composition associations). Artifacts can be instantiated to represent detailed copy semantics, where different instances of the same Artifact may be deployed to various Node instances (and each may have separate property values, e.g., for a 'time-stamp' property).

Package Node

As part of the Nodes package, an Artifact is extended to become the source of a deployment to a Node. This is achieved by specializing the abstract superclass DeployedArtifact defined in the Nodes package.

Attributes

Package Artifacts

- 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.

Associations

Package Artifacts

- nestedArtifact: 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 [*]
  The attributes or association ends defined for the Artifact. {Subsets *Namespace::ownedMember*}
- ownedOperation : Operation [*]
  The Operations defined for the Artifact. {Subsets *Namespace::ownedMember*}
- manifestation : Manifestation [*]
  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. {Subsets *NamedElement::clientDependency*, Subsets *Element::ownedElement*}

Constraints

No additional constraints

Semantics

An Artifact defined by the user represents a concrete element in the physical world. A particular instance (or ‘copy’) of an artifact is deployed to a node instance. Artifacts may have composition associations to other artifacts that are nested within it. For instance, a deployment descriptor artifact for a component may be contained within the artifact that implements that component. In that way, the component and its descriptor are deployed to a node instance as one artifact instance.

Specific profiles are expected to stereotype artifact to model sets of files (e.g., as characterized by a ‘file extension’ on a file system). The UML Standard Profile defines several standard stereotypes that apply to Artifacts, e.g., «source» or «executable» (See Annex C - Standard Stereotypes). These stereotypes can be further specialized into implementation and platform specific stereotypes in profiles. For example, an EJB profile might define «jar» as a subclass of «executable» for executable Java archives.

Notation

An artifact is presented using an ordinary class rectangle with the key-word «artifact». Alternatively, it may be depicted by an icon.

Optionally, the underlining of the name of an artifact instance may be omitted, as the context is assumed to be known to users.

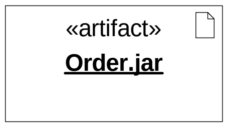


Figure 10.6 - An Artifact instance

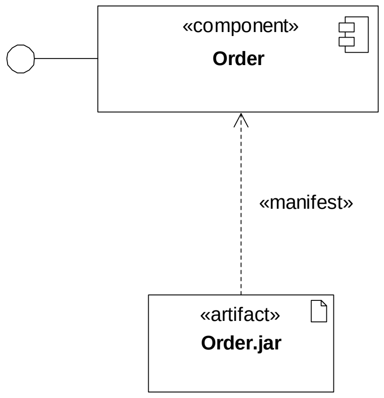


Figure 10.7 - A visual representation of the manifestation relationship between artifacts and components

Changes from previous UML

The following changes from UML 1.x have been made: Artifacts can now manifest any PackageableElement (not just Components, as in UML 1.x).

10.3.2 CommunicationPath (from Nodes)

A communication path is an association between two DeploymentTargets, through which they are able to exchange signals and messages.

Generalizations

- “Association (from Kernel)” on page 39

Description

In the metamodel, CommunicationPath is a subclass of Association.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] The association ends of a CommunicationPath are typed by DeploymentTargets.

```
self.endType->forAll (t | t.oclIsKindOf(DeploymentTarget))
```

Semantics

A communication path is an association that can only be defined between deployment targets, to model the exchange of signals and messages between them.

Notation

No additional notation

Changes from previous UML

The following changes from UML 1.x have been made: CommunicationPath was implicit in UML 1.x. It has been made explicit to formalize the modeling of networks of complex Nodes.

10.3.3 DeployedArtifact (from Nodes)

A deployed artifact is an artifact or artifact instance that has been deployed to a deployment target.

Generalizations

- “NamedElement (from Kernel, Dependencies)” on page 98

Description

In the metamodel, DeployedArtifact is an abstract metaclass that is a specialization of NamedElement. A DeployedArtifact is involved in one or more Deployments to a DeploymentTarget.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Deployed artifacts are deployed to a deployment target.

Notation

No additional notation

Changes from previous UML

The following changes from UML 1.x have been made: The capability to deploy artifacts and artifact instances to nodes has been made explicit based on UML 2.0 instance modeling through the addition of this abstract metaclass.

10.3.4 Deployment (from ComponentDeployments, Nodes)

Package Nodes

A deployment is the allocation of an artifact or artifact instance to a deployment target.

Package ComponentDeployments

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.

Generalizations

- "Dependency (from Dependencies)" on page 62

Description

In the metamodel, Deployment is a subtype of Dependency.

Attribute

No additional attributes

Associations

Package Nodes

- deployedArtifact : Artifact [*]
  The Artifacts that are deployed onto a Node. This association specializes the supplier association.
- location : DeploymentTarget [1]
  The DeploymentTarget that is the target of a Deployment. This association specializes the client association.

Package ComponentDeployments

- configuration : DeploymentSpecification [*]
  The specification of properties that parameterize the deployment and execution of one or more Artifacts. This association is specialized from the ownedMember association.

Constraints

No additional constraints

Semantics

The deployment relationship between a DeployedArtifact and a DeploymentTarget can be defined at the "type" level and at the "instance level." For example, a 'type level' deployment relationship can be defined between an "application server" Node and an "order entry request handler" executable Artifact. At the 'instance level' 3 specific instances "app-server1" ... "app-server3" may be the deployment target for six "request handler*" instances. Finally, for modeling complex deployment target models consisting of nodes with a composite structure defined through 'parts,' a Property (that functions as a part) may also be the target of a deployment.

Notation

Deployment diagrams show the allocation of Artifacts to Nodes according to the Deployments defined between them.

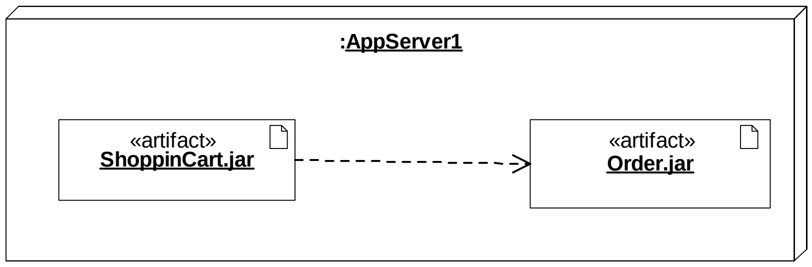


Figure 10.8 - A visual representation of the deployment location of artifacts (including a dependency between the artifacts).

An alternative notation to containing the deployed artifacts within a deployment target symbol is to use a dependency labeled «deploy» that is drawn from the artifact to the deployment target.

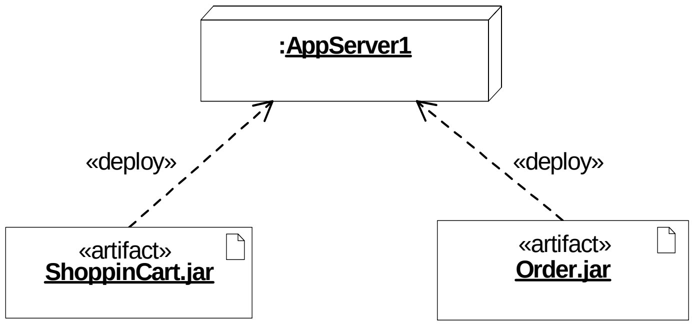


Figure 10.9 - Alternative deployment representation of using a dependency called «deploy»

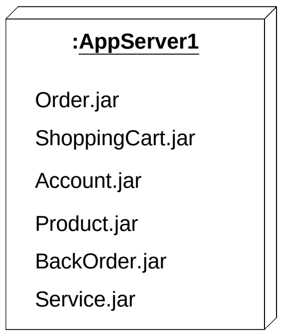


Figure 10.10 - Textual list based representation of the deployment location of artifacts

Changes from previous UML

The following changes from UML 1.x have been made — an association to DeploymentSpecification has been added.

10.3.5 DeploymentSpecification (from ComponentDeployments)

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.

Generalizations

- "Artifact (from Artifacts, Nodes)" on page 197

Description

In the metamodel, a DeploymentSpecification is a subtype of Artifact. It defines a set of deployment properties that are specific to a certain Container type. An instance of a DeploymentSpecification with specific values for these properties may be contained in a complex Artifact.

Attributes

ComponentDeployments Package

- 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.

Associations

ComponentDeployments Package

- deployment : Deployment [0..1]
  The deployment with which the DeploymentSpecification is associated.

Constraints

[1] The DeploymentTarget of a DeploymentSpecification is a kind of ExecutionEnvironment.

self.deployment->forAll (d | d.location..oclIsKindOf(ExecutionEnvironment))

[2] 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).

self.deployment->forAll (d | d.location.deployedElements->forAll (de | de.oclIsKindOf(Component)))

Semantics

A Deployment specification is a general mechanism to parameterize a Deployment relationship, as is common in various hardware and software technologies. The deployment specification element is expected to be extended in specific component profiles. Non-normative examples of the standard stereotypes that a profile might add to deployment specification are, for example, «concurrencyMode» with tagged values {thread, process, none}, or «transactionMode» with tagged values {transaction, nestedTransaction, none}.

Notation

A DeploymentSpecification is graphically displayed as a classifier rectangle (Figure 10.11) attached to a component artifact deployed on a container using a regular dependency arrow.

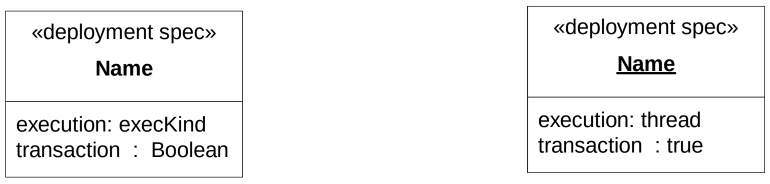


Figure 10.11 - DeploymentSpecification for an artifact (specification and instance levels)

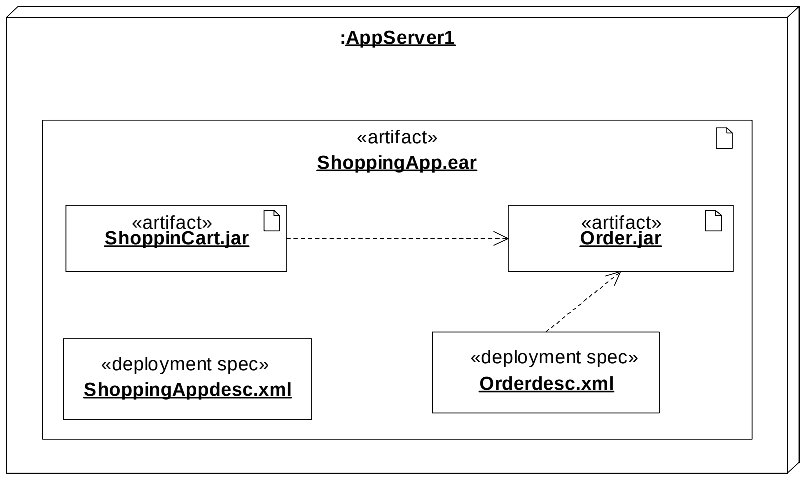


Figure 10.12 - DeploymentSpecifications related to the artifacts that they parameterize

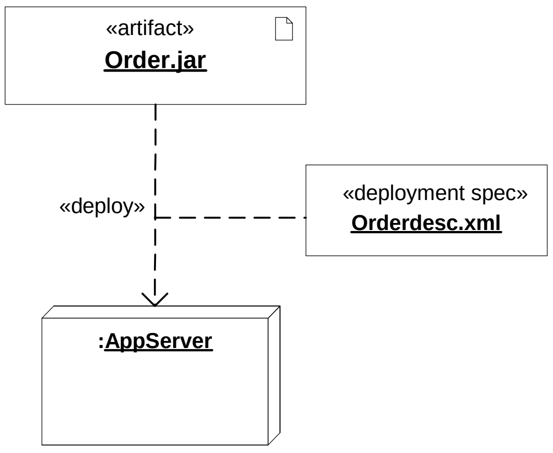


Figure 10.13 - A DeploymentSpecification for an artifact

Changes from previous UML

The following changes from UML 1.x have been made — DeploymentSpecification does not exist in UML 1.x.

10.3.6 DeploymentTarget (from Nodes)

A deployment target is the location for a deployed artifact.

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 98

Description

In the metamodel, DeploymentTarget is an abstract metaclass that is a specialization of NamedElement. A DeploymentTarget owns a set of Deployments.

Attributes

No additional attributes

Associations

Nodes Package

- deployment : Deployment [*]
  The set of Deployments for a DeploymentTarget. {Subsets *NamedElement::clientDependency*, Subsets *Element::ownedElement*}
- / deployedElement : PackageableElement [*]
  The set of elements that are manifested in an Artifact that is involved in Deployment to a DeploymentTarget. The association is a derived association.

context DeploymentTarget::deployedElement **derive**:
  ((self.deployment->collect(deployedArtifact))->collect(manifestation))->collect(utilizedElement)

Constraints

No additional constraints

Semantics

Artifacts are deployed to a deployment target. The deployment target owns the set of deployments that target it.

Notation

No additional notation

Changes from previous UML

The following changes from UML 1.x have been made: The capability to deploy artifacts and artifact instances to nodes has been made explicit based on UML 2.0 instance modeling.

10.3.7 Device (from Nodes)

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).

Generalizations

- “Node (from Nodes)” on page 210

Description

In the metamodel, a Device is a subclass of Node.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

A device may be a nested element, where a physical machine is decomposed into its elements, either through namespace ownership or through attributes that are typed by Devices.

Notation

A Device is notated by a perspective view of a cube tagged with the keyword «device».

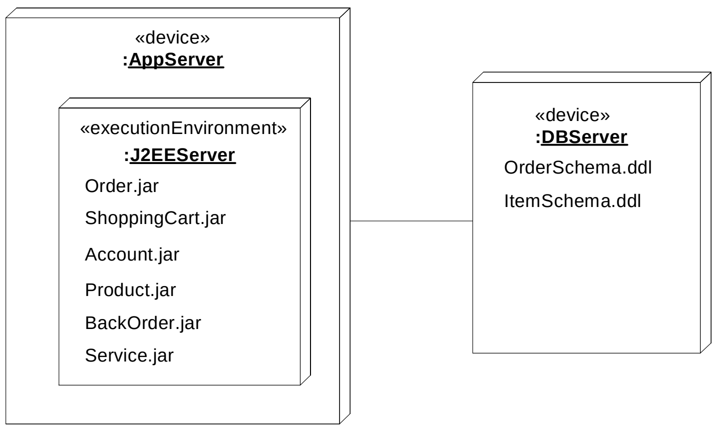


Figure 10.14 - Notation for a Device

Changes from previous UML

The following changes from UML 1.x have been made — Device is not defined in UML 1.x.

10.3.8 ExecutionEnvironment (from Nodes)

An ExecutionEnvironment is a node that offers an execution environment for specific types of components that are deployed on it in the form of executable artifacts.

Generalizations

- “Node (from Nodes)” on page 210

Description

In the metamodel, an ExecutionEnvironment is a subclass of Node. It is usually part of a general Node, representing the physical hardware environment on which the ExecutionEnvironment resides. In that environment, the ExecutionEnvironment implements a standard set of services that Components require at execution time (at the modeling level these services are usually implicit). For each component Deployment, aspects of these services may be determined by properties in a DeploymentSpecification for a particular kind of ExecutionEnvironment.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

ExecutionEnvironment instances are assigned to node instances by using composite associations between nodes and ExecutionEnvironments, where the ExecutionEnvironment plays the role of the part. ExecutionEnvironments can be nested (e.g., a database ExecutionEnvironment may be nested in an operating system ExecutionEnvironment). Components of the appropriate type are then deployed to specific ExecutionEnvironment nodes.

Typical examples of standard ExecutionEnvironments that specific profiles might define stereotypes for are «OS», «workflow engine», «database system», and «J2EE container».

An ExecutionEnvironment can optionally have an explicit interface of system level services that can be called by the deployed elements, in those cases where the modeler wants to make the ExecutionEnvironment software execution environment services explicit.

Notation

A ExecutionEnvironment is notated by a Node annotated with the stereotype «executionEnvironment».

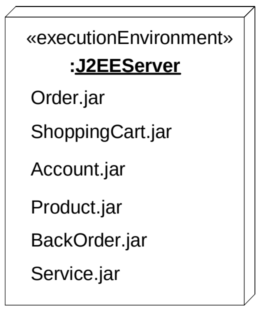


Figure 10.15 - Notation for a ExecutionEnvironment (example of an instance of a J2EEServer ExecutionEnvironment)

Changes from previous UML

The following changes from UML 1.x have been made — ExecutionEnvironment is not defined in UML 1.x.

10.3.9 InstanceSpecification (from Nodes)

An instance specification is extended with the capability of being a deployment target in a deployment relationship, in the case that it is an instance of a node. It is also extended with the capability of being a deployed artifact, if it is an instance of an artifact.

Generalizations

- "DeployedArtifact (from Nodes)" on page 200
- "DeploymentTarget (from Nodes)" on page 205
- "InstanceSpecification (from Kernel)" on page 82 *(merge increment)*

Description

In the metamodel, InstanceSpecification is a specialization of DeploymentTarget and DeployedArtifact.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] An InstanceSpecification can be a DeploymentTarget if it is the instance specification of a Node and functions as a part in the internal structure of an encompassing Node.

[2] An InstanceSpecification can be a DeployedArtifact if it is the instance specification of an Artifact.

Semantics

No additional semantics

Notation

An instance can be attached to a node using a deployment dependency, or it may be visually nested inside the node.

Changes from previous UML

The following changes from UML 1.x have been made — the capability to deploy artifact instances to node instances existed in UML 1.x, and has been made explicit based on UML 2.0 instance modeling.

10.3.10 Manifestation (from Artifacts)

A manifestation is the concrete physical rendering of one or more model elements by an artifact.

Generalizations

- “Abstraction (from Dependencies)” on page 38

Description

In the metamodel, a Manifestation is a subtype of Abstraction. A Manifestation is owned by an Artifact.

Attributes

No additional attributes

Associations

Artifacts

- utilizedElement : PackageableElement [1]
  The model element that is utilized in the manifestation in an Artifact. {Subsets *Dependency::supplier*}

Constraints

No additional associations

Semantics

An artifact embodies or manifests a number of model elements. The artifact owns the manifestations, each representing the utilization of a packageable element.

Specific profiles are expected to stereotype the manifestation relationship to indicate particular forms of manifestation. For example, <<tool generated>> and <<custom code>> might be two manifestations for different classes embodied in an artifact.

Notation

A manifestation is notated in the same way as an abstraction dependency, i.e., as a general dashed line with an open arrow-head labeled with the keyword <<manifest>>.

Changes from previous UML

The following changes from UML 1.x have been made: Manifestation is defined as a meta association in UML 1.x, prohibiting stereotyping of manifestations. In UML 1.x, the concept of Manifestation was referred to as 'implementation' and annotated in the notation as <<implement>>. Since this was one of the many uses of the word 'implementation' this has been replaced by <<manifest>>.

10.3.11 Node (from Nodes)

A node is computational resource upon which artifacts may be deployed for execution.

Nodes can be interconnected through communication paths to define network structures.

Generalizations

- "Class (from StructuredClasses)" on page 166
- "DeploymentTarget (from Nodes)" on page 205

Description

In the metamodel, a Node is a subclass of Class. It is associated with a Deployment of an Artifact. It is also associated with a set of Elements that are deployed on it. This is a derived association in that these PackageableElements are involved in a Manifestation of an Artifact that is deployed on the Node. Nodes may have an internal structure defined in terms of parts and connectors associated with them for advanced modeling applications.

Attributes

No additional attributes

Associations

Nodes Package

- nestedNode : Node [*]
  The Nodes that are defined (nested) within the Node. {Subsets *Namespace::ownedMember}*

Constraints

[1] The internal structure of a Node (if defined) consists solely of parts of type Node.

Semantics

Nodes can be connected to represent a network topology by using communication paths. Communication paths can be defined between nodes such as "application server" and "client workstation" to define the possible communication paths between nodes. Specific network topologies can then be defined through links between node instances.

Hierarchical nodes (i.e., nodes within nodes) can be modeled using composition associations, or by defining an internal structure for advanced modeling applications.

Non-normative examples of nodes are «application server», «client workstation», «mobile device», «embedded device».

Notation

A node is shown as a figure that looks like a 3-dimensional view of a cube.

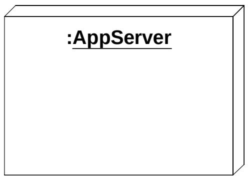


Figure 10.16 - An instance of a Node

Dashed arrows with the keyword «deploy» show the capability of a node type to support a component type. Alternatively, this may be shown by nesting component symbols inside the node symbol.

Nodes may be connected by associations to other nodes. A link between node instances indicates a communication path between the nodes.

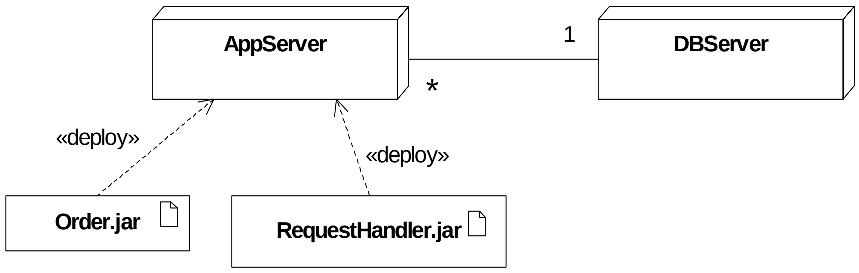


Figure 10.17 - Communication path between two Node types with deployed Artifacts

Artifacts may be contained within node instance symbols. This indicates that the items are deployed on the node instances.

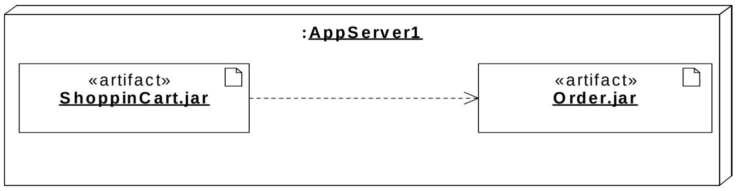


Figure 10.18 - A set of deployed component artifacts on a Node

Changes from previous UML

The following changes from UML 1.x have been made: to be written.

10.3.12 Property (from Nodes)

A Property is extended with 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.

Generalizations

- "Property (from InternalStructures)" on page 183 *(merge increment)*

Description

In the metamodel, Property is a specialization of DeploymentTarget.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] 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.

Semantics

No additional semantics

Notation

No additional notation

Changes from previous UML

The following changes from UML 1.x have been made — the capability to deploy to Nodes with an internal structure has been added to UML 2.0.

10.4 Diagrams

Deployment diagram

Graphical nodes

The graphic nodes that can be included in deployment diagrams are shown in Table 10.1.

Table 10.1 - Graphic nodes included in deployment diagrams

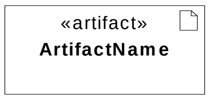
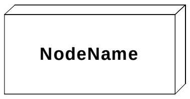
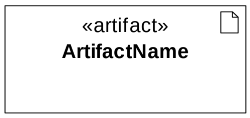
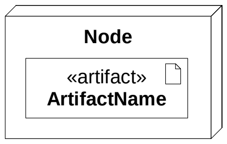
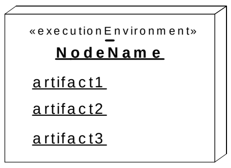
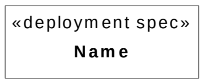
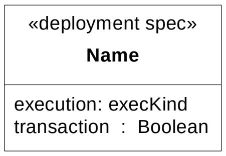
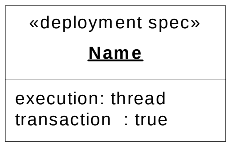
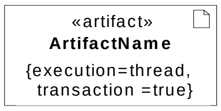
Node Type	Notation	Reference
Artifact		See "Artifact."
Node		See "Node."
Artifact deployed on Node		See "Deployment."
Node with deployed Artifacts		See "Deployment."
Node with deployed Artifacts	 «executionEnvironment» NodeName artifact1 artifact2 artifact3	See "Deployment" (alternative, textual notation).
Deployment specification		See "Deployment Specification."
Deployment specification - with properties	 «deployment spec» Name execution: execKind transaction : Boolean	See "Deployment Specification."

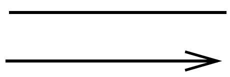
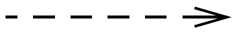
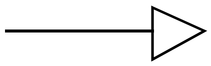
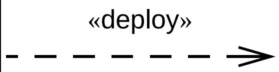
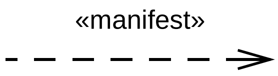
Table 10.1 - Graphic nodes included in deployment diagrams

Node Type	Notation	Reference
Deployment specification - with property values	 «deployment spec» Name execution: thread transaction : true	See “Deployment Specification.”
Artifact with annotated deployment properties	 «artifact» ArtifactName {execution=thread, transaction =true}	See “Artifact.”

Graphical paths

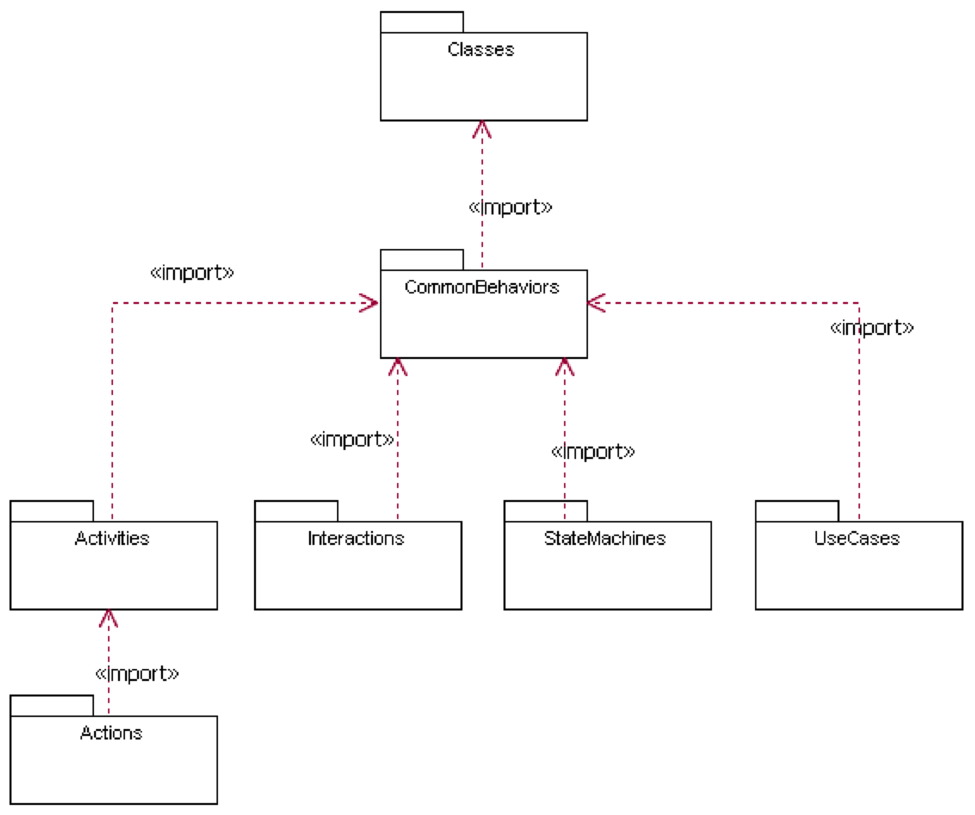
The graphic paths that can be included in deployment diagrams are shown in Table 10.2 .

Table 10.2 - Graphic nodes included in deployment diagrams

Path Type	Notation	Reference
Association		See “Association (from Kernel)” on page 39. Used to model communication paths between DeploymentTargets.
Dependency		See “Dependency (from Dependencies)” on page 62. Used to model general dependencies. In Deployment diagrams, this notation is used to depict the following metamodel associations: (i) the relationship between an Artifact and the model element(s) that it implements, and (ii) the deployment of an Artifact (instance) on a Node (instance).
Generalization		See “Generalization (from Kernel, PowerTypes)” on page 71.
Deployment		The Deployment relationship
Manifestation		The Manifestation relationship

Part II - Behavior

This part specifies the dynamic, behavioral constructs (e.g., activities, interactions, state machines) used in various behavioral diagrams, such as activity diagrams, sequence diagrams, and state machine diagrams. The UML packages that support behavioral modeling, along with the structure packages they depend upon (CompositeStructures and Classes) are shown in the figure below.



Part II, Figure 1 - UML packages that support behavioral modeling

The function and contents of these packages are described in following clauses, which are organized by major subject areas.

11 Actions

11.1 Overview

Basic Concepts

An action is the fundamental unit of behavior specification. An action takes a set of inputs and converts them into a set of outputs, though either or both sets may be empty. This clause defines semantics for a number of specialized actions, as described below. Some of the actions modify the state of the system in which the action executes. The values that are the inputs to an action may be described by value specifications, obtained from the output of actions that have one output (in StructuredActions), or in ways specific to the behaviors that use them. For example, the activity flow model supports providing inputs to actions from the outputs of other actions.

Actions are contained in behaviors, which provide their context. Behaviors provide constraints among actions to determine when they execute and what inputs they have. The Actions clause is concerned with the semantics of individual, primitive actions.

Basic actions include those that perform operation calls, signal sends, and direct behavior invocations. Operations are specified in the model and can be dynamically selected only through polymorphism. Signals are specified by a signal object, whose type represents the kind of message transmitted between objects, and can be dynamically created. Note that operations may be bound to activities, state machine transitions, or other behaviors. The receipt of signals may be bound to activities, state machine transitions, or other behaviors.

Intermediate Concepts

The intermediate level describes the various primitive actions. These primitive actions are defined in such a way as to enable the maximum range of 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 model, 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.

A surface action language would encompass both primitive actions and the control mechanisms provided by behaviors. In addition, a surface language may map higher-level constructs to the actions. For example, creating an object may involve initializing attribute values or creating objects for mandatory associations. The specification defines the create action to only create the object, and requires further actions to initialize attribute values and create objects for mandatory associations. A surface language could choose to define a creation operation with initialization as a single unit as a shorthand for several actions.

A particular surface language could implement each semantic construct one-to-one, or it could define higher-level, composite constructs to offer the modeler both power and convenience. This specification, then, expresses the fundamental semantics in terms of primitive behavioral concepts that are conceptually simple to implement. Modelers can work in terms of higher-level constructs as provided by their chosen surface language or notation.

The semantic primitives are defined to enable the construction of different execution engines, each of which may have different performance characteristics. A model compiler builder can optimize the structure of the software to meet specific performance requirements, so long as the semantic behavior of the specification and the implementation remain the same. For example, one engine might be fully sequential within a single task, while another may separate the classes into different processors based on potential overlapping of processing, and yet others may separate the classes in a client-server, or even a three-tier model.

The modeler can provide “hints” to the execution engine when the modeler has special knowledge of the domain solution that could be of value in optimizing the execution engine. For example, instances could—by design—be partitioned to match the distribution selected, so tests based on this partitioning can be optimized on each processor. The execution engines are not required to check or enforce such hints. An execution engine 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 an action violates aspects of static UML modeling that constrain runtime behavior, the semantics is left undefined. For example, attempting to create an instance of an abstract class is undefined - some languages may make this action illegal, others may create a partial instance for testing purposes. 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 no error or undefined semantics is implied. (Otherwise, it is impossible to use actions to pass through the intermediate configurations necessary to construct object configurations that satisfy multiplicity constraints.) The modeler must determine when minimum multiplicity should be enforced, and these points cannot be everywhere or the configuration cannot change.

Invocation Actions

More invocation actions are defined for broadcasting signals to the available “universe” and transmitting objects that are not signals.

Read Write Actions

Objects, structural features, links, and variables have values that are available to actions. Objects can be created and destroyed; structural features and variables have values; links can be created and destroyed, and can reference values through their ends; all of which are available to actions. Read actions get values, while write actions modify values and create and destroy objects and links. Read and write actions share the structures for identifying the structural features, links, and variables they access.

Object actions create and destroy objects. Structural feature actions support the reading and writing of structural features. The abstract metaclass StructuralFeatureAction statically specifies the structural feature being accessed. The object to access is specified dynamically, by referring to an input pin on which the object will be placed at runtime. The semantics for static features is undefined. Association actions operate on associations and links. In the description of these actions, the term “associations” does not include those modeled with association classes, unless specifically indicated. Similarly, a “link” is not a link object unless specifically indicated. The semantics of actions that read and write associations that have a static end is undefined.

Value specifications cover various expressions ranging from implementation-dependent constants to complex expressions, with side-effects. An action is defined for evaluating these. Also see “ValuePin (from BasicActions)” on page 289.

Complete Concepts

The major constructs associated with complete actions are outlined below.

Read Write Actions

Additional actions deal with the relation between object and class and link objects. These read the instances of a given classifier, check which classifier an instance is classified by, and change the classifier of an instance. Link object actions operate on instances of association classes. Also the reading and writing actions of associations are extended to support qualifiers.

Other Actions

Actions are defined for accepting events, including operation calls, and retrieving the property values of an object all at once. The StartClassifierBehaviorAction provides a way to indicate when the classifier behavior of a newly created object should begin to execute.

Structured Concepts

These actions operate in the context of activities and structured nodes. Variable actions support the reading and writing of variables. The abstract metaclass VariableAction statically specifies the variable being accessed. Variable actions can only access variables within the activity of which the action is a part. An action is defined for raising exceptions and a kind of input pin is defined for accepting the output of an action without using flows.

11.2 Abstract Syntax

The package dependencies of the Actions clause are shown in Figure 11.1.

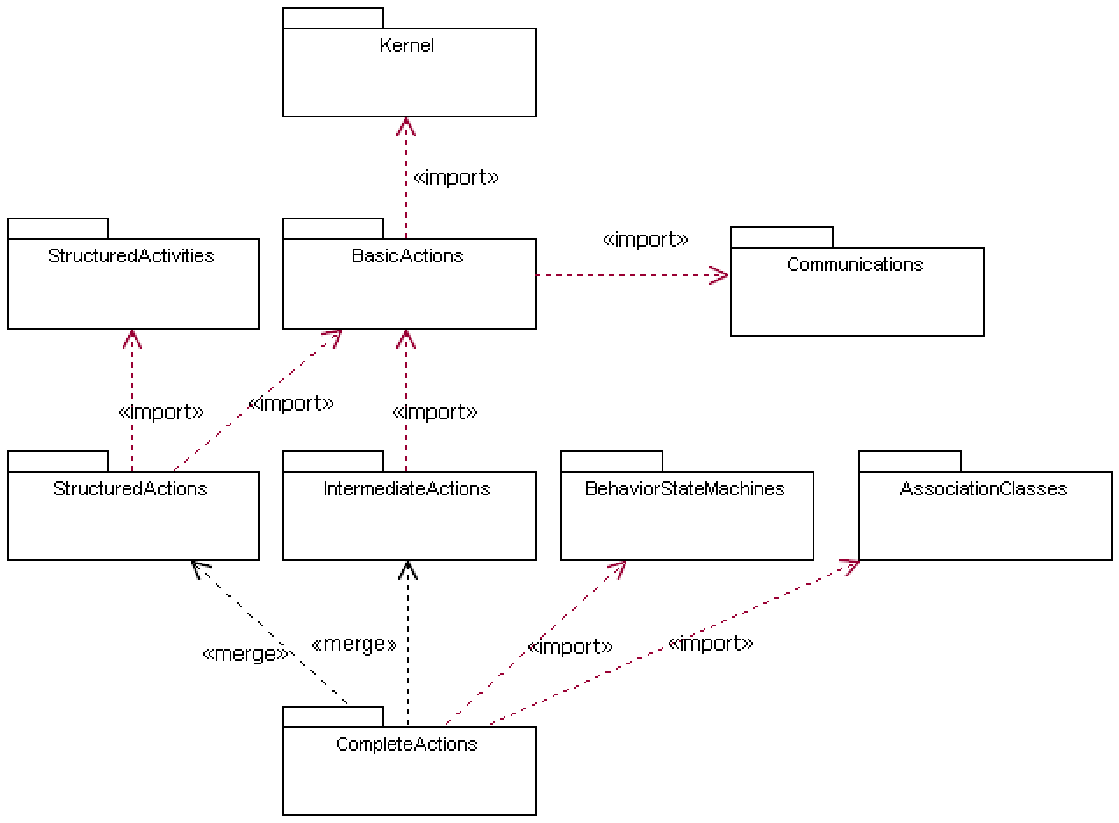


Figure 11.1 - Dependencies of the Action packages

Package BasicActions

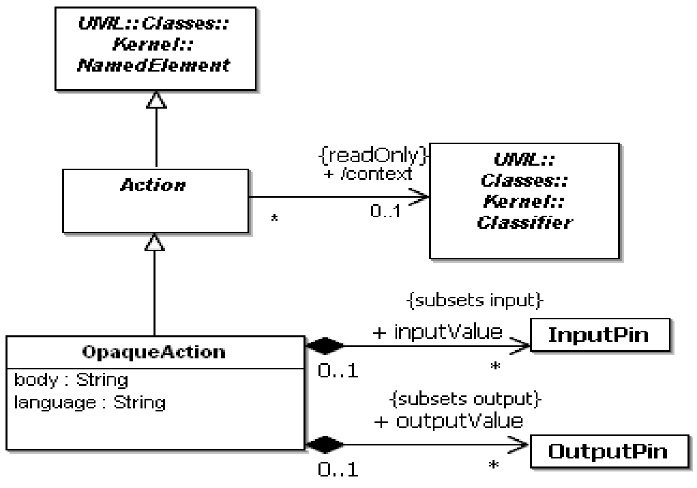


Figure 11.2 - Basic actions

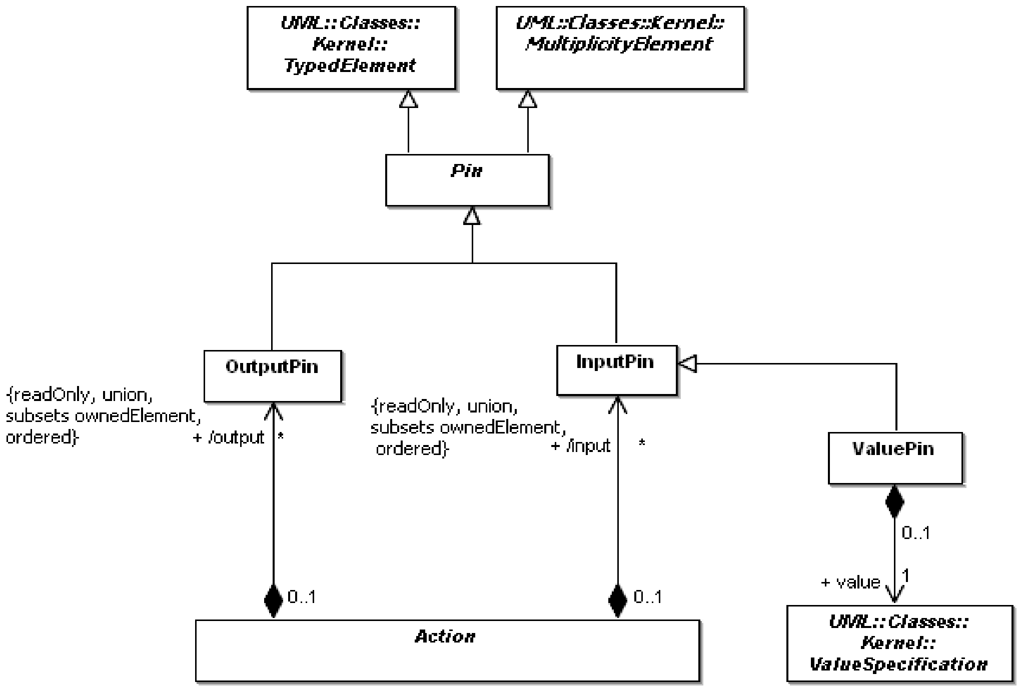


Figure 11.3 - Basic pins

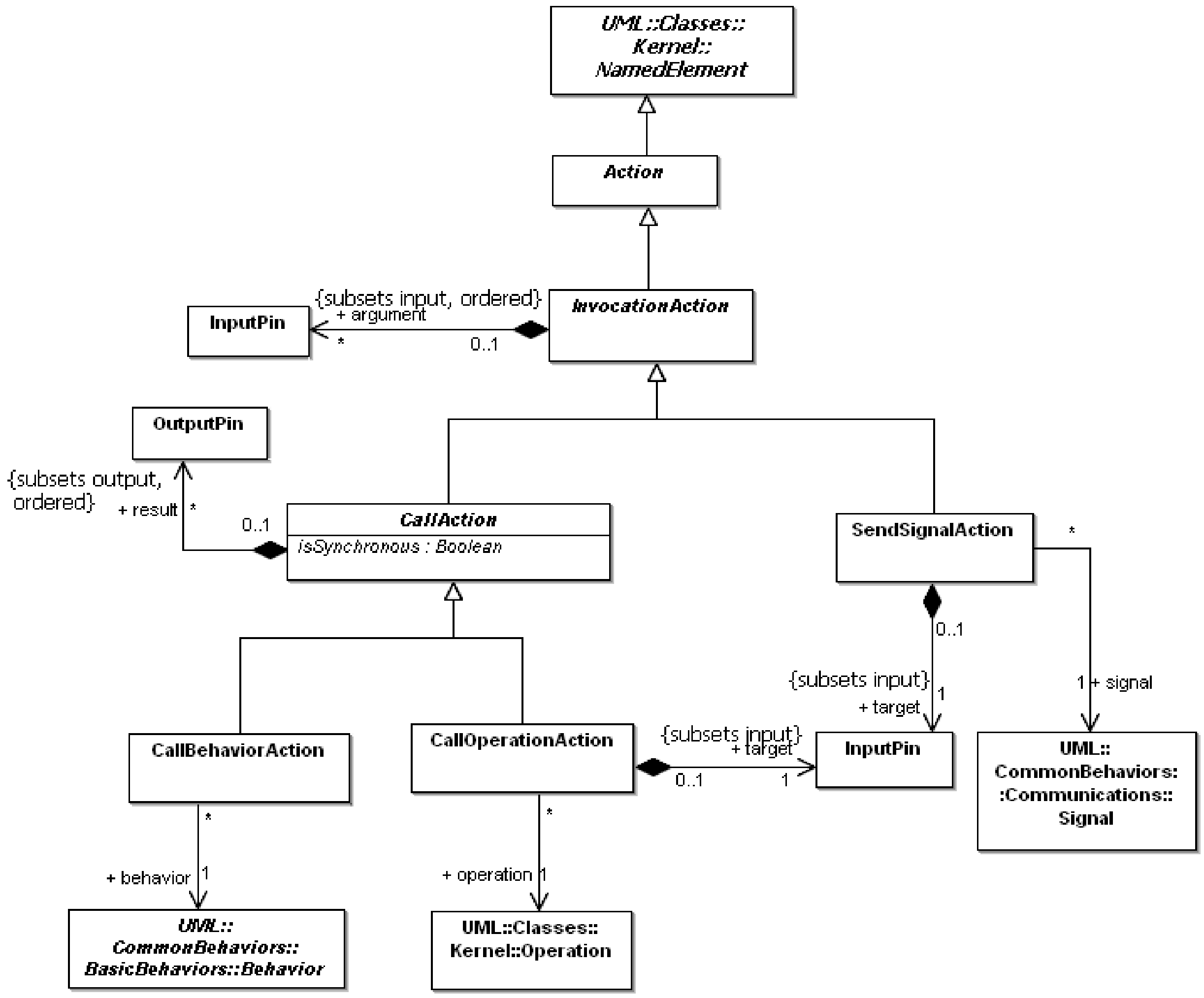


Figure 11.4 - Basic invocation actions

Package IntermediateActions

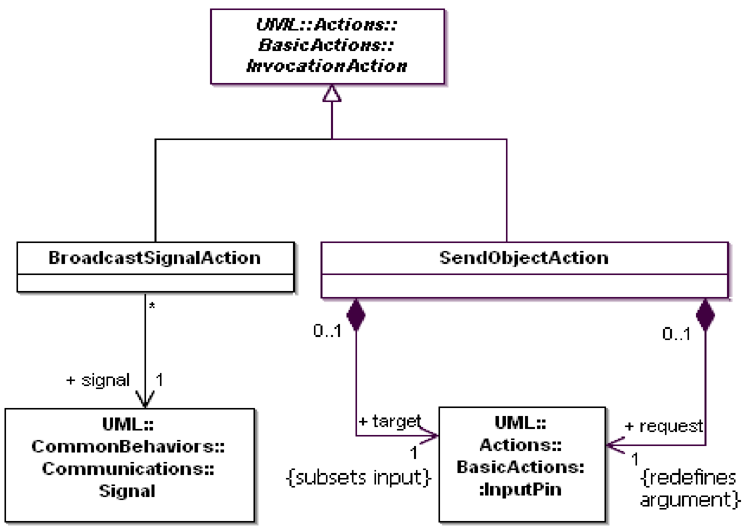
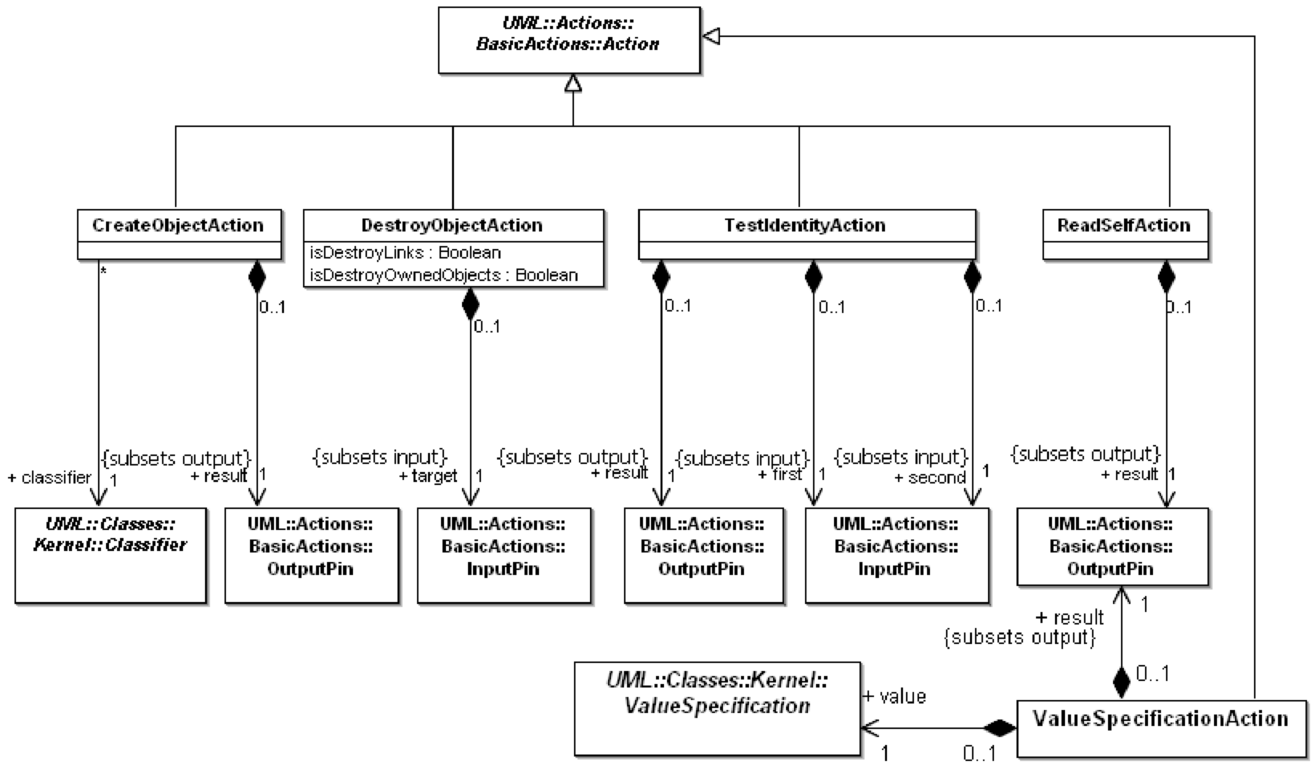


Figure 11.5 - Intermediate invocation actions



UML::Actions::BasicActions::Action
CreateObjectAction
DestroyObjectAction
isDestroyLinks : Boolean
isDestroyOwnedObjects : Boolean
TestIdentityAction
ReadSelfAction
UML::Classes::Kernel::Classifier
UML::Actions::BasicActions::OutputPin
UML::Actions::BasicActions::InputPin
UML::Classes::Kernel::ValueSpecification
ValueSpecificationAction
+ classifier
*
1
{subsets output}
+ result
0..1
{subsets input}
+ target
+ first
+ second
+ value

Figure 11.6 - Object actions

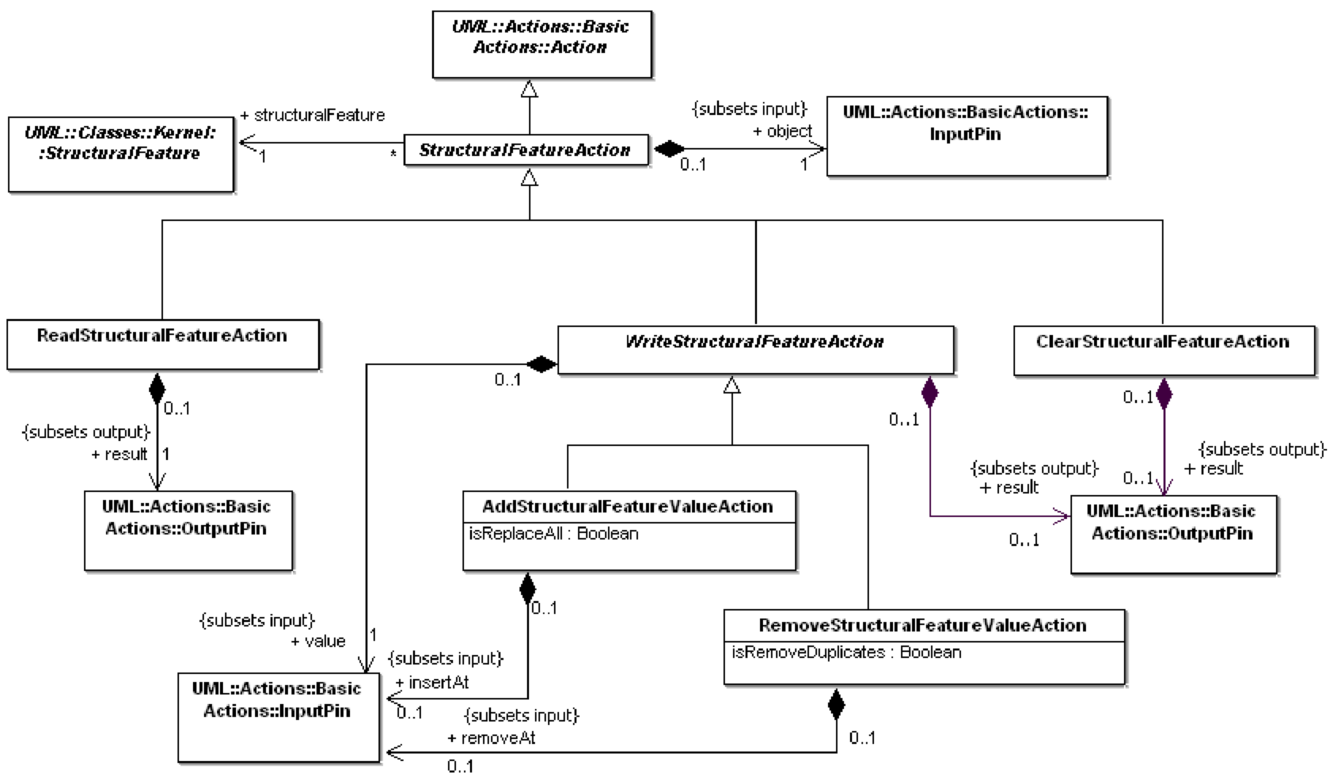

UML::Actions::Basic Actions::Action
UML::Classes::Kernel: :StructuralFeature
+ structuralFeature
1
*
StructuralFeatureAction
{subsets input}
+ object
0..1
1
UML::Actions::BasicActions:: InputPin
ReadStructuralFeatureAction
WriteStructuralFeatureAction
ClearStructuralFeatureAction
0..1
{subsets output}
+ result
1
UML::Actions::Basic Actions::OutputPin
AddStructuralFeatureValueAction
isReplaceAll : Boolean
RemoveStructuralFeatureValueAction
isRemoveDuplicates : Boolean
{subsets input}
+ value
{subsets input}
+ insertAt
{subsets input}
+ removeAt
UML::Actions::Basic Actions::InputPin
UML::Actions::Basic Actions::OutputPin

Figure 11.7 - Structural Feature Actions

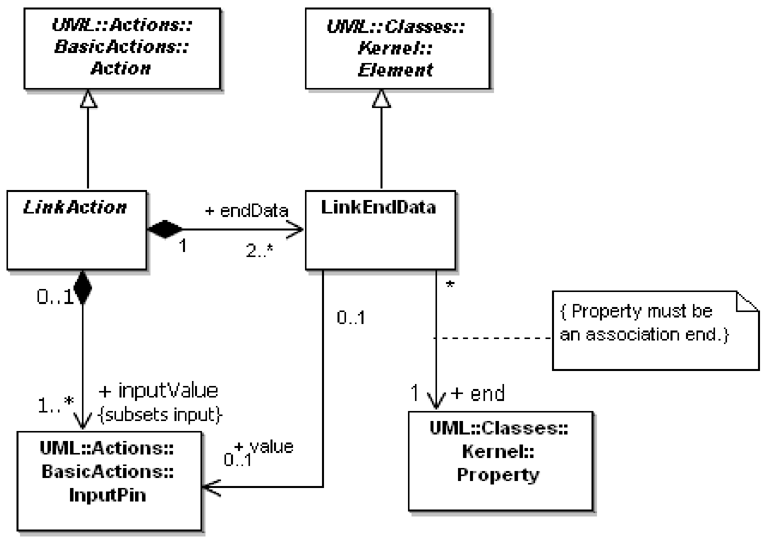


Figure 11.8 - Link identification

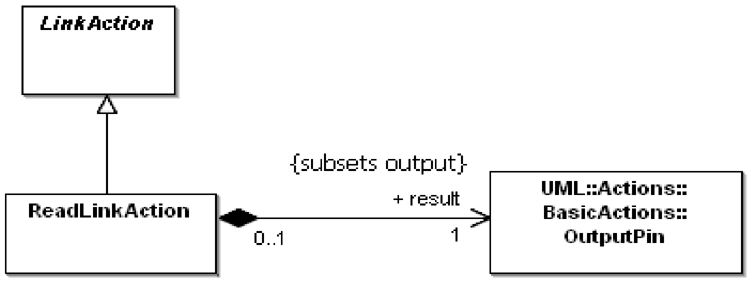


Figure 11.9 - Read link actions

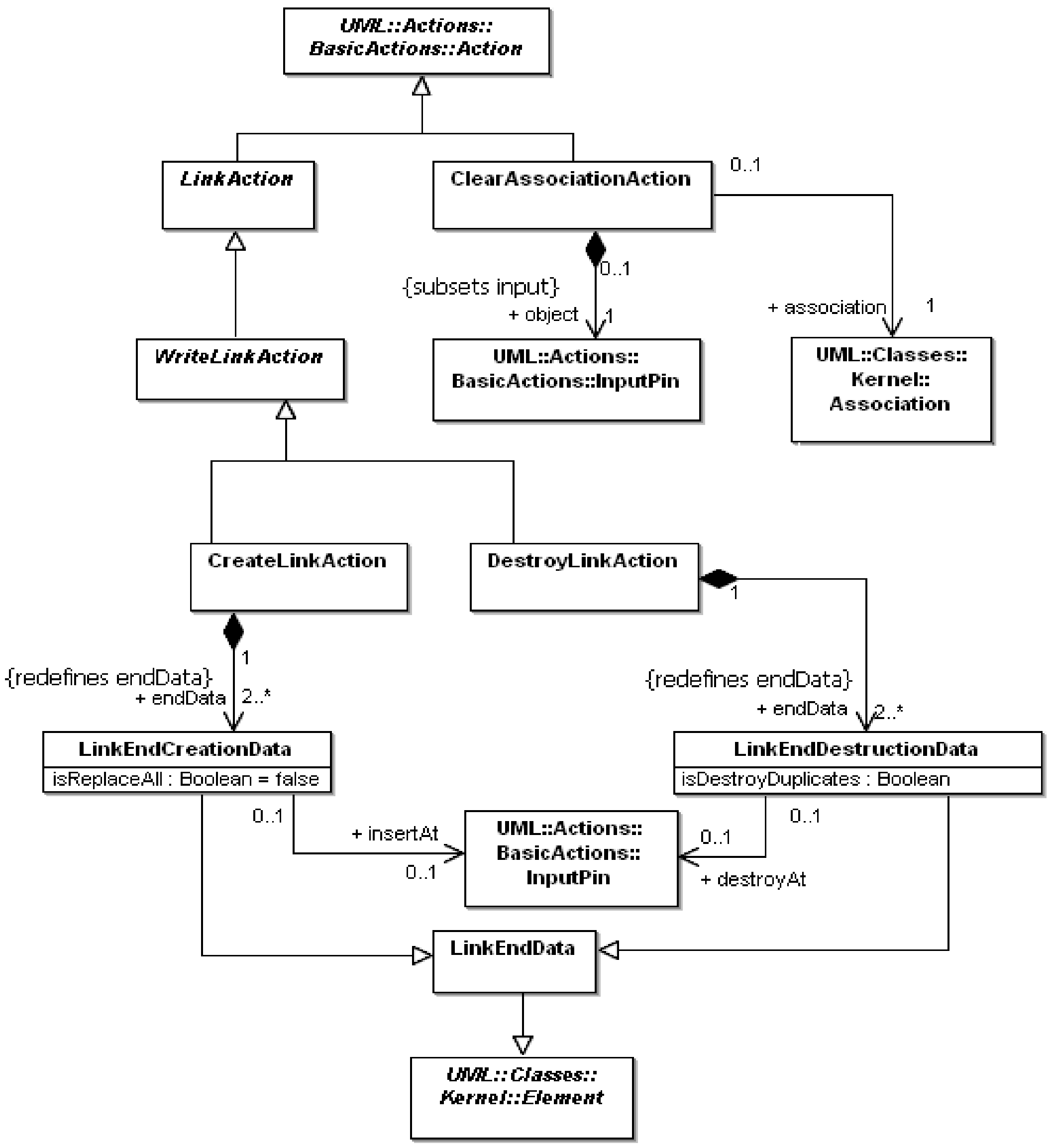


Figure 11.10 - Write link actions

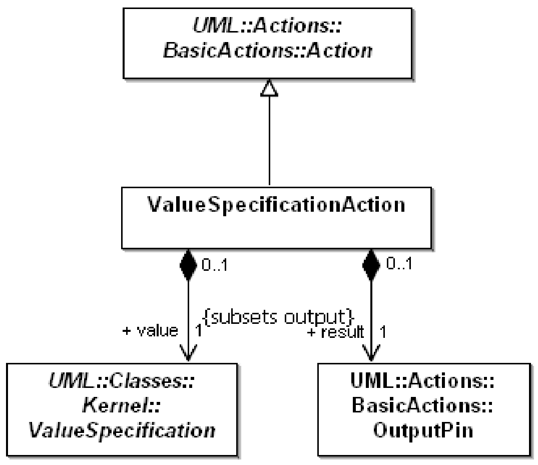


Figure 11.11 - Miscellaneous actions

Package CompleteActions

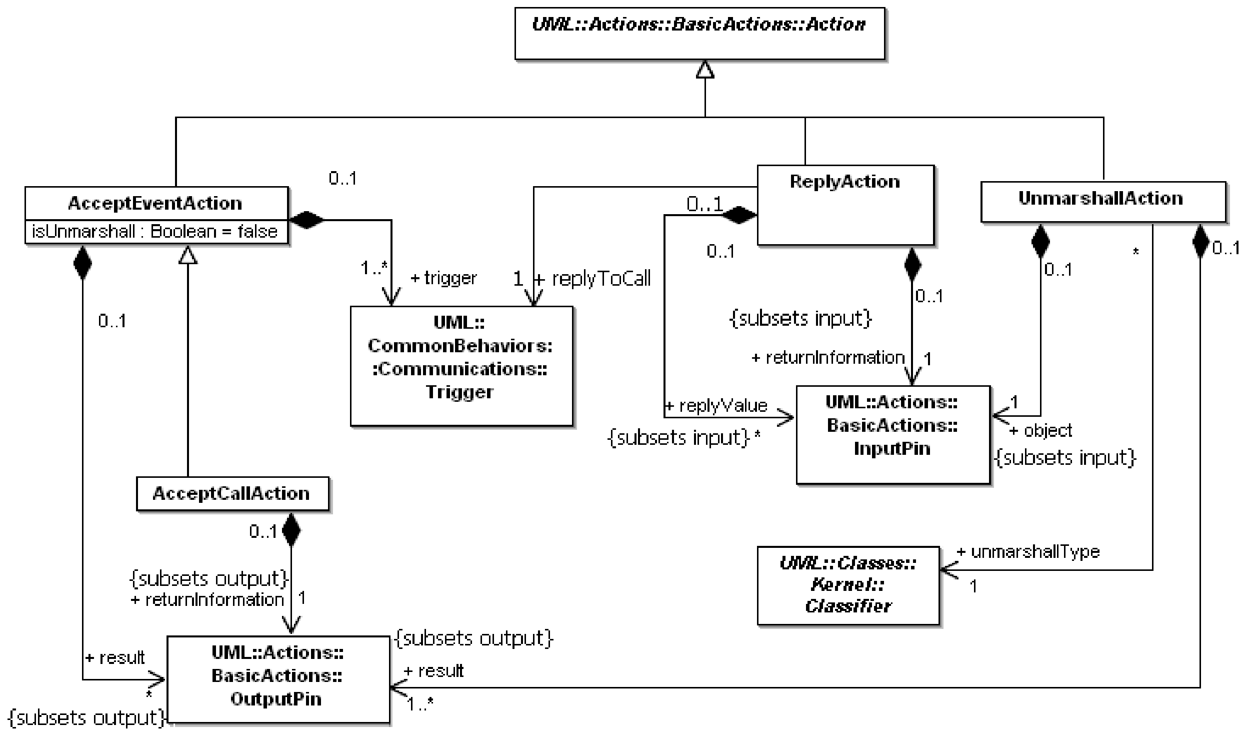


Figure 11.12 - Accept event actions

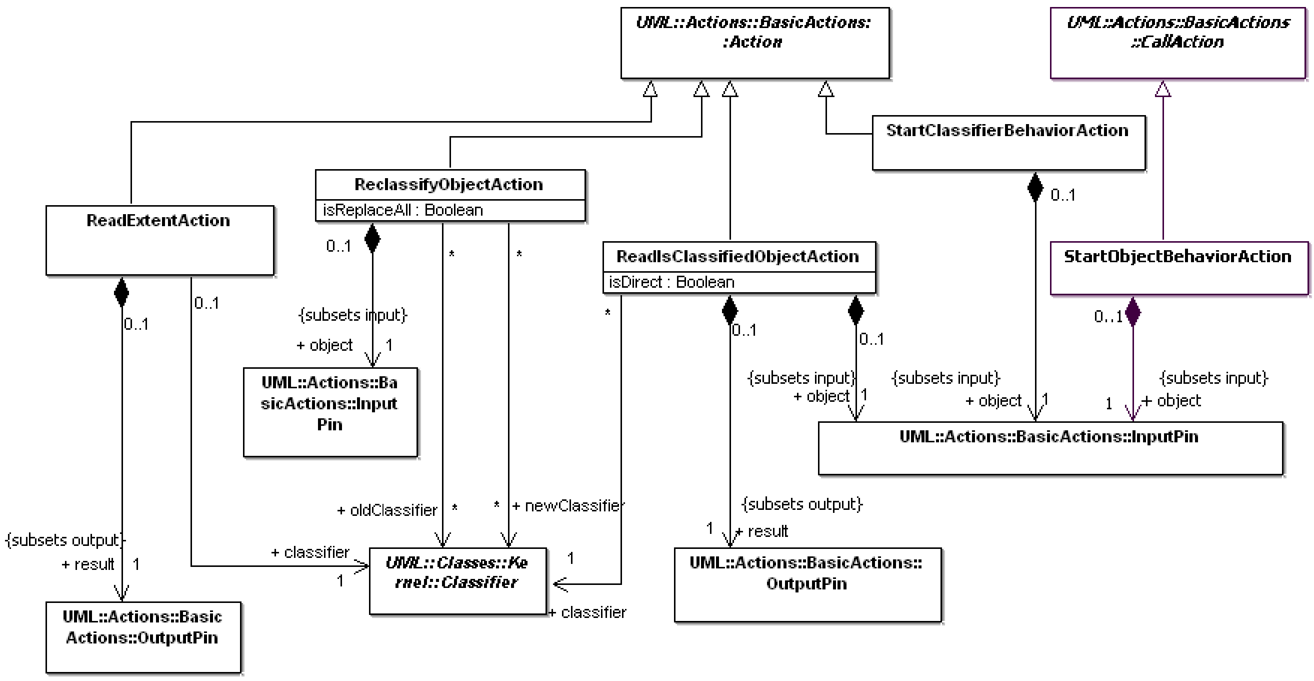

UML::Actions::BasicActions::Action
UML::Actions::BasicActions::CallAction
StartClassifierBehaviorAction
ReclassifyObjectAction
isReplaceAll : Boolean
ReadExtentAction
ReadIsClassifiedObjectAction
isDirect : Boolean
StartObjectBehaviorAction
UML::Actions::BasicActions::InputPin
UML::Actions::BasicActions::InputPin
UML::Actions::BasicActions::OutputPin
UML::Classes::Kernel::Classifier
UML::Actions::BasicActions::OutputPin
{subsets input}
+ object
{subsets output}
+ result
+ oldClassifier
+ newClassifier
+ classifier
0..1
1
*

Figure 11.13 - Object actions (CompleteActions)

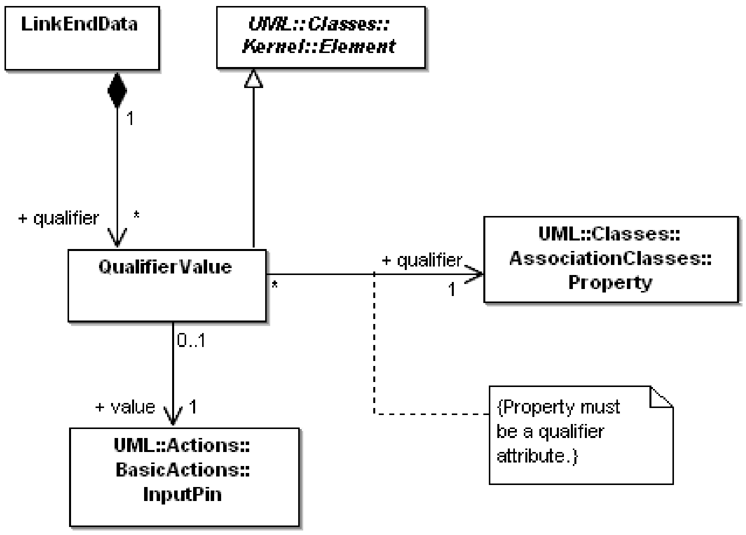

LinkEndData
UML::Classes::Kernel::Element
QualifierValue
UML::Classes::AssociationClasses::Property
UML::Actions::BasicActions::InputPin
+ qualifier
+ value
{Property must be a qualifier attribute.}
1
*
0..1

Figure 11.14 - Link identification (CompleteActions)

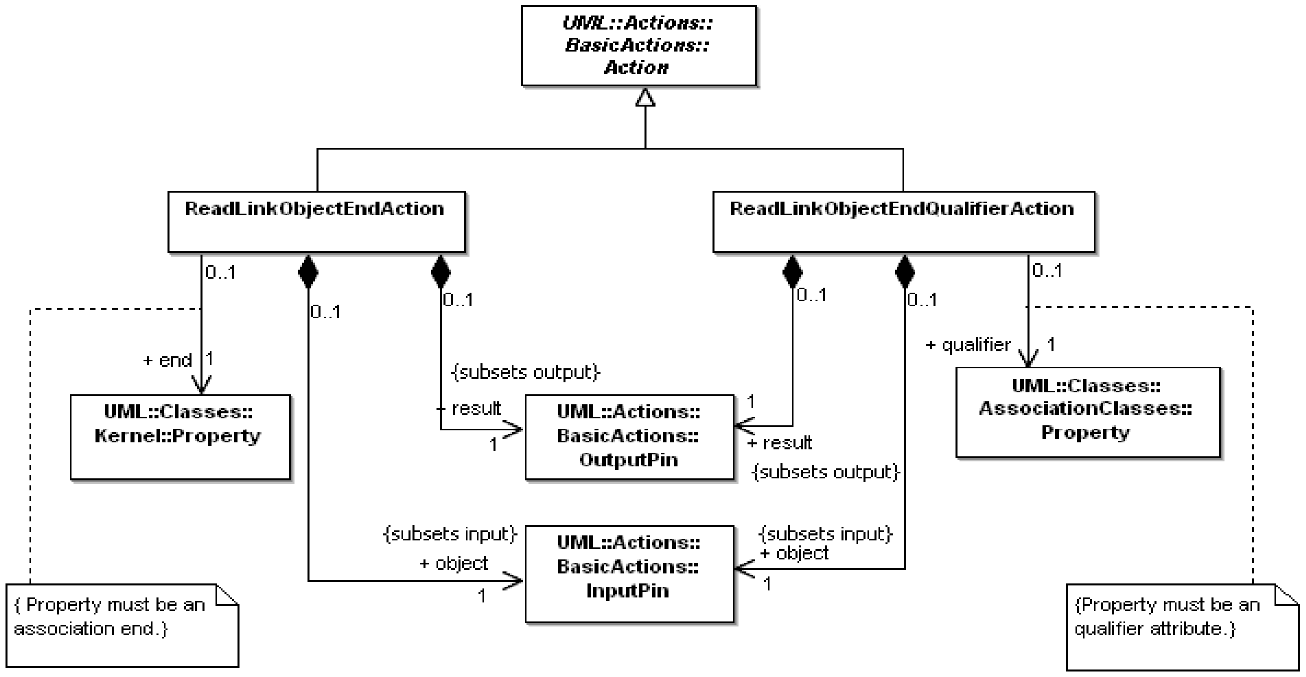


Figure 11.15 - Read link actions (CompleteActions)

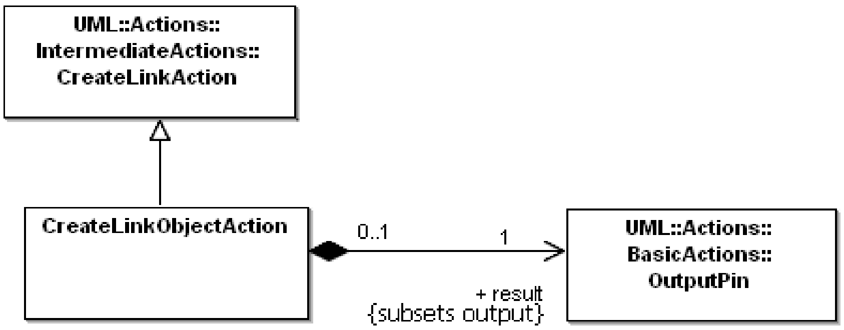


Figure 11.16 - Write link actions (CompleteActions)

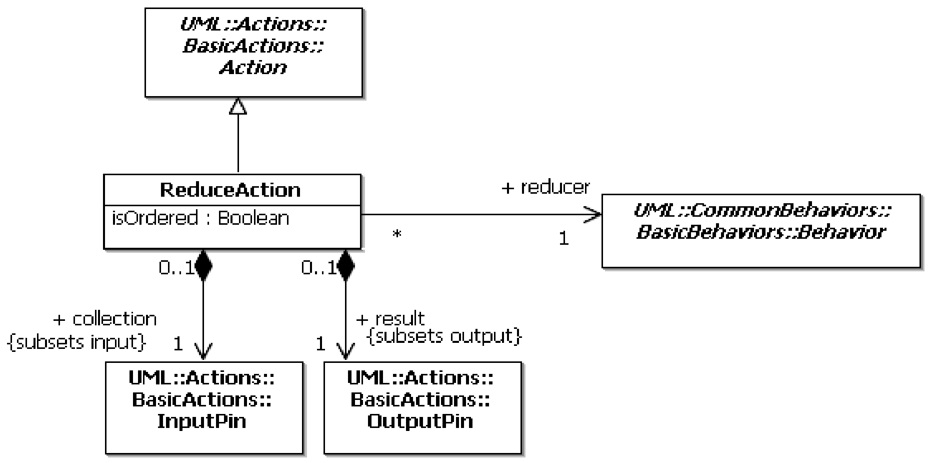


Figure 11.17 - ReduceAction (CompleteActions)

Package StructuredActions

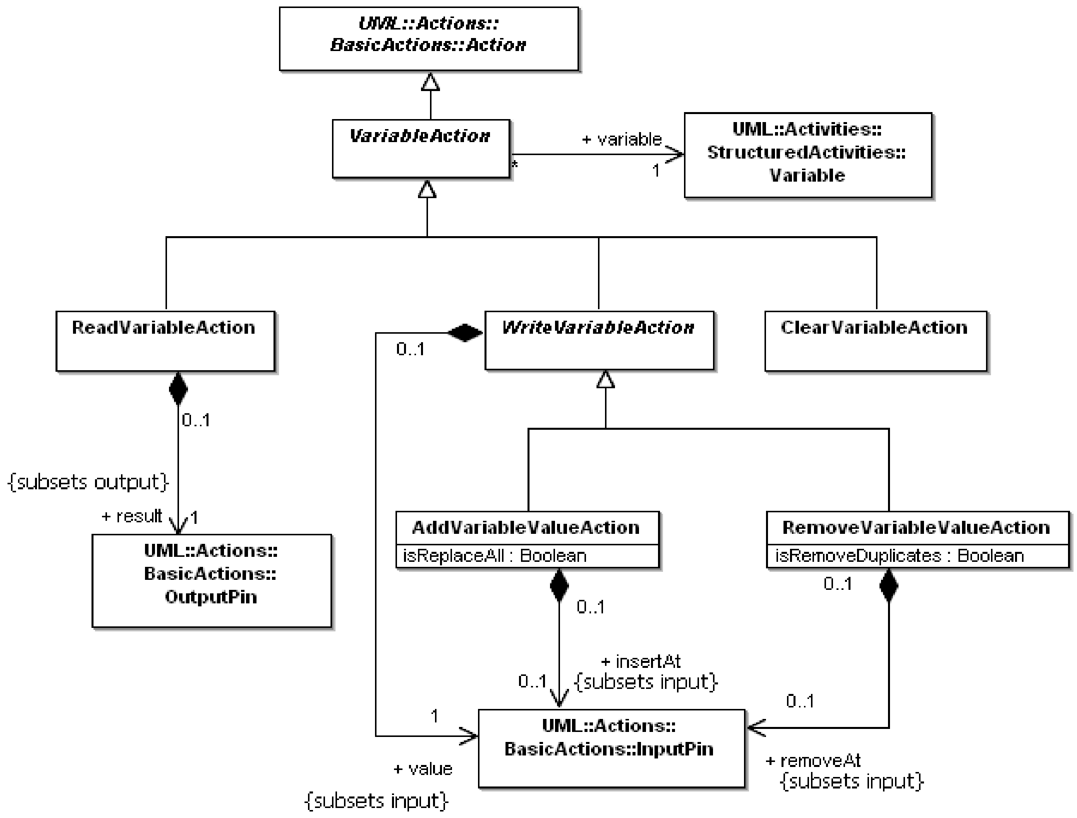
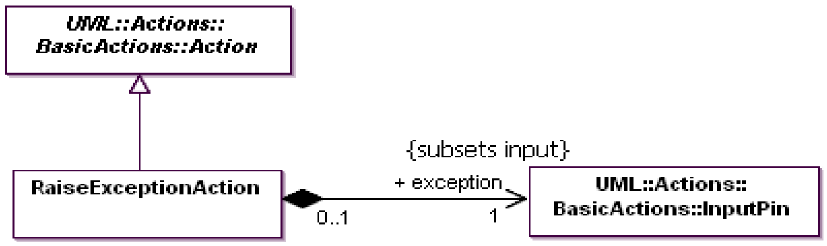


Figure 11.18 - Variable actions



UML::Actions::BasicActions::Action
RaiseExceptionAction
{subsets input}
+ exception
0..1
1
UML::Actions::BasicActions::InputPin

Figure 11.19 - Raise exception action

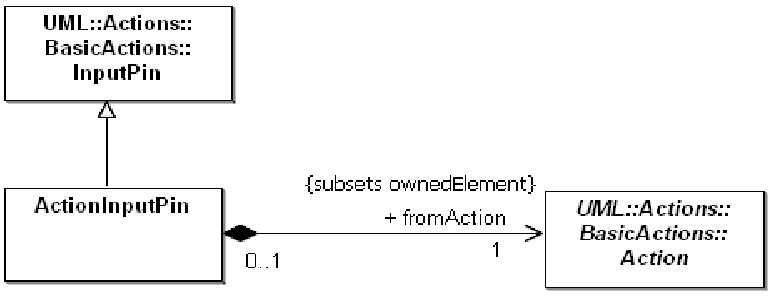


Figure 11.20 - Action input pin

11.3 Class Descriptions

11.3.1 AcceptCallAction (from CompleteActions)

Generalizations

- “AcceptEventAction (from CompleteActions)” on page 234

Description

AcceptCallAction is an accept event action representing the receipt of a synchronous call request. In addition to the normal operation parameters, the action produces an output that is needed later to supply the information to the ReplyAction necessary to return control to the caller.

This action is for synchronous calls. If it is used to handle an asynchronous call, execution of the subsequent reply action will complete immediately with no effects.

Attributes

No additional attributes

Associations

- returnInformation: OutputPin [1..1]
  Pin where a value is placed containing sufficient information to perform a subsequent reply 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. {Subsets *Action::output*}

Constraints

[1] The result pins must match the in and inout parameters of the operation specified by the trigger event in number, type, and order.

[2] The trigger event must be a CallEvent.

```
trigger.event.oclIsKindOf(CallEvent)
```

[3] isUnmarshall must be true for an AcceptCallAction.

isUnmarshall = true

Semantics

This action accepts (event) occurrences representing the receipt of calls on the operation specified by the trigger call event. If an ongoing activity behavior has executed an accept call action that has not completed and the owning object has an event occurrence representing a call of the specified operation, the accept call action claims the event occurrence and removes it from the owning object. If several accept call actions concurrently request a call on the same operation, it is unspecified which one claims the event occurrence, but one and only one action will claim the event. The argument values of the call are placed on the result output pins of the action. Information sufficient to perform a subsequent reply action is placed in the returnInformation output pin. The execution of the accept call action is then complete. This return information value is opaque and may only be used by ReplyAction.

Note that the target class must not define a method for the operation being received; otherwise, the operation call will be dispatched to that method instead of being put in the event buffer to be handled by AcceptCallAction. In general, classes determine how operation calls are handled, namely by a method, by a behavior owned directly by the class, by a state machine transition, and so on. The class must ensure that the operation call is handled in a way that AcceptCallAction has access to the call event.

11.3.2 AcceptEventAction (from CompleteActions)

Generalizations

- "Action (from BasicActions)" on page 236

Description

AcceptEventAction is an action that waits for the occurrence of an event meeting specified condition.

Attributes

- isUnmarshall : Boolean = false
  Indicates whether there is a single output pin for the event, or multiple output pins for attributes of the event.

Associations

- trigger : Trigger [1..*]
  The type of events accepted by the action, as specified by triggers. For triggers with signal events, a signal of the specified type or any subtype of the specified signal type is accepted.
- result: OutputPin [0..*]
  Pins holding the received event objects or their attributes. Event objects may be copied in transmission, so identity might not be preserved. {Subsets *Action::output*}

Constraints

[1] AcceptEventActions may have no input pins.

[2] There are no output pins if the trigger events are only ChangeEvents, or if they are only CallEvents when this action is an instance of AcceptEventAction and not an instance of a descendant of AcceptEventAction (such as AcceptCallAction).

[3] If the trigger events are all TimeEvents, there is exactly one output pin.

[4] If isUnmarshalled is true, there must be exactly one trigger for events of type 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.

Semantics

Accept event actions handle event occurrences detected by the object owning the behavior (also see “InterruptibleActivityRegion (from CompleteActivities)” on page 379). Event occurrences are detected by objects independently of actions and the occurrences are stored by the object. The arrangement of detected event occurrences is not defined, but it is expected that extensions or profiles will specify such arrangements. If the accept event action is executed and the object detected an event occurrence matching one of the triggers on the action and the occurrence has not been accepted by another action or otherwise consumed by another behavior, then the accept event action completes and outputs a value describing the occurrence. If such a matching occurrence is not available, the action waits until such an occurrence becomes available, at which point the action may accept it. In a system with concurrency, several actions or other behaviors might contend for an available event occurrence. Unless otherwise specified by an extension or profile, only one action accepts a given occurrence, even if the occurrence would satisfy multiple concurrently executing actions.

If the occurrence is a signal event occurrence and isUnmarshall is false, the result value contains a signal object whose reception by the owning object caused the occurrence. If the occurrence is a signal event occurrence and isUnmarshall is true, the attribute values of the signal are placed on the result output pins of the action. Signal objects may be copied in transmission and storage by the owning object, so identity might not be preserved. An action whose trigger is a signal event is informally called an accept signal action. If the occurrence is a time event occurrence, the result value contains the time at which the occurrence transpired. Such an action is informally called a wait time action. If the occurrences are all occurrences of ChangeEvent, or all CallEvent, or a combination of these, there are no output pins (however, see “AcceptCallAction (from CompleteActions)” on page 233). See CommonBehavior for a description of Event specifications. If the triggers are a combination of SignalEvents and ChangeEvents, the result is a null value if a change event occurrence or a call event occurrence is accepted.

This action handles asynchronous messages, including asynchronous calls. It cannot be used with synchronous calls (except see “AcceptCallAction (from CompleteActions)” on page 233).

Notation

An accept event action is notated with a concave pentagon. A wait time action is notated with an hour glass.

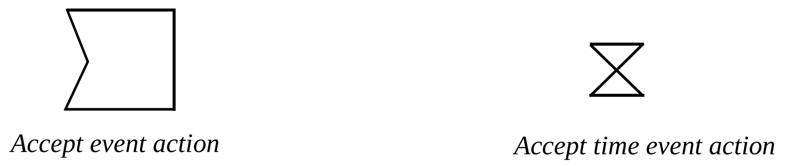


Figure 11.21 - Accept event notations

Examples

“AcceptEventAction (as specialized)” on page 309

Rationale

Accept event actions are introduced to handle processing of events during the execution of a behavior.

Changes from previous UML

AcceptEventAction is new in UML 2.0.

11.3.3 Action (from BasicActions)

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 98

Description

An action is a named element that is the fundamental unit of executable functionality. The execution of an action represents some transformation or processing in the modeled system, be it a computer system or otherwise.

Attributes

No additional attributes

Associations

- /input : InputPin [*]
  The ordered set of input pins connected to the Action. These are among the total set of inputs. {Specializes *Element::ownedElement*}
- /output : OutputPin [*]
  The ordered set of output pins connected to the Action. The action places its results onto pins in this set. {Specializes *Element::ownedElement*}
- /context : Classifier [0..1]
  The classifier that owns the behavior of which this action is a part.

Constraints

No additional constraints

Semantics

An action execution represents the run-time behavior of executing an action within a specific behavior execution. As Action is an abstract class, all action executions will be executions of specific kinds of actions. When the action executes, and what its actual inputs are, is determined by the concrete action and the behaviors in which it is used.

Notation

No specific notation. See extensions in Activities clause.

Changes from previous UML

Action is the same concept as in UML 1.5, but modeled independently of the behaviors that use it.

11.3.4 ActionInputPin (from StructuredActions)

Generalizations

- "InputPin (from BasicActions)" on page 255

Description

An action input pin is a kind of pin that executes an action to determine the values to input to another.

Attributes

No additional attributes

Associations

- fromAction : Action [1]
  The action used to provide values. {Subsets Element::ownedElement}

Constraints

[1] The fromAction of an action input pin must have exactly one output pin.

[2] The fromAction of an action input pin must only have action input pins as input pins.

[3] The fromAction of an action input pin cannot have control or data flows coming into or out of it or its pins.

Semantics

If an action is otherwise enabled, the fromActions on action input pins are enabled. The outputs of these are used as the values of the corresponding input pins. The process recurs on the input pins of the fromActions, if they also have action input pins. The recursion bottoms out at actions that have no inputs, such as for read variables or the self object. This forms a tree that is an action model for nested expressions.

Notation

No specific notation

Example

Example (in action language provided just for example, not normative):

```
self.foo->bar(self.baz);
```

meaning get the foo attribute of self, then send a bar signal to it with argument from the baz attribute of self. The repository model is shown below.

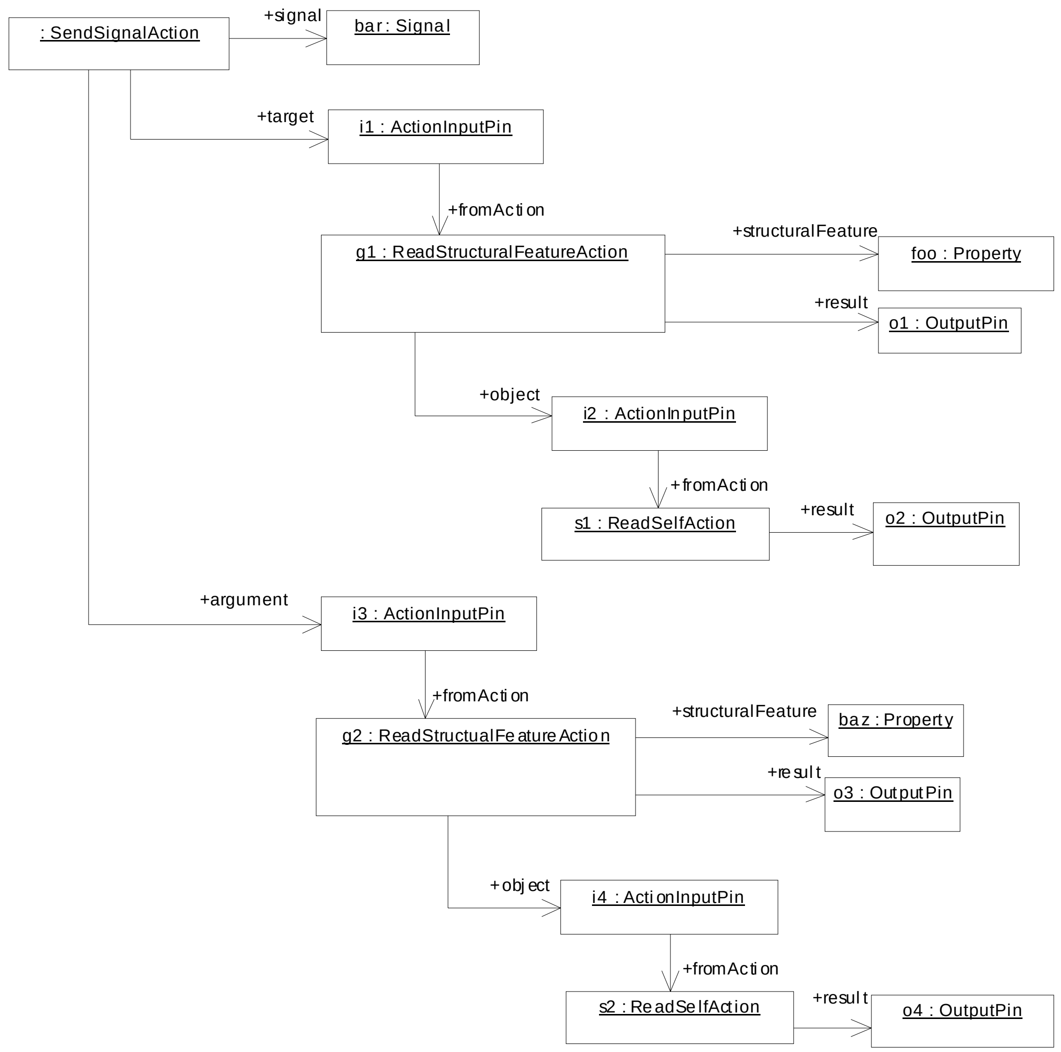


Figure 11.22 - Example repository model

Rationale

ActionInputPin is introduced to pass values between actions in expressions without using flows.

11.3.5 AddStructuralFeatureValueAction (from IntermediateActions)

AddStructuralFeatureValueAction is a write structural feature action for adding values to a structural feature.

Generalizations

- "WriteStructuralFeatureAction (from IntermediateActions)" on page 293.

Description

Structural Features are potentially multi-valued and ordered, so the action supports specification of insertion points for new values. It also supports the removal of existing values of the structural feature before the new value is added.

The object to access is specified dynamically, by referring to an input pin on which the object will be placed at runtime. The type of the value of this pin is the classifier that owns the specified structural feature, and the value's multiplicity is 1..1.

Attributes

- isReplaceAll : Boolean [1..1] = false
  Specifies whether existing values of the structural feature of the object should be removed before adding the new value.

Associations

- insertAt : InputPin [0..1]
  Gives the position at which to insert a new value or move an existing value in ordered structural features. The type of the pin is UnlimitedNatural, but the value cannot be zero. This pin is omitted for unordered structural features. (Subsets *Action::input*)

Constraints

[1] Actions adding a value to ordered structural features must have a single input pin for the insertion point with type UnlimitedNatural and multiplicity of 1..1; otherwise, the action has no input pin for the insertion point.

```
let insertAtPins : Collection = self.insertAt in
    if self.structuralFeature.isOrdered = #false
    then insertAtPins->size() = 0
    else let insertAtPin : InputPin= insertAt->asSequence()->first() in
            insertAtPins->size() = 1
            and insertAtPin.type = UnlimitedNatural
            and insertAtPin.multiplicity.is(1,1))
    endif
```

Semantics

If isReplaceAll is true, then the existing values of the structural feature are removed before the new one added, except if the new value already exists, then it is not removed under this option. If isReplaceAll is false and the structural feature is unordered and unique, then adding an existing value has no effect. If the feature is an association end, the semantics are the same as creating a link, the participants of which are the object owning the structural feature and the new value.

Values of a structural feature may be ordered or unordered, even if the multiplicity maximum is 1. Adding values to ordered structural features requires an insertion point for a new value using the insertAt input pin. The insertion point is a positive integer giving the position to insert the value, or unlimited, to insert at the end. 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 one meaning the new value will be first in the sequence. A value of unlimited for insertAt means to insert the new value at the end of the sequence. The semantics is undefined for a value of zero or an integer greater than the number of existing values. The insertion point is required for ordered structural features and omitted for unordered structural features. Reinserting an existing value at a new position in an ordered unique structural feature moves the value to that position (this works because structural feature values are sets). The insertion point is ignored when replacing all values.

The semantics is undefined for adding a value that violates the upper multiplicity of the structural feature. Removing a value succeeds even when that violates the minimum multiplicity—the same as if the minimum were zero. The modeler must determine when minimum multiplicity of structural features should be enforced.

The semantics is undefined for adding a new value for a structural feature with isReadonly=true after initialization of the owning object.

Notation

No specific notation

Rationale

AddStructuralFeatureValueAction is introduced to add structural feature values. isReplaceAll is introduced to replace and add in a single action, with no intermediate states of the object where only some of the existing values are present.

Changes from previous UML

AddStructuralFeatureValueAction is new in UML 2.0. It generalizes AddAttributeAction in UML 1.5.

11.3.6 AddVariableValueAction (from StructuredActions)

AddVariableValueAction is a write variable action for adding values to a variable.

Generalizations

- “WriteVariableAction (from StructuredActions)” on page 294

Description

Variables are potentially multi-valued and ordered, so the action supports specification of insertion points for new values. It also supports the removal of existing values of the variable before the new value is added.

Attributes

- isReplaceAll : Boolean [1..1] = false
  Specifies whether existing values of the variable should be removed before adding the new value.

Associations

- insertAt : InputPin [0..1]
  Gives the position at which to insert a new value or move an existing value in ordered variables. The type is UnlimitedINatural, but the value cannot be zero. This pin is omitted for unordered variables. (Subsets *Action::input*)

Constraints

[1] Actions adding values to ordered variables must have a single input pin for the insertion point with type UnlimitedNatural and multiplicity of 1..1; otherwise, the action has no input pin for the insertion point.

```
let insertAtPins : Collection = self.insertAt in
    if self.variable.ordering = #unordered
    then insertAtPins->size() = 0
    else let insertAtPin : InputPin = insertAt->asSequence()->first() in
            insertAtPins->size() = 1
```

```
                and insertAtPin.type = UnlimitedNatural
                and insertAtPin.multiplicity.is(1,1))
        endif
```

Semantics

If isReplaceAll is true, then the existing values of the variable are removed before the new one added, except if the new value already exists, then it is not removed under this option. If isReplaceAll is false and the variable is unordered and non-unique, then adding an existing value has no effect.

Values of a variable may be ordered or unordered, even if the multiplicity maximum is 1. Adding values to ordered variables requires an insertion point for a new value using the insertAt input pin. The insertion point is a positive integer giving the position to insert the value, or unlimited, to insert at the end. 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 one meaning the new value will be first in the sequence. A value of unlimited for insertAt means to insert the new value at the end of the sequence. The semantics is undefined for a value of zero or an integer greater than the number of existing values. The insertion point is required for ordered variables and omitted for unordered variables. Reinserting an existing value at a new position in an ordered unique variable moves the value to that position (this works because variable values are sets). The insertion point is ignored when replacing all values.

The semantics is undefined for adding a value that violates the upper multiplicity of the variable. Removing a value succeeds even when that violates the minimum multiplicity—the same as if the minimum were zero. The modeler must determine when minimum multiplicity of variables should be enforced.

Notation

No specific notation

Rationale

AddVariableValueAction is introduced to add variable values. isReplaceAll is introduced to replace and add in a single action, with no intermediate states of the variable where only some of the existing values are present.

Changes from previous UML

AddVariableValueAction is unchanged from UML 1.5.

11.3.7 BroadcastSignalAction (from IntermediateActions)

Generalizations

- "InvocationAction (from BasicActions)" on page 256

Description

BroadcastSignalAction is an action that transmits a signal instance to all the potential target objects in the system, which may cause the firing of a state machine transitions or the execution of associated activities of a target object. The argument values are available to the execution of associated behaviors. The requestor continues execution immediately after the signals are sent out. It does not wait for receipt. Any reply messages are ignored and are not transmitted to the requestor.

Attributes

No additional attributes

Associations

- signal: Signal [1]
  The specification of signal object transmitted to the target objects.

Constraints

[1] The number and order of argument pins must be the same as the number and order of attributes in the signal.

[2] The type, ordering, and multiplicity of an argument pin must be the same as the corresponding attribute of the signal.

Semantics

When all the prerequisites of the action execution are satisfied, a signal object is generated from the argument values according to signal and this signal object is transmitted concurrently to each of the implicit broadcast target objects in the system. The manner of identifying the set of objects that are broadcast targets is a semantic variation point and may be limited to some subset of all the objects that exist. There is no restriction on the location of target objects. The manner of transmitting the signal object, the amount of time required to transmit it, the order in which the transmissions reach the various target objects, and the path for reaching the target objects are undefined.

[1] When a transmission arrives at a target object, it may invoke a behavior in the target object. The effect of receiving such transmission is specified in Clause 13, “Common Behaviors.” Such effects include executing activities and firing state machine transitions.

[2] A broadcast signal action receives no reply from the invoked behavior; any attempted reply is simply ignored, and no transmission is performed to the requestor.

Semantic Variation Points

The determination of the set of broadcast target objects is a semantic variation point.

Notation

No specific notation

Rationale

Sends a signal to a set of system defined target objects.

Changes from previous UML

Same as UML 1.5.

11.3.8 CallAction (from BasicActions)

Generalizations

- “InvocationAction (from BasicActions)” on page 256.

Description

CallAction is an abstract class for actions that invoke behavior and receive return values.

Attributes

- isSynchronous: Boolean = 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 does not expect a return value.

Associations

- result: OutputPin [0..*]
  A list of output pins where the results of performing the invocation are placed. {Subsets *Action::input*}

Constraints

[1] Only synchronous call actions can have result pins.

[2] The number and order of argument pins must be the same as the number and order of parameters of the invoked behavior or behavioral feature. Pins are matched to parameters by order.

[3] The type, ordering, and multiplicity of an argument pin must be the same as the corresponding parameter of the behavior or behavioral feature.

Semantics

Parameters on behaviors and operations are totally ordered lists. To match parameters to pins on call actions, select the sublist of that list that corresponds to in and inout owned parameters (i.e., Behavior.ownedParameter). The input pins on Action::input are matched in order against these parameters in the sublist order. Then take the sublist of the parameter list that corresponds to out, inout, and return parameters. The output pins on Action::output are matched in order against these parameters in sublist order.

See children of CallAction.

11.3.9 CallBehaviorAction (from BasicActions)

Generalizations

- “CallAction (from BasicActions)” on page 242

Description

CallBehaviorAction is a call action that invokes a behavior directly rather than invoking a behavioral feature that, in turn, results in the invocation of that behavior. The argument values of the action are available to the execution of the invoked behavior. For synchronous calls the execution of the call behavior action waits until the execution of the invoked behavior completes and a result is returned on its output pin. The action completes immediately without a result, if the call is asynchronous.

Attributes

No additional attributes

Associations

- behavior : Behavior [1..1]
  The invoked behavior. It must be capable of accepting and returning control.

Constraints

[1] The number of argument pins and the number of parameters of the behavior of type *in* and *in-out* must be equal.

[2] The number of result pins and the number of parameters of the behavior of type *return*, *out*, and *in-out* must be equal.

[3] The type, ordering, and multiplicity of an argument or result pin is derived from the corresponding parameter of the behavior.

Semantics

[1] When all the prerequisites of the action execution are satisfied, CallBehaviorAction invokes its specified behavior with the values on the input pins as arguments. When the behavior is finished, the output values are put on the output pins. Each parameter of the behavior of the action provides output to a pin or takes input from one. No other implementation specifics are implied, such as call stacks, and so on. See "Pin (from BasicActions)" on page 263.

[2] If the call is asynchronous, the action completes immediately. Execution of the invoked behavior proceeds without any further dependency on the execution of the behavior containing the invoking action. Once the invocation of the behavior has been initiated, execution of the asynchronous action is complete.

[3] An asynchronous invocation completes when its behavior is started, or is at least ensured to be started at some point. Any return or out values from the invoked behavior are not passed back to the containing behavior. When an asynchronous invocation is done, the containing behavior continues regardless of the status of the invoked behavior. For example, the containing behavior may complete even though the invoked behavior is not finished.

[4] If the call is synchronous, execution of the calling action is blocked until it receives a reply from the invoked behavior. The reply includes values for any return, out, or inout parameters.

[5] If the call is synchronous, when the execution of the invoked behavior completes, the result values are placed on the result pins of the call behavior action, and the execution of the action is complete (StructuredActions, ExtraStructuredActivities). If the execution of the invoked behavior yields an exception, the exception is transmitted to the call behavior action to begin search for a handler. See RaiseExceptionAction.

Notation

See specialization of "CallBehaviorAction (as specialized)" on page 348.

Presentation Options

See specialization of "CallBehaviorAction (as specialized)" on page 348.

Rationale

Invokes a behavior directly without the need for a behavioral feature.

Changes from previous UML

Same as UML 1.5

11.3.10 CallOperationAction (from BasicActions)

Generalizations

- "CallAction (from BasicActions)" on page 242

Description

CallOperationAction is an action that transmits an operation call request to the target object, where it may cause the invocation of associated behavior. The argument values of the action are available to the execution of the invoked behavior. If the action is marked synchronous, the execution of the call operation action waits until the execution of the invoked behavior completes and a reply transmission is returned to the caller; otherwise, execution of the action is complete when the invocation of the operation is established and the execution of the invoked operation proceeds concurrently with the execution of the calling behavior. Any values returned as part of the reply transmission are put on the result output pins of the call operation action. Upon receipt of the reply transmission, execution of the call operation action is complete.

Attributes

No additional attributes

Associations

- operation: Operation [1]
  The operation to be invoked by the action execution.
- target: InputPin [1]
  The target object to which the request is sent. The classifier of the target object is used to dynamically determine a behavior to invoke. This object constitutes the context of the execution of the operation. {Subsets *Action::input*}

Constraints

[1] The number of argument pins and the number of owned parameters of the operation of type *in* and *in-out* must be equal.

[2] The number of result pins and the number of owned parameters of the operation of type *return*, *out*, and *in-out* must be equal.

[3] The type, ordering, and multiplicity of an argument or result pin is derived from the corresponding owned parameter of the operation.

[4] The type of the target pin must be the same as the type that owns the operation.

Semantics

The inputs to the action determine the target object and additional actual arguments of the call.

[1] When all the prerequisites of the action execution are satisfied, information comprising the operation and the argument pin values of the action execution is created and transmitted to the target object. The target objects may be local or remote. The manner of transmitting the call, the amount of time required to transmit it, the order in which the transmissions reach the various target objects, and the path for reaching the target objects are undefined.

[2] When a call arrives at a target object, it may invoke a behavior in the target object. The effect of receiving such call is specified in 13, "Common Behaviors." Such effects include executing activities and firing state machine transitions.

[3] If the call is synchronous, when the execution of the invoked behavior completes, its return results are transmitted back as a reply to the calling action execution. The manner of transmitting the reply, the time required for transmission, the

representation of the reply transmission, and the transmission path are unspecified. If the execution of the invoked behavior yields an exception, the exception is transmitted to the caller where it is reraised as an exception in the execution of the calling action. Possible exception types may be specified by attaching them to the called Operation using the *raisedException* association.

[4] If the call is asynchronous, the caller proceeds immediately and the execution of the call operation action is complete. Any return or out values from the invoked operation are not passed back to the containing behavior. If the call is synchronous, the caller is blocked from further execution until it receives a reply from the invoked behavior.

[5] When the reply transmission arrives at the invoking action execution, the return result values are placed on the result pins of the call operation action, and the execution of the action is complete.

Semantic Variation Points

The mechanism for determining the method to be invoked as a result of a call operation is unspecified.

Notation

See “CallOperationAction (as specialized)” on page 350

Presentation Options

See “CallOperationAction (as specialized)” on page 350

Rationale

Calls an operation on a specified target object.

Changes from previous UML

Same as UML 1.5.

11.3.11 ClearAssociationAction (from IntermediateActions)

ClearAssociationAction is an action that destroys all links of an association in which a particular object participates.

Generalizations

- “Action (from BasicActions)” on page 236

Description

This action destroys all links of an association that have a particular object at one end.

Attributes

No additional attributes

Associations

- association : Association [1..1]
  Association to be cleared.

- object : InputPin [1..1]
  Gives the input pin from which is obtained the object whose participation in the association is to be cleared. (Subsets *Action::input*)

Constraints

[1] The type of the input pin must be the same as the type of at least one of the association ends of the association.

```
self.association->exists(end.type = self.object.type)
```

[2] The multiplicity of the input pin is 1..1.

```
self.object.multiplicity.is(1,1)
```

Semantics

This action has a statically-specified association. It has an input pin for a runtime object that must be of the same type as at least one of the association ends of the association. All links of the association in which the object participates are destroyed even when that violates the minimum multiplicity of any of the association ends. If the association is a class, then link object identities are destroyed.

Notation

No specific notation

Rationale

ClearAssociationAction is introduced to remove all links from an association in which an object participates in a single action, with no intermediate states where only some of the existing links are present.

Changes from previous UML

ClearAssociationAction is unchanged from UML 1.5.

11.3.12 ClearStructuralFeatureAction (from IntermediateActions)

ClearStructuralFeatureAction is a structural feature action that removes all values of a structural feature.

Generalizations

- "StructuralFeatureAction (from IntermediateActions)" on page 286

Description

This action removes all values of a structural feature.

Attributes

No additional attributes

Associations

- result : OutputPin [0..1]Gives the output pin on which the result is put. {Subsets *Action:output*}

Constraints

[1] The type of the result output pin is the same as the type of the inherited object input pin.

```
result->notEmpty() implies self.result.type = self.object.type
```

[2] The multiplicity of the result output pin must be 1..1.

```
result->notEmpty() implies self.result.multiplicity.is(1,1)
```

Semantics

All values are removed even when that violates the minimum multiplicity of the structural feature—the same as if the minimum were zero. The semantics is undefined for a structural feature with isReadOnly = true after initialization of the object owning the structural feature, unless the structural feature has no values. The action has no effect if the structural feature has no values. If the feature is an association end, the semantics are the same as for ClearAssociationAction on the object owning the structural feature.

If a result output pin is provided, then the input object, as modified, is placed on the output pin. If the input object is actually a data value, then a copy of the input data value is placed on the output pin, but with the appropriate structural feature cleared.

Notation

No specific notation

Rationale

ClearStructuralFeatureAction is introduced to remove all values from a structural feature in a single action, with no intermediate states where only some of the existing values are present.

Changes from previous UML

ClearStructuralFeatureAction is new in UML 2.0. It generalizes ClearAttributeAction from UML 1.5.

11.3.13 ClearVariableAction (from StructuredActions)

ClearVariableAction is a variable action that removes all values of a variable.

Generalizations

- “VariableAction (from StructuredActions)” on page 291

Description

This action removes all values of a variable.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

All values are removed even when that violates the minimum multiplicity of the variable—the same as if the minimum were zero.

Notation

No specific notation

Rationale

ClearVariableAction is introduced to remove all values from a variable in a single action, with no intermediate states where only some of the existing values are present.

Changes from previous UML

ClearVariableAction is unchanged from UML 1.5.

11.3.14 CreateLinkAction (from IntermediateActions)

(IntermediateActions) CreateLinkAction is a write link action for creating links.

Generalizations

- “WriteLinkAction (from IntermediateActions)” on page 292

Description

This action can be used to create links and link objects. There is no return value in either case. This is so that no change of the action is required if the association is changed to an association class or vice versa. CreateLinkAction uses a specialization of LinkEndData called LinkEndCreationData, to support ordered associations. The insertion point is specified at runtime by an additional input pin, which is required for ordered association ends and omitted for unordered ends. The insertion point is an integer greater than 0 giving the position to insert the link, or unlimited, to insert at the end. Reinserting an existing end at a new position in an ordered unique structural feature moves the end to that position.

CreateLinkAction also uses LinkEndCreationData to support the destruction of existing links of the association that connect any of the objects of the new link. When the link is created, this option is available on an end-by-end basis, and causes all links of the association emanating from the specified ends to be destroyed before the new link is created.

Attributes

No additional attributes

Associations

- endData : LinkEndCreationData [2..*]
  Specifies ends of association and inputs. (Redefines *LinkAction::endData*)

Constraints

[1] The association cannot be an abstract classifier.

```
self.association().isAbstract = #false
```

Semantics

CreateLinkAction creates a link or link object for an association or association class. It has no output pin, because links are not necessarily values that can be passed to and from actions. When the action creates a link object, the object could be returned on output pin, but it is not for consistency with links. This allows actions to remain unchanged when an association is changed to an association class or vice versa. The semantics of CreateLinkObjectAction applies to creating link objects with CreateLinkAction.

This action also supports the destruction of existing links of the association that connect any of the objects of the new link. This option is available on an end-by-end basis, and causes all links of the association emanating from the specified ends to be destroyed before the new link is created. If the link already exists, then it is not destroyed under this option; otherwise, recreating an existing link has no effect if the structural feature is unordered and non-unique.

The semantics is undefined for creating a link for an association class 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. Objects participating in the association across from a writeable end can have links created as long as the objects across from all read only ends are still being initialized. This means that objects participating in links with two or more read only ends cannot have links created unless all the linked objects are being initialized.

Creating ordered association ends requires an insertion point for a new link using the insertAt input pin of LinkEndCreationData. The pin is of type UnlimitedNatural with multiplicity of 1..1. A pin value 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 one meaning the new link will be first in the sequence. A value of unlimited for insertAt means to insert the new link at the end of the sequence. The semantics is undefined for value of zero or an integer greater than the number of existing links. The insertAt input pin does not exist for unordered association ends. Reinserting an existing end at a new position in an ordered unique structural feature moves the end so that it is in the position specified after the action is complete.

Notation

No specific notation

Rationale

CreateLinkAction is introduced to create links.

Changes from previous UML

CreateLinkAction is unchanged from UML 1.5.

11.3.15 CreateLinkObjectAction (from CompleteActions)

CreateLinkObjectAction creates a link object.

Generalizations

- "CreateLinkAction (from IntermediateActions)" on page 249

Description

This action is exclusively for creating links of association classes. It returns the created link object.

Attributes

No additional attributes

Associations

- result [1..1] : OutputPin [1..1]
  Gives the output pin on which the result is put. (Subsets *Action::output*)

Constraints

[1] The association must be an association class.
self.association().oclIsKindOf(Class)

[2] The type of the result pin must be the same as the association of the action.
self.result.type = self.association()

[3] The multiplicity of the output pin is 1..1.
self.result.multiplicity.is(1,1)

Semantics

CreateLinkObjectAction inherits the semantics and constraints of CreateLinkAction, except that it operates on association classes to create a link object. The additional semantics over CreateLinkAction is that the new or found link object is put on the output pin. If the link already exists, then the found link object is put on the output pin. The semantics of CreateObjectAction applies to creating link objects with CreateLinkObjectAction.

Notation

No specific notation

Rationale

CreateLinkObjectAction is introduced to create link objects in a way that returns the link object. Compare CreateLinkAction.

Changes from previous UML

CreateLinkObjectAction is unchanged from UML 1.5.

11.3.16 CreateObjectAction (from IntermediateActions)

CreateObjectAction is an action that creates an object that conforms to a statically specified classifier and puts it on an output pin at runtime.

Generalizations

- "Action (from BasicActions)" on page 236

Description

This action instantiates a classifier.

Attributes

No additional attributes

Associations

- classifier : Classifier [1..1]
  Classifier to be instantiated.
- result : OutputPin [1..1]
  Gives the output pin on which the result is put. (Subsets *Action::output*)

Constraints

[1] The classifier cannot be abstract.
not (self.classifier.isAbstract = #true)

[2] The classifier cannot be an association class.
not self.classifier.oclIsKindOf(AssociationClass)

[3] The type of the result pin must be the same as the classifier of the action.
self.result.type = self.classifier

[4] The multiplicity of the output pin is 1..1.
self.result.multiplicity.is(1,1)

Semantics

The new object is created, and the classifier of the object is set to the given classifier. The new object is returned as the value of the action. The action has no other effect. In particular, no behaviors are executed, no initial expressions are evaluated, and no state machine transitions are triggered. The new object has no structural feature values 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 actually start immediately on instantiation. Rather, it must be explicitly started using a StartObjectBehaviorAction..

Notation

No specific notation

Rationale

CreateObjectAction is introduced for creating new objects.

Changes from previous UML

Same as UML 1.5

11.3.17 DestroyLinkAction (from IntermediateActions)

DestroyLinkAction is a write link action that destroys links and link objects.

Generalizations

- "WriteLinkAction (from IntermediateActions)" on page 292.

Description

This action destroys a link or a link object. Link objects can also be destroyed with DestroyObjectAction. The link is specified in the same way as link creation, even for link objects. This allows actions to remain unchanged when their associations are transformed from ordinary ones to association classes and vice versa.

DestroyLinkAction uses a specialization of LinkEndData, called LinkEndDestructionData, to support ordered non-unique associations. The position of the link to be destroyed is specified at runtime by an additional input pin, which is required for ordered non-unique association ends and omitted for other kinds of ends. This is a positive integer giving the position of the link to destroy.

DestroyLinkAction also uses LinkEndDestructionData to support the destruction of duplicate links of the association on ends that are non-unique. This option is available on an end-by-end basis, and causes all duplicate links of the association emanating from the specified ends to be destroyed.

Attributes

No additional attributes

Associations

- endData : LinkEndDestructionData [2..*]
  Specifies ends of association and inputs. {Redefines *LinkAction::endData*}

Constraints

No additional constraints

Semantics

Destroying a link that does not exist has no effect. The semantics of DestroyObjectAction applies to destroying a link that has a link object with DestroyLinkAction.

The semantics is undefined for destroying a link that has an association end with isReadOnly = true after initialization of the other end objects, unless the link being destroyed does not exist. Objects participating in the association across from a writeable end can have links destroyed as long as the objects across from all read only ends are still being initialized. This means objects participating in two or more readOnly ends cannot have links destroyed, unless all the linked objects are being initialized.

Destroying links for non-unique ordered association ends requires identifying the position of the link using the input pin of LinkEndDestructionData. The pin is of type UnlimitedNatural with multiplicity 1..1. A pin value 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 one meaning the first link in the sequence. The semantics is undefined for value of zero, for an integer greater than the number of existing links, and for unlimited. The destroyAt input pin only exists for ordered non-unique association ends.

Notation

No specific notation

Rationale

DestroyLinkAction is introduced for destroying links.

Changes from previous UML

DestroyLinkAction is unchanged from UML 1.5.

11.3.18 DestroyObjectAction (from IntermediateActions)

DestroyObjectAction is an action that destroys objects.

Generalizations

- “Action (from BasicActions)” on page 236

Description

This action destroys the object on its input pin at runtime. The object may be a link object, in which case the semantics of DestroyLinkAction also applies.

Attributes

- isDestroyLinks : Boolean = false
  Specifies whether links in which the object participates are destroyed along with the object. Default value is *false*.
- isDestroyOwnedObjects : Boolean = false
  Specifies whether objects owned by the object through composite aggregation are destroyed along with the object. Default value is *false*.

Associations

- target : InputPin [1..1]
  The input pin providing the object to be destroyed. (Subsets *Action::input*)

Constraints

[1] The multiplicity of the input pin is 1..1.
self.target.multiplicity.is(1,1)

[2] The input pin has no type.
self.target.type->size() = 0

Semantics

The classifiers of the object are removed as its classifiers, and the object is destroyed. The default 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. 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.

Notation

No specific notation

Rationale

DestroyObjectAction is introduced for destroying objects.

Changes from previous UML

Same as UML 1.5

11.3.19 InputPin (from BasicActions)

Generalizations

- "Pin (from BasicActions)" on page 263

Description

An input pin is a pin that holds input values to be consumed by an action.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

An action cannot start execution if an input pin has fewer values than the lower multiplicity. The upper multiplicity determines the maximum number of values that can be consumed by a single execution of the action.

Notation

No specific notation. See extensions in Activities.

Rationale

Changes from previous UML

InputPin is the same concept as in UML 1.5, but modeled independently of the behaviors that use it.

11.3.20 InvocationAction (from BasicActions)

Generalizations

- “Action (from BasicActions)” on page 236

Description

Invocation is an abstract class for the various actions that invoke behavior.

Attributes

No additional attributes

Associations

- argument : InputPin [0..*]
  Specification of the ordered set of argument values that appear during execution.

Constraints

No additional constraints

Semantics

See children of InvocationAction.

11.3.21 LinkAction (from IntermediateActions)

LinkAction is an abstract class for all link actions that identify their links by the objects at the ends of the links and by the qualifiers at ends of the links.

Generalizations

- “Action (from BasicActions)” on page 236

Description

A link action creates, destroys, or reads links, identifying a link by its end objects and qualifier values, if any.

Attributes

No additional attributes

Associations

- endData : LinkEndData [2..*]
  Data identifying one end of a link by the objects on its ends and qualifiers.

- inputValue : InputPin [1..*]
  Pins taking end objects and qualifier values as input. (Subsets *Action::input*)

Constraints

[1] The association ends of the link end data must all be from the same association and include all and only the association ends of that association.

```
self.endData->collect(end) = self.association()->collect(connection)
```

[2] The association ends of the link end data must not be static.

```
self.endData->forall(end.oclisKindOf(NavigableEnd) implies end.isStatic = #false)
```

[3] The input pins of the action are the same as the pins of the link end data and insertion pins.

```
self.input->asSet() =
    let ledpins : Set = self.endData->collect(value) in
        if self.oclIsKindOf(LinkEndCreationData)
        then ledpins->union(self.endData.oclAsType(LinkEndCreationData).insertAt)
        else ledpins
```

Package CompleteActions

[4] The input pins of the action are the same as the pins of the link end data, qualifier values, and insertion pins.

```
self.input->asSet() =
    let ledpins : Set =
    if self.endData.oclIsKindOf(CompleteActions::LinkEndData)
    then self.endData->collect(value)->union(self.endData.qualifier.value)
    else self.endData->collect(value) in
        if self.oclIsKindOf(LinkEndCreationData)
        then ledpins->union(self.endData.oclAsType(LinkEndCreationData).insertAt)
        else ledpins
```

Additional operations:

[1] association operates on LinkAction. It returns the association of the action.

```
association();
association = self.endData->asSequence().first().end.association
```

Semantics

For actions that write links, all association ends must have a corresponding input pin so that all end objects are specified when creating or deleting a link. An input pin identifies the end object by being given a value at runtime. It has the type of the association end and multiplicity of 1..1 (see “LinkEndData (from IntermediateActions, CompleteActions)” on page 259), since a link always has exactly one object at its ends. The input pins owned by the action are referenced by the link end data, and as insertion pins (see “LinkEndCreationData (from IntermediateActions)” on page 258), and qualifier value pins in CompleteActions.

The behavior is undefined for links of associations that are static on any end.

For the semantics of link actions see the children of LinkAction.

Notation

No specific notation

Rationale

LinkAction is introduced to abstract aspects of link actions that identify links by the objects on their ends.

In CompleteActions, LinkAction is extended for qualifiers.

Changes from previous UML

LinkAction is unchanged from UML 1.5.

11.3.22 LinkEndCreationData (from IntermediateActions)

LinkEndCreationData is not an action. It is an element that identifies links. It identifies one end of a link to be created by CreateLinkAction.

Generalizations

- "LinkEndData (from IntermediateActions, CompleteActions)" on page 259.

Description

This class is required when using CreateLinkAction to specify insertion points for ordered ends and for replacing all links at end. A link cannot be passed as a runtime value to or from an action. Instead, a link is identified by its end objects and qualifier values, as required. This requires more than one piece of data, namely, the statically-specified end in the user model, the object on the end, and the qualifier values for that end. These pieces are brought together around LinkEndData. Each association end is identified separately with an instance of the LinkEndData class.

Qualifier values are used in CompleteActions to specify links to create.

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.

Associations

- insertAt : InputPin [0..1]
  Specifies where the new link should be inserted for ordered association ends, or where an existing link should be moved to. The type of the input is UnlimitedNatural, but the input cannot be zero. This pin is omitted for association ends that are not ordered.

Constraints

[1] LinkEndCreationData can only be end data for CreateLinkAction or one of its specializations.

```
self.LinkAction.oclIsKindOf(CreateLinkAction)
```

[2] Link end creation data for ordered association ends must have a single input pin for the insertion point with type UnlimitedNatural and multiplicity of 1..1; otherwise, the action has no input pin for the insertion point.

```
let insertAtPins : Collection = self.insertAt in
    if self.end.ordering = #unordered
    then insertAtPins->size() = 0
    else let insertAtPin : InputPin = insertAts->asSequence()->first() in
            insertAtPins->size() = 1
            and insertAtPin.type = UnlimitedNatural
            and insertAtPin.multiplicity.is(1,1))
    endif
```

Semantics

See CreateLinkAction, also see LinkAction and all its children.

Notation

No specific notation

Rationale

LinkEndCreationData is introduced to indicate which inputs are for which link end objects and qualifiers.

Changes from previous UML

LinkEndCreationData is unchanged from UML 1.5.

11.3.23 LinkEndData (from IntermediateActions, CompleteActions)

Generalizations

- "Element (from Kernel)" on page 64

Description

Package IntermediateActions

LinkEndData is not an action. It is an element that identifies links. It identifies one end of a link to be read or written by the children of LinkAction. A link cannot be passed as a runtime value to or from an action. Instead, a link is identified by its end objects and qualifier values, if any. This requires more than one piece of data, namely, the statically-specified end in the user model, the object on the end, and the qualifier values for that end, if any. These pieces are brought together around LinkEndData. Each association end is identified separately with an instance of the LinkEndData class.

Attributes

No additional attributes

Associations

- end : Property [1..1]
  Association end for which this link-end data specifies values.

- value : InputPin [0..1]
  Input pin that provides the specified object for the given end. This pin is omitted if the link-end data specifies an "open" end for reading.

Associations

Package CompleteActions

- qualifier : QualifierValue [*]
  List of qualifier values.

Constraints

[1] The property must be an association end.
self.end.association->size() = 1

[2] The type of the end object input pin is the same as the type of the association end.
self.value.type = self.end.type

[3] The multiplicity of the end object input pin must be "1..1."
self.value.multiplicity.is(1,1)

Constraints

Package CompleteActions

[1] The qualifiers include all and only the qualifiers of the association end.
self.qualifier->collect(qualifier) = self.end.qualifier

[2] The end object input pin is not also a qualifier value input pin.
self.value->excludesAll(self.qualifier.value)

Semantics

See LinkAction and its children.

Notation

No specific notation

Rationale

LinkEndData is introduced to indicate which inputs are for which link end objects and qualifiers.

Changes from previous UML

LinkEndData is unchanged from UML 1.5.

11.3.24 LinkEndDestructionData (from IntermediateActions)

LinkEndDestructionData is not an action. It is an element that identifies links. It identifies one end of a link to be destroyed by DestroyLinkAction.

Generalizations

- "LinkEndData (from IntermediateActions, CompleteActions)" on page 259.

Description

This class is required when using DestroyLinkAction, to specify links to destroy for non-unique ordered ends. A link cannot be passed as a runtime value to or from an action. See description of "LinkEndData (from IntermediateActions, CompleteActions)" on page 259.

Qualifier values are used in CompleteActions to identify which links to destroy.

Attributes

- isDestroyDuplicates : Boolean = false
  Specifies whether to destroy duplicates of the value in non-unique association ends.

Associations

- destroyAt : InputPin [0..1]
  Specifies the position of an existing link to be destroyed in ordered non-unique association ends. The type of the pin is UnlimitedNatural, but the value cannot be zero or unlimited.

Constraints

[1] LinkEndDestructionData can only be end data for DestroyLinkAction or one of its specializations.

[2] LinkEndDestructionData for ordered non-unique association ends must have a single destroyAt input pin if isDestroyDuplicates is false. It must be of type UnlimitedNatural and have a multiplicity of 1..1; otherwise, the action has no input pin for the removal position.

Semantics

See "DestroyLinkAction (from IntermediateActions)" on page 253, also see "LinkAction (from IntermediateActions)" on page 256 and all of its subclasses.

Notation

No specific notation

Rationale

LinkEndDestructionData is introduced to indicate which links to destroy for ordered non-unique ends.

11.3.25 MultiplicityElement (from BasicActions)

Generalizations

- "MultiplicityElement (from Kernel)" on page 94 *(merge increment)*

Operations

[1] The operation compatibleWith takes another multiplicity as input. It checks if one multiplicity is compatible with another.

```
compatibleWith(other : Multiplicity) : Boolean;
compatibleWith(other) = Integer.allInstances()->
            forAll(i : Integer | self.includesCardinality(i) implies other.includesCardinality(i))
```

[2] The operation determines if the upper and lower bound of the ranges are the ones given.

```
is(lowerbound : integer, upperbound : integer) : Boolean;
is(lowerbound, upperbound) = (lowerbound = self.lowerbound and upperbound = self.upperbound)
```

11.3.26 OpaqueAction (from BasicActions)

Generalizations

- “Action (from BasicActions)” on page 236

Description

An action with implementation-specific semantics.

Attributes

- body : String [0..*] {ordered}
  Specifies the action in one or more languages.
- language : String [0..*] {ordered}
  Languages the body strings use, in the same order as the body strings.

Associations

- inputValue : InputPin [0..*]
  Provides input to the action. (Specializes *Action::input*)
- outputValue : OutputPin [0..*]
  Takes output from the action. (Specializes *Action::output*)

Constraints

No additional constraints

Semantics

The semantics of the action are determined by the implementation.

Notation

No specific notation

Rationale

OpaqueAction is introduced for implementation-specific actions or for use as a temporary placeholder before some other action is chosen.

11.3.27 OutputPin (from BasicActions)

Generalizations

- “Pin (from BasicActions)” on page 263

Description

An output pin is a pin that holds output values produced by an action.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

An action cannot terminate itself if an output pin has fewer values than the lower multiplicity. An action may not put more values in an output pin in a single execution than the upper multiplicity of the pin.

Notation

No specific notation. See extensions in Activities.

Changes from previous UML

OutputPin is the same concept as in UML 1.5, but modeled independently of the behaviors that use it.

11.3.28 Pin (from BasicActions)

Generalizations

- “MultiplicityElement (from BasicActions)” on page 261
- “TypedElement (from Kernel)” on page 136

Description

A pin is a typed element and multiplicity element that provides values to actions and accepts result values from them.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] If the action is an invocation action, the number and types of pins must be the same as the number of parameters and types of the invoked behavior or behavioral feature. Pins are matched to parameters by order.

Semantics

A pin represents an input to an action or an output from an action. The definition on an action assumes that pins are ordered.

Pin multiplicity controls action execution, not the number of tokens in the pin (see upperBound on "ObjectNode (from BasicActivities, CompleteActivities)" on page 393). See "InputPin (from BasicActions)" and "OutputPin (from BasicActions)" for semantics of multiplicity. Pin multiplicity is not unique, because multiple tokens with the same value can reside in an object node.

Notation

No specific notation. See extensions in Activities.

Rationale

Pins are introduced to model inputs and outputs of actions.

Changes from previous UML

Pin is the same concept as in UML 1.5, but modeled independently of the behaviors that use it.

11.3.29 QualifierValue (from CompleteActions)

QualifierValue is not an action. It is an element that identifies links. It gives a single qualifier within a link end data specification. See LinkEndData.

Generalizations

- "Element (from Kernel)" on page 64

Description

A link cannot be passed as a runtime value to or from an action. Instead, a link is identified by its end objects and qualifier values, as required. This requires more than one piece of data, namely, the end in the user model, the object on the end, and the qualifier values for that end. These pieces are brought together around LinkEndData. Each association end is identified separately with an instance of the LinkEndData class.

Attributes

No additional attributes

Associations

- qualifier : Property [1..1]
  Attribute representing the qualifier for which the value is to be specified.
- value : InputPin [1..1]
  Input pin from which the specified value for the qualifier is taken.

Constraints

[1] The qualifier attribute must be a qualifier of the association end of the link-end data.
self.LinkEndData.end->collect(qualifier)->includes(self.qualifier)

[2] The type of the qualifier value input pin is the same as the type of the qualifier attribute.
self.value.type = self.qualifier.type

[3] The multiplicity of the qualifier value input pin is "1..1."
self.value.multiplicity.is(1,1)

Semantics

See LinkAction and its children.

Notation

No specific notation

Rationale

QualifierValue is introduced to indicate which inputs are for which link end qualifiers.

Changes from previous UML

QualifierValue is unchanged from UML 1.5.

11.3.30 RaiseExceptionAction (from StructuredActions)

Generalizations

- "Action (from BasicActions)" on page 236

Description

RaiseExceptionAction is an action that causes an exception to occur. The input value becomes the exception object.

Attributes

No additional attributes

Associations

- exception : InputPin [1..1]
  An input pin whose value becomes an exception object. {Subsets *Action::input*}

Semantics

When a raise exception action is executed, the value on the input pin is raised as an exception. The value may be copied in this process, so identity may not be preserved. Raising the exception terminates the immediately containing structured node or activity and begins a search of enclosing nested scopes for an exception handler that matches the type of the exception object. See "ExceptionHandler (from ExtraStructuredActivities)" on page 363 for details of handling exceptions.

Notation

No specific notation

Rationale

Raise exception action allows models to generate exceptions; otherwise, the only exception types would be predefined built-in exception types, which would be too restrictive.

Changes from previous UML

RaiseExceptionAction replaces JumpAction from UML 1.5. Their behavior is essentially the same, except that it is no longer needed for performing simple control constructs such as break and continue.

11.3.31 ReadExtentAction (from CompleteActions)

Generalizations

- “Action (from BasicActions)” on page 236

Description

ReadExtentAction is an action that retrieves the current instances of a classifier.

Attributes

No additional attributes

Associations

- classifier : Classifier [1..1]
  The classifier whose instances are to be retrieved.
- result : OutputPin [1..1]
  The runtime instances of the classifier. {Subsets *Action::input*}

Constraints

[1] The type of the result output pin is the classifier.

[2] The multiplicity of the result output pin is “0..*.”

```
self.result.multiplicity.is(0,#null)
```

Semantics

The extent of a classifier is the set of all instances of a classifier that exist at any one time.

Semantic Variation Points

It is not generally practical to require that reading the extent produce all the instances of the classifier that exist in the entire universe. Rather, an execution engine typically manages only a limited subset of the total set of instances of any classifier and may manage multiple distributed extents for any one classifier. It is not formally specified which managed extent is actually read by a ReadExtentAction.

Notation

No specific notation

Rationale

ReadExtentAction is introduced to provide access to the runtime instances of a classifier.

Changes from previous UML

ReadExtentAction is unchanged from UML 1.5.

11.3.32 ReadIsClassifiedObjectAction (from CompleteActions)

ReadIsClassifiedObjectAction is an action that determines whether a runtime object is classified by a given classifier.

Generalizations

- "Action (from BasicActions)" on page 236

Description

This action tests the classification of an object against a given class. It can be restricted to testing direct instances.

Attributes

- isDirect : Boolean [1..1]
  Indicates whether the classifier must directly classify the input object. The default value is *false*.

Associations

- classifier : Classifier [1..1]
  The classifier against which the classification of the input object is tested.
- object : InputPin [1..1]
  Holds the object whose classification is to be tested. (Subsets *Action.input*.)
- result : OutputPin [1..1]
  After termination of the action, will hold the result of the test. (Subsets *Action.output*.)

Constraints

[1] The multiplicity of the input pin is 1..1.
```
self.object.multiplicity.is(1,1)
```
[2] The input pin has no type.
```
self.object.type->isEmpty()
```
[3] The multiplicity of the output pin is 1..1.
```
self.result.multiplicity.is(1,1)
```
[4] The type of the output pin is Boolean.
```
self.result.type = Boolean
```

Semantics

The action returns *true* if the input object is classified by the specified classifier. It returns *true* if the *isDirect* attribute is *false* and the input object is classified by the specified classifier, or by one of its (direct or indirect) descendents; otherwise, the action returns *false*.

Notation

No specific notation

Rationale

ReadisClassifiedObjectAction is introduced for run-time type identification.

Changes from previous UML

ReadisClassifiedObjectAction is unchanged from UML 1.5.

11.3.33 ReadLinkAction (from IntermediateActions)

ReadLinkAction is a link action that navigates across associations to retrieve objects on one end.

Generalizations

- "LinkAction (from IntermediateActions)" on page 256

Description

This action navigates an association towards one end, which is the end that does not have an input pin to take its object (the "open" end). The objects put on the result output pin are the ones participating in the association at the open end, conforming to the specified qualifiers, in order if the end is ordered. The semantics is undefined for reading a link that violates the navigability or visibility of the open end.

Attributes

No additional attributes

Associations

- result : OutputPin [1]
  The pin on which are put the objects participating in the association at the end not specified by the inputs. (Subsets *Action::output*)

Constraints

[1] Exactly one link-end data specification (the "open" end) must not have an end object input pin.

```
self.endData->select(ed | ed.value->size() = 0)->size() = 1
```

[2] The type and ordering of the result output pin are the same as the type and ordering of the open association end.

```
let openend : AssociationEnd = self.endData->select(ed | ed.value->size() = 0)->asSequence()->first().end in
    self.result.type = openend.type
    and self.result.ordering = openend.ordering
```

[3] The multiplicity of the open association end must be compatible with the multiplicity of the result output pin.

```
let openend : AssociationEnd = self.endData->select(ed | ed.value->size() = 0)->asSequence()->first().end in
    openend.multiplicity.compatibleWith(self.result.multiplicity)
```

[4] The open end must be navigable.

```
let openend : AssociationEnd = self.endData->select(ed | ed.value->size() = 0)->asSequence()->first().end in
    openend.isNavigable()
```

[5] Visibility of the open end must allow access to the object performing the action.

```
let host : Classifier = self.context in
let openend : AssociationEnd = self.endData->select(ed | ed.value->size() = 0)->asSequence()->first().end in
    openend.visibility = #public
    or self.endData->exists(oed | not oed.end = openend
                                  and (host = oed.end.participant
                                       or (openend.visibility = #protected
                                           and host.allSupertypes->includes(oed.end.participant))))
```

Semantics

Navigation of a binary association requires the specification of the source end of the link. The target end of the link is not specified. When qualifiers are present, one navigates to a specific end by giving objects for the source end of the association and qualifier values for all the ends. These inputs identify a subset of all the existing links of the association that match the end objects and qualifier values. The result is the collection of objects for the end being navigated towards, one object from each identified link.

In a ReadLinkAction, generalized for n-ary associations, one of the link-end data must have an unspecified object (the “open” end). The result of the action is a collection of objects on the open end of links of the association, such that the links have the given objects and qualifier values for the other ends and the given qualifier values for the open end. The order of the retrieved values in the output pin is the same as the ordering of the values of the links. This result is placed on the output pin of the action, which has a type and ordering given by the open end. The multiplicity of the open end must be compatible with the multiplicity of the output pin. For example, the modeler can set the multiplicity of this pin to support multiple values even when the open end only allows a single value. This way the action model will be unaffected by changes in the multiplicity of the open end. The semantics are defined only when the open end is navigable, and visible to the host object of the action.

Notation

No specific notation

Rationale

ReadLinkAction is introduced to navigate across links.

Changes from previous UML

ReadLinkAction is unchanged from UML 1.5.

11.3.34 ReadLinkObjectEndAction (from CompleteActions)

ReadLinkObjectEndAction is an action that retrieves an end object from a link object.

Generalizations

- “Action (from BasicActions)” on page 236

Description

This action reads the object on an end of a link object. The association end to retrieve the object from is specified statically, and the link object to read is provided on the input pin at run time.

Attributes

No additional attributes

Associations

- end : Property [1..1]
  Link end to be read.
- object : InputPin [1..1]
  Gives the input pin from which the link object is obtained. {Subsets *Action::input*}
- result : OutputPin [1..1]
  Pin where the result value is placed. {Subsets *Action::output*}

Constraints

[1] The property must be an association end.
self.end.association.notEmpty()

[2] The association of the association end must be an association class.
self.end.Association.oclIsKindOf(AssociationClass)

[3] The ends of the association must not be static.
self.end.association.memberEnd->forall(e | **not** e.isStatic)

[4] The type of the object input pin is the association class that owns the association end.
self.object.type = self.end.association

[5] The multiplicity of the object input pin is “1..1.”
self.object.multiplicity.is(1,1)

[6] The type of the result output pin is the same as the type of the association end.
self.result.type = self.end.type

[7] The multiplicity of the result output pin is 1..1.
self.result.multiplicity.is(1,1)

Semantics

ReadLinkObjectEndAction retrieves an end object from a link object. The value of the specified end of the input link object is placed on the output pin of the action. Note that this is *not* the same as reading links of the link object’s association with the specified end as the open end. Identifying a link object explicitly identifies a single specific link, independently of the values of link ends other than the one specified to be read. Even if the multiplicity of the specified end is different from 1..1 in the association, it only has a single value from the point of view of a specified link object. This is why the output pin of a ReadLinkObjectEndAction always has a multiplicity of 1..1.

Notation

No specific notation

Rationale

ReadLinkObjectEndAction is introduced to navigate from a link object to its end objects.

Changes from previous UML

ReadLinkObjectEndAction is unchanged from UML 1.5.

11.3.35 ReadLinkObjectEndQualifierAction (from CompleteActions)

ReadLinkObjectEndAction is an action that retrieves a qualifier end value from a link object.

Generalizations

- "Action (from BasicActions)" on page 236

Description

This action reads a qualifier value or values on an end of a link object. The association end to retrieve the qualifier from is specified statically, and the link object to read is provided on the input pin at run time.

Attributes

No additional attributes

Associations

- qualifier : Property [1..1]
  The attribute representing the qualifier to be read.
- object : InputPin [1..1]
  Gives the input pin from which the link object is obtained. (Subsets *Action::input*)
- result : OutputPin [1..1]
  Pin where the result value is placed. (Subsets *Action::output*)

Constraints

[1] The qualifier attribute must be a qualifier attribute of an association end.
self.qualifier.associationEnd->size() = 1

[2] The association of the association end of the qualifier attribute must be an association class.
self.qualifier.associationEnd.association.oclIsKindOf(AssociationClass)

[3] The ends of the association must not be static.
self.qualifier.associationEnd.association.memberEnd->forall(e | **not** e.isStatic)

[4] The type of the object input pin is the association class that owns the association end that has the given qualifier attribute.
self.object.type = self.qualifier.associationEnd.association

[5] The multiplicity of the qualifier attribute is 1..1.
self.qualifier.multiplicity.is(1,1)

[6] The multiplicity of the object input pin is “1..1.”

```
self.object.multiplicity.is(1,1)
```

[7] The type of the result output pin is the same as the type of the qualifier attribute.

```
self.result.type = self.qualifier.type
```

[8] The multiplicity of the result output pin is “1..1.”

```
self.result.multiplicity.is(1,1)
```

Semantics

ReadLinkObjectEndAction retrieves a qualifier end value from a link object.

Notation

No specific notation

Rationale

ReadLinkObjectEndQualifierAction is introduced to navigate from a link object to its end objects.

Changes from previous UML

ReadLinkObjectEndQualifierAction is unchanged from UML 1.5, except the name was corrected from ReadLinkObjectQualifierAction.

11.3.36 ReadSelfAction (from IntermediateActions)

ReadSelfAction is an action that retrieves the host object of an action.

Generalizations

- “Action (from BasicActions)” on page 236

Description

Every action is ultimately a part of some behavior, which is in turn optionally attached in some way to the specification of a classifier (for example, as the body of a method or as part of a state machine). When the behavior executes, it does so in the context of some specific host instance of that classifier. This action produces this host instance, if any, on its output pin. The type of the output pin is the classifier to which the behavior is associated in the user model.

Attributes

No additional attributes

Associations

- result : OutputPin [1..1]
  Gives the output pin on which the hosting object is placed. (Subsets *Action::output*)

Constraints

[1] The action must be contained in a behavior that has a host classifier.

```
self.context->size() = 1
```

[2] If the action is contained in a behavior that is acting as the body of a method, then the operation of the method must not be static.

[3] The type of the result output pin is the host classifier.

```
self.result.type = self.context
```

[4] The multiplicity of the result output pin is “1..1.”

```
self.result.multiplicity.is(1,1)
```

Semantics

Every action is part of some behavior, as are behaviors invoked by actions or other elements of behaviors. Behaviors are optionally attached in some way to the specification of a classifier.

For behaviors that have no other context object, the behavior itself is the context object. See behaviors as classes in Common Behaviors and discussion of reflective objects in Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities).

Notation

No specific notation

Rationale

ReadSelfAction is introduced to provide access to the context object when it is not available as a parameter.

Changes from previous UML

ReadSelfAction is unchanged from UML 1.5.

11.3.37 ReadStructuralFeatureAction (from IntermediateActions)

ReadStructuralFeatureAction is a structural feature action that retrieves the values of a structural feature.

Generalizations

- “StructuralFeatureAction (from IntermediateActions)” on page 286.

Description

This action reads the values of a structural feature in order, if the structural feature is ordered.

Attributes

No additional attributes

Associations

- result : OutputPin [1..1]
  Gives the output pin on which the result is put. (Subsets *Action::output*)

Constraints

[1] The type and ordering of the result output pin are the same as the type and ordering of the structural feature.
self.result.type = self.structuralFeature.type
and self.result.ordering = self.structuralFeature.ordering

[2] The multiplicity of the structural feature must be compatible with the multiplicity of the output pin.
self.structuralFeature.multiplicity.compatibleWith(self.result.multiplicity)

Semantics

The values of the structural feature of the input object are placed on the output pin of the action. If the feature is an association end, the semantics are the same as reading links of the association with the feature as the open end. The type and ordering of the output pin are the same as the specified structural feature. The order of the retrieved values in the output pin is the same as the ordering of the values of the structural feature. The multiplicity of the structural feature must be compatible with the multiplicity of the output pin. For example, the modeler can set the multiplicity of this pin to support multiple values even when the structural feature only allows a single value. This way the action model will be unaffected by changes in the multiplicity of the structural feature.

Notation

No specific notation

Rationale

ReadStructuralFeatureAction is introduced to retrieve the values of a structural feature.

Changes from previous UML

ReadStructuralFeatureAction is new in UML 2.0. It generalizes ReadAttributeAction from UML 1.5.

11.3.38 ReadVariableAction (from StructuredActions)

ReadVariableAction is a variable action that retrieves the values of a variable.

Generalizations

- “VariableAction (from StructuredActions)” on page 291.

Description

This action reads the values of a variable in order, if the variable is ordered.

Attributes

No additional attributes

Associations

- result : OutputPin [1..1]
  Gives the output pin on which the result is put. (Subsets *Action::output*)

Constraints

[1] The type and ordering of the result output pin of a read-variable action are the same as the type and ordering of the variable.

```
self.result.type =self.variable.type
and self.result.ordering = self.variable.ordering
```

[2] The multiplicity of the variable must be compatible with the multiplicity of the output pin.

```
self.variable.multiplicity.compatibleWith(self.result.multiplicity)
```

Semantics

The values of the variable are placed on the output pin of the action. The type and ordering of the output pin are the same as the specified variable. The order of the retrieved values in the output pin is the same as the ordering of the values of the variable. The multiplicity of the variable must be compatible with the multiplicity of the output pin. For example, the modeler can set the multiplicity of this pin to support multiple values even when the variable only allows a single value. This way the action model will be unaffected by changes in the multiplicity of the variable.

Notation

No specific notation

Rationale

ReadVariableAction is introduced to retrieve the values of a variable.

Changes from previous UML

ReadVariableAction is unchanged from UML 1.5.

11.3.39 ReclassifyObjectAction (from CompleteActions)

ReclassifyObjectAction is an action that changes which classifiers classify an object.

Generalizations

- “Action (from BasicActions)” on page 236

Description

ReclassifyObjectAction adds given classifier to an object and removes given classifiers from that object. Multiple classifiers may be added and removed at a time.

Attributes

- isReplaceAll : Boolean [1..1]
  Specifies whether existing classifiers should be removed before adding the new classifiers. The default value is *false*.

Associations

- object : InputPin [1..1]
  Holds the object to be reclassified. (Subsets *Action::input*.)

- newClassifier : Classifier [0..*]
  A set of classifiers to be added to the classifiers of the object.
- oldClassifier : Classifier [0..*]
  A set of classifiers to be removed from the classifiers of the object.

Constraints

[1] None of the new classifiers may be abstract.
not self.newClassifier->exists(isAbstract = true)

[2] The multiplicity of the input pin is 1..1.
self.argument.multiplicity.is(1,1)

[3] The input pin has no type.
self.argument.type->size() = 0

Semantics

After the action completes, the input object is classified by its existing classifiers and the "new" classifiers given to the action; however, the "old" classifiers given to the actions no longer classify the input object. The identity of the object is preserved, no behaviors are executed, and no initial expressions are evaluated. "New" classifiers replace existing classifiers in an atomic step, so that structural feature values and links are not lost during the reclassification, when the "old" and "new" classifiers have structural features and associations in common.

Neither adding a classifier that duplicates an already existing classifier, nor removing a classifier that is not classifying the input object, has any effect. Adding and removing the same classifiers has no effect.

If *isReplaceAll* is *true*, then the existing classifiers are removed before the "new" classifiers are added, except if the "new" classifier already classifies the input object, in which case this classifier is not removed. If *isReplaceAll* is *false*, then adding an existing value has no effect.

It is an error, if any of the "new" classifiers is abstract or if all classifiers are removed from the input object.

Notation

No specific notation

Rationale

ReclassifyObjectAction is introduced to change the classifiers of an object.

Changes from previous UML

ReclassifyObjectAction is unchanged from UML 1.5.

11.3.40 ReduceAction (from CompleteActions)

(CompleteActions) ReduceAction is an action that reduces a collection to a single value by combining the elements of the collection.

Generalizations

- "Action (from BasicActions)" on page 236

Description

This action takes a collection as input and produces an output by applying a behavior with two inputs pairwise to the elements of the collection.

Attributes

- isOrdered : Boolean = false
  Tells whether the order of the input collection should determine the order in which the behavior is applied to its elements.

Associations

- collection : InputPin [1]
  The collection to be reduced (subsets Action::input)
- reducer : Behavior [1]
  Behavior that is applied to two elements of the input collection to produce a value that is the same type as elements of the collection.
- result : OutputPin [1]
  Gives the output pin on which the result is put (subsets Action::output).

Constraints

[1] The type of the input must be a collection.

[2] The type of the output must be compatible with the type of the output of the reducer behavior.

[3] The reducer behavior must have two input parameters and one output parameter, of types compatible with the types of elements of the input collection.

Semantics

The behavior is invoked repeatedly on pairs of elements in the input collection. Each time it is invoked, it produces one output that is put back in an intermediate version of the collection. This repeats until the collection is reduced to a single value, which is the output of the action.

If isOrdered is false, the order in which the behavior is applied to pairs of values is indeterminate. This will not affect the result of the action if the behavior is commutative and associative, see below. If separate invocations of the behavior affect each other, for example, through side-effects, the result of the actions may be unpredictable, however. If the reducing behavior is not commutative and associative, as with matrix multiplication, the order of the elements in the collection will affect the result of the behavior and the action. In this case, isOrdered should be set to true, so the behavior will be applied to adjacent pairs according to the collection order. The result of each invocation of the behavior replaces the two values taken as input in the same position in the order as the two values. If isOrdered = false, the reducer behavior should be commutative and associative so it will produce the same reduced value regardless of which two elements are paired at each invocation. For example, addition is commutative because a + b = b + a. It is also associative because ((a + b) + c) = (a + (b + c)). Commutativity and associativity are not required, but the result will be indeterminate if isOrdered = false.

Notation

None

Examples

ReduceAction can be used to reduce a list of numbers to the sum of the numbers. It would have one input pin for a collection of numbers, one result pin 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 reduce action to this collection with an addition function is 11. This can be computed in a number of ways, for example, (((2+7) + 5) + -3), (2 + (7 + (5 + -3))), ((2 + 7) + (5 + -3)).

Rationale

The purpose of ReduceAction is to specify the transformation of a collection to a single value by pairwise application of a behavior, without necessarily committing to the order in which the pairs are chosen.

Changes from previous UML

ReduceAction replaces ReduceAction in UML 1.5. It has the same functionality, except it takes one collection instead of multiple as input, and produces one result instead of multiple. The effect of multiple input collections can be achieved in UML 2 with an input that is a collection of collections, where the nested collections are created by taking one element from each of the multiple collection inputs to the UML 1.5 ReduceAction.

11.3.41 RemoveStructuralFeatureValueAction (from IntermediateActions)

RemoveStructuralFeatureValueAction is a write structural feature action that removes values from structural features.

Generalizations

- "WriteStructuralFeatureAction (from IntermediateActions)" on page 293

Description

The object to access is specified dynamically, by referring to an input pin on which the object will be placed at runtime. The type of the value of this pin is the classifier that owns the specified structural feature, and the value's multiplicity is 1..1.

Structural features are potentially multi-valued and ordered, and may support duplicates, so the action supports specification of removal points for new values. It also supports the removal of all duplicate values.

Attributes

- isRemoveDuplicates : Boolean = false [1..1]
  Specifies whether to remove duplicates of the value in non-unique structural features.

Associations

- removeAt : InputPin [0..1]
  Specifies the position of an existing value to remove in ordered non-unique structural features. The type of the pin is UnlimitednNatural, but the value cannot be zero or unlimited. {Subsets *Action::input*}

Constraints

[1] Actions removing a value from ordered non-unique structural features must have a single removeAt input pin if isRemoveDuplicates is false. It must be of type Unlimited Natural with multiplicity 1..1; otherwise, the action has no removeAt input pin.

Semantics

Structural features are potentially multi-valued. Removing a value succeeds even when it violates the minimum multiplicity. Removing a value that does not exist has no effect. If the feature is an association end, the semantics are the same as for destroying links, the participants of which are the object owning the structural feature and the value being removed.

Values of a structural feature may be duplicate in non-unique structural features. The isRemoveDuplicates attribute indicates whether to remove all duplicates of the specified value. The removeAt input pin is required if isRemoveDuplicates is false in ordered non-unique structural features. It indicates the position of an existing value to remove. It must be a positive integer less than or equal to the current number of values. The semantics is undefined for zero or an integer greater than the number of existing values, and for unlimited.

The semantics is undefined for removing an existing value for a structural feature with isReadOnly=true. The semantics is undefined for removing an existing value of a structural feature with settability readOnly after initialization of the owning object.

Notation

No specific notation

Rationale

RemoveStructuralFeatureValueAction is introduced to remove structural feature values.

Changes from previous UML

RemoveStructuralFeatureValueAction is new in UML 2.0.

11.3.42 RemoveVariableValueAction (from StructuredActions)

RemoveVariableValueAction is a write variable action that removes values from variables.

Generalizations

- “WriteVariableAction (from StructuredActions)” on page 294

Description

One value is removed from the set of possible variable values.

Attributes

- isRemoveDuplicates : Boolean = false [1..1]
  Specifies whether to remove duplicates of the value in non-unique variables.

Associations

- removeAt : InputPin [0..1]
  Specifies the position of an existing value to remove in ordered non-unique variables. The type of the pin is UnlimitedNatural, but the value cannot be zero or unlimited. {Subsets *Action::input*}

Constraints

[1] Actions removing a value from ordered non-unique variables must have a single removeAt input pin if isRemoveDuplicates is false. It must be of type UnlimitedNatural with multiplicity of 1..1; otherwise, the action has no removeAt input pin.

Semantics

Variables are potentially multi-valued. Removing a value succeeds even when it violates the minimum multiplicity. Removing a value that does not exist has no effect. Variables are potentially multi-valued and ordered, and may support duplicates, so the action supports specification of removal points for new values. It also supports the removal of all duplicate values.

Values of a variable may be duplicate in non-unique variables. The isRemoveDuplicates attribute indicates whether to remove all duplicates of the specified value. The removeAt input pin is required if isRemoveDuplicates is false in ordered non-unique variables. It indicates the position of an existing value to remove. It must be a positive integer less than or equal to the current number of values. The semantics is undefined for zero, for an integer greater than the number of existing values and for unlimited.

Notation

No specific notation

Rationale

RemoveVariableValueAction is introduced to remove variable values.

Changes from previous UML

RemoveVariableValueAction is unchanged from UML 1.5.

11.3.43 ReplyAction (from CompleteActions)

Generalizations

- “Action (from BasicActions)” on page 236

Description

ReplyAction is an action that accepts a set of return values and a value containing return information produced by a previous accept call action. The reply action returns the values to the caller of the previous call, completing execution of the call.

Attributes

No additional attributes

Associations

- replyToCall : Trigger [1..1]
  The trigger specifying the operation whose call is being replied to.

- replyValue : InputPin [0..*]
  A list of pins containing the reply values of the operation. These values are returned to the caller. {Subsets *Action::input*}
- returnInformation : InputPin [1..1]
  A pin containing the return information value produced by an earlier AcceptCallAction. {Subsets *Action::input*}

Constraints

[1] The reply value pins must match the return, out, and inout parameters of the operation on the event on the trigger in number, type, and order.

[2] The event on replyToCall trigger must be a CallEvent.

```
replyToCallEvent.oclIsKindOf(CallEvent)
```

Semantics

The execution of a reply action completes the execution of a call that was initiated by a previous AcceptCallAction. The two are connected by the returnInformation value, which is produced by the AcceptCallAction and consumed by the ReplyAction. The information in this value is used by the execution engine to return the reply values to the caller and to complete execution of the original call. The details of transmitting call requests, encoding return information, and transmitting replies are opaque and unavailable to models, therefore they need not be and are not specified in this document.

Return information may be copied, stored in objects, and passed around, but it may only be used in a reply action once. If the same return information value is supplied to a second ReplyAction, the execution is in error and the behavior of the system is unspecified. It is not intended that any profile give any other meaning the return information. The operation specified by the call event on the trigger must be consistent with the information returned at runtime.

If the return information is lost to the execution or if a reply is never made, the caller will never receive a reply and therefore will never complete execution. This is not inherently illegal but it represents an unusual situation at the very least.

11.3.44 SendObjectAction (from IntermediateActions)

Generalizations

- “InvocationAction (from BasicActions)” on page 256

Description

SendObjectAction is an action that transmits an object to the target object, where it may invoke behavior such as the firing of state machine transitions or the execution of an activity. The value of the object is available to the execution of invoked behaviors. The requestor continues execution immediately. Any reply message is ignored and is not transmitted to the requestor.

Attributes

No additional attributes

Associations

- request: InputPin [1]
  The request object, which is transmitted to the target object. The object may be copied in transmission, so identity might not be preserved. (Redefines *InvocationActon.::argument*)
- target: InputPin [1]
  The target object to which the object is sent. (Subsets *Action::input*)

Constraints

No additional constraints

Semantics

[1] When all the control and data flow prerequisites of the action execution are satisfied, the object on the input pin is transmitted to the target object. The target object may be local or remote. The object on the input pin may be copied during transmission, so identity might not be preserved. The manner of transmitting the object, the amount of time required to transmit it, the order in which the transmissions reach the various target objects, and the path for reaching the target objects are undefined.

[2] When a transmission arrives at a target object, it may invoke behavior in the target object. The effect of receiving an object is specified in 13, “Common Behaviors.” Such effects include executing activities and firing state machine transitions.

[3] A send object action receives no reply from the invoked behavior; any attempted reply is simply ignored, and no transmission is performed to the requestor.

Notation

No specific notation

Presentation Options

If the activity in which a send object action is used will always send a signal, then the SendSignalAction notation can be used.

Rationale

Sends any object to a specified target object.

Changes from previous UML

SendObjectAction is new in UML 2.0.

11.3.45 SendSignalAction (from BasicActions)

Generalizations

- “InvocationAction (from BasicActions)” on page 256

Description

SendSignalAction is an action that creates a signal instance from its inputs, and transmits it to the target object, where it may cause the firing of a state machine transition or the execution of an activity. The argument values are available to the execution of associated behaviors. The requestor continues execution immediately. Any reply message is ignored and is not transmitted to the requestor. If the input is already a signal instance, use SendObjectAction.

Attributes

No additional attributes

Associations

- signal: Signal [1]
  The type of signal transmitted to the target object.
- target: InputPin [1]
  The target object to which the signal is sent. {Subsets *Action::input*}

Constraints

[1] The number and order of argument pins must be the same as the number and order of attributes in the signal.

[2] The type, ordering, and multiplicity of an argument pin must be the same as the corresponding attribute of the signal.

Semantics

[1] When all the prerequisites of the action execution are satisfied, a signal instance of the type specified by *signal* is generated from the argument values and this signal instance is transmitted to the identified target object. The target object may be local or remote. The signal instance may be copied during transmission, so identity might not be preserved. The manner of transmitting the signal object, the amount of time required to transmit it, the order in which the transmissions reach the various target objects, and the path for reaching the target objects are undefined.

[2] When a transmission arrives at a target object, it may invoke behavior in the target object. The effect of receiving a signal object is specified in 13, “Common Behaviors.” Such effects include executing activities and firing state machine transitions.

[3] A send signal action receives no reply from the invoked behavior; any attempted reply is simply ignored, and no transmission is performed to the requestor.

Notation

A send signal action is notated with a convex pentagon.

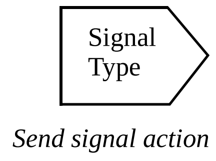


Figure 11.23 - Send signal notation

Examples

See extension in “SendSignalAction (as specialized)” on page 407.

Rationale

Sends a signal to a specified target object.

Changes from previous UML

Same as UML 1.5.

11.3.46 StartClassifierBehaviorAction (from CompleteActions)

Generalizations

- “Action (from BasicActions)” on page 236

Description

StartClassifierBehaviorAction is an action that starts the classifier behavior of the input.

Attributes

No additional attributes

Associations

- object : InputPin [1..1]
  Holds the object on which to start the owned behavior. (Subsets *Action::input.*)

Constraints

[1] The multiplicity of the input pin is 1..1.

[2] If the input pin has a type, then the type must have a classifier behavior.

Semantics

When a StartClassifierBehaviorAction is invoked, it initiates the classifier behavior of the classifier of the input object. If the behavior has already been initiated, or the object has no classifier behavior, this action has no effect.

Notation

No specific notation

Rationale

This action is provided to permit the explicit initiation of classifier behaviors, such as state machines and code, in a detailed, low-level “raw” specification of behavior.

Changes from previous UML

StartClassifierBehaviorAction is a generalization of the UML 1.5 StartStateMachineAction.

11.3.47 StartObjectBehaviorAction (from CompleteActions)

Generalizations

- "CallAction (from BasicActions)" on page 242

Description

StartObjectBehaviorAction is an action that starts the execution either of a directly instantiated behavior or of the classifier behavior 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.

Attributes

No additional attributes.

Associations

- object : InputPin [0..1]
  Holds the object which is either a behavior to be started or has a classifier behavior to be started. (Subsets *Action::input*)

Constraints

[1] The type of the object input pin must be either a Behavior or a BehavioredClassifier with a classifier behavior.

[2] The multiplicity of the object input pin must be [1..1].

[3] The number and order of the argument pins must be the same as the number and order of the in and in-out parameters of the invoked behavior. Pins are matched to parameters by order.

[4] The number and order of result pins must be the same as the number and order of the in-out, out and return parameters of the invoked behavior. Pins are matched to parameters by order.

[5] The type, ordering, and multiplicity of an argument or result pin must be the same as the corresponding parameter of the behavior.

Semantics

A StartObjectBehaviorAction invokes an instantiated behavior or the classifier behavior of the input object. If the input object is an instantiated behavior that is not already executing, then it begins executing. If the input object has a classifier behavior that is not already executing, then it is instantiated and started. In either case, if the invoked behavior has already been initiated, then the action has no effect.

Note that, if the input object is not an instantiated behavior, then it must have a classifier behavior. If the input object is an instantiated behavior, then it may also have a classifier behavior, which is also started. If this classifier behavior itself has a classifier behavior, then this is also recursively started, and so on.

As a kind of CallAction, a StartObjectBehaviorAction must also provide argument values for all the in and inout parameters of the invoked behavior. Argument values provided on the input pins are available to the execution of the invoked behavior. If the invoked behavior is started asynchronously, StartObjectBehaviorAction completes after the behavior starts. If the invoked behavior is started synchronously, StartObjectBehaviorAction completes after the behavior does, and if the behavior has output parameters, then values produced for those parameters during execution are available on the result output pins of the action.

11.3.48 StructuralFeatureAction (from IntermediateActions)

(IntermediateActions) StructuralFeatureAction is an abstract class for all structural feature actions.

Generalizations

- “Action (from BasicActions)” on page 236

Description

This abstract action class statically specifies the structural feature being accessed.

The object to access is specified dynamically, by referring to an input pin on which the object will be placed at runtime. The type of the value of this pin is the classifier that owns the specified structural feature, and the value’s multiplicity is 1..1.

Attributes

No additional attributes

Associations

- structuralFeature : StructuralFeature [1..1]
  Structural feature to be read.
- object : InputPin [1..1]
  Gives the input pin from which the object whose structural feature is to be read or written is obtained. (Subsets *Action::input*)

Constraints

[1] The structural feature must not be static.

```
self.structuralFeature.isStatic = #false
```

[2] The type of the object input pin is the same as the classifier of the object passed on this pin.

[3] The multiplicity of the input pin must be 1..1.

```
self.object.multiplicity.is(1,1)
```

[4] Visibility of structural feature must allow access to the object performing the action.

```
let host : Classifier = self.context in
    self.structuralFeature.visibility = #public
    or host = self.structuralFeature.featuringClassifier.type
    or (self.structuralFeature.visibility = #protected and host.allSupertypes
        ->includes(self.structuralFeature.featuringClassifier.type)))
```

[5] A structural feature has exactly one featuringClassifier.

```
self.structuralFeature.featuringClassifier->size() = 1
```

Semantics

A structural feature action operates on a statically specified structural feature of some classifier. The action requires an object on which to act, provided at runtime through an input pin. If the structural feature is an association end, then actions on the feature have the same semantics as actions on the links that have the feature as an end. See specializations of StructuralFeatureAction. The semantics is undefined for accessing a structural feature that violates its visibility. The semantics for static features are undefined.

The structural features of an object may change over time due to dynamic classification. However, the structural feature specified in a structural feature action is inherited from a single classifier, and it is assumed that the object passed to a structural feature action is classified by that classifier directly or indirectly. The structural feature is referred to as a user model element, so it is uniquely identified, even if there are other structural features of the same name on other classifiers.

Notation

No specific notation

Rationale

StructuralFeatureAction is introduced for the abstract aspects of structural feature actions.

Changes from previous UML

StructuralFeatureAction is new in UML 2.0. It generalizes AttributeAction in UML 1.5.

11.3.49 TestIdentityAction (from IntermediateActions)

TestIdentifyAction is an action that tests if two values are identical objects.

Generalizations

- “Action (from BasicActions)” on page 236

Description

This action returns true if the two input values are the same identity, false if they are not.

Attributes

No additional attributes

Associations

- first: InputPin [1..1]
  Gives the pin on which an object is placed. (Subsets *Action::input*)
- result: OutputPin [1..1])
  Tells whether the two input objects are identical. (Subsets *Action::output*)
- second: InputPin [1..1]
  Gives the pin on which an object is placed. (Subsets *Action::input*)

Constraints

[1] The input pins have no type.

```
self.first.type->size() = 0
and self.second.type->size() = 0
```

[2] The multiplicity of the input pins is 1..1.

```
self.first.multiplicity.is(1,1)
and self.second.multiplicity.is(1,1)
```

[3] The type of the result is Boolean.

```
self.result.type.oclIsTypeOf(Boolean)
```

Semantics

When all the prerequisites of the action have been satisfied, the input values are obtained from the input pins and made available to the computation. If the two input values represent the same object (regardless of any implementation-level encoding), the value true is placed on the output pin of the action execution; otherwise, the value false is placed on the output pin. The execution of the action is complete.

Notation

No specific notation

Rationale

TestIdentityAction is introduced to tell when two values refer to the same object.

Changes from previous UML

TestIdentityAction is unchanged from UML 1.5.

11.3.50 UnmarshallAction (from CompleteActions)

UnmarshallAction is an action that breaks an object of a known type into outputs each of which is equal to a value from a structural feature of the object.

Generalizations

- “Action (from BasicActions)” on page 236

Description

The outputs of this action correspond to the structural features of the specified type. The input must be of this type.

Attributes

No additional attributes

Associations

- object : InputPin [1..1]
  The object to be unmarshalled. {Subsets *Action::input*}

- unmarshallType : Classifier [1..1]
  The type of the object to be unmarshalled.
- result : OutputPin [1..*]
  The values of the structural features of the input object. {Subsets *Action::output*}

Constraints

[1] The type of the object input pin must be the same as the unmarshall classifier.

[2] The multiplicity of the object input pin is 1..1.

[3] The number of result output pins must be the same as the number of structural features of the unmarshall classifier.

[4] The type and ordering of each result output pin must be the same as the corresponding structural features of the unmarshall classifier.

[5] The multiplicity of each result output pin must be compatible with the multiplicity of the corresponding structural features of the unmarshall classifier.

[6] The unmarshall classifier must have at least one structural feature.

[7] unmarshallType must be a Classifier with ordered attributes.

Semantics

When an object is available on the input pin, the values of the structural features of the specified classifier are retrieved from the object and placed on the output pins, in the order of the structural features of the specified classifier. The order of the values in an output pin are the same as the order of the corresponding structural features, if any.

Notation

No specific notation

Examples

See "UnmarshallAction (as specialized)" on page 411.

Rationale

UnmarshallAction is introduced to read all the structural features of an object at once.

Changes from previous UML

UnmarshallAction is the same as UML 1.5, except that the name of the metaassociation to the input pin is changed.

11.3.51 ValuePin (from BasicActions)

Generalizations

- "InputPin (from BasicActions)" on page 255

Description

A value pin is an input pin that provides a value by evaluating a value specification.

Attributes

No additional attributes

Associations

- value : ValueSpecification [1..1]
  Value that the pin will provide.

Constraints

[1] The type of value specification must be compatible with the type of the value pin.

Semantics

The value of the pin is the result of evaluating the value specification.

Notation

No specific notation. See extensions in Activities.

Rationale

ValuePin is introduced to provide the most basic way of providing inputs to actions.

Changes from previous UML

ValuePin is new to UML 2.

11.3.52 ValueSpecificationAction (from IntermediateActions)

ValueSpecificationAction is an action that evaluates a value specification.

Generalizations

- "Action (from BasicActions)" on page 236

Description

The action returns the result of evaluating a value specification.

Attributes

No additional attributes

Associations

- value : ValueSpecification [1]
  Value specification to be evaluated.
- result : OutputPin [1]
  Gives the output pin on which the result is output. {Subsets *Action::output*}

Constraints

[1] The type of value specification must be compatible with the type of the result pin.

[2] The multiplicity of the result pin is 1..1.

Semantics

The value specification is evaluated when the action is enabled.

Notation

See "ValueSpecificationAction (as specialized)" on page 413.

Examples

See "ValueSpecificationAction (as specialized)" on page 413.

Rationale

ValueSpecificationAction is introduced for injecting constants and other value specifications into behavior.

Changes from previous UML

ValueSpecificationAction replaces LiteralValueAction from UML 1.5.

11.3.53 VariableAction (from StructuredActions)

Generalizations

- "Action (from BasicActions)" on page 236

Description

VariableAction is an abstract class for actions that operate on a statically specified variable.

Attributes

No additional attributes

Associations

- variable : Variable [1..1]
  Variable to be read.

Constraints

[1] The action must be in the scope of the variable.
self.variable.isAccessibleBy(self)

Semantics

Variable action is an abstract metaclass. For semantics see its concrete subtypes.

Notation

No specific notation

Rationale

VariableAction is introduced for the abstract aspects of variable actions.

Changes from previous UML

VariableAction is unchanged from UML 1.5.

11.3.54 WriteLinkAction (from IntermediateActions)

WriteLinkAction is an abstract class for link actions that create and destroy links.

Generalizations

- "LinkAction (from IntermediateActions)" on page 256

Description

A write link action takes a complete identification of a link and creates or destroys it.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] All end data must have exactly one input object pin.
self.endData.forall(value->size() = 1)

[2] The visibility of at least one end must allow access to the class using the action.

Semantics

See children of WriteLinkAction.

Notation

No specific notation

Rationale

WriteLinkAction is introduced to navigate across links.

Changes from previous UML

WriteLinkAction is unchanged from UML 1.5.

11.3.55 WriteStructuralFeatureAction (from IntermediateActions)

WriteStructuralFeatureAction is an abstract class for structural feature actions that change structural feature values.

Generalizations

- "StructuralFeatureAction (from IntermediateActions)" on page 286

Description

A write structural feature action operates on a structural feature of an object to modify its values. It has an input pin on which the value that will be added or removed is put. Other aspects of write structural feature actions are inherited from StructuralFeatureAction.

Attributes

No additional attributes

Associations

- value : InputPin [1..1])
  Value to be added or removed from the structural feature. (Subsets *Action::input*)
- result : OutputPin [0..1]
  Gives the output pin on which the result is put. {Subsets *Action::output*}

Constraints

[1] The type input pin is the same as the classifier of the structural feature.
self.value.type = self.structuralFeature.featuringClassifier

[2] The multiplicity of the input pin is 1..1.
self.value.multiplicity.is(1,1)

[3] The type of the result output pin is the same as the type of the inherited object input pin.
result->notEmpty() implies self.result.type = self.object.type

[4] The multiplicity of the result output pin must be 1..1.
result->notEmpty() implies self.result.multiplicity.is(1,1)

Semantics

A write structural feature action operates on a structural feature of an object to modify its values. The semantics of this modification depend on the specific kind of structural feature action. However, in all cases, if a result output pin is provided, then the input object, as modified, is placed on the output pin. If the input object is actually a data value, then a copy of the input data value is placed on the output pin, but with the appropriate structural feature modified.

Notation

No specific notation

Rationale

WriteStructuralFeatureAction is introduced to abstract aspects of structural feature actions that change structural feature values.

Changes from previous UML

WriteStructuralFeatureAction is new in UML 2.0. It generalizes WriteAttributeAction in UML 1.5.

11.3.56 WriteVariableAction (from StructuredActions)

WriteVariableAction is an abstract class for variable actions that change variable values.

Generalizations

- “VariableAction (from StructuredActions)” on page 291

Description

A write variable action operates on a variable to modify its values. It has an input pin on which the value that will be added or removed is put. Other aspects of write variable actions are inherited from VariableAction.

Attributes

No additional attributes

Associations

- value : InputPin [1..1]
  Value to be added or removed from the variable. (Subsets *Action::input*)

Constraints

[1] The type input pin is the same as the type of the variable.
self.value.type = self.variable.type

[2] The multiplicity of the input pin is 1..1.
self.value.multiplicity.is(1,1)

Semantics

See children of WriteVariableAction.

Notation

No specific notation

Rationale

WriteVariableAction is introduced to abstract aspects of structural feature actions that change variable values.

Changes from previous UML

WriteVariableAction is unchanged from UML 1.5.

11.4 Diagrams

See “Diagrams” on page 415.

12 Activities

12.1 Overview

Activity modeling emphasizes the sequence and conditions for coordinating lower-level behaviors, rather than which classifiers own those behaviors. These are commonly called control flow and object flow models. The actions coordinated by activity models can be initiated because other actions finish executing, because objects and data become available, or because events occur external to the flow.

Actions and activities

An *action execution* corresponds to the execution of a particular action. Similarly, an activity *execution* is the execution of an activity, ultimately including the executions of actions within it. Each action in an activity may execute zero, one, or more times for each activity execution. At the minimum, actions need access to data, they need to transform and test data, and actions may require sequencing. The activities specification (at the higher compliance levels) allows for several (logical) threads of control executing at once and synchronization mechanisms to ensure that activities execute in a specified order. Semantics based on concurrent execution can then be mapped easily into a distributed implementation. However, the fact that the UML allows for concurrently executing objects does not necessarily imply a distributed software structure. Some implementations may group together objects into a single task and execute sequentially—so long as the behavior of the implementation conforms to the sequencing constraints of the specification.

There are potentially many ways of implementing the same specification, and any implementation that preserves the information content and behavior of the specification is acceptable. Because the implementation can have a different structure from that of the specification, there is a mapping between the specification and its implementation. This mapping need not be one-to-one: an implementation need not even use object-orientation, or it might choose a different set of classes from the original specification.

The mapping may be carried out by hand by overlaying physical models of computers and tasks for implementation purposes, or the mapping could be carried out automatically. This specification neither provides the overlays, nor does it provide for code generation explicitly, but the specification makes both approaches possible.

See the “Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)” and “Action (from CompleteActivities, FundamentalActivities, StructuredActivities)” metaclasses for more introduction and semantic framework.

FundamentalActivities

The fundamental level defines activities as containing nodes, which includes actions. This level is shared between the flow and structured forms of activities.

BasicActivities

This level includes control sequencing and data flow between actions, but explicit forks and joins of control, as well as decisions and merges, are not supported. The basic and structured levels are orthogonal. Either can be used without the other or both can be used to support modeling that includes both flows and structured control constructs.

IntermediateActivities

The intermediate level supports modeling of activity diagrams that include concurrent control and data flow, and decisions. It supports modeling similar to traditional Petri nets with queuing. It requires the basic level.

The intermediate and structured levels are orthogonal. Either can be used without the other or both can be used to support modeling that includes both concurrency and structured control constructs.

CompleteActivities

The complete level adds constructs that enhance the lower level models, such as edge weights and streaming.

StructuredActivities

The structured level supports modeling of traditional structured programming constructs, such as sequences, loops, and conditionals, as an addition to fundamental activity nodes. It requires the fundamental level. It is compatible with the intermediate and complete levels.

CompleteStructuredActivities

This level adds support for data flow output pins of sequences, conditionals, and loops. It depends on the basic layer for flows.

ExtraStructuredActivities

The extra structure level supports exception handling as found in traditional programming languages and invocation of behaviors on sets of values. It requires the structured level.

12.2 Abstract Syntax

Figure 12.1 shows the dependencies of the activity packages.

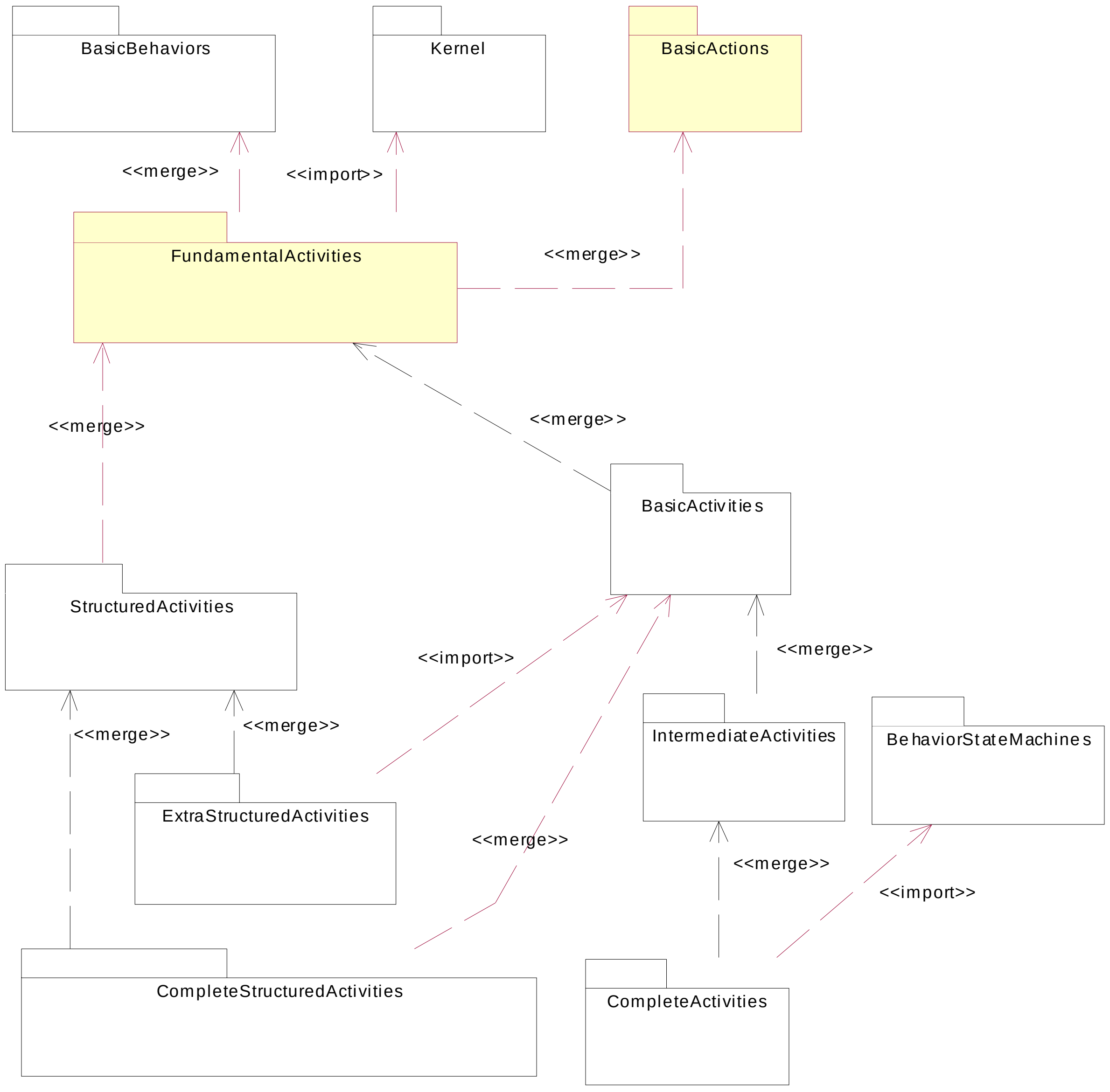


Figure 12.1 - Dependencies of the Activity packages

Package FundamentalActivities

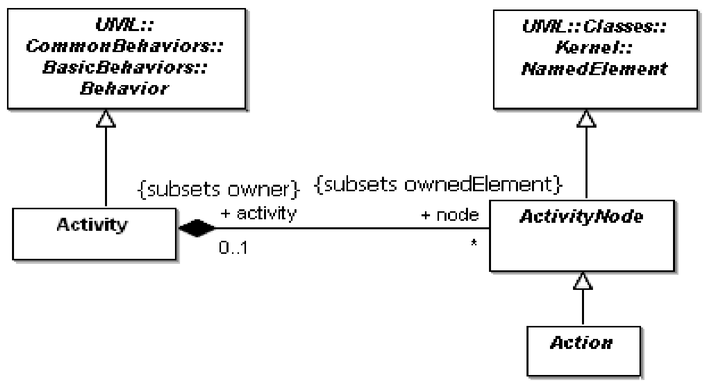


Figure 12.2 - Fundamental nodes

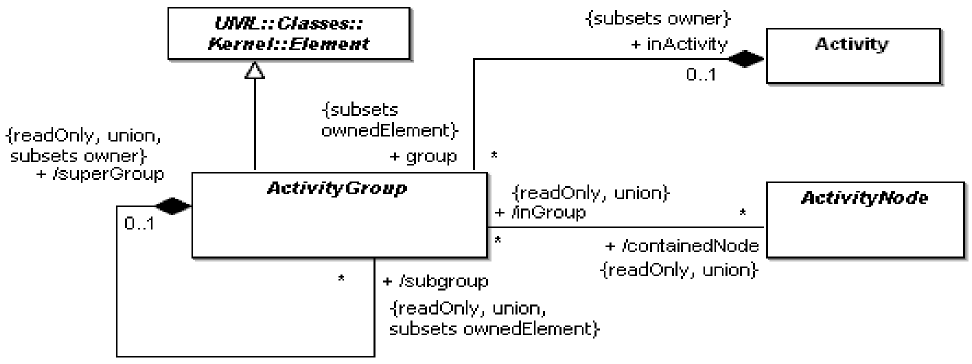


Figure 12.3 - Fundamental groups

Package BasicActivities

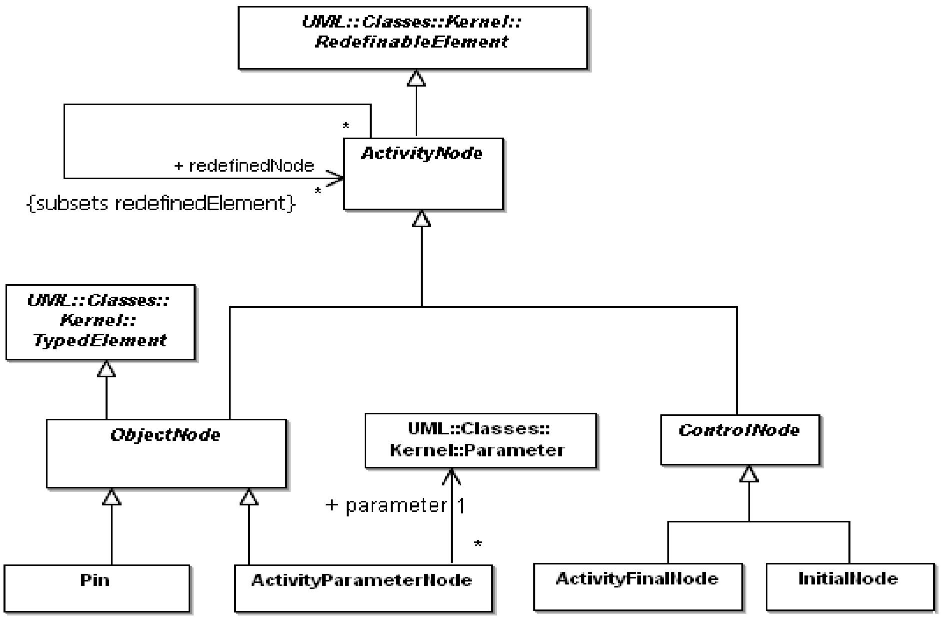


Figure 12.4 - Nodes (BasicActivities)

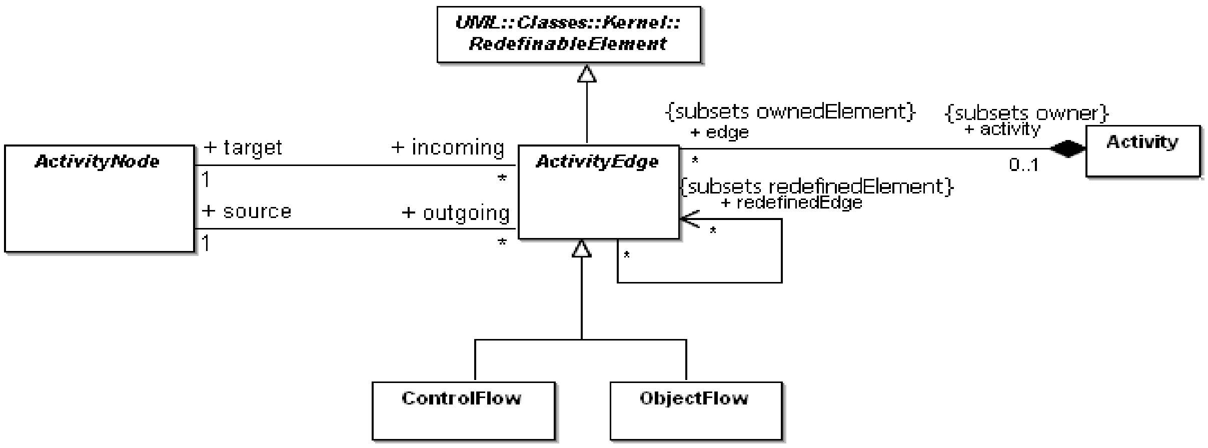


Figure 12.5 - Flows

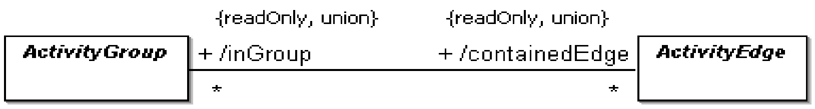


Figure 12.6 - Groups

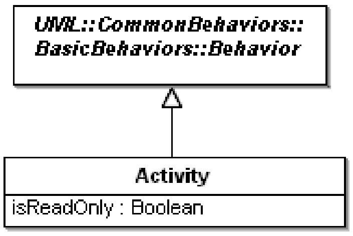


Figure 12.7 - Elements

Package IntermediateActivities

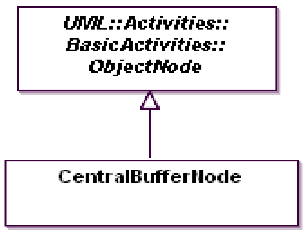


Figure 12.8 - Object nodes (IntermediateActivities)

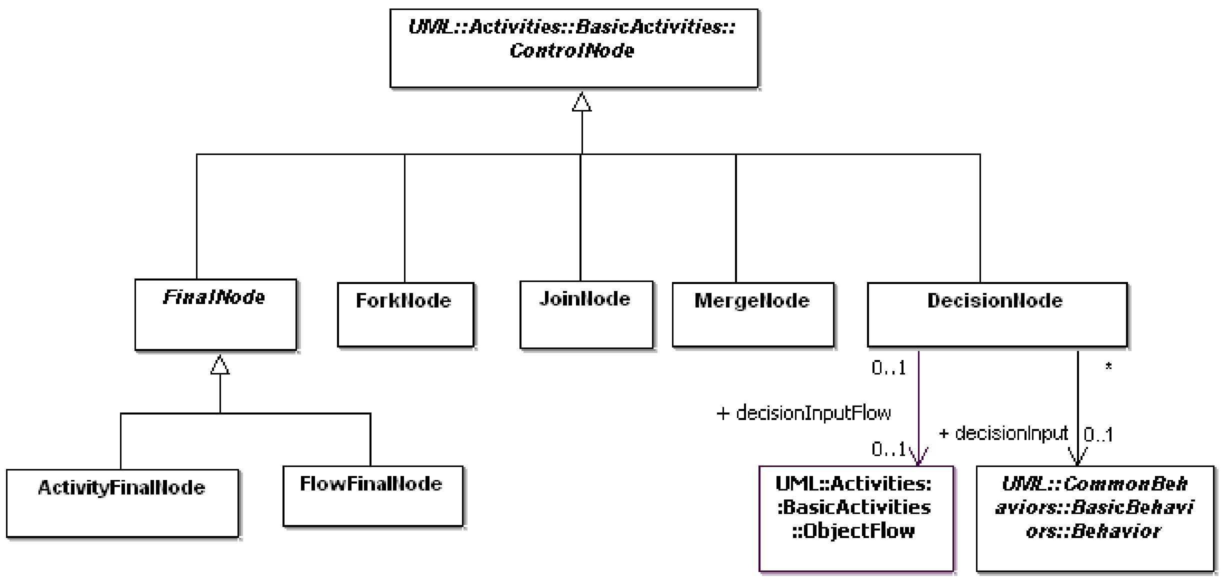

UML::Activities::BasicActivities::ControlNode
FinalNode
ForkNode
JoinNode
MergeNode
DecisionNode
ActivityFinalNode
FlowFinalNode
0..1
*
+ decisionInputFlow
0..1
+ decisionInput
0..1
UML::Activities::BasicActivities::ObjectFlow
UML::CommonBehaviors::BasicBehaviors::Behavior

Figure 12.9 - Control nodes (IntermediateActivities)

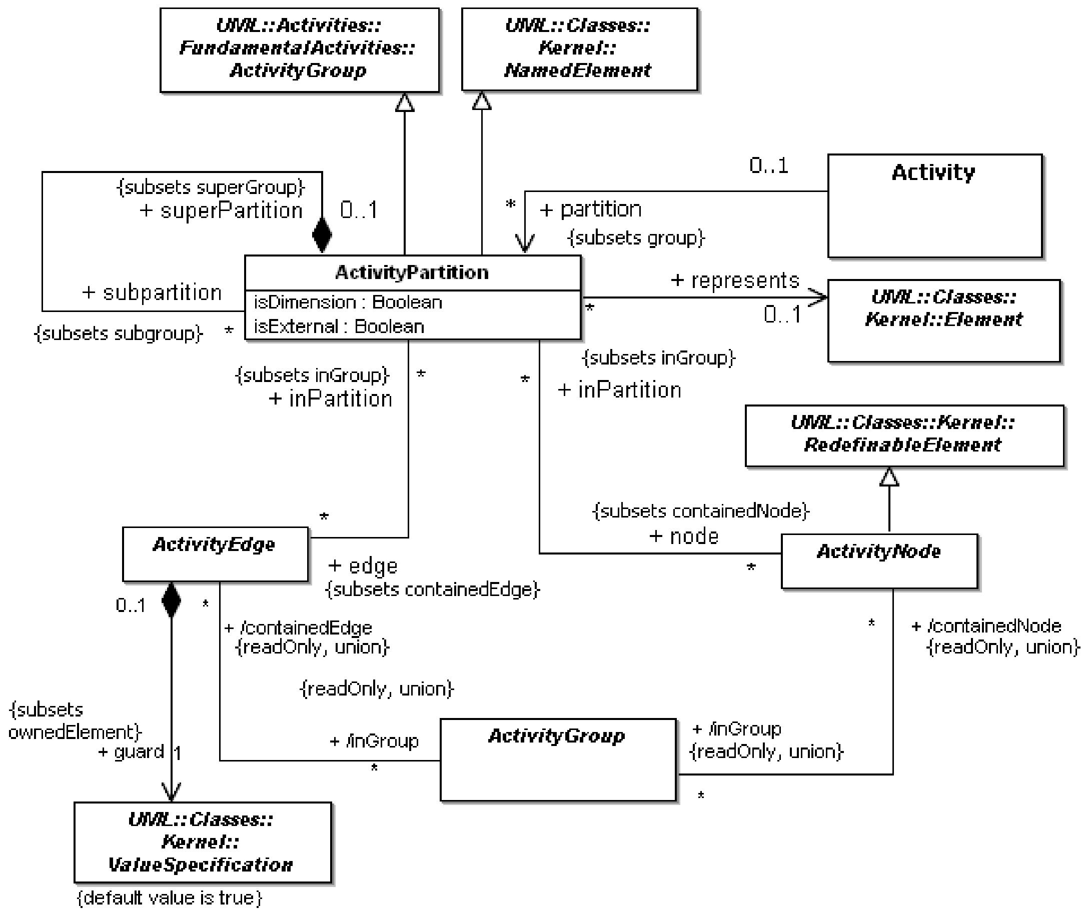


Figure 12.10 - Partitions

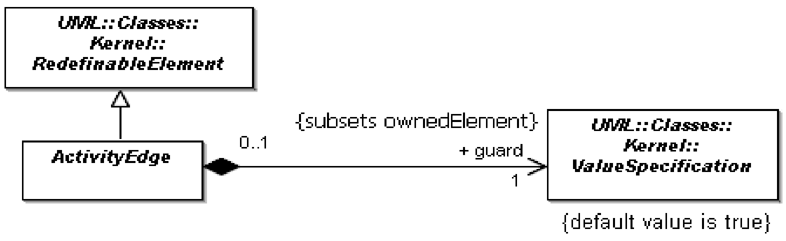


Figure 12.11 - Flows (IntermediateActivities)

Package CompleteActivities

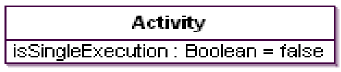


Figure 12.12 - Elements (CompleteActivities)

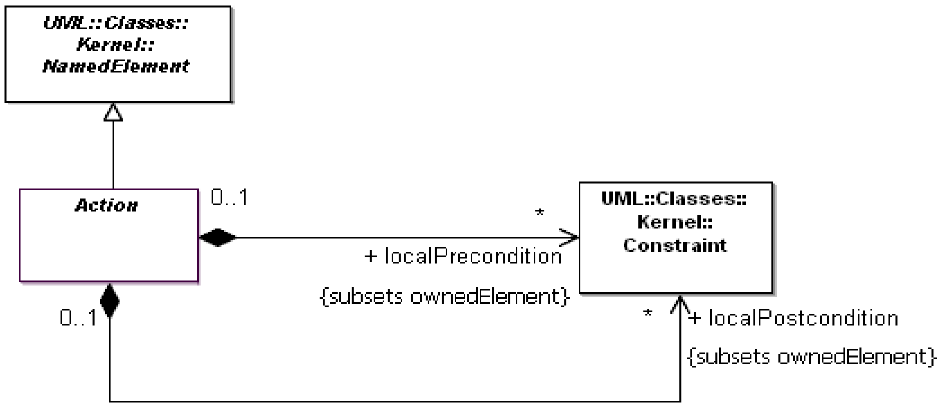


Figure 12.13 - Constraints (CompleteActivities)

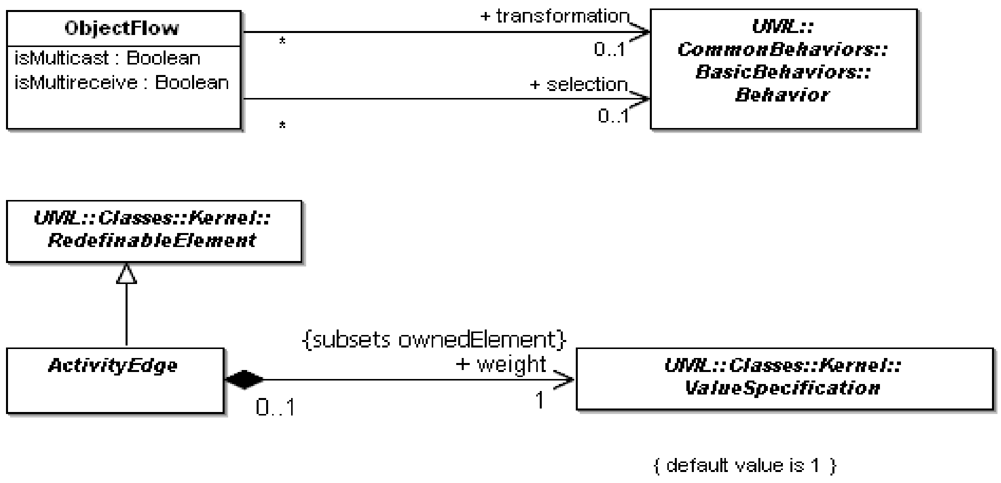


Figure 12.14 - Flows (CompleteActivities)

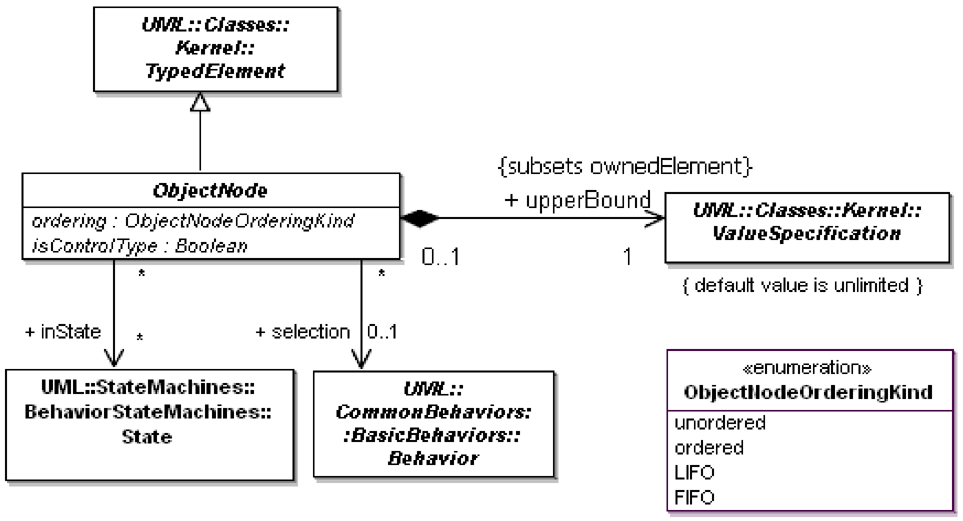

UML::Classes::Kernel::TypedElement
ObjectNode
ordering : ObjectNodeOrderingKind
isControlType : Boolean
{subsets ownedElement}
+ upperBound
0..1
1
UML::Classes::Kernel::ValueSpecification
{ default value is unlimited }
*
+ inState
*
*
+ selection
0..1
UML::StateMachines::BehaviorStateMachines::State
UML::CommonBehaviors::BasicBehaviors::Behavior
«enumeration»
ObjectNodeOrderingKind
unordered
ordered
LIFO
FIFO

Figure 12.15 - Object nodes (CompleteActivities)

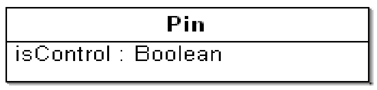


Figure 12.16 - Control pins

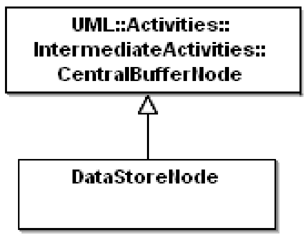


Figure 12.17 - Data stores

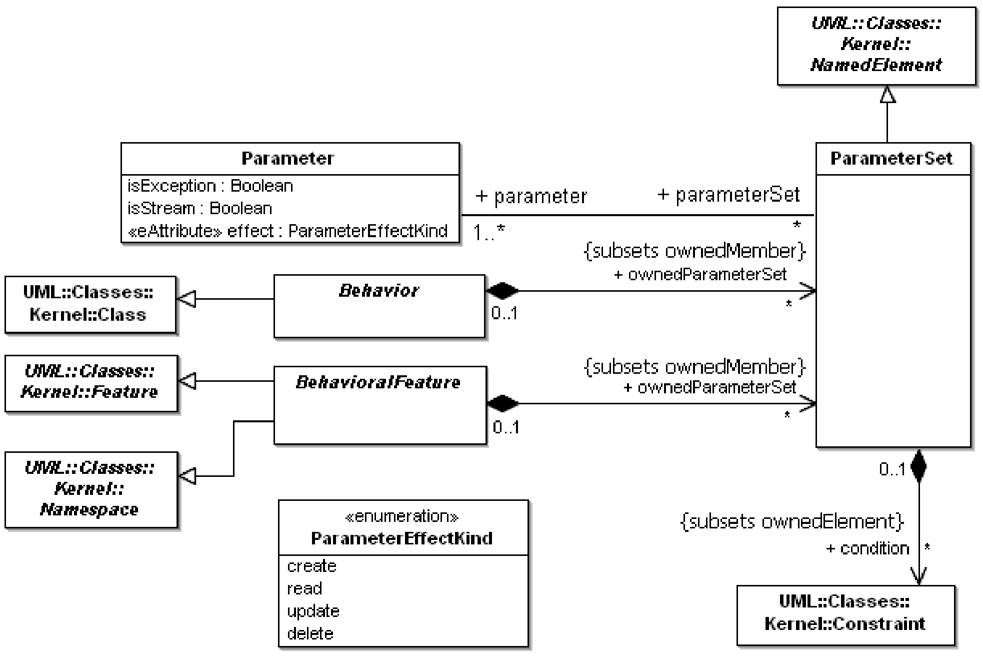


Figure 12.18 - Parameter sets

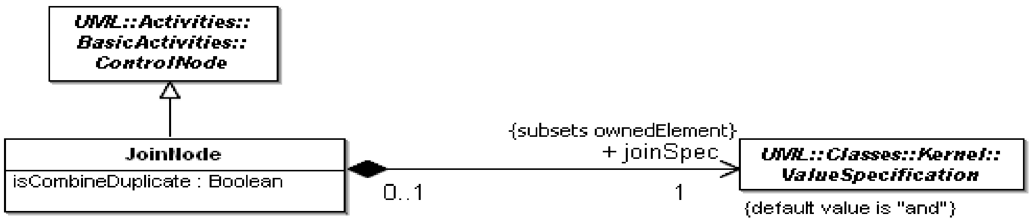


Figure 12.19 - Control nodes (CompleteActivities)

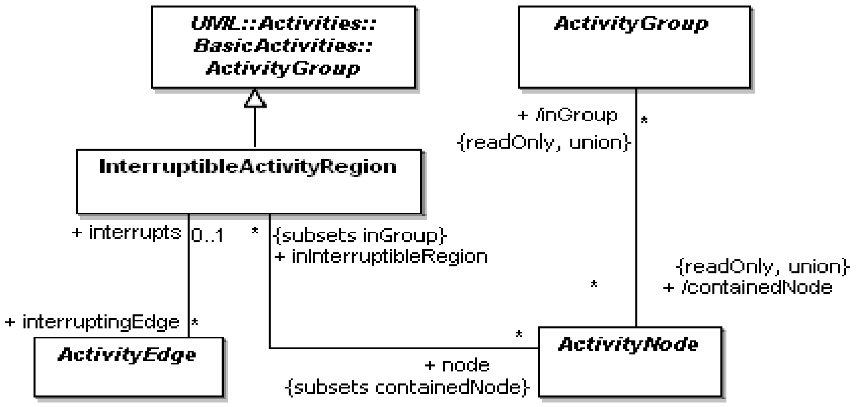


Figure 12.20 - Interruptible regions

Package StructuredActivities

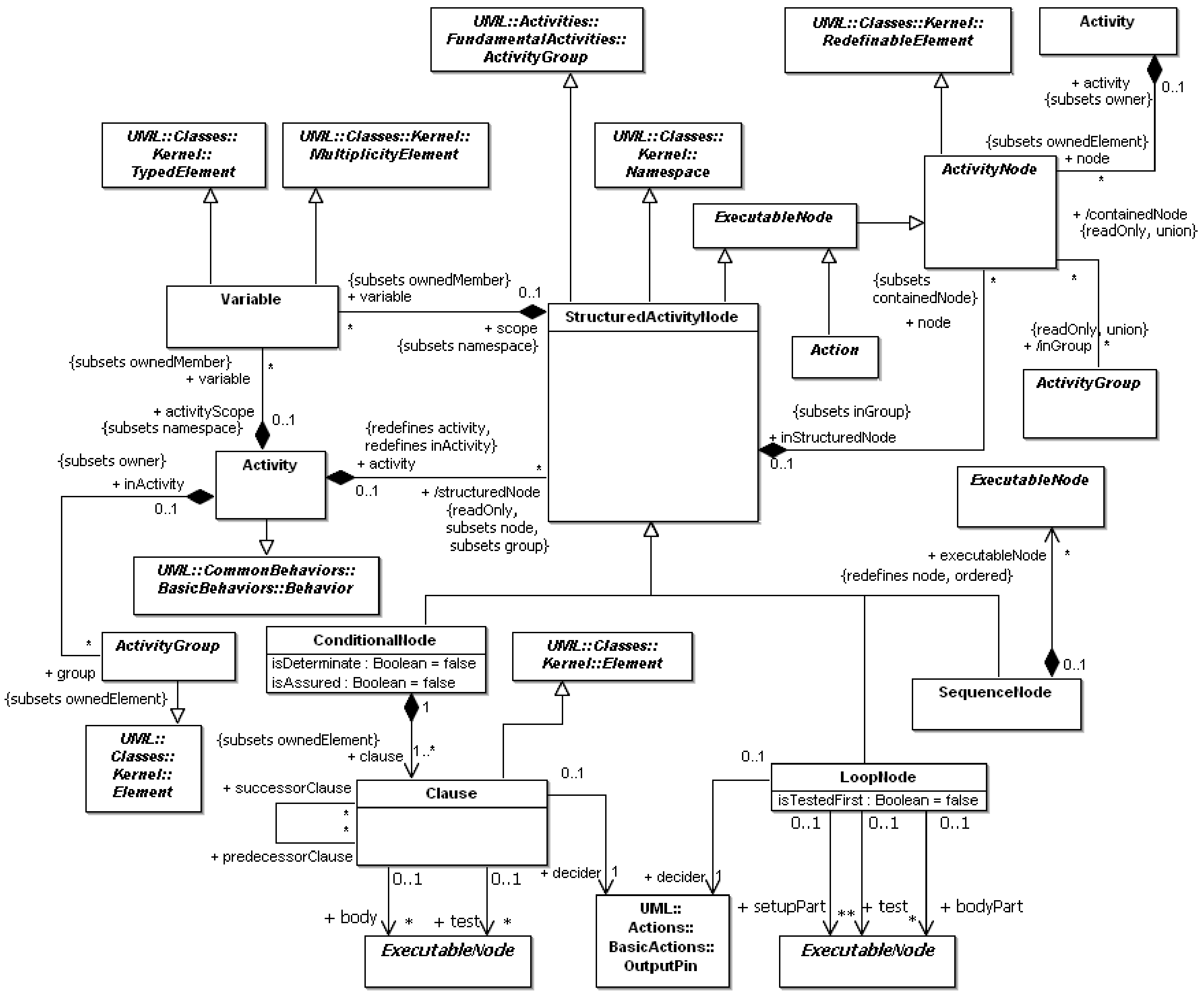
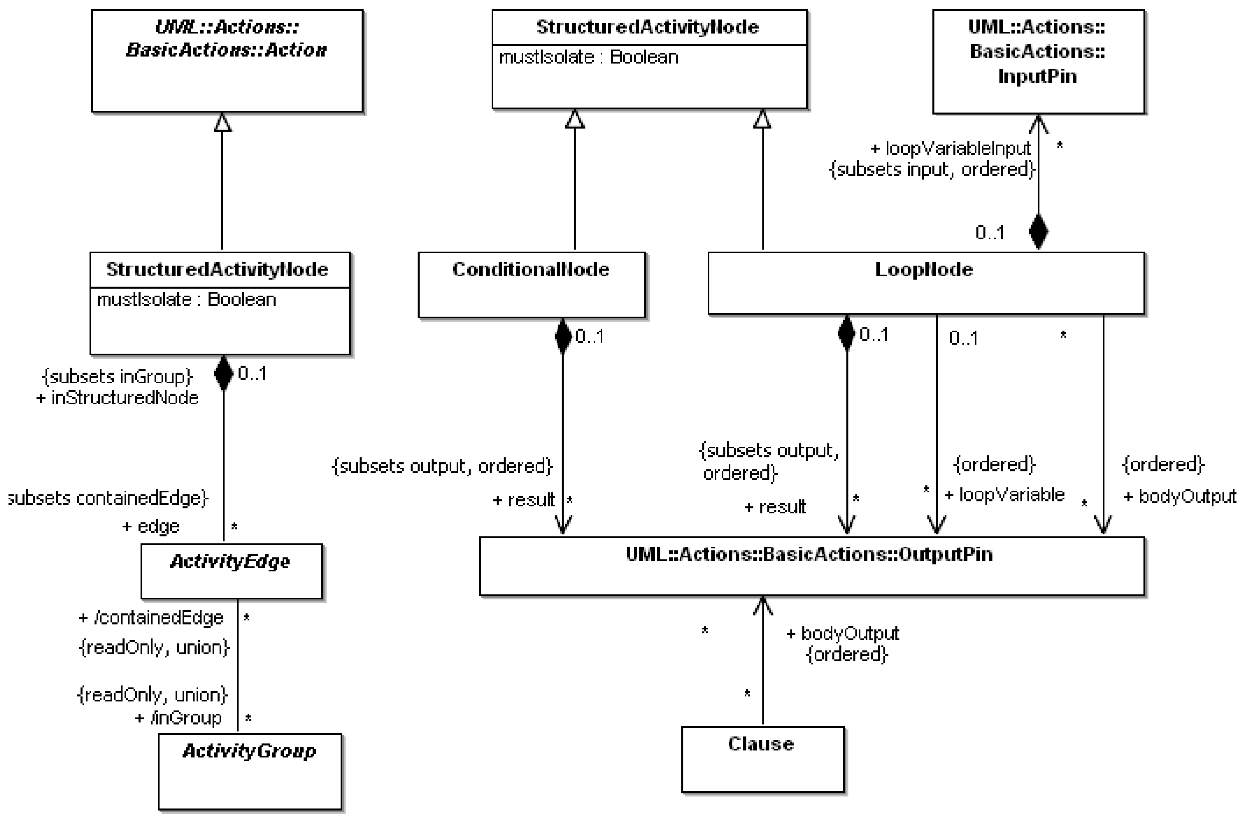

UML::Activities::FundamentalActivities::ActivityGroup
UML::Classes::Kernel::RedefinableElement
Activity
+ activity {subsets owner} 0..1
UML::Classes::Kernel::TypedElement
UML::Classes::Kernel::MultiplicityElement
UML::Classes::Kernel::Namespace
{subsets ownedElement} + node *
ActivityNode
ExecutableNode
+ /containedNode {readOnly, union}
{subsets ownedMember} + variable 0..1
Variable
StructuredActivityNode
* + scope {subsets namespace}
{subsets containedNode} + node *
Action
*
{readOnly, union} + /inGroup *
ActivityGroup
{subsets ownedMember} + variable *
+ activityScope {subsets namespace} 0..1
{subsets inGroup} + inStructuredNode 0..1
{redefines activity, redefines inActivity} + activity *
{subsets owner} + inActivity 0..1
Activity
0..1 + /structuredNode {readOnly, subsets node, subsets group}
ExecutableNode
UML::CommonBehaviors::BasicBehaviors::Behavior
+ executableNode * {redefines node, ordered}
* + group {subsets ownedElement}
ActivityGroup
ConditionalNode
isDeterminate : Boolean = false
isAssured : Boolean = false
UML::Classes::Kernel::Element
0..1
SequenceNode
UML::Classes::Kernel::Element
{subsets ownedElement} + clause 1 1..*
+ successorClause * *
+ predecessorClause
Clause
0..1
0..1
LoopNode
isTestedFirst : Boolean = false
0..1 0..1 0..1
0..1 0..1
+ body * + test *
+ decider 1
+ decider 1
+ setupPart * * + test * + bodyPart *
ExecutableNode
UML::Actions::BasicActions::OutputPin
ExecutableNode

Figure 12.21 - Structured nodes

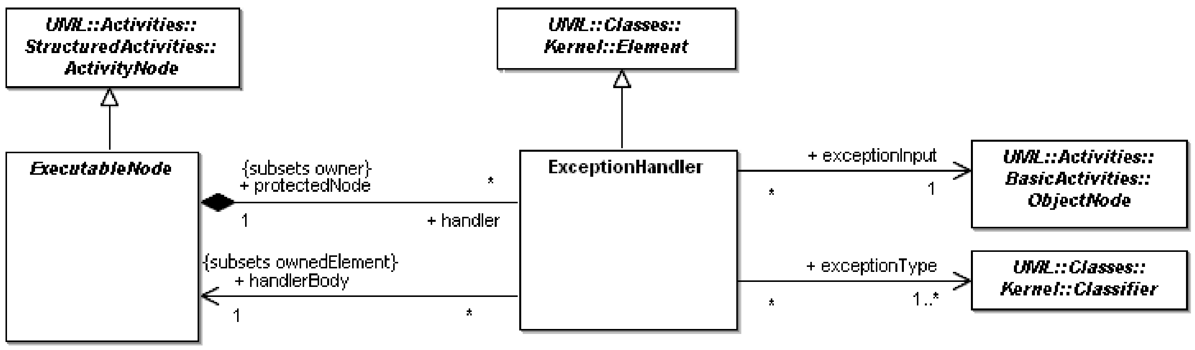
Package CompleteStructuredActivities



UML::Actions::BasicActions::Action
StructuredActivityNode
mustIsolate : Boolean
UML::Actions::BasicActions::InputPin
+ loopVariableInput {subsets input, ordered}
*
0..1
StructuredActivityNode
mustIsolate : Boolean
ConditionalNode
LoopNode
{subsets inGroup} + inStructuredNode
0..1
{subsets containedEdge} + edge
*
{subsets output, ordered} + result
*
{subsets output, ordered} + result
{ordered} + loopVariable
{ordered} + bodyOutput
ActivityEdge
UML::Actions::BasicActions::OutputPin
+ /containedEdge {readOnly, union}
{readOnly, union} + /inGroup
+ bodyOutput {ordered}
ActivityGroup
Clause

Figure 12.22 - Structured nodes (CompleteStructuredActivities)

Package ExtraStructuredActivities



UML::Activities::StructuredActivities::ActivityNode
UML::Classes::Kernel::Element
ExecutableNode
{subsets owner} + protectedNode
1
+ handler
*
ExceptionHandler
+ exceptionInput
1
UML::Activities::BasicActivities::ObjectNode
{subsets ownedElement} + handlerBody
1
+ exceptionType
1..*
UML::Classes::Kernel::Classifier

Figure 12.23 - Exceptions

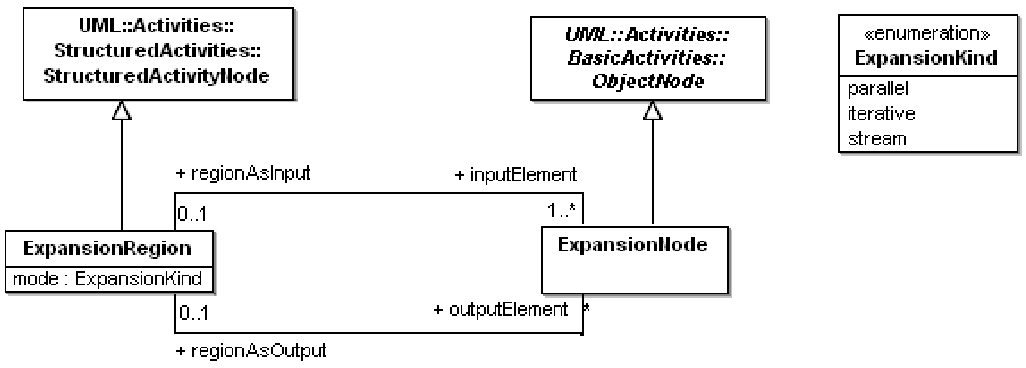


Figure 12.24 - Expansion regions

12.3 Class Descriptions

12.3.1 AcceptEventAction (as specialized)

See “AcceptEventAction (from CompleteActions)” on page 234.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

If an AcceptEventAction has no incoming edges, then the action starts when the containing activity or structured node does, whichever most immediately contains the action. In addition, an AcceptEventAction with no incoming edges remains enabled after it accepts an event. It does not terminate after accepting an event and outputting a value, but continues to wait for other events. This semantic is an exception to the normal execution rules in Activities. An AcceptEventAction with no incoming edges and contained by a structured node is terminated when its container is terminated.

Notation

See “AcceptEventAction (from CompleteActions)” on page 234.

Examples

Figure 12.25 is an example of the acceptance of 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 no input arrow is shown.

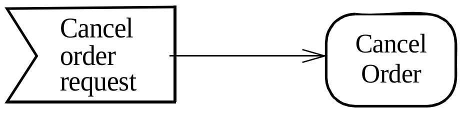


Figure 12.25 - Accept signal - top level in scope

In Figure 12.26, a request payment signal is sent after an order is processed. 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.

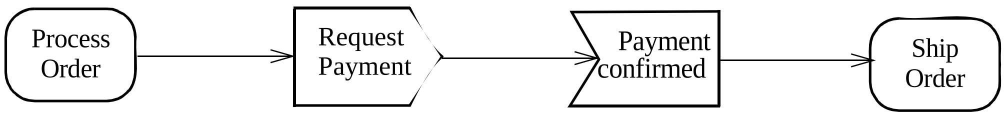


Figure 12.26 - Accept signal - explicit enable

In Figure 12.27, the end-of-month accept time event action generates an output at the end of the month. Since there are no incoming edges to the time event action, it is enabled as long as its containing activity or structured node is. It will generate an output at the end of every month.

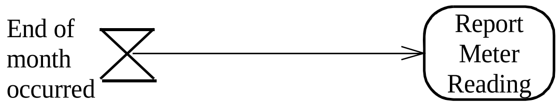


Figure 12.27 - Repetitive time event

Rationale

See “AcceptEventAction (from CompleteActions)” on page 234.

Changes from previous UML

See “AcceptEventAction (from CompleteActions)” on page 234.

12.3.2 Action (from CompleteActivities, FundamentalActivities, StructuredActivities)

Generalizations

- “ActivityNode (from BasicActivities, CompleteActivities, FundamentalActivities, IntermediateActivities, CompleteStructuredActivities)” on page 333.
- “ExecutableNode (from ExtraStructuredActivities, StructuredActivities)” on page 366.
- “Action (from BasicActions)” on page 236 *(merge increment)*.

Description

An action represents a single step within an activity, that is, one that is not further decomposed within the activity. An activity represents a behavior that is composed of individual elements that are actions. Note, however, that a call behavior action may reference an activity definition, in which case the execution of the call action involves the execution of the referenced activity and its actions (similarly for all the invocation actions). An action is therefore simple from the point of view of the activity containing it, but may be complex in its effect and not be atomic. As a piece of structure within an activity model, it is a single discrete element; as a specification of behavior to be performed, it may invoke referenced behavior that is arbitrarily complex. As a consequence, an activity defines a behavior that can be reused in many places, whereas an instance of an action is only used once at a particular point in an activity.

An action may have sets of incoming and outgoing activity edges that specify control flow and data flow from and to other nodes. An action will not begin execution until all of its input conditions are satisfied. The completion of the execution of an action may enable the execution of a set of successor nodes and actions that take their inputs from the outputs of the action.

Package CompleteActivities

In CompleteActivities, action is extended to have pre- and postconditions.

Attributes

No additional attributes

Associations

Package CompleteActivities

- localPrecondition : Constraint [0..*]
  Constraint that must be satisfied when execution is started. {Subsets *Element::ownedElement*}
- localPostcondition : Constraint [0..*]
  Constraint that must be satisfied when execution is completed. {Subsets *Element::ownedElement*}

Constraints

No additional constraints

Operations

[1] activity operates on Action. It returns the activity containing the action.

```
activity() : Activity;
activity = if self.Activity->size() > 0 then self.Activity else self.group.activity() endif
```

Semantics

The sequencing of actions are controlled by control edges and object flow edges within activities, which carry control and object tokens respectively (see Activity). Alternatively, the sequencing of actions is controlled by structured nodes, or by a combination of structured nodes and edges. Except where noted, an action can only begin execution when all incoming control edges have tokens, and all input pins have object tokens. The action begins execution by taking tokens from its incoming control edges and input pins. When the execution of an action is complete, it offers tokens in its outgoing control edges and output pins, where they are accessible to other actions.

The steps of executing an action with control and data flow are as follows:

[1] An action execution is created when all its object flow and control flow prerequisites have been satisfied (implicit join). Exceptions to this are listed below. The object flow prerequisite is satisfied when all of the input pins are offered all necessary tokens and accept them all at once, precluding them from being consumed by any other actions. This ensures that multiple action executions competing for tokens do not accept only some of the tokens they need to begin, causing deadlock as each execution waits for tokens that are already taken by others.

[2] An action execution consumes the input control and object tokens and removes them from the sources of control edges and from input pins. The action execution is now enabled and may begin execution. If multiple control tokens are available on a single edge, they are all consumed.

[3] An action continues executing until it has completed. Most actions operate only on their inputs. Some give access to a wider context, such as variables in the containing structured activity node, or the self object, which is the object owning the activity containing the executing action. The detailed semantic of execution an action and definition of completion depends on the particular subclass of action.

[4] When completed, an action execution offers object tokens on all its output pins and control tokens on all its outgoing control edges (implicit fork), and it terminates. Exceptions to this are listed below. The output tokens are now available to satisfy the control or object flow prerequisites for other action executions.

[5] After an action execution has terminated, its resources may be reclaimed by an implementation, but the details of resource management are not part of this specification and are properly part of an implementation profile.

See ValuePin and Parameter for exceptions to rule for starting action execution.

If a behavior is not reentrant, then no more than one execution of it will exist at any given time. An invocation of a non-reentrant behavior does not start the behavior when the behavior is already executing. In this case, control tokens are discarded, and data tokens collect at the input pins of the invocation action, if their upper bound is greater than one, or upstream otherwise. An invocation of a reentrant behavior will start a new execution of the behavior with newly arrived tokens, even if the behavior is already executing from tokens arriving at the invocation earlier.

Package ExtraStructuredActivities

If an exception occurs during the execution of an action, the execution of the action is abandoned and no regular output is generated by this action. If the action has an exception handler, it receives the exception object as a token. If the action has no exception handler, the exception propagates to the enclosing node and so on until it is caught by one of them. If an exception propagates out of a nested node (action, structured activity node, or activity), all tokens in the nested node are terminated. The data describing an exception is represented as an object of any class.

Package CompleteActivities

Streaming allows an action execution to take inputs and provide outputs while it is executing. During one execution, the action may consume multiple tokens on each streaming input and produce multiple tokens on each streaming output. See Parameter.

Local pre- and post-conditions are constraints that should hold when the execution starts and completes, respectively. They hold only at the point in the flow that they are specified, not globally for other invocations of the behavior at other places in the flow or on other diagrams. Compare to pre and postconditions on Behavior (in Activities). See semantic variations below for their effect on flow.

Semantic Variation Points

Package CompleteActivities

How local pre- and postconditions are enforced is determined by the implementation. For example, violations may be detected at compile time or runtime. The effect may be an error that stops the execution or just a warning, and so on. Since local pre- and post-conditions are modeler-defined constraints, violations do not mean that the semantics of the invocation is undefined as far as UML goes. They only mean the model or execution trace 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).

See variations in ActivityEdge and ObjectNode.

Notation

Use of action and activity notation is optional. A textual notation may be used instead.

Actions are notated as round-cornered rectangles. The name of the action or other description of it may appear in the symbol. See children of action for refinements.

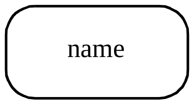


Figure 12.28 - Action

Package CompleteActivities

Local pre- and post-conditions are shown as notes attached to the invocation with the keywords «localPrecondition» and «localPostcondition», respectively.

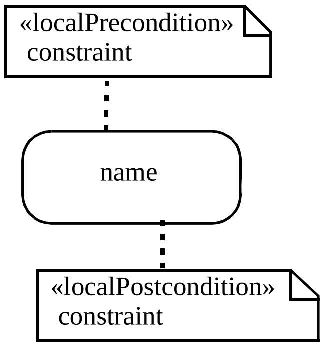


Figure 12.29 - Local pre- and post-conditions

Examples

Examples of actions are illustrated below. These perform behaviors called Send Payment and Accept Payment.

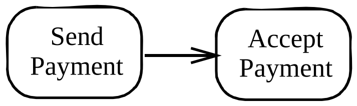


Figure 12.30 - Examples of actions

Below is an example of an action expressed in an application-dependent action language:

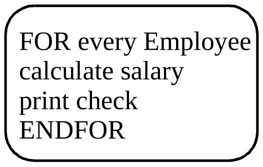


Figure 12.31 - Example of action with tool-dependent action language

Package CompleteActivities

The example 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 would be all that is required. (Global conditions are described in Activity specification, in the next sub clause.) For example, a global pre-condition for a Dispense Drink activity 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.

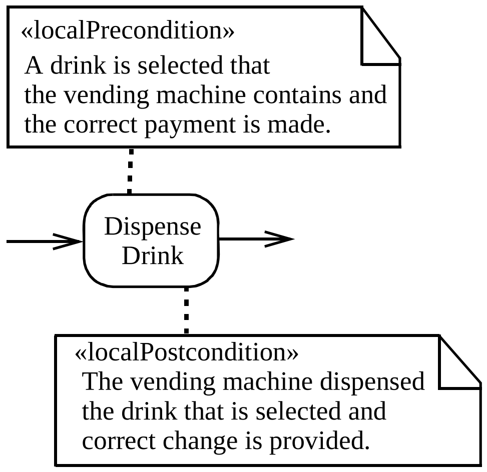

«localPrecondition»
A drink is selected that the vending machine contains and the correct payment is made.
Dispense Drink
«localPostcondition»
The vending machine dispensed the drink that is selected and correct change is provided.

Figure 12.32 - Example of an action with local pre- and post-conditions

Changes from previous UML

Explicitly modeled actions as part of activities are new in UML 2.0, and replace ActionState, CallState, and SubactivityState in UML 1.5. They represent a merger of activity graphs from UML 1.5 and actions from UML 1.5.

Local pre- and post-conditions are new to UML 2.0.

12.3.3 ActionInputPin (as specialized)

See “ActionInputPin (from StructuredActions)” on page 237.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See “ActionInputPin (from StructuredActions)” on page 237.

Notation

An action input pin with a ReadVariableAction as a fromAction is notated as an input pin with the variable name written beside it. An action input pin with a ReadSelfObject as a fromAction is notated as an input pin with the word “self” written beside it. An action input pin with a ValueSpecification as a fromAction is notated as an input pin with the value specification written beside it.

Examples

See “ActionInputPin (from StructuredActions)” on page 237.

12.3.4 Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)

An activity is the specification of parameterized behavior as the coordinated sequencing of subordinate units whose individual elements are actions. There are actions that invoke activities (directly by “CallBehaviorAction (from BasicActions)” on page 243 or indirectly as methods by “CallOperationAction (from BasicActions)” on page 245).

Generalizations

- “Behavior (from BasicBehaviors)” on page 430

Description

An activity specifies the coordination of executions of subordinate behaviors, using a control and data flow model. The 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 external to the flow. The flow of execution is modeled as activity nodes connected by activity edges. A node can be the execution of a subordinate behavior, such as an arithmetic computation, a call to an operation, or manipulation of object contents. Activity nodes also include flow-of-control constructs, such as synchronization, decision, and concurrency control. Activities may form invocation hierarchies invoking other activities, ultimately resolving to individual actions. In an object-oriented model, activities are usually invoked indirectly as methods bound to operations that are directly invoked.

Activities may describe procedural computation. In this context, they are the methods corresponding to operations on classes. 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.

Activities may contain actions of various kinds:

- Occurrences of primitive functions, such as arithmetic functions.
- Invocations of behavior, such as activities.
- Communication actions, such as sending of signals.
- Manipulations of objects, such as reading or writing attributes or associations.

Actions have no further decomposition in the activity containing them. However, the execution of a single action may induce the execution of many other actions. For example, a call action invokes an operation that is implemented by an activity containing actions that execute before the call action completes.

Most of the constructs in the Activity clause deal with various mechanisms for sequencing the flow of control and data among the actions:

- Object flows for sequencing data produced by one node that is used by other nodes.
- Control flows for sequencing the execution of nodes.
- Control nodes to structure control and object flow. These include decisions and merges to model contingency. These also include initial and final nodes for starting and ending flows. In IntermediateActivities, they include forks and joins for creating and synchronizing concurrent subexecutions.
- Activity generalization to replace nodes and edges.
- Object nodes to represent objects and data as they flow in and out of invoked behaviors, or to represent collections of tokens waiting to move downstream.

Package StructuredActivities

- Composite nodes to represent structured flow-of-control constructs, such as loops and conditionals.

Package IntermediateActivities

- Partitions to organize lower-level activities according to various criteria, such as the real-world organization responsible for their performance.

Package CompleteActivities

- Interruptible regions and exceptions to represent deviations from the normal, mainline flow of control.

Attributes

Package BasicActivities

- isReadOnly : Boolean = false
  If *true*, this activity must not make any changes to variables outside the activity or to objects. (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 action, then the model is ill formed.) The default is false (an activity may make non-local changes).

Package CompleteActivities

- isSingleExecution : Boolean = false
  If *true*, all invocations of the activity are handled by the same execution.

Associations

Package FundamentalActivities

- group : ActivityGroup [0..*]
  Top-level groups in the activity. {Subsets *Namespace::ownedElement*}
- node : ActivityNode [0..*]
  Nodes coordinated by the activity. {Subsets *Namespace::ownedElement*}

Package BasicActivities

- edge : ActivityEdge [0..*]
  Edges expressing flow between nodes of the activity. {Subsets *Namespace::ownedElement*}

Package IntermediateActivities

- partition : ActivityPartition [0..*]
  Top-level partitions in the activity. {Subsets *Activity::group*}

Package StructuredActivities

- /structuredNode : StructuredActivityNode [0..*]
  Top-level structured nodes in the activity. Subsets
- variable : Variable [0..*]
  Top-level variables in the activity. Subsets *Namespace::ownedMember.*

Constraints

[1] The nodes of the activity must include one ActivityParameterNode for each parameter.

[2] An activity cannot be autonomous and have a classifier or behavioral feature context at the same time.

[3] The groups of an activity have no supergroups.

Semantics

The semantics of activities is based on token flow. By *flow*, we mean that the execution of one node affects, and is affected by, the execution of other nodes, and such dependencies are represented by *edges* in the activity diagram. A *token* contains an object, datum, or locus of control, and is present in the activity diagram at a particular node. Each token is distinct from any other, even if it contains the same value as another. A node may begin execution when specified conditions on its input tokens are satisfied; the conditions depend on the kind of node. When a node begins execution, tokens are accepted from some or all of its input edges 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 output edges. See later in this sub clause for more about how tokens are managed.

All restrictions on the relative execution order of two or more actions are explicitly constrained by flow relationships. If two actions are not directly or indirectly ordered by flow relationships, they may execute concurrently. This does not require parallel execution; a specific execution engine may choose to perform the executions sequentially or in parallel, as long as any explicit ordering constraints are satisfied. In most cases, there are some flow relationships that constrain execution order. Concurrency is supported in IntermediateActivities, but not in BasicActivities.

Activities can be parameterized, which is a capability inherited from Behavior (see 12.3.9, "ActivityParameterNode (from BasicActivities)," on page 336). Functionality inherited from Behavior also supports the use of activities on classifiers and as methods for behavioral features. The classifier, if any, is referred to as the *context* of the activity. At runtime, the activity has access to the attributes and operations of its context object and any objects linked to the context object, transitively. An activity that is also a method of a behavioral feature has access to the parameters of the behavioral feature. In workflow terminology, the scope of information an activity uses is called the process-relevant data. Implementations that have access to metadata can define parameters that accept entire activities or other parts of the user model.

An activity with a classifier context, but that is not a method of a behavioral feature, can be invoked after the classifier is instantiated. An activity that is a method of a behavioral feature is invoked when the behavioral feature is invoked. The Behavior metaclass also provides parameters, which must be compatible with the behavioral feature it is a method of, if any. Behavior also supports overriding of activities used as inherited methods. See the Behavior metaclass for more information.

Activities can also be invoked directly by other activities rather than through the call of a behavioral feature that has an activity as a method. This functional or monomorphic style of invocation is useful at the stage of development where focus is on the activities to be completed and goals to be achieved. Classifiers responsible for each activity can be assigned at a later stage by declaring behavioral features on classifiers and assigning activities as methods for these features. For example, in business reengineering, an activity flow can be optimized independently of which departments or positions are later assigned to handle each step. This is why activities are autonomous when they are not assigned to a classifier.

Regardless of whether an activity is invoked through a behavioral feature or directly, inputs to the invoked activity are supplied by an invocation action in the calling activity, which gets its inputs from incoming edges. Likewise an activity invoked from another activity produces outputs that are delivered to an invocation action, which passes them onto its outgoing edges. See "Parameter (from CompleteActivities)" on page 396 for more about how activities start and stop execution.

An activity execution represents an execution of the activity. An activity execution, as a reflective object, can support operations for managing execution, such as starting, stopping, aborting, and so on; attributes, such as how long the process has been executing or how much it costs; and links to objects, such as the performer of the execution, who to report completion to, or resources being used, and states of execution such as started, suspended, and so on. Used this way activity is the modeling basis for the WfProcess interface in the OMG Workflow Management Facility, www.omg.org/cgi-bin/doc?formal/00-05-02. It is expected that profiles will include class libraries with standard classes that are used as root classes for activities in the user model. Vendors may define their own libraries, or support user-defined features on activity classes.

Nodes and edges have token flow rules. Nodes control when tokens enter or leave them. Edges have rules about when a token may be taken from the source node and moved to the target node. A token traverses an edge when it satisfies the rules for target node, edge, and source node all at once. This means a source node can only offer tokens to the outgoing edges, rather than force them along the edge, because the tokens may be rejected by the edge or the target node on the other side. Multiple tokens offered to an edge at once is the same as if they were offered one at a time. Since multiple edges can leave the same node, the same token can be offered to multiple targets. However, a token can only be accepted at one target. This means flow semantics is highly distributed and subject to timing issues and race conditions, as is any distributed system. There is no specification of the order in which rules are applied on the various nodes and edges in an activity. It is the responsibility of the modeler to ensure that timing issues do not affect system goals, or that they are eliminated from the model. Execution profiles may tighten the rules to enforce various kinds of execution semantics. Start at ActivityEdge and ActivityNode to see the token management rules.

Tokens cannot “rest” at *control nodes*, such as decisions and merges, waiting to move downstream. Control nodes act as traffic switches managing tokens as they make their way between object nodes and actions, which are the nodes where tokens can rest for a period of time. Initial nodes are excepted from this rule.

A data token with no value in is called the *null* token. It can be passed along and used like any other token. For example, an action can output a null token and a downstream decision point can test for it and branch accordingly. Null tokens satisfy the type of all object nodes.

The semantics of activities is specified in terms of these token rules, but only for the purpose of describing the expected runtime behavior. Token semantics is not intended to dictate the way activities are implemented, despite the use of the term “execution.” They only define the sequence and conditions for behaviors to start and stop. Token rules may be optimized in particular cases as long as the effect is the same.

Package IntermediateActivities

Activities can have multiple tokens flowing in them at any one time, if required. Special nodes called *object nodes* provide and accept objects and data as they flow in and out of invoked behaviors, and may act as buffers, collecting tokens as they wait to move downstream.

Package CompleteActivities

Each time an activity is invoked, the isSingleExecution attribute 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. Since there is only one plant, one execution of the activity handles all orders. This applies even if the behavior is a method, 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 nodes and edges. 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 activity final.

If a separate execution of the activity is used for each invocation, tokens from the various invocations do not interact. For example, an activity that is the behavior of a classifier, is invoked when the classifier is instantiated, 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. A new activity execution for each invocation reduces token interaction, but might not eliminate it. For example, an activity may have a loop creating tokens to be handled by the rest of the activity, or an unsynchronized flow that is aborted by an activity final. In these cases, modelers must consider the same token interaction issues as using a single activity execution for all invocations. Also see the effect of non-reentrant behaviors described at Except in CompleteActivities, each invocation of an activity is executed separately; tokens from different invocations do not interact.

Nodes and edges inherited from more general activities can be replaced. See RedefinableElement for more information on overriding inherited elements.

Package IntermediateActivities

If a single execution of the activity is used for all invocations, the modeler must consider additional interactions between tokens. Tokens may reach bottlenecks waiting for tokens ahead of them to move downstream, they may overtake each other due to the ordering algorithm used in object node buffers, or due to variations in the execution time of invoked behaviors, and most importantly, may abort each other with constructs such as activity final, exception outputs, and interruptible regions.

Package CompleteActivities

Complete activities add functionality that also increases interaction. For example, streaming outputs create tokens to be handled by the rest of the activity. In these cases, modelers must consider the same token interaction issues even when using a separate execution of activity execution for all invocations.

Interruptible activity regions are groups of nodes within which all execution can be terminated if an interruptible activity edge is traversed leaving the region.

See “ActivityNode (from BasicActivities, CompleteActivities, FundamentalActivities, IntermediateActivities, CompleteStructuredActivities)” and “ActivityEdge (from BasicActivities, CompleteActivities, CompleteStructuredActivities, IntermediateActivities)” for more information on the way activities function. An activity with no nodes and edges is well formed, but unspecified. It may be used as an alternative to a generic behavior in activity modeling. See “ActivityPartition (from IntermediateActivities)” for more information on grouping mechanisms in activities.

Semantic Variation Points

No specific variations in token management are defined, but extensions may add new types of tokens that have their own flow rules. For example, a BPEL extension might define a failure token that flows along edges that reject other tokens. Or an extension for systems engineering might define a new control token that terminates executing actions.

Notation

Use of action and activity notation is optional. A textual notation may be used instead.

The notation for an activity is a combination of the notations of the nodes and edges it contains, plus a border and name displayed in the upper left corner. Activity parameter nodes are displayed on the border. Actions and flows that are contained in the activity are also depicted.

Pre- and post-condition constraints, inherited from Behavior, are shown as with the keywords «precondition» and «postcondition», respectively. These apply globally to all uses of the activity. See Figure 12.33 and Behavior in Common Behavior; compare to local pre- and post-conditions on Action.

(CompleteActivities) The keyword «singleExecution» is used for activities that execute as a single shared execution; otherwise, each invocation executes in its space. See the notation sub clauses of the various kinds of nodes and edges for more information.

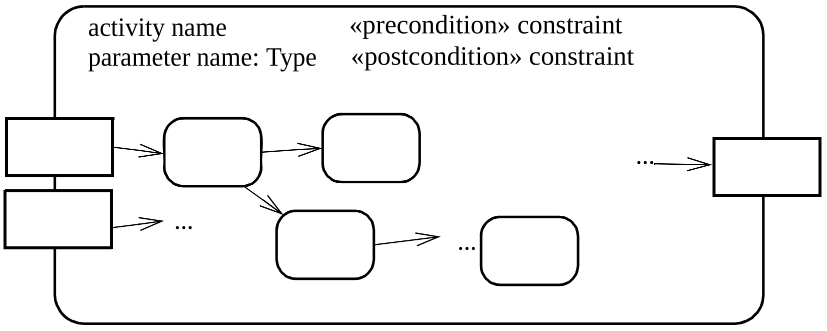


Figure 12.33 - Activity notation

The notation for classes can be used for diagramming the features of a reflective activity as shown below, with the keyword "activity" to indicate it is an activity class. Association and state machine notation can also be used as necessary.

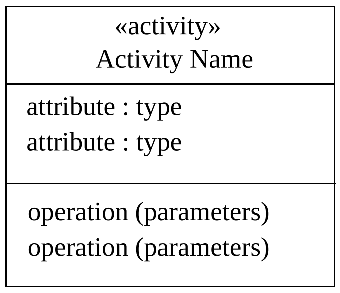


Figure 12.34 - Activity class notation

Presentation Options

The round-cornered border of Figure 12.33 may be replaced with the frame notation described in Annex A. Activity parameter nodes are displayed on the frame. The round-cornered border or frame may be omitted completely. See the presentation option for "ActivityParameterNode (from BasicActivities)" on page 336.

Examples

The definition of Process Order below uses the border notation to indicate that it is an activity. It has pre- and post conditions on the order (see Behavior). All invocations of it use the same execution.

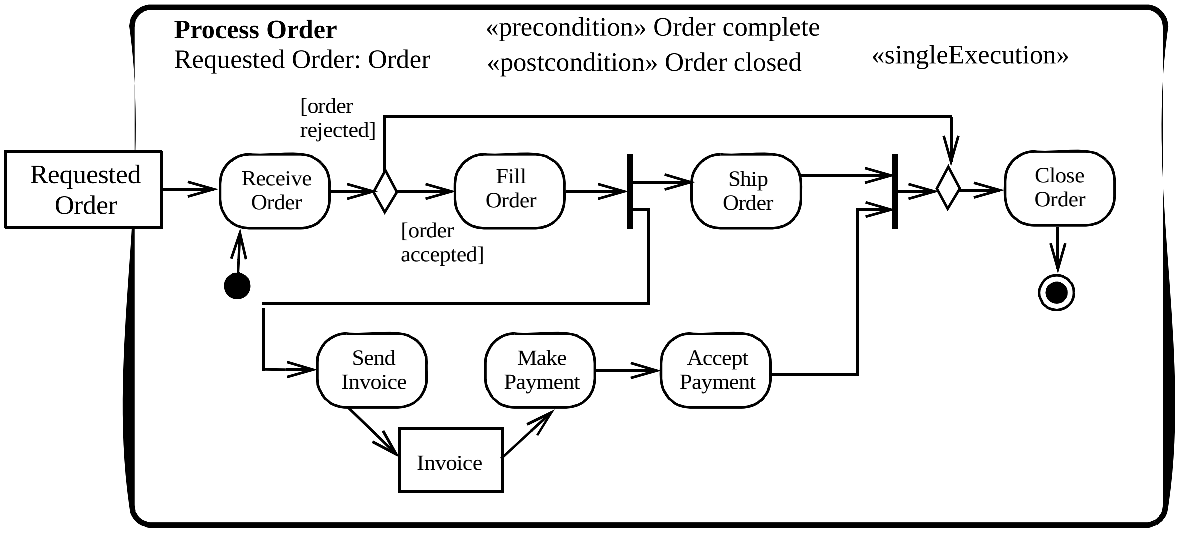


Figure 12.35 - Example of an activity with input parameter

The diagram below is based on a standard part selection workflow within an airline design process. Notice that the Standards Engineer insures 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 them 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, it is assigned to the Assign Standards Engineer activity. Lastly, Schedule Part Mod Workflow invocation produces entire activities and they are passed to subsequent invocations for scheduling and execution (i.e., Schedule Part Mod Workflow, Execute Part Mod Workflow, and Research Production Possibility). In other words, behaviors can produce tokens that are activities that can in turn be executed; in short, runtime activity generation and execution.

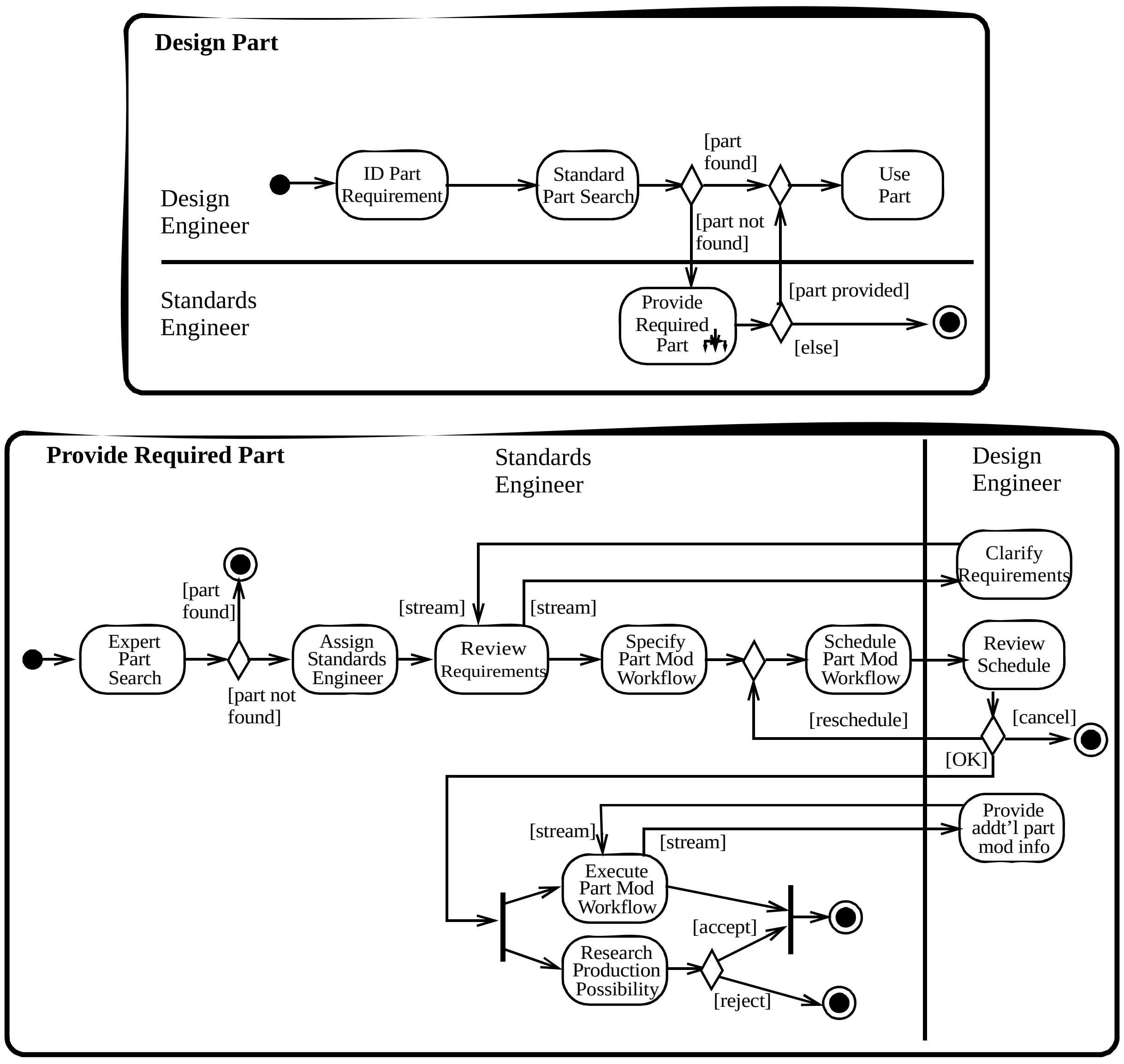


Figure 12.36 - Workflow example

Figure 12.37shows another example activity for a process to resolve a trouble ticket.

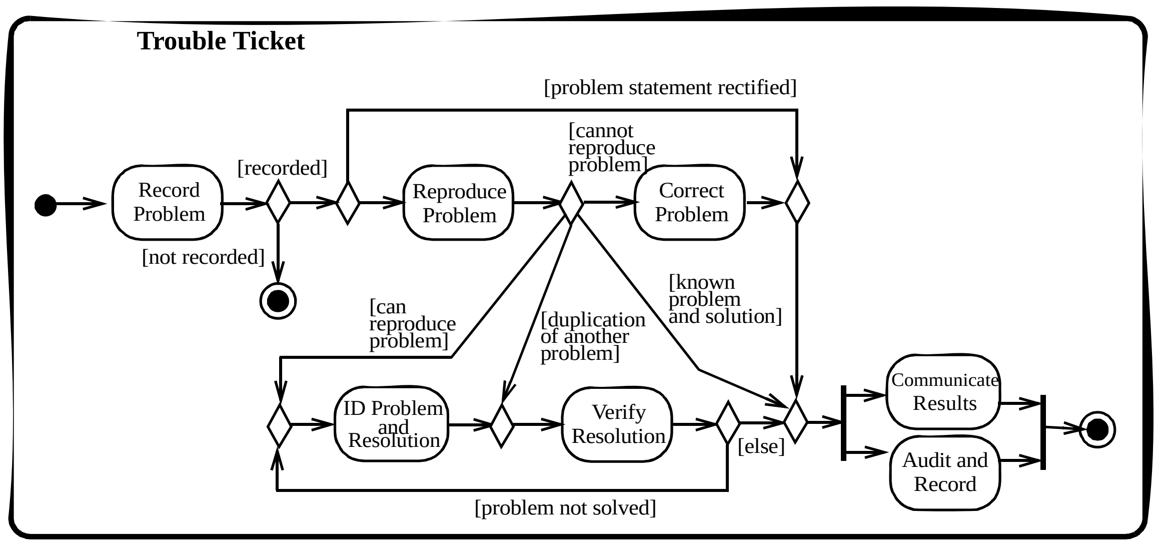


Figure 12.37 - Workflow example

Below is an example of using class notation to show the class features of an activity. Associations and state machines can also be shown.

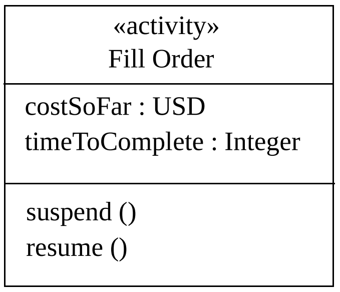


Figure 12.38 - Activity class with attributes and operations

Rationale

Activities are introduced to flow models that coordinate other behaviors, including other flow models. It supports class features to model control and monitoring of executing processes, and relating them to other objects (for example, in an organization model).

Changes from previous UML

Activity replaces ActivityGraph in UML 1.5. Activities are redesigned to use a Petri-like semantics instead of state machines. Among other benefits, this widens the number of flows that can be modeled, especially those that have parallel flows. Activity also replaces procedures in UML 1.5, as well as the other control and sequencing aspects, including composite and collection actions.

12.3.5 ActivityEdge (from BasicActivities, CompleteActivities, CompleteStructuredActivities, IntermediateActivities)

An activity edge is an abstract class for directed connections between two activity nodes.

Generalizations

- "RedefinableElement (from Kernel)" on page 130

Description

ActivityEdge is an abstract class for the connections along which tokens flow between activity nodes. It covers control and data flow edges. Activity edges can control token flow.

Package CompleteActivities

Complete activity edges can be contained in interruptible regions.

Attributes

No additional attributes

Associations

Package BasicActivities

- activity : Activity[0..1]
  Activity containing the edge. {Subsets *Element::owner*}
- /inGroup : ActivityGroup[0..*]
  Groups containing the edge. Multiplicity specialized to [0..1] for StructuredActivityGroup.
- redefinedEdge: ActivityEdge [0..*]
  Inherited edges replaced by this edge in a specialization of the activity. {Subsets *RedfinableElement::redefinedElement*}
- source ActivityNode [1..1]
  Node from which tokens are taken when they traverse the edge.
- target : ActivityNode [1..1]
  Node to which tokens are put when they traverse the edge.

Package IntermediateActivities

- inPartition : Partition [0..*]
  Partitions containing the edge. {Subsets *ActivityEdge::inGroup*}
- guard : ValueSpecification [1..1] = true
  Specification evaluated at runtime to determine if the edge can be traversed. {Subsets *Element::ownedElement*}

Package CompleteStructuredActivities

- inStructuredNode : StructuredActivityNode [0..1]
  Structured activity node containing the edge. {Subsets *ActivityEdge::inGroup*}

Package CompleteActivities

- interrupts : InterruptibleActivityRegion [0..1]
  Region that the edge can interrupt.
- weight : ValueSpecification [1..1] = 1
  The minimum number of tokens that must traverse the edge at the same time. {Subsets *Element::ownedElement*}

Constraints

[1] The source and target of an edge must be in the same activity as the edge.

[2] Activity edges may be owned only by activities or groups.

Package CompleteStructuredActivities

[1] Activity edges may be owned by at most one structured node.

Semantics

Activity edges are directed connections, that is, they have a source and a target, along which tokens may flow.

Other rules for when tokens may be passed along the edge depend on the kind of edge and characteristics of its source and target. See the children of ActivityEdge and ActivityNode. The rules may be optimized to a different algorithm as long as the effect is the same.

The guard must evaluate to true for every token that is offered to pass along the edge. Tokens in the intermediate level of activities can only pass along the edge individually at different times. See application of guards at DecisionNode.

Package CompleteActivities

Any number of tokens can pass along the edge, in groups at one time, or individually at different times. The weight attribute dictates the minimum number of tokens that must traverse the edge at the same time. It is a value specification evaluated every time a new token becomes available at the source. It must evaluate to a positive LiteralUnlimitedNatural, and may be a constant. When the minimum number of tokens are offered, all the tokens at the source are offered to the target all at once. The guard must evaluate to true for each token. 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 at the source are offered to the target. This can be combined with a join to take all of the tokens at the source when certain conditions hold (see examples in Figure 12.45). A weaker but simpler alternative to weight is grouping information into larger objects so that a single token carries all necessary data (see additional functionality for guards at DecisionNode).

Other rules for when tokens may be passed along the edge depend on the kind of edge and characteristics of its source and target. See the children of ActivityEdge and ActivityNode. The rules may be optimized to a different algorithm as long as the effect is the same. For example, if the target is an object node that has reached its upper bound, no token can be passed. The implementation can omit unnecessary weight evaluations until the downstream object node can accept tokens.

Edges can be named, by inheritance from RedefinableElement, which is a NamedElement. However, edges are not required to have unique names within an activity. The fact that Activity is a Namespace, inherited through Behavior, does not affect this, because the containment of edges is through ownedElement, the general ownership metaassociation for Element that does not imply unique names, rather than ownedMember.

Edges inherited from more general activities can be replaced. See RedefinableElement for more information on overriding inherited elements.

Semantic Variation Points

See variations at children of ActivityEdge and ActivityNode.

Notation

An activity edge is notated by an open arrowhead line connecting two activity nodes. If the edge has a name, it is notated near the arrow.

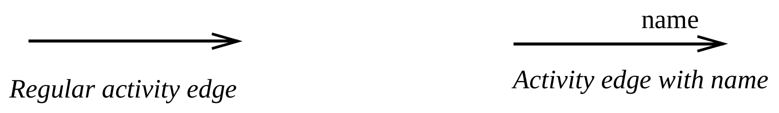


Figure 12.39 - Activity edge notation

An activity edge can 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 activity edge 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. This assumes the UML 2.0 Diagram Interchange specification supports the interchange of diagram elements and their mapping to model elements.

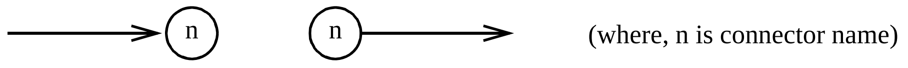


Figure 12.40 - Activity edge connector notation

Package CompleteActivities

The weight of the edge may be shown in curly braces that contain the weight. 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 “*”. When regions have interruptions, a lightning-bolt style activity edge expresses this interruption (see InterruptibleActivityRegion; see Pin for filled arrowhead notation).

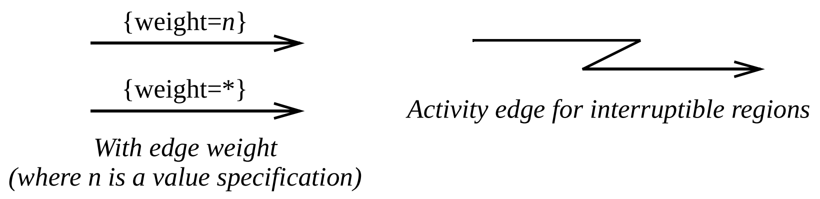

{weight=*n*}
{weight=*}
With edge weight
(where n is a value specification)
Activity edge for interruptible regions

Figure 12.41 - Activity edge notation

Examples

Package BasicActivities

In the example illustrated below, the arrowed line connecting Fill Order to Ship Order is a control flow edge. This means that when the Fill Order behavior is completed, control is passed to the Ship Order. Below it, the same control flow 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 (via the Invoice object node) are object flow edges. This indicates that the flow of Invoice objects goes from Send Invoice to Make Payment.

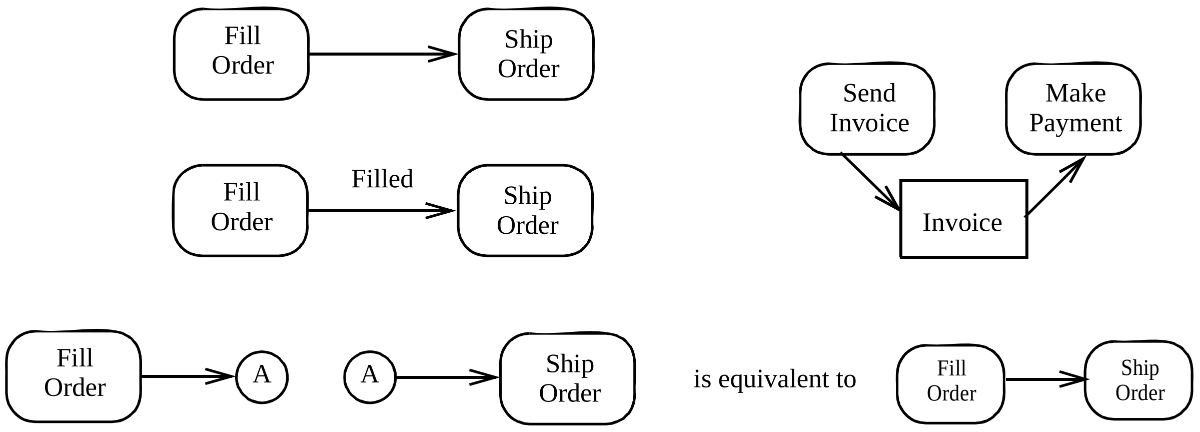


Figure 12.42 - Activity edge examples

In the example below, a connector is used to avoid drawing a long edge around one tine of the fork. If a problem is not priority one, the token going to the connector is sent to the merge instead of the one that would arrive from Revise Plan for priority one problems. This is equivalent to the activity shown in Figure 12.44, which is how Figure 12.43 is stored in the model.

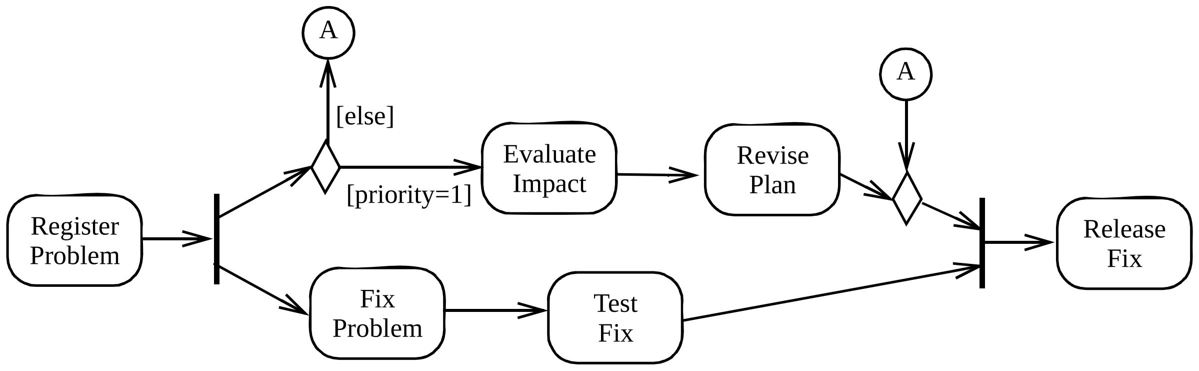


Figure 12.43 - Connector example

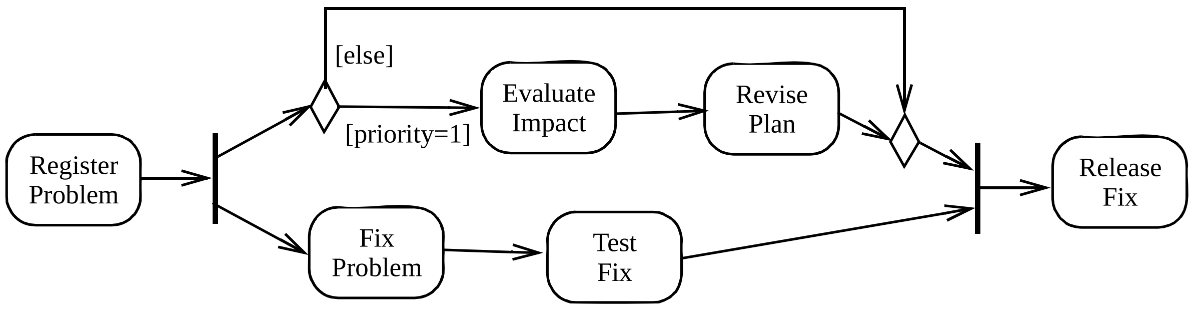


Figure 12.44 - Equivalent model to Figure 12.43

Package CompleteActivities

The figure below illustrates three examples of using the weight attribute. 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 proposal example depicts an activity for placing bids for a proposal, where many such bids can be placed. Then, when the bidding period is over, the Award Proposal Bid activity reads all the bids as a single set and determines which vendor to award the bid.

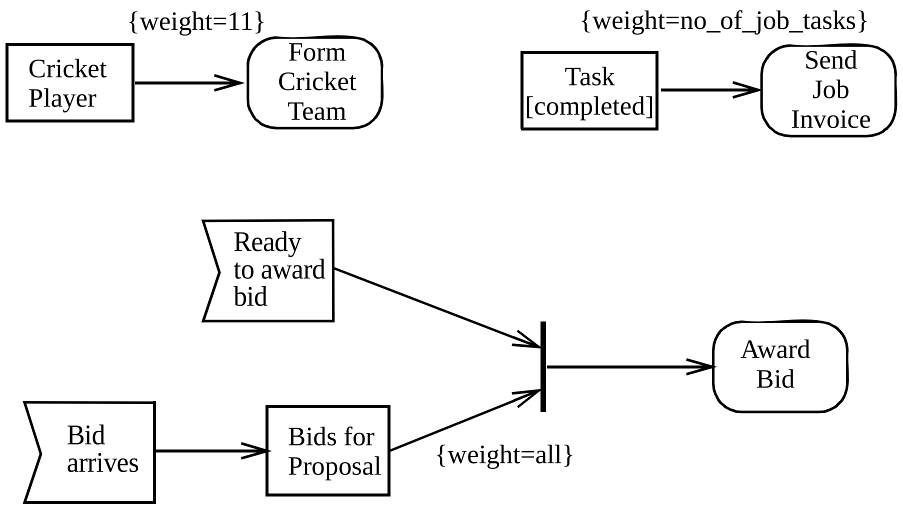


Figure 12.45 - Activity edge examples

Rationale

Activity edges are introduced to provide a general class for connections between activity nodes.

Changes from previous UML

ActivityEdge replaces the use of (state) Transition in UML 1.5 activity modeling. It also replaces data flow and control flow links in UML 1.5 action model.

12.3.6 ActivityFinalNode (from BasicActivities, IntermediateActivities)

An activity final node is a final node that stops all flows in an activity.

Generalizations

- “ControlNode (from BasicActivities)” on page 356
- “FinalNode (from IntermediateActivities)” on page 373

Description

An activity may have more than one activity final node. The first one reached stops all flows in the activity.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

A token reaching an activity final node terminates the activity (or structured node, see “StructuredActivityNode (from CompleteStructuredActivities, StructuredActivities)” on page 409). In particular, it stops all executing actions in the activity, and destroys all tokens in object nodes, except in the output activity parameter nodes. Terminating the execution of synchronous invocation actions also terminates whatever behaviors they are waiting on for return. Any behaviors invoked asynchronously by the activity are not affected. All tokens offered on the incoming edges are accepted. The content of output activity parameter nodes are passed out of the containing activity, using the null token for object nodes that have nothing in them. If there is more than one final node in an activity, the first one reached terminates the activity, including the flow going towards the other activity final.

If it is not desired to abort all flows in the activity, use flow final instead. For example, if the same execution of an activity is being used for all its invocations, then multiple streams of tokens will be flowing through the same activity. In this case, it is probably not desired to abort all tokens just because one reaches an activity final. Using a flow final will simply consume the tokens reaching it without aborting other flows. Or arrange for separate invocations of the activity to use separate executions of the activity, so tokens from separate invocations will not affect each other.

Notation

Activity final nodes are notated as a solid circle with a hollow circle, as indicated in the figure below. It can be thought of as a goal notated as “bull’s eye,” or target.

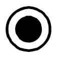


Figure 12.46 - Activity final notation

Examples

The first example below depicts that when the Close Order behavior is completed, all tokens in the activity are terminated. This is indicated by passing control to an activity final node.

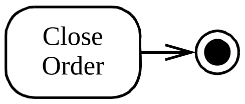


Figure 12.47 - Activity final example

The next figure is based on an example for an employee expense reimbursement process. It uses an activity diagram that illustrates two parallel flows racing to complete. The first one to reach the activity final aborts the others. The two flows appear in the same activity so they can share data. For example, who to notify in the case of no action.

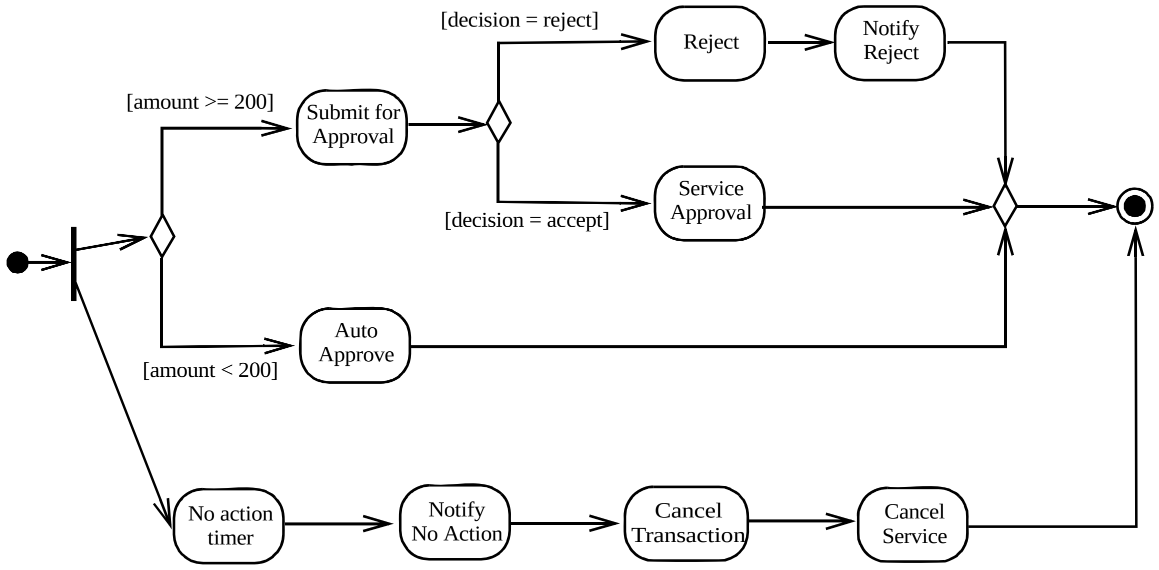


Figure 12.48 - Activity final example

In Figure 12.49, two ways to reach an activity final exist; but it is the result of exclusive "or" branching, not a "race" situation like the previous example. This example uses two activity final nodes, which has the same semantics as using one with two edges targeting it. The Notify of Modification behavior must not take long or the activity finals might kill it.

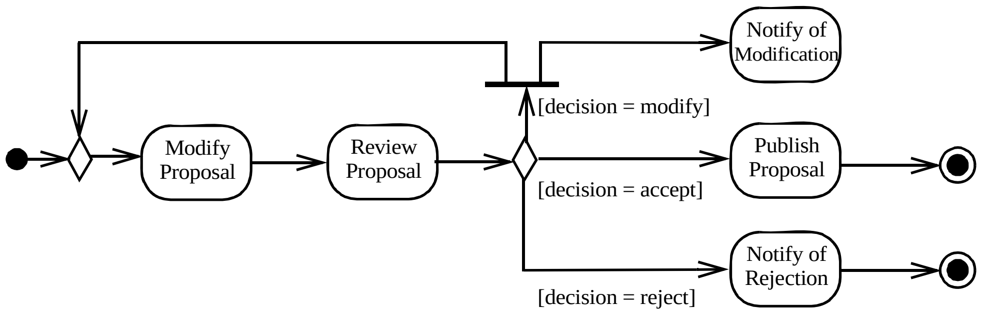


Figure 12.49 - Activity final example

Rationale

Activity final nodes are introduced to model non-local termination of all flows in an activity.

Changes from previous UML

ActivityFinal is new in UML 2.0.

12.3.7 ActivityGroup (from BasicActivities, FundamentalActivities)

An activity group is an abstract class for defining sets of nodes and edges in an activity.

Generalizations

- "Element (from Kernel)" on page 64

Description

Activity groups are a generic grouping construct for nodes and edges. Nodes and edges can belong to more than one group. They have no inherent semantics and can be used for various purposes. Subclasses of ActivityGroup may add semantics.

Attributes

No additional attributes

Associations

Package FundamentalActivities

- inActivity : Activity [0..1]
  Activity containing the group. {Subsets *NamedElement::owner*}
- /containedNode : ActivityNode [0..*] {readOnly}
  Nodes immediately contained in the group. This is a derived union.

- /superGroup : ActivityGroup [0..1]
  Group immediately containing the group.
- /subgroup : ActivityGroup [0..*]
  Groups immediately contained in the group.

Package BasicActivities

- /containedEdge : ActivityEdge [0..*]
  Edges immediately contained in the group. This is a derived union.

Constraints

[1] All nodes and edges of the group must be in the same activity as the group.

[2] No node or edge in a group may be contained by its subgroups or its containing groups, transitively.

[3] Groups may only be owned by activities or groups.

Semantics

None

Notation

No specific notation

Rationale

Activity groups provide a generic grouping mechanism that can be used for various purposes, as defined in the subclasses of ActivityGroup, and in extensions and profiles.

Changes from previous UML

ActivityGroups are new in UML 2.0.

12.3.8 ActivityNode (from BasicActivities, CompleteActivities, FundamentalActivities, IntermediateActivities, CompleteStructuredActivities)

An activity node is an abstract class for points in the flow of an activity connected by edges.

Generalizations

- “NamedElement (from Kernel, Dependencies)” on page 98
- “RedefinableElement (from Kernel)” on page 130

Description

An activity node is an abstract class for the steps of an activity. It covers executable nodes, control nodes, and object nodes.

(BasicActivities) Nodes can be replaced in generalization and (CompleteActivities) be contained in interruptible regions.

Attributes

No additional attributes

Associations

Package FundamentalActivities

- activity : Activity[0..1]
  Activity containing the node. {Subsets *NamedElement::owner*}
- /inGroup : Group [0..*]
  Groups containing the node. Multiplicity specialized to [0..1] for StructuredActivityGroup.

Package BasicActivities

- incoming : ActivityEdge [0..*]
  Edges that have the node as target.
- outgoing : ActivityEdge [0..*]
  Edges that have the node as source.
- redefinedNode : ActivityNode [0..*]
  Inherited nodes replaced by this node in a specialization of the activity. {Subsets *RedefinableElement::redefinedElement*}

Package IntermediateActivities

- inPartition : Partition [0..*]
  Partitions containing the node. {Subsets *ActivityNode::inGroup*}

Package CompleteActivities

- inInterruptibleRegion : InterruptibleActivityRegion [0..*]
  Interruptible regions containing the node. {Subsets *ActivityNode::inGroup*}
- inStructuredNode : StructuredActivityNode [0..1]
  Structured activity node containing the node. {Subsets *ActivityNode::inGroup*}

Constraints

[1] Activity nodes can only be owned by activities or groups.

Package StructuredActivities

[1] Activity nodes may be owned by at most one structured node.

Semantics

Nodes can be named, however, nodes are not required to have unique names within an activity to support multiple invocations of the same behavior or multiple uses of the same action. See Action, which is a kind of node. The fact that Activity is a Namespace, inherited through Behavior, does not affect this, because the containment of nodes is through ownedElement, the general ownership metaassociation for Element that does not imply unique names, rather than ownedMember. Other than naming, and functionality added by the complete version of activities, an activity node is only a point in an activity at this level of abstraction. See the children of ActivityNode for additional semantics.

Package BasicActivities

Nodes inherited from more general activities can be replaced. See RedefinableElement for more information on overriding inherited elements, and Activity for more information on activity generalization. See children of ActivityNode for additional semantics.

Notation

The notations for activity nodes are illustrated below. There are three kinds of nodes: action node, object node, and control node. See these classes for more information.

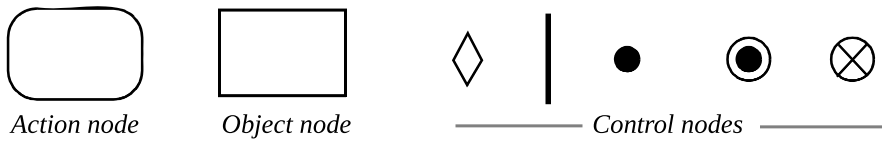


Figure 12.50 - Activity node notation

Examples

This figure illustrates the following kinds of activity node: action nodes (e.g., Receive Order, Fill Order), object nodes (Invoice), and control nodes (the initial node before Receive Order, the decision node after Receive Order, and the fork node and Join node around Ship Order, merge node before Close Order, and activity final after Close Order).

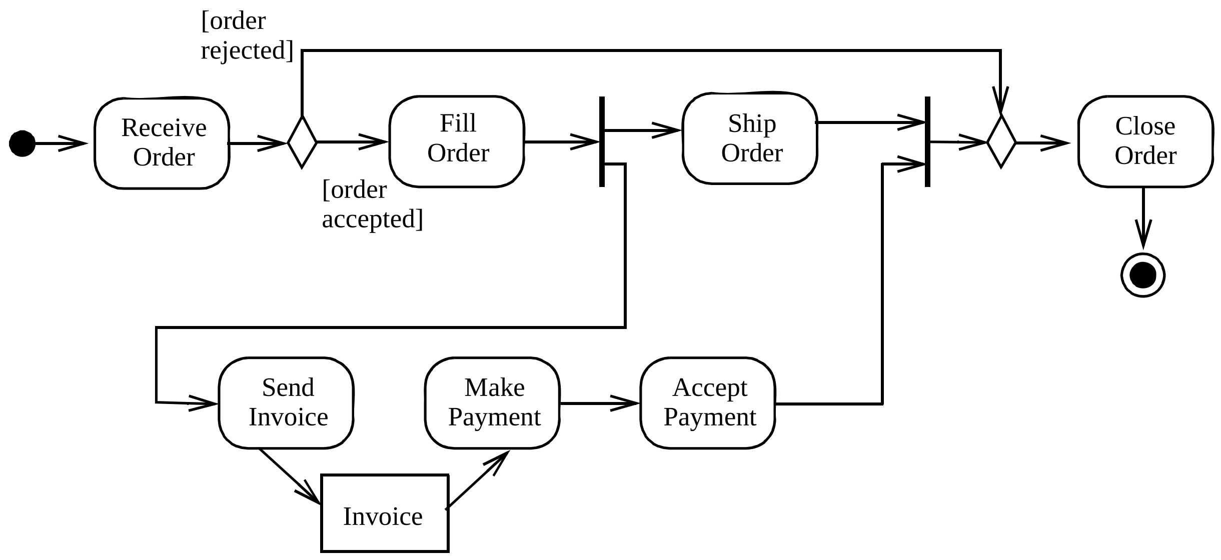


Figure 12.51 - Activity node example (where the arrowed lines are only the non-activity node symbols)

Rationale

Activity nodes are introduced to provide a general class for nodes connected by activity edges.

Changes from previous UML

ActivityNode replaces the use of StateVertex and its children for activity modeling in UML 1.5.

12.3.9 ActivityParameterNode (from BasicActivities)

An activity parameter node is an object node for inputs and outputs to activities.

Generalizations

- "ObjectNode (from BasicActivities, CompleteActivities)" on page 393

Description

Activity parameter nodes are object nodes at the beginning and end of flows that provide a means to accept inputs to an activity and provide outputs from the activity, through the activity parameters.

Activity parameters inherit support for streaming and exceptions from Parameter.

Attributes

No additional attributes

Associations

- parameter : Parameter [1..1]
  The parameter the object node will be accepting or providing values for.

Constraints

[1] Activity parameter nodes must have parameters from the containing activity.

[2] The type of an activity parameter node is the same as the type of its parameter.

[3] An activity parameter node may have either all incoming edges or all outgoing edges, but it must not have both incoming and outgoing edges.

[4] Activity parameter object nodes with no incoming edges and one or more outgoing edges must have a parameter with in or inout direction.

[5] Activity parameter object nodes with no outgoing edges and one or more incoming edges must have a parameter with out, inout, or return direction.

[6] A parameter with direction other than inout must have at most one activity parameter node in an activity.

[7] A parameter with direction inout must have at most two activity parameter nodes in an activity, one with incoming flows and one with outgoing flows.

See "Action (from CompleteActivities, FundamentalActivities, StructuredActivities)" on page 311.

Semantics

As a kind of behavior, an activity may have owned parameters. Within the activity, in and inout parameters may be associated with activity parameter nodes that have no incoming edges—they provide a source within the activity for the overall "input values" of the activity. Similarly, inout, out, and return parameters may be associated with activity nodes that have no outgoing edges—they provide a sink within the activity for the overall "output values" of the activity.

Per the general semantics of a behavior, when the activity is invoked, its in and inout parameters may be given actual values. These input values are placed as tokens on those activity parameter nodes within the activity that are associated with the corresponding in and inout parameters, the ones which do not have incoming edges. The overall activity input values are then available within the activity via the outgoing edges of the activity parameter nodes.

During the course of execution of the activity, tokens may flow into those activity parameter nodes within the activity that have incoming edges. When the execution of the activity completes, the output values held by these activity parameter nodes are given to the corresponding inout, out, and return parameters of the activity.

If the parameter associated with an activity parameter node is marked as streaming, then the above semantics are extended to allow for inputs to arrive and outputs to be posted during the execution of the activity (see the semantics for Parameter). In this case, for an activity parameter node with no incoming edges, an input value is placed on the activity parameter node whenever an input arrives on the corresponding streaming in or inout parameter. For an activity parameter node with no outgoing edges, an output value is posted on the corresponding inout, out or return parameter whenever a token arrives at the activity parameter node.

Notation

The label for parameter nodes can be a full specification of the corresponding parameter.

Also see notation at Activity.

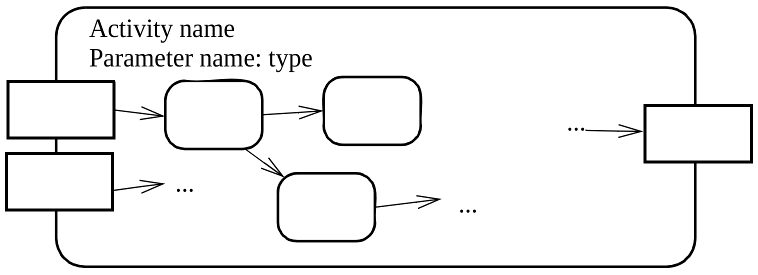


Figure 12.52 - Activity notation

The figure below shows annotations for streaming and exception activity parameters, which are the same as for pins. See Parameter for semantics of stream and exception parameters.

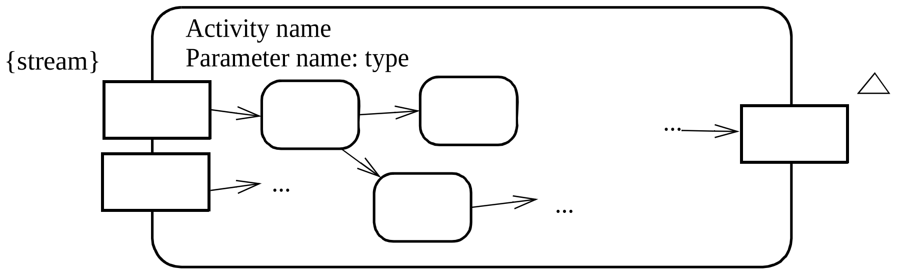


Figure 12.53 - Activity notation

Presentation Options

If the round-cornered border of Figure 12.53 is replaced with the frame notation that is described in Annex A, then activity parameter nodes overlap the frame instead. If the round-cornered border or frame is omitted completely, then the activity parameter nodes can be placed anywhere, but it is clearer if they are placed in the same locations they would be in if the frame or border was shown.

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.

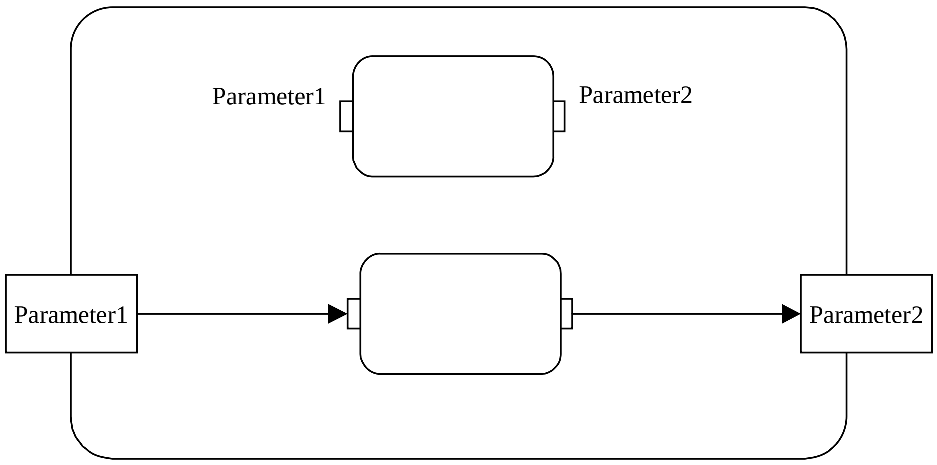


Figure 12.54 - Presentation option for flows between pins and parameter nodes

See presentation option for Pin when parameter is streaming. This can be used for activity parameters also.

Examples

In the example below, production materials are fed into printed circuit board. At the end of the activity, computers are quality checked.

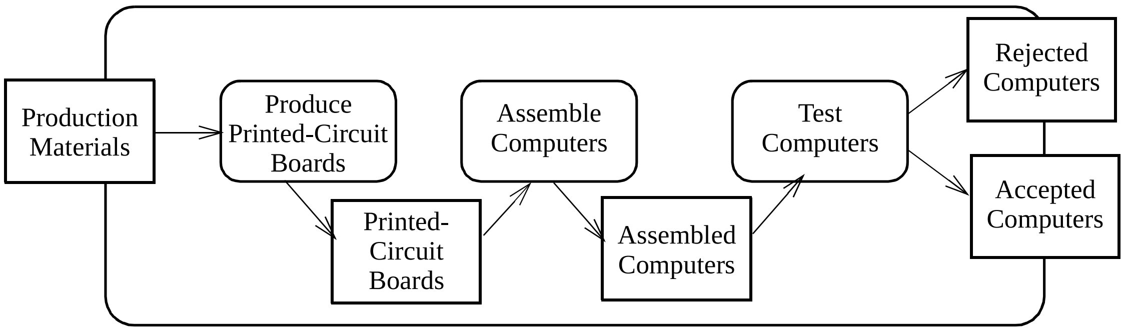


Figure 12.55 - Example of activity parameters.nodes

In the example below, 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. See Parameter for semantics of streaming and exception parameters.

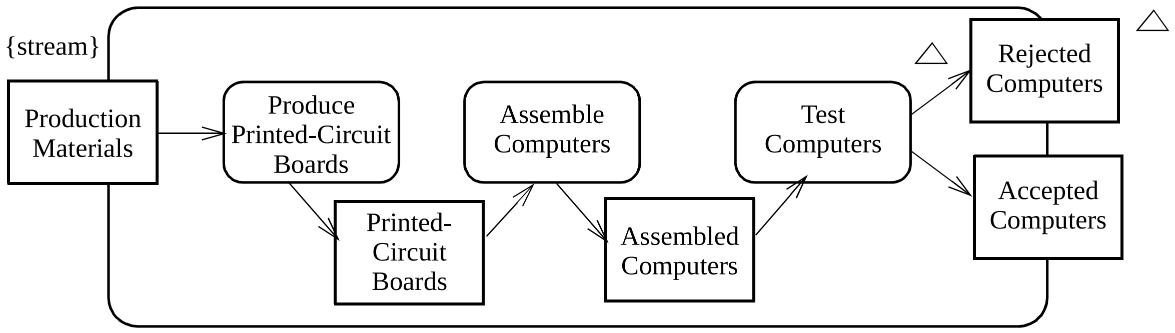


Figure 12.56 - Example of activity parameter nodes for streaming and exceptions

Rationale

Activity parameter nodes are introduced to model parameters of activities in a way that integrates easily with the rest of the flow model.

Changes from previous UML

ActivityParameterNode is new in UML 2.0.

12.3.10 ActivityPartition (from IntermediateActivities)

An activity partition is a kind of activity group for identifying actions that have some characteristic in common.

Generalizations

- "ActivityGroup (from BasicActivities, FundamentalActivities)" on page 332
- "NamedElement (from Kernel, Dependencies)" on page 98

Description

Partitions divide the nodes and edges to constrain and show a view of the contained nodes. Partitions can share contents. They often correspond to organizational units in a business model. They may be used to allocate characteristics or resources among the nodes of an activity.

Attributes

- isDimension : Boolean [1..1] = false
  Tells whether the partition groups other partitions along a dimension.
- isExternal : Boolean [1..1] = false
  Tells whether the partition represents an entity to which the partitioning structure does not apply.

Associations

- superPartition : ActivityPartition [0..1]
  Partition immediately containing the partition. (Subsets *ActivityGroup::superGroup*)
- represents : Element [0..1]
  An element constraining behaviors invoked by nodes in the partition.
- subpartition : ActivityPartition [0..*]
  Partitions immediately contained in the partition. (Subsets *ActivityGroup::subgroup)*.
- node : ActivityNode [0..*]
  Nodes immediately contained in the partition. (Subsets *ActivityGroup::containedNode)*
- edge : ActivityEdge [0..*]
  Edges immediately contained in the partition. (Subsets *ActivityGroup::containedEdge)*

Constraints

[1] A partition with isDimension = true may not be contained by another partition.

[2] If a partition represents a part, then all the non-external partitions in the same dimension and at the same level of nesting in that dimension must represent parts directly contained in the internal structure of the same classifier.

[3] If a non-external partition represents a classifier and is contained in another partition, then the containing partition must represent a classifier, and the classifier of the subpartition must be nested in the classifier represented by the containing partition, or be at the contained end of a strong composition association with the classifier represented by the containing partition.

[4] If a partition represents a part and is contained by another partition, then the part must be of a classifier represented by the containing partition, or of a classifier that is the type of a part representing the containing partition.

Semantics

Partitions do not affect the token flow of the model. They constrain and provide a view on the behaviors invoked in activities. Constraints vary according to the type of element that the partition represents. The following constraints are normative:

1) Classifier

Behaviors of invocations contained by the partition are the responsibility of instances of the classifier represented by the partition. This means the context of invoked behaviors is the classifier. Invoked procedures containing a call to an operation or sending a signal must target objects at runtime that are instances of the classifier.

2) Instance

This imposes the same constraints as classifier, but restricted to a particular instance of the classifier.

3) Part

Behaviors of invocations contained by the partition are the responsibility of instances playing the part represented by the partition. This imposes the constraints for classifiers above according to the type of the part. In addition, invoked procedures containing a call to an operation or sending a signal must target objects at runtime that play the part at the time the message is sent. Just as partitions in the same dimension and nesting must be represented by parts of the same classifier's internal structure, all the runtime target objects of operation and signal passing invoked by the same execution of the activity must play parts of the same instance of the structured classifier. In particular, if an activity is executed in the context of a particular object at runtime, the parts of that object will be used as targets. If a part has more than one object playing it at runtime, the invocations are treated as if they were multiple, that is, the calls are sent in parallel, and the invocation does not complete until all the operations return.

4) Attribute and Value

A partition may be represented by an attribute and its subpartitions by values of that attribute. Behaviors of invocations contained by the subpartition have this attribute and the value represented by the subpartition. For example, a partition may represent the location at which a behavior is carried out, and the subpartitions would represent specific values for that attribute, such as Chicago. The location attribute could be on the process class associated with an activity, or added in a profile to extend behaviors with these attributes.

A partition may be marked as being a dimension for its subpartitions. For example, an activity may have one dimension of partitions for location at which the contained behaviors are carried out, and another for the cost of performing them. Dimension partitions cannot be contained in any other partition.

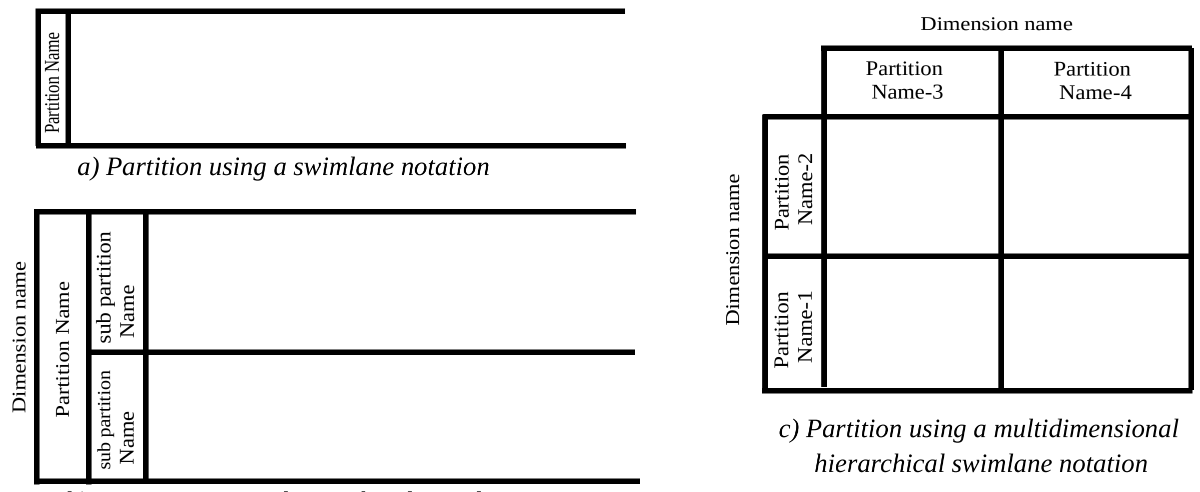
Elements other than actions that have behaviors or value specifications, such as transformation behaviors on edges, adhere to the same partition rules above for actions.

Partitions 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 in that partition are the responsibility of instances of the classifier, but the model may or may not say which instance in particular. In particular, a behavior in the partition calling an operation would be limited to an operation on that classifier, but an input object flow to the invocation might not be specified to tell which instance should be the target at runtime. The object flow could be specified in a later stage of development to support execution. Another option would be to use partitions 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, as described above.

External partitions are intentional exceptions to the rules for partition structure. For example, a dimension may have partitions showing parts of a structured classifier. It can 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.

Notation

Activity partition may be indicated with two, usually parallel lines, either horizontal or vertical, and a name labeling the partition in a box at one end. Any activity nodes and edges placed between these lines are considered to be contained within the partition. Swimlanes can express hierarchical partitioning by representing the children in the hierarchy as further partitioning of the parent partition, as illustrated in b), below. Diagrams can also be partitioned multidimensionally, as depicted in c), below, where, each swim cell is an intersection of multiple partitions. The specification for each dimension (e.g., part, attribute) is expressed in next to the appropriate partition set.



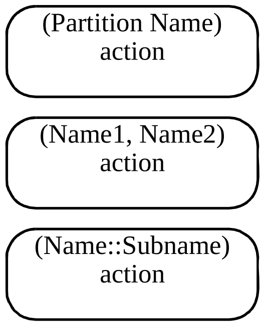
a) Partition using a swimlane notation

b) Partition using a hierarchical swimlane notation

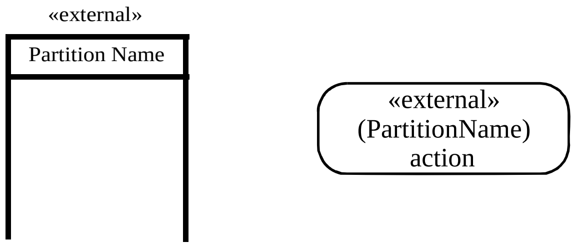
c) Partition using a multidimensional hierarchical swimlane notation

Figure 12.57 - Activity partition notations

In some diagramming situations, using parallel lines to delineate partitions is not practical. An alternate is to place the partition name in parenthesis above the activity name, as illustrated for actions in a), below. A comma-delimited 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 larger partitions coming earlier in the name. When activities are considered to occur outside the domain of a particular model, the partition can be labeled with the keyword «external», as illustrated in b) below. Whenever an activity in a swimlane is marked «external», this overrides the swimlane and dimension designation.



a) Partition notated on a specific activity



b) Partition notated to occur outside the primary concern of the model.

Figure 12.58 - Activity partition notations

Presentation Options

When partitions are combined with the frame notation for Activity, the outside edges of the top level partition can be merged with the activity frame.

Examples

The figures below illustrate an example of partitioning the order processing activity diagram into “swim lanes.” 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. These are attributes of the behavior invoked in the partitions, except for Customer, which is external to the domain. The flow of the invoice is not a behavior, so it does not need to appear in a partition.

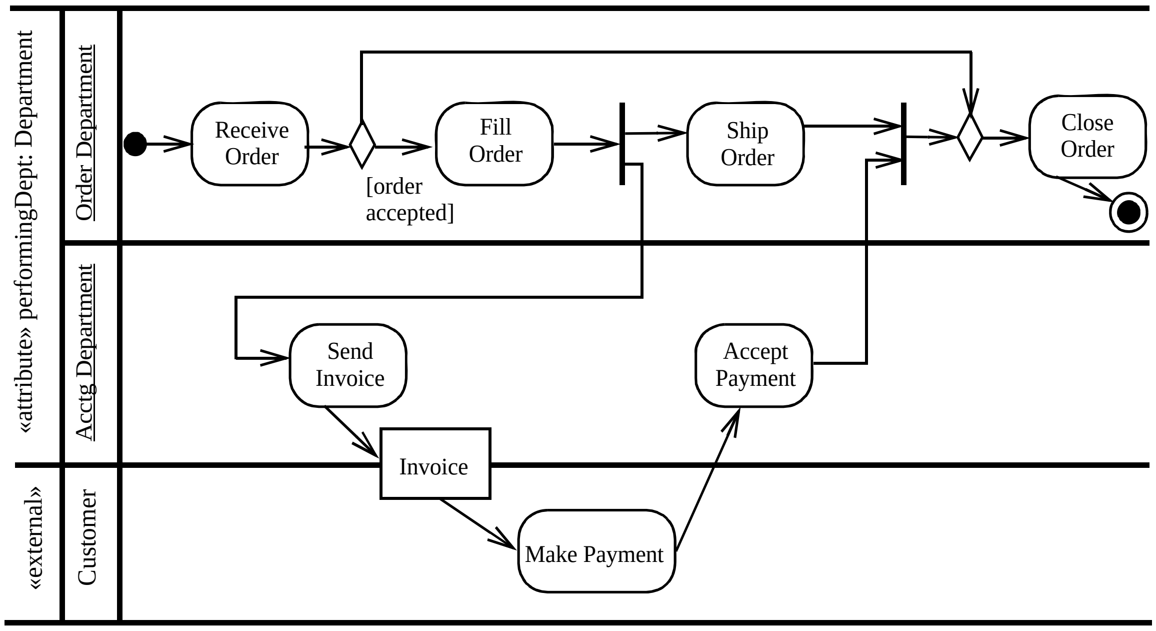


Figure 12.59 - Activity partition using swimlane example

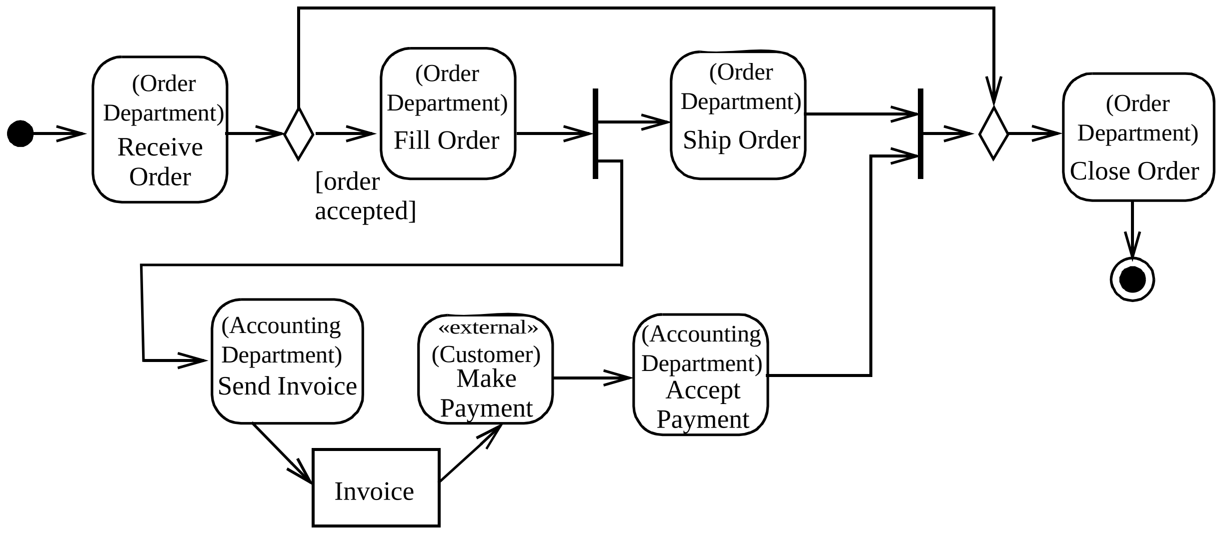

(Order Department) Receive Order
[order accepted]
(Order Department) Fill Order
(Order Department) Ship Order
(Order Department) Close Order
(Accounting Department) Send Invoice
«external» (Customer) Make Payment
(Accounting Department) Accept Payment
Invoice

Figure 12.60 - Activity partition using annotation example

The example below depicts multidimensional swim lanes. 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 is contained within the Seattle/Accounting Clerk swim cell, its performer and location are not specified by the containing partition, because it has an overriding partition.

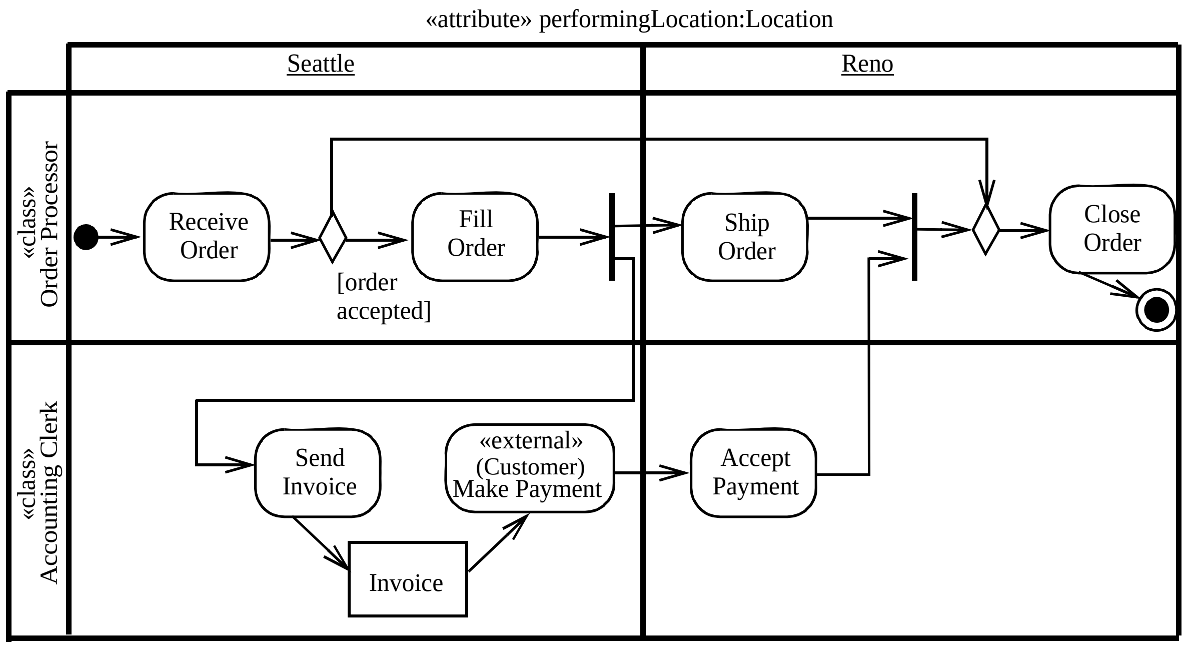


Figure 12.61 - Activity partition using multidimensional swimlane example

Rationale

Activity partitions are introduced to support the assignment of domain-specific information to nodes and edges.

Changes from previous UML

Edges can be contained in partitions in UML 2.0. Additional notation is provided for cases when swimlanes are too cumbersome. Partitions can be hierarchical and multidimensional. The relation to classifier, parts, and attributes is formalized, including external partitions as exceptions to these rules.

12.3.11 AddVariableValueAction (as specialized)

See “AddVariableValueAction (from StructuredActions)” on page 240.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See “AddVariableValueAction (from StructuredActions)” on page 240.

Notation

Presentation Options

The presentation option at the top of Figure 12.62 may be used as notation for a model corresponding to the notation at the bottom of the figure. If the action has non-defaulted metaattribute values, these can be shown with a property list near the variable name.

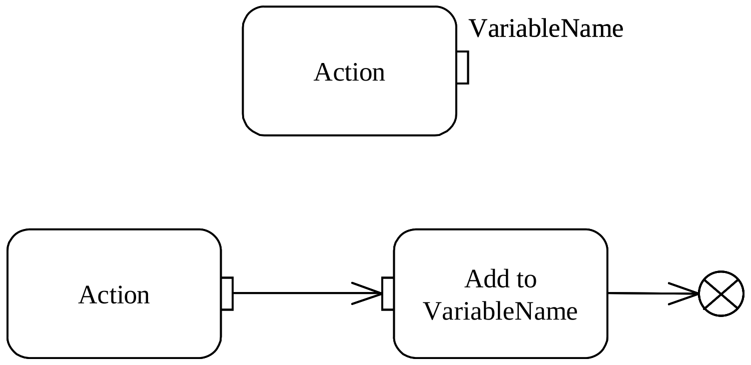


Figure 12.62 - Presentation option for AddVariableValueAction

12.3.12 Behavior (from CompleteActivities)

Behavior is specialized to own zero or more ParameterSets.

Generalizations

- “Behavior (from BasicBehaviors)” on page 430 *(merge increment)*.

Description

The concept of Behavior is extended to own ParameterSets.

Attributes

No additional attributes

Associations

- ownedParameterSet : ParameterSet[0..*]
  The ParameterSets owned by this Behavior. {Subsets *Namespace::ownedMember*}

Constraints

See “ParameterSet (from CompleteActivities)” on page 399.

Semantics

See semantics of “ParameterSet (from CompleteActivities)” on page 399.

Notation

See notation for “ParameterSet (from CompleteActivities)” on page 399.

Examples

See examples for “ParameterSet (from CompleteActivities)” on page 399.

Changes from previous UML

ParameterSet is new in UML 2.0.

12.3.13 BehavioralFeature (from CompleteActivities)

BehavioralFeature is specialized to own zero or more ParameterSets.

Generalizations

- “BehavioralFeature (from BasicBehaviors, Communications)” on page 432 *(merge increment)*.

Description

The concept of BehavioralFeature is extended to own ParameterSets.

Attributes

No additional attributes

Associations

- ownedParameterSets : ParameterSet[0..*]
  The ParameterSets owned by this BehavioralFeature.

Constraints

See “ParameterSet (from CompleteActivities)” on page 399.

Semantics

See semantics of “ParameterSet (from CompleteActivities)” on page 399.

Notation

See notation for “ParameterSet (from CompleteActivities)” on page 399.

Examples

See examples for “ParameterSet (from CompleteActivities)” on page 399.

Changes from previous UML

ParameterSet is new in UML 2.0.

12.3.14 CallBehaviorAction (as specialized)

“CallBehaviorAction (from BasicActions)” on page 243

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

[1] When all the control and data flow prerequisites of the action execution are satisfied, CallBehaviorAction consumes its input tokens and invokes its specified behavior. The values in the input tokens are made available to the invoked behavior as argument values. When the behavior is finished, tokens are offered on all outgoing control edges, with a copy made for each control edge. Object and data tokens are offered on the outgoing object flow edges as determined by the output pins. Each parameter of the behavior of the action provides output to a pin or takes input from one (see Pin). The inputs to the action determine the actual arguments of the call.

[2] If the call is asynchronous, a control token is offered to each outgoing control edge of the action and execution of the action is complete. Execution of the invoked behavior proceeds without any further dependency on the execution of the activity containing the invoking action. Once the invocation of the behavior has been initiated, execution of the asynchronous action is complete.

[3] An asynchronous invocation completes when its behavior is started, or is at least ensured to be started at some point. When an asynchronous invocation is done, the flow continues regardless of the status of the invoked behavior. Any return or out values from the invoked behavior are not passed back to the containing activity. For example, the containing activity may complete even though the invoked behavior is not finished. This is why asynchronous invocation is not the same as using a fork to invoke the behavior followed by a flow final. A forked behavior still needs to finish for the containing activity to finish. If it is desired to complete the invocation, but have some outputs provided later when they are needed, then use a fork to give the invocation its own flow line, and rejoin the outputs of the invocation to the original flow when they are needed.

[4] If the call is synchronous, execution of the calling action is blocked until it receives a reply token from the invoked behavior. The reply token includes values for any return, out, or inout parameters.

[5] If the call is synchronous, when the execution of the invoked behavior completes, the result values are placed as object tokens on the result pins of the call behavior action, a control token is offered on each outgoing control edge of the call behavior action, and the execution of the action is complete. (StructuredActions, ExtraStructuredActivities) If the execution of the invoked behavior yields an exception, the exception is transmitted to the call behavior action to begin the search for the handler. See “RaiseExceptionAction (from StructuredActions)” on page 265.

Notation

The name of the behavior, or other description of it, that is performed by the action is placed inside the rectangle. If the node name is different than the behavior name, then it appears in the symbol instead. Pre- and post-conditions on the behavior can be shown similarly to Figure 12.29 on page 313, using keywords «precondition» and «postcondition».

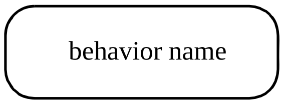


Figure 12.63 - CallBehaviorAction

The call of an activity is indicated by placing a rake-style symbol within the symbol. 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. Edges flowing into the invocation connect to the parameter object nodes in the invoked activity. The parameter object nodes are shown on the border of the invoked activity. The model is the same regardless of the choice of notation. This assumes the UML 2.0 Diagram Interchange specification supports the interchange of diagram elements and their mapping to model elements.

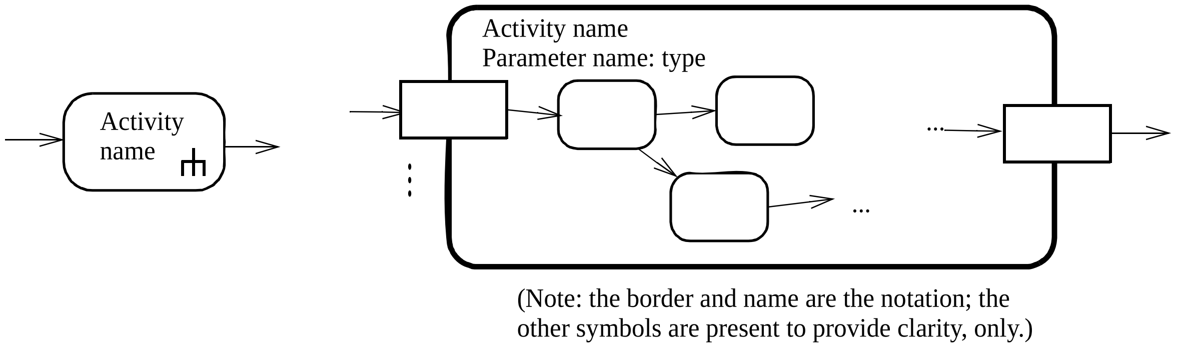

Activity name
Activity name
Parameter name: type
...
...
(Note: the border and name are the notation; the other symbols are present to provide clarity, only.)

Figure 12.64 - Invoking Activities that have nodes and edges

Below is an example of invoking an activity called FillOrder.

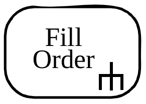


Figure 12.65 - Example of invoking an activity

Rationale

"CallBehaviorAction (from BasicActions)" on page 243

Changes from previous UML

"CallBehaviorAction (from BasicActions)" on page 243

12.3.15 CallOperationAction (as specialized)

See "CallOperationAction (from BasicActions)" on page 245.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See "CallOperationAction (from BasicActions)" on page 245.

Notation

The name of the operation, or other description of it, is displayed in the symbol. Pre- and post-conditions on the operation can be shown similarly to Figure 12.29 on page 313, using keywords «precondition» and «postcondition».

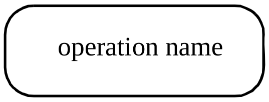


Figure 12.66 - Calling an operation

Presentation Options

If the node has a different name than the operation, then this is used in the symbol instead. The name of the class may optionally appear below the name of the operation, in parentheses postfixed by a double colon. If the node name is different than the operation name, then the behavioral feature name may be shown after the double colon.

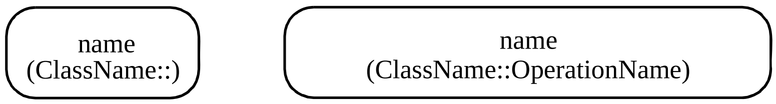


Figure 12.67 - Invoking behavioral feature notations

Rationale

See "CallOperationAction (from BasicActions)" on page 245.

Changes from previous UML

See “CallOperationAction (from BasicActions)” on page 245.

12.3.16 CentralBufferNode (from IntermediateActivities)

A central buffer node is an object node for managing flows from multiple sources and destinations.

Generalizations

- “ObjectNode (from BasicActivities, CompleteActivities)” on page 393

Description

A central buffer node accepts tokens from upstream object nodes and passes them along to downstream object nodes. They act as a buffer for multiple in flows and out flows from other object nodes. They do not connect directly to actions.

Attributes

No additional attributes

Associations

No additional associations

Semantics

See semantics at ObjectNode. All object nodes have buffer functionality, but central buffers differ in that they are not tied to an action as pins are, or to an activity as activity parameter nodes are. See example below.

Notation

See notation at ObjectNode. A central buffer may also have the keyword «centralBuffer» as shown below. This is useful when it needs to be distinguished from the standalone notation for pins shown at the top of Figure 12.120 and Figure 12.127.

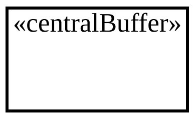


Figure 12.68 - Optional central buffer notation

Examples

In the example below, the behaviors for making parts at two factories produce finished parts. The central buffer node 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 object node by one outgoing edge. The choice in this example is non-deterministic.

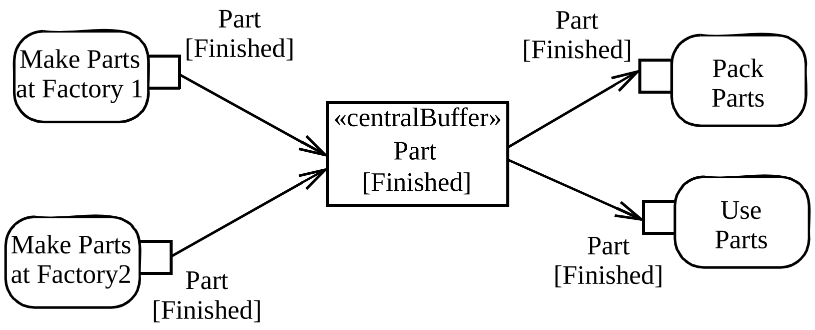


Figure 12.69 - Central buffer node example

Rationale

Central buffer nodes give additional support for queuing and competition between flowing objects.

Changes from previous UML

CentralBufferNode is new in UML 2.0.

12.3.17 Clause (from CompleteStructuredActivities, StructuredActivities)

Generalizations

- "Element (from Kernel)" on page 64

Description

A clause is an element that represents a single branch of a conditional construct, including a test and a body section. The body section is executed only if (but not necessarily if) the test section evaluates true.

Attributes

No additional attributes

Associations

Package StructuredActivities

- test : ExecutableNode [0..*]
  A nested activity fragment with a designated output pin that specifies the result of the test.

- body : ExecutableNode [0..*]
  A nested activity fragment that is executed if the test evaluates to true and the clause is chosen over any concurrent clauses that also evaluate to true.
- predecessorClause : Clause [*]
  A set of clauses whose tests must all evaluate false before the current clause can be tested.
- successorClause : Clause [*]
  A set of clauses that may not be tested unless the current clause tests false.
- decider : OutputPin [1]
  An output pin within the test fragment the value of which is examined after execution of the test to determine whether the body should be executed.

Package CompleteStructuredActivities

- bodyOutput : OutputPin [0..*] {ordered}
  A list of output pins within the body fragment whose values are moved to the result pins of the containing conditional node after execution of the clause body.

Constraints

Package StructuredActivities

[1] The decider output pin must be for the test body or a node contained by the test body as a structured node.

Package CompleteStructuredActivities

[1] The bodyOutput pins are output pins on actions in the body of the clause.

Semantics

The semantics are explained under "ConditionalNode (from CompleteStructuredActivities, StructuredActivities)."

12.3.18 ConditionalNode (from CompleteStructuredActivities, StructuredActivities)

A conditional node is a structured activity node that represents an exclusive choice among some number of alternatives.

Generalizations

- "StructuredActivityNode (from CompleteStructuredActivities, StructuredActivities)" on page 409

Description

A conditional node consists of one or more clauses. Each clause consists of a test section and a body section. When the conditional node begins execution, the test sections of the clauses are executed. If one or more test sections yield a true value, one of the corresponding body sections will be executed. If more than one test section yields a true value, only one body section will be executed. The choice is nondeterministic unless the test sequence of clauses is specified. If no test section yields a true value, then no body section is executed; this may be a semantic error if output values are expected from the conditional node.

In general, test section may be executed in any order, including simultaneously (if the underlying execution architecture supports it). The result may therefore be nondeterministic if more than one test section can be true concurrently. To enforce ordering of evaluation, sequencing constraints may be specified among clauses. One frequent case is a total ordering of clauses, in which case the clause execution order is determinate. If it is impossible for more than one test section to evaluate true simultaneously, the result is deterministic and it is unnecessary to order the clauses, as ordering may impose undesirable and unnecessary restrictions on implementation. Note that, although evaluation of test sections may be specified as concurrent, this does not require that the implementation evaluate them in parallel; it merely means that the model does not impose any order on evaluation.

An "else" clause is a clause that is a successor to all other clauses in the conditional and whose test part always returns true.

Output values created in the test or body section of a clause are potentially available for use outside the conditional. However, any value used outside the conditional must be created in every clause; otherwise, an undefined value would be accessed if a clause not defining the value were executed.

Attributes

Package StructuredActivities

- isAssured : Boolean
  If true, the modeler asserts that at least one test will succeed. Default value is *false*.
- isDeterminate: Boolean
  If true, the modeler asserts that at most one test will succeed. Default value is *false*.

Associations

Package StructuredActivities

- clause : Clause[1..*]
  Set of clauses composing the conditional.

Package CompleteStructuredActivities

- result : OutputPin [0..*]
  A list of output pins that constitute the data flow outputs of the conditional. {Subsets *Action::output*}

Constraints

[1] The result output pins have no incoming edges.

Semantics

No part of a conditional node is executed until all control-flow or data-flow predecessors of the conditional node have completed execution. When all such predecessors have completed execution and made tokens available to inputs of the conditional node, the conditional node captures the input tokens and begins execution.

The test section of any clause without a predecessorClause is eligible for execution immediately. If a test section yields a false value, a control token is delivered to all of its successorClauses. Any test section with a predecessorClause is eligible for execution when it receives control tokens from each of its predecessor clauses.

If a test section yields a true value, then the corresponding body section is executed provided another test section does not also yield a true value. If more than one test section yields a true value, exactly one body section will be executed, but it is indeterminate which one will be executed. When a body section is chosen for execution, the evaluation of all other test parts is terminated (just like an interrupting edge). If some of the test parts have external effects, terminating them may be another source of indeterminacy. Although test parts are permitted to produce side effects, avoiding side effects in tests will greatly reduce the chance of logical errors and race conditions in a model and in any code generated from it.

If no test section yields a true value, the execution of the conditional node terminates with no outputs. This may be a semantic error if a subsequent node requires an output from the conditional. It is safe if none of the clauses create outputs. If the *isAssured* attribute of the conditional node has a true value, the modeler asserts that at least one test section will yield a test value. If the *isDeterminate* attribute has a true value, the modeler asserts that at most one test section will yield a test value (the predecessor relationship may be used to enforce this assertion). Note that it is, in general, impossible for a computer system to verify these assertions, so they may provide useful information to a code generator, but if the assertions are incorrect, then incorrect code may be generated.

When a body section is chosen for execution, all of its nodes without predecessor flows within the conditional receive control tokens and are enabled for execution. When execution of all nodes within the body section has completed, execution of the conditional node is complete and its successors are enabled.

Within the body section, variables defined in the loop node or in some higher-level enclosing node may be accessed and updated with new values. Values that are used in a data flow manner must be created or updated in all clauses of the conditional; otherwise, undefined values would be accessed.

Notation

No specific notation.

Style Guidelines

Mixing sequential and concurrent tests in one conditional may be confusing, although it is permitted.

Rationale

Conditional nodes are introduced to provide a structured way to represent decisions.

Changes from previous UML

Conditional nodes replace ConditionalAction from the UML 1.5 action model.

12.3.19 ControlFlow (from BasicActivities)

A control flow is an edge that starts an activity node after the previous one is finished.

Generalizations

- “ActivityEdge (from BasicActivities, CompleteActivities, CompleteStructuredActivities, IntermediateActivities)” on page 325.

Description

Objects and data cannot pass along a control flow edge.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] Control flows may not have object nodes at either end, except for object nodes with control type.

Semantics

See semantics inherited from ActivityEdge. A control flow is an activity edge that only passes control tokens. Tokens offered by the source node are all offered to the target node.

Notation

A control flow is notated by an arrowed line connecting two actions.

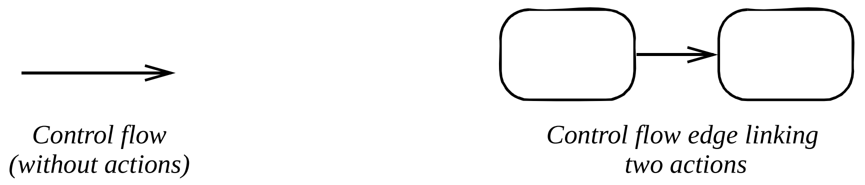


Figure 12.70 - Control flow notation

Examples

The figure below depicts an example of the Fill Order action passing control to the Ship Order action. The activity edge between the two is a control flow, which indicates that when Fill Order is completed, Ship Order is invoked.

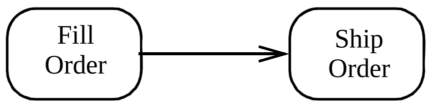


Figure 12.71 - Control flow example

Rationale

Control flow is introduced to model the sequencing of behaviors that does not involve the flow of objects.

Changes from previous UML

Explicitly modeled control flows are new to activity modeling in UML 2.0. They replace the use of (state) Transition in UML 1.5 activity modeling. They replace control flows in UML 1.5 action model.

12.3.20 ControlNode (from BasicActivities)

A control node is an abstract activity node that coordinates flows in an activity.

Generalizations

- “ActivityNode (from BasicActivities, CompleteActivities, FundamentalActivities, IntermediateActivities, CompleteStructuredActivities)” on page 333.

Description

A control node is an activity node used to coordinate the flows between other nodes. It covers initial node, final node and its children, fork node, join node, decision node, and merge node.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See semantics at Activity. See subclasses for the semantics of each kind of control node.

Notation

The notations for control nodes are illustrated below: decision node, initial node, activity final, and flow final.

Fork node and join node are the same symbol, they have different semantics and are distinguished notationally by the way edges are used with them. For more information, see ForkNode and JoinNode below.

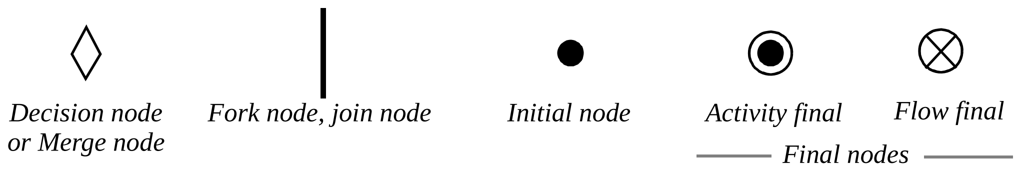


Figure 12.72 - Control node notations

Examples

The figure below contains examples of various kinds of control nodes. An initial node is depicted in the upper left as triggering the Receive Order action. A decision node after Received Order illustrates branching based on order rejected or order accepted conditions. Fill Order is followed by a fork node that passes control both to Send Invoice and Ship Order.

The join node indicates that control will be passed to the merge when both Ship Order and Accept Payment are completed. Since a merge will just pass the token along, Close Order activity will be invoked. (Control is also passed to Close Order whenever an order is rejected.) When Close Order is completed, control passes to an activity final.

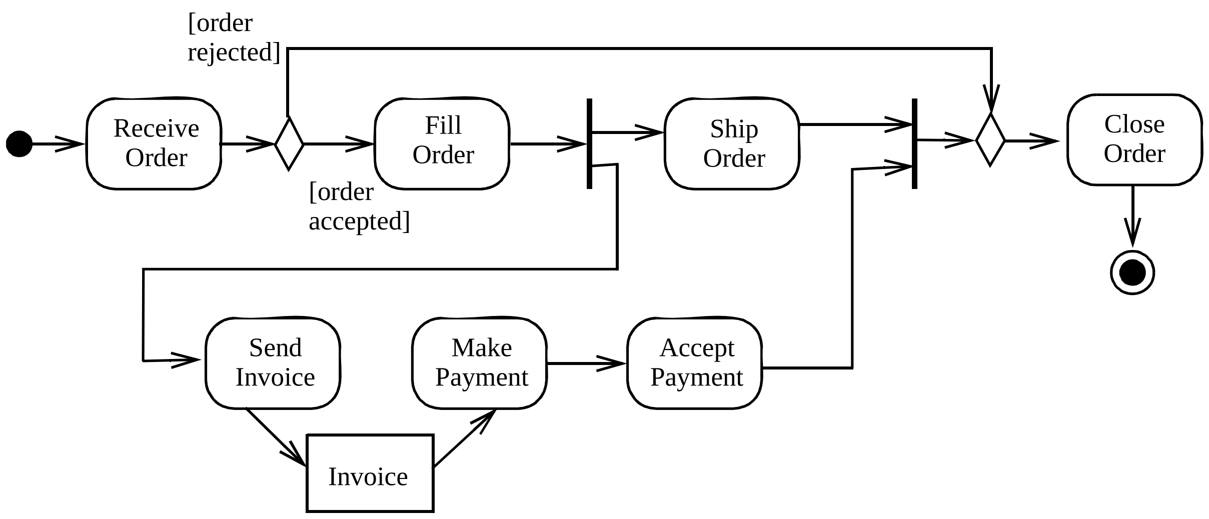


Figure 12.73 - Control node examples (with accompanying actions and control flows)

Rationale

Control nodes are introduced to provide a general class for nodes that coordinate flows in an activity.

Changes from previous UML

ControlNode replaces the use of PseudoState in UML 1.5 activity modeling.

12.3.21 DataStoreNode (from CompleteActivities)

A data store node is a central buffer node for non-transient information.

Generalizations

- "CentralBufferNode (from IntermediateActivities)" on page 351

Description

A data store keeps all tokens that enter it, copying them when they are chosen to move downstream. Incoming tokens containing a particular object replace any tokens in the object node containing that object.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Tokens chosen to move downstream are copied so that tokens appear to never leave the data store. If a token containing an object is chosen to move into a data store, and there is a token containing that object already in the data store, then the chosen token replaces the existing one. Selection and transformation behavior on outgoing edges can be designed to get information out of the data store, 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. Selection can also be designed to only succeed when a downstream action has control passed to it, thereby implementing the pull semantics of earlier forms of data flow.

Notation

The data store notation is a special case of the object node notation, using the label «datastore».

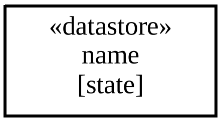


Figure 12.74 - Data store node notation.

Examples

The figure below is an example of using a data store node.

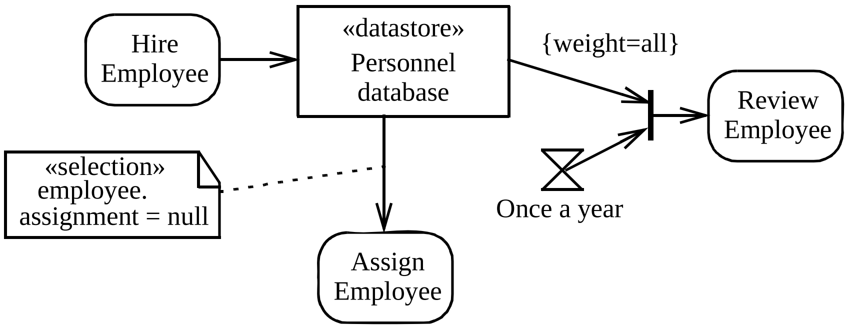

Hire Employee
«datastore» Personnel database
{weight=all}
Review Employee
«selection» employee. assignment = null
Once a year
Assign Employee

Figure 12.75 - Data store node example

Rationale

Data stores are introduced to support earlier forms of data flow modeling in which data is persistent and used as needed, rather than transient and used when available.

Changes from previous UML

Data stores are new in UML 2.0.

12.3.22 DecisionNode (from IntermediateActivities)

A decision node is a control node that chooses between outgoing flows.

Generalizations

- "ControlNode (from BasicActivities)" on page 356

Description

A decision node accepts tokens on an incoming edge and presents them to multiple outgoing edges. Which of the edges is actually traversed depends on the evaluation of the guards on the outgoing edges.

Attributes

No additional attributes

Associations

- decisionInput : Behavior [0..1]
  Provides input to guard specifications on edges outgoing from the decision node.
- decisionInputFlow : ObjectFlow [0..1]
  An additional edge incoming to the decision node that provides a decision input value.

Constraints

[1] A decision node has one or two incoming edges and at least one outgoing edge.

[2] The edges coming into and out of a decision node, other than the decision input flow (if any), must be either all object flows or all control flows.

[3] The decisionInputFlow of a decision node must be an incoming edge of the decision node.

[4] A decision input behavior has no output parameters, no in-out parameters and one return parameter.

[5] If the decision node has no decision input flow and an incoming control flow, then a decision input behavior has zero input parameters.

[6] If the decision node has no decision input flow and an incoming object flow, then a decision input behavior has one input parameter whose type is the same as or a supertype of the type of object tokens offered on the incoming edge.

[7] If the decision node has a decision input flow and an incoming control flow, then a decision input behavior has one input parameter whose type is the same as or a supertype of the type of object tokens offered on the decision input flow.

[8] If the decision node has a decision input flow and a second incoming object flow, then a decision input behavior has two input parameters, the first of which has a type that is the same as or a supertype of the type of the type of object tokens offered on the non-decision input flow 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 decision input flow.

Semantics

Each token arriving at a decision node can traverse only one outgoing edge. Tokens are not duplicated. Each token offered by the incoming edge is offered to the outgoing edges.

Most commonly, guards of the outgoing edges are evaluated to determine which edge should be traversed. The order in which guards are evaluated is not defined, because edges in general are not required to determine which tokens they accept in any particular order. The modeler should arrange that each token only be chosen to traverse one outgoing edge; otherwise, there will be race conditions among the outgoing edges. If the implementation can ensure that only one guard will succeed, it is not required to evaluate all guards when one is found that does. For decision points, a predefined guard "else" may be defined for at most one outgoing edge. This guard succeeds for a token only if the token is not accepted by all the other edges outgoing from the decision point.

Notice that the semantics only requires that the token traverse one edge, rather than be offered to only one edge. Multiple edges may be offered the token, but if only one of them has a target that accepts the token, then that edge is traversed. If multiple edges accept the token and have approval from their targets for traversal at the same time, then the semantics is not defined.

If a decision input behavior is specified, then each data token is passed to the behavior before guards are evaluated on the outgoing edges. The behavior is invoked without input for control tokens. The output of the behavior is available to the guard. Because the behavior is used during the process of offering tokens to outgoing edges, it may be run many times on the same token before the token is accepted by those edges. This means the behavior cannot have side effects. It may not modify objects, but it may for example, navigate from one object to another or get an attribute value from an object.

If there is a decision input flow, but no decision input behavior, then it is the tokens offered on the decision input flow that are made available to the guard on each outgoing edge to determine whether the offer on the regular incoming edge is passed along that outgoing edge. If there is a decision input behavior and a decision input flow, the token offered on the decision input flow is passed to the behavior (as the only argument if the regular incoming edge is control flow, as the second argument if it is an object flow). Decision nodes with the additional decision input flow offer tokens to outgoing edges only when one token is offered on each incoming edge.

Notation

The notation for a decision node is a diamond-shaped symbol, as illustrated on the left side of the figure below. Decision input behavior is specified by the keyword «decisionInput» placed in a note symbol, and attached to the appropriate decision node symbol as illustrated in Figure 12.76. A decision input flow is specified by the keyword «decisionInputFlow» annotating that flow.

A decision node must have one non-decision input activity edge entering it and one or more edges leaving it. It may also have a second decision input flow entering it, which must be marked «decisionInputFlow» as shown at the bottom of Figure 12.76. The functionality of a decision node and a merge node can be combined by using the same node symbol, as illustrated at the right side of the figure below. At most one of the incoming flows may be annotated as a decision input flow. This case maps to a model containing a merge node with all the incoming edges shown in the diagram, except a decision input flow, and one outgoing edge to a decision node that has any decision input flow and all the outgoing edges shown in the diagram. This assumes the UML 2 Diagram Interchange standard support of the interchange of diagram elements and their mapping to model elements.

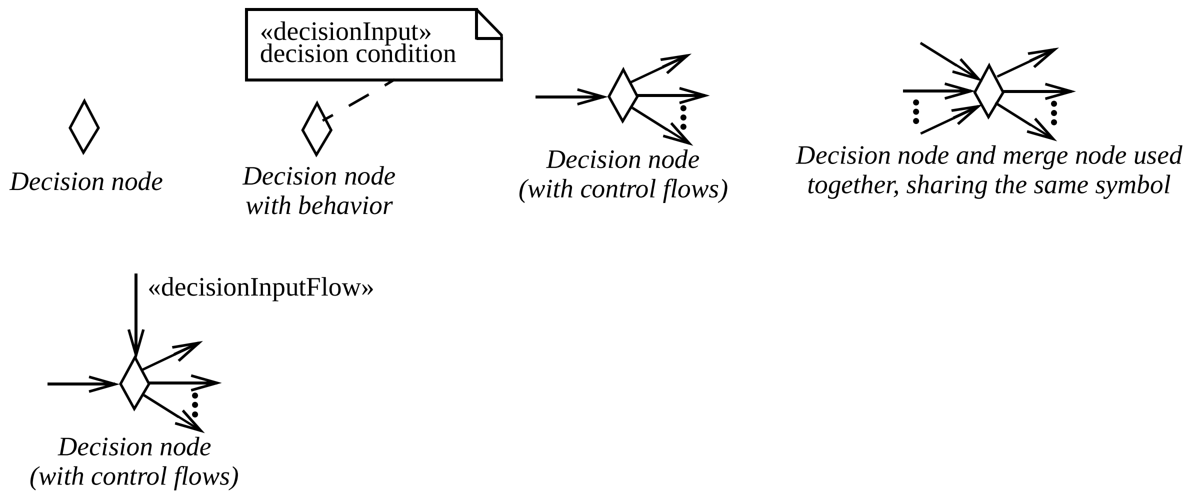


Figure 12.76 - Decision node notation

Examples

The figure below contains a decision node that follows the Received Order behavior. The branching is based on whether order was rejected or accepted. An order accepted condition results in passing control to Fill Order and rejected orders to Close Order.

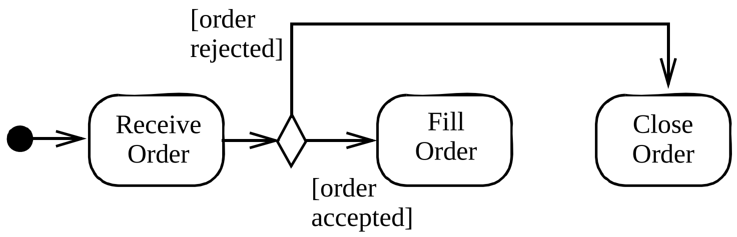


Figure 12.77 - Decision node example

The example in the figure below illustrates an order process example. Here, an order item is pulled from stock and prepared for delivery. Since 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.

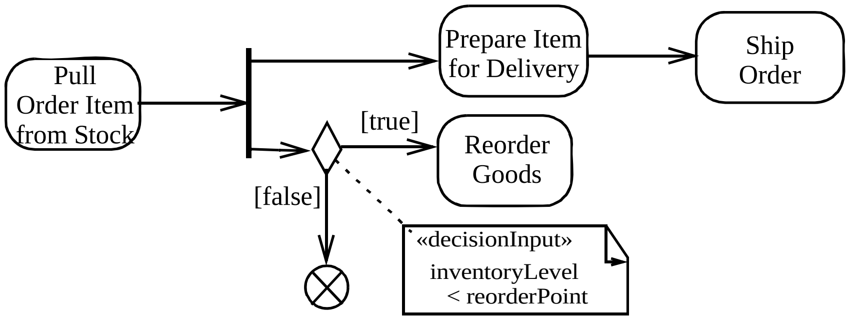

Pull Order Item from Stock
Prepare Item for Delivery
Ship Order
[true]
Reorder Goods
[false]
«decisionInput»
inventoryLevel < reorderPoint

Figure 12.78 - Decision node example

Rationale

Decision nodes are introduced to support conditionals in activities. Decision input behaviors are introduced to avoid redundant recalculations in guards.

Changes from previous UML

Decision nodes replace the use of PseudoState with junction kind in UML 1.5 activity modeling.

12.3.23 ExceptionHandler (from ExtraStructuredActivities)

Generalizations

- "Element (from Kernel)" on page 64

Description

An exception handler is an element that specifies a body to execute in case the specified exception occurs during the execution of the protected node.

Associations

- protectedNode : ExecutableNode [1..1]
  The node protected by the handler. The handler is examined if an exception propagates to the outside of the node. {Subsets *Element::owner*}
- handlerBody : ExecutableNode [1..1]
  A node that is executed if the handler satisfies an uncaught exception.
- exceptionType : Classsifier [1..*]
  The kind of instances that the handler catches. If an exception occurs whose type is any of the classifiers in the set, the handler catches the exception and executes its body.
- exceptionInput : ObjectNode [1..1]
  An object node within the handler body. When the handler catches an exception, the exception token is placed in this node, causing the body to execute.

Constraints

[1] The exception handler and its input object node are not the source or target of any edge.

[2] An edge that has a source in an exception handler structured node must also have its target in the handler, and vice versa.

[3] The result pins of the exception handler body must correspond in number and types to the result pins of the protected node.

[4] The handler body has one input, and that input is the same as the exception input.

Semantics

If a RaiseExceptionAction is executed, all the tokens in the immediately containing structured node or activity are terminated. Then the set of execution handlers on the structured node or invocation action of the activity is examined for a handler that matches the exception. A handler matches if the type of the exception is the same as or a descendant of one of the exception classifiers specified in 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. The exception object is placed in the exceptionInput node as a token to start execution of the handler body.

If the exception is not caught by any of the handlers on the node or invocation action, the exception handling process repeats, propagating to the enclosing structured node or activity. If the exception is not caught there, and the action that invoked the activity is asynchronous, the exception is lost, because the connection to the invoker is broken. If the action that invoked the activity is synchronous, the exception propagates up to that action. The process of exception propagation recurs until the exception is caught, or reaches the topmost level of the system. If the exception propagates to the topmost level of the system and is not caught, the behavior of the system is unspecified. Profiles may specify what happens in such cases.

The handler body has no explicit input or output edges. It has the same access to its surrounding context as the protected node. The result tokens of the handler body become the result tokens of the protected node. Any control edges leaving the protected node receive control tokens on completion of execution of the handler body with the handler catching the exception. When the handler body completes execution, it is as if the protected node had completed execution.

When an expansion region is complete, tokens in the input expansion node and pins are removed.

Notation

The notation for exception handlers is illustrated in Figure 12.79. An exception handler for a protected node is shown by drawing a "lightning bolt" symbol from the boundary of the protected node to a small square on the boundary of the exception handler. The name of the exception type is placed next to the lightning bolt. The small square is the exception input node, and it must be owned by the handler body. Its type is the given exception type. Both the protected node and the exception handler must be at the same nesting level. (Otherwise the notation could be misinterpreted as an interrupting edge, which crosses a boundary.) Multiple exception handlers may be attached to the same protected node, each by its own lightning bolt.

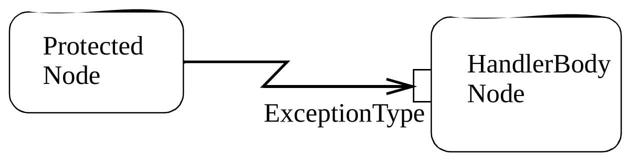


Figure 12.79 - Exception Handler Notation

Presentation Options

An option for notating an exception handler is a zig-zag adornment on a straight line.

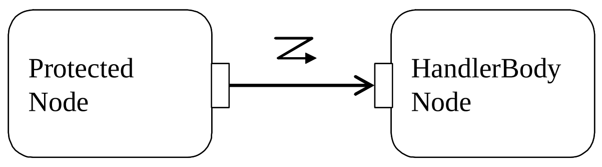


Figure 12.80 - Exception Handler Presentation option

Examples

Figure 12.81 shows a matrix calculation. First a matrix is inverted, then it is multiplied by a vector to produce a vector. If the matrix is singular, the inversion will fail and a SingularMatrix exception occurs. This exception is handled by the exception handler labeled SingularMatrix, which executes the region containing the SubstituteVector1 action. If an overflow exception occurs during either the matrix inversion or the vector multiplication, the region containing the SubstituteVector2 action is executed.

The successors to an exception handler body are the same as the successors to the protected node. It is unnecessary to show control flow from the handler body. Regardless of whether the matrix operations complete without exception or whether one of the exception handlers is triggered, the action PrintResults is executed next.

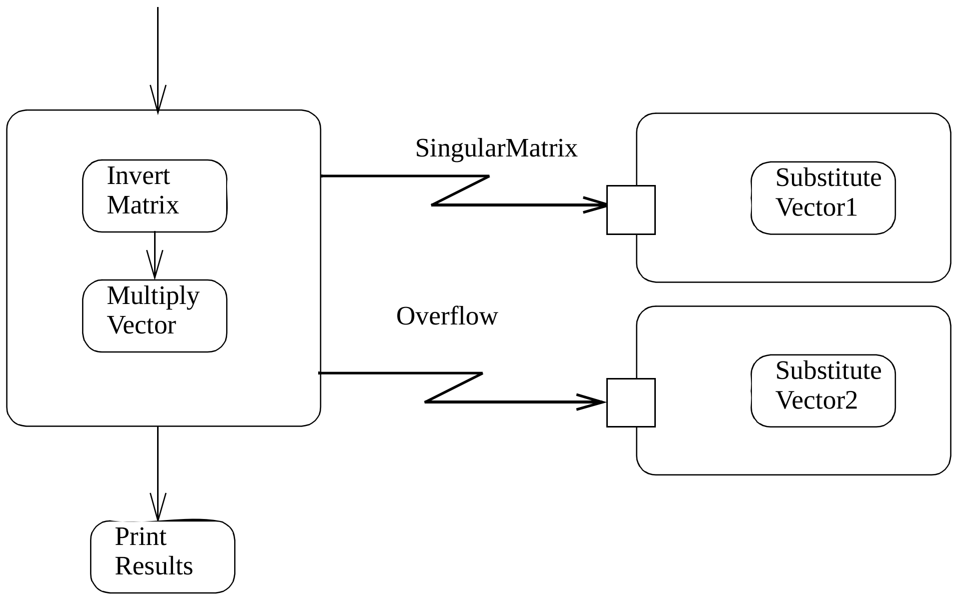


Figure 12.81 - Exception Handler example

Changes from previous UML

ExceptionHandler replaces JumpHandler in UML 1.5.

Modeling of traditional break and continue statements can be accomplished using direct control flow from the statement to the control target. UML 1.5 combined the modeling of breaks and continues with exceptions, but that is no longer necessary and it is not recommended in this specification.

12.3.24 ExecutableNode (from ExtraStructuredActivities, StructuredActivities)

Generalizations

- "ActivityNode (from BasicActivities, CompleteActivities, FundamentalActivities, IntermediateActivities, CompleteStructuredActivities)" on page 333.

Description

An executable node is an abstract class for activity nodes that may be executed. It is used as an attachment point for exception handlers.

Associations

Package ExtraStructuredActivities

- handler : ExceptionHandler [0..*]
  A set of exception handlers that are examined if an uncaught exception propagates to the outer level of the executable node. {Subsets *Element::ownedElement*}

12.3.25 ExpansionKind (from ExtraStructuredActivities)

Generalizations

None

Description

ExpansionKind is an enumeration type used to specify how multiple executions of an expansion region interact. See "ExpansionRegion (from ExtraStructuredActivities)."

Enumeration Literals

- parallel
  The executions are independent. They may be executed concurrently.
- iterative
  The executions are dependent and must be executed one at a time, in order of the collection elements.
- stream
  A stream of collection elements flows into a single execution, in order of the collection elements.

12.3.26 ExpansionNode (from ExtraStructuredActivities)

Generalizations

- "ObjectNode (from BasicActivities, CompleteActivities)" on page 393

Description

An expansion node is an object node used to indicate a flow across the boundary of an expansion region. A flow into a region contains a collection that is broken into its individual elements inside the region, which is executed once per element. A flow out of a region combines individual elements into a collection for use outside the region.

Associations

- regionAsInput : ExpansionRegion[0..1]
  The expansion region for which the node is an input.
- regionAsOutput : ExpansionRegion[0..1]
  The expansion region for which the node is an output.

Semantics

See "ExpansionRegion (from ExtraStructuredActivities)."

Notation

See "ExpansionRegion (from ExtraStructuredActivities)."

12.3.27 ExpansionRegion (from ExtraStructuredActivities)

An expansion region is a structured activity region that executes multiple times corresponding to elements of an input collection.

Generalizations

- "StructuredActivityNode (from CompleteStructuredActivities, StructuredActivities)" on page 409

Description

An expansion region is a strictly nested region of an activity with explicit input and outputs (modeled as ExpansionNodes). Each input is a collection of values. If there are multiple inputs, each of them must hold the same kind of collection, although the types of the elements in the different collections may vary. The expansion region is executed once for each element (or position) in the input collection.

The number of output collections can differ from the number of input collections. On each execution of the region, an output value from the region is inserted into an output collection at the same position as the input elements. If the region execution ends with no output, then nothing is added to the output collection. When this happens the output collection will not have the same number of elements as the input collections, the region acts as a filter. If all the executions provide an output to the collection, then the output collections will have the same number of elements as the input collections.

The inputs and outputs to an expansion region are modeled as ExpansionNodes. 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. Object flow edges connect pins outside the region to input and output expansion nodes as collections. Object flow edges connect pins inside the region to input and output expansion nodes as individual elements. From the inside of the region, these nodes are visible as individual values. If an expansion node has a name, it is the name of the individual element within the region.

Any object flow edges that cross the boundary of the region, without passing through expansion nodes, provide values that are fixed within the different executions of the region Input pins, introduced by merge with CompleteStructuredActivities, provide values that are also constant during the execution of the region.

Attributes

- mode : ExpansionKind
  The way in which the executions interact (default value is *iterative*):
  parallel - all interactions are independent
  iterative - the interactions occur in order of the elements
  stream - a stream of values flows into a single execution

Associations

- inputElement : ExpansionNode[1..*]
  An object node that holds a separate element of the input collection during each of the multiple executions of the region.

- outputElement : ExpansionNode[0..*]
  An object node that accepts a separate element of the output collection during each of the multiple executions of the region. The values are formed into a collection that is available when the execution of the region is complete.

Constraints

[1] An ExpansionRegion must have one or more argument ExpansionNodes and zero or more result ExpansionNodes.

Semantics

When an execution of an activity makes a token available to the input of an expansion region, the expansion region consumes the token and begins execution. The expansion region is executed once for each element in the collection. If there are multiple inputs, a value is taken from each for each execution of the internals of the region. The mode attribute controls how the executions proceed:

- If the value is *parallel*, the execution may happen in parallel, or overlapping in time, but they are not required to.
- If the value is *iterative*, the executions of the region must happen in sequence, with one finishing before another can begin. The first iteration begins immediately. Subsequent iterations start when the previous iteration is completed. During each of these cases, one element of the collection is made available to the execution of the region as a token during each execution of the region. If the collection is ordered, the elements will be presented to the region in order; if the collection is unordered, the order of presenting elements is undefined and not necessarily repeatable. On each execution of the region, an output value from the region is inserted into an output collection at the same position as the input elements.
- If the value is *stream*, there is a single execution of the region, but its input place receives a stream of elements from the collection. The values in the input collection are extracted and placed into the execution of the expansion region as a stream in order, if the collection is ordered. Such a region must handle streams properly or it is ill defined. When the execution of the entire stream is complete, any output streams are assembled into collections of the same kinds as the inputs.

Notation

An expansion region is shown as a dashed rounded box with one of the keywords *parallel*, *iterative*, or *streaming* in the upper left corner.

Input and output expansion nodes are drawn as small rectangles divided by vertical bars into small compartments. (The symbol is meant to suggest a list of elements.) The expansion node symbols are placed on the boundary of the dashed box. Usually arrows inside and outside the expansion region will distinguish input and output expansion nodes. If not, then a small arrow can be used as with Pins (see Figure 12.124 on page 403).

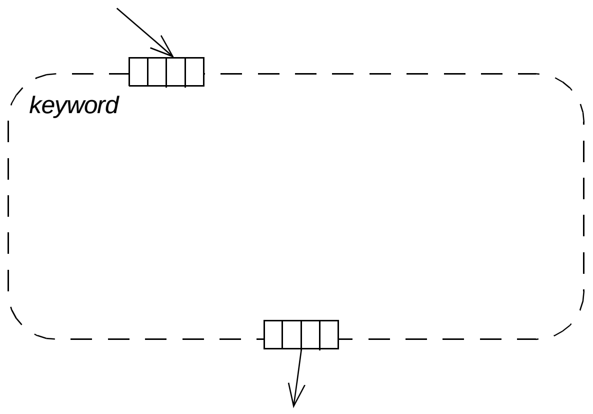


Figure 12.82 - Expansion region

As a shorthand notation, the “list box pin” notation may be placed directly on an action symbol, replacing the pins of the action (Figure 12.83). This indicates an expansion region containing a single action. The equivalent full form is shown in Figure 12.84.

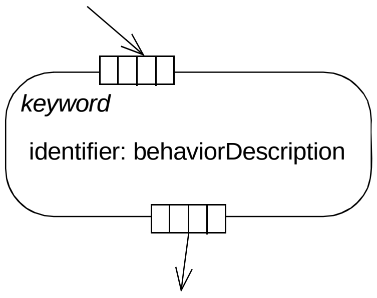


Figure 12.83 - Shorthand notation for expansion region containing single node

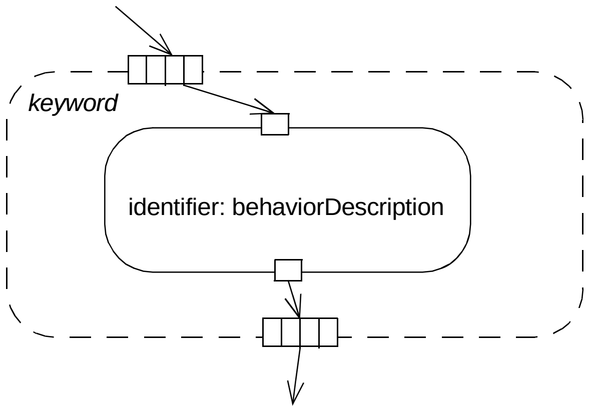


Figure 12.84 - Full form of previous shorthand notation

Presentation Options

The notation in Figure 12.85 maps to an expansion region in parallel mode, with one behavior invoked in the region, as shown below.

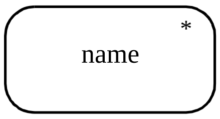


Figure 12.85 - Notation for expansion region with one behavior invocation

Examples

Figure 12.86 shows an expansion region with two inputs and one output that is executed in parallel. Execution of the region does not begin until both input collections are available. Both collections must have the same number of elements. The interior activity fragment is executed once for each position in the input collections. During each execution of the region, a pair of values, one from each collection, is available to the region on the expansion nodes. Each execution of the region produces a result value on the output expansion node. 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 region on the result node after all the parallel executions of the region have completed.

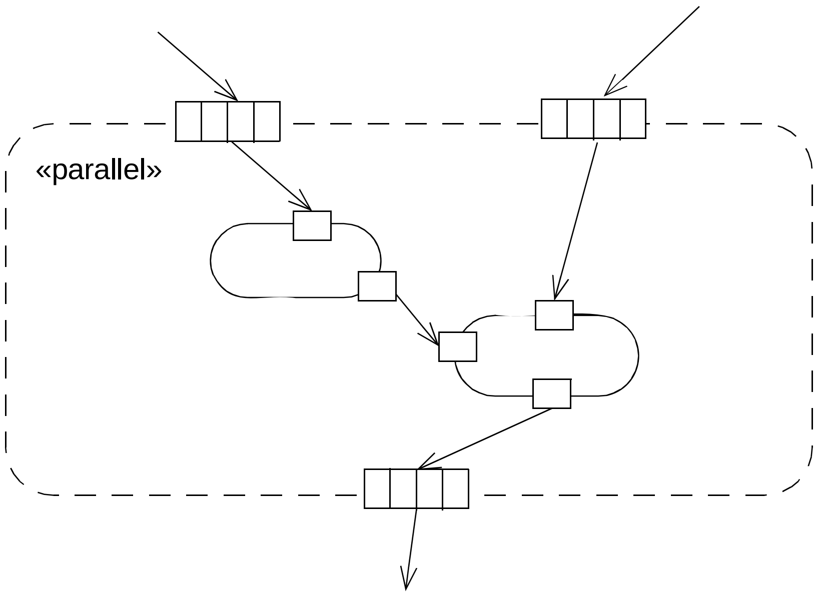


Figure 12.86 - Expansion region with 2 inputs and 1 output

Figure 12.86 shows a fragment of a Fast Fourier Transform (FFT) computation containing an expansion region. Outside the region, 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 3 input arrays, yielding 2 output arrays. Different positions in the arrays do not interact, therefore the region can be executed in parallel on all positions.

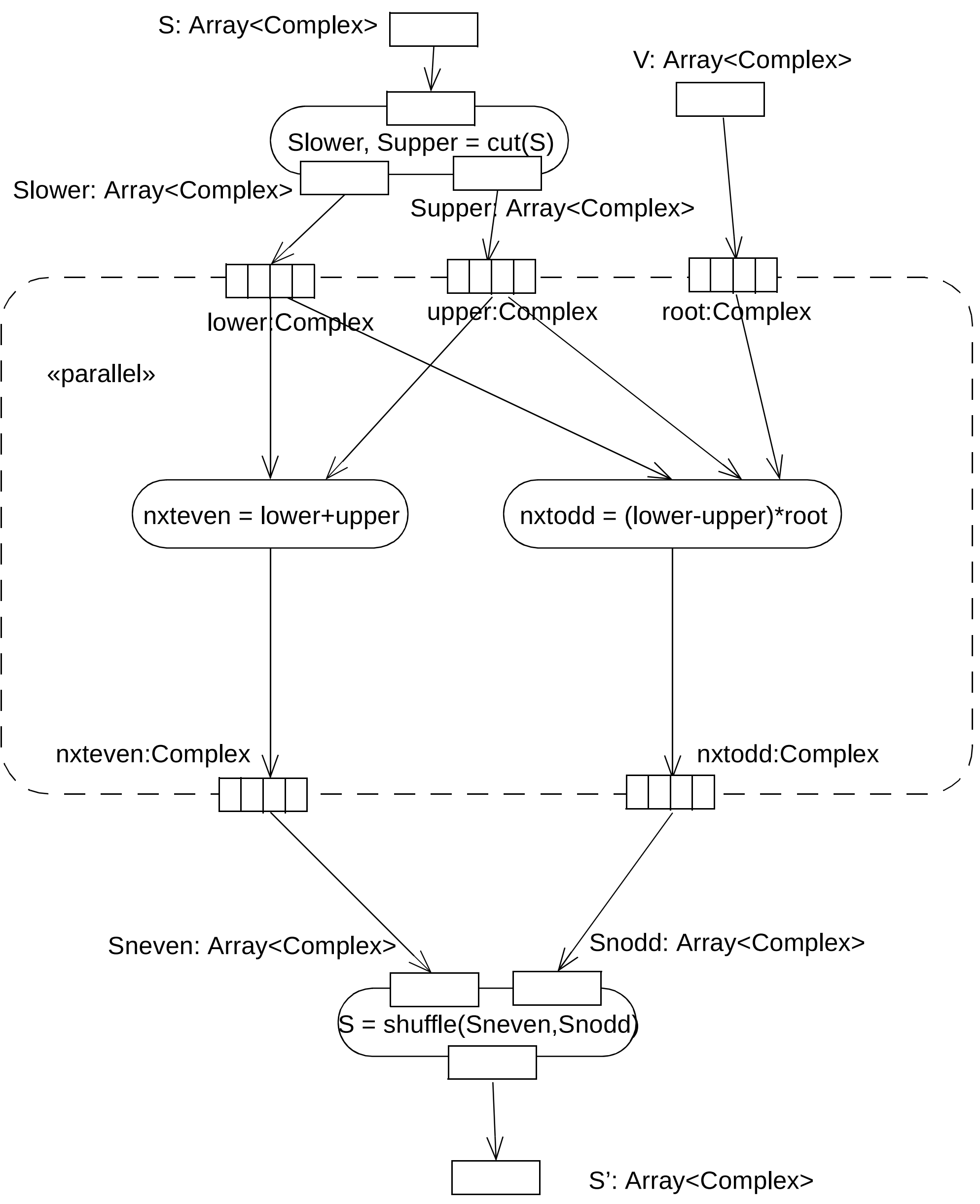


Figure 12.87 - Expansion region

The following example shows a use of the shorthand notation for an expansion region with a single action. In this example, the trip route 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.

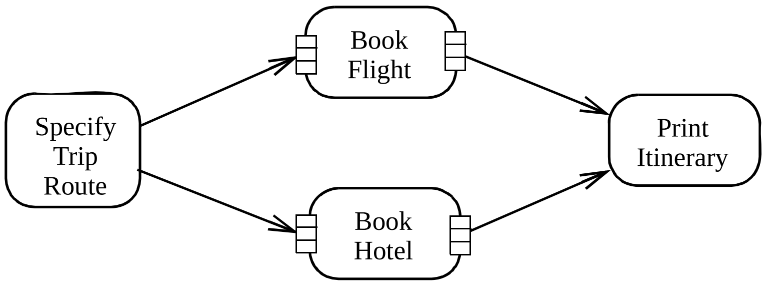


Figure 12.88 -Examples of expansion region shorthand

Specify Trip Route below 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 BookFlight.

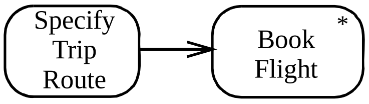


Figure 12.89 - Shorthand notation for expansion region

Rationale

Expansion regions are introduced to support applying behaviors to elements of a set without constraining the order of application.

Changes from previous UML

ExpansionRegion replaces MapAction, FilterAction, and dynamicConcurrency and dynamicMultiplicity attributes on ActionState. Dynamic multiplicities less than unlimited are not supported in UML 1.5.

12.3.28 FinalNode (from IntermediateActivities)

A final node is an abstract control node at which a flow in an activity stops.

Generalizations

- “ControlNode (from BasicActivities)” on page 356

Description

See descriptions at children of final node.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] A final node has no outgoing edges.

Semantics

All tokens offered on incoming edges are accepted. See children of final node for other semantics.

Notation

The notations for final node are illustrated below. There are two kinds of final node: activity final and (IntermediateActivities) flow final. For more details on each of these specializations, see ActivityFinal and FlowFinal.

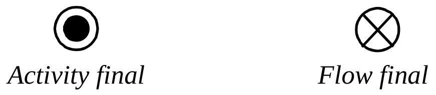


Figure 12.90 - Final node notation

Examples

The figure below illustrates two kinds of final node: flow final and activity final. In this example, it is assumed that many components can be built and installed before finally delivering the resulting application. Here, the Build Component behavior occurs iteratively for each component. When the last component is built, the end of the building iteration is indicated with a flow final. However, even though all component building has come to an end, other behaviors are still executing. When the last component has been installed, the application is delivered. When Deliver Application has completed, control is passed to an activity final node—indicating that all processing in the activity is terminated.

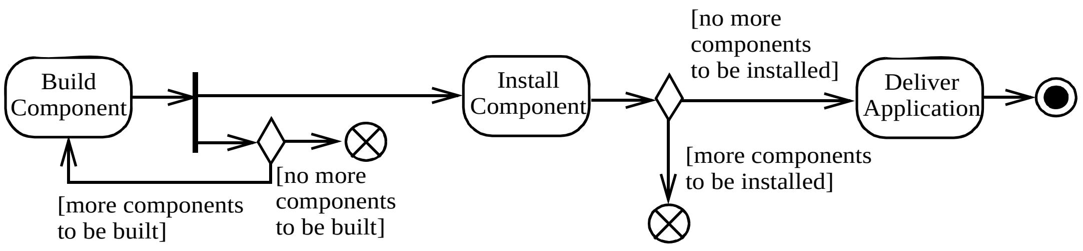

Build Component
Install Component
Deliver Application
[no more components to be installed]
[more components to be installed]
[more components to be built]
[no more components to be built]

Figure 12.91 - Flow final and activity final example

Rationale

Final nodes are introduced to model where flows end in an activity.

Changes from previous UML

FinalNode replaces the use of FinalState in UML 1.5 activity modeling, but its concrete classes have different semantics than FinalState.

12.3.29 FlowFinalNode (from IntermediateActivities)

A flow final node is a final node that terminates a flow.

Generalizations

- "FinalNode (from IntermediateActivities)" on page 373

Description

A flow final destroys all tokens that arrive at it. It has no effect on other flows in the activity.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Flow final destroys tokens flowing into it.

Notation

The notation for flow final is illustrated below.

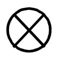


Figure 12.92 - Flow final notation

Examples

In the example below, it is assumed that many components can be built and installed. Here, the Build Component behavior occurs iteratively for each component. When the last component is built, the end of the building iteration is indicated with a flow final. However, even though all component building has come to an end, other behaviors are still executing (such as Install Component).

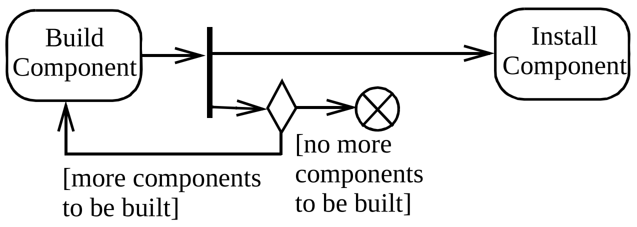

Build Component
Install Component
[no more components to be built]
[more components to be built]

Figure 12.93 - Flow final example without merge edge

Rationale

Flow final nodes are introduced to model termination of a flow in an activity.

Changes from previous UML

Flow final is new in UML 2.0.

12.3.30 ForkNode (from IntermediateActivities)

A fork node is a control node that splits a flow into multiple concurrent flows.

Generalizations

- “ControlNode (from BasicActivities)” on page 356

Description

A fork node has one incoming edge and multiple outgoing edges.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] A fork node has one incoming edge.

[2] The edges coming into and out of a fork node must be either all object flows or all control flows.

Semantics

Tokens arriving at a fork are duplicated across the outgoing edges. If at least one outgoing edge accepts the token, duplicates of the token are made and one copy traverses each edge that accepts the token. The outgoing edges that did not accept the token due to failure of their targets to accept it, keep their copy in an implicit FIFO queue until it can be accepted by the target. The rest of the outgoing edges do not receive a token (these are the ones with failing guards). This is an exception to the rule that control nodes cannot hold tokens if they are blocked from moving downstream (see

"Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)" on page 315). When an offered token is accepted on all the outgoing edges, duplicates of the token are made and one copy traverses each edge. No duplication is necessary if there is only one outgoing edge, but it is not a useful case.

If guards are used on edges outgoing from forks, the modelers should ensure that no downstream joins depend on the arrival of tokens passing through the guarded edge. If that cannot be avoided, then a decision node should be introduced to have the guard, and shunt the token to the downstream join if the guard fails. See example in Figure 12.44 on page 329.

Notation

The notation for a fork node is simply a line segment, as illustrated on the left side of the figure below. In usage, however, the fork node must have a single activity edge entering it, and two or more edges leaving it. The functionality of join node and fork node can be combined by using the same node symbol, as illustrated at the right side of the figure below. This case maps to a model containing a join node with all the incoming edges shown in the diagram and one outgoing edge to a fork node that has all the outgoing edges shown in the diagram. It assumes the UML 2.0 Diagram Interchange RFP supports the interchange of diagram elements and their mapping to model elements.

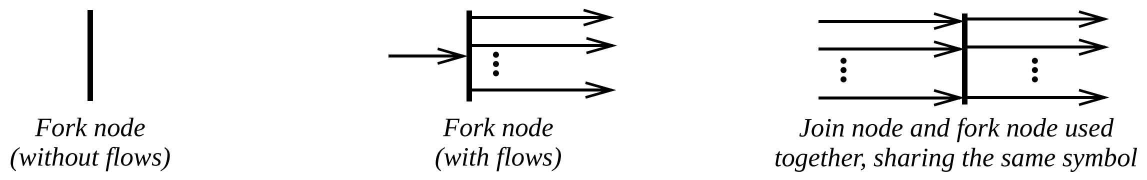

Fork node
(without flows)
Fork node
(with flows)
Join node and fork node used
together, sharing the same symbol

Figure 12.94 - Fork node notation

Examples

In the example below, the fork node passes control to both the Ship Order and Send Invoice behaviors when Fill Order is completed.

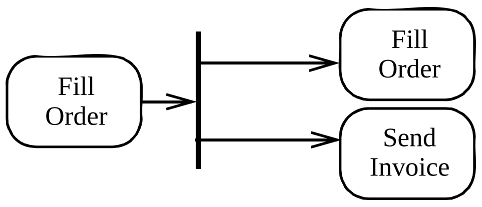


Figure 12.95 - Fork node example

Rationale

Fork nodes are introduced to support parallelism in activities.

Changes from previous UML

Fork nodes replace the use of PseudoState with fork kind in UML 1.5 activity modeling. State machine forks in UML 1.5 required synchronization between parallel flows through the state machine RTC step. UML 2.0 activity forks model unrestricted parallelism.

12.3.31 InitialNode (from BasicActivities)

An initial node is a control node at which flow starts when the activity is invoked.

Generalizations

- “ControlNode (from BasicActivities)” on page 356

Description

An activity may have more than one initial node.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] An initial node has no incoming edges.

[2] Only control edges can have initial nodes as source.

Semantics

An initial node is a starting point for executing an activity (or structured node, see “StructuredActivityNode (from CompleteStructuredActivities, StructuredActivities)” on page 409). A control token is placed at the initial node when the activity starts, but not in initial nodes in structured nodes contained by the activity. Tokens in an initial node are offered to all outgoing edges. If an activity has more than one initial node, then invoking the activity starts multiple flows, one at each initial node. For convenience, initial nodes are an exception to the rule that control nodes cannot hold tokens if they are blocked from moving downstream, for example, by guards (see Activity). This is equivalent to interposing a CentralBufferNode between the initial node and its outgoing edges.

Note that flows can also start at other nodes, see ActivityParameterNode and AcceptEventAction, so initial nodes are not required for an activity to start execution. In addition, when an activity starts, a control token is placed at each action or structured node that has no incoming edges, except if it is a handler body (see “ExceptionHandler (from ExtraStructuredActivities)” on page 363, it is the fromAction of an action input pin (see “ActionInputPin (as specialized)” on page 315), or it is contained in a structured node.

Notation

Initial nodes are notated as a solid circle, as indicated in the figure below.

Figure 12.96 - Initial node notation

Examples

In the example below, the initial node passes control to the Receive Order behavior at the start of an activity.

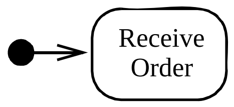


Figure 12.97 - Initial node example

Rationale

Initial nodes are introduced to model where flows start in an activity.

Changes from previous UML

InitialNode replaces the use of PseudoState with kind initial in UML 1.5 activity modeling.

12.3.32 InputPin (as specialized)

Input pins are object nodes that receive values from other actions through object flows. See Pin, Action, and ObjectNode for more details.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] Input pins may have outgoing edges only when they are on actions that are structured nodes, and these edges must target a node contained by the structured node.

Semantics

See "InputPin (from BasicActions)" on page 255

12.3.33 InterruptibleActivityRegion (from CompleteActivities)

An interruptible activity region is an activity group that supports termination of tokens flowing in the portions of an activity.

Generalizations

- "ActivityGroup (from BasicActivities, FundamentalActivities)" on page 332

Description

An interruptible region contains activity nodes. When a token leaves an interruptible region via edges designated by the region as interrupting edges, all tokens and behaviors in the region are terminated.

Attributes

No additional attributes

Associations

- interruptingEdge : ActivityEdge [0..*]
  The edges leaving the region that will abort other tokens flowing in the region.
- node : ActivityNode [0..*]
  Nodes directly contained in the region. {Subsets *ActivityGroup::containedNode*}

Constraints

[1] Interrupting edges of a region must have their source node in the region and their target node outside the region in the same activity containing the region.

Semantics

The region is interrupted, including accept event actions in the region, when a token traverses an interrupting edge. At this point the interrupting token has left the region and is not terminated. 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 accept event action.

Token transfer is still atomic, even when using interrupting regions. If a non-interrupting edge is passing a token from a source node in the region to target node outside the region, then the transfer is completed and the token arrives at the target even if an interruption occurs during the traversal. In other words, a token transition is never partial; it is either complete or it does not happen at all.

Do not use an interrupting region if it is not desired to abort all flows in the region in some cases. For example, if the same execution of an activity is being used for all its invocations, then multiple streams of tokens will be flowing through the same activity. In this case, it is probably not desired to abort all tokens just because one leaves the region. Arrange for separate invocations of the activity to use separate executions of the activity when employing interruptible regions, so tokens from each invocation will not affect each other.

Notation

An interruptible activity region is notated by a dashed, round-cornered rectangle drawn around the nodes contained by the region. An interrupting edge is notation with a lightning-bolt activity edge.

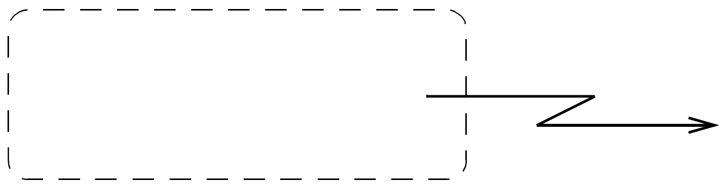


Figure 12.98 - InterruptibleActivityRegion notation with interrupting edge

Presentation Options

An option for notating an interrupting edge is a zig zag adornment on a straight line.

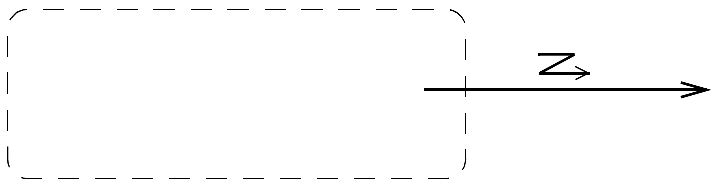


Figure 12.99 - InterruptibleActivityRegion notation with interrupting edge

Examples

The first figure below illustrates that when an order cancellation request is made—only while receiving, filling, or shipping) orders—the Cancel Order behavior is invoked.

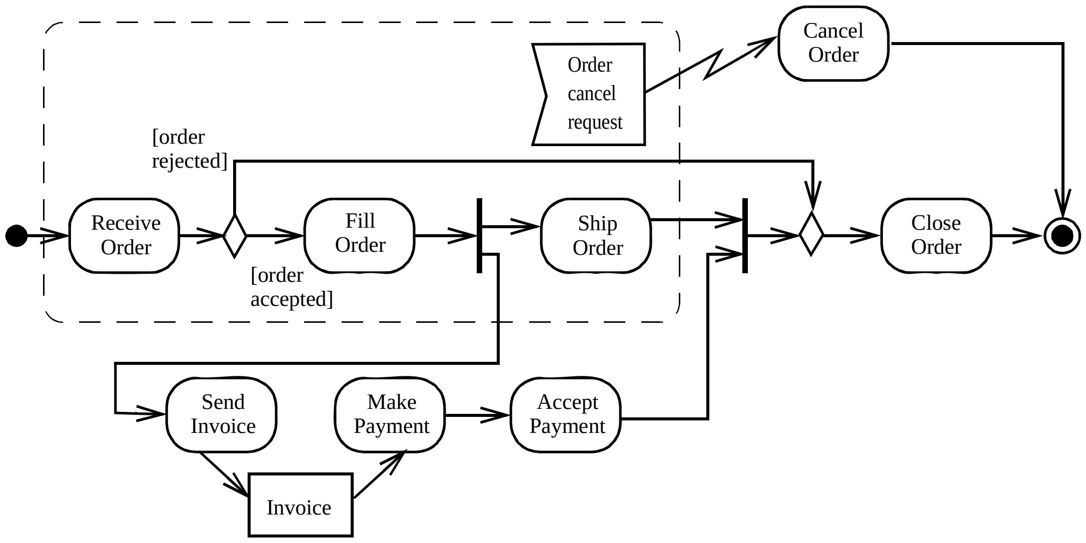


Figure 12.100 - InterruptibleActivityRegion example

Rationale

Interruptible regions are introduced to support more flexible non-local termination of flow.

Changes from previous UML

Interruptible regions in activity modeling are new to UML 2.0.

12.3.34 JoinNode (from CompleteActivities, IntermediateActivities)

A join node is a control node that synchronizes multiple flows.

Generalizations

- "ControlNode (from BasicActivities)" on page 356

Description

A join node has multiple incoming edges and one outgoing edge.

Package CompleteActivities

Join nodes have a Boolean value specification using the names of the incoming edges to specify the conditions under which the join will emit a token.

Attributes

Package CompleteActivities

- isCombineDuplicate : Boolean [1..1]
  Tells whether tokens having objects with the same identity are combined into one by the join. Default value is true.

Associations

Package CompleteActivities

- joinSpec : ValueSpecification [1..1]
  A specification giving the conditions under which the join will emit a token.Default is "and." {Subsets *Element::ownedElement*}

Constraints

[1] A join node has one outgoing edge.

```
self.outgoing->size() = 1
```

[2] If a join node has an incoming object flow, it must have an outgoing object flow, otherwise, it must have an outgoing control flow.

```
(self.incoming.select( e | e.isTypeOf(ObjectFlow)->notEmpty() implies self.outgoing.isTypeOf(ObjectFlow)) and
(self.incoming.select( e | e.isTypeOf(ObjectFlow)->empty() implies self.outgoing.isTypeOf(ControlFlow))
```

Semantics

If there is a token offered on all incoming edges, then tokens are offered on the outgoing edge according to the following join 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 data tokens, then only the data tokens are offered on the outgoing edge. Tokens are offered on the outgoing edge in the same order they were offered to the join.

Multiple control tokens offered on the same incoming edge are combined into one before applying the above rules. No joining of tokens is necessary if there is only one incoming edge, but it is not a useful case.

Package CompleteActivities

The reserved string “and” used as a join specification is equivalent to a specification that requires at least one token offered on each incoming edge. It is the default. The join specification is evaluated whenever a new token is offered on any incoming edge. The evaluation is not interrupted by any new tokens offered during the evaluation, nor are concurrent evaluations started when new tokens are offered during an evaluation.

If any tokens are offered to the outgoing edge, they must be accepted or rejected for traversal before any more tokens are offered to the outgoing edge. If tokens are rejected for traversal, they are no longer offered to the outgoing edge. The join specification may contain the names of the incoming edges to refer to whether a token was offered on that edge at the time the evaluation started.

If isCombinedDuplicate is true, then before object tokens are offered to the outgoing edge, those containing objects with the same identity are combined into one token.

Other rules for when tokens may be passed along the outgoing edge depend on the characteristics of the edge and its target. For example, if the outgoing edge targets an object node that has reached its upper bound, no token can be passed. The rules may be optimized to a different algorithm as long as the effect is the same. In the full object node example, the implementation can omit the unnecessary join evaluations until the down stream object node can accept tokens.

Notation

The notation for a join node is a line segment, as illustrated on the left side of the figure below. The join node must have one or more activity edges entering it, and only one edge leaving it. The functionality of join node and fork node can be combined by using the same node symbol, as illustrated at the right side of the figure below. This case maps to a model containing a join node with all the incoming edges shown in the diagram and one outgoing edge to a fork node that has all the outgoing edges shown in the diagram. It assumes the UML 2.0 Diagram Interchange specification supports the interchange of diagram elements and their mapping to model elements.

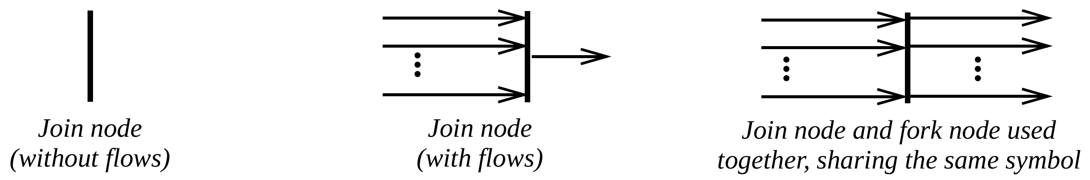

Join node
(without flows)
Join node
(with flows)
Join node and fork node used
together, sharing the same symbol

Figure 12.101 - Join node notations

Package CompleteActivities

Join specifications are shown near the join node, as shown below.

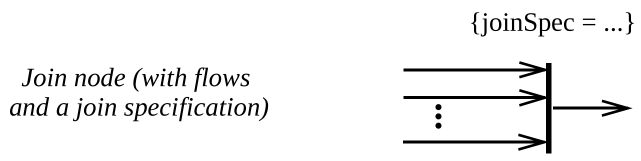


Figure 12.102 - Join node notations

Examples

The example at the left of the figure indicates that a Join is used to synchronize the processing of the Ship Order and Send Invoice behaviors. Here, when both have been completed, control is passed to Close Order.

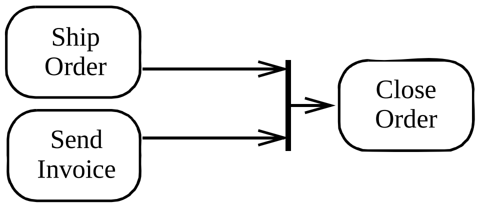


Figure 12.103 - Join node example

Package CompleteActivities

The example below illustrates how a join specification 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.

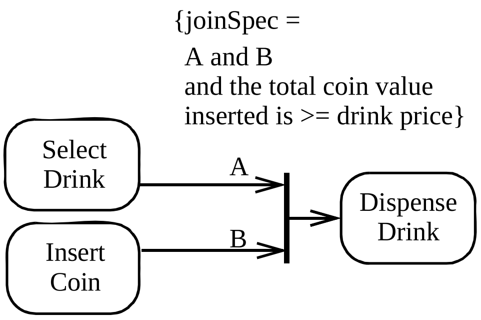


Figure 12.104 - Join node example

Rationale

Join nodes are introduced to support parallelism in activities.

Changes from previous UML

Join nodes replace the use of PseudoState with join kind in UML 1.5 activity modeling.

12.3.35 LoopNode (from CompleteStructuredActivities, StructuredActivities)

A loop node is a structured activity node that represents a loop with setup, test, and body sections.

Generalizations

- "StructuredActivityNode (from CompleteStructuredActivities, StructuredActivities)" on page 409.

Description

Each section is a well-nested sub region of the activity whose nodes follow any predecessors of the loop and precede any successors of the loop. The test section may precede or follow the body section. The setup section is executed once on entry to the loop, and the test and body sections are executed repeatedly until the test produces a false value. The results of the final execution of the test or body are available after completion of execution of the loop.

Attributes

- isTestedFirst : Boolean [1]
  If true, the test is performed before the first execution of the body. If false, the body is executed once before the test is performed. Default value is *false*.

Associations

Package StructuredActivities

- setupPart : ExecutableNode[0..*]
  The set of nodes and edges that initialize values or perform other setup computations for the loop.
- bodyPart : ExecutableNode[0..*]
  The set of nodes and edges that perform the repetitive computations of the loop. The body section is executed as long as the test section produces a true value.
- test : ExecutableNode[0..*]
  The set of nodes, edges, and designated value that compute a Boolean value to determine if another execution of the body will be performed.
- decider : OutputPin [1]
  An output pin within the test fragment the value of which is examined after execution of the test to determine whether to execute the loop body.

Package CompleteStructuredActivities

- result : OutputPin [0..*] {ordered}
  A list of output pins that constitute the data flow output of the entire loop. {Subsets *Action::output*}
- loopVariable : OutputPin [0..*] {ordered}
  A list of output pins 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 pins of the loop.
- bodyOutput : OutputPin [0..*] {ordered}
  A list of output pins within the body fragment the values of which are moved to the loop variable pins after completion of execution of the body, before the next iteration of the loop begins or before the loop exits.
- loopVariableInput : InputPin[0..*] {ordered}
  A list of values that are moved into the loop variable pins before the first iteration of the loop. {Subsets *Action::input*}

Constraints

Package CompleteStructuredActivities

[1] Loop variable inputs must not have outgoing edges.

[2] The bodyOutput pins are output pins on actions in the body of the loop node.

[3] The result output pins have no incoming edges.

Semantics

No part of a loop node is executed until all control-flow or data-flow predecessors of the loop node have completed execution. When all such predecessors have completed execution and made tokens available to inputs of the loop node, the loop node captures the input tokens and begins execution.

First the setup section of the loop node is executed. A *front end node* is a node within a nested section (such as the setup section, test section, or body section) that has no predecessor dependencies within the same section. A control token is offered to each front end node within the setup section. Nodes in the setup section may also have individual dependencies (typically data flow dependencies) on nodes external to the loop node. To begin execution, such nodes must receive their individual tokens in addition to the control token from the overall loop.

A *back end node* is a node within a nested section that has no successor dependencies within the same section. When all the back end nodes have completed execution, the overall section is considered to have completed execution. (It may be thought of as delivering a control token to the next section within the loop.)

When the setup section has completed execution, the iterative execution of the loop begins. The test section may precede or follow the body section (test-first loop or test-last loop). The following description assumes that the test section comes first. If the body section comes first, it is always executed at least once, after which this description applies to subsequent iterations.

When the setup section has completed execution (if the test comes first) or when the body section has completed execution of an iteration, the test section is executed. A control token is offered to each front end node within the test section. When all back end nodes in the test section have completed execution, execution of the test section is complete. Typically there will only be one back end node and it will have a Boolean value, but for generality it is permitted to perform arbitrary computation in the test section.

When the test section has completed execution, the Boolean value on the designated *decider* pin within the test section is examined. If the value is true, the body section is executed again. If the value is false, execution of the loop node is complete.

When the setup section has completed execution (if the body comes first) or when the iteration section has completed execution and produced a true value, execution of the body section begins. Each front end node in the body section is offered a control token. When all back end nodes in the body section have completed execution, execution of the body section is complete.

Within the body section, variables defined in the loop node or in some higher-level enclosing node are updated with any new values produced during the iteration and any temporary values are discarded.

Notation

No specific notation.

Rationale

Loop nodes are introduced to provide a structured way to represent iteration.

Changes from previous UML

Loop nodes are new in UML 2.0.

12.3.36 MergeNode (from IntermediateActivities)

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.

Generalizations

- “ControlNode (from BasicActivities)” on page 356

Description

A merge node has multiple incoming edges and a single outgoing edge.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] A merge node has one outgoing edge.

[2] The edges coming into and out of a merge node must be either all object flows or all control flows.

Semantics

All tokens offered on incoming edges are offered to the outgoing edge. There is no synchronization of flows or joining of tokens.

Notation

The notation for a merge node is a diamond-shaped symbol, as illustrated on the left side of the figure below. In usage, however, the merge node must have two or more edges entering it and a single activity edge leaving it. The functionality of merge node and decision node can be combined by using the same node symbol, as illustrated at the right side of the figure below. This case maps to a model containing a merge node with all the incoming edges shown in the diagram and one outgoing edge to a decision node that has all the outgoing edges shown in the diagram. It assumes the UML 2.0 Diagram Interchange specification supports the interchange of diagram elements and their mapping to model elements.

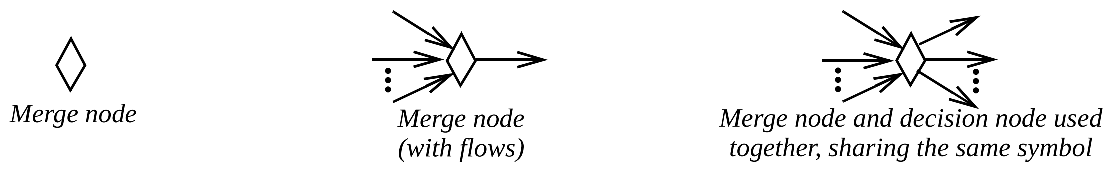

Merge node
Merge node (with flows)
Merge node and decision node used together, sharing the same symbol

Figure 12.105 - Merge node notation

Examples

In the example below, either one or both of the behaviors, Buy Item or Make Item could have been invoked. 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 invoked only once; if both complete, Ship Item is invoked twice.

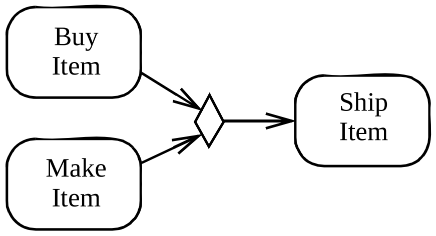


Figure 12.106 - Merge node example

Rationale

Merge nodes are introduced to support bringing multiple flows together in activities. For example, if a decision is used after a fork, the two flows coming out of the decision need to be merged into one before going to a join; otherwise, the join will wait for both flows, only one of which will arrive.

Changes from previous UML

Merge nodes replace the use of PseudoState with junction kind in UML 1.5 activity modeling.

12.3.37 ObjectFlow (from BasicActivities, CompleteActivities)

An object flow is an activity edge that can have objects or data passing along it.

Generalizations

- “ActivityEdge (from BasicActivities, CompleteActivities, CompleteStructuredActivities, IntermediateActivities)” on page 325.

Description

An object flow models the flow of values to or from object nodes.

Package CompleteActivities

Object flows add support for multicast/receive, token selection from object nodes, and transformation of tokens.

Attributes

Package CompleteActivities

- isMulticast : Boolean [1..1] = false
  Tells whether the objects in the flow are passed by multicasting.
- isMultireceive : Boolean [1..1] = false
  Tells whether the objects in the flow are gathered from respondents to multicasting.

Associations

Package CompleteActivities

- selection : Behavior [0..1]
  Selects tokens from a source object node.
- transformation : Behavior [0..1]
  Changes or replaces data tokens flowing along edge.

Constraints

Package BasicActivities

[1] Object flows may not have actions at either end.

[2] Object nodes connected by an object flow, with optionally intervening control nodes, must have compatible types. In particular, the downstream object node type must be the same or a supertype of the upstream object node type.

[3] Object nodes connected by an object flow, with optionally intervening control nodes, must have the same upper bounds.

Package CompleteActivities

[1] An edge with constant weight may not target an object node, or lead to an object node downstream with no intervening actions, that has an upper bound less than the weight.

[2] A transformation behavior has one input parameter and one output parameter. The input parameter must be the same 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.

[3] An object flow may have a selection behavior only if it has an object node as a source.

[4] A selection behavior has one input parameter and one output parameter. The input parameter must be a bag of elements of the same or a supertype of the type of source object node. The output parameter must be the same or a subtype of the type of source object node. The behavior cannot have side effects.

[5] isMulticast and isMultireceive cannot both be true.

Semantics

Package BasicActivities

See semantics inherited from ActivityEdge. An object flow is an activity edge that only passes object and data tokens. Tokens offered by the source node are all offered to the target node, subject to the restrictions inherited from ActivityEdge.

Two object flows may have the same object node as source. In this case the edges will compete for objects. Once an edge takes an object from an object node, the other edges do not have access to it. Use a fork to duplicate tokens for multiple uses.

Package CompleteActivities

If a transformation behavior is specified, then each token offered to the edge is passed to the behavior, and the output of the behavior is given to the target node for consideration instead of the token that was input to the transformation behavior. Because the 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 may not modify objects, but it may for example, navigate from one object to another, get an attribute value from an object, or replace a data value with another. Transformation behaviors with an output parameter with multiplicity greater than 1 may replace one token with many.

If a selection behavior is specified, then it is used to offer a token from the source object node to the edge, rather than using object node's ordering. It has the same semantics as selection behavior on object nodes (see ObjectNode; see application at DataStoreNode).

Multicasting and multireceiving are used in conjunction with partitions to model flows between behaviors that are the responsibility of objects determined by a publish and subscribe facility. To support execution the model must be refined to specify the particular publish/subscribe facility employed. This is illustrated in the Figure 12.113 on page 392.

Notation

An object flow is notated by an arrowed line. In Figure 12.107, 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 also the discussion on Figure 12.120 on page 402.

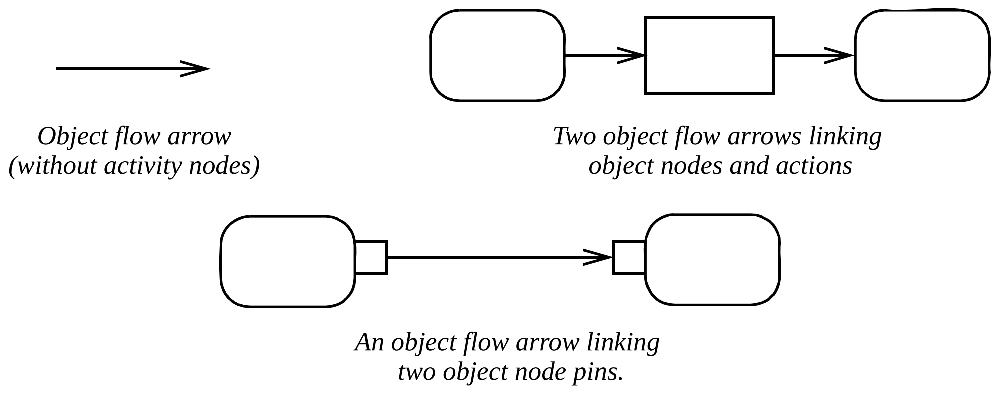

Object flow arrow
(without activity nodes)
Two object flow arrows linking
object nodes and actions
An object flow arrow linking
two object node pins.

Figure 12.107 - Object flow notations

Package CompleteActivities

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.

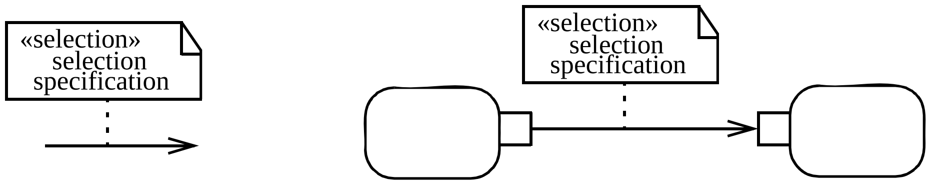


Figure 12.108 - Specifying selection behavior on an Object flow

Presentation Options

To reduce clutter in complex diagrams, object nodes may be elided. The names of the invoked behaviors can suggest their parameters. Tools may support hyperlinking from the edge 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.

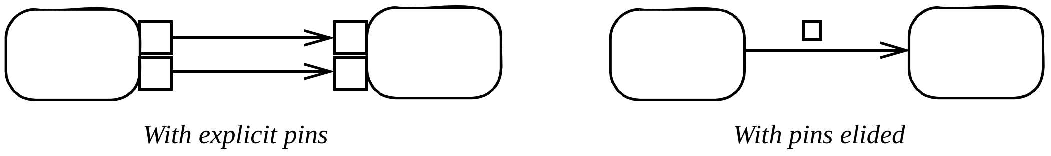


Figure 12.109 - Eliding objects flowing on the edge

Examples

In the example on the left below, the two arrowed lines are both object flow edges. This indicates that order objects flow from Fill Order to Ship Order. In the example on the right, the one arrowed line starts from the Fill Order object node pin and ends at Ship Order object node pin. This also indicates that order objects flow from Fill Order to Ship Order.

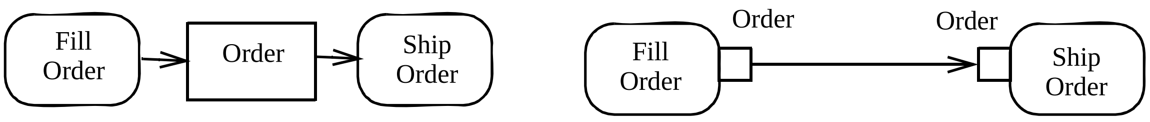


Figure 12.110 - Object flow example

On the left, the example below shows the Pick Materials activity provides an order along with its associated materials for assembly. On the right, the object flow has been simplified through eliding the object flow details.

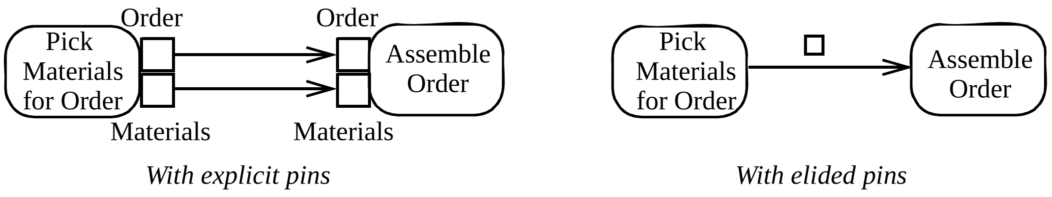


Figure 12.111 - Eliding objects flowing on the edge

Package CompleteActivities

In the figure below, two examples of selection behavior are illustrated. 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 a Close Order activity produces closed order objects, but the Send Customer Notice activity requires a customer object. The transformation, specifies that a query operation that takes an Order evaluates the customer object via the Order.customer:Party association.

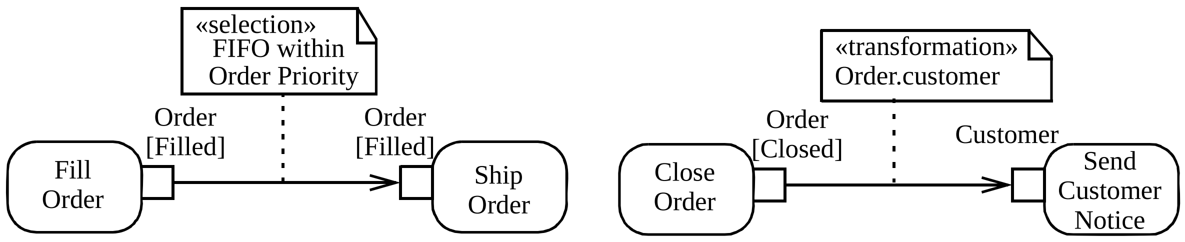


Figure 12.112 - Specifying selection behavior on an Object flow

In the example below, 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. Since multiple responses can be received, the edge is labeled for the multiple-receipt option. Publish/subscribe and other brokered mechanisms can be handled using the multicast and multireceive mechanisms. Note that the swimlanes are an important feature for indicating the subject and source of this.

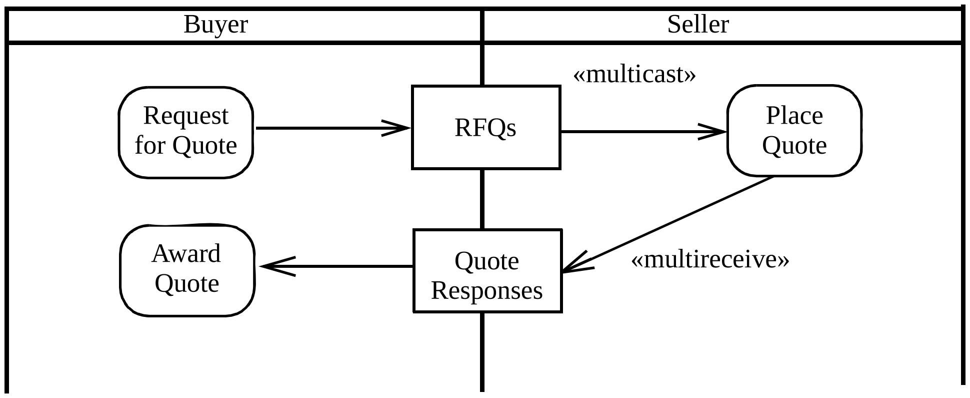


Figure 12.113 - Specifying multicast and multireceive on the edge

Rationale

Object flow is introduced to model the flow of data and objects in an activity.

Changes from previous UML

Explicitly modeled object flows are new in UML 2.0. They replace the use of (state) Transition in UML 1.5 activity modeling. They also replace data flow dependencies from UML 1.5 action model.

12.3.38 ObjectNode (from BasicActivities, CompleteActivities)

An object node is an abstract activity node that is part of defining object flow in an activity.

Generalizations

- "ActivityNode (from BasicActivities, CompleteActivities, FundamentalActivities, IntermediateActivities, CompleteStructuredActivities)" on page 333
- "TypedElement (from Kernel)" on page 136

Description

An object node is an activity node that indicates an instance of a particular classifier, possibly in a particular state, may be available at a particular point in the activity. Object nodes can be used in a variety of ways, depending on where objects are flowing from and to, as described in the semantics sub clause.

Package CompleteActivities

Complete object nodes add support for token selection, limitation on the number of tokens, specifying the state required for tokens, and carrying control values.

Attributes

Package CompleteActivities

- ordering : ObjectNodeOrderingKind [1..1] = FIFO
  Tells whether and how the tokens in the object node are ordered for selection to traverse edges outgoing from the object node.
- isControlType : Boolean [1..1] = false
  Tells whether the type of the object node is to be treated as control.

Associations

Package CompleteActivities

- inState : State [0..*]
  The required states of the object available at this point in the activity.
- selection : Behavior [0..1]
  Selects tokens for outgoing edges.
- upperBound : ValueSpecification [1..1] = *
  The maximum number of tokens allowed in the node. Objects cannot flow into the node if the upper bound is reached.

Constraints

Package BasicActivities

[1] All edges coming into or going out of object nodes must be object flow edges.

[2] Object nodes are not unique typed elements.
isUnique = false

Package CompleteActivities

[1] If an object node has a selection behavior, then the ordering of the object node is ordered and vice versa.

[2] A selection behavior has one input parameter and one output parameter. The input parameter must be a bag of elements of the same type as the object node or a supertype of the type of object node. The output parameter must be the same or a subtype of the type of object node. The behavior cannot have side effects.

Semantics

Object nodes may only contain values at runtime that conform to the type of the object node, in the state or states specified, if any. If no type is specified, then the values may be of any type. Multiple tokens containing the same value may reside in the object node at the same time. This includes data values. A token in an object node can traverse only one of the outgoing edges.

An object node may indicate that its type is to be treated as a control value, even if no type is specified for the node. Control edges may be used with the object node having control type.

Package CompleteActivities

An object node may not contain more tokens than its upper bound. The upper bound must be a LiteralUnlimitedNatural. An upper bound of * means the upper bound is unlimited. See ObjectFlow for additional rules regarding when objects may traverse the edges incoming and outgoing from an object node.

The ordering of an object node specifies the order in which tokens in the node are offered to the outgoing edges. This can be set to require that tokens do not overtake each other as they pass through the node (FIFO), or that they do (LIFO or modeler-defined ordering). Modeler-defined ordering is indicated by an ordering value of ordered, and a selection behavior that determines what token to offer to the edges. The selection behavior takes all the tokens in the object node as input and chooses a single token from those. It is executed whenever a token is to be offered to an edge. Because the behavior is used while offering tokens to outgoing edges, it may be run many times on the same token before the token is accepted by those edges. This means the behavior cannot have side effects. The selection behavior of an object node is overridden by any selection behaviors on its outgoing edges (see “ObjectFlow”). Overtaking due to ordering is distinguished from the case where each invocation of the activity is handled by a separate execution of the activity. In this case, the tokens have no interaction with each other, because they flow through separate executions of the activity (see “Activity”).

Notation

Object nodes are notated as rectangles. A name labeling the node is placed inside the symbol, where the name indicates the type of the object node, or the name and type of the node in the format “name:type.” Object nodes whose instances are sets of the “name” type are labeled as such. Object nodes with a signal as type are shown with the symbol on the right.

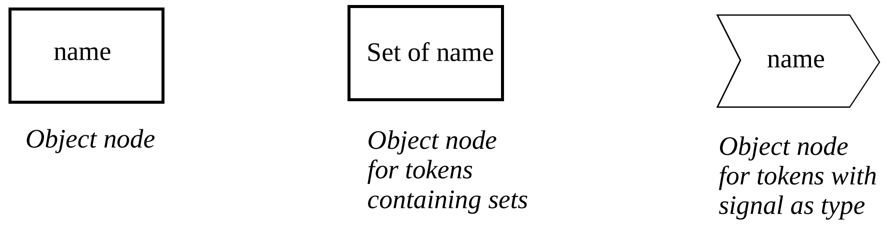


Figure 12.114 - Object node notations

Package CompleteActivities

A name labeling the node indicates the type of the object node. The name can also be qualified by a state or states, which is to be written within brackets below the name of the type. Upper bounds, ordering, and control type other than the defaults are notated in braces underneath the object node.

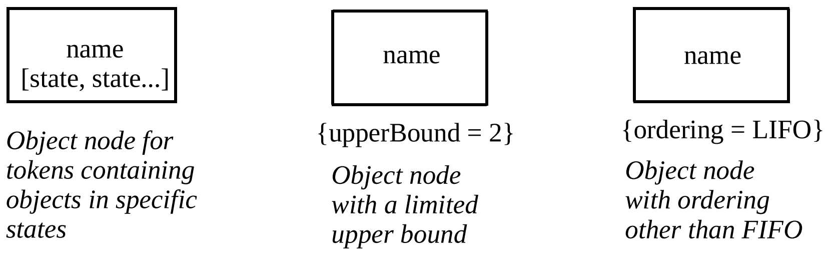

name
[state, state...]
Object node for tokens containing objects in specific states
name
{upperBound = 2}
Object node with a limited upper bound
name
{ordering = LIFO}
Object node with ordering other than FIFO

Figure 12.115 - Object node notations

Selection behavior is specified with the keyword «selection» placed in a note symbol, and attached to an ObjectNode symbol as illustrated in the figure below.

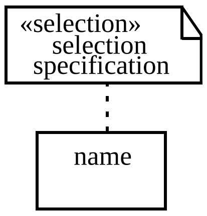


Figure 12.116 - Specifying selection behavior on an Object node

Presentation Options

It is expected that the UML 2.0 Diagram Interchange specification will define a metaassociation between model elements and view elements, like diagrams. It can be used to link an object node to an object diagram showing the classifier that is the type of the object and its relations to other elements. Tools can use this information in various ways to integrate the activity and class diagrams, such as a hyperlink from the object node to the diagram, or insertion of the class diagram in the activity diagram as desired. See example in Figure 12.127.

Examples

See examples at ObjectFlow and children of ObjectNode.

Rationale

Object nodes are introduced to model the flow of objects in an activity.

Changes from previous UML

ObjectNode replaces and extends ObjectFlowState in UML 1.5. In particular, it and its children support collection of tokens at runtime, single sending and receipt, and the new “pin” style of activity model.

12.3.39 ObjectNodeOrderingKind (from CompleteActivities)

Generalizations

None

Description

ObjectNodeOrderingKind is an enumeration indicating queuing order within a node.

Enumeration Values

- unordered
- ordered
- LIFO
- FIFO

12.3.40 OutputPin

Output pins are object nodes that deliver values to other actions through object flows. See Pin, Action, and ObjectNode for more details.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] Output pins may have incoming edges only when they are on actions that are structured nodes, and these edges may not target a node contained by the structured node.

Semantics

See “OutputPin (from BasicActions)” on page 263.

12.3.41 Parameter (from CompleteActivities)

Parameter is specialized when used with complete activities.

Generalizations

- “Parameter (from Kernel)” on page 120

Description

Parameters are extended in complete activities to add support for streaming, exceptions, and parameter sets.

Attributes

- 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 post values while the behavior is executing.
- parameterSet : ParameterSet [0..*]
  The parameter sets containing the parameter. See ParameterSet.

Associations

No additional associations

Constraints

[1] A parameter cannot be a stream and exception at the same time.

[2] An input parameter cannot be an exception.

[3] Reentrant behaviors cannot have stream parameters.

[4] Only in and inout parameters may have a delete effect. Only out, inout, and return parameters may have a create effect.

Semantics

isException applies to output parameters. An output posted to an exception excludes outputs from being posted to other data and control outputs of the behavior. A token arriving at an exception output parameter of an activity aborts all flows in the activity. Any objects previously posted to non-stream outputs never leave the activity. Streaming outputs posted before any exception 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, then multiple streams of tokens will be flowing through the same activity. In this case, it is probably not desired to abort all tokens just because one reaches an exception. Arrange for separate executions of the activity to use separate executions of the activity when employing exceptions, so tokens from separate executions will not affect each other.

Streaming parameters give an action access to tokens passed from its invoker while the action is executing. Values for streaming parameters may arrive anytime during the execution of the action, not just at the beginning. Multiple value may arrive on a streaming parameter during a single action execution and be consumed by the action. In effect, streaming parameters give an action access to token flows outside of the action while it is executing. In addition to the execution rules given at Action, these rules also apply to invoking a behavior with streaming parameters:

- All required non-stream inputs must arrive for the behavior to be invoked. If there are only required stream inputs, then at least one must arrive for the behavior to be invoked.
- All required inputs must arrive for the behavior to finish.
- Either all required non-exception outputs must be posted by the time the activity is finished (output of required streaming parameters may be posted before execution finishes), or one of the exception outputs must be. An activity finishes when all its tokens are in its output parameter nodes. If some output parameter nodes are empty at that time, they are assigned the null token (see “Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)” on page 315), and the activity terminates.

The execution rules above provide for the arrival of inputs after a behavior is started and the posting of outputs before a behavior is finished. These are stream inputs and outputs. Multiple stream input and output tokens may be consumed and posted while a behavior is running. Since an activity is a kind of behavior, the above rules apply to invoking an activity, even if the invocation is not from another activity. A reentrant behavior cannot have streaming parameters because there are potentially multiple executions of the behavior going at the same time, and it is ambiguous which execution should receive streaming tokens.

The effect of a parameter is a declaration of the modeler's intent, and does not have execution semantics. The modeler must ensure that the owner of the parameter has the stated effect.

See semantics of Action and ActivityParameterNode. Also, see "MultiplicityElement (from Kernel)" on page 94, which inherits to Parameter. It defines a lower and upper bound on the values passed to 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.

Notation

See notation at Pin and ActivityParameterNode. The notation in class diagrams for exceptions and streaming parameters on operations has the keywords "exception" or "stream" in the property string. See notation for Operation.

Examples

See examples at Pin and ActivityParameterNode.

Rationale

Parameter (in Activities) is extended to support invocation of behaviors by activities.

Changes from previous UML

Parameter (in Activities) is new in UML 2.0.

12.3.42 ParameterEffectKind (from CompleteActivities)

Generalizations

None

Description

The datatype ParameterEffectKind is an enumeration that indicates the effect of a behavior on values passed in or out of its parameters (see "Parameter (from CompleteActivities)" on page 396).

Enumeration Values

- create
- read
- update
- delete

12.3.43 ParameterSet (from CompleteActivities)

A parameter set is an element that provides alternative sets of inputs or outputs that a behavior may use.

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 98

Description

A parameter set acts as a complete set of inputs and outputs to a behavior, exclusive of other parameter sets on the behavior.

Attributes

No additional attributes

Associations (CompleteActivities)

- condition : Constraint [0..*]
  Constraint that should be satisfied for the owner of the parameters in an input parameter set to start execution using the values provided for those parameters, or the owner of the parameters in an output parameter set to end execution providing the values for those parameters, if all preconditions and conditions on input parameter sets were satisfied. {Subsets *Element::ownedElement*}
- parameter : Parameter [1..*]
  Parameters in the parameter set.

Constraints

[1] The parameters in a parameter set must all be inputs or all be outputs of the same parameterized entity, and the parameter set is owned by that entity.

[2] If a behavior has input parameters that are in a parameter set, then any inputs that are not in a parameter set must be streaming. Same for output parameters.

[3] Two parameter sets cannot have exactly the same set of parameters.

Semantics

A behavior with input parameter sets can only accept inputs from parameters in one of the sets per execution. A behavior with output parameter sets can only post outputs to the parameters in one of the sets per execution. The same is true for operations with parameter sets. The semantics described at Action and ActivityParameter apply to each set separately. The semantics of conditions of input and output parameter sets is the same as Behavior preconditions and postconditions, respectively, but apply only to the set of parameters specified.

Notation

Multiple object flows entering or leaving a behavior invocation are typically treated as "and" conditions. However, sometimes one group of flows are permitted to the exclusion of another. This is modeled as parameter set and notated with rectangles surrounding one or more pins. The notation in the figure below expresses a disjunctive normal form where one group of "and" flows are separated by "or" groupings. For input, when one group or another has a complete set of input flows, the activity may begin. For output, based on the internal processing of the behavior, one group or other of output flows may occur.

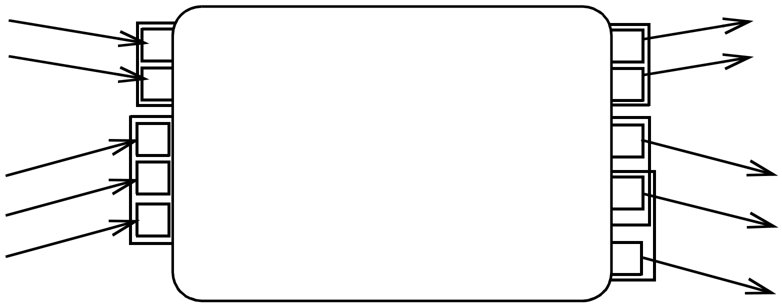
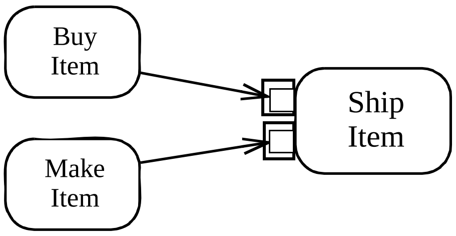


Figure 12.117 - Alternative input/outputs using parameter sets notation

Examples

In the figure below, the Ship Item activity begins whenever it receives a bought item or a made item.



Using parameter sets to express "or" invocation

Figure 12.118 - Example of alternative input/outputs using parameter sets

Rationale

Parameter sets provide a way for behaviors to direct token flow in the activity that invokes those behaviors.

Changes from previous UML

ParameterSet is new in UML 2.0.

12.3.44 Pin (from BasicActivities, CompleteActivities)

Generalizations

- "ObjectNode (from BasicActivities, CompleteActivities)" on page 393
- "Pin (from BasicActions)" on page 263 *(merge increment)*

Description

A pin is an object node for inputs and outputs to actions.

Attributes

Package CompleteActivities

- isControl : Boolean [1..1] = false
  Tells whether the pins provide data to the actions, or just controls when it executes it.

Associations

No additional associations

Constraints

See constraints on ObjectFlow.

Constraints

Package CompleteActivities

[1] Control pins have a control type.
isControl **implies** isControlType

Semantics

See “Pin (from BasicActions)” on page 263.

(CompleteActivities) Control pins always have a control type, so they can be used with control edges. Control pins are ignored in the constraints that actions place on pins, including matching to behavior parameters for actions that invoke behaviors. Tokens arriving at control input pins have the same semantics as control arriving at an action, except that control tokens can queue up in control pins. Tokens are placed on control output pins according to the same semantics as tokens placed on control edges coming out of actions.

Notation

Pin rectangles may be notated as small rectangles that are attached to action rectangles. See figure below and examples. The name of the pin can 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. It can also be a full specification of the corresponding behavior parameter for invocation actions, using the same notation as parameters for behavioral features on classes. The pins may be elided in the notation even though they are present in the model. Pins that do not correspond to parameters can be labeled as “name:type.”

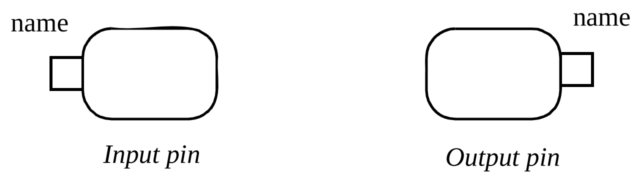


Figure 12.119 - Pin notations

The situation in which the output pin of one action is connected to the input pin of the same name in another action may be shown by the optional notations of Figure 12.120. The standalone pin in the notation maps to an output pin and an input pin and one object flow 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 output pin. These variations in notation assume the UML 2.0 Diagram Interchange specification supports the interchange of diagram elements and their mapping to model elements, so that the chosen variation is preserved on interchange.

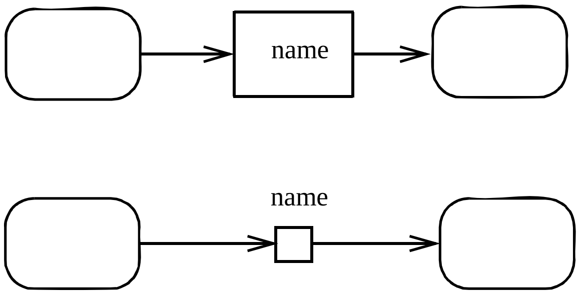


Figure 12.120 - Standalone pin notations

See ObjectNode for other notations applying to pins, with examples for pins below.

Package CompleteActivities

To show streaming, a text annotation is placed near the pin symbol: {stream} or {nonstream}. See figure below. The notation is the same for a standalone object node. Nonstream is the default where the notation is omitted.

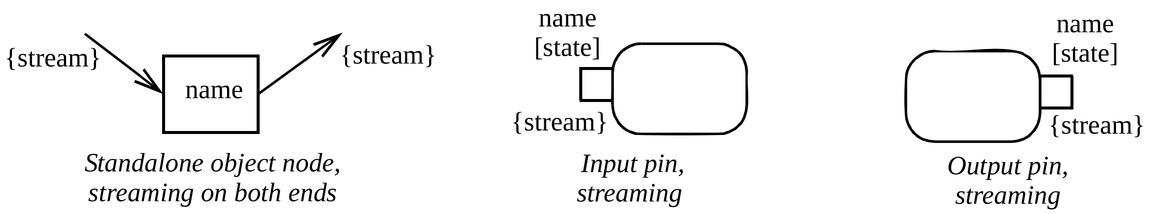
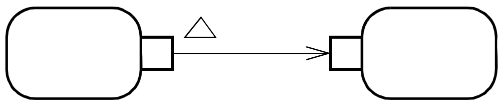


Figure 12.121 - Stream pin notations

Pins for exception parameters are indicated with a small triangle annotating the source end of the edge that comes out of the exception pin. The notation is the same even if the notation uses a standalone notation. See figure below.


Output pin, pin style, exception

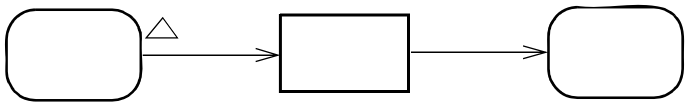

Input and output pin, standalone style, exception

Figure 12.122 - Exception pin notations

Specifying the effect that the behavior of actions has on the objects passed in and out of their parameters can be represented by placing the effect in braces near the edge leading to or from the pin for the parameter.

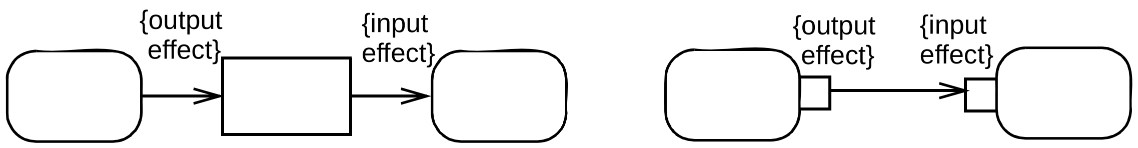


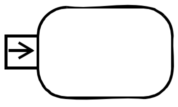
Figure 12.123 - Specifying effect that actions have on objects

Control pins are shown with a text annotation placed near the pin symbol {control}.

See ObjectNode for other notations applying to pins, with examples for pins below.

Presentation Options

When edges are not present to distinguish input and output pins, an optional arrow may be placed inside the pin rectangle, as shown below. Input pins have the arrow pointing toward the action and output pins have the arrow pointing away from the action.


Input pin,
pin-style, with arrow

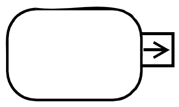

Output pin,
pin-style, with arrow

Figure 12.124 - Pin notations, with arrows

Package CompleteActivities

Additional emphasis may be added to streaming parameters by using a graphical notation instead of the textual adornment. Object nodes can be connected with solid arrows containing filled arrowheads to indicate streaming. Pins can be shown as filled rectangles. When combined with the option above, the arrows are shown as normal arrowheads.

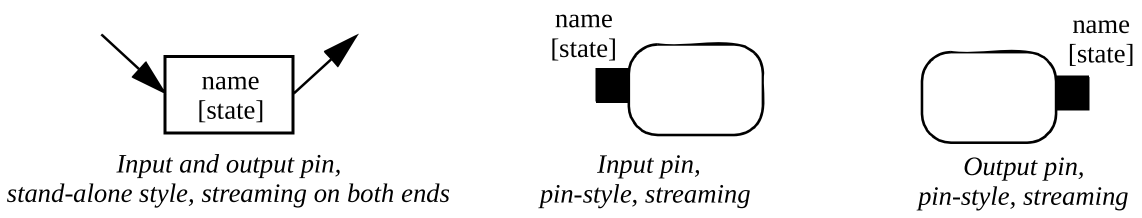


Figure 12.125 - Stream pin notations, with filled arrows and rectangles

Examples

In the example below, the pin named “Order” represents Order objects. In this example at the upper left, the Fill Order behavior produces filled orders and Ship Order consumes them and an invocation of Fill Order must complete for Ship Order to begin. The pin symbols have been elided from the action symbols; both pins are represented by the single box on the arrow. The example on the upper right shows the same thing with explicit pin symbols on actions. The example at the bottom of the figure illustrates the use of multiple pins.

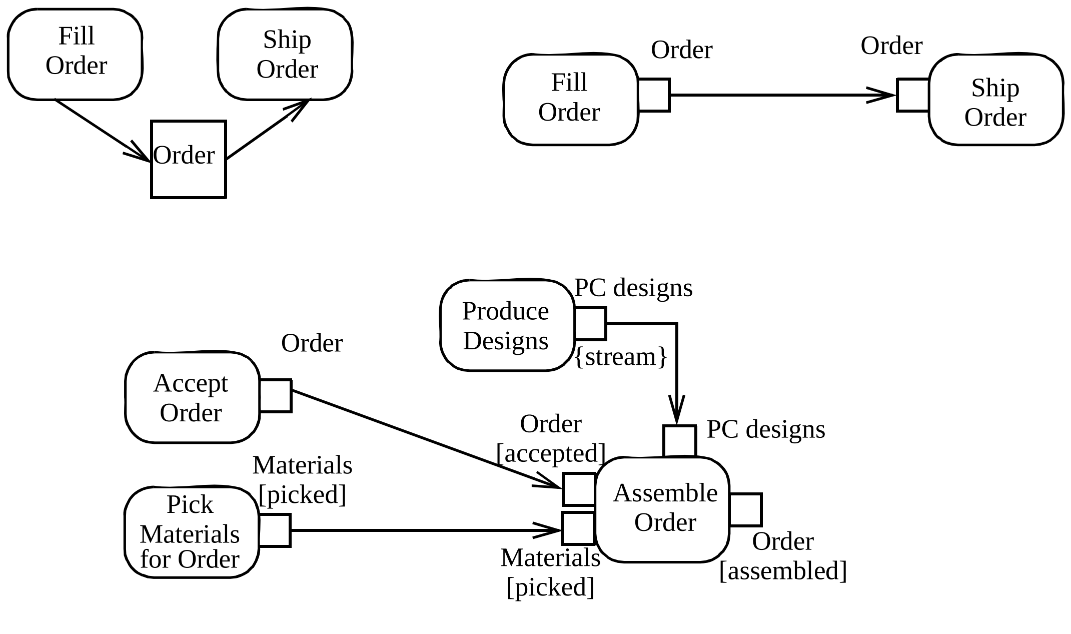


Figure 12.126 - Pin examples

In the figure below, 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.

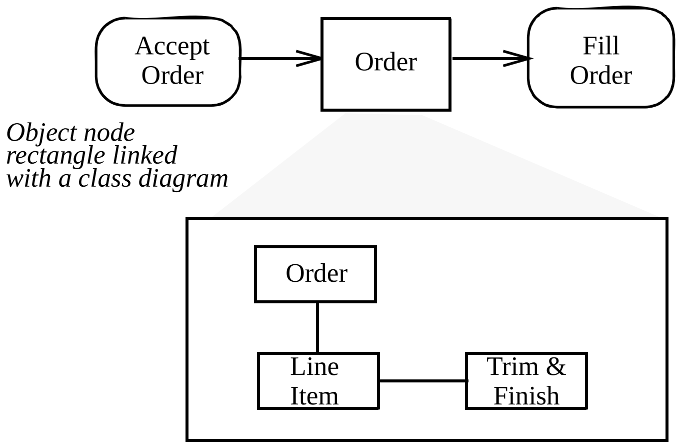


Figure 12.127 - Linking a class diagram to an object node

Package CompleteActivities

In the example below Order Filling is a continuous behavior that periodically emits (streams out) filled-order objects, without necessarily concluding as an activity. The Order Shipping behavior 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.

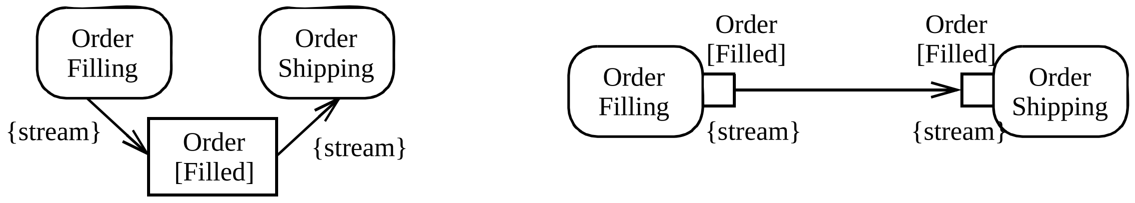


Figure 12.128 - Pin examples

Example of exception notation is shown at the top of the figure below. 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 raised that the payment is not valid, and the payment is rejected.

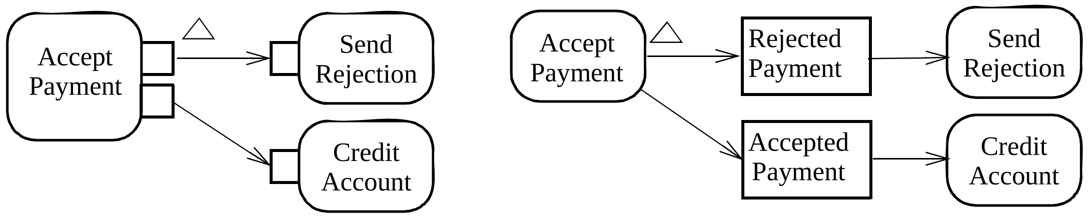


Figure 12.129 - Exception pin examples

The figure below shows two examples of selection behavior. Both examples indicate that orders are to be shipped based or order priority—and those with the same priority should be filled on a first-in/first-out (FIFO) basis.

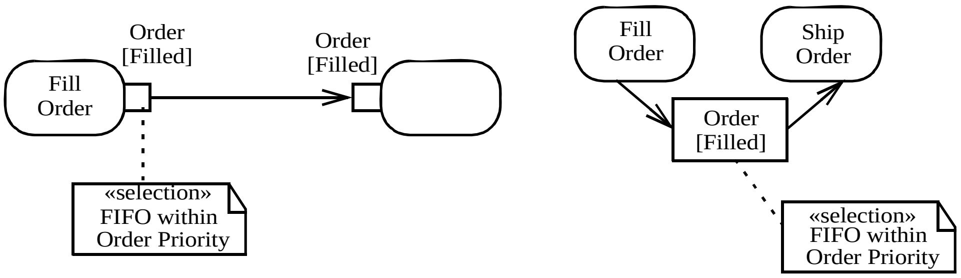

Order [Filled]
Order [Filled]
Fill Order
«selection» FIFO within Order Priority
Fill Order
Ship Order
Order [Filled]
«selection» FIFO within Order Priority

Figure 12.130 - Specifying selection behavior on an ObjectFlow

In the figure below, an example depicts a Place Order activity that creates orders and Fill Order activity that reads these placed orders for the purpose of filling them.

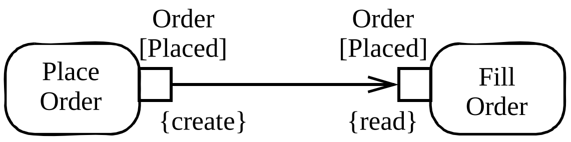


Figure 12.131 Pin example with effects

Rationale

Pin is specialized in Activities to make it an object node and to give it a notation.

Changes from previous UML

Pin is new to activity modeling in UML 2.0. It replaces pins from UML 1.5 action model.

12.3.45 SendObjectAction (as specialized)

See "SendObjectAction (from IntermediateActions)" on page 281.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See “SendObjectAction (from IntermediateActions)” on page 281.

Notation

No specific notation

Presentation Options

See “SendObjectAction (from IntermediateActions)” on page 281.

Changes from previous UML

See “SendObjectAction (from IntermediateActions)” on page 281.

12.3.46 SendSignalAction (as specialized)

See “SendSignalAction (from BasicActions)” on page 282.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See “SendSignalAction (from BasicActions)” on page 282.

Notation

See “SendSignalAction (from BasicActions)” on page 282.

Examples

Figure 12.132 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.

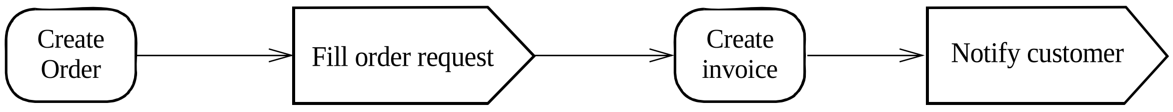


Figure 12.132 - Signal node notations

Rationale

See “SendSignalAction (from BasicActions)” on page 282.

Changes from previous UML

See “SendSignalAction (from BasicActions)” on page 282.

12.3.47 SequenceNode (from StructuredActivities)

Generalizations

- “StructuredActivityNode (from CompleteStructuredActivities, StructuredActivities)” on page 409

Description

(StructuredActivities) A sequence node is a structured activity node that executes its actions in order.

Attributes

No additional attributes

Associations

- executableNode : ExecutableNode [*] {ordered}
  An ordered set of executable nodes. {Redefines *StructuredActivityNode::node*}

Constraints

No additional constraints

Semantics

When the sequence node is enabled, its executable nodes are executed in the order specified. When combined with flows, actions must also satisfy their control and data flow inputs before starting execution.

Notation

No specific notation

Rationale

SequenceNode is introduced to provide a way for structured activities to model a sequence of actions.

Changes from previous UML

SequenceNode is new to UML 2.

12.3.48 StructuredActivityNode (from CompleteStructuredActivities, StructuredActivities)

(StructuredActivities) A structured activity node is an executable activity node that may have an expansion into subordinate nodes as an ActivityGroup. The subordinate nodes must belong to only one structured activity node, although they may be nested.

Generalizations

- "Action (from CompleteActivities, FundamentalActivities, StructuredActivities)" on page 311
- "ActivityGroup (from BasicActivities, FundamentalActivities)" on page 332
- "ExecutableNode (from ExtraStructuredActivities, StructuredActivities)" on page 366
- "Namespace (from Kernel)" on page 99

Description

A structured activity node represents a structured portion of the activity that is not shared with any other structured node, except for nesting. It may have control edges connected to it, and pins when merged with CompleteActivities or on specializations in CompleteStructuredActivities. The execution of any embedded actions may not begin until the structured activity node has received its object and control tokens. The availability of output tokens from the structured activity node does not occur until all embedded actions have completed execution (see exception at AcceptEventAction (from CompleteActions)).

Package CompleteStructuredActivities

Because of the concurrent nature of the execution of actions within and across activities, it can be difficult to guarantee the 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 structured activity node. If a structured activity node is "isolated," then any object used by an action within the node cannot be accessed by any action outside the node until the structured activity node 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 node.

Note – Any required isolation may be achieved using a locking mechanism, or it may simply sequentialize execution to avoid concurrency conflicts. 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.

Attributes

- mustIsolate : Boolean
  If *true*, then the actions in the node execute in isolation from actions outside the node. Default value is *false*.

Associations

- variable: Variable [0..*]
  A variable defined in the scope of the structured activity node. It has no value and may not be accessed outside the node. {Subsets *Namespace::ownedMember*}
- node : ActivityNode [0..*]
  Nodes immediately contained in the group. (Subsets *ActivityGroup::containedNode)*
- activity : Activity [0..1]
  Activity immediately containing the node. {Redefines *ActivityNode::activity* and *ActivityGroup::inActivity*}
- edge : ActivityEdge [0..*]
  Edges immediately contained in the structured node. {Subsets *ActivityGroup::containedEdge*}

Constraints

Package CompleteStructuredActivities

[1] The edges owned by a structured node must have source and target nodes in the structured node, and vice versa.

Semantics

Nodes and edges contained by a structured node cannot be contained by any other structured node. This constraint is modeled as a specialized multiplicity from ActivityNode and ActivityEdge to StructuredActivityNode. Edges not contained by a structured node can have sources or targets in the structured node, but not both. See children of StructuredActivityNode.

No subnode in the structured node, including initial nodes and accept event actions, may begin execution until the structured node itself has started. Subnodes begin executing according to the same rules as the subnodes of an activity (see “InitialNode (from BasicActivities)” on page 378 and “AcceptEventAction (from CompleteActions)” on page 234). A control flow from a structured activity node implies that a token is produced on the flow only after no tokens are left in the node or its contained nodes recursively. Tokens reaching an activity final node in a structured node abort all flows in the immediately containing structured node only. The other aspects of termination are the same as for activity finals contained directly by activities (see “ActivityFinalNode (from BasicActivities, IntermediateActivities)” on page 330).

Package CompleteStructuredActivities

A pin of a structured activity node is accessible within the node. The same rules apply as for control flow. Input pins on a structured activity node imply that actions in the node begin execution when all input pins have received tokens. An output pin on a structured activity node will make tokens available outside the node only after no tokens are left in the node or its contained nodes recursively.

If the mustIsolate flag is true for an activity node, 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 node may be interleaved with the execution of the node. This specification does not constrain the ways in which this rule may be enforced. If it is impossible to execute a model in accordance with these rules, then it is ill formed.

Notation

A structured activity node is notated with a dashed round cornered rectangle enclosing its nodes and edges, with the keyword «structured» at the top (see Figure 12.133). Also see children of StructuredActivityNode.

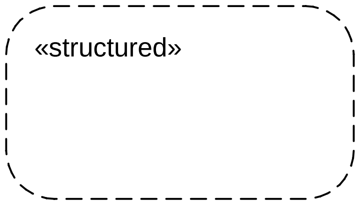


Figure 12.133 Notation for structured nodes

Examples

See children of StructuredActivityNode.

Rationale

StructuredActivityNode is for applications that require well-nested nodes. It provides well-nested nodes that were enforced by strict nesting rules in UML 1.5.

Changes from previous UML

StructuredActivityNode is new in UML 2.0.

12.3.49 UnmarshallAction (as specialized)

See "UnmarshallAction (from CompleteActions)" on page 288.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See "UnmarshallAction (from CompleteActions)" on page 288.

Notation

No specific notation

Examples

In Figure 12.134, an order is unmarshalled into its name, shipping address, and product.

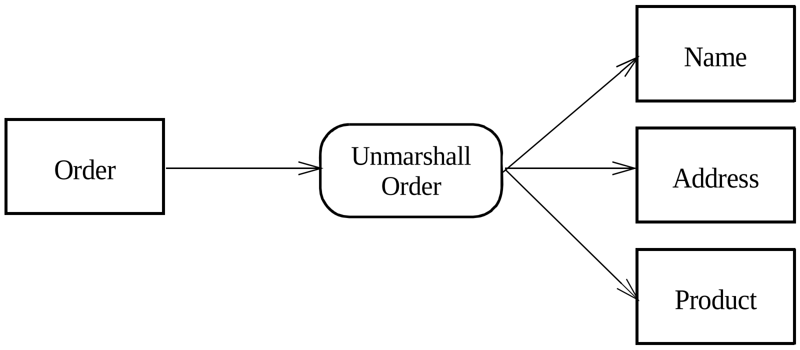


Figure 12.134 - Example of UnmarshallAction

Rationale

See “UnmarshallAction (from CompleteActions)” on page 288.

Changes from previous UML

See “UnmarshallAction (from CompleteActions)” on page 288.

12.3.50 ValuePin (as specialized)

A value pin is an input pin that provides a value to an action that does not come from an incoming object flow edge. See “ValuePin (from BasicActions)” on page 289.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] Value pins have no incoming edges.

Semantics

ValuePins provide values to their actions, but only when the actions are otherwise enabled. If an action has no incoming edges or other way to start execution, a value pin will not start the execution by itself or collect tokens waiting for execution to start. When the action is enabled by other means, the value specification of the value pin is evaluated and the result provided as input to the action, which begins execution. This is an exception to the normal token flow semantics of activities.

Notation

A value pin is notated as an input pin with the value specification written beside it.

Rationale

ValuePin is introduced to reduce the size of activity models that use constant values. See “ValueSpecificationAction (from IntermediateActions)” on page 290.

Changes from UML 1.5

ValuePin replaces LiteralValueAction from UML 1.5.

12.3.51 ValueSpecificationAction (as specialized)

See “ValueSpecificationAction (from IntermediateActions)” on page 290.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See “ValueSpecificationAction (from IntermediateActions)” on page 290.

Notation

The action is labeled with the value specification, as shown in Figure 12.135.

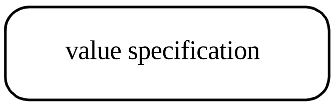


Figure 12.135 - ValueSpecificationAction notation

Examples

Figure 12.136 shows a value specification action used to output a constant from an activity.

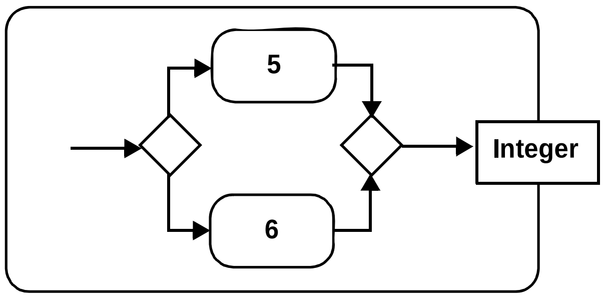


Figure 12.136 - Example ValueSpecificationAction

Rationale

See “ValueSpecificationAction (from IntermediateActions)” on page 290.

Changes from previous UML

See “ValueSpecificationAction (from IntermediateActions)” on page 290.

12.3.52 Variable (from StructuredActivities)

Variables are elements for passing data between actions indirectly. A local variable stores values shared by the actions within a structured activity group but not accessible outside it. The output of an action may be written to a variable and read for the input to a subsequent action, which is effectively an indirect data flow path. Because there is no predefined relationship between actions that read and write 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.

Generalizations

- “MultiplicityElement (from Kernel)” on page 94
- “TypedElement (from Kernel)” on page 136

Description

A variable specifies data storage shared by the actions within a group. There are actions to write and read variables. These actions are treated as side effecting actions, similar to the actions to write and read object attributes and associations. There are no sequencing constraints among actions that access the same variable. Such actions must be explicitly coordinated by control flows or other constraints.

Any values contained by a variable must conform to the type of the variable and have cardinalities allowed by the multiplicity of the variable.

Associations

- scope : StructuredActivityNode [0..1]
  A structured activity node that owns the variable. {Subsets *NamedElement::namespace*}

- activityScope : Activity [0..1]
  An activity that owns the variable. {Subsets *NamedElement::namespace*}

Attributes

No additional attributes

Constraints

[1] A variable is owned by a StructuredNode or Activity, but not both.

Additional operations

[1] The isAccessibleBy() operation is not defined in standard UML. Implementations should define it to specify which actions can access a variable.

isAccessibleBy(a: Action) : Boolean

Semantics

A variable specifies a slot able to hold a value or a sequence of values, consistent with the multiplicity of the variable. The values held in this slot may be accessed from any action contained directly or indirectly within the group action or activity that is the scope of the variable.

Notation

No specific notation

Rationale

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 attributes for the variables, or one object per variable combined with data store nodes.

Changes from UML 1.5

Variable is unchanged from UML 1.5, except that it is used on StructuredActivityNode instead of GroupNode.

12.4 Diagrams

The focus of activity modeling is the sequence and conditions for coordinating lower-level behaviors, rather than which classifiers own those behaviors. These are commonly called control flow and object flow models. The behaviors coordinated by these models can be initiated because other behaviors finish executing, because objects and data become available, or because events occur external to the flow. See 12.3.4, "Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)," on page 315 for more introduction and semantic framework.

The notation for activities is optional. A textual notation may be used instead.

The following sub clauses describe the graphic nodes and paths that may be shown in activity diagrams.

Graphic Nodes

The graphic nodes that can be included in activity diagrams are shown in Table 12.1.

Table 12.1 - Graphic nodes included in activity diagrams

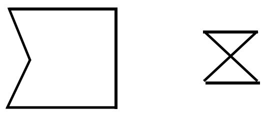
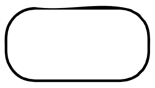
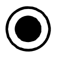
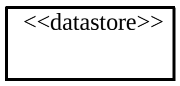
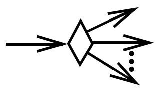
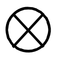
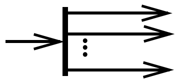
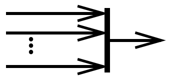
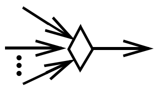
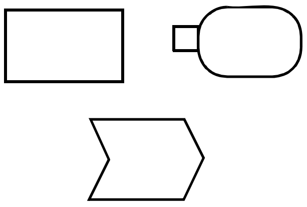
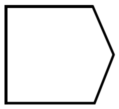
Node Type	**Notation**	**Reference**
AcceptEventAction		See “AcceptEventAction (as specialized)” on page 309.
Action		See “Action (from CompleteActivities, FundamentalActivities, StructuredActivities)” on page 311.
ActivityFinal		See “ActivityFinalNode (from BasicActivities, IntermediateActivities)” on page 330.
ActivityNode	*See ExecutableNode, ControlNode, and ObjectNode.*	See “ActivityNode (from BasicActivities, CompleteActivities, FundamentalActivities, IntermediateActivities, CompleteStructuredActivities)” on page 333.
ControlNode	*See DecisionNode, FinalNode, ForkNode, InitialNode, JoinNode, and MergeNode.*	See “ControlNode (from BasicActivities)” on page 356.
DataStore		See “DataStoreNode (from CompleteActivities)” on page 358.
DecisionNode		See “DecisionNode (from IntermediateActivities)” on page 360.
FinalNode	*See ActivityFinal and FlowFinal.*	See “FinalNode (from IntermediateActivities)” on page 373.
FlowFinal		See “FlowFinalNode (from IntermediateActivities)” on page 375.

Table 12.1 - Graphic nodes included in activity diagrams

Node Type	Notation	Reference
ForkNode		See "ForkNode (from IntermediateActivities)" on page 376.
InitialNode		See "InitialNode (from BasicActivities)" on page 378.
JoinNode		See "JoinNode (from CompleteActivities, IntermediateActivities)" on page 381.
MergeNode		See "MergeNode (from IntermediateActivities)" on page 387.
ObjectNode		See "ObjectNode (from BasicActivities, CompleteActivities)" on page 393 and its children.
SendSignalAction		See "SendSignalAction (as specialized)" on page 407.

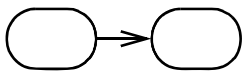
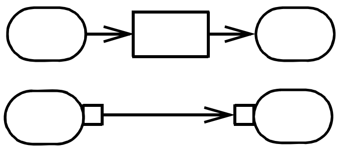
Graphic Paths

The graphic paths that can be included in activity diagrams are shown in Table 12.2

Table 12.2 - Graphic paths included in activity diagrams

Path Type		Reference
ActivityEdge	*See ControlFlow and ObjectFlow.*	See "ActivityEdge (from BasicActivities, CompleteActivities, CompleteStructuredActivities, IntermediateActivities)" on page 325.

Table 12.2 - Graphic paths included in activity diagrams

Path Type		Reference
ControlFlow		See "ControlFlow (from BasicActivities)" on page 355.
ObjectFlow		See "ObjectFlow (from BasicActivities, CompleteActivities)" on page 388 and its children.

Other Graphical Elements

Activity diagrams have graphical elements for containment. These are included in Table 12.3.

Table 12.3 - Graphic elements for containment in activity diagrams

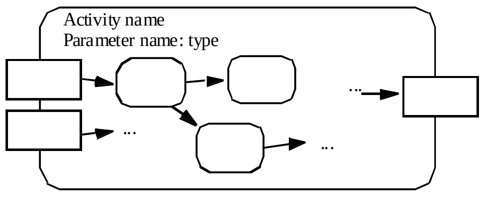
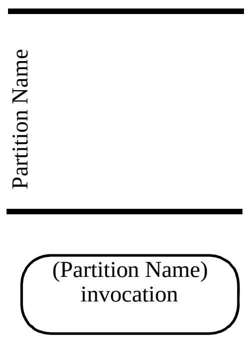
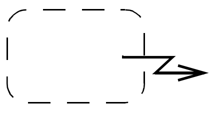
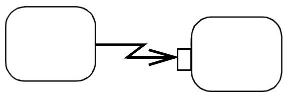
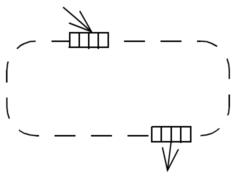
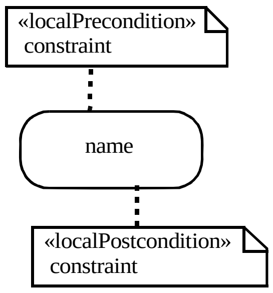
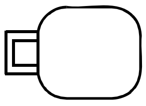
Type	Notation	Reference
Activity		See "Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)" on page 315.
ActivityPartition		See "ActivityPartition (from IntermediateActivities)" on page 340.
InterruptibleActivityRegion		See "InterruptibleActivityRegion (from CompleteActivities)" on page 379.

Table 12.3 - Graphic elements for containment in activity diagrams

Type	Notation	Reference
ExceptionHandler		See “ExceptionHandler (from ExtraStructuredActivities)” on page 363.
ExpansionRegion		“ExpansionRegion (from ExtraStructuredActivities)” on page 368
Local pre- and postconditions.	 «localPrecondition» constraint name «localPostcondition» constraint	See “Action (from CompleteActivities, FundamentalActivities, StructuredActivities)” on page 311.
ParameterSet		See “ParameterSet (from CompleteActivities)” on page 399.

13 Common Behaviors

13.1 Overview

The Common Behaviors packages specify the core concepts required for dynamic elements and provides the infrastructure to support more detailed definitions of behavior. Figure 13.1 shows a domain model explaining the relationship between occurrences of behaviors.

Note – The models shown in Figure 13.1 through Figure 13.4 are not metamodels but show objects in the semantic domain and relationships between these objects. These models are used to give an informal explication of the dynamic semantics of the classes of the UML metamodel.

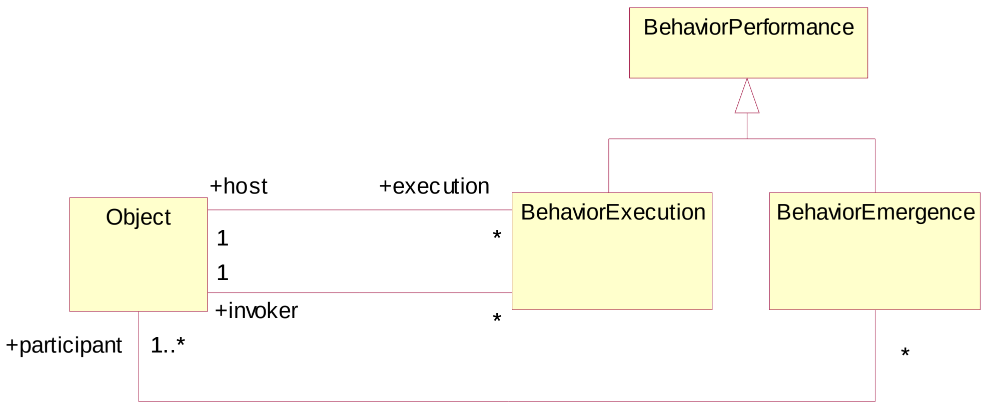


Figure 13.1 - Common Behaviors Domain Model

Any behavior is the direct consequence of the action of at least one object. A behavior describes how the states of these objects, as reflected by their structural features, change over time. Behaviors, as such, do not exist on their own, and they do not communicate. If a behavior operates on data, that data is obtained from the host object.

There are two kinds of behaviors, emergent behavior and executing behavior. An *executing behavior* is performed by an object (its host) and is the description of the behavior of this object. An executing behavior is directly caused by the invocation of a behavioral feature of that object or by its creation. In either case, it is a consequence of the execution of an action by some related object. A behavior has access to the structural features of its host object. Objects that may host behaviors are specified by the concrete subtypes of the *BehavioredClassifier* metaclass.

Emergent behavior results from the interaction of one or more participant objects. If the participating objects are parts of a larger composite object, an emerging behavior can be seen as indirectly describing the behavior of the container object also. Nevertheless, an emergent behavior can result from the executing behaviors of the participant objects.

Occurring behaviors are specified by the concrete subtypes of the abstract *Behavior* metaclass. Behavior specifications can be used to define the behavior of an object, or they can be used to describe or illustrate the behavior of an object. The latter may only focus on a relevant subset of the behavior an object may exhibit (allowed behavior), or it may focus on behavior an object must not exhibit (forbidden behavior).

Albeit behavior is ultimately related to an object, emergent behavior may also be specified for non-instantiable classifiers, such as interfaces or collaborations. Such behaviors describe the interaction of the objects that realize the interfaces or the parts of the collaboration (see “Collaboration (from Collaborations)” on page 168).

BasicBehaviors

The BasicBehaviors subpackage of the Common Behavior package introduces the framework that will be used to specify behaviors. The concrete subtypes of Behavior will provide different mechanisms to specify behaviors. A variety of specification mechanisms are supported by the UML, such as automata (“StateMachine (from BehaviorStateMachines)” on page 564), Petri-net like graphs (“Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)” on page 315), informal descriptions (“UseCase (from UseCases)” on page 596), or partially-ordered sequences of event occurrences (“Interaction (from BasicInteraction, Fragments)” on page 483). Profiles may introduce additional styles of behavioral specification. The styles of behavioral specification differ in their expressive power and domain of applicability. Further, they may specify behaviors either explicitly, by describing the observable event occurrences resulting from the execution of the behavior, or implicitly, by describing a machine that would induce these events. The relationship between a specified behavior and its hosting or participating instances is independent of the specification mechanism chosen and described in the common behavior package. The choice of specification mechanism is one of convenience and purpose; typically, the same kind of behavior could be described by any of the different mechanisms. Note that not all behaviors can be described by each of the different specification mechanisms, as these do not all have the same expressive power. However, for many behaviors, the choice of specification mechanism is one of convenience.

As shown in the domain model of Figure 13.2, the execution of a behavior may be caused by a call behavior occurrence (representing the direct invocation of a behavior through an action) or a trigger occurrence (representing an indirect invocation of a behavior, such as through an operation call). A start occurrence marks the beginning of a behavior execution, while its completion is accompanied by a termination occurrence.

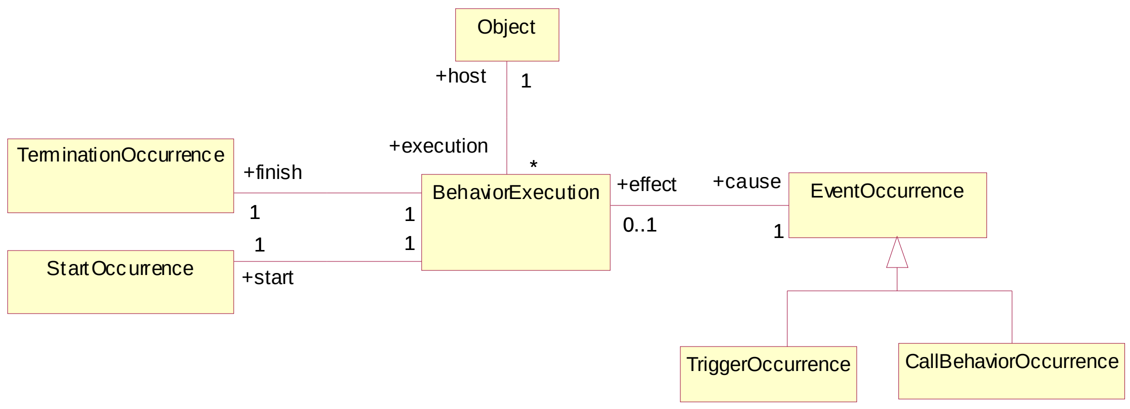


Figure 13.2 - Invocation Domain Model

Communications

The Communications subpackage of the Common Behavior package adds the infrastructure to communicate between objects in the system and to invoke behaviors. The domain model shown in Figure 13.3 explains how communication takes place. Note that this domain model specifies the semantics of communication between objects in a system. Not all aspects of the domain model are explicitly represented in the specification of the system, but may be implied by the dynamic semantics of the constructs used in a specification.

An action representing the invocation of a behavioral feature is executed by a sender object resulting in an invocation event occurring. The invocation event may represent the sending of a signal or the call to an operation. As a result of the invocation event a request is generated. A request is an object capturing the data that was passed to the action causing the invocation event (the arguments that must match the parameters of the invoked behavioral feature); information about the nature of the request (i.e., the behavioral feature that was invoked); the identities of the sender and receiver objects; as well as sufficient information about the behavior execution to enable the return of a reply from the invoked behavior, where appropriate. (In profiles, the request object may include additional information, for example, a time stamp.)

While each request is targeted at exactly one receiver object and caused by exactly one sending object, an occurrence of an invocation event may result in a number of requests being generated (as in a signal broadcast). The receiver may be the same object that is the sender, it may be local (i.e., an object held in a slot of the currently executing object, or the currently executing object itself, or the object owning the currently executing object), or it may be remote. The manner of transmitting the request object, the amount of time required to transmit it, the order in which the transmissions reach their receiver objects, and the path for reaching the receiver objects are undefined. Once the generated request arrives at the receiver object, a receiving event will occur.

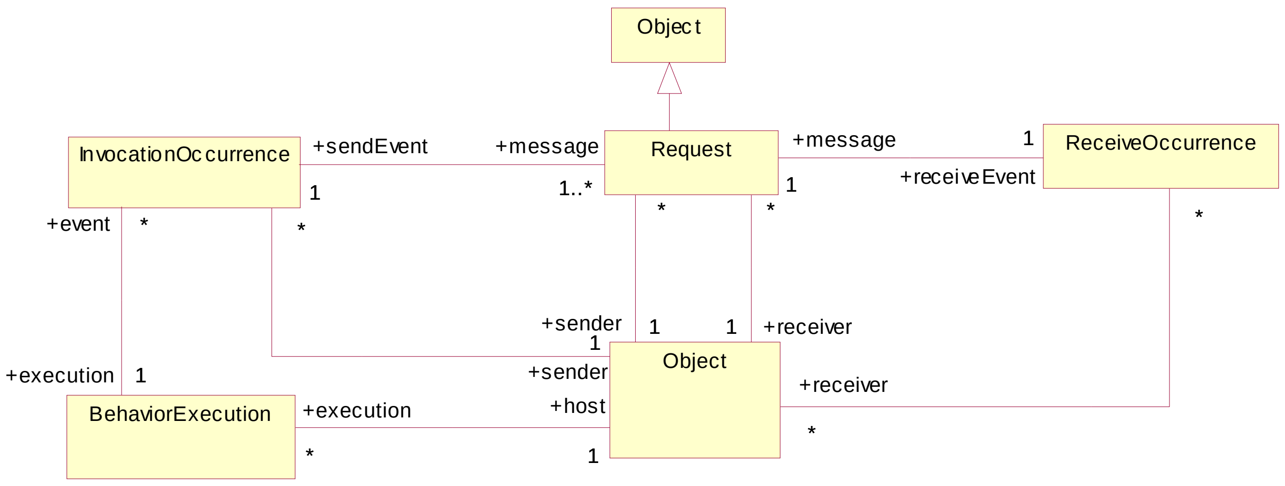


Figure 13.3 - Communication Domain Model

Several kinds of requests exist between instances, for example, sending a signal or invoking an operation. The kind of request is determined by the kind of invocation occurrence that caused it, as shown in Figure 13.4. The former is used to trigger a reaction in the receiver in an asynchronous way and without a reply, while the latter applies an operation to an instance, which may be either synchronous or asynchronous and may require a reply from the receiver to the sender. A send invocation occurrence creates a send request and causes a signal occurrence in the receiver. A call invocation occurrence creates a call request and causes a call occurrence in the receiver.

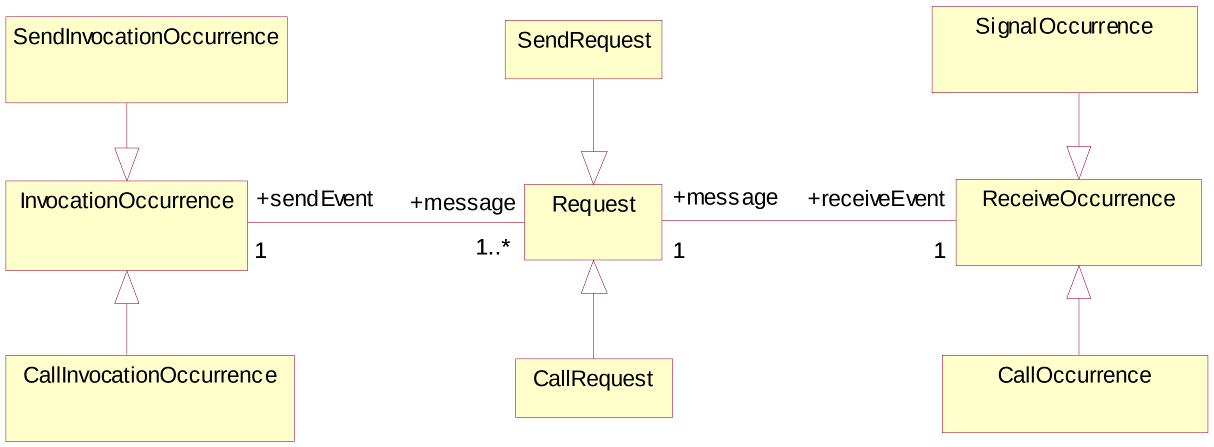


Figure 13.4 - Domain Model Showing Request Kinds

An invocation occurrence represents the recognition of an invocation request after its receipt by a target object. Invocation event occurrences are the result of the execution of invocation actions (see “InvocationAction (from BasicActions)” on page 256). Invocation actions include send actions and call actions. A send action is specified by a Signal (see “Signal (from Communications)” on page 449) and argument values. The execution of a send action results in a send request, which results in a signal event occurrence when it is recognized by the target object. A call action is specified by an Operation and argument values. The execution of a call action results in a call request, which results in a call event occurrence when it is recognized by the target object. Signal event occurrences and call event occurrences are specified by the corresponding metaclasses (see “SignalEvent (from Communications)” on page 450 and “CallEvent (from Communications)” on page 436).

As shown in Figure 13.3, an object hosts a behavior execution (i.e., a behavior will be executed in the context of that object). The execution of an invocation action by the behavior constitutes an invocation occurrence. The invocation occurrence results in a request object that transmits the invocation request from the sender object (caller) to the receiver object (target). The receipt of the request by the receiver is manifest as a receive occurrence. When the receive occurrence matches a trigger defined in the class of the target object, it causes the execution of a behavior. The details of identifying the behavior to be invoked in response to the occurrence of an event are a semantic variation point. The resulting behavior execution is hosted by the target object. The specific mechanism by which the data passed with the request (the attributes of the request object) are made available as arguments to the invoked behavior (e.g., whether the data or copies are passed with the request) is a semantic variation point. If the invocation action is synchronous, the request object also includes sufficient information to identify the execution that invoked the behavior, but this information is not available for the use of the invoked behavior (and, therefore, is not modeled). When a synchronous execution completes, this information is used to direct a reply message to the original behavior execution.

The detection of an (event) occurrence by an object may cause a behavioral response. For example, a state machine may transition to a new state upon the detection of the occurrence of an event specified by a trigger owned by the state machine, or an activity may be enabled upon the receipt of a message. When an event occurrence is recognized by an object, it may have an immediate effect or the event may be saved in an event pool and have a later effect when it is matched by a trigger specified for a behavior.

The occurrence of a change event (see “ChangeEvent (from Communications)” on page 437) is based on some expression becoming true. A time event occurs when a predetermined deadline expires (see “TimeEvent (from Communications, SimpleTime)” on page 453). No data is passed by the occurrence of a change event or a time event. Figure 13.12 shows the hierarchy of events.

SimpleTime

The SimpleTime subpackage of the Common Behavior package adds metaclasses to represent time and durations, as well as actions to observe the passing of time.

The simple model of time described here is intended as an approximation for situations where the more complex aspects of time and time measurement can safely be ignored. For example, this model does not account for the relativistic effects that occur in many distributed systems, or 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.

13.2 Abstract Syntax

Figure 13.5 shows the dependencies of the CommonBehaviors packages.

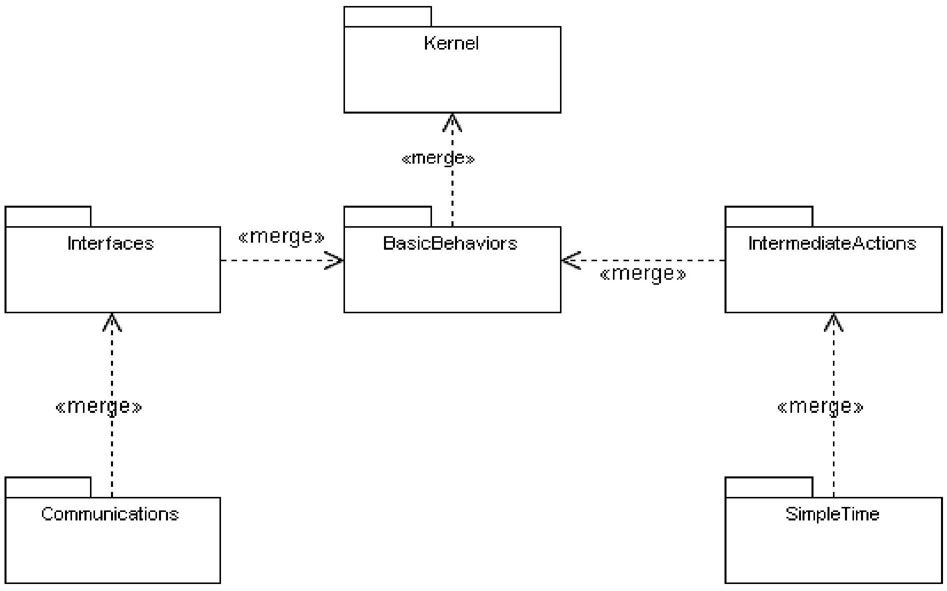


Figure 13.5 - Dependencies of the CommonBehaviors packages

Package BasicBehaviors

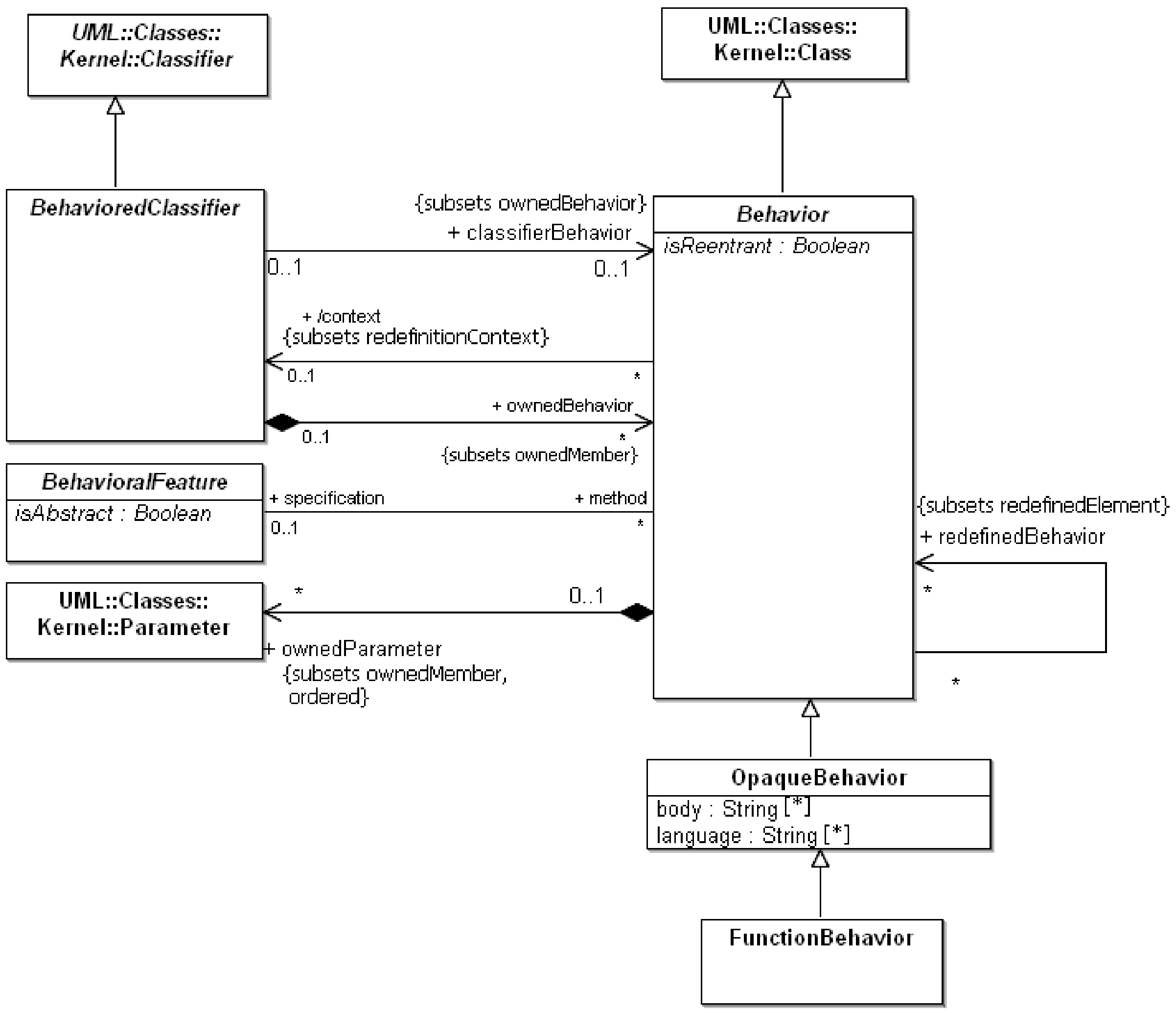
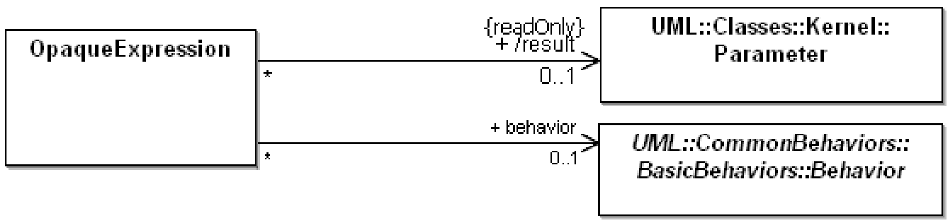


Figure 13.6 - Common Behavior



OpaqueExpression
{readOnly}
+ /result
*
0..1
UML::Classes::Kernel::Parameter
+ behavior
*
0..1
UML::CommonBehaviors::BasicBehaviors::Behavior

Figure 13.7 - Expression

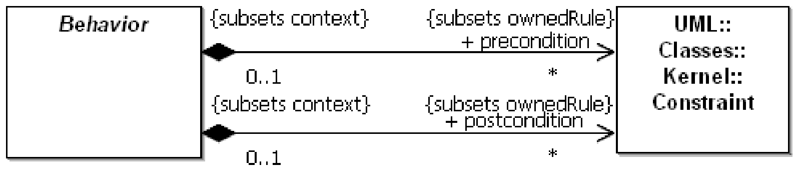


Figure 13.8 - Precondition and postcondition constraints for behavior

Package Communications

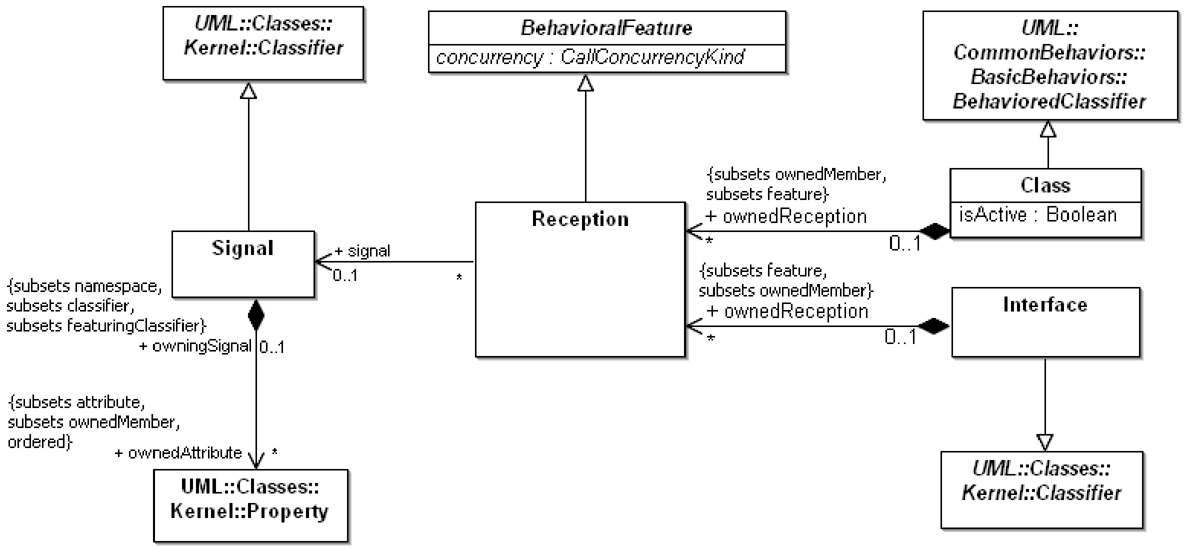


Figure 13.9 - Reception

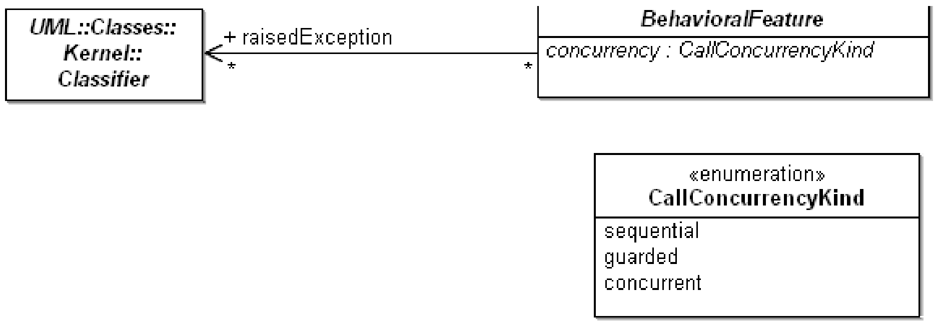
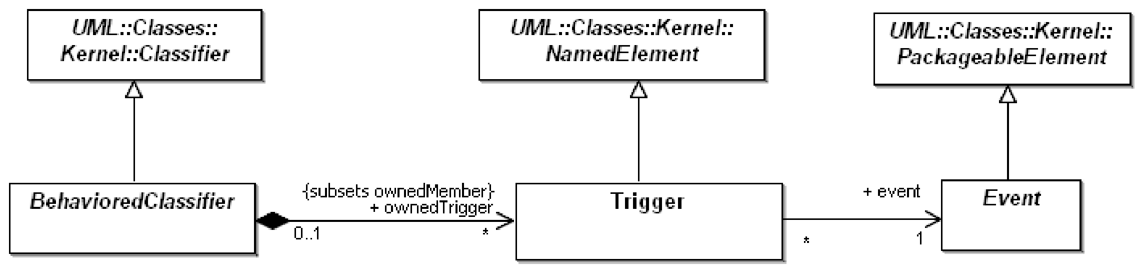


Figure 13.10 - Extensions to behavioral features



UML::Classes::Kernel::Classifier
UML::Classes::Kernel::NamedElement
UML::Classes::Kernel::PackageableElement
BehavioredClassifier
{subsets ownedMember}
+ ownedTrigger
0..1
*
Trigger
+ event
*
1
Event

Figure 13.11 - Triggers

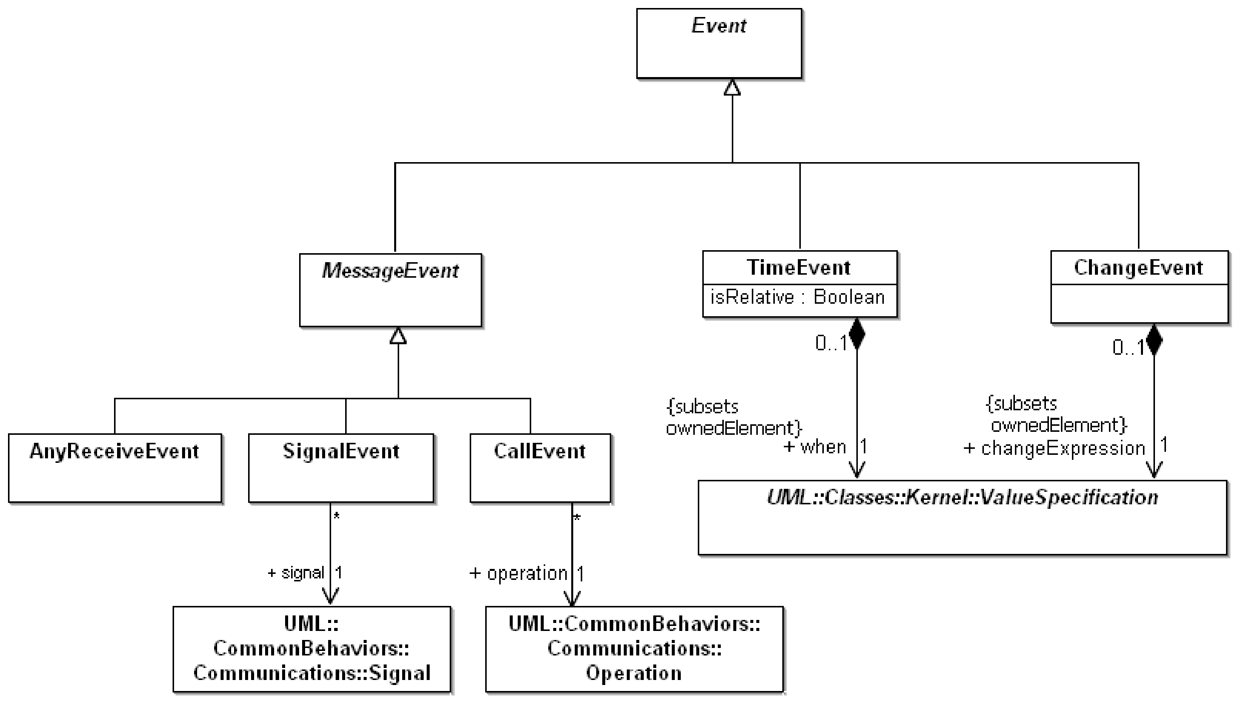


Figure 13.12 - Events

SimpleTime

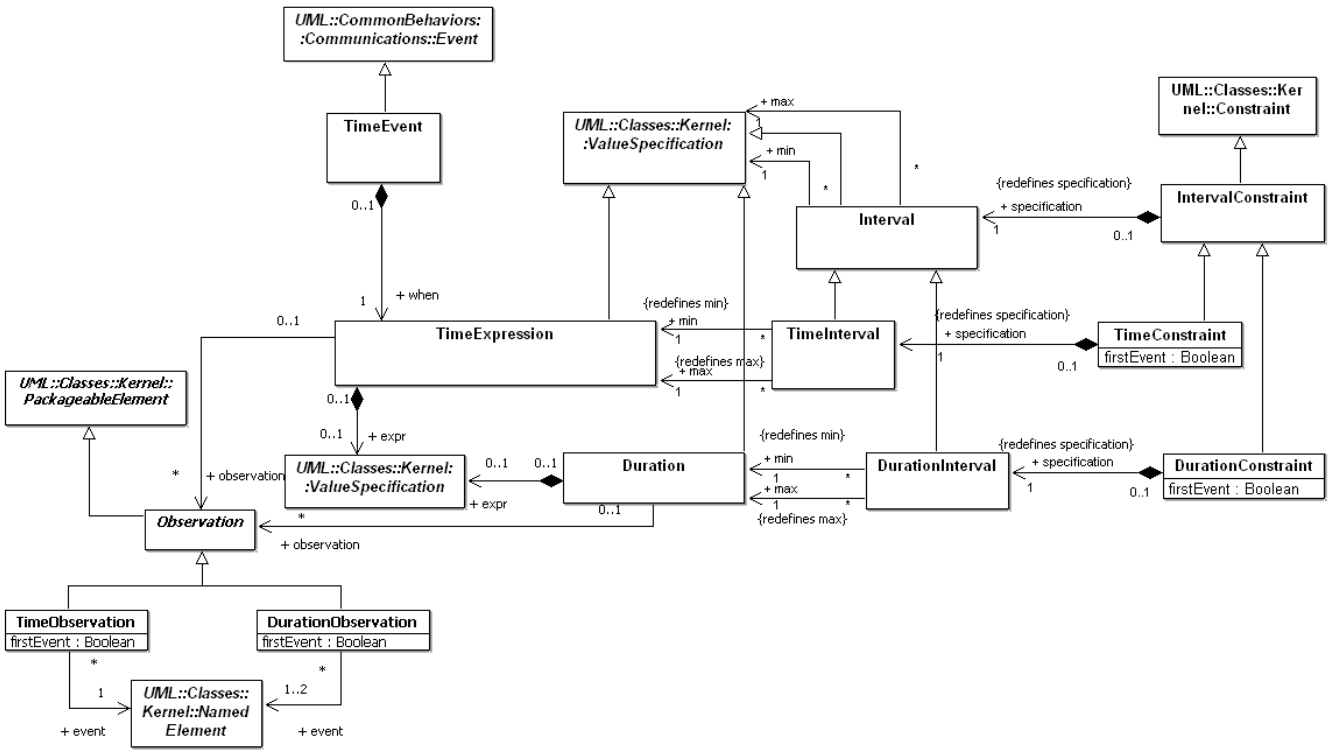


Figure 13.13 - SimpleTime

13.3 Class Descriptions

13.3.1 AnyReceiveEvent (from Communications)

Generalizations

- “MessageEvent (from Communications)” on page 445

Description

A transition trigger associated with AnyReceiveEvent specifies that the transition is to be triggered by the receipt of any message that is not explicitly referenced in another transition from the same vertex.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

An AnyReceiveEvent associated with a transition trigger specifies that the transition is triggered for all applicable message receive events except for those specified explicitly on other transitions having the same vertex as a source.

Notation

Any AnyReceiveEvent is denoted by the string “all” used as the trigger.

<any-receive-event> ::= ‘all’

Changes from previous UML

This construct has been added.

13.3.2 Behavior (from BasicBehaviors)

Generalizations

- “Class (from Kernel)” on page 49

Description

Behavior is a specification of how its context classifier 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.

A classifier behavior is always a definition of behavior and not an illustration. It describes the sequence of state changes an instance of a classifier may undergo in the course of its lifetime. Its precise semantics depends on the kind of classifier. For example, the classifier behavior of a collaboration represents emergent behavior of all the parts, whereas the classifier behavior of a class is just the behavior of instances of the class separated from the behaviors of any of its parts.

When a behavior is associated as the method of a behavioral feature, it defines the implementation of that feature (i.e., the computation that generates the effects of the behavioral feature).

As a classifier, a behavior can be specialized. Instantiating a behavior is referred to as “invoking” the behavior, an instantiated behavior is also called a behavior “execution.” A behavior may be invoked directly or its invocation may be the result of invoking the behavioral feature that specifies this behavior. A behavior can also be instantiated as an object in virtue of it being a class.

The specification of a behavior can take a number of forms, as described in the subclasses of Behavior. Behavior is an abstract metaclass factoring out the commonalities of these different specification mechanisms.

When a behavior is invoked, its execution receives a set of input values that are used to affect the course of execution, and as a result of its execution it produces a set of output values that are returned, as specified by its parameters. The observable effects of a behavior execution may include changes of values of various objects involved in the execution, the creation and destruction of objects, generation of communications between objects, as well as an explicit set of output values.

Attributes

- isReentrant: Boolean [1]
  Tells whether the behavior can be invoked while it is still executing from a previous invocation. Default value is *false*.

Associations

- specification: BehavioralFeature [0..1]
  Designates a behavioral feature that the behavior implements. The behavioral feature must be owned by the classifier that owns the behavior or be inherited by it. The parameters of the behavioral feature and the implementing behavior must match. A behavior does not need to have a specification, in which case it either is the classifier behavior of a BehavioredClassifier or it is a behavior that can only invoked by another behavior of the classifier.
- /context: BehavioredClassifier [0..1]
  The classifier that is the context for the execution of the behavior. If the behavior is owned by a BehavioredClassifier, that classifier is the context; otherwise, the context is the first BehavioredClassifier reached by following the chain of owner relationships. For example, following this algorithm, the context of an entry action in a state machine is the classifier that owns the state machine. The features of the context classifier as well as the elements visible to the context classifier are visible to the behavior. (Subsets *RedefinableElement::redefinitionContext*)
- ownedParameter: Parameter
  References a list of parameters to the behavior that describes the order and type of arguments that can be given when the behavior is invoked and of the values that will be returned when the behavior completes its execution. (Subsets *Namespace::ownedMember*)
- redefinedBehavior: Behavior
  References a behavior that this behavior redefines. A subtype of Behavior may redefine any other subtype of Behavior. If the behavior implements a behavioral feature, it replaces the redefined behavior. If the behavior is a classifier behavior, it extends the redefined behavior. (Subsets *RedefineableElement::redefinedElement*)
- precondition: Constraint
  An optional set of Constraints specifying what must be fulfilled when the behavior is invoked. (Subsets *Namespace::ownedRule*)
- postcondition: Constraint
  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. (Subsets *Namespace::ownedRule*)

Constraints

[1] The parameters of the behavior must match the parameters of the implemented behavioral feature.

[2] The implemented behavioral feature must be a feature (possibly inherited) of the context classifier of the behavior.

[3] If the implemented behavioral feature has been redefined in the ancestors of the owner of the behavior, then the behavior must realize the latest redefining behavioral feature.

[4] There may be at most one behavior for a given pairing of classifier (as owner of the behavior) and behavioral feature (as specification of the behavior).

Semantics

The detailed semantics of behavior is determined by its subtypes. The features of the context classifier and elements that are visible to the context classifier are also visible to the behavior, provided that is allowed by the visibility rules.

When a behavior is invoked, its attributes and parameters (if any) are created and appropriately initialized. Upon invocation, the arguments of the original invocation action are made available to the new behavior execution corresponding to its parameters with direction 'in' and 'inout,' if any. When a behavior completes its execution, a value or set of values is returned corresponding to each parameter with direction 'out,' 'inout,' or 'return,' if any. If such a parameter has a default value associated and the behavior does not explicitly generate a value for this parameter, the default value describes the value that will be returned corresponding to this parameter. If the invocation was synchronous, any return values from the behavior execution are returned to the original caller, which is unblocked and allowed to continue execution.

The behavior executes within its context object, independently of and concurrently with any existing behavior executions. The object that is the context of the behavior manages the input pool holding the event occurrences to which a behavior may respond (see 13.3.4, "BehavioredClassifier (from BasicBehaviors, Communications)," on page 434). As an object may have a number of behaviors associated, all these behaviors may access the same input pool. The object ensures that each event occurrence on the input pool is consumed by only one behavior.

When a behavior is instantiated as an object, it is its own context.

Semantic Variation Points

The means by which requests are transported to their target depend on the type of requesting action, the target, the properties of the communication medium, and numerous other factors. In some cases, this is instantaneous and completely reliable while in others it may involve transmission delays of variable duration, loss of requests, reordering, or duplication (see also the discussion on page 423).

How the parameters of behavioral features or a behavior match the parameters of a behavioral feature is a semantic variation point (see BehavioralFeature on page 432).

Notation

None

Changes from previous UML

This metaclass has been added. It abstracts the commonalities between the various ways that behavior can be implemented in the UML. It allows the various ways of implementing behavior (as expressed by the subtypes of Behavior) to be used interchangeably.

13.3.3 BehavioralFeature (from BasicBehaviors, Communications)

Generalizations

- "BehavioralFeature (from Kernel)" on page 48 *(merge increment)*

Description

A behavioral feature is implemented (realized) by a behavior. A behavioral feature specifies that a classifier will respond to a designated request by invoking its implementing method.

Attributes

Package BasicBehaviors

- isAbstract: Boolean
  If *true*, then the behavioral feature does not have an implementation, and one must be supplied by a more specific element. If *false*, the behavioral feature must have an implementation in the classifier or one must be inherited from a more general element. Default value is *false*.

Package Communications

- concurrency: CallConcurrencyKind
  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 behavioral features. Default value is *sequential*.

Associations

Package BasicBehaviors

- method: Behavior [0..*]
  A behavioral description that implements the behavioral feature. There may be at most one behavior for a particular pairing of a classifier (as owner of the behavior) and a behavioral feature (as specification of the behavior).

Package Communications

- raisedException: Classifier [0..*]
  The signals that the behavioral feature raises as exceptions. (Subsets *BehavioralFeature::raisedException*)

Constraints

No additional constraints

Semantics

The invocation of a method is caused by receiving a request invoking the behavioral feature specifying that behavior. The details of invoking the behavioral feature are defined by the subclasses of BehavioralFeature.

Semantic Variation Points

How the parameters of behavioral features or a behavior match the parameters of a behavioral feature is a semantic variation point. Different languages and methods rely on exact match (i.e., the type of the parameters must be the same), co-variant match (the type of a parameter of the behavior may be a subtype of the type of the parameter of the behavioral feature), contra-variant match (the type of a parameter of the behavior may be a supertype of the type of the parameter of the behavioral feature), or a combination thereof.

Changes from previous UML

The metaattributes *isLeaf* and *isRoot* have been replaced by properties inherited from RedefinableElement.

13.3.4 BehavioredClassifier (from BasicBehaviors, Communications)

Generalizations

- “Classifier (from Kernel, Dependencies, PowerTypes)” on page 52

Description

A classifier can have behavior specifications defined in its namespace. One of these may specify the behavior of the classifier itself.

Attributes

No additional attributes

Associations

Package BasicBehaviors

- ownedBehavior: Behavior [0..*]
  References behavior specifications owned by a classifier. (Subsets *Namespace::ownedMember*)
- classifierBehavior: Behavior [0..1]
  A behavior specification that specifies the behavior of the classifier itself. (Subsets *BehavioredClassifier::ownedBehavior*)

Package Communications

- ownedTrigger : Trigger [0..*]
  References Trigger descriptions owned by a Classifier (Subsets *Namespace::ownedMember*)

Constraints

[1] If a behavior is classifier behavior, it does not have a specification.

self.classifierBehavior->notEmpty() **implies** self.specification->isEmpty()

Semantics

The behavior specifications owned by a classifier are defined in the context of the classifier. Consequently, the behavior specifications may reference features of the classifier. Any invoked behavior may, in turn, invoke other behaviors visible to its context classifier. When an instance of a behaviored classifier is created, its classifier behavior is invoked.

When an event occurrence is recognized by an object that is an instance of a behaviored classifier, it may have an immediate effect or the occurrence may be saved for later *triggered* effect. An immediate effect is manifested by the invocation of a behavior as determined by the event (the type of the occurrence). A triggered effect is manifested by the storage of the occurrence in the input event pool of the object and the later consumption of the occurrence by the execution of an ongoing behavior that reaches a point in its execution at which a trigger matches the event (type) of the occurrence in the pool. At this point, a behavior may be invoked as determined by the event.

When an executing behavior owned by an object comes to a point where it needs a trigger to continue its execution, the input pool is examined for an event that satisfies the outstanding trigger or triggers. If an event satisfies one of the triggers, the event is removed from the input pool and the behavior continues its execution, as specified. Any data associated with the event are made available to the triggered behavior.

Semantic Variation Points

It is a semantic variation whether one or more behaviors are triggered when an event satisfies multiple outstanding triggers.

If an event in the pool satisfies no triggers at a wait point, it is a semantic variation point what to do with it.

The ordering of the events in the input pool is a semantic variation.

Notation

See "Classifier (from Kernel, Dependencies, PowerTypes)" on page 52.

Changes from previous UML

In UML 1.4, there was no separate metaclass for classifiers with behavior.

13.3.5 CallConcurrencyKind (from Communications)

Generalizations

None

Description

CallConcurrencyKind is an enumeration with the following literals:

- sequential - No concurrency management mechanism is associated with the operation and, therefore, concurrency conflicts may occur. Instances that invoke a behavioral feature need to coordinate so that only one invocation to a target on any behavioral feature occurs at once.
- guarded - Multiple invocations of a behavioral feature may occur simultaneously to one instance, but only one is allowed to commence. The others are blocked until the performance of the currently executing behavioral feature 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 behavioral feature may occur simultaneously to one instance and all of them may proceed concurrently.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See Description sub clause above.

Notation

None

Changes from previous UML

None

13.3.6 CallEvent (from Communications)

A CallEvent models the receipt by an object of a message invoking a call of an operation.

Generalizations

- “MessageEvent (from Communications)” on page 445

Description

A call event represents the *reception* of a request to invoke a specific operation. A call event is distinct from the call action that caused it. A call event may cause the invocation of a behavior that is the method of the operation referenced by the call request, if that operation is owned or inherited by the classifier that specified the receiver object.

Attributes

No additional attributes

Associations

- operation: Operation [1]
  Designates the operation whose invocation raised the call event.

Constraints

No additional constraints

Semantics

A call event represents the reception of a request to invoke a specific operation on an object. The call event may result in the execution of the behavior that implements the called operation. A call event may, in addition, cause other responses, such as a state machine transition, as specified in the classifier behavior of the classifier that specified the receiver object. In that case, the additional behavior is invoked after the completion of the operation referenced by the call event.

A call event makes available any argument values carried by the received call request to all behaviors caused by this event (such as transition actions or entry actions).

Notation

Call events are denoted by a list of names of the triggering operations, followed by an assignment specification:

> *<call-event> :: <name> [‘(‘ [<assignment-specification>] ‘)’]*
> *<assignment-specification> ::= <attr-name> [‘,’ <attr-name>]**

where:

- *<attr-name>* is an implicit assignment of the corresponding parameter of the operation to an attribute (with this name) of the context object owning the triggered behavior.

<assignment-specification> is optional and may be omitted even if the operation has parameters.

13.3.7 ChangeEvent (from Communications)

A change event models a change in the system configuration that makes a condition true.

Generalizations

- "Event (from Communications)" on page 442

Description

A change event occurs when a Boolean-valued expression becomes true. For example, as a result of a change in the value held in a slot corresponding to an attribute, or a change in the value referenced by a link corresponding to an association. A change event is raised implicitly and is *not* the result of an explicit action.

Attributes

No additional attributes

Associations

- changeExpression: Expression [1]
  A Boolean-valued expression that will result in a change event whenever its value changes from *false* to *true*.
  {Subsets *Element::ownedElement*}

Constraints

No additional constraints

Semantics

Each time the value of the change expression changes from false to true, a change event is generated.

Semantic Variation Points

It is a semantic variation when the change expression is evaluated. For example, the change expression may be continuously evaluated until it becomes *true*. It is further a semantic variation whether a change event remains until it is consumed, even if the change expression changes to *false* after a change event.

Notation

A change event is denoted in a trigger by a Boolean expression.

<change-event> ::= 'when' <expression>

Changes from previous UML

This metaclass replaces change event.

13.3.8 Class (from Communications)

Generalizations

- "BehavioredClassifier (from BasicBehaviors, Communications)" on page 434
- "Class (from Kernel)" on page 49 *(merge increment)*

Description

A class may be designated as active (i.e., each of its instances having its own thread of control) or passive (i.e., each of its instances executing within the context of some other object).

A class may also specify which signals the instances of this class handle.

Attributes

- isActive: Boolean
  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*. Default value is *false*.

Associations

- ownedReception: Reception [0..*]
  Receptions that objects of this class are willing to accept. (Subsets *Namespace::ownedMember* and *Classifier::feature*)

Constraints

[1] A passive class cannot have receptions.

(**not** self.isActive) **implies** self.ownedReception->isEmpty()

Semantics

An active object is an object that, as a direct consequence of its creation, commences to execute its classifier behavior, 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 classifier behavior of an active object completes, the object is terminated.

Notation

See presentation options below.

Presentation options

A class with the property *isActive* = *true* can be shown by a class box with an additional vertical bar on either side, as depicted in Figure 13.14.

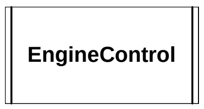


Figure 13.14 - Active class

13.3.9 Duration (from SimpleTime)

Generalizations

- "ValueSpecification (from Kernel)" on page 137.

Description

A duration defines a value specification that specifies the temporal distance between two time instants.

Associations

- observation : Observation[*]
  Refers to the time and duration observations that are involved in expr
- expr : ValueSpecification [0..1]
  The value of the Duration.

Constraints

No additional constraints.

Semantics

A Duration defines a ValueSpecification that denotes some duration in time. The duration is given by the difference in time between a starting point in time and an ending point in time.

Notation

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.

Changes from previous UML

This metaclass has been added.

13.3.10 DurationConstraint (from SimpleTime)

Generalizations

- “IntervalConstraint (from SimpleTime)” on page 444

Description

A DurationConstraint defines a Constraint that refers to a DurationInterval.

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]. Default value is true applied when constrainedElement[i] refers an element that represents only one time instant.

Associations

- specification: DurationInterval [1]
  A duration used to determine whether the constraint is satisfied. Redefines *IntervalConstraint::specification.*

Constraints

[1] The multiplicity of firstEvent must be 2 if the multiplicity of constrainedElement is 2; otherwise, the multiplicity of firstEvent is 0. (The constraint is a requirement on the duration from the execution time from (constrainedElement[1], firstEvent[1]) to (constrainedElement[2], firstEvent[2]). If the multiplicity of constrainedElement is 1, then the constraint is a requirement on the duration given by the duration of the execution of that constrainedElement.)

if (constrainedElement->size() = 2) **then** (firstEvent->size() = 2) **else** (firstEvent->size() = 0)

Semantics

The semantics of a DurationConstraint is inherited from Constraints. All traces where the constraints are violated are negative traces i.e., if they occur in practice the system has failed.

Notation

A DurationConstraint is shown as some graphical association between a DurationInterval and the constructs that it constrains. The notation is specific to the diagram type.

Examples

See example in Figure 13.15 on page 440 where the TimeConstraint is associated with the duration of a Message and the duration between two OccurrenceSpecifications.

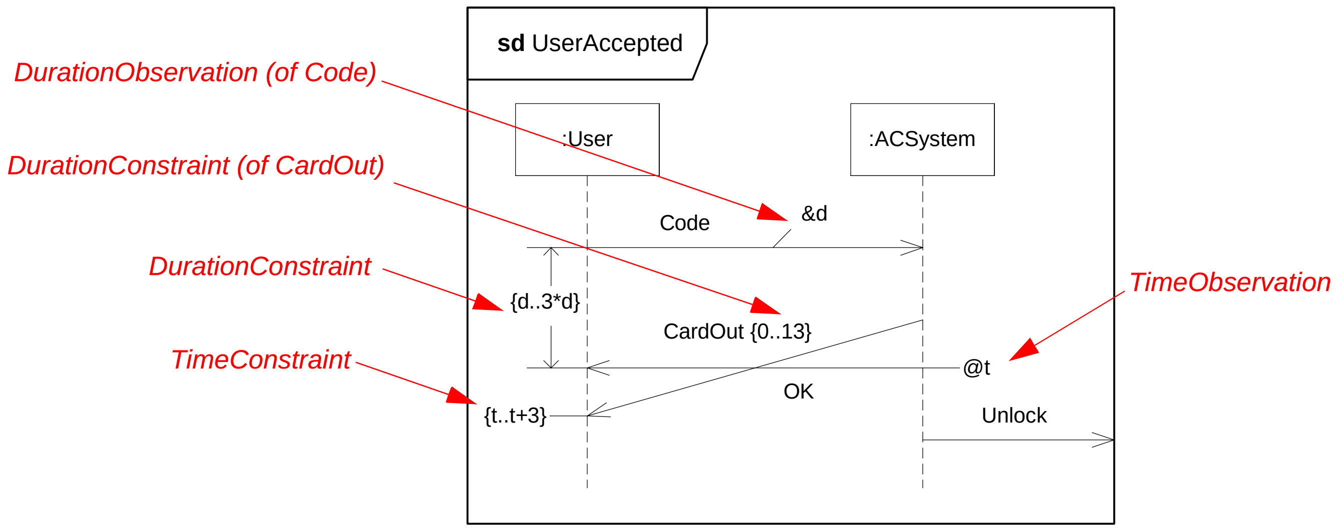


Figure 13.15 - DurationConstraint and other time-related concepts

Changes from previous UML

This metaclass has been added.

13.3.11 DurationInterval (from SimpleTime)

Generalizations

- "Interval (from SimpleTime)" on page 444

Description

A DurationInterval defines the range between two Durations.

Attributes

No additional attributes

Associations

- min: Duration [1]
  Refers to the Duration denoting the minimum value of the range.
- max: Duration [1]
  Refers to the Duration denoting the maximum value of the range.

Constraints

No additional constraints

Semantics

None

Notation

A DurationInterval is shown using the notation of Interval where each value specification element is a DurationExpression.

13.3.12 DurationObservation (from SimpleTime)

Generalizations

- "Observation (from SimpleTime)" on page 446

Description

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

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]. Default value is true applied when event[i] refers an element that represents only one time instant.

Associations

- event:NamedElement[1..2]
  The observation is determined by the entering or exiting of the event element during execution.

Constraints

[1] The multiplicity of firstEvent must be 2 if the multiplicity of event is 2; otherwise, the multiplicity of firstEvent is 0.

if (event->size() = 2) **then** (firstEvent->size() = 2) **else** (firstEvent->size() = 0)

Semantics

A duration observation denotes some interval of time.

Notation

A duration observation is often denoted by a straight line attached to a model element. The observation is given a name that is shown close to the unattached end of the line.

13.3.13 Event (from Communications)

Generalizations

- “PackageableElement (from Kernel)” on page 109

Description

An event is the specification of some occurrence that may potentially trigger effects by an object.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

An event is the specification of some occurrence that may potentially trigger effects by an object. This is an abstract metaclass.

Notation

None

Changes from previous UML

None

13.3.14 FunctionBehavior (from BasicBehaviors)

Generalizations

- “OpaqueBehavior (from BasicBehaviors)” on page 446

Description

A function behavior is an opaque behavior that does not access or modify any objects or other external data.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] A function behavior has at least one output parameter.

```
(self.ownedParameters->notEmpty) and self.ownedParameters->exists (p|
    (p.direction = ParameterDirectionKind::out) or
    (p.direction = ParameterDirectionKind::inout) or
    (p.direction = ParameterDirectionKind::return))
```

[2] The types of parameters are all data types, which may not nest anything but other datatypes.

Semantics

Primitive functions transform a set of input values to a set of output values by invoking a function. They represent functions from a set of input values to a set of output values. The execution of a primitive function depends only on the input values and has no other effect than to compute output values. A primitive function does not read or write structural feature or link values, nor otherwise interact with object memory or other objects. Its behavior is completely self-contained. Specific primitive functions are not defined in the UML, but would be defined in domain-specific extensions. Typical primitive functions would include arithmetic, Boolean, and string functions.

During the execution of the function, no communication or interaction with the rest of the system is possible. The amount of time to compute the results is undefined. FunctionBehavior may raise exceptions for certain input values, in which case the computation is abandoned.

Notation

None

Examples

Mathematical functions are examples of function behaviors.

Rationale

FunctionBehavior is introduced to model external functions that only take inputs and produce outputs and have no effect on the specified system.

13.3.15 Interface (from Communications)

Generalizations

- “Interface (from Interfaces)” on page 86 *(merge increment)*

Description

Adds the capability for interfaces to include receptions (in addition to operations).

Associations

- ownedReception: Reception [0..*]
  Receptions that objects providing this interface are willing to accept. (Subsets *Namespace::ownedMember* and *Classifier::feature*)

13.3.16 Interval (from SimpleTime)

Generalizations

- "ValueSpecification (from Kernel)" on page 137

Description

An Interval defines the range between two value specifications.

Attributes

No additional attributes

Associations

- min: ValueSpecification[1]
  Refers to the ValueSpecification denoting the minimum value of the range.
- max: ValueSpecification[1]
  Refers to the ValueSpecification denoting the maximum value of the range.

Constraints

No additional constraints

Semantics

The semantics of an Interval is always related to Constraints in which it takes part.

Notation

An Interval is denoted textually by two ValueSpecifications separated by "..":

<interval> ::= <min-value> '..' <max-value>

Changes from previous UML

This metaclass has been added.

13.3.17 IntervalConstraint (from SimpleTime)

Generalizations

- "Constraint (from Kernel)" on page 58

Description

An IntervalConstraint defines a Constraint that refers to an Interval.

Attributes

No additional attributes

Associations

- specification : Interval [1]
  An interval that determines if the constraint is satisfied. Redefines *Constraint::specification*.

Constraints

No additional constraints

Semantics

The semantics of an IntervalConstraint is inherited from Constraint. All traces where the constraints are violated are negative traces (i.e., if they occur in practice the system has failed).

Notation

An IntervalConstraint is shown as a graphical association between an Interval and the constructs that this Interval constrains. The concrete form is given in its subclasses.

Changes from previous UML

This metaclass has been added.

13.3.18 MessageEvent (from Communications)

Generalizations

- “Event (from Communications)” on page 442

Description

A message event specifies the receipt by an object of either a call or a signal. MessageEvent is an abstract metaclass.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

No additional semantics

Notation

None

Changes from previous UML

The metaclass has been added.

13.3.19 Observation (from SimpleTime)

Generalizations

- "PackageableElement (from Kernel)" on page 109

Description

Observation is an abstract superclass of TimeObservation and DurationObservation into allow TimeExpressions and Durations to refer to either in a common way.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

No additional semantics

Notation

None

Changes from previous UML

The metaclass has been added.

13.3.20 OpaqueBehavior (from BasicBehaviors)

Generalizations

- "Behavior (from BasicBehaviors)" on page 430

Description

A behavior with implementation-specific semantics.

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.

Associations

No additional associations

Constraints

No additional constraints

Semantics

The semantics of the behavior is determined by the implementation.

Notation

None

Rationale

OpaqueBehavior is introduced for implementation-specific behavior or for use as a place-holder before one of the other behaviors is chosen.

13.3.21 OpaqueExpression (from BasicBehaviors)

Generalizations

- "OpaqueExpression (from Kernel)" on page 101 *(merge increment)*

Description

Provides a mechanism for precisely defining the behavior of an opaque expression. An opaque expression is defined by a behavior restricted to return one result.

Attributes

No additional attributes

Associations

- behavior: Behavior [0..1]
  Specifies the behavior of the opaque expression.
- result: Parameter [0..1]
  Restricts an opaque expression to return exactly one return result. When the invocation of the opaque expression completes, a single set of values is returned to its owner. This association is derived from the single return result parameter of the associated behavior.

Constraints

[1] The *behavior* can only have return result parameters.

```
self.behavior.notEmpty() implies
    (self.behavior.ownedParameters->select(p | p.direction <> ParameterDirectionKind::return))->isEmpty()
```

[2] The *behavior* must have exactly one return result parameter.

```
self.behavior.notEmpty() implies
    (self.behavior.ownedParameter->select(p | p.direction = ParameterDirectionKind::return))->size() = 1
```

Semantics

An opaque expression is invoked by the execution of its owning element. An opaque expression does not have formal parameters and thus cannot be passed data upon invocation. It accesses its input data through elements of its behavioral description. Upon completion of its execution, a single value or a single set of values is returned to its owner.

13.3.22 Operation (from Communications)

Generalizations

- "Operation (from Kernel, Interfaces)" on page 103 *(merge increment)*

Description

An operation may invoke both the execution of method behaviors as well as other behavioral responses.

Semantics

If an operation is not mentioned in a trigger of a behavior owned or inherited by the behaviored classifier owning the operation, then upon occurrence of a call event (representing the receipt of a request for the invocation of this operation) a resolution process is performed that determines the method behavior to be invoked, based on the operation and the data values corresponding to the parameters of the operation transmitted by the request; otherwise, the call event is placed into the input pool of the object (see BehavioredClassifier on page 434). If a behavior is triggered by this event, it begins with performing the resolution process and invoking the so determined method. Then the behavior continues its execution as specified.

Operations specify immediate or triggered effects (see "BehavioredClassifier" on page 434).

Semantic Variation Points

Resolution specifies how a particular behavior is identified to be executed in response to the invocation of an operation, using mechanisms such as inheritance. The mechanism by which the behavior to be invoked is determined from an operation and the transmitted argument data is a semantic variation point. In general, this mechanism may be complicated to include languages with features such as before-after methods, delegation, etc. In some of these variations, multiple behaviors may be executed as a result of a single call. The following defines a simple object-oriented process for this semantic variation point.

- Object-oriented resolution

  When a call request is received, the class of the target object is examined for an owned operation with matching parameters (see "BehavioralFeature" on page 432). If such operation is found, the behavior associated as *method* is the result of the resolution; otherwise the parent classifier is examined for a matching 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 parents, all of them are examined for a method. If a method is found in exactly one ancestor class, then that method is the result of the resolution. If a method is found in more than one ancestor class along different paths, then the model is ill formed under this semantic variation.

If no method by the resolution process, then it is a semantic variation point what is to happen.

13.3.23 Reception (from Communications)

Generalizations

- "BehavioralFeature (from Kernel)" on page 48

Description

A reception is a declaration stating that a classifier is prepared to react to the receipt of a signal. A reception designates a signal and specifies the expected behavioral response. The details of handling a signal are specified by the behavior associated with the reception or the classifier itself.

Attributes

No additional attributes

Associations

- signal: Signal [0..1]
  The signal that this reception handles.

Constraints

[1] A Reception cannot be a query.
`not self.isQuery`

Semantics

The receipt of a signal instance by the instance of the classifier owning a matching reception will cause the asynchronous invocation of the behavior specified as the method of the reception. A reception matches a signal if the received signal is a subtype of the signal referenced by the reception. The details of how the behavior responds to the received signal depend on the kind of behavior associated with the reception. (For example, if the reception is implemented by a state machine, the signal event will trigger a transition and subsequent effects as specified by that state machine.)

Receptions specify triggered effects (see “BehavioredClassifier” on page 434).

Notation

Receptions are shown using the same notation as for operations with the keyword «signal», as shown in Figure 13.16.

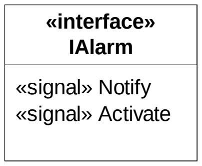


Figure 13.16 - Showing signal receptions in classifiers

Changes from previous UML

None

13.3.24 Signal (from Communications)

Generalizations

- “Classifier (from Kernel, Dependencies, PowerTypes)” on page 52

Description

A signal is a specification of send request instances communicated between objects. The receiving object handles the received request instances as specified by its receptions. The data carried by a send request (which was passed to it by the send invocation occurrence that caused that request) are represented as attributes of the signal. A signal is defined independently of the classifiers handling the signal occurrence.

Attributes

No additional attributes

Associations

- ownedAttribute: Property [0..*]
  The attributes owned by the signal. (Subsets *Classifier::attribute, Namespace::ownedMember*). This association end is ordered.

Constraints

No additional constraints

Semantics

A signal triggers a reaction in the receiver in an asynchronous way and without a reply. 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.

Notation

A signal is depicted by a classifier symbol with the keyword «signal».

Changes from previous UML

None

13.3.25 SignalEvent (from Communications)

A signal event represents the *receipt* of an asynchronous signal instance. A signal event may, for example, cause a state machine to trigger a transition.

Generalizations

- "MessageEvent (from Communications)" on page 445

Description

A signal event represents the *receipt* of an asynchronous signal. A signal event may cause a response, such as a state machine transition as specified in the classifier behavior of the classifier that specified the receiver object, if the signal referenced by the send request is mentioned in a reception owned or inherited by the classifier that specified the receiver object.

Attributes

- signal: Signal [1]
  The specific signal that is associated with this event.

Associations

No additional associations

Constraints

No additional constraints

Semantics

A signal event occurs when a signal message, originally caused by a send action executed by some object, is received by another (possibly the same) object. A signal event may result in the execution of the behavior that implements the reception matching the received signal.

A signal event makes available any argument values carried by the received send request to all behaviors caused by this event (such as transition actions or entry actions).

Semantic Variation Points

The means by which requests are transported to their target depend on the type of requesting action, the target, the properties of the communication medium, and numerous other factors. In some cases, this is instantaneous and completely reliable while in others it may involve transmission delays of variable duration, loss of requests, reordering, or duplication. (See also the discussion on page 423.)

Notation

Signal events are denoted by a list of names of the triggering signals, followed by an assignment specification:

<signal-event> ::= <name> [‘(‘ [<assignment-specification>] ‘)’]
*<assignment-specification> ::= <attr-name> [‘,’<attr-name>]**

where:

- *<attr-name>* is an implicit assignment of the corresponding attributes of the signal to an attribute (with this name) of the context object owning the triggered behavior.
- *<assignment-specification>* is optional and may be omitted even if the signal has attributes.

Changes from previous UML

This metaclass replaces SignalEvent.

13.3.26 TimeConstraint (from SimpleTime)

Generalizations

- “IntervalConstraint (from SimpleTime)” on page 444

Description

A TimeConstraint defines a Constraint that refers to a TimeInterval.

Attributes

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

Associations

- specification: TimeInterval [1]
  A time expression used to determine whether the constraint is satisfied. Redefines *IntervalConstraint::specification*

Constraints

No additional constraints

Semantics

The semantics of a TimeConstraint is inherited from Constraints. All traces where the constraints are violated are negative traces (i.e., if they occur in practice, the system has failed).

Notation

A TimeConstraint is shown as graphical association between a TimeInterval and the construct that it constrains. Typically this graphical association is a small line (e.g., between an OccurrenceSpecification and a TimeInterval).

Examples

See example in Figure 13.17 where the TimeConstraint is associated with the reception of a Message.

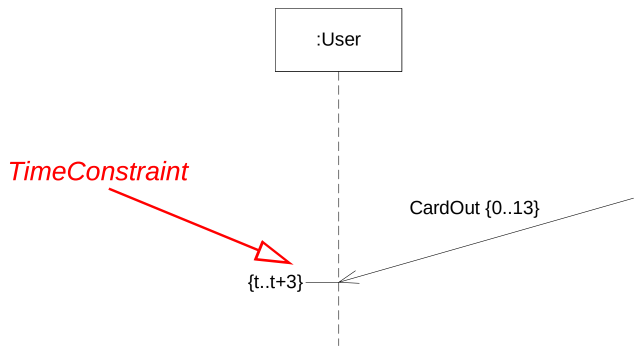


Figure 13.17 - TimeConstraint

Changes from previous UML

This metaclass has been added.

13.3.27 TimeEvent (from Communications, SimpleTime)

A TimeEvent specifies a point in time. At the specified time, the event occurs.

Generalizations

- "Event (from Communications)" on page 442

Description

A time event specifies a point in time by an expression. The expression might be absolute or might be relative to some other point in time.

Attributes

- isRelative: Boolean
  Specifies whether it is relative or absolute time. Default value is *false*.

Associations

- when: TimeExpression [1]
  Specifies the corresponding time deadline.

Constraints

No additional constraints

Semantics

A time event specifies an instant in time by an expression. The expression might be absolute or it might be relative to some other point in time. Relative time events must always be used in the context of a trigger and the starting point is the time at which the trigger becomes active.

Semantic Variation Points

There may be a variable delay between the time of reception and the time of dispatching of the TimeEvent (e.g., due to queueing delays).

Notation

A relative time trigger is specified with the keyword 'after' followed by an expression that evaluates to a time value, such as "after (5 seconds)." An absolute time trigger is specified with the keyword 'at' followed by an expression that evaluates to a time value, such as "Jan. 1, 2000, Noon."

<time-event> ::= <relative-time-event> | <absolute-time-event>
<relative-time-event> ::= 'after' <expression>
<absolute-time-event> ::= 'at' <expression>

Changes from previous UML

The attribute *isRelative* has been added for clarity.

13.3.28 TimeExpression (from SimpleTime)

Generalizations

- “ValueSpecification (from Kernel)” on page 137

Description

A TimeExpression defines a value specification that represents a time value.

Attributes

No additional attributes

Associations

- observation : Observation[*]
  Refers to the time and duration observations that are involved in expr.
- expr : ValueSpecification[0..1]
  The value of the time expression.

Constraints

No additional constraints

Semantics

A TimeExpression denotes a time instant value. The time expression is given by expr which may contain usage of the observations given by observation. In the case where there are no observations, the expr will contain a time constant. In the case where there is no expr, there shall be a single observation that indicates the time expression value.

Notation

A time expression is given by a string. The string is a formula where names of observations and constants are included.

Changes from previous UML

This metaclass has been added.

13.3.29 TimeInterval (from SimpleTime)

Generalizations

- “Interval (from SimpleTime)” on page 444.

Description

A TimeInterval defines the range between two TimeExpressions.

Attributes

No additional attributes

Associations

- min: TimeExpression [1]
  Refers to the TimeExpression denoting the minimum value of the range.
- max: TimeExpression [1]
  Refers to the TimeExpression denoting the maximum value of the range.

Constraints

No additional constraints

Semantics

None

Notation

A TimeInterval is shown with the notation of Interval where each value specification element is a TimeExpression.

Changes from previous UML

This metaclass has been added.

13.3.30 TimeObservation (from SimpleTime)

Generalizations

- “Observation (from SimpleTime)” on page 446

Description

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

Attributes

- firstEvent:Boolean
  The value of firstEvent is related to event. If firstEvent is true, then the corresponding observation event is the first time instant the execution enters event. If firstEvent is false, then the corresponding observation event is the time instant the execution exits event.

Associations

- event:NamedElement[1]
  The observation is determined by the entering or exiting of the event element during execution.

Constraints

No additional constraints

Semantics

A TimeObservation denotes an instant in time.

Notation

A time observation is often denoted by a straight line attached to a model element. The observation is given a name that is shown close to the unattached end of the line.

Changes from previous UML

This metaclass has been added.

13.3.31 Trigger (from Communications)

A trigger relates an event to a behavior that may affect an instance of the classifier.

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 98

Description

A trigger specifies an event that may cause the execution of an associated behavior. An event is often ultimately caused by the execution of an action, but need not be.

Attributes

No additional attributes

Associations

- event : Event [1]
  The event that causes the trigger.

Constraints

No additional constraints

Semantics

Events may cause execution of behavior (e.g., the execution of the effect activity of a transition in a state machine). A trigger specifies the event that may trigger a behavior execution as well as any constraints on the event to filter out events not of interest.

Events are often generated as a result of some action either within the system or in the environment surrounding the system. Upon their occurrence, events are placed into the input pool of the object where they occurred (see BehavioredClassifier on page 434). An event is dispatched when it is taken from the input pool and is processed by the classifier. At this point, the event is considered consumed and referred to as the current event. A consumed event is no longer available for processing.

Semantic Variation Points

No assumptions are made about the time intervals between event occurrence, event dispatching, and consumption. This leaves open the possibility of different semantic variations such as zero-time semantics.

It is a semantic variation whether an event is discarded if there is no appropriate trigger defined for them.

Notation

A trigger is used to define an unnamed event. The details of the syntax for the event are defined by the different subclasses of Event:

<trigger> ::= <call-event> | *<signal-event>* | *<any-receive-event>* | *<time-event>* | *<change-event>*

Changes from previous UML

The corresponding metaclass in 1.x was Event. In 1.x, events were specified with Parameters. Instead, the data that may be communicated by an event is accessed via the properties of the specification element defining the event.

14 Interactions

14.1 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.

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 makes 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 do, 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 the 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.

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 <eventoccurrence1, eventoccurrence2, ...,eventoccurrence-n>. We are aware that other parts of the UML language definition of the term “trace” is used also for other purposes.

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.

14.2 Abstract Syntax

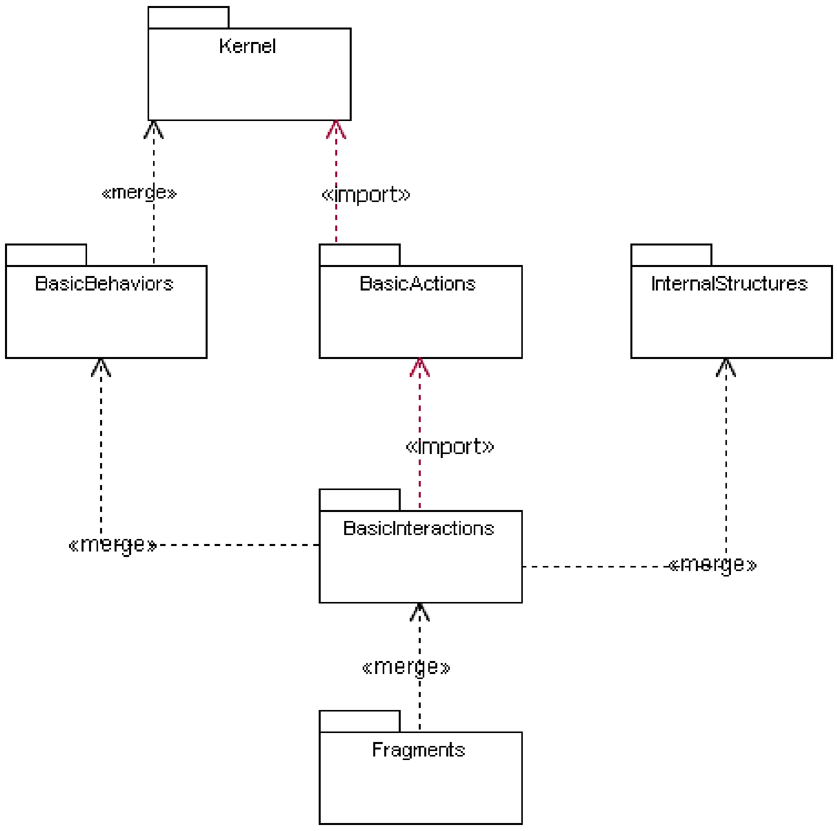


Figure 14.1 - Dependencies of the Interactions packages

Package BasicInteractions

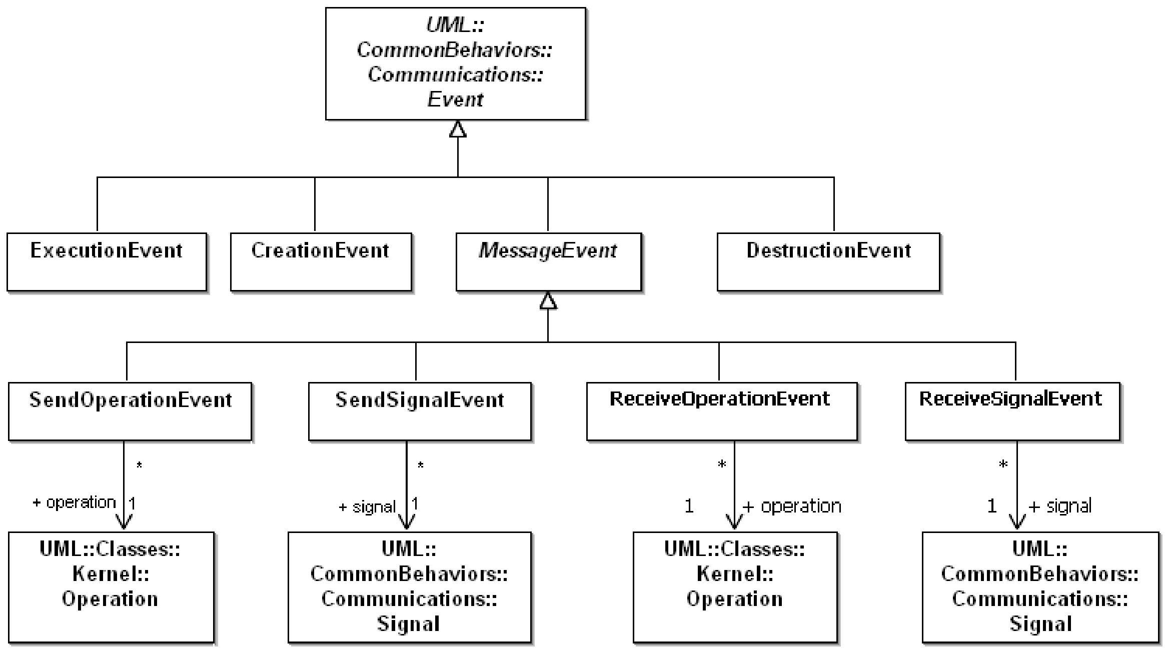


Figure 14.2 Events

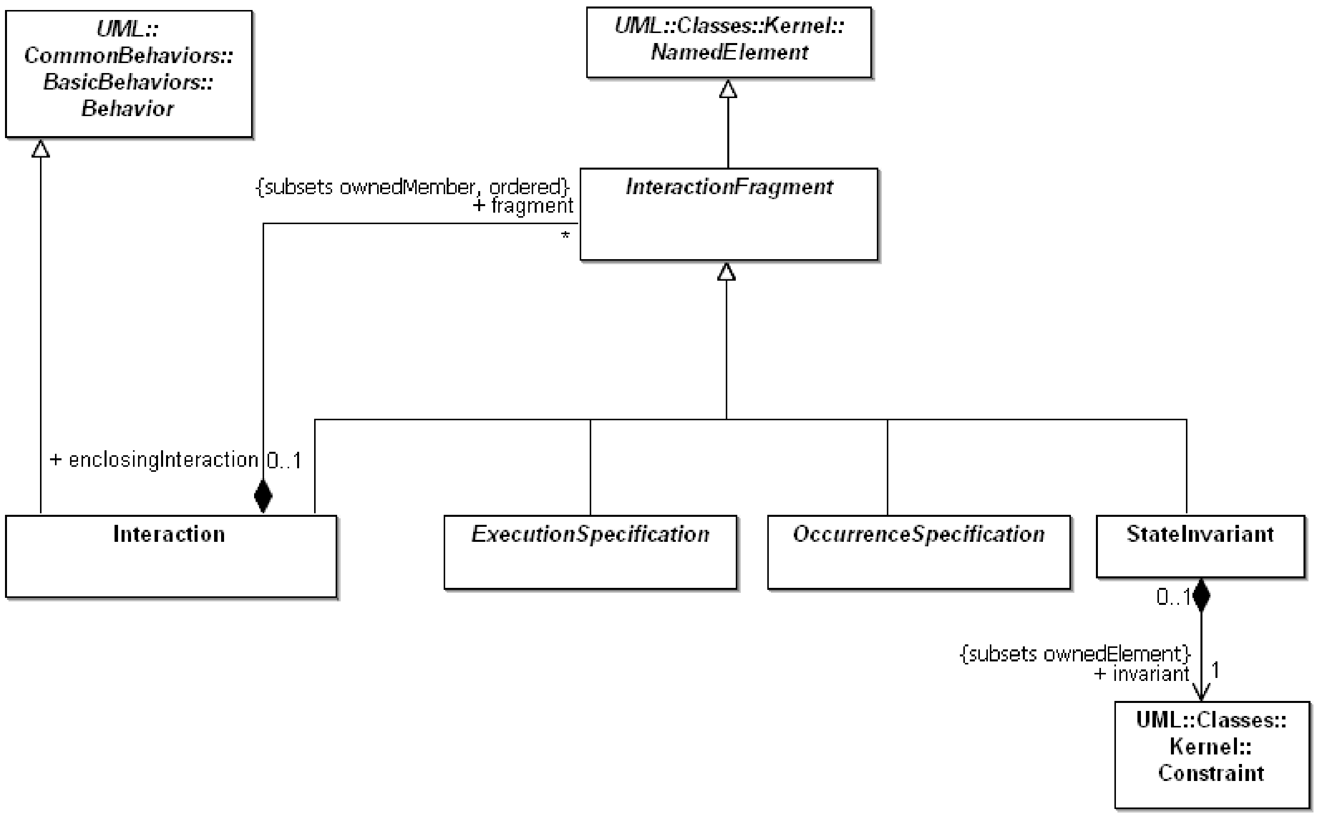


Figure 14.3 - Interactions

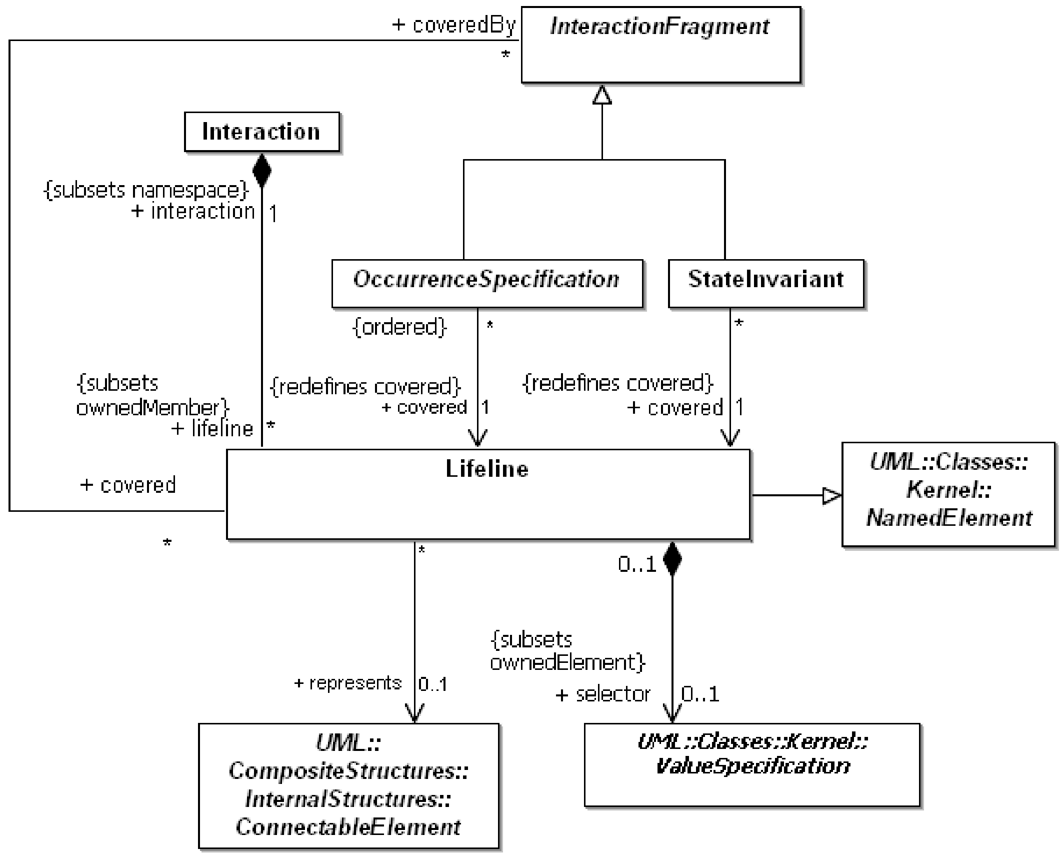


Figure 14.4 - Lifelines

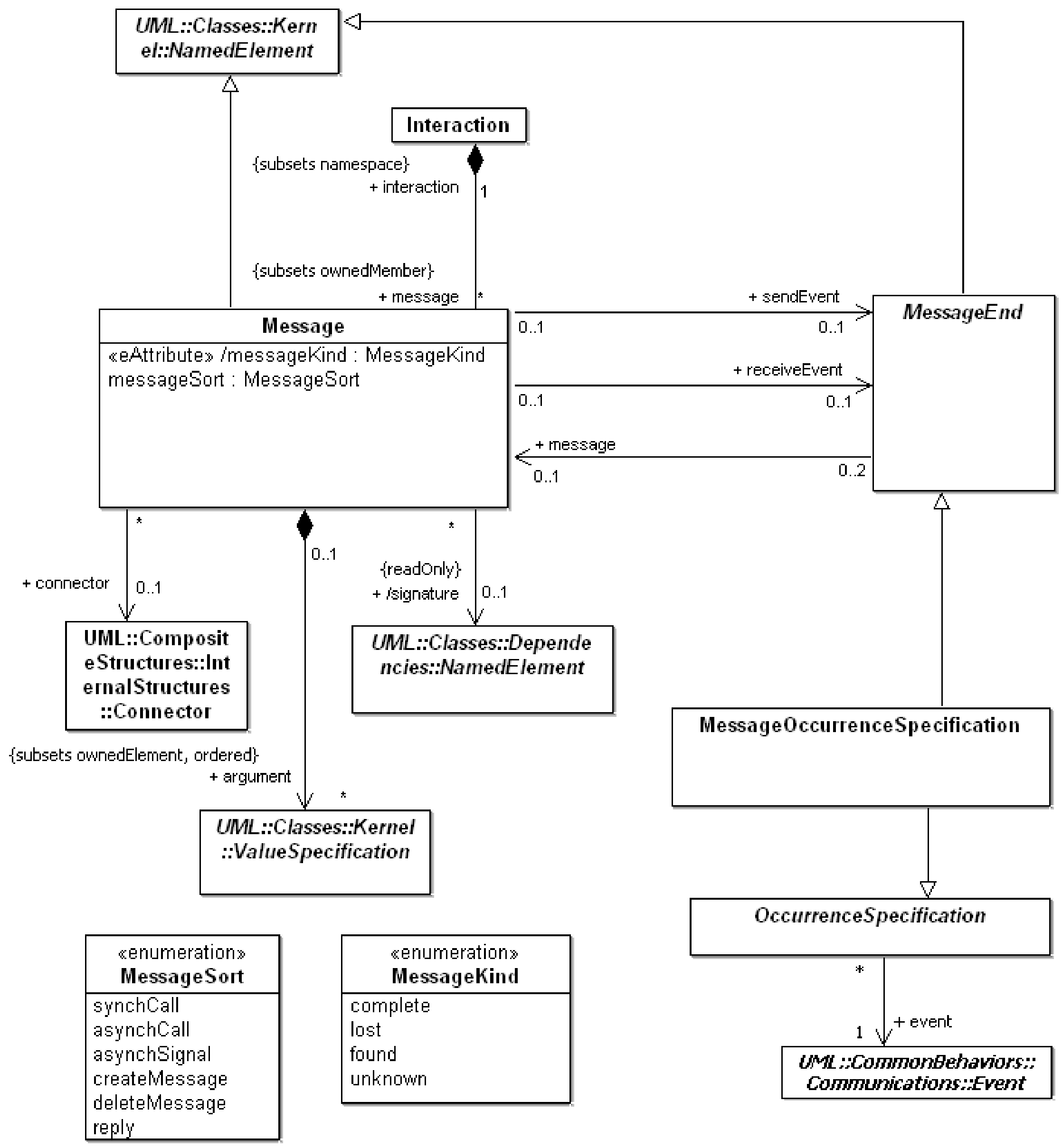

UML::Classes::Kernel::NamedElement
Interaction
{subsets namespace}
+ interaction
1
{subsets ownedMember}
+ message
*
Message
«eAttribute» /messageKind : MessageKind
messageSort : MessageSort
+ sendEvent
0..1
0..1
+ receiveEvent
0..1
0..1
+ message
0..1
0..2
MessageEnd
*
+ connector
0..1
0..1
*
{readOnly}
+ /signature
0..1
UML::CompositeStructures::InternalStructures::Connector
UML::Classes::Dependencies::NamedElement
MessageOccurrenceSpecification
{subsets ownedElement, ordered}
+ argument
*
UML::Classes::Kernel::ValueSpecification
OccurrenceSpecification
*
+ event
1
UML::CommonBehaviors::Communications::Event
«enumeration» MessageSort
synchCall
asynchCall
asynchSignal
createMessage
deleteMessage
reply
«enumeration» MessageKind
complete
lost
found
unknown

Figure 14.5 - Messages

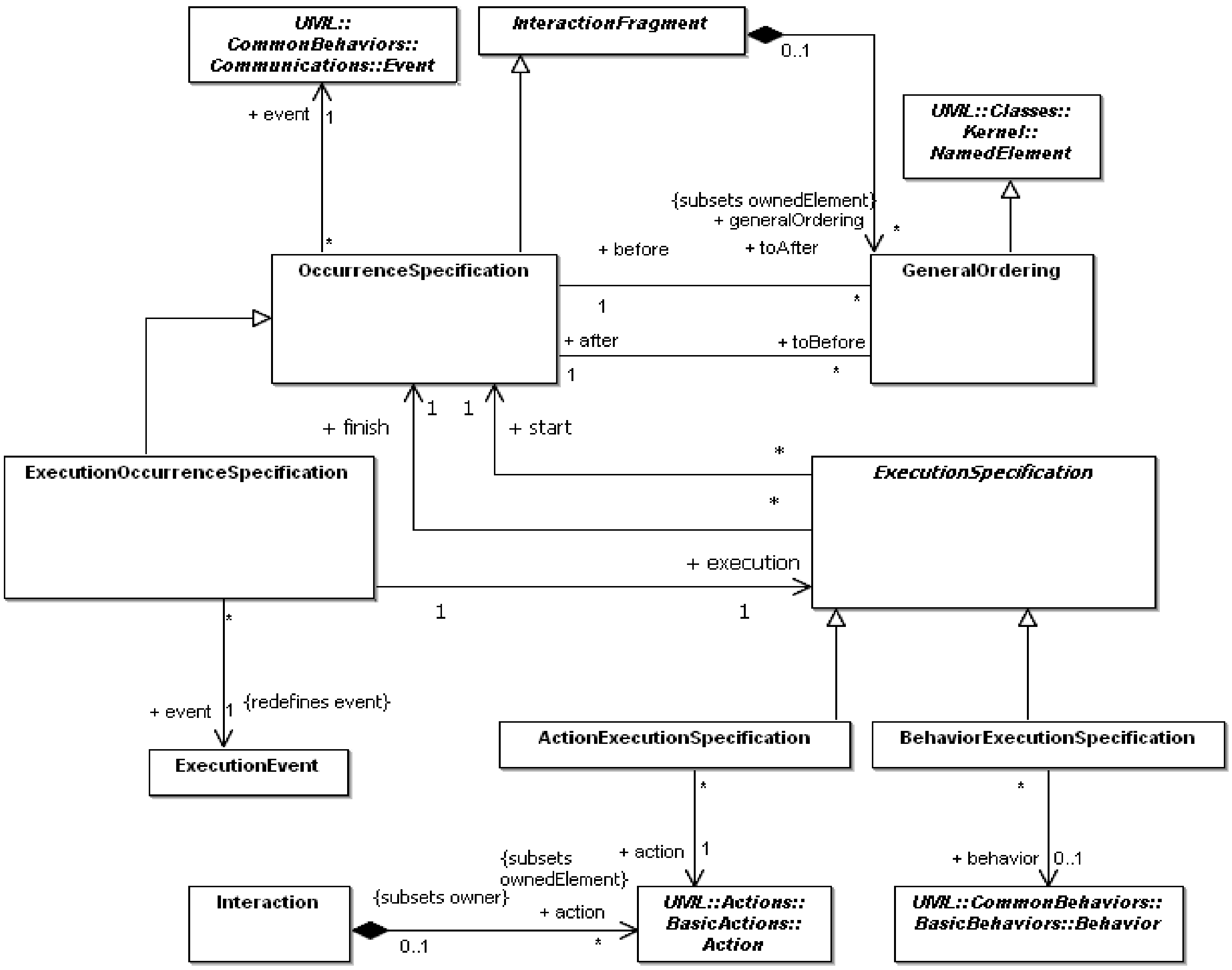


Figure 14.6 - Occurrences

Package Fragments

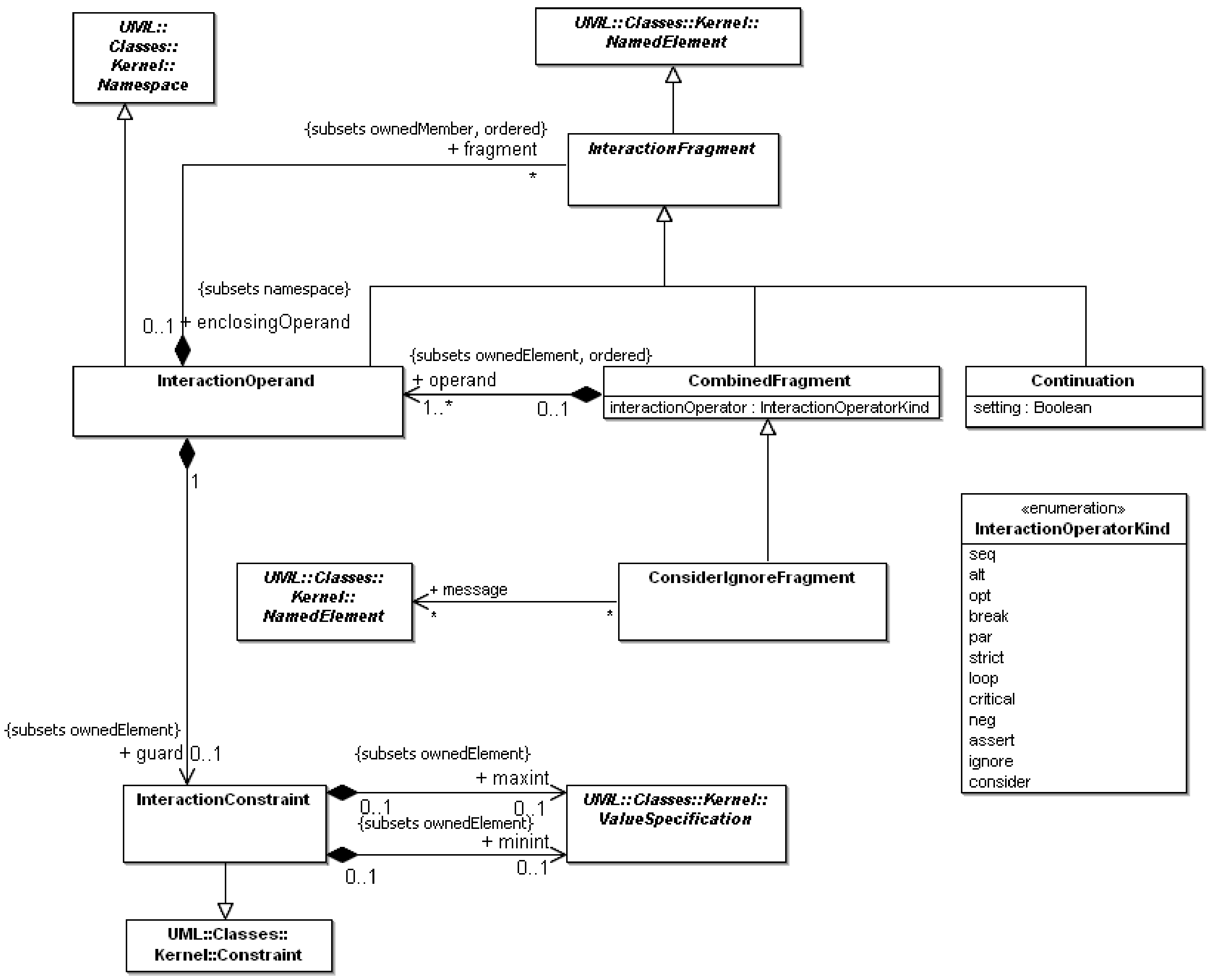


Figure 14.7 - CombinedFragments

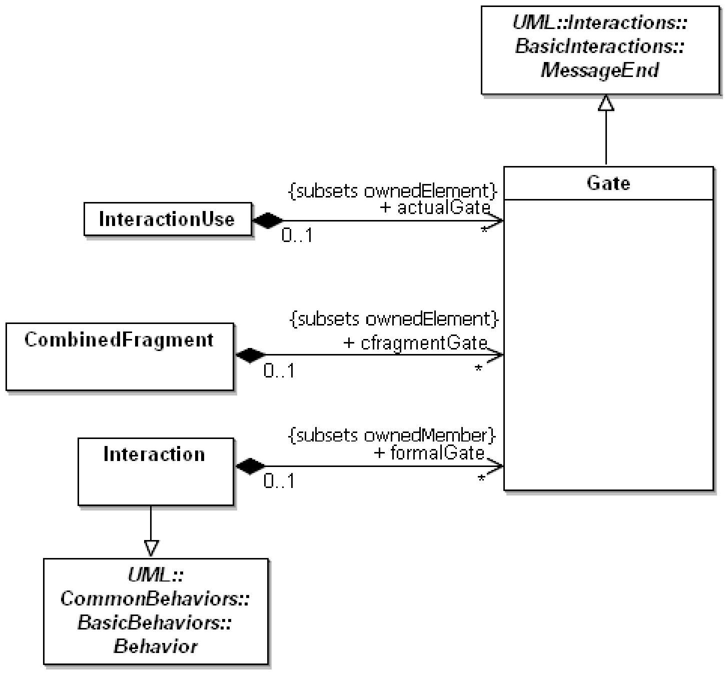


Figure 14.8 - Gates

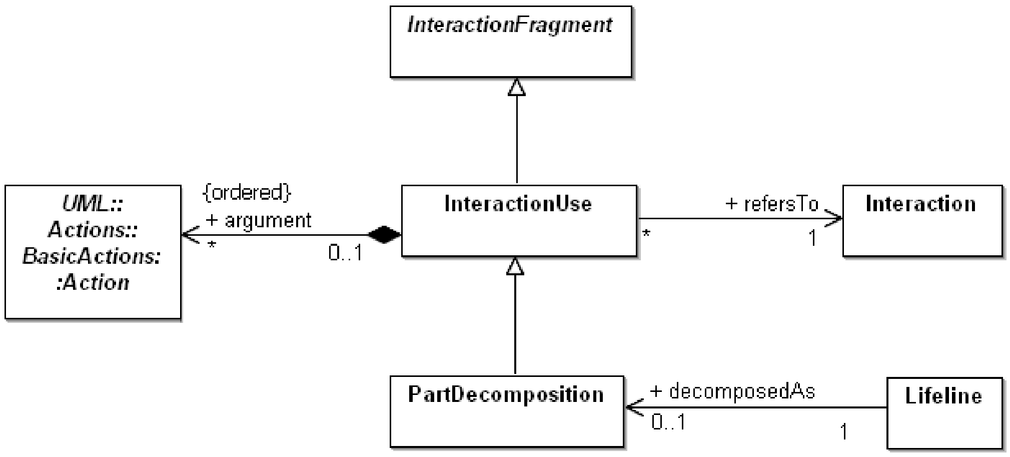


Figure 14.9 - InteractionUses

14.3 Class Descriptions

14.3.1 ActionExecutionSpecification (from BasicInteractions)

Generalizations

- “ExecutionSpecification (from BasicInteractions)” on page 480.

Description

ActionExecutionSpecification is a kind of ExecutionSpecification representing the execution of an action.

Attributes

No additional attributes

Associations

- action : Action [1]
  Action whose execution is occurring.

Constraints

[1] The Action referenced by the ActionExecutionOccurrence, if any, must be owned by the Interaction owning the ActionExecutionOccurrence.

Semantics

See “ExecutionSpecification (from BasicInteractions)” on page 480.

Notation

See “ExecutionSpecification (from BasicInteractions)” on page 480.

Rationale

ActionExecutionSpecification is introduced to support interactions specifying messages that result from actions, which may be actions owned by other behaviors.

14.3.2 BehaviorExecutionSpecification (from BasicInteractions)

Generalizations

- “ExecutionSpecification (from BasicInteractions)” on page 480

Description

BehaviorExecutionSpecification is a kind of ExecutionSpecification representing the execution of a behavior.

Attributes

No additional attributes

Associations

- behavior : Behavior [0..1]
  Behavior whose execution is occurring.

Constraints

No additional constraints

Semantics

See "ExecutionSpecification (from BasicInteractions)" on page 480.

Notation

See "ExecutionSpecification (from BasicInteractions)" on page 480.

Rationale

BehaviorExecutionSpecification is introduced to support interactions specifying messages that result from behaviors.

14.3.3 CombinedFragment (from Fragments)

Generalizations

- "InteractionFragment (from BasicInteractions, Fragments)" on page 487

Description

A combined fragment defines an expression of interaction fragments. A combined fragment is defined by an interaction operator and corresponding interaction operands. Through the use of CombinedFragments the user will be able to describe a number of traces in a compact and concise manner.

CombinedFragment is a specialization of InteractionFragment.

Attributes

- interactionOperator : InteractionOperatorKind
  Specifies the operation that defines the semantics of this combination of InteractionFragments. Default value is *seq*.

Associations

- cfragmentGate : Gate[*]
  Specifies the gates that form the interface between this CombinedFragment and its surroundings.
- operand: InteractionOperand[1..*]
  The set of operands of the combined fragment. {Subsets *Element::ownedElement*}

Constraints

[1] If the interactionOperator is *opt*, *loop*, *break*, or *neg*, there must be exactly one operand.

[2] The InteractionConstraint with minint and maxint only apply when attached to an InteractionOperand where the interactionOperator is *loop*.

[3] If the interactionOperator is *break*, the corresponding InteractionOperand must cover all Lifelines within the enclosing InteractionFragment.

[4] The interaction operators ‘consider’ and ‘ignore’ can only be used for the CombineIgnoreFragment subtype of CombinedFragment.

((interactionOperator = #consider) **or** (interactionOperator = #ignore)) **implies** oclsisTypeOf(CombineIgnoreFragment)

Semantics

The semantics of a CombinedFragment is dependent upon the interactionOperator as explained below.

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.

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.

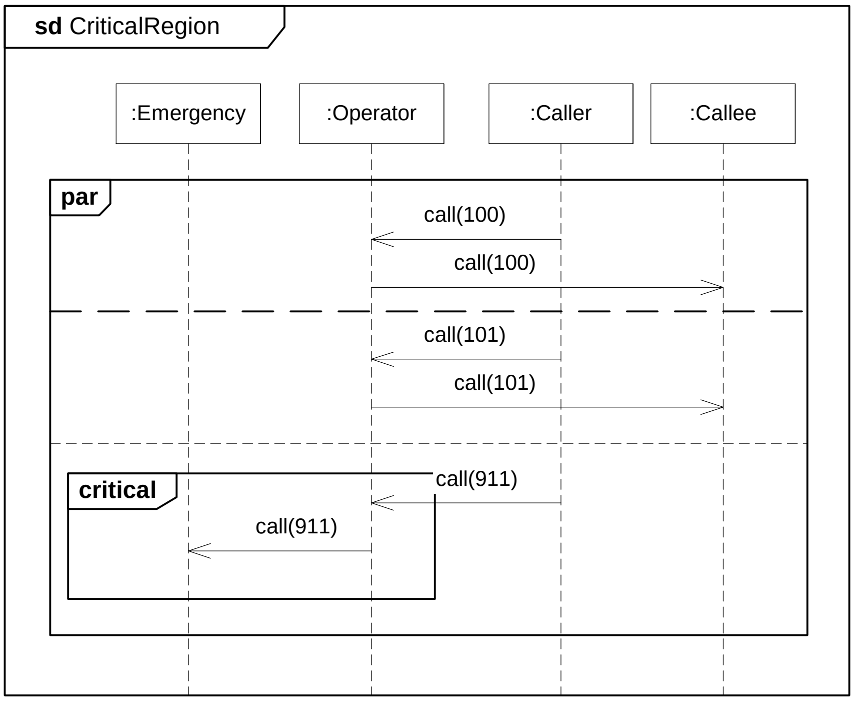


Figure 14.10 - Critical Region

The example, Figure 14.10 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.

Ignore / Consider

See the semantics of 14.3.4, “ConsiderIgnoreFragment (from Fragments),” on page 474.

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 14.24.

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.

The Semantics of Gates (see also "Gate (from Fragments)" on page 482)

The gates of a CombinedFragment represent the syntactic interface between the CombinedFragment and its surroundings, which means the interface towards other InteractionFragments.

The only purpose of gates is to define the source and the target of Messages.

Notation

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.

Strict

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

See the notation for "ConsiderIgnoreFragment (from Fragments)" on page 474.

Loop

Textual syntax of the loop operand:

‘**loop**[‘(‘ <minint> [‘,’ <maxint>] ‘)’]

<minint> ::= non-negative natural

<maxint> ::= non-negative natural (greater than or equal to *<minint>* | ‘*’

‘*’ means infinity.

If only *<minint>* is present, this means that *<minint>* = *<maxint>* = *<integer>*.

If only **loop**, then this means a loop with infinity upper bound and with 0 as lower bound.

Presentation Options for "coregion area"

A notational shorthand for parallel combined fragments are available 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 14.12.

Examples

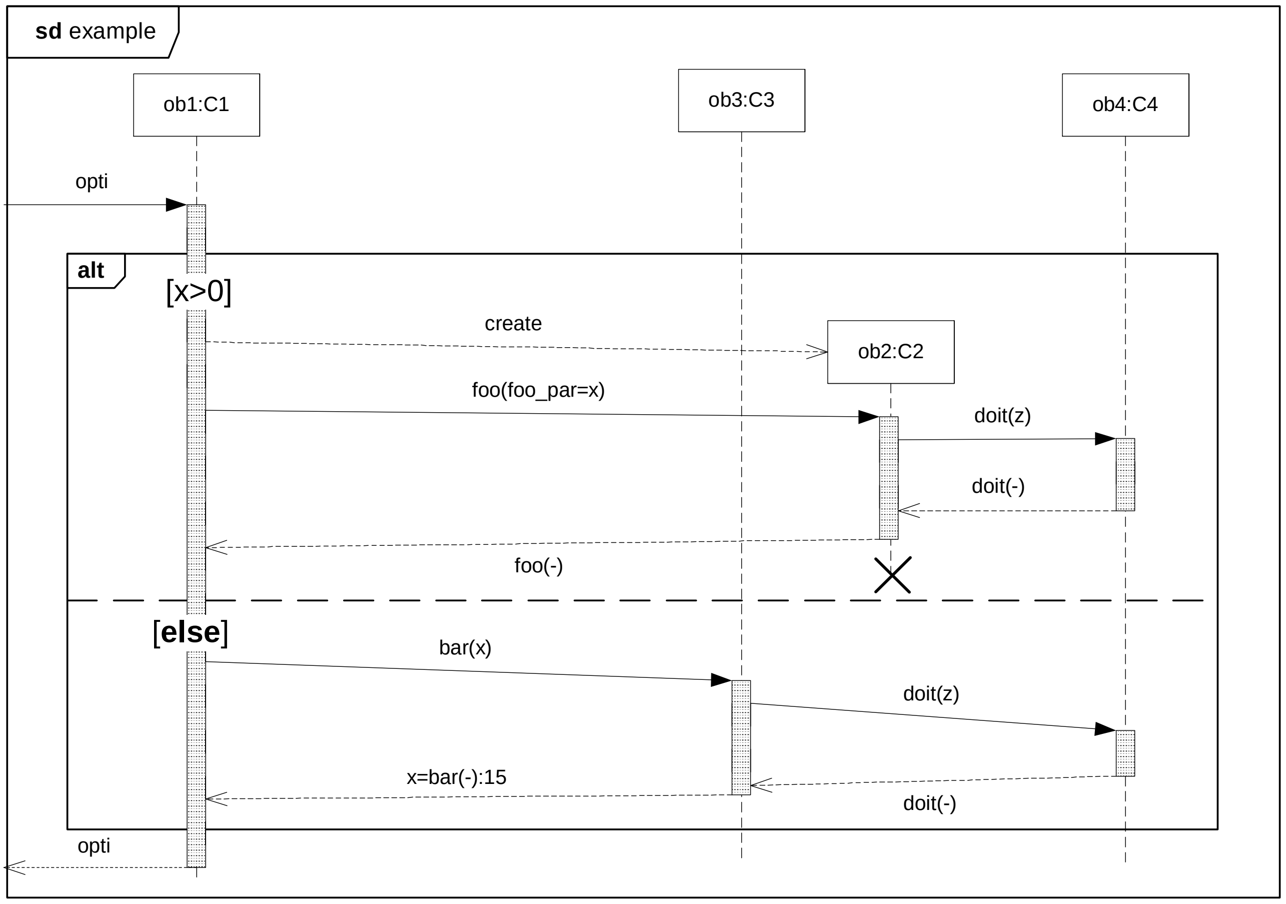


Figure 14.11 - CombinedFragment

Changes from previous UML

This concept was not included in UML 1.x.

14.3.4 ConsiderIgnoreFragment (from Fragments)

Generalizations

- “CombinedFragment (from Fragments)” on page 469

Description

A ConsiderIgnoreFragment is a kind of combined fragment that is used for the consider and ignore cases, which require lists of pertinent messages to be specified.

Attributes

No additional attributes.

Associations

- message : NamedElement [0..*]
  The set of messages that apply to this fragment.

Constraints

[1] The interaction operator of a ConsiderIgnoreFragment must be either ‘consider’ or ‘ignore.’
(interactionOperator = #consider) **or** (interactionOperator = #ignore)

[2] The NamedElements must be of a type of element that identifies a message (e.g., an Operation, Reception, or a Signal).
message->forAll(m | m.oclIsKindOf(Operation) **or** m.oclIsKindOf(Reception) **or** m.oclIsKindOf(Signal))

Semantics

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*.

Notation

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}.

Examples

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 14.24 on page 513 shows an example of consider/ignore fragments.

Changes from previous UML

This concept did not exist in UML 1.x.

14.3.5 Continuation (from Fragments)

Generalizations

- “InteractionFragment (from BasicInteractions, Fragments)” on page 487

Description

A Continuation is a syntactic way to define continuations of different branches of an Alternative CombinedFragment. Continuation is intuitively similar to labels representing intermediate points in a flow of control.

Attributes

- setting : Boolean
  True when the Continuation is at the end of the enclosing InteractionFragment and False when it is in the beginning.

Constraints

[1] Continuations with the same name may only cover the same set of Lifelines (within one Classifier).

[2] Continuations are always global in the enclosing InteractionFragment (e.g., it always covers all Lifelines covered by the enclosing InteractionFragment).

[3] Continuations always occur as the very first InteractionFragment or the very last InteractionFragment of the enclosing InteractionFragment.

Semantics

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.

Notation

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.

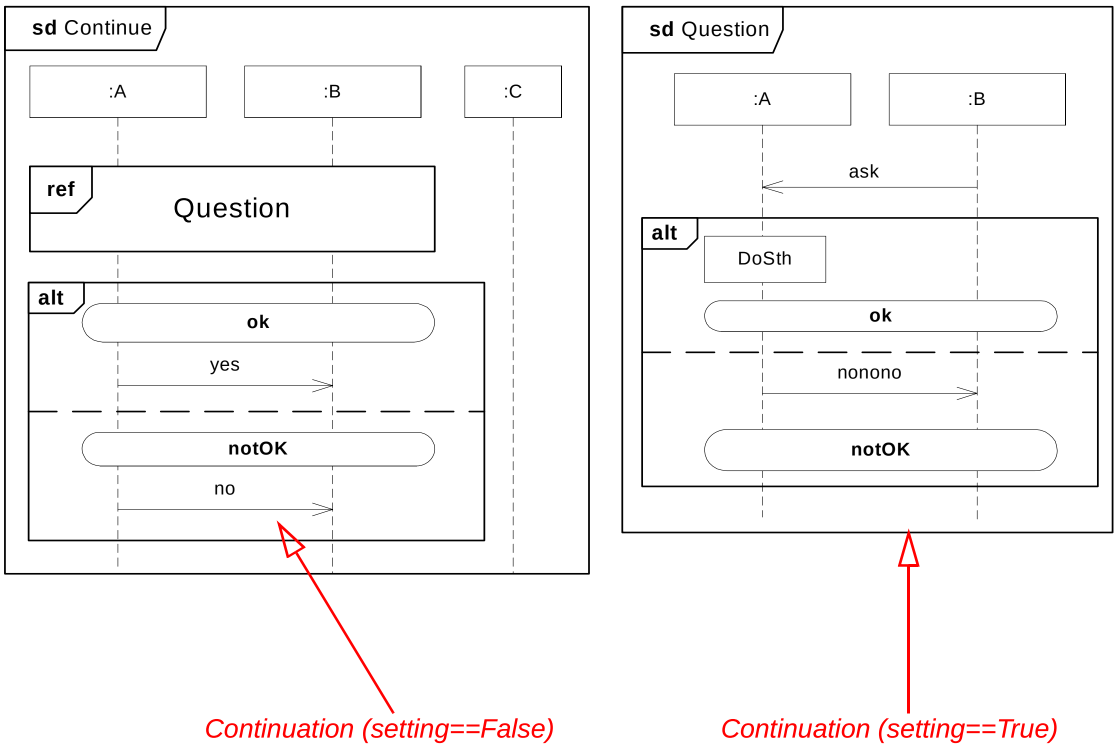


Figure 14.12 - Continuation

The two diagrams in Figure 14.12 are together equivalent to the diagram in Figure 14.13.

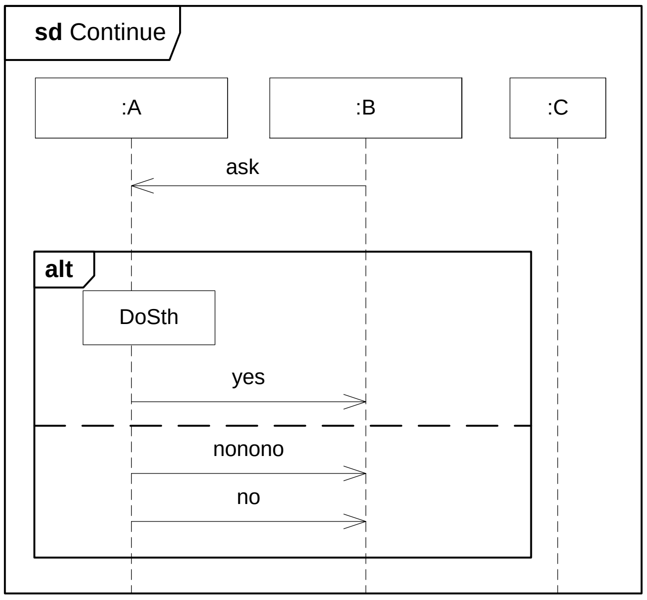


Figure 14.13 - Continuation interpretation

14.3.6 CreationEvent (from BasicInteractions)

Generalizations

- "Event (from Communications)" on page 442

Description

A CreationEvent models the creation of an object.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] No other OccurrenceSpecification may appear above an OccurrenceSpecification which references a CreationEvent on a given Lifeline in an InteractionOperand.

Semantics

A creation event represents the creation of an instance. It may result in the subsequent execution of initializations as well as the invocation of the specified classifier behavior (see "Common Behaviors" on page 421).

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.7 DestructionEvent (from BasicInteractions)

Generalizations

- "Event (from Communications)" on page 442

Description

A DestructionEvent models the destruction of an object.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] No other OccurrenceSpecifications may appear below an OccurrenceSpecification that references a DestructionEvent on a given Lifeline in an InteractionOperand.

Semantics

A destruction event represents the destruction of the instance described by the lifeline containing the OccurrenceSpecification that references the destruction event. It may result in the subsequent destruction of other objects that this object owns by composition (see “Common Behaviors” on page 421).

Notation

The DestructionEvent is depicted by a cross in the form of an X at the bottom of a Lifeline.

Figure 14.14 - DestructionEvent symbol

See example in Figure 14.11.

Changes from previous UML

DestructionEvent is new in UML 2.0.

14.3.8 ExecutionEvent (from BasicInteractions)

Generalizations

- “Event (from Communications)” on page 442

Description

An ExecutionEvent models the start or finish of an execution occurrence.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

An execution event represents the start or finish of the execution of an action or a behavior.

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.9 ExecutionOccurrenceSpecification (from BasicInteractions)

Generalizations

- "OccurrenceSpecification (from BasicInteractions)" on page 498

Description

An ExecutionOccurrenceSpecification represents moments in time at which actions or behaviors start or finish.

Attributes

No additional attributes

Associations

- event : ExecutionEvent [1]
  Redefines the event referenced to be restricted to an execution event.
- execution : ExecutionSpecification [1]
  References the execution specification describing the execution that is started or finished at this execution event.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.10 ExecutionSpecification (from BasicInteractions)

Generalizations

- "InteractionFragment (from BasicInteractions, Fragments)" on page 487

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 ExecutionOccurrenceSpecifications, the start ExecutionOccurrenceSpecification and the finish ExecutionOccurrenceSpecification.

Associations

- start : OccurrenceSpecification[1]
  References the OccurrenceSpecification that designates the start of the Action or Behavior.
- finish: OccurrenceSpecification[1]
  References the OccurrenceSpecification that designates the finish of the Action or Behavior.

Constraints

[1] The startEvent and the finishEvent must be on the same Lifeline.

```
start.lifeline = finish.lifeline
```

Semantics

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).

Notation

ExecutionOccurences are represented as thin rectangles (grey or white) on the lifeline (see “Lifeline (from BasicInteractions, Fragments)” on page 492).

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 14.13 on page 477.

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 execution occurrences on the same lifeline are represented by overlapping rectangles as shown in Figure 14.15.

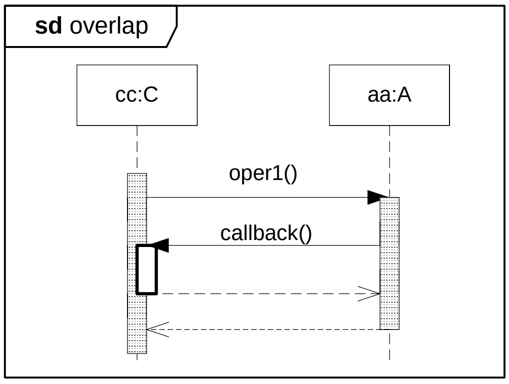


Figure 14.15 - Overlapping execution occurrences

14.3.11 Gate (from Fragments)

Generalizations

- "MessageEnd (from BasicInteractions)" on page 496

Description

A Gate is a connection point for relating a Message outside an InteractionFragment with a Message inside the InteractionFragment.

Gate is a specialization of MessageEnd.

Gates are connected through Messages. A Gate is actually a representative of an OccurrenceSpecification that is not in the same scope as the Gate.

Gates play different roles: we have formal gates on Interactions, actual gates on InteractionUses, expression gates on CombinedFragments.

Constraints

[1] The message leading to/from an actualGate of an InteractionUse must correspond to the message leading from/to the formalGate with the same name of the Interaction referenced by the InteractionUse.

[2] The message leading to/from an (expression) Gate within a CombinedFragment must correspond to the message leading from/to the CombinedFragment on its outside.

Semantics

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*). The gates and the messages between gates have one purpose, namely to establish the concrete sender and receiver for every message.

Notation

Gates are just points on the frame, the ends of the messages. They may have an explicit name (see Figure 14.19).

The same gate may appear several times in the same or different diagrams.

14.3.12 GeneralOrdering (from BasicInteractions)

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 98

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.

A GeneralOrdering is a specialization of NamedElement.

A GeneralOrdering may appear anywhere in an Interaction, but only between OccurrenceSpecifications.

Associations

- before: OccurrenceSpecification[1]
  The OccurrenceSpecification referenced comes before the OccurrenceSpecification referenced by *after*.
- after:OccurrenceSpecification[1]
  The OccurrenceSpecification referenced comes after the OccurrenceSpecification referenced by *before*.

Semantics

A GeneralOrdering is introduced to restrict the set of possible sequences. A partial order of OccurrenceSpecifications is defined by a set of GeneralOrdering.

Notation

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).

14.3.13 Interaction (from BasicInteraction, Fragments)

Generalizations

- "Behavior (from BasicBehaviors)" on page 430
- "InteractionFragment (from BasicInteractions, Fragments)" on page 487

Description

An interaction is a unit of behavior that focuses on the observable exchange of information between ConnectableElements.

An Interaction is a specialization of InteractionFragment and of Behavior.

Associations

- formalGate: Gate[*]
  Specifies the gates that form the message interface between this Interaction and any InteractionUses that reference it.
- lifeline: LifeLine[0..*]
  Specifies the participants in this Interaction.
- message:Message[*]
  The Messages contained in this Interaction.
- fragment:InteractionFragment[*]
  The ordered set of fragments in the Interaction.
- action:Action[*]
  Actions owned by the Interaction. See "ActionExecutionSpecification (from BasicInteractions)" on page 468.

Semantics

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 “Weak Sequencing” on page 470.

The invalid set of traces are 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.

Basic trace model: The semantics of an Interaction is given by 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 <e1, e2, ... , en>.

An event occurrence will 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.

Relation of trace model to execution model: In Chapter 13, “Common Behaviors” 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.

Notation

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 14.28. 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.

Please refer to Section 14.4 to see examples of notation for Interactions.

Examples

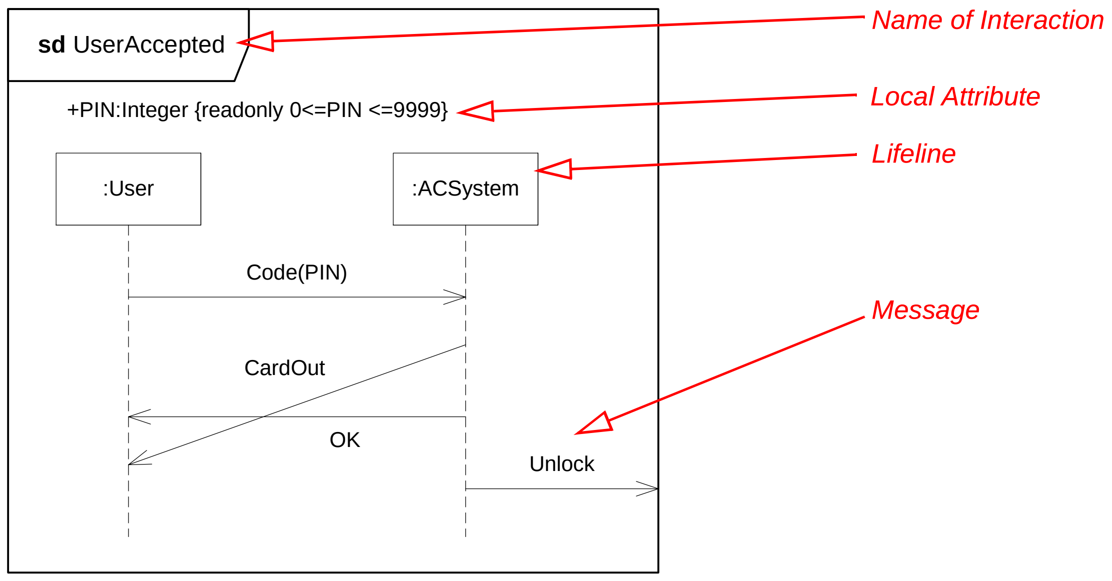


Figure 14.16 - An example of an Interaction in the form of a Sequence Diagram

The example in Figure 14.16 shows three messages communicated between two (anonymous) lifelines of types *User* and *ACSystem*. 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 *ACSystem* 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.

Changes from previous UML

Interactions are now contained within Classifiers and not only within Collaborations. Their participants are modeled by Lifelines instead of ClassifierRoles.

14.3.14 InteractionConstraint (from Fragments)

Generalizations

- "Constraint (from Kernel)" on page 58

Description

An InteractionConstraint is a Boolean expression that guards an operand in a CombinedFragment.

InteractionConstraint is a specialization of Constraint.

Furthermore the InteractionConstraint contains two expressions designating the minimum and maximum number of times a loop CombinedFragment should execute.

Associations

- minint: ValueSpecification[0..1]
  The minimum number of iterations of a loop.
- maxint: ValueSpecification[0..1]
  The maximum number of iterations of a loop.

Constraints

[1] The dynamic variables that take part in the constraint must be owned by the ConnectableElement corresponding to the covered Lifeline.

[2] The constraint may contain references to global data or write-once data.

[3] Minint/maxint can only be present if the InteractionConstraint is associated with the operand of a loop CombinedFragment.

[4] If minint is specified, then the expression must evaluate to a non-negative integer.

[5] If maxint is specified, then the expression must evaluate to a positive integer.

[6] If maxint is specified, then minint must be specified and the evaluation of maxint must be >= the evaluation of minint.

Semantics

InteractionConstraints are always used in connection with CombinedFragments, see "CombinedFragment (from Fragments)" on page 469.

Notation

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.

Please refer to an example of InteractionConstraints in Figure 14.11 on page 474 and Figure 14.28 on page 520.

14.3.15 InteractionFragment (from BasicInteractions, Fragments)

Generalizations

- “NamedElement (from Kernel, Dependencies)” on page 98.

Description

InteractionFragment is an abstract notion of the most general interaction unit. An interaction fragment is a piece of an interaction. Each interaction fragment is conceptually like an interaction by itself. InteractionFragment is an abstract class and a specialization of NamedElement.

Associations

- enclosingOperand: InteractionOperand[0..1]
  The operand enclosing this InteractionFragment (they may nest recursively).
- covered : Lifeline[*]
  References the Lifelines that the InteractionFragment involves.
- generalOrdering:GeneralOrdering[*]
  The general ordering relationships contained in this fragment.
- enclosingInteraction: Interaction[0..1]
  The Interaction enclosing this InteractionFragment.

Semantics

The semantics of an InteractionFragment is a pair of set of traces. See “Interaction (from BasicInteraction, Fragments)” for explanation of how to calculate the traces.

Notation

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

Changes from previous UML

This concept did not appear in UML 1.x.

14.3.16 InteractionOperand (from Fragments)

Generalizations

- “InteractionFragment (from BasicInteractions, Fragments)” on page 487
- “Namespace (from Kernel)” on page 99

Description

An InteractionOperand is contained in a CombinedFragment. An InteractionOperand represents one operand of the expression given by the enclosing CombinedFragment.

An InteractionOperand is an InteractionFragment with an optional guard expression. An InteractionOperand may be guarded by an InteractionConstraint. Only InteractionOperands with a guard that evaluates to true at this point in the interaction will be considered for the production of the traces for the enclosing CombinedFragment.

InteractionOperand contains an ordered set of InteractionFragments.

In Sequence Diagrams these InteractionFragments are ordered according to their geometrical position vertically. The geometrical position of the InteractionFragment is given by the topmost vertical coordinate of its contained OccurrenceSpecifications or symbols.

Associations

- fragment: InteractionFragment[*]
  The fragments of the operand.
- guard: InteractionConstraint[0..1]
  Constraint of the operand.

Constraints

[1] The guard must be placed directly prior to (above) the OccurrenceSpecification that will become the first OccurrenceSpecification within this InteractionOperand.

[2] The guard must contain only references to values local to the Lifeline on which it resides, or values global to the whole Interaction (See “InteractionConstraint (from Fragments)” on page 486).

Semantics

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 “CombinedFragment (from Fragments)” on page 469.

Notation

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.

See Figure 14.11 for examples of InteractionOperand.

14.3.17 InteractionOperatorKind (from Fragments)

Generalizations

None

Description

Interaction OperatorKind is an enumeration designating the different kinds of operators of CombinedFragments. The InteractionOperand defines the type of operator of a CombinedFragment. The literal values of this enumeration are:

- alt
- opt
- par
- loop
- critical
- neg
- assert
- strict
- seq
- ignore
- consider

Semantics

The value of the interactionOperator is significant for the semantics of “CombinedFragment (from Fragments)” on page 469.

Notation

The value of the InteractionOperandKind is given as text in a small compartment in the upper left corner of the CombinedFragment frame. See Figure 14.11 on page 474 for examples of InteractionOperatorKind.

14.3.18 InteractionUse (from Fragments)

Generalizations

- “InteractionFragment (from BasicInteractions, Fragments)” on page 487

Description

An InteractionUse refers to an Interaction. The InteractionUse is a shorthand for copying the contents of the referred 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.

It is common to want to share portions of an interaction between several other interactions. An InteractionUse allows multiple interactions to reference an interaction that represents a common portion of their specification.

Description

InteractionUse is a specialization of InteractionFragment.

An InteractionUse has a set of actual gates that must match the formal gates of the referenced Interaction.

Associations

- refersTo: Interaction[1]
  Refers to the Interaction that defines its meaning.
- argument:Action[*]
  The actual arguments of the Interaction.

• actualGate:Gate[*]
  The actual gates of the InteractionUse.

Constraints

[1] Actual Gates of the InteractionUse must match Formal Gates of the referred Interaction. Gates match when their names are equal.

[2] The InteractionUse must cover all Lifelines of the enclosing Interaction that appear within the referred Interaction.

[3] The arguments of the InteractionUse must correspond to parameters of the referred Interaction.

[4] The arguments must only be constants, parameters of the enclosing Interaction or attributes of the classifier owning the enclosing Interaction.

Semantics

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.

Notation

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-oargument>] ')'] [':' <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 14.25.

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 ("Message (from BasicInteractions)" on page 493).

If the InteractionUse returns a value, this may be described following a colon at the end of the clause.

Examples

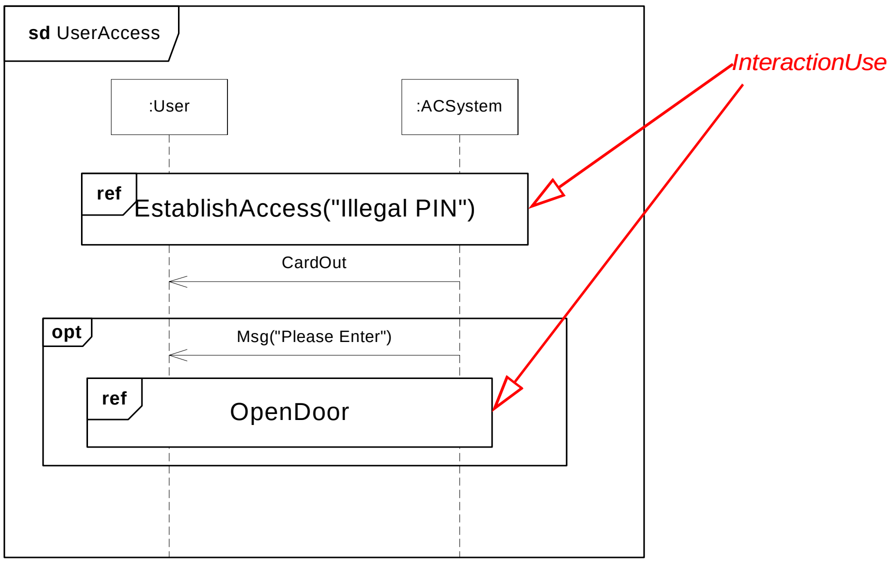


Figure 14.17 - InteractionUse

In Figure 14.17 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.*

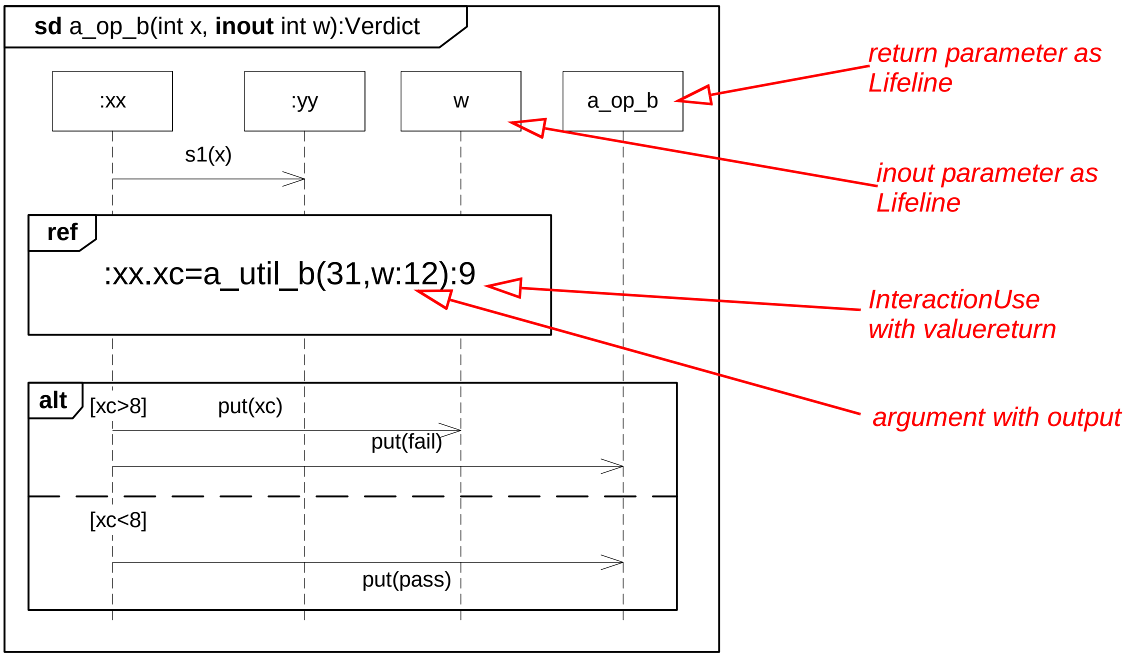

sd a_op_b(int x, inout int w):Verdict
:xx
:yy
w
a_op_b
s1(x)
ref
:xx.xc=a_util_b(31,w:12):9
alt
[xc>8]
put(xc)
put(fail)
[xc<8]
put(pass)
return parameter as Lifeline
inout parameter as Lifeline
InteractionUse with valuereturn
argument with output

Figure 14.18 - InteractionUse with value return

In Figure 14.18 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.

Changes from previous UML

InteractionUse was not a concept in UML 1.x.

14.3.19 Lifeline (from BasicInteractions, Fragments)

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 98.

Description

A lifeline represents an individual participant in the Interaction. While Parts and StructuralFeatures may have multiplicity greater than 1, Lifelines represent only one interacting entity.

Lifeline is a specialization of NamedElement.

If the referenced ConnectableElement is multivalued (i.e, has a multiplicity > 1), then the Lifeline may have an expression (the 'selector') that specifies which particular part is represented by this Lifeline. If the selector is omitted, this means that an arbitrary representative of the multivalued ConnectableElement is chosen.

Associations

- selector : ValueSpecification[0..1]
  If the referenced ConnectableElement is multivalued, then this specifies the specific individual part within that set.
- interaction: Interaction[1]
  References the Interaction enclosing this Lifeline.
- represents: ConnectableElement[0..1]
  References the ConnectableElement within the classifier that contains the enclosing interaction.
- decomposedAs : PartDecomposition[0..1]
  References the Interaction that represents the decomposition.
- coveredBy : InteractionFragment [0..1]
  References the InteractionFragments in which this Lifeline takes part

Constraints

[1] If two (or more) InteractionUses within one Interaction, refer to Interactions with common Lifelines, those Lifelines must also appear in the Interaction with the InteractionUses. By 'common Lifelines' we mean Lifelines with the same selector and represents associations.

[2] The selector for a Lifeline must only be specified if the referenced Part is multivalued.
(self.selector->isEmpty() **implies not** self.represents.isMultivalued()) **or**
(**not** self.selector->isEmpty() **implies** self.represents.isMultivalued())

[3] The classifier containing the referenced ConnectableElement must be the same classifier, or an ancestor, of the classifier that contains the interaction enclosing this lifeline.

```
if (represents->notEmpty()) then
   (if selector->notEmpty() then represents.isMultivalued() else not represents.isMultivalued())
```

Semantics

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.

Notation

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 encloser may be shown separately even when *self* is included.

To depict method activations we apply a thin grey or white rectangle that covers the Lifeline line.

Examples

See Figure 14.16 where the Lifelines are pointed to.

See Figure 14.11 to see method activations.

Changes from previous UML

Lifelines are basically the same concept as before in UML 1.x.

14.3.20 Message (from BasicInteractions)

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 98

Description

A Message defines a particular communication between Lifelines of an Interaction.

A Message is a NamedElement that defines one specific kind of communication in an Interaction. A communication can be, for example, raising a signal, invoking an Operation, creating or destroying an Instance. The Message specifies not only the kind of communication given by the dispatching ExecutionSpecification, but also the sender and the receiver.

A Message associates normally two OccurrenceSpecifications - one sending OccurrenceSpecification and one receiving OccurrenceSpecification.

Attributes

- messageKind:MessageKind
  The derived kind of the Message (*complete*, *lost*, *found*, or *unknown*). Default value is *unknown*.
- messageSort:MessageSort
  The sort of communication reflected by the Message. Default value is *synchCall*.

Associations

- interaction:Interaction[1]
  The enclosing Interaction owning the Message.
- sendEvent : MessageEnd[0..1]
  References the Sending of the Message.
- receiveEvent: MessageEnd[0..1]
  References the Receiving of the Message.
- connector: Connector[0..1]
  The Connector on which this Message is sent.
- argument:ValueSpecification[*]
  The arguments of the Message.
- /signature:NamedElement[0..1]
  The definition of the type or signature of the Message (depending on its kind). The associated named element is derived from the message end that constitutes the sending or receiving message event. If both a sending event and a receiving message event are present, the signature is obtained from the sending event.

Constraints

[1] If the sending MessageEvent and the receiving MessageEvent of the same Message are on the same Lifeline, the sending MessageEvent must be ordered before the receiving MessageEvent.

[2] The signature must either refer to an Operation (in which case messageSort is either synchCall or asynchCall) 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.

[3] In the case when the Message signature is an Operation, the arguments of the Message must correspond to the 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.

[4] 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.

[5] 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.

[6] Messages cannot cross boundaries of CombinedFragments or their operands.

[7] If the MessageEnds are both OccurrenceSpecifications, then the connector must go between the Parts represented by the Lifelines of the two MessageEnds.

Semantics

The semantics of a *complete* Message is simply the trace <sendEvent, receiveEvent>.

A *lost* message is a message where the sending event occurrence is known, but there is no receiving event occurrence. We interpret this to be because the message never reached its destination. The semantics is simply the trace <sendEvent>.

A *found* message is a message where the receiving event occurrence is known, but there is no (known) sending event occurrence. We interpret this to be because the origin of the message is outside the scope of the description. This may for example be noise or other activity that we do not want to describe in detail. The semantics is simply the trace <receiveEvent>.

A Message reflects either an Operation call and start of execution or a sending and reception of a Signal.

When a Message represents an Operation invocation, the arguments of the Message are the arguments of the CallOperationAction on the sending Lifeline as well as the arguments of the CallEvent occurrence on the receiving Lifeline.

When a Message represents a Signal, the arguments of the Message are the arguments of the SendAction on the sending Lifeline and on the receiving Lifeline the arguments are available in the SignalEvent.

If the Message represents a CallAction, there will normally be a reply message from the called Lifeline back to the calling lifeline before the calling Lifeline will proceed.

Notation

A message is shown as a line from the sender message end to the receiver message end. 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:

- Asynchronous Messages have an open arrow head.
- Synchronous Messages typically represent operation calls and are shown with a filled arrow head.
- The reply message from a method has a dashed line.
- Object creation Message has a dashed line with an open arrow.
- Lost Messages are described as a small black circle at the arrow end of the Message.
- Found Messages are described as 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 14.4 and Figure 14.27).

Syntax for the Message name is the following:

<messageident> ::= ([<attribute> '='] <signal-or-operation-name> ['(' [<argument> [','<argument>] ')']*
[':' <return-value>]) | ''*

<argument> ::= ([<parameter-name> '='] <argument-value>) | (<attribute> '=' <out-parameter-name> [':' <argument-value>]) | ' -'

Messageident equaling '*' is a shorthand for more complex alternative CombinedFragment to represent a message of any type. This is to match asterisk triggers in State Machines.

Return-value and attribute assignment are used only for reply messages. Attribute assignment is a shorthand for including the Action that assigns the return-value to that attribute. This holds both for the possible return value of the message (the return value of the associated operation), and the out values of (in)out parameters.

When the argument list contains only argument-values, all the parameters must be matched either by a value or by a dash (-). If parameter-names are used to identify the argument-value, then arguments may freely be omitted. Omitted parameters get an unknown argument-value.

Examples

In Figure 14.16 we see only asynchronous Messages. Such Messages may overtake each other.

In Figure 14.11 we see method calls that are synchronous accompanied by replies. We also see a Message that represents the creation of an object.

In Figure 14.27 we see how Messages are denoted in Communication Diagrams.

Examples of syntax:

```
mymessage(14, - , 3.14, "hello")    // second argument is undefined
v=mymsg(16, variab):96              // this is a reply message carrying the return value 96 assigning it to v
mymsg(myint=16)                     // the input parameter 'myint' is given the argument value 16
```

See Figure 14.11 for a number of different applications of the textual syntax of message identification.

Changes from previous UML

Messages may have Gates on either end.

14.3.21 MessageEnd (from BasicInteractions)

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 98.

Description

A MessageEnd is an abstract NamedElement that represents what can occur at the end of a Message.

Associations

- message : Message [0..1]
  References a Message.

Semantics

Subclasses of MessageEnd define the specific semantics appropriate to the concept they represent.

14.3.22 MessageKind (from BasicInteractions)

This is an enumerated type that identifies the type of message.

Generalizations

None

Description

MessageKind is an enumeration of the following values:

- 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).

14.3.23 MessageOccurrenceSpecification (from BasicInteractions)

Generalizations

- “MessageEnd (from BasicInteractions)” on page 496
- “OccurrenceSpecification (from BasicInteractions)” on page 498

Description

Specifies the occurrence of events, such as sending and receiving of signals or invoking or receiving of operation calls. A message occurrence specification is a kind of message end. Messages are generated either by synchronous operation calls or asynchronous signal sends. They are received by the execution of corresponding accept event actions.

Attributes

No additional attributes

Associations

- event : Event [1]
  The event associated with the message occurrence.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.24 MessageSort (from BasicInteractions)

This is an enumerated type that identifies the type of communication action that was used to generate the message.

Generalizations

None

Description

MessageSort is an enumeration of the following values:

- 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.

14.3.25 OccurrenceSpecification (from BasicInteractions)

Generalizations

- "InteractionFragment (from BasicInteractions, Fragments)" on page 487

Description

An OccurrenceSpecification is the basic semantic unit of Interactions. The sequences of occurrences specified by them are the meanings of Interactions.

OccurrenceSpecifications are ordered along a Lifeline.

The namespace of an OccurrenceSpecification is the Interaction in which it is contained.

Associations

- event : Event [1]
  References a specification of the occurring event.
- covered: Lifeline[1]
  References the Lifeline on which the OccurrenceSpecification appears. Redefines InteractionFragment.covered.
- toBefore:GeneralOrdering[*]
  References the GeneralOrderings that specify EventOcurrences that must occur before this OccurrenceSpecification.
- toAfter: GeneralOrdering[*]
  References the GeneralOrderings that specify EventOcurrences that must occur after this OccurrenceSpecification.

Semantics

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.

Notation

OccurrenceSpecifications are merely syntactic points at the ends of Messages or at the beginning/end of an ExecutionSpecification.

Examples

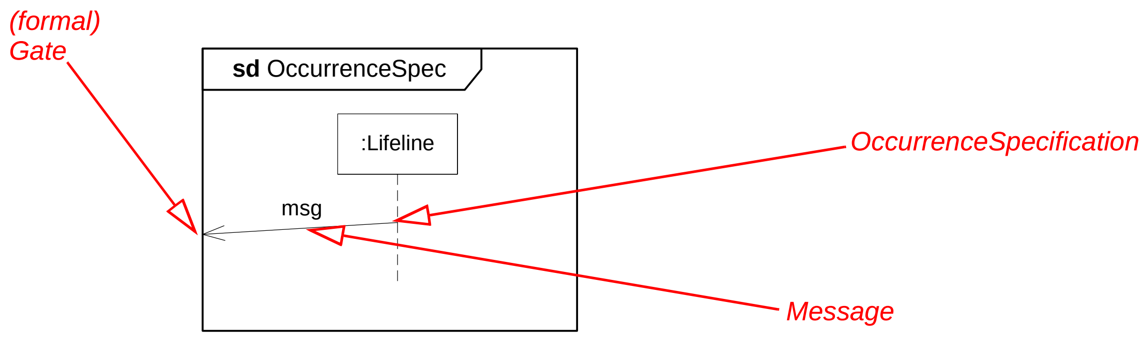


Figure 14.19 - OccurrenceSpecification

14.3.26 PartDecomposition (from Fragments)

Generalizations

- "InteractionUse (from Fragments)" on page 489

Description

PartDecomposition is a description of the internal interactions of one Lifeline relative to an Interaction.

A Lifeline has a class associated as the type of the ConnectableElement that the Lifeline represents. That class may have an internal structure and the PartDecomposition is an Interaction that describes the behavior of that internal structure relative to the Interaction where the decomposition is referenced.

A PartDecomposition is a specialization of InteractionUse. It associates with the ConnectableElement that it decomposes.

Constraints

[1] PartDecompositions apply only to Parts that are Parts of Internal Structures not to Parts of Collaborations.

[2] 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 are 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.

[3] 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.)

Semantics

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.

Since 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.

Notation

PartDecomposition is designated by a referencing clause in the head of the Lifeline as can be seen in the notation sub clause of "Lifeline (from BasicInteractions, Fragments)" on page 492 (see also Figure 14.20).

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 "CombinedFragment (from Fragments)" on page 469 on the "strict" operator).

Extraglobal 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 14.20 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.

Examples

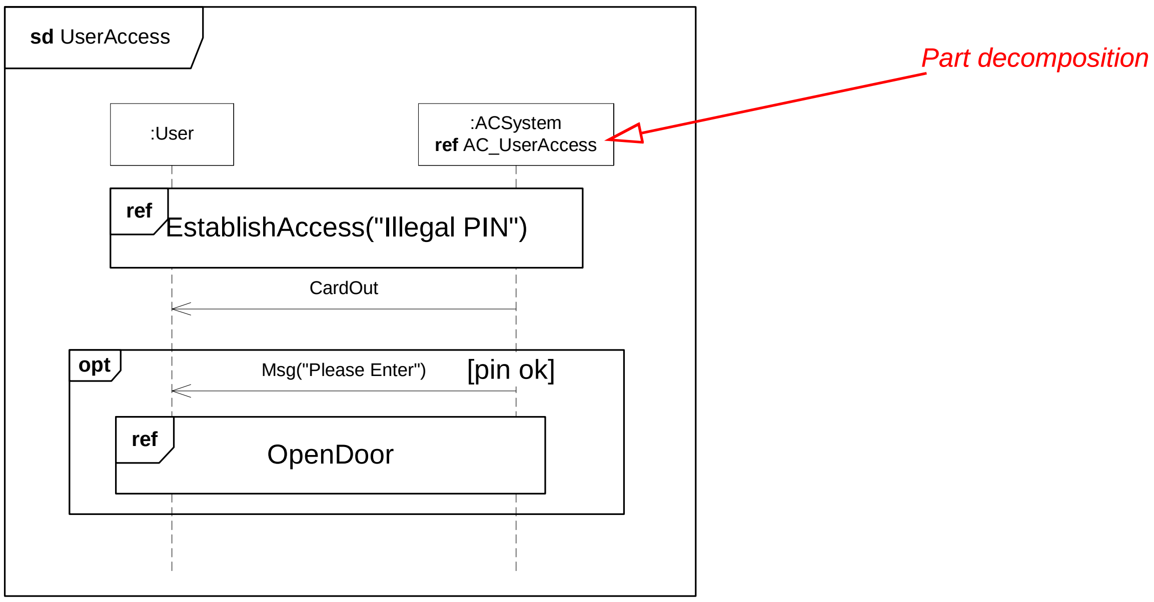


Figure 14.20 - Part Decomposition - the decomposed part

In Figure 14.20 we see how *ACSystem* within *UserAccess* is to be decomposed to *AC_UserAccess*, which is an Interaction owned by class *ACSystem*.

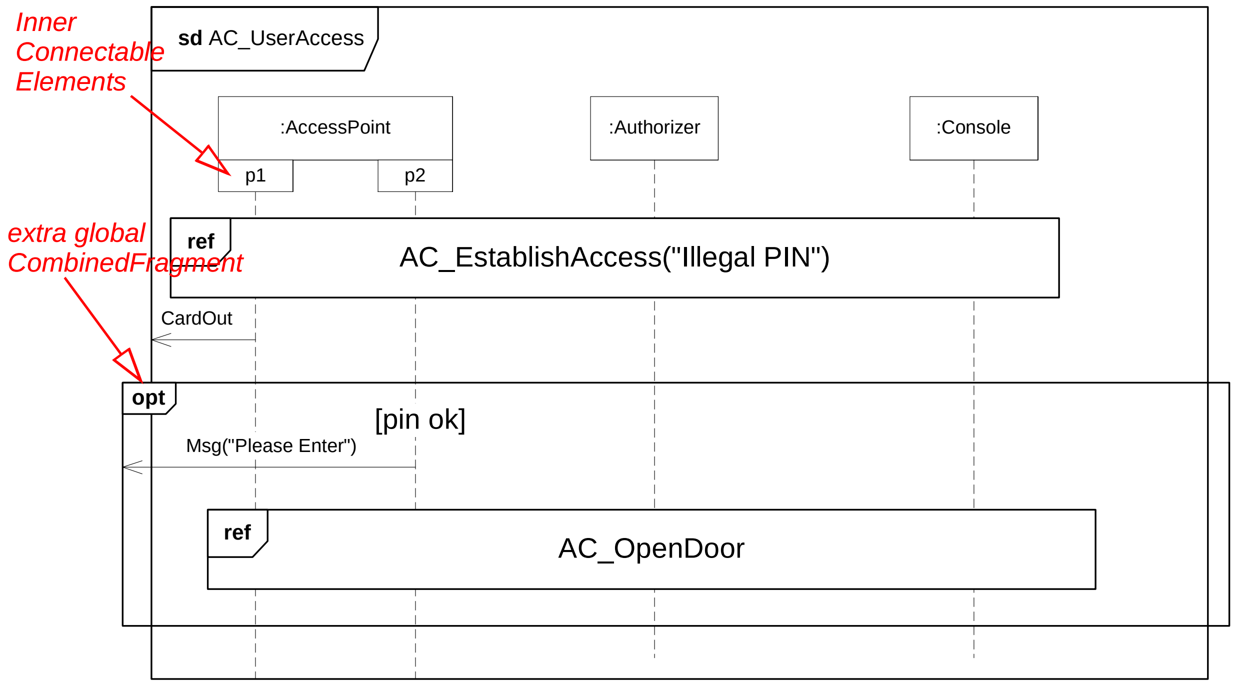


Figure 14.21 - PartDecomposition - the decomposition

In Figure 14.21 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 14.21 we see that the inner ConnectableElements of *:AccessPoint* (p1 and p2) are represented by Lifelines already on this level.

Changes from previous UML

PartDecomposition did not appear in UML 1.x.

14.3.27 ReceiveOperationEvent (from BasicInteractions)

Generalizations

- "MessageEvent (from Communications)" on page 445

Description

This specifies the event of receiving an operation invocation for a particular operation by the target entity.

Attributes

No additional attributes

Associations

- operation::Operation[1]
  Specifies the operation associated with this event.

Constraints

No additional constraints

Semantics

A receive operation event occurs when an operation invocation is received by the target object.

Notation

None

Changes from previous UML

New in UML 2.

14.3.28 ReceiveSignalEvent (from BasicInteractions)

Generalizations

- "MessageEvent (from Communications)" on page 445

Description

This specifies the event of receiving signal by the target entity

Attributes

No additional attributes

Associations

- signal::Signal [1]
  Specifies the signal associated with this event.

Constraints

No additional constraints

Semantics

A receive signal event occurs when a signal is received by the target object.

Notation

None

Changes from previous UML

New in UML 2.

14.3.29 SendOperationEvent (from BasicInteractions)

Generalizations

- "MessageEvent (from Communications)" on page 445

Description

A SendOperationEvent models the invocation of an operation call.

Attributes

No additional attributes

Associations

- operation : Operation [1]
  The operation associated with this event.

Constraints

No additional constraints

Semantics

A send operation event specifies the sending of a request to invoke a specific operation on an object. The send operation event may result in the occurrence of a call event in the receiver object (see "Common Behaviors" on page 421), and may consequentially cause the execution of a behavior by the receiver object.

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.30 SendSignalEvent (from BasicInteractions)

Generalizations

- "MessageEvent (from Communications)" on page 445

Description

A SendSignalEvent models the sending of a signal.

Attributes

No additional attributes

Associations

- signal : Signal [1]
  The signal associated with this event.

Constraints

No additional constraints

Semantics

A send signal event specifies the sending of a message to a receiver object. The send signal event may result in the occurrence of a signal event in the receiver object (see "Common Behaviors" on page 421), and may consequentially cause the execution of a behavior by the receiver object. The sending object will not block waiting for a reply, but will continue its execution immediately.

Notation

None

Changes from previous UML

New in UML 2.0.

14.3.31 StateInvariant (from BasicInteractions)

Generalizations

- "InteractionFragment (from BasicInteractions, Fragments)" on page 487

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.

Associations

- invariant: Constraint[1]
  A Constraint that should hold at runtime for this StateInvariant.

- covered: Lifeline[1]
  References the Lifeline on which the StateInvariant appears. Subsets InteractionFragment.covered.

Semantics

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.

Notation

The possible associated Constraint is shown as text in curly brackets on the lifeline. See example in Figure 14.24 on page 513.

Presentation Options

A StateInvariant can optionally be shown 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 14.24 also shows this presentation option.

14.4 Diagrams

Interaction diagrams come in different variants. The most common variant is the Sequence Diagram ("Sequence Diagrams" on page 506) that focuses on the Message interchange between a number of Lifelines. Communication Diagrams ("Communication Diagrams" on page 515) show interactions through an architectural view where the arcs between the communicating Lifelines are decorated with description of the passed Messages and their sequencing. Interaction Overview Diagrams ("Interaction Overview Diagrams" on page 518) define interactions in a way that promotes overview of the control flow. In the Annexes one may also find optional diagram notations such as Timing Diagrams and Interaction Tables. (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).

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. The Interactions that are described by Sequence Diagrams are described in this clause.

Graphic Nodes

The graphic nodes that can be included in sequence diagrams are shown in Table 14.1.

Table 14.1 - Graphic nodes included in sequence diagrams

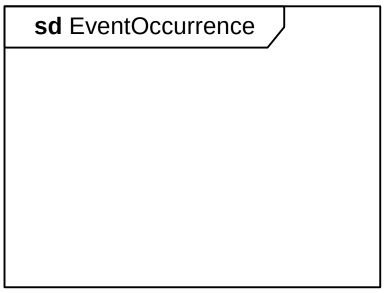
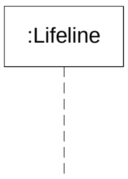
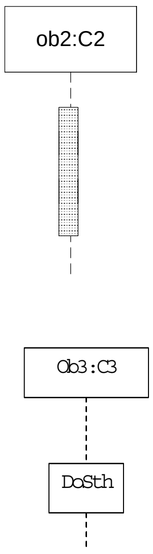
Node Type	Notation	Reference
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See "Interaction (from BasicInteraction, Fragments)" on page 483.
Lifeline		See "Lifeline (from BasicInteractions, Fragments)" on page 492.
Execution Specification		See "CombinedFragment (from Fragments)" on page 469. See also "Lifeline (from BasicInteractions, Fragments)" on page 492 and "ExecutionSpecification (from BasicInteractions)" on page 480.

Table 14.1 - Graphic nodes included in sequence diagrams

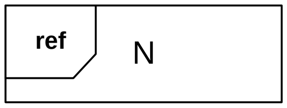
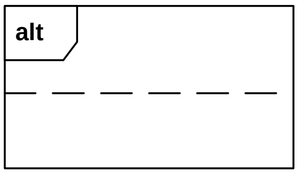
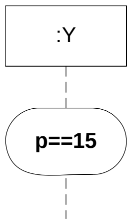
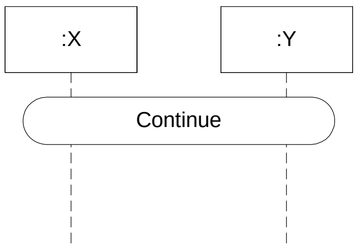
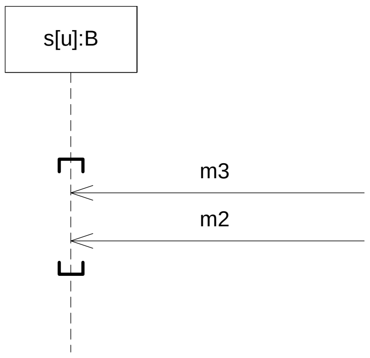
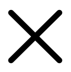
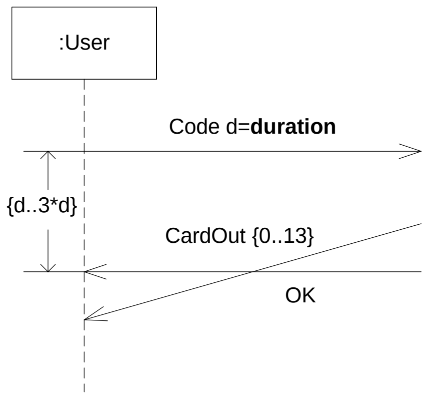
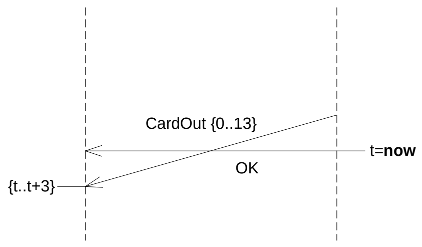
Node Type	Notation	Reference
InteractionUse		See "InteractionUse (from Fragments)" on page 489.
CombinedFragment		See "CombinedFragment (from Fragments)" on page 469.
StateInvariant		See "StateInvariant (from BasicInteractions)" on page 505.
Continuations		See "Continuation (from Fragments)" on page 475.
Coregion		See explanation under *parallel* in "CombinedFragment (from Fragments)" on page 469.

Table 14.1 - Graphic nodes included in sequence diagrams

Node Type	Notation	Reference
DestructionEvent		See Figure 14.11 on page 474.
DurationConstraint Duration Observation	 :User Code d=**duration** {d..3*d} CardOut {0..13} OK	See Figure 14.26 on page 515.
TimeConstraint TimeObservation		See Figure 14.26 on page 515.

Graphic Paths

The graphic paths between the graphic nodes are given in Table 14.2.

Table 14.2 - Graphic paths included in sequence diagrams

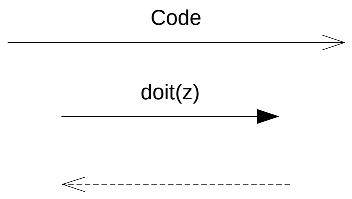
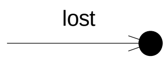
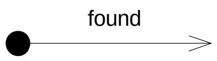
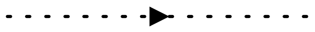
Node Type	Notation	Reference
Message		Messages come in different variants depending on what kind of Message they convey. Here we show an asynchronous message, a call and a reply. These are all *complete* messages. See “Message (from BasicInteractions)” on page 493.

Table 14.2 - Graphic paths included in sequence diagrams

Node Type	Notation	Reference
Lost Message		Lost messages are messages with known sender, but the reception of the message does not happen. See "Message (from BasicInteractions)" on page 493.
Found Message		Found messages are messages with known receiver, but the sending of the message is not described within the specification. See "Message (from BasicInteractions)" on page 493.
GeneralOrdering		See "GeneralOrdering (from BasicInteractions)" on page 482.

Examples

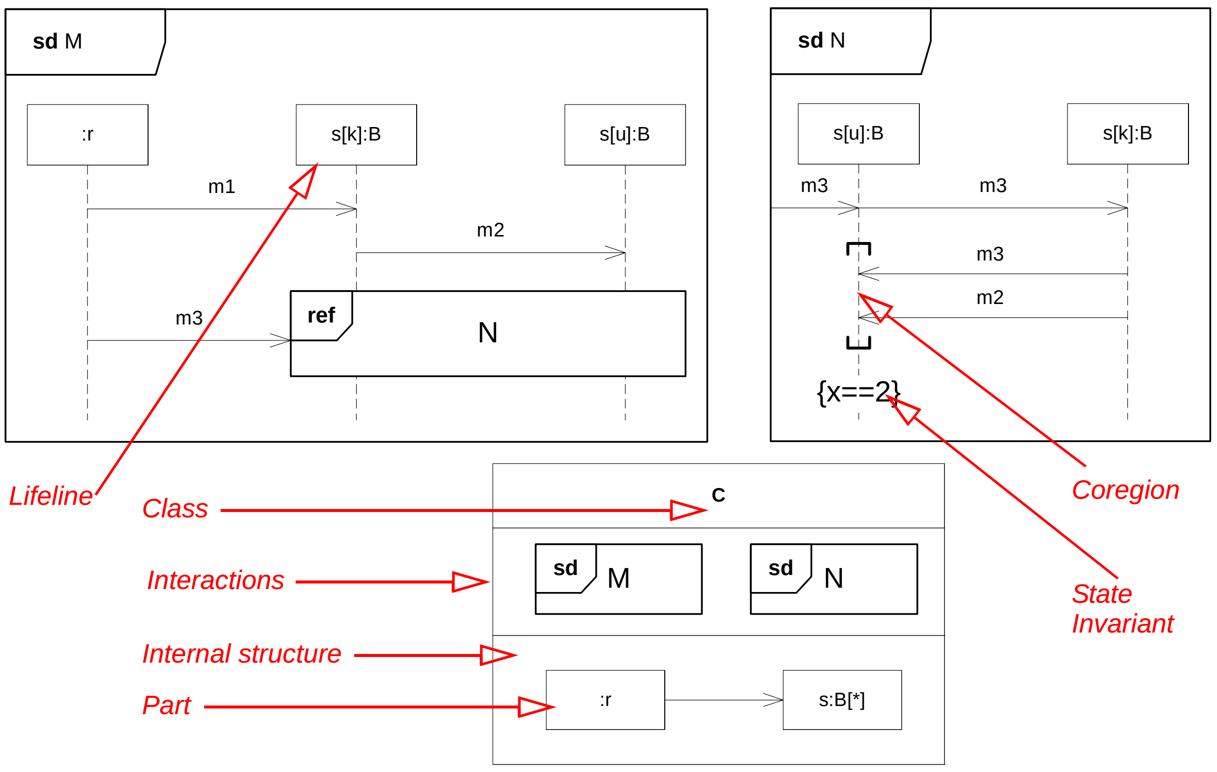


Figure 14.22 - Sequence Diagrams where two Lifelines refer to the same set of Parts (and Internal Structure)

The sequence diagrams shown in Figure 14.22 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]).

In order to explain the mapping of the notation onto the metamodel we have pointed out areas and their corresponding metamodel concept in Figure 14.23. 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.

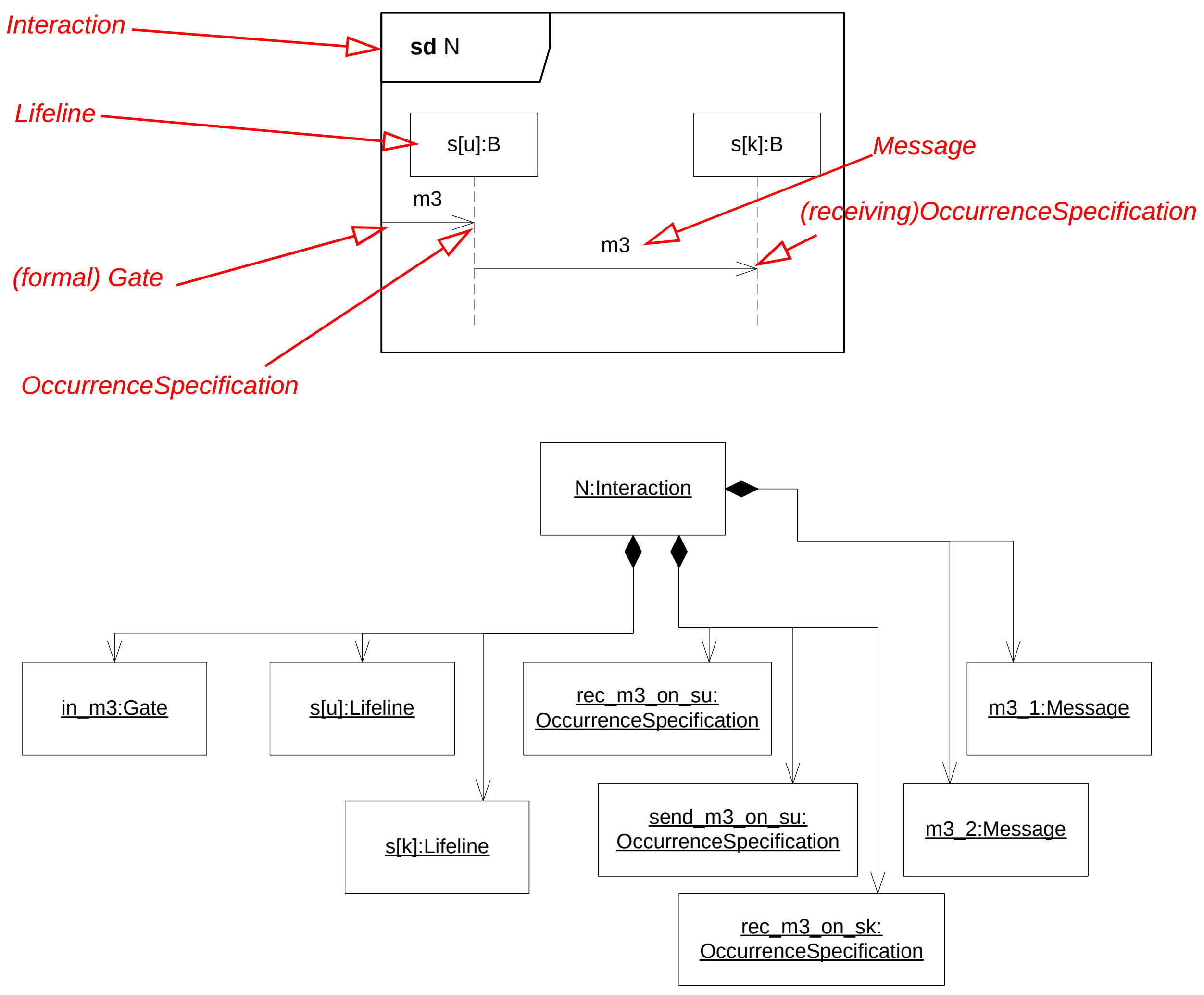


Figure 14.23 - Metamodel elements of a sequence diagrams

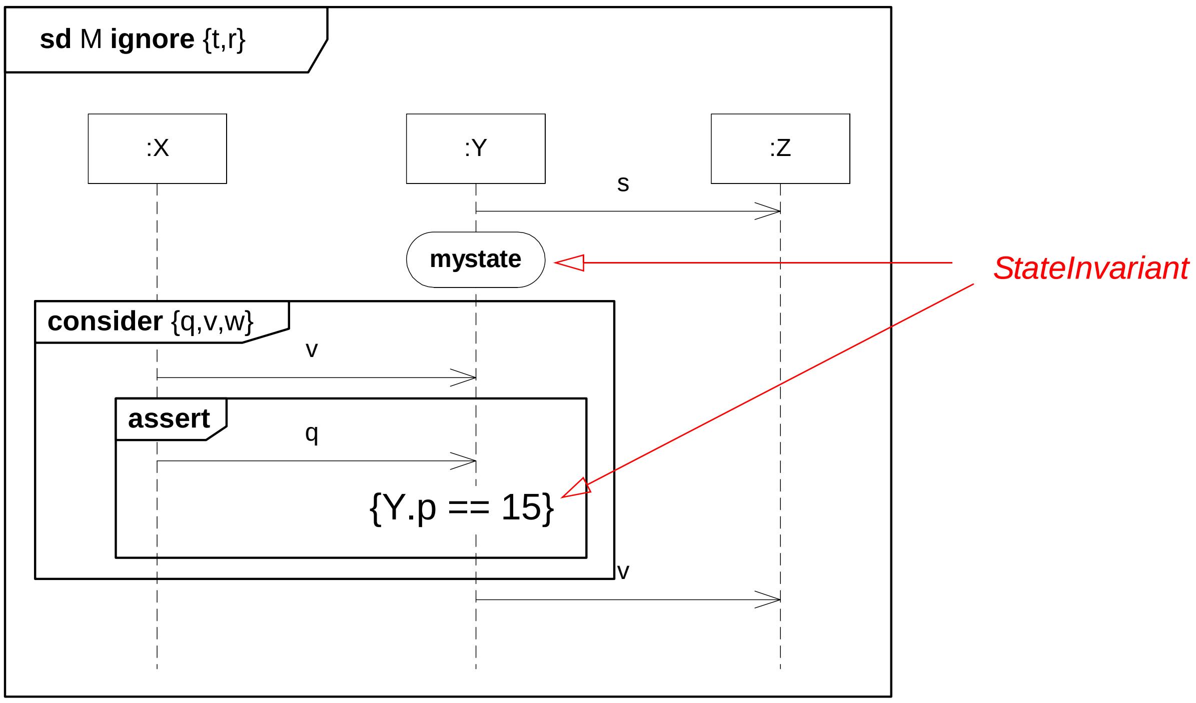


Figure 14.24 - Ignore, Consider, assert with State Invariants

In Figure 14.24 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.

Internal Structure and corresponding Collaboration Uses

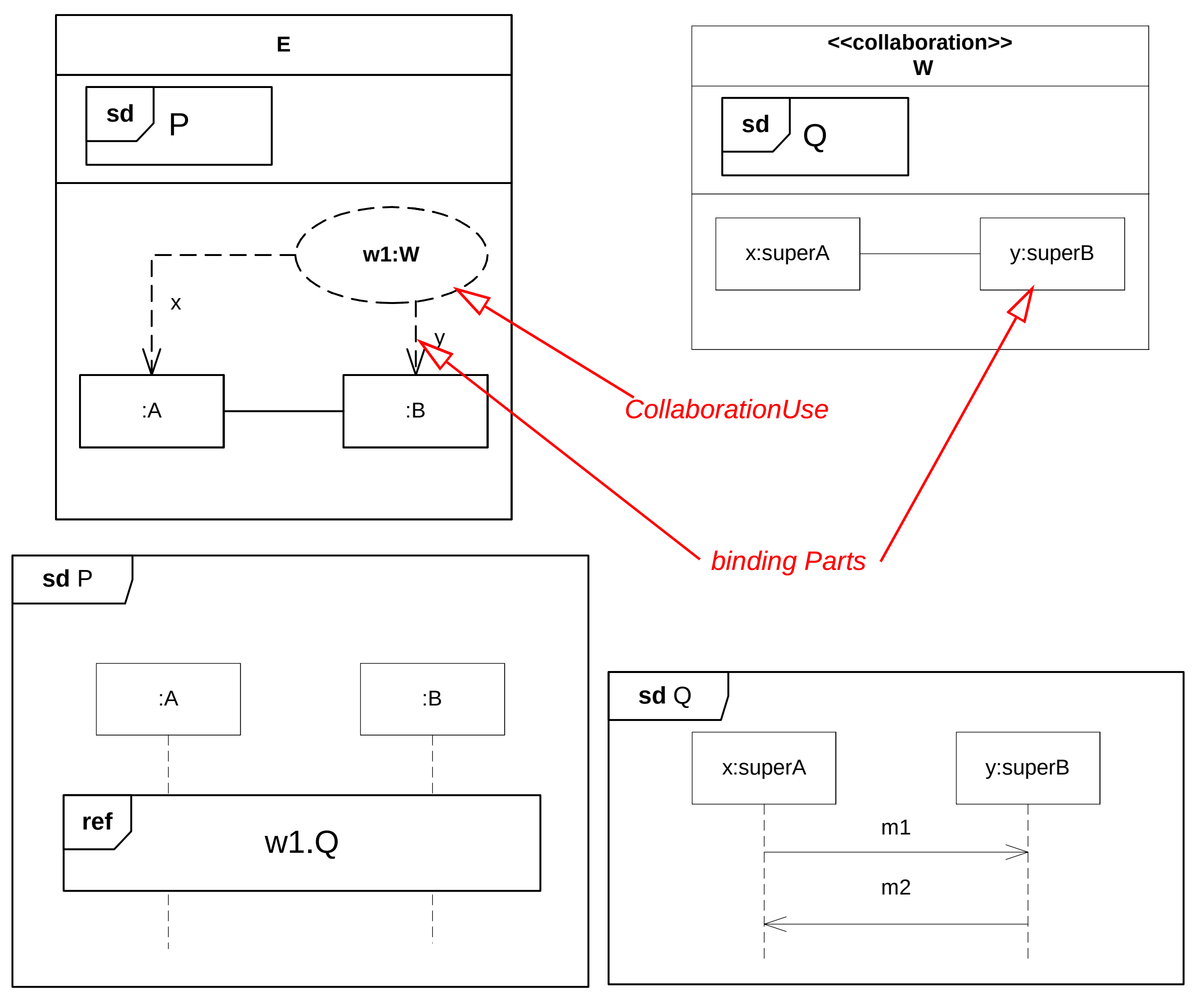


Figure 14.25 - Describing collaborations and their binding

The example in Figure 14.25 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 since :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.

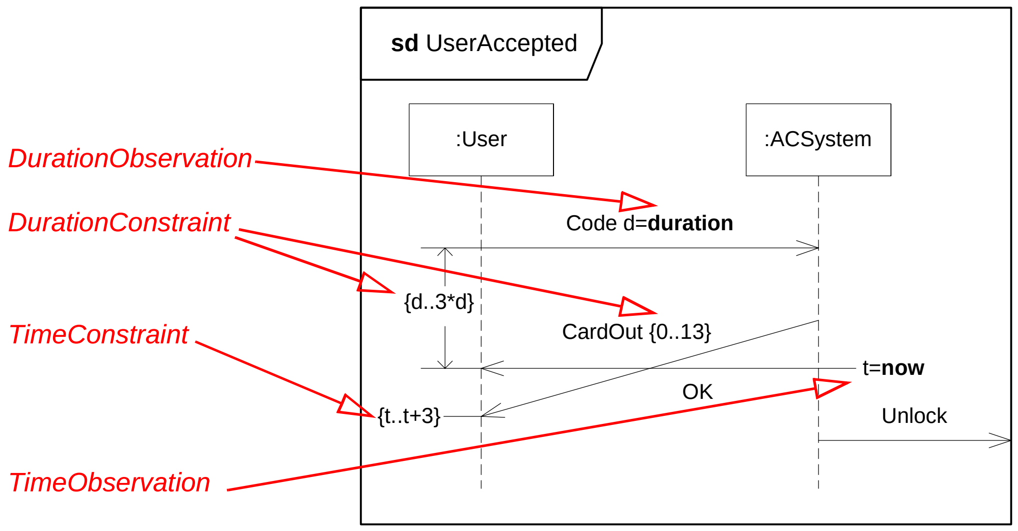


Figure 14.26 - Sequence Diagram with time and timing concepts

The Sequence Diagram in Figure 14.26 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.

Communication Diagrams

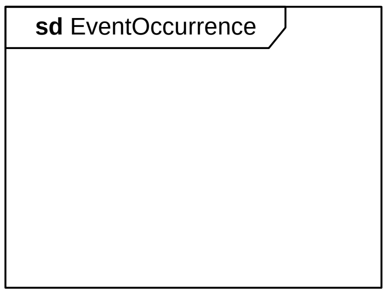
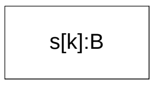
Communication Diagrams focus on the interaction between Lifelines where the architecture of the internal structure and how this corresponds with the message passing is central. The sequencing of Messages is given through a sequence numbering scheme.

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.

Graphical Nodes

Communication diagram nodes are shown in Table 14.3.

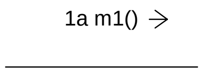
Table 14.3 - Graphic nodes included in communication diagrams

Node Type	Notation	Reference
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See "Interaction (from BasicInteraction, Fragments)" on page 483.
Lifeline		See "Lifeline (from BasicInteractions, Fragments)" on page 492.

Graphic Paths

Graphic paths of communication diagrams are given in Table 14.4.

Table 14.4 - Graphic paths included in communication diagrams

Node Type	Notation	Reference
Message		See "Message (from BasicInteractions)" on page 493 and "Sequence expression" on page 517. The arrow shown here indicates the communication direction.

Examples

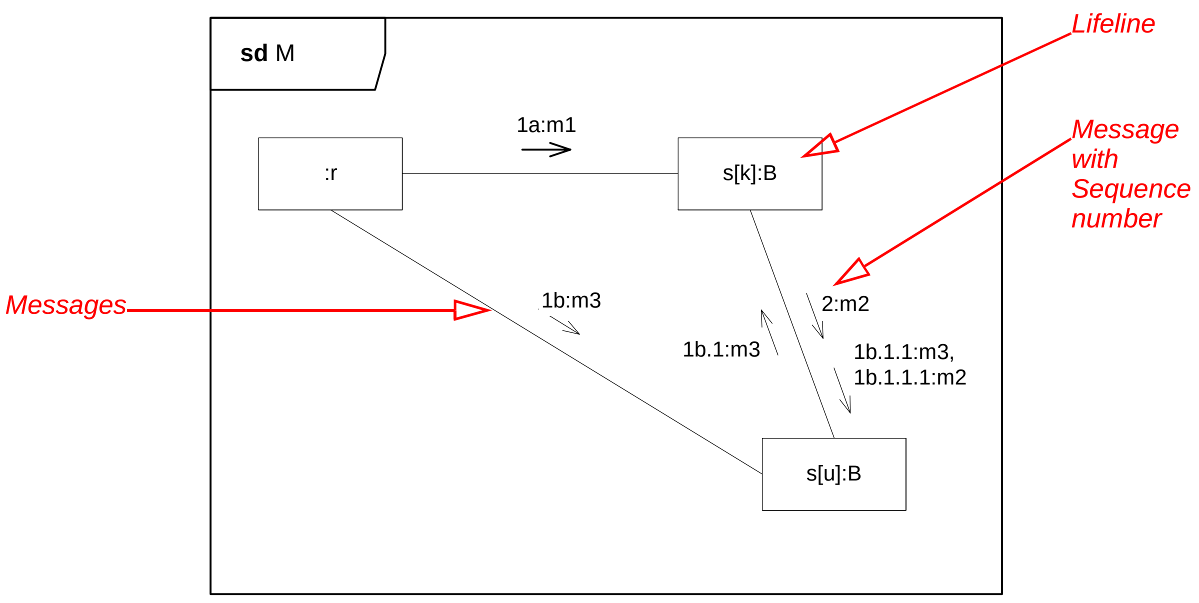


Figure 14.27 - Communication diagram

The Interaction described by a Communication Diagram in Figure 14.27 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.

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.

Interaction Overview Diagrams

Interaction Overview Diagrams define Interactions through a variant of Activity Diagrams (described in Clause 12, “Activities”) 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.

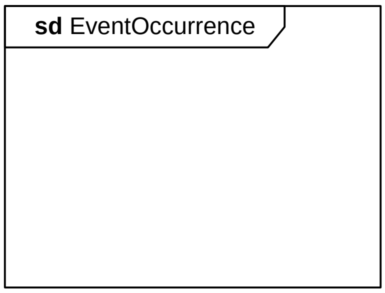
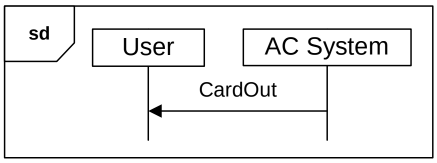
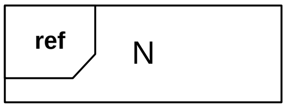
Graphic Nodes

Interaction Overview Diagrams are specialization of Activity Diagrams that represent Interactions. Interaction Overview Diagrams differ from Activity Diagrams in some respects.

1. 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.
2. Alternative Combined Fragments are represented by a Decision Node and a corresponding Merge Node.
3. Parallel Combined Fragments are represented by a Fork Node and a corresponding Join Node.
4. Loop Combined Fragments are represented by simple cycles.
5. Branching and joining of branches must in Interaction Overview Diagrams be properly nested. This is more restrictive than in Activity Diagrams.
6. 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 14.5 - Graphic nodes included in Interaction Overview Diagrams in addition to those borrowed from Activity Diagrams

Node Type	Notation	Reference
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See “Interaction (from BasicInteraction, Fragments)” on page 483.
Interaction		An Interaction diagram of any kind may appear inline as an ActivityInvocation. See “Interaction (from BasicInteraction, Fragments)” on page 483. The inline Interaction diagrams may be either anonymous (as here) or named.
InteractionUse		ActivityInvocation in the form of InteractionUse. See “InteractionUse (from Fragments)” on page 489. 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.

Examples

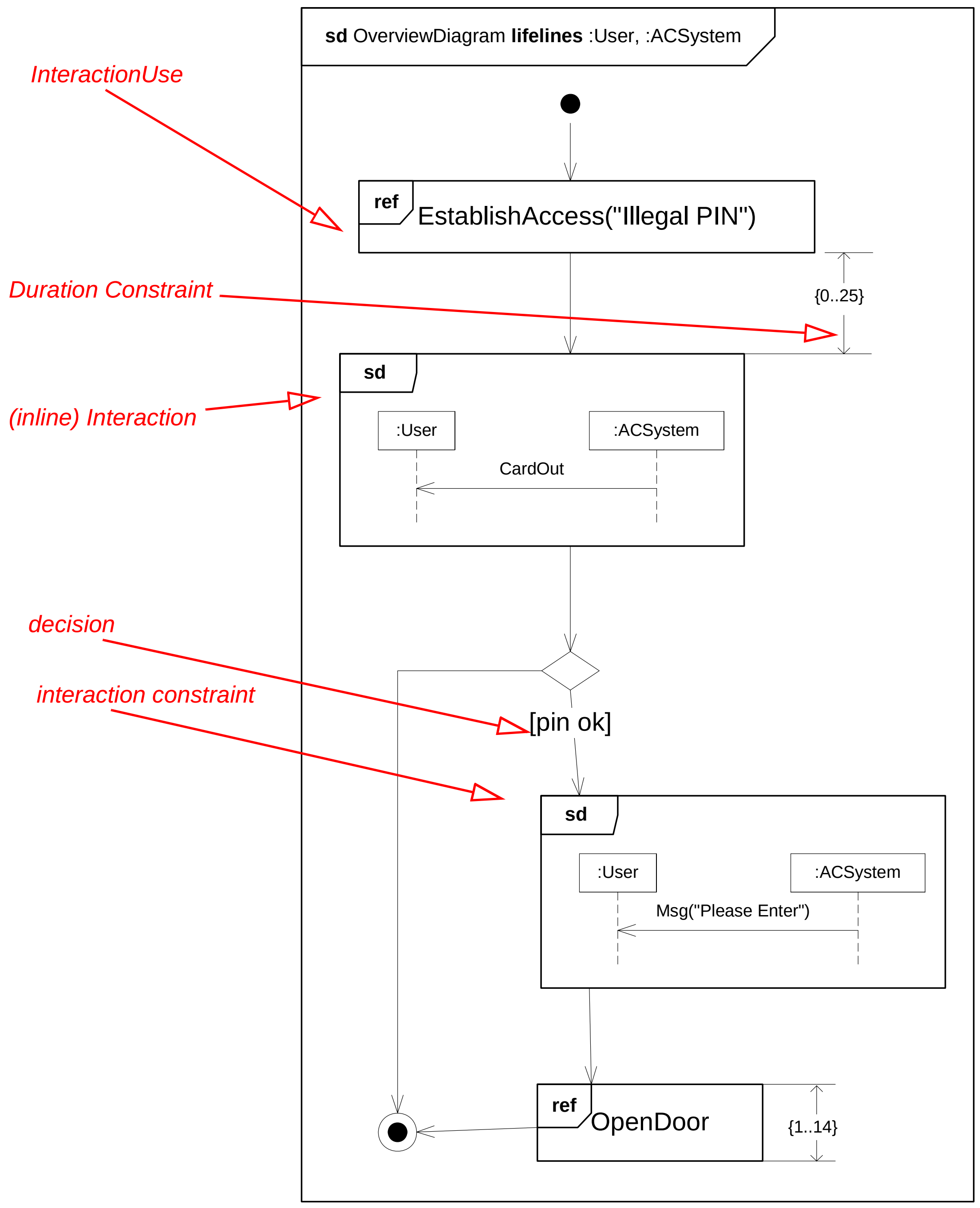


Figure 14.28 - Interaction Overview Diagram representing a High Level Interaction diagram

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.

Figure 14.28 is another way to describe the behavior shown in Figure 14.17, 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.

Timing Diagram

Timing Diagrams are used to show interactions when a primary purpose of the diagram is to reason about time. 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.

Graphic Nodes

The graphic nodes and paths that can be included in timing diagrams are shown in Table 14.6.

Table 14.6 - Graphic nodes and paths included in timing diagrams

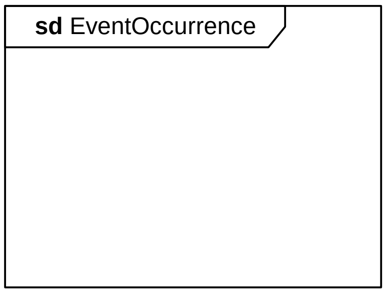
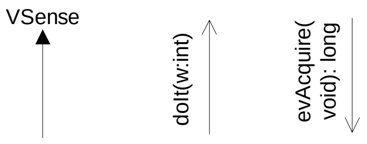
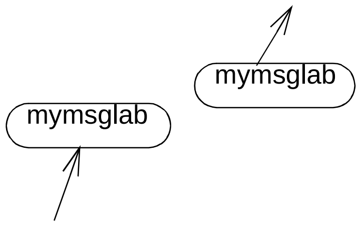
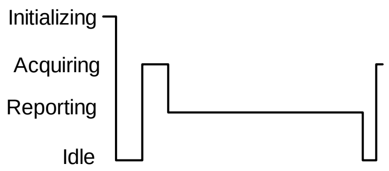
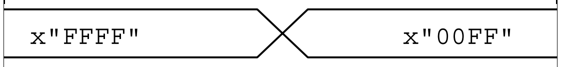
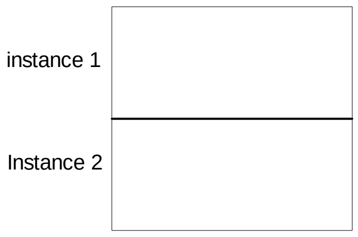
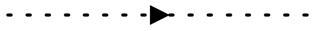
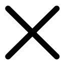
Node Type	Notation	Reference
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See "Interaction (from BasicInteraction, Fragments)" on page 483.
Message		Messages come in different variants depending on what kind of Message they convey. Here we show an asynchronous message, a call and a reply. See "Message (from BasicInteractions)" on page 493.

Table 14.6 - Graphic nodes and paths included in timing diagrams

Node Type	Notation	Reference
Message label		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.
State or condition timeline		This is the state of the classifier or attribute, or some testable condition, such as an discrete enumerable value. See also “StateInvariant (from BasicInteractions)” on page 505. It is also permissible to let the state-dimension be continuous as well as discrete. This is 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 “Lifeline (from BasicInteractions, Fragments)” on page 492.
GeneralOrdering		See “GeneralOrdering (from BasicInteractions)” on page 482.
DestructionEvent		See “DestructionEvent (from BasicInteractions)” on page 478.

Examples

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 14.26 an example in Sequence Diagram that we will also give in Timing Diagrams.

The :User of the Sequence Diagram in Figure 14.26 is depicted with a simple Timing Diagram in Figure 14.29.

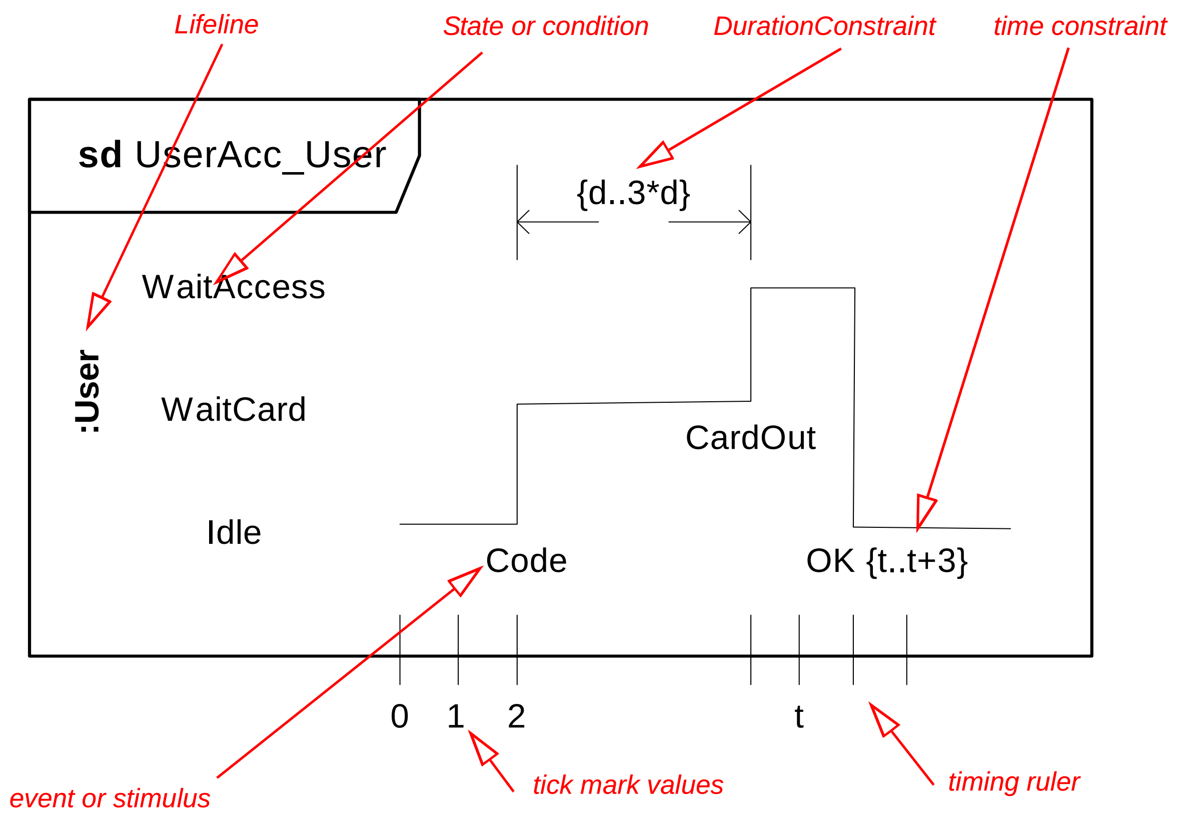


Figure 14.29 - A Lifeline for a discrete object

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 14.30.

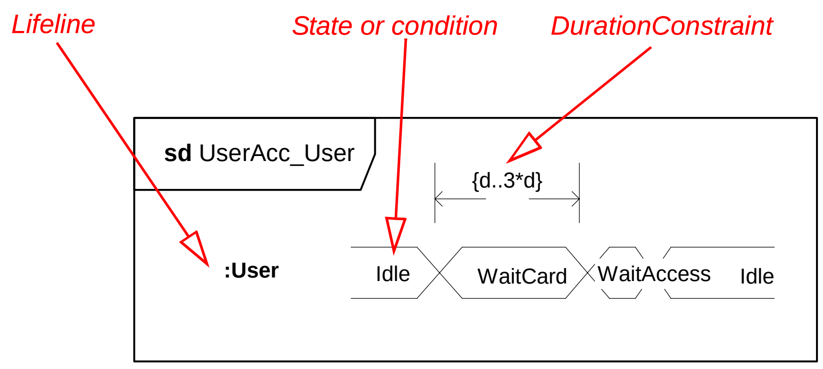


Figure 14.30 - Compact Lifeline with States

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 14.31 corresponding to the Sequence Diagram in Figure 14.26.

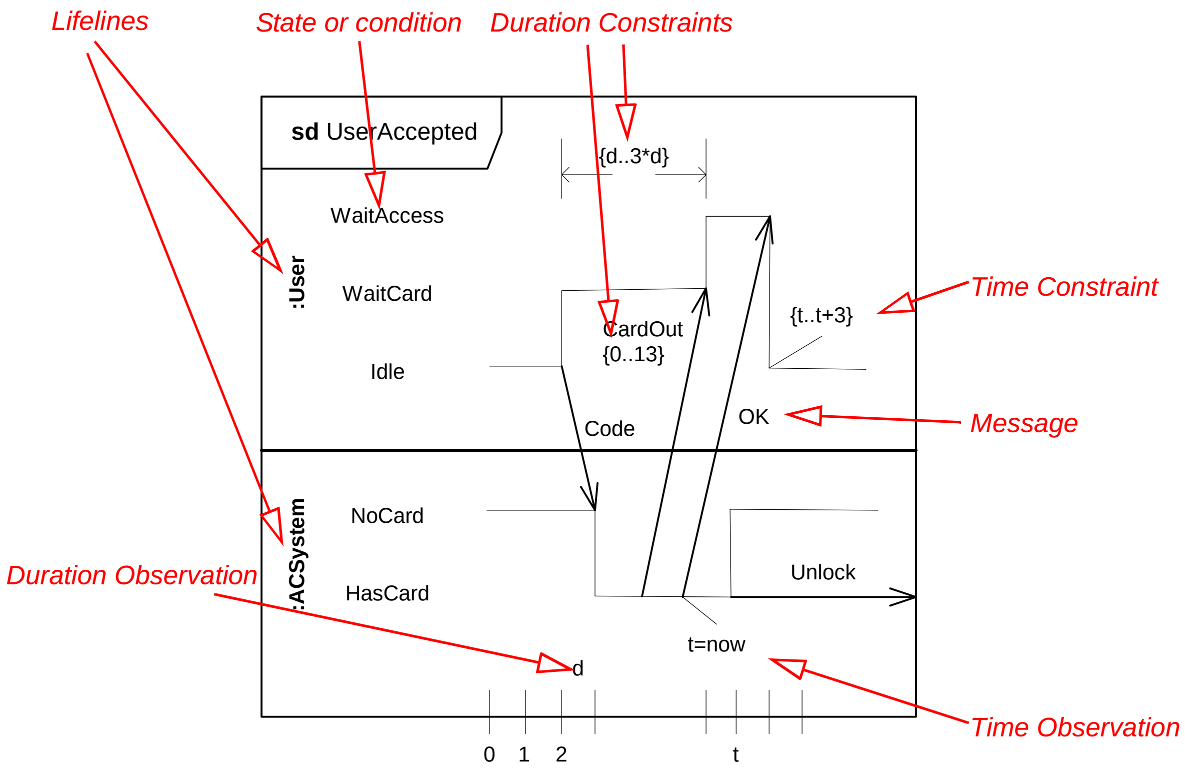


Figure 14.31 - Timing Diagram with more than one Lifeline and with Messages

Changes from previous UML

The Timing Diagrams were not available in UML 1.4.

15 State Machines

15.1 Overview

The StateMachine package defines a set of concepts that can be used for modeling discrete behavior through finite state-transition systems. In addition to expressing the behavior of a part of the system, state machines can also be used to express the usage protocol of part of a system. These two kinds of state machines are referred to here as *behavioral state machines* and *protocol state machines*.

Behavioral state machines

State machines can be used to specify behavior of various model elements. For example, they can be used to model the behavior of individual entities (e.g., class instances). The state machine formalism described in this sub clause is an object-based variant of Harel statecharts.

Protocol State machines

Protocol state machines are used to express usage protocols. Protocol state machines express the legal transitions that a classifier can trigger. The state machine notation is a convenient way to define a lifecycle for objects, or an order of the invocation of its operation. Because protocol state machines do not preclude any specific behavioral implementation, and enforces legal usage scenarios of classifiers, interfaces, and ports can be associated to this kind of state machines.

15.2 Abstract Syntax

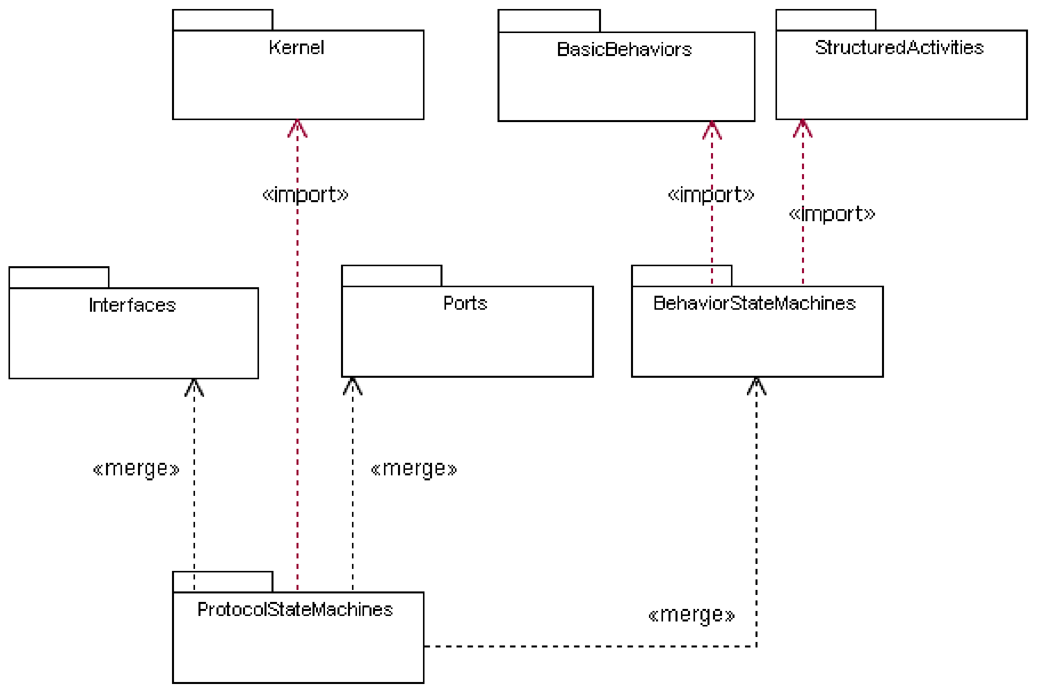


Figure 15.1 - Package Dependencies

Package BehaviorStateMachines

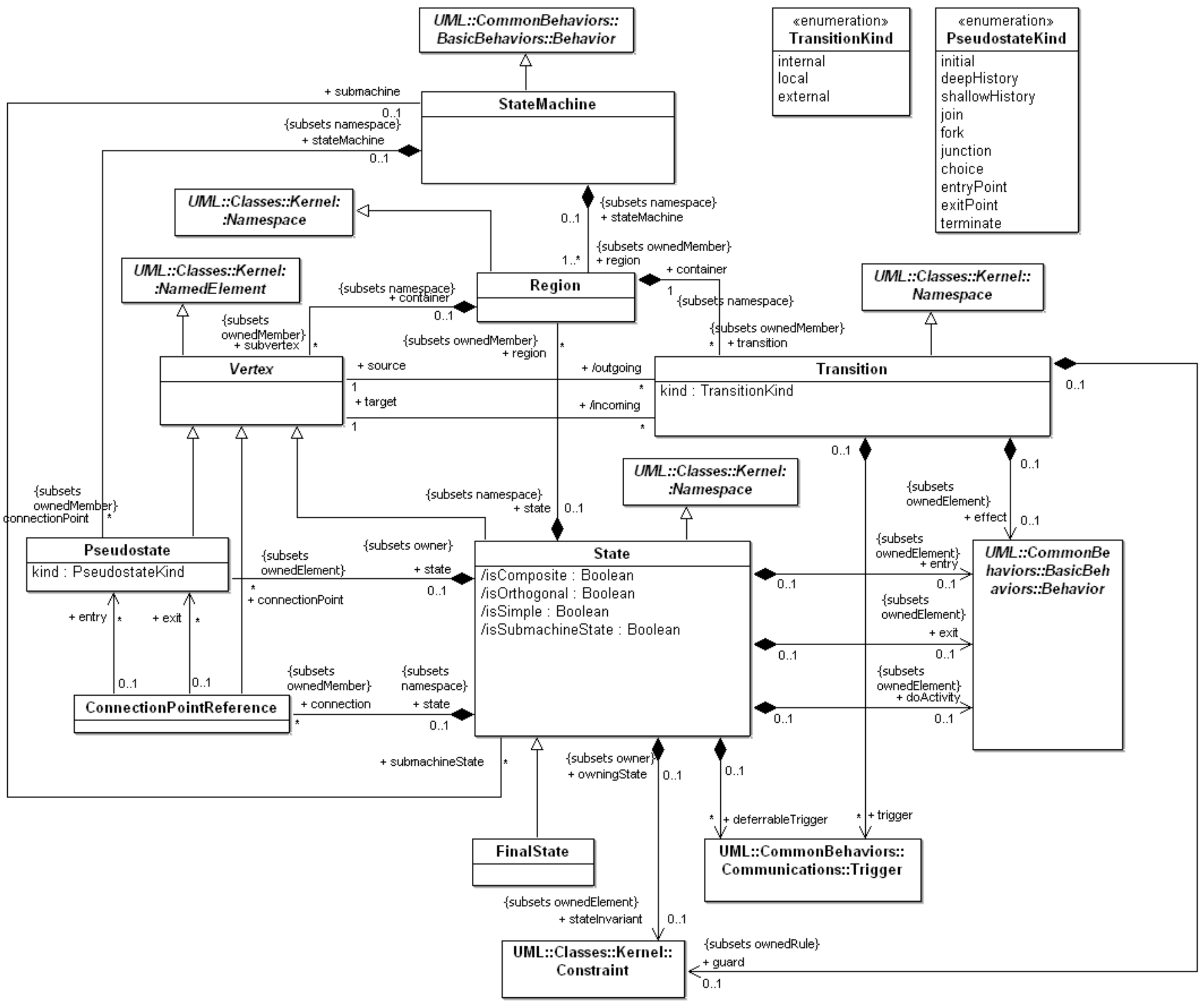


Figure 15.2 - State Machines

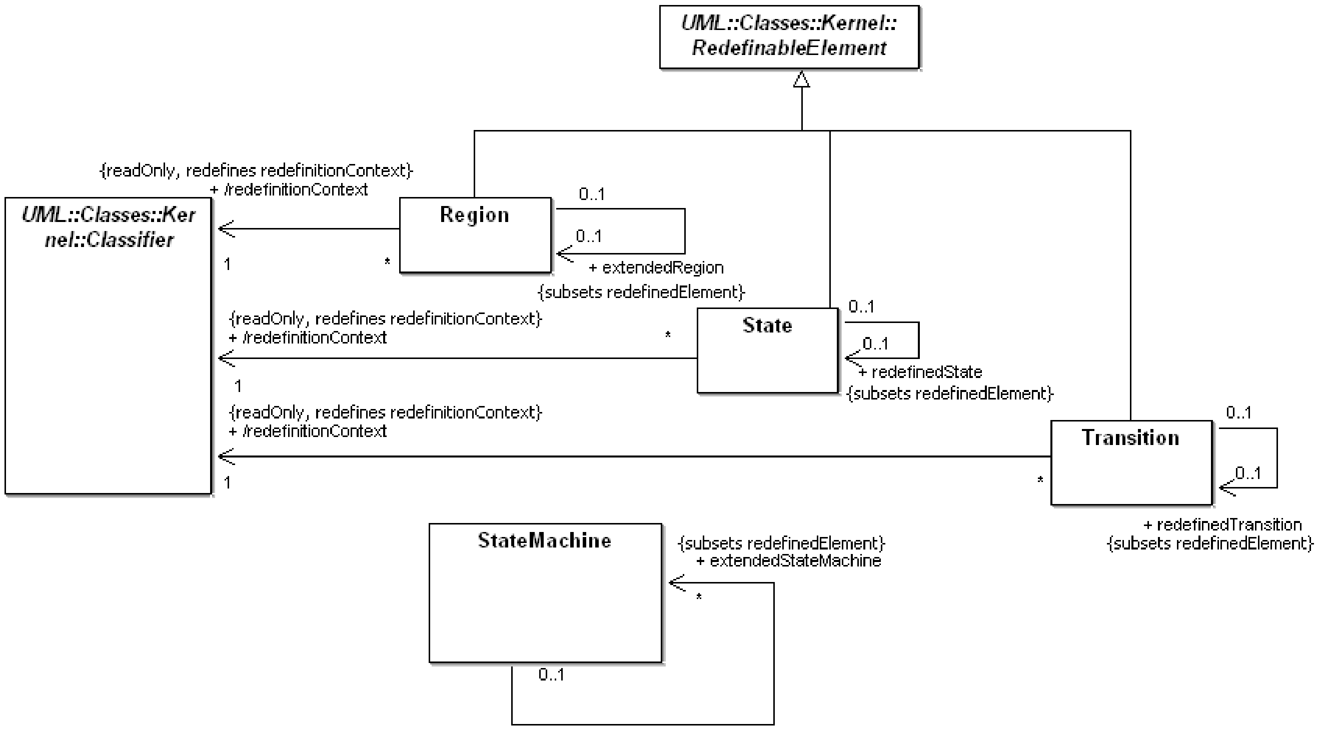

UML::Classes::Kernel::
RedefinableElement
{readOnly, redefines redefinitionContext}
+ /redefinitionContext
UML::Classes::Kernel::Classifier
1
*
Region
0..1
0..1
+ extendedRegion
{subsets redefinedElement}
{readOnly, redefines redefinitionContext}
+ /redefinitionContext
*
1
State
0..1
0..1
+ redefinedState
{subsets redefinedElement}
{readOnly, redefines redefinitionContext}
+ /redefinitionContext
1
*
Transition
0..1
0..1
+ redefinedTransition
{subsets redefinedElement}
StateMachine
{subsets redefinedElement}
+ extendedStateMachine
*
0..1

Figure 15.3 - State Machine Redefinitions

Figure 15.4 - Time events

Package ProtocolStateMachines

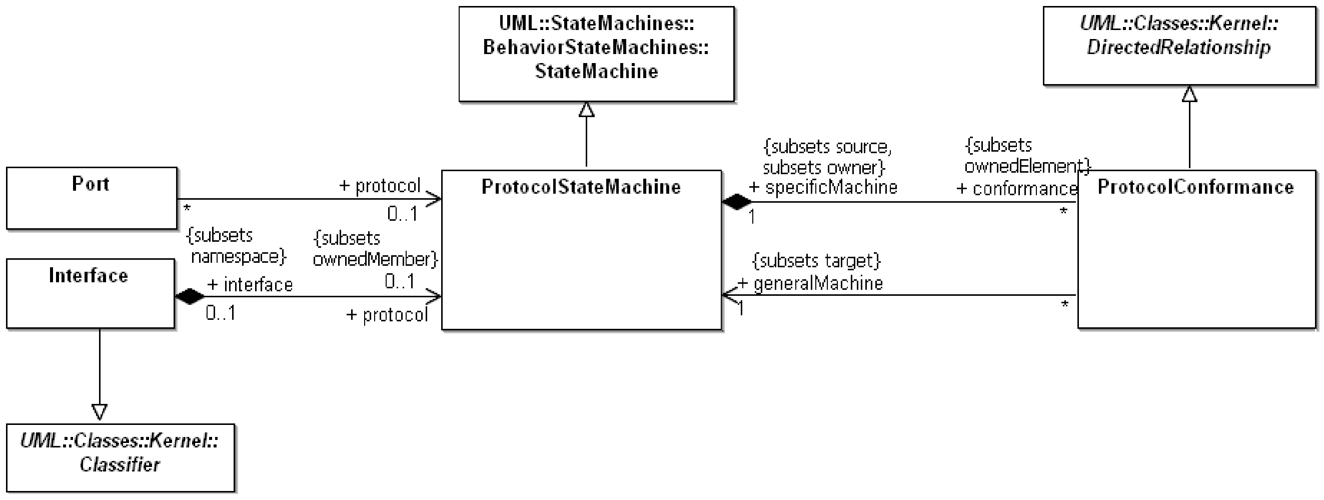


Figure 15.5 - Protocol State Machines

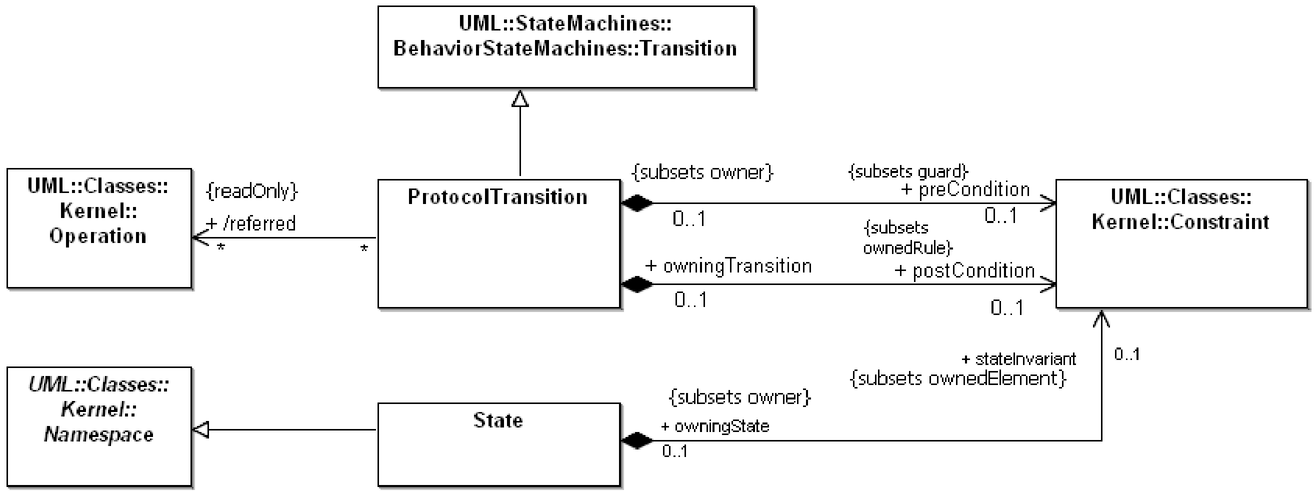


Figure 15.6 - Constraints

15.3 Class Descriptions

15.3.1 ConnectionPointReference (from BehaviorStateMachines)

A connection point reference represents a usage (as part of a submachine state) of an entry/exit point defined in the statemachine reference by the submachine state.

Generalizations

- “Vertex (from BehaviorStateMachines)” on page 581

Description

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 state machine referenced by the submachine state.

Attributes

No additional attributes

Associations

- entry: Pseudostate[0..*]
  The entryPoint kind pseudostates corresponding to this connection point.
- exit: Pseudostate[0..*]
  The exitPoints kind pseudostates corresponding to this connection point.
- state : State [0..1]
  The State in which the connection point references are defined. {Subsets *Element::namespace*}

Constraints

[1] The entry Pseudostates must be Pseudostates with kind entryPoint.

```
entry->notEmpty() implies entry->forAll(e | e.kind = #entryPoint)
```

[2] The exit Pseudostates must be Pseudostates with kind exitPoint.

```
exit->notEmpty() implies exit->forAll(e | e.kind = #exitPoint)
```

Semantics

Connection point references are sources/targets of transitions implying exits out of/entries into the submachine state machine 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 entry point pseudostate as defined in the submachine of the submachine state. As a result, the regions of the submachine state machine are entered at the corresponding entry point 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 state machine has reached the corresponding exit points, the submachine state exits at this exit point.

Notation

A connection point reference to an entry point has the same notation as an entry point pseudostate. The circle is placed on the border of the state symbol of a submachine state.

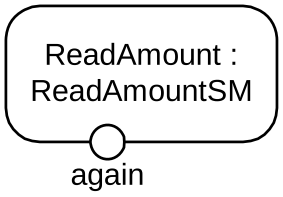


Figure 15.7 - Entry Point

A connection point reference to an exit point has the same notation as an exit point pseudostate. The encircled cross is placed on the border of the state symbol of a submachine state.

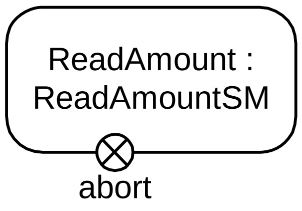


Figure 15.8 - Exit Point

Presentation Options

A connection point reference to an entry point can also be visualized using a rectangular symbol as shown in Figure 15.9. 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 transition-oriented control icon notation as defined in “Transition (from BehaviorStateMachines)” on page 572.

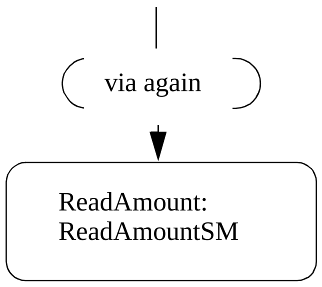


Figure 15.9 - Alternative Entry Point notation

A connection point reference to an exit point can also be visualized using a rectangular symbol as shown in Figure 15.10. 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 transition-oriented control icon notation as defined in "Transition (from BehaviorStateMachines)" on page 572.

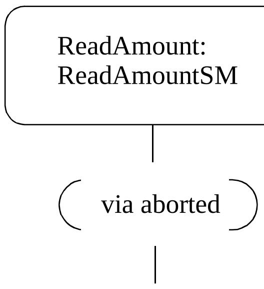


Figure 15.10 - Alternative Exit Point notation

15.3.2 FinalState (from BehaviorStateMachines)

Generalizations

- "State (from BehaviorStateMachines, ProtocolStateMachines)" on page 550

Description

A special kind of state signifying that the enclosing region is completed. If the enclosing region is directly contained in a state machine and all other regions in the state machine also are completed, then it means that the entire state machine is completed.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] A final state cannot have any outgoing transitions.
self.outgoing->size() = 0

[2] A final state cannot have regions.
self.region->size() =0

[3] A final state cannot reference a submachine.
self.submachine->isEmpty()

[4] A final state has no entry behavior.
self.entry->isEmpty()

[5] A final state has no exit behavior.
self.exit->isEmpty()

[6] A final state has no state (doActivity) behavior.

```
self.doActivty->isEmpty()
```

Semantics

When the final state is entered, its containing region is completed, which means that it satisfies the completion condition. The containing state for this region is considered completed when all contained regions are completed. If the region is contained in a state machine and all other regions in the state machine also are completed, the entire state machine terminates, implying the termination of the context object of the state machine.

Notation

A final state is shown as a circle surrounding a small solid filled circle (see Figure 15.11). The corresponding completion transition on the enclosing state has as notation an unlabeled transition.

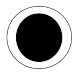


Figure 15.11 - Final State

Example

Figure 15.33 on page 559 has an example of a final state (the right most of the states within the composite state).

15.3.3 Interface (from ProtocolStateMachines)

Interface is defined as a specialization of the general Interface, adding an association to a protocol state machine.

Generalizations

- "Interface (from Communications)" on page 443 *(merge increment)*

Description

Since an interface specifies conformance characteristics, it does not own detailed behavior specifications. Instead, interfaces may own a protocol state machine that specifies event sequences and pre/post conditions for the operations and receptions described by the interface.

Attributes

No additional attributes

Associations

- protocol: ProtocolStateMachine [0..1]
  References a protocol state machine specifying the legal sequences of the invocation of the behavioral features described in the interface.

Semantics

Interfaces can specify behavioral constraints on the features using a protocol state machine. A classifier realizing an interface must comply with the protocol state machine owned by the interface.

Changes from previous UML

Interfaces can own a protocol state machine.

15.3.4 Port (from ProtocolStateMachines)

Generalizations

- "Port (from Ports)" on page 179 *(merge increment)*

Description

Port is defined as a specialization of the general Port, adding an association to a protocol state machine.

Attributes

No additional attributes

Associations

- protocol: ProtocolStateMachine [0..1]
  References an optional protocol state machine that describes valid interactions at this interaction point.

Semantics

The protocol references a protocol state machine (see "ProtocolStateMachine (from ProtocolStateMachines)" on page 535) that describes valid sequences of operation and reception invocations that may occur at this port.

15.3.5 ProtocolConformance (from ProtocolStateMachines)

Generalizations

- "DirectedRelationship (from Kernel)" on page 63

Description

Protocol state machines can be redefined into more specific protocol state machines, or into behavioral state machines. Protocol conformance declares that the specific protocol state machine specifies a protocol that conforms to the general state machine one, or that the specific behavioral state machine abides by the protocol of the general protocol state machine.

A protocol state machine is owned by a classifier. The classifiers owning a general state machine and an associated specific state machine are generally also connected by a generalization or a realization link.

Attributes

No additional attributes

Associations

- specificMachine: ProtocolStateMachine [1]
  Specifies the state machine that conforms to the general state machine.{Subsets DirectedRelationship::source and Element::owner}

- generalMachine: ProtocolStateMachine [1]
  Specifies the protocol state machine to which the specific state machine conforms. (Subsets DirectedRelationship::target}

Constraints

No additional constraints

Semantics

Protocol conformance means that every rule and constraint specified for the general protocol state machine (state invariants, pre- and post-conditions for the operations referred by the protocol state machine) apply to the specific protocol or behavioral state machine.

In most cases there are relationships between the classifier being the context of the specific state machine and the classifier being the context of the general protocol state machine. Generally, the former specializes or realizes the latter. It is also possible that the specific state machine is a behavioral state machine that implements the general protocol state machine, both state machines having the same class as a context.

15.3.6 ProtocolStateMachine (from ProtocolStateMachines)

Generalizations

- “StateMachine (from BehaviorStateMachines)” on page 564

Description

A *protocol state machine* is always defined in the context of a classifier. It specifies which operations of the classifier can be called in which state and under which condition, thus specifying the allowed call sequences on the classifier’s operations. A protocol state machine presents the possible and permitted transitions on the instances of its context classifier, together with the operations that carry the transitions. In this manner, an instance lifecycle can be created for a classifier, by specifying the order in which the operations can be activated and the states through which an instance progresses during its existence.

Attributes

No additional attributes

Associations

- conformance: ProtocolConformance[*]
  Conformance between protocol state machines. {Subsets *Element::ownedElement*}

Constraints

[1] A protocol state machine must only have a classifier context, not a behavioral feature context.

(**not** context->isEmpty( )) **and** specification->isEmpty()

[2] All transitions of a protocol state machine must be protocol transitions. (transitions as extended by the ProtocolStateMachines package).

region->forAll(r | r.transition->forAll(t | t.oclIsTypeOf(ProtocolTransition)))

[3] The states of a protocol state machine cannot have entry, exit, or do activity actions.

```
region->forAll(r | r.subvertex->forAll(v | v.oclIsKindOf(State) implies
  (v.entry->isEmpty() and v.exit->isEmpty() and v.doActivity->isEmpty())))
```

[4] Protocol state machines cannot have deep or shallow history pseudostates.

```
region->forAll (r | r.subvertex->forAll (v | v.oclIsKindOf(Psuedostate) implies
  ((v.kind <> #deepHistory) and (v.kind <> #shallowHistory)))))
```

[5] If two ports are connected, then the protocol state machine of the required interface (if defined) must be conformant to the protocol state machine of the provided interface (if defined).

Semantics

Protocol state machines help define the usage mode of the operations and receptions of a classifier by specifying:

- In which context (under which states and pre conditions) they can be used.
- If there is a protocol order between them.
- What result is expected from their use.

The states of a protocol state machine (protocol states) present an external view of the class that is exposed to its clients. Depending on the context, protocol states can correspond to the internal states of the instances as expressed by behavioral state machines, or they can be different.

A protocol state machine expresses parts of the constraints that can be formulated for pre- and post-conditions on operations. The translation from protocol state machine to pre- and post-conditions on operations might not be straightforward, because the conditions would need to account for the operation call history on the instance, which may or may not be directly represented by its internal states. A protocol state machine provides a direct model of the state of interaction with the instance, so that constraints on interaction are more easily expressed.

The protocol state machine defines all allowed transitions for each operation. The protocol state machine must represent all operations that can generate a given change of state for a class. Those operations that do not generate a transition are not represented in the protocol state machine.

Protocol state machines constitute a means to formalize the interface of classes, and do not express anything except consistency rules for the implementation or dynamics of classes.

Protocol state machine interpretation can vary from:

1. Declarative protocol state machines that specify the legal transitions for each operation. The exact triggering condition for the operations is not specified. This specification only defines the contract for the user of the context classifier.

2. Executable protocol state machines, that specify all events that an object may receive and handle, together with the transitions that are implied. In this case, the legal transitions for operations will exactly be the triggered transitions. The call trigger specifies the effect action, which is the call of the associated operation.

The representation for both interpretations is the same, the only difference being the direct dynamic implication that the interpretation 2 provides.

Elaborated forms of state machine modeling such as compound transitions, sub-state machines, composite states, and concurrent regions can also be used for protocol state machines. For example, concurrent regions make it possible to express protocol where an instance can have several active states simultaneously. Sub state machines and compound transitions are used as in behavioral state machines for factorizing complex protocol state machines.

A classifier may have several protocol state machines. This happens frequently, for example, when a class inherits several parent classes having protocol state machine, when the protocols are orthogonal. An alternative to multiple protocol state machines can always be found by having one protocol state machine, with sub state machines in concurrent regions.

Notation

The notation for protocol state machine is very similar to the one of behavioral state machines. The keyword {protocol} placed close to the name of the state machine differentiates graphically protocol state machine diagrams.

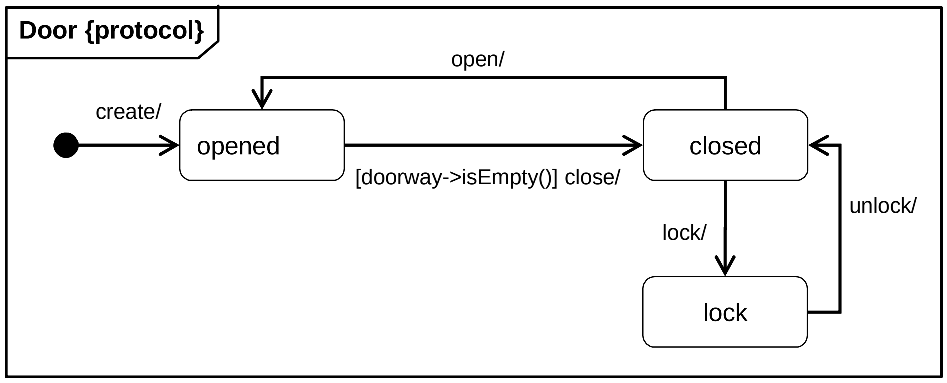


Figure 15.12 - Protocol state machine

15.3.7 ProtocolTransition (from ProtocolStateMachines)

Generalizations

- “Transition (from BehaviorStateMachines)” on page 572

Description

A protocol transition (transition as specialized in the ProtocolStateMachines package) specifies a legal transition for an operation. Transitions of protocol state machines have the following information: a pre-condition (guard), on trigger, and a post-condition. Every protocol transition is associated to zero or one operation (referred BehavioralFeature) that belongs to the context classifier of the protocol state machine.

The protocol transition specifies that the associated (referred) operation can be called for an instance in the origin state under the initial condition (guard), and that at the end of the transition, the destination state will be reached under the final condition (post).

Attributes

No additional attributes

Associations

- /referred: Operation[0..*]
  This association refers to the associated operation. It is derived from the operation of the call trigger when applicable.

- postCondition: Constraint[0..1]
  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. {Subsets *Element::ownedElement* and subsets *Namespace::ownedRule*}
- preCondition: Constraint[0..1]
  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. {Subsets *Transition::guard* and subsets *Namespace::ownedRule*}

Constraints

[1] A protocol transition always belongs to a protocol state machine.

```
container.belongsToPSM()
```

[2] A protocol transition never has associated actions.

```
effect->isEmpty()
```

[3] If a protocol transition refers to an operation (i.e., has a call trigger corresponding to an operation), then that operation should apply to the context classifier of the state machine of the protocol transition.

Additional Operations

[1] The operation belongsToPSM () checks if the region belongs to a protocol state machine.

```
context Region::belongsToPSM () : Boolean
result = if not stateMachine->isEmpty() then
            oclIsTypeOf(ProtocolStateMachine)
        else if not state->isEmpty() then
            state.container.belongsToPSM ()
        else false
```

Semantics

No "effect" action

The effect action is never specified. It is implicit, when the transition has a call trigger: the effect action will be the operation specified by the call trigger. It is unspecified in the other cases, where the transition only defines that a given event can be received under a specific state and pre-condition, and that a transition will lead to another state under a specific post-condition, whatever action will be made through this transition.

Unexpected event reception

The interpretation of the reception of an event in an unexpected situation (current state, state invariant, and pre-condition) is a semantic variation point: the event 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 final state or final state invariant, or post-condition) is also a semantic variation point. However, this should be interpreted as an error of the implementation of the protocol state machine.

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 15.13 can be interpreted in the following way:

1. The operation “m1” can be called on an instance when it is in the protocol state “S1” under the condition “C1.”
2. When “m1” is called in the protocol state “S1” under the condition “C1,” then the protocol state “S2” must be reached under the condition “C2.”

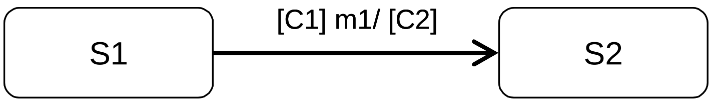


Figure 15.13 - Example of a protocol transition associated to the "m1" operation

Operations referred by several transitions

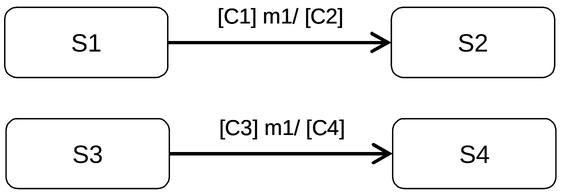


Figure 15.14 - Example of several transitions referring to the same operation

In a protocol state machine, several transitions can refer to the same operation as illustrated in Figure 15.14. In that case, all pre-and post-conditions will be combined in the operation pre-condition as shown below.

```
Operation m1()
Pre: S1 is in the configuration state and C1
        or
        S3 is in the configuration state and C3

Post:     if the initial condition was “S1 is in the configuration state and C1”
               then S2 is in the configuration state and C2
         else
               if the initial condition was “S3 is in the configuration state and C3”
               then S4 is in the configuration state and C4
```

A protocol state machine specifies all the legal transitions for each operation referred by its transitions. This means that for any operation referred by a protocol state machine, the part of its pre-condition relative to legal initial or final state is completely specified by the protocol state machine.

Unreferred Operations

If an operation is not referred by any transition of a protocol state machine, then the operation can be called for any state of the protocol state machine, and does not change the current state.

Using events in protocol state machines

Apart from the operation call event, events are generally used for expressing a dynamic behavior interpretation of protocol state machines. An event that is not a call event can be specified on protocol transitions.

In this case, this specification is a requirement to the environment external to the state machine: it is legal to send this event to an instance of the context classifier only under the conditions specified by the protocol state machine.

Just like call event, this can also be interpreted in a dynamic way, as a semantic variation point.

Notation

The usual state machine notation applies. The difference is that no actions are specified for protocol transitions, and that post-conditions can exist. Post-conditions have the same syntax as guard conditions, but appear at the end of the transition syntax.

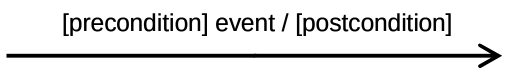


Figure 15.15 - Protocol transition notation

15.3.8 Pseudostate (from BehaviorStateMachines)

A pseudostate is an abstraction that encompasses different types of transient vertices in the state machine graph.

Generalizations

- “Vertex (from BehaviorStateMachines)” on page 581

Description

Pseudostates are typically used to connect multiple transitions into more complex state transitions paths. For example, by combining a transition entering a fork pseudostate with a set of transitions exiting the fork pseudostate, we get a compound transition that leads to a set of orthogonal target states.

Attributes

- kind: PseudostateKind
  Determines the precise type of the Pseudostate. Default value is *initial*.

Associations

- stateMachine : Statemachine [0..1]
  The StateMachine in which this Pseudostate is defined. This only applies to Pseudostates of the kind entryPoint or exitPoint. {Subsets *NamedElement::namespace*}
- state : State [0..1]
  State that owns the Pseudostate. {Subsets *Element::owner*}

Constraints

[1] An initial vertex can have at most one outgoing transition.

```
(self.kind = #initial) implies
    (self.outgoing->size <= 1)
```

[2] History vertices can have at most one outgoing transition.

```
((self.kind = #deepHistory) or (self.kind = #shallowHistory)) implies
    (self.outgoing->size <= 1)
```

[3] In a complete statemachine, a join vertex must have at least two incoming transitions and exactly one outgoing transition.

```
(self.kind = #join) implies
    ((self.outgoing->size = 1) and (self.incoming->size >= 2))
```

[4] All transitions incoming a join vertex must originate in different regions of an orthogonal state.

```
(self.kind = #join)
    implies
    self.incoming->forAll (t1, t2 | t1<>t2 implies
        (self.stateMachine.LCA(t1.source, t2.source).container.isOrthogonal))
```

[5] In a complete statemachine, a fork vertex must have at least two outgoing transitions and exactly one incoming transition.

```
(self.kind = #fork) implies
    ((self.incoming->size = 1) and (self.outgoing->size >= 2))
```

[6] All transitions outgoing a fork vertex must target states in different regions of an orthogonal state.

```
(self.kind = #fork)
     implies
    self.outgoing->forAll (t1, t2 | t1<>t2 implies
        (self.stateMachine.LCA(t1.target, t2.target).
            container.isOrthogonal))
```

[7] In a complete statemachine, a junction vertex must have at least one incoming and one outgoing transition.

```
(self.kind = #junction) implies
    ((self.incoming->size >= 1) and (self.outgoing->size >= 1))
```

[8] In a complete statemachine, a choice vertex must have at least one incoming and one outgoing transition.

```
(self.kind = #choice) implies
    ((self.incoming->size >= 1) and (self.outgoing->size >= 1))
```

[9] The outgoing transition from an initial vertex may have a behavior, but not a trigger or guard.

```
(self.kind = PseudostateKind::initial) implies (self.outgoing.guard->isEmpty() and self.outgoing.trigger->isEmpty())
```

Semantics

The specific semantics of a Pseudostate depends on the setting of its kind attribute.

- An *initial* pseudostate represents a default vertex that is the source for a single transition to the *default* state of a composite state. There can be at most one initial vertex in a region. The outgoing transition from the initial vertex may have a behavior, but not a trigger or guard.

- *deepHistory* represents the most recent active configuration of the composite state that directly contains this pseudostate (e.g., the state configuration that was active when the composite state was last exited). A composite state can have at most one deep history vertex. At most one transition may originate from the history connector to the *default* deep history state. This transition is taken in case the composite state had never been active before. Entry actions of states entered on the path to the state represented by a deep history are performed.
- *shallowHistory* represents the most recent active substate of its containing state (but *not* the substates of that substate). A composite state can have at most one shallow history vertex. A transition coming into the shallow history vertex is equivalent to a transition coming into the most recent active substate of a state. At most one transition may originate from the history connector to the *default* shallow history state. This transition is taken in case the composite state had never been active before. Entry actions of states entered on the path to the state represented by a shallow history are performed.
- *join* vertices serve to merge several transitions emanating from source vertices in different orthogonal regions. The transitions entering a join vertex cannot have guards or triggers.
- *fork* vertices serve to split an incoming transition into two or more transitions terminating on orthogonal target vertices (i.e., vertices in different regions of a composite state). The segments outgoing from a fork vertex must not have guards or triggers.
- *junction* vertices are semantic-free vertices that are used to chain together multiple transitions. They are used to construct compound transition paths between states. For example, a junction can be used to converge multiple incoming transitions into a single outgoing transition representing a shared transition path (this is known as a *merge*). Conversely, they can be used to split an incoming transition into multiple outgoing transition segments with different guard conditions. This realizes a *static conditional branch*. (In the latter case, outgoing transitions whose guard conditions evaluate to false are disabled. A predefined guard denoted “else” may be defined for at most one outgoing transition. This transition is enabled if all the guards labeling the other transitions are false.) Static conditional branches are distinct from dynamic conditional branches that are realized by choice vertices (described below).
- *choice* vertices which, when reached, result in the dynamic evaluation of the guards of the triggers of its outgoing transitions. This realizes a *dynamic conditional branch*. It allows splitting of transitions into multiple outgoing paths such that the decision on which path to take may be a function of the results of prior actions performed in the same run-to-completion step. If more than one of the guards evaluates to true, an arbitrary one is selected. 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 vertex.) Choice vertices should be distinguished from static branch points that are based on junction points (described above).
- An *entry point* pseudostate is an entry point of a state machine or composite state. In each region of the state machine or composite state it has a single transition to a vertex within the same region.
- An *exit point* pseudostate is an exit point of a state machine or composite state. Entering an exit point within any region of the composite state or state machine referenced by a submachine state implies the exit of this composite state or submachine state and the triggering of the transition that has this exit point as source in the state machine enclosing the submachine or composite state.
- Entering a *terminate* pseudostate implies that the execution of this state machine by means of its context object is terminated. The state machine does not exit any states nor does it perform any exit actions other than those associated with the transition leading to the terminate pseudostate. Entering a terminate pseudostate is equivalent to invoking a DestroyObjectAction.

Notation

An initial pseudostate is shown as a small solid filled circle (see Figure 15.16). In a region of a classifierBehavior state machine, the transition from an initial pseudostate may be labeled with the trigger event that creates the object; otherwise, it must be unlabeled. If it is unlabeled, it represents any transition from the enclosing state.

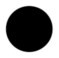


Figure 15.16 - Initial Pseudostate

A shallowHistory is indicated by a small circle containing an 'H' (see Figure 15.17). It applies to the state region that directly encloses it.

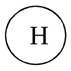


Figure 15.17 - Shallow History

A deepHistory is indicated by a small circle containing an 'H*' (see Figure 15.18). It applies to the state region that directly encloses it.

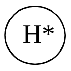


Figure 15.18 - Deep History

An entry point is shown as a small circle on the border of the state machine diagram or composite state, with the name associated with it (see Figure 15.19).

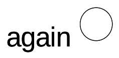


Figure 15.19 - Entry point

Optionally it may be placed both within the state machine diagram and outside the border of the state machine diagram or composite state.

An exit point is shown as a small circle with a cross on the border of the state machine diagram or composite state, with the name associated with it (see Figure 15.20).

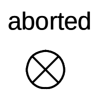


Figure 15.20 - Exit point

Optionally it may be placed both within the state machine diagram or composite state and outside the border of the state machine diagram or composite state.

Figure 15.21 illustrates the notation for depicting entry and exit points to composite states (the case of submachine states is illustrated in the corresponding Notation sub clause of “State (from BehaviorStateMachines, ProtocolStateMachines)” on page 550).

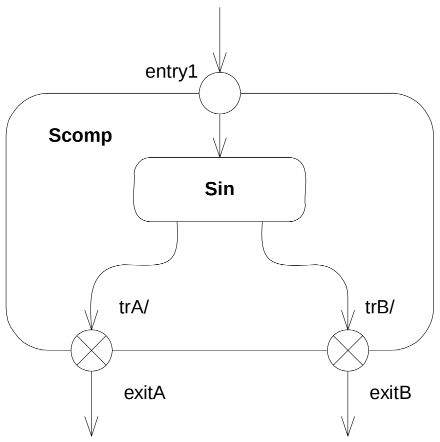


Figure 15.21 - Entry and exit points on composite states

Alternatively, the “bracket” notation shown in Figure 15.9 and Figure 15.10 on page 532 can also be used for the transition-oriented notation.

A junction is represented by a small black circle (see Figure 15.22).

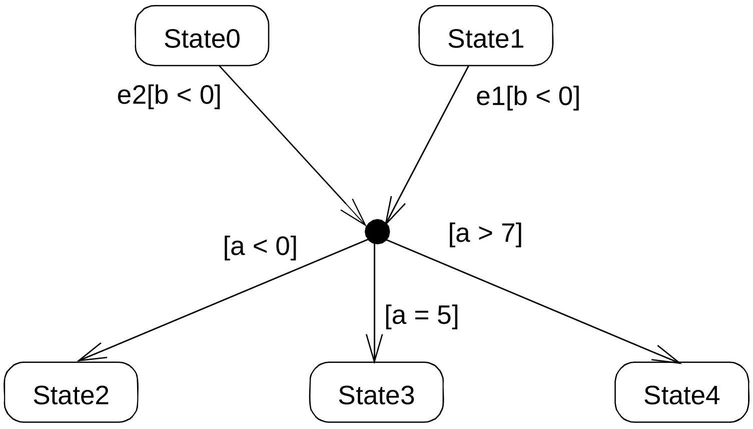


Figure 15.22 - Junction

A choice pseudostate is shown as a diamond-shaped symbol as exemplified by Figure 15.23.

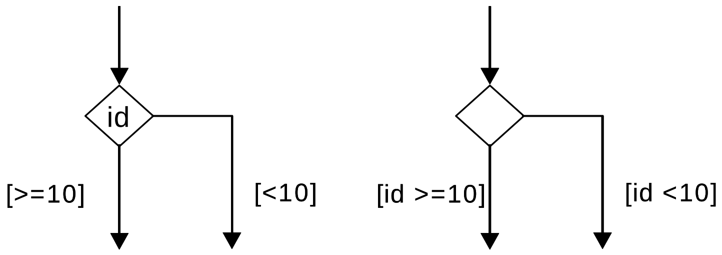


Figure 15.23 - Choice Pseudostate

A terminate pseudostate is shown as a cross, see Figure 15.24.

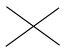


Figure 15.24 - Terminate node

The notation for a fork and join is a short heavy bar (Figure 15.25). 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.

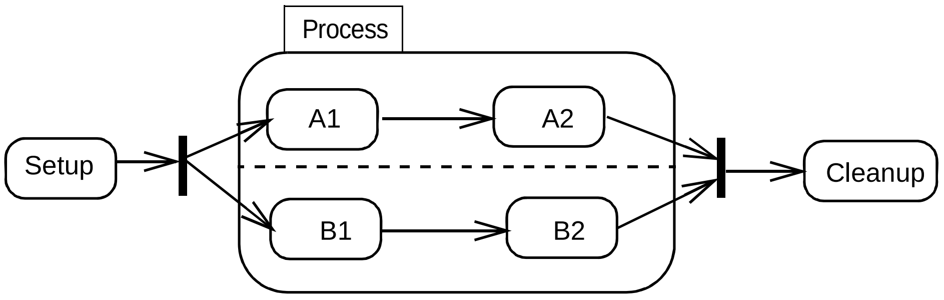


Figure 15.25 - Fork and Join

Presentation Options

If 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 exemplified in Figure 15.26.

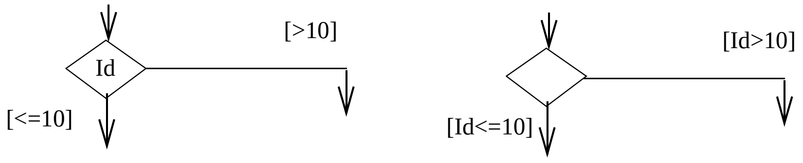


Figure 15.26 - Alternative Notation for Choice Pseudostate

Multiple trigger-free and effect-free transitions originating on a set of states and targeting a junction vertex with a single outgoing transition may be presented as a state symbol with a list of the state names and an outgoing transition symbol corresponding to the outgoing transition from the junction.

The special case of the transition from the junction having a history as target may optionally be presented as the target being the state list state symbol. See Figure 15.27 and Figure 15.28 for examples.

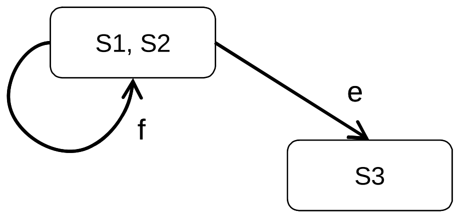


Figure 15.27 - State List Option

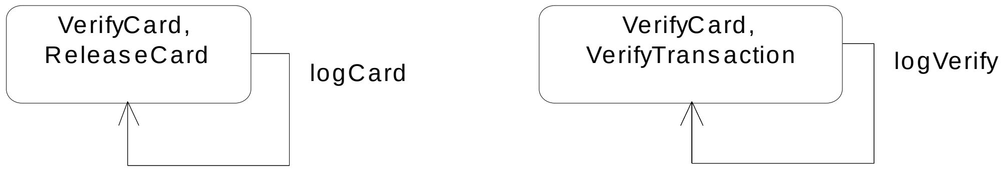


Figure 15.28 - State Lists

Changes from previous UML

- Entry and exit point and terminate Pseudostates have been introduced.
- The semantics of deepHistory has been aligned with shallowHistory in that the containing state does not have to be exited in order for deepHistory to be defined. The implication of this is that deepHistory (as is the case for shallowHistory) can be the target of transitions also within the containing state and not only from states outside.
- The state list presentation option is an extension to UML1.x.

15.3.9 PseudostateKind (from BehaviorStateMachines)

PseudostateKind is an enumeration type.

Generalizations

None

Description

PseudostateKind is an enumeration of the following literal values:

- *initial*
- *deepHistory*
- *shallowHistory*
- *join*
- *fork*

- *junction*
- *choice*
- *entryPoint*
- *exitPoint*
- *terminate*

Attributes

No additional attributes

Associations

No additional associations

Changes from previous UML

EntryPoint, exitPoint, and terminate have been added.

15.3.10 Region (from BehaviorStateMachines)

Generalizations

- "Namespace (from Kernel)" on page 99
- "RedefinableElement (from Kernel)" on page 130

Description

A region is an orthogonal part of either a composite state or a state machine. It contains states and transitions.

Attributes

No additional attributes

Associations

- statemachine: StateMachine[0..1]
  The StateMachine that owns the Region. If a Region is owned by a StateMachine, then it cannot also be owned by a State. {Subsets *NamedElement::namespace*}
- state: State[0..1]
  The State that owns the Region. If a Region is owned by a State, then it cannot also be owned by a StateMachine. {Subsets *NamedElement::namespace*}
- transition:Transition[*]
  The set of transitions owned by the region. Note that internal transitions are owned by a region, but applies to the source state. {Subsets *Namespace::ownedMember*}
- subvertex: Vertex[*]
  The set of vertices that are owned by this region. {Subsets *Namespace::ownedMember*}
- extendedRegion: Region[0..1]
  The region of which this region is an extension. {Subsets *RedefinableElement::redefinedElement*}

- /redefinitionContext: Classifier[1]
  References the classifier in which context this element may be redefined. {Redefines *RedefinableElement::redefinitionContext*}

Constraints

[1] A region can have at most one initial vertex.

```
self.subvertex->select (v | v.oclIsKindOf(Pseudostate))->
    select(p : Pseudostate | p.kind = #initial)->size() <= 1
```

[2] A region can have at most one deep history vertex.

```
self.subvertex->select (v | v.oclIsKindOf(Pseudostate))->
    select(p : Pseudostate | p.kind = #deepHistory)->size() <= 1
```

[3] A region can have at most one shallow history vertex.

```
self.subvertex->select(v | v.oclIsKindOf(Pseudostate))->
    select(p : Pseudostate | p.kind = #shallowHistory)->size() <= 1
```

[4] If a Region is owned by a StateMachine, then it cannot also be owned by a State and vice versa.

```
(stateMachine->notEmpty() implies state->isEmpty()) and (state->notEmpty() implies stateMachine->isEmpty())
```

[5] The redefinition context of a region is the nearest containing statemachine.

```
redefinitionContext =
    let sm = containingStateMachine() in
    if sm.context->isEmpty() or sm.general->notEmpty() then
        sm
    else
        sm.context
    endif
```

Additional constraints

[1] 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/State of a redefining region must redefine the containing StateMachine/State of the redefined region.

[2] 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., it adds vertices and transitions and it redefines states and transitions of the redefined region).

Additional operations

[1] The operation containingStatemachine() returns the StateMachine in which this Region is defined.

```
context Region::containingStatemachine() : StateMachine
  post: result = if stateMachine->isEmpty() then
     state.containingStateMachine()
   else
     stateMachine
```

Semantics

The semantics of regions is tightly coupled with states or state machines having regions, and it is therefore defined as part of the semantics for state and state machine.

When a composite state or state machine is extended, each inherited region may be extended, and regions may be added.

Notation

A composite state or state machine with regions is shown by tiling the graph region of the state/state machine using dashed lines to divide it into regions. 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.

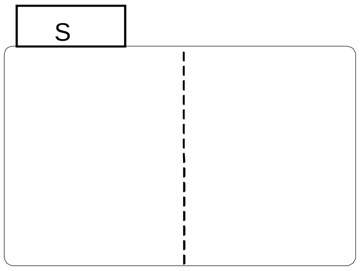


Figure 15.29 - Notation for composite state/state machine with regions

A composite state or state machine with just one region is shown by showing a nested state diagram within the graph region.

In order to indicate that an inherited region is extended, the keyword «extended» is associated with the name of the region.

15.3.11 State (from BehaviorStateMachines, ProtocolStateMachines)

A state models a situation during which some (usually implicit) invariant condition holds.

Generalizations

- “Namespace (from Kernel)” on page 99
- “RedefinableElement (from Kernel)” on page 130
- “Vertex (from BehaviorStateMachines)” on page 581

Description

State in Behavioral State machines

A state models a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some behavior (i.e., the model element under consideration enters the state when the behavior commences and leaves it as soon as the behavior is completed).

The following kinds of states are distinguished:

- Simple state,
- composite state, and
- submachine state.

A composite state is either a simple composite state (with just one region) or an orthogonal state (with more than one region).

Simple state

A simple state is a state that does not have substates (i.e., it has no regions and it has no submachine state machine).

Composite state

A composite state either contains one region or is decomposed into two or more orthogonal *regions*. Each region has a set of mutually exclusive disjoint subvertices and a set of transitions. A given state may only be decomposed in one of these two ways. In Figure 15.35 on page 560, 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 by any other state; otherwise, it is referred to as an *indirect substate*.

Each region of a composite state may have an initial pseudostate and a final state. A transition to the enclosing state represents a transition to the initial pseudostate in each region. A newly-created object takes its topmost default transitions, originating from the topmost initial pseudostates of each region.

A transition to a final state represents the completion of behavior in the enclosing region. Completion of behavior in all orthogonal regions represents completion of behavior by the enclosing state and triggers a completion event on the enclosing state. Completion of the topmost regions of an object corresponds to its termination.

An entry pseudostate is used to join an external transition terminating on that entry point to an internal transition emanating from that entry point. An exit pseudostate is used to join an internal transition terminating on that exit point to an external transition emanating from that exit point. The main purpose of such entry and exit points is to execute the state entry and exit actions respectively in between the actions that are associated with the joined transitions.

Semantic variation point (default entry rule)

If a transition terminates on an enclosing state and the enclosed regions do not have an initial pseudostate, the interpretation of this situation is a semantic variation point. In some interpretations, this is considered an ill-formed model. That is, in those cases the initial pseudostate is mandatory.

An alternative interpretation allows this situation and it means that, when such a transition is taken, the state machine stays in the composite state, *without entering any of the regions or their substates*.

Submachine state

A submachine state specifies the insertion of the specification of a submachine state machine. The state machine that contains the submachine state is called the *containing* state machine. The same state machine may be a submachine more than once in the context of a single containing state machine.

A submachine state is semantically equivalent to a composite state. The regions of the submachine state machine are the regions of the composite state. The entry, exit, and behavior actions and internal transitions are defined as part of the state. Submachine state is a decomposition mechanism that allows factoring of common behaviors and their reuse.

Transitions in the containing state machine can have entry/exit points of the inserted state machine as targets/sources.

State in Protocol State machines

The states of protocol state machines 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 is not processing any operation, users of this instance can always know its state configuration.

Attributes

- /isComposite : Boolean [1]
  A state with isComposite=true is said to be a *composite state*. A composite state is a state that contains at least one region. Default value is *false*.
- /isOrthogonal: Boolean [1]
  A state with isOrthogonal=true is said to be an *orthogonal* composite *state*. An orthogonal composite state contains two or more regions. Default value is *false*.
- /isSimple: Boolean [1]
  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 state machine. Default value is *true*.
- /isSubmachineState: Boolean [1]
  A state with isSubmachineState=true is said to be a *submachine state*. Such a state refers to a state machine (submachine). Default value is *false*.

Associations

Package BehaviorStateMachines

- connection: ConnectionPointReference [0..*]
  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 submachinestate. {Subsets *Namespace::ownedMember*}
- connectionPoint: Pseudostate [0..*]
  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. {Subsets *Namespace::ownedMember*}
- deferrableTrigger: Trigger [0..*]
  A list of triggers that are candidates to be retained by the state machine if they trigger no transitions out of the state (not consumed). A deferred trigger is retained until the state machine reaches a state configuration where it is no longer deferred.
- doActivity: Behavior[0..1]
  An optional behavior that is executed while being in the state. The execution starts when this state is entered, and stops either by itself or when the state is exited whichever comes first. {Subsets *Element::ownedElement*}
- entry: Behavior[0..1]
  An optional behavior that is executed whenever this state is entered regardless of the transition taken to reach the state. If defined, entry actions are always executed to completion prior to any internal behavior or transitions performed within the state. {Subsets *Element::ownedElement*}
- exit: Behavior[0..1]
  An optional behavior that is executed whenever this state is exited regardless of which transition was taken out of the state. If defined, exit actions are always executed to completion only after all internal activities and transition actions have completed execution. {Subsets *Element::ownedElement*}

- redefinedState: State[0..1]
  The state of which this state is a redefinition. {Subsets *RedefinableElement::redefinedElement*}
- region: Region[0..*] {subsets *ownedMember*}
  The regions owned directly by the state.
- submachine: StateMachine[0..1]
  The state machine that is to be inserted in place of the (submachine) state.
- stateInvariant: Constraint [0..1]
  Specifies conditions that are always true when this state is the current state. In protocol state machines, state invariants are additional conditions to the preconditions of the outgoing transitions, and to the postcondition of the incoming transitions.
- /redefinitionContext: Classifier[1]
  References the classifier in which context this element may be redefined. {Redefines *RedefinableElement::redefinitionContext*}

Constraints

[1] Only submachine states can have connection point references.

```
isSubmachineState implies connection->notEmpty ( )
```

[2] 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 state machine.

```
self.isSubmachineState implies (self.connection->forAll (cp |
    cp.entry->forAll (p | p.statemachine = self.submachine) and cp.exit->forAll (p | p.statemachine = self.submachine)))
```

[3] A state is not allowed to have both a submachine and regions.

```
isComposite implies not isSubmachineState
```

[4] A simple state is a state without any regions.

```
isSimple = region.isEmpty()
```

[5] A composite state is a state with at least one region.

```
isComposite = region.notEmpty()
```

[6] An orthogonal state is a composite state with at least 2 regions.

```
isOrthogonal = (region->size () > 1)
```

[7] Only submachine states can have a reference statemachine.

```
isSubmachineState = submachine.notEmpty()
```

[8] The redefinition context of a state is the nearest containing statemachine.

```
redefinitionContext =
    let sm = containingStateMachine() in
    if sm.context->isEmpty() or sm.general->notEmpty() then
        sm
    else
        sm.context
    endif
```

[9] Only composite states can have entry or exit pseudostates defined.

```
connectionPoint->notEmpty() implies isComoposite
```

[10] Only entry or exit pseudostates can serve as connection points.

```
connectionPoint->forAll(cp|cp.kind = #entry or cp.kind = #exit)
```

Additional Operations

[1] 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. The containing region of a redefining state must redefine the containing region of the redefined state.

[2] 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 a region) 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" actions.

[3] The query containingStateMachine() returns the state machine that contains the state either directly or transitively.

```
context State::containingStateMachine() : StateMachine
    post: result =  container.containingStateMachine()
```

Semantics

States in general

The following applies to states in general. Special semantics applies to composite states and submachine states.

Active states

A state can be active or inactive during execution. A state becomes *active* when it is entered as a result of some transition, and becomes *inactive* if it is exited as a result of a transition. A state can be exited and entered as a result of the same transition (e.g., self transition).

State entry and exit

Whenever a state is entered, it executes its entry behavior *before* any other action is executed. Conversely, whenever a state is exited, it executes its exit behavior as the final step prior to leaving the state.

Behavior in state (do-activity)

The behavior represents the execution of a behavior, that occurs while the state machine is in the corresponding state. The behavior starts executing upon entering the state, following the entry behavior. If the behavior completes while the state is still active, it raises a completion event. In case where there is an outgoing completion transition (see below) the state will be exited. Upon exit, the behavior is terminated before the exit behavior is executed. If the state is exited as a result of the firing of an outgoing transition before the completion of the behavior, the behavior is aborted prior to its completion.

Deferred events

A state may specify a set of event types that may be *deferred* in that state. An event that does not trigger any transitions in the current state, will not be dispatched if its type matches one of the types in the deferred event set of that state. Instead, it remains in the event pool while another non-deferred event is dispatched instead. This situation persists until a state is reached where either the event is no longer deferred or where the event triggers a transition.

State redefinition

A state may be redefined. A simple state can be redefined (extended) to become a composite state (by adding a region) and a composite state can be redefined (extended) by adding regions and by adding vertices, states, entry/exit/do activities (if the general state has none), and transitions to inherited regions. The redefinition of a state applies to the whole state machine. For example, if a state list as part of the extended state machine includes a state that is redefined, then the state list for the extension state machine includes the redefined state.

Composite state

Active state configurations

In a hierarchical state machine more than one state can be active at the same time. If the state machine is in a simple state that is contained in a composite state, then all the composite states that either directly or transitively contain the simple state are also active. Furthermore, since the state machine as a whole and some of the composite states in this hierarchy may be orthogonal (i.e., containing regions), the current active "state" is actually represented by a set of trees of states starting with the top-most states of the root regions down to the innermost active substate. We refer to such a state tree as a *state configuration*.

Except during transition execution, the following invariants always apply to state configurations:

- If a composite state is active and not orthogonal, at most one of its substates is active.
- If the composite state is active and orthogonal, all of its regions are active, with at most one substate in each region.

Entering a non-orthogonal composite state

Upon entering a composite state, the following cases are differentiated:

- *Default entry*: Graphically, this is indicated by an incoming transition that terminates on the outside edge of the composite state. In this case, the default entry rule is applied (see *Semantic variation point (default entry rule)*). If there is a guard on the trigger of the transition, it must be enabled (true). (A disabled initial transition is an ill-defined execution state and its handling is not defined.) The entry behavior of the composite state is executed before the behavior associated with the initial transition.
- *Explicit entry*: If the transition goes to a substate of the composite state, then that substate becomes active and its entry code is executed after the execution of the entry code of the composite state. This rule applies recursively if the transition terminates on a transitively nested substate.
- *Shallow history entry*: If the transition terminates on a shallow history pseudostate, the active substate becomes the most recently active substate prior to this entry, unless the most recently active substate is the final state or if this is the first entry into this state. In the latter two cases, the *default history state* is entered. This is the substate that is target of the transition originating from the history pseudostate. (If no such transition is specified, the situation is ill-defined and its handling is not defined.) If the active substate determined by history is a composite state, then it proceeds with its default entry.
- *Deep history entry*: The rule here is the same as for shallow history except that 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 entry point pseudostate, then the entry behavior is executed before the action associated with the internal transition emanating from the entry point.

Entering an orthogonal composite state

Whenever an orthogonal composite state is entered, each one of its orthogonal regions is also entered, either by default or explicitly. If the transition terminates on the edge of the composite state, then all the regions are entered using default entry. 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 non-orthogonal state

When exiting from a composite state, the active substate is exited recursively. This means that the exit activities are executed in sequence starting with the innermost active state in the current state configuration.

If, in a composite state, the exit occurs through an exit point pseudostate the exit behavior of the state is executed *after* the behavior associated with the transition incoming to the exit point.

Exiting an orthogonal state

When exiting from an orthogonal state, each of its regions is exited. After that, the exit activities of the state are executed.

Deferred events

Composite states introduce potential event deferral conflicts. Each of the substates may defer or consume an event, potentially conflicting with the composite state (e.g., a substate defers an event while the composite state consumes it, or vice versa). In case of a composite orthogonal state, substates of orthogonal regions may also introduce deferral conflicts. The conflict resolution follows the triggering priorities, where nested states override enclosing states. In case of a conflict between states in different orthogonal regions, a consumer state overrides a deferring state.

Submachine state

A submachine state is semantically equivalent to the composite state defined by the referenced state machine. Entering and leaving this composite state is, in contrast to an ordinary composite state, via entry and exit points.

A submachine composite state machine 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 with a junction pseudostate (fork in case the composite state is orthogonal): Entering via an entry point implies that the entry behavior of the composite state is executed, followed by the (partial) transition(s) from the entry point to the target state(s) within the composite state. As for default initial transitions, guards associated with the triggers of these entry point transitions must evaluate to true in order for the specification not to be ill formed.

Similarly, it can be exited as a result of reaching its final state, by a “group” transition that applies to all substates in the submachine state composite state, or via any of its exit points. Exiting via a final state or by a group transition has the same meaning as for ordinary composite states. An exit point is equivalent with a junction pseudostate (join in case the composite state is orthogonal): Exiting via an exit point implies that first behavior of the transition with the exit point as target is executed, followed by the exit behavior of the composite state.

Notation

States in general

A state is in general shown as a rectangle with rounded corners, with the state name shown inside the rectangle.

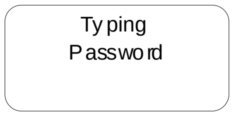


Figure 15.30 - State

Optionally, it may have an attached name tab, see Figure 15.31. 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. The state in Figure 15.25 on page 546 illustrates the use of the name tab.

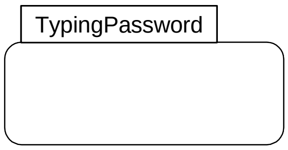


Figure 15.31 - State with name tab

A state may be subdivided into multiple compartments separated from each other by a horizontal line, see Figure 15.32.

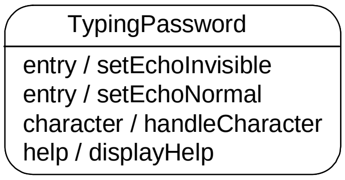


Figure 15.32 - State with compartments

The compartments of a state are:

- name compartment
- internal activities compartment
- internal transitions compartment

A composite state has in addition 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 (page 577) are used to show a transition oriented view of the state machine. 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 state machine is shown as a string following ':' after the name of the state.

- Internal activities compartment

  This compartment holds a list of internal actions or state (do) activities (behaviors) that are performed while the element is in the state.

The activity label identifies the circumstances under which the behavior specified by the activity expression will be invoked. The behavior expression may use any attributes and association ends that are in the scope of the owning entity. For list items where the expression is empty, the backslash separator is optional.

A number of labels are reserved for various special purposes and, therefore, cannot be used as event names. The following are the reserved activity labels and their meaning:

- *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 ("do activity") that is performed as long as the modeled element is in the state or until the computation specified by the expression is completed (the latter may result in a completion event being generated).

- Internal transition compartment

  This compartment contains a list of internal transitions, where each item has the form as described for Trigger.

  Each event name may appear more than once per state if the guard conditions are different. The event parameters and the guard conditions are optional. If the event has parameters, they can be used in the expression through the current event variable.

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 15.34). 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 state machine diagram. Instead, the contents of the composite state are shown in a separate diagram. Note that 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).

Examples

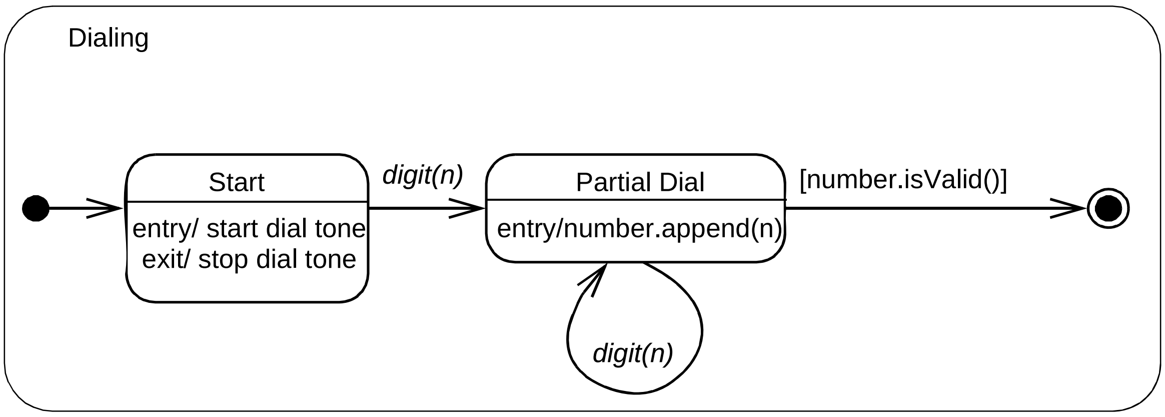


Figure 15.33 - Composite state with two states

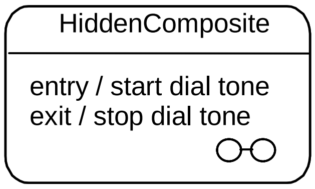


Figure 15.34 - Composite State with hidden decomposition indicator icon

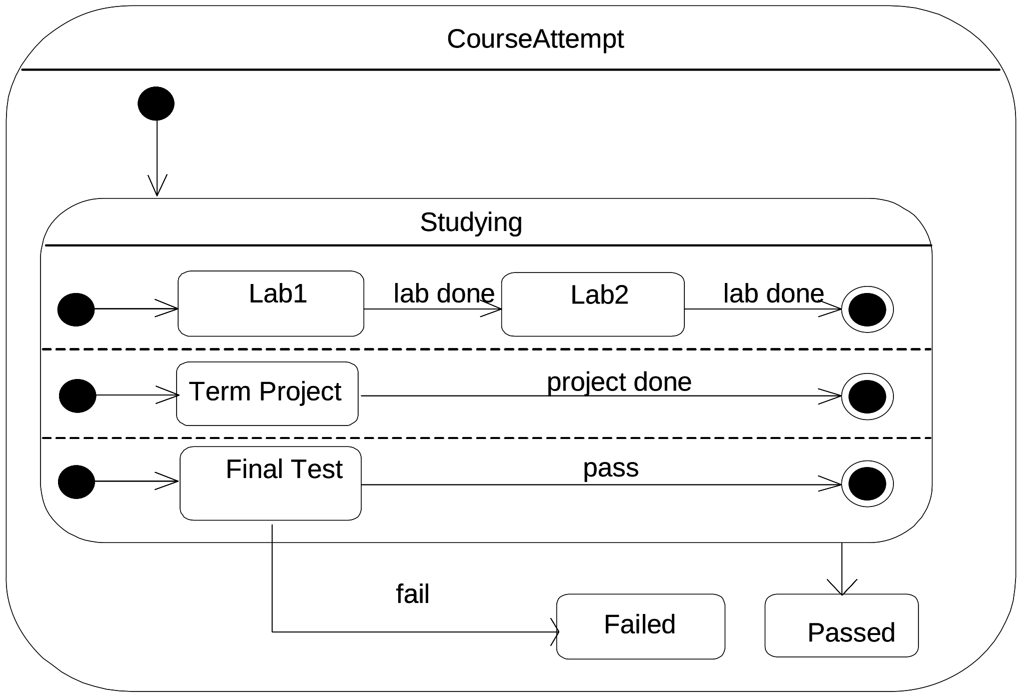


Figure 15.35 - Orthogonal state with regions

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 state machine>

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 are entry/exit point pseudostates on the border of the submachine state. The names are the names of the corresponding entry/exit points *defined* within the referenced state machine. See ("ConnectionPointReference (from BehaviorStateMachines)" on page 529).

If the substate machine 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 emanates 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.

Examples

The diagram in Figure 15.36 shows a fragment from a state machine diagram in which a submachine state (the FailureSubmachine) is referenced. The actual sub state machine is defined in some enclosing or imported name space.

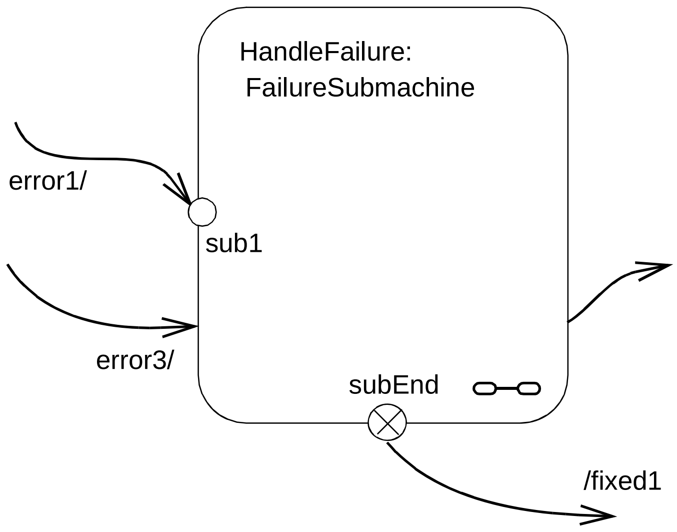


Figure 15.36 - Submachine State

In the above example, the transition triggered by event “error1” will terminate on entry point “sub1” of the FailureSubmachine state machine. The “error3” transition implies taking of the default transition of the FailureSubmachine.

The transition emanating from the “subEnd” exit point of the submachine will execute the “fixed1” behavior in addition to what is executed within the HandleFailure state machine. This transition must have been triggered within the HandleFailure state machine. Finally, the transition emanating from the edge of the submachine state is taken as a result of the completion event generated when the FailureSubmachine reaches its final state.

Note that the same notation would apply to composite states with the exception that there would be no reference to a state machine in the state name.

Figure 15.37 is an example of a state machine defined with two exit points. Entry and exit points may be shown on the frame or within the state graph. Figure 15.37 is an example of a state machine defined with an exit point shown within the state graph. Figure 15.38 shows the same state machine using a notation shown on the frame of the state machine.

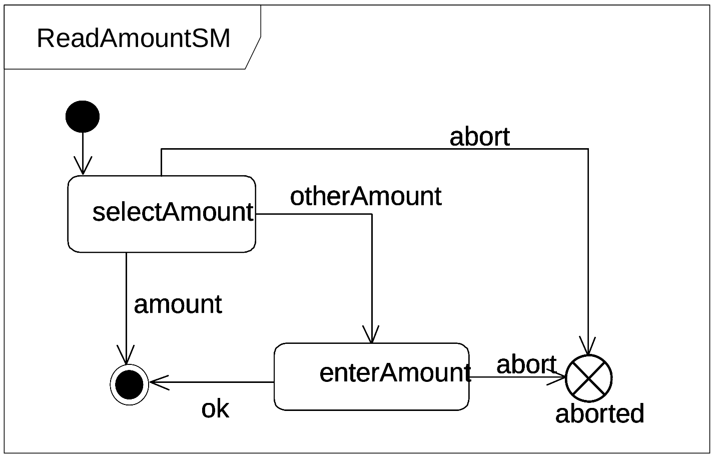


Figure 15.37 - State machine with exit point as part of the state graph

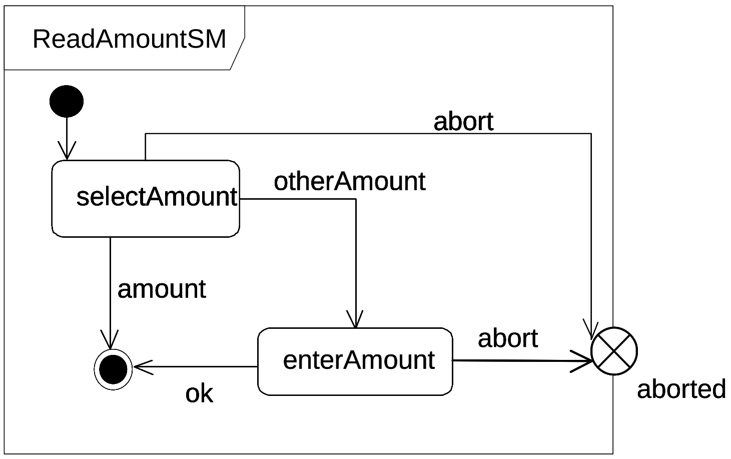


Figure 15.38 - State machine with exit point on the border of the statemachine

In Figure 15.39 the statemachine shown in Figure 15.38 on page 562 is referenced in a submachine state, and the presentation option with the exit points on the state symbol is shown.

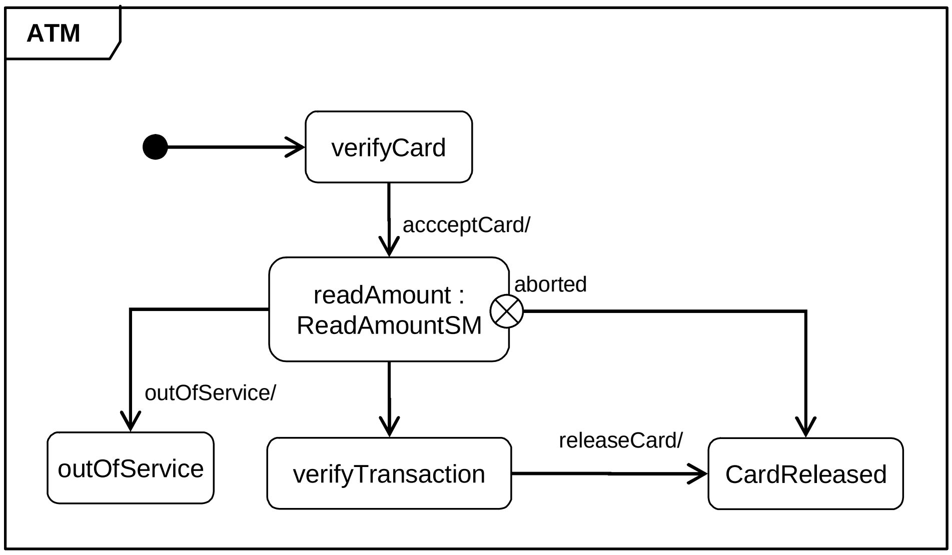


Figure 15.39 - SubmachineState with usage of exit point

An example of the notation for entry and exit points for composite states is shown in Figure 15.21 on page 544.

Notation for protocol state machines

The two differences that exist for state in protocol state machine, versus states in behavioral state machine, are as follows: Several features in behavioral state machine do not exist for protocol state machines (entry, exit, do); States in protocol state machines can have an invariant. The textual expression of the invariant will be represented by placing it after or under the name of the state, surrounded by square brackets.

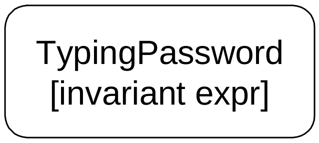


Figure 15.40 - State with invariant - notation

Rationale

Submachine states with usages of entry and exit points defined in the corresponding state machine have been introduced in order for state machines with submachines to scale and in order to provide encapsulation.

15.3.12 StateMachine (from BehaviorStateMachines)

State machines can be used to express the behavior of part of a system. Behavior is modeled as a traversal of a graph of state nodes interconnected by one or more joined transition arcs that are triggered by the dispatching of series of (event) occurrences. During this traversal, the state machine executes a series of activities associated with various elements of the state machine.

Generalizations

- "Behavior (from BasicBehaviors)" on page 430

Description

A state machine owns one or more regions, which in turn own vertices and transitions.

The behaviored classifier context owning a state machine defines which signal and call triggers are defined for the state machine, and which attributes and operations are available in activities of the state machine. Signal triggers and call triggers for the state machine are defined according to the receptions and operations of this classifier.

As a kind of behavior, a state machine may have an associated behavioral feature (specification) and be the method of this behavioral feature. In this case the state machine specifies the behavior of this behavioral feature. The parameters of the state machine in this case match the parameters of the behavioral feature and provide the means for accessing (within the state machine) the behavioral feature parameters.

A state machine without a context classifier may use triggers that are independent of receptions or operations of a classifier, i.e., either just signal triggers or call triggers based upon operation template parameters of the (parameterized) statemachine.

Attributes

No additional attributes

Associations

- region: Region[1..*] {subsets *ownedMember*}
  The regions owned directly by the state machine.
- connectionPoint: Pseudostate[*]
  The connection points defined for this state machine. They represent the interface of the state machine when used as part of submachine state.
- extendedStateMachine: StateMachine[*]
  The state machines of which this is an extension. {Subsets *RedefineableElement::redefinedElement*}

Constraints

[1] The classifier context of a state machine cannot be an interface.

context->notEmpty() **implies not** context.oclIsKindOf(Interface)

[2] The context classifier of the method state machine of a behavioral feature must be the classifier that owns the behavioral feature.

specification->notEmpty() **implies** (context->notEmpty() **and** specification->featuringClassifier->exists (c | c = context))

[3] The connection points of a state machine are pseudostates of kind entry point or exit point.

conectionPoint->forAll (c | c.kind = #entryPoint **or** c.kind = #exitPoint)

[4] A state machine as the method for a behavioral feature cannot have entry/exit connection points.

```
specification->notEmpty() implies connectionPoint->isEmpty()
```

Additional Operations

[1] The operation LCA(s1,s2) returns an orthogonal state or region that is the least common ancestor of states s1 and s2, based on the statemachine containment hierarchy.

[2] The query ancestor(s1, s2) checks whether s2 is an ancestor state of state s1.

```
context StateMachine::ancestor (s1 : State, s2 : State) : Boolean
result = if (s2 = s1) then
            true
      else if (s1.container->isEmpty) then
            true
      else if (s2.container->isEmpty) then
            false

      else (ancestor (s1, s2.container))
```

[3] The query isRedefinitionContextValid() specifies whether the redefinition contexts of a statemachine are properly related to the redefinition contexts of the specified statemachine to allow this element to redefine the other. The containing classifier of a redefining statemachine must redefine the containing classifier of the redefined statemachine.

[4] The query isConsistentWith() specifies that a redefining state machine is consistent with a redefined state machine provided that the redefining state machine is an extension of the redefined state machine: Regions are inherited and regions can be added, inherited regions can be redefined. In case of multiple redefining state machines, extension implies that the redefining state machine gets orthogonal regions for each of the redefined state machines.

Semantics

The event pool for the state machine is the event pool of the instance according to the behaviored context classifier, or the classifier owning the behavioral feature for which the state machine is a method.

Event processing - run-to-completion step

Event occurrences are detected, dispatched, and then processed by the state machine, one at a time. The order of dequeuing is not defined, leaving open the possibility of modeling different priority-based schemes.

The semantics of event occurrence processing is based on the *run-to-completion* assumption, interpreted as run-to-completion processing. Run-to-completion processing means that an event occurrence can only be taken from the pool and dispatched if the processing of the previous current occurrence is fully completed.

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 as a monitor.

The processing of a single event occurrence by a state machine is known as a *run-to-completion step*. Before commencing on a run-to-completion step, a state machine is in a stable state configuration with all entry/exit/internal activities (but not necessarily state (do) activities) completed. The same conditions apply after the run-to-completion step is completed. Thus, an event occurrence will never be processed while the state machine is in some intermediate and inconsistent situation. The *run-to-completion* step *is* the passage between two state configurations of the state machine.

The run-to-completion assumption simplifies the transition function of the state machine, since concurrency conflicts are avoided during the processing of event, allowing the state machine to safely complete its run-to-completion step.

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 event (type) is not in the deferred event list of the current state configuration, the event occurrence is discarded and the run-to-completion step is completed.

In the presence of orthogonal regions it is possible to fire multiple transitions as a result of the same event occurrence — as many as one transition in each region in the current state configuration. In case where one or more transitions are enabled, the state machine selects a subset and fires them. Which of the enabled transitions actually fire is determined by the transition selection algorithm described below. The order in which selected transitions fire is not defined.

Each orthogonal region in the active state configuration that is not decomposed into orthogonal regions (i.e., “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.

During a transition, a number of actions may be executed. If such an action is a synchronous operation invocation on an object executing a state machine, then the transition step is not completed until the invoked object completes its run-to-completion step.

Run-to-completion and concurrency

It is possible to define state machine semantics by allowing the run-to-completion steps to be applied orthogonally to the orthogonal regions of a composite state, rather than to the whole state machine. This would allow the event serialization constraint to be relaxed. However, such semantics are quite subtle and difficult to implement. Therefore, the dynamic semantics defined in this document are based on the premise that a single run-to-completion step applies to the entire state machine and includes the steps taken by orthogonal regions in the active state configuration.

In case of active objects, where each object has its own thread of execution, it is very important to clearly distinguish the notion of run to completion from the concept of thread pre-emption. Namely, run-to-completion event handling is performed by a thread that, in principle, *can* be pre-empted and its execution suspended in favor of another thread executing on the same processing node. (This is determined by the scheduling policy of the underlying thread environment — no assumptions are made about this policy.) When the suspended thread is assigned processor time again, it resumes its event processing from the point of pre-emption and, eventually, completes its event processing.

Conflicting transitions

It was already noted that it is possible for more than one transition to be enabled within a state machine. 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 only one transition will fire. In other words, in case of conflicting transitions, only one of them will fire in a single 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 transition conflicts, but not all of them. 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 joined transitions is based on the priority of the transition with the most transitively 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 transitively 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 easily 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.

StateMachine extension

A state machine is generalizable. A specialized state machine is an extension of the general state machine, in that regions, vertices, and transitions may be added; regions and states may be redefined (extended: simple states to composite states and composite states by adding states and transitions); and transitions can be redefined.

As part of a classifier generalization, the classifierBehavior state machine of the general classifier and the method state machines of behavioral features of the general classifier can be redefined (by other state machines). These state machines may be specializations (extensions) of the corresponding state machines of the general classifier or of its behavioral features.

A specialized state machine will have all the elements of the general state machine, and it may have 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 state machine may be redefined.

A simple state can be redefined (extended) to a composite state, by adding one or more regions.

A composite state can be redefined (extended) by either extending inherited regions or by adding regions as well as by adding entry and exit points. A region is extended by adding vertices, states, and transitions and by redefining states and transitions.

A submachine state may be redefined. The submachine state machine may be replaced by another submachine state machine, provided that it has the same entry/exit points as the redefined submachine state machine, but it may add entry/exit points.

Transitions can have their content and target state replaced, while the source state and trigger are preserved.

In case of multiple general classifiers, extension implies that the extension state machine gets orthogonal regions for each of the state machines of the general classifiers in addition to the one of the specific classifier.

Notation

A state machine diagram is a graph that represents a state machine. States and various other types of vertices (pseudostates) in the state machine graph are rendered by appropriate state and pseudostate symbols, while transitions are generally rendered by directed arcs that connect them or by control icons representing the actions of the behavior on the transition (page 577).

The association between a state machine and its context classifier or behavioral feature does not have a special notation.

A state machine that is an extension of the state machine in a general classifier will have the keyword «extended» associated with the name of the state machine.

The default notation for classifier is used for denoting state machines. The keyword is «statemachine».

Inherited states are drawn with dashed lines or gary-toned lines.

Presentation option

Inherited states are drawn with gray-toned lines.

Examples

Figure 15.41 is an example statemachine diagram for the state machine for simple telephone object. In addition to the initial state, the state machine has an entry point called activeEntry, and in addition to the final state, it has an exit point called aborted.

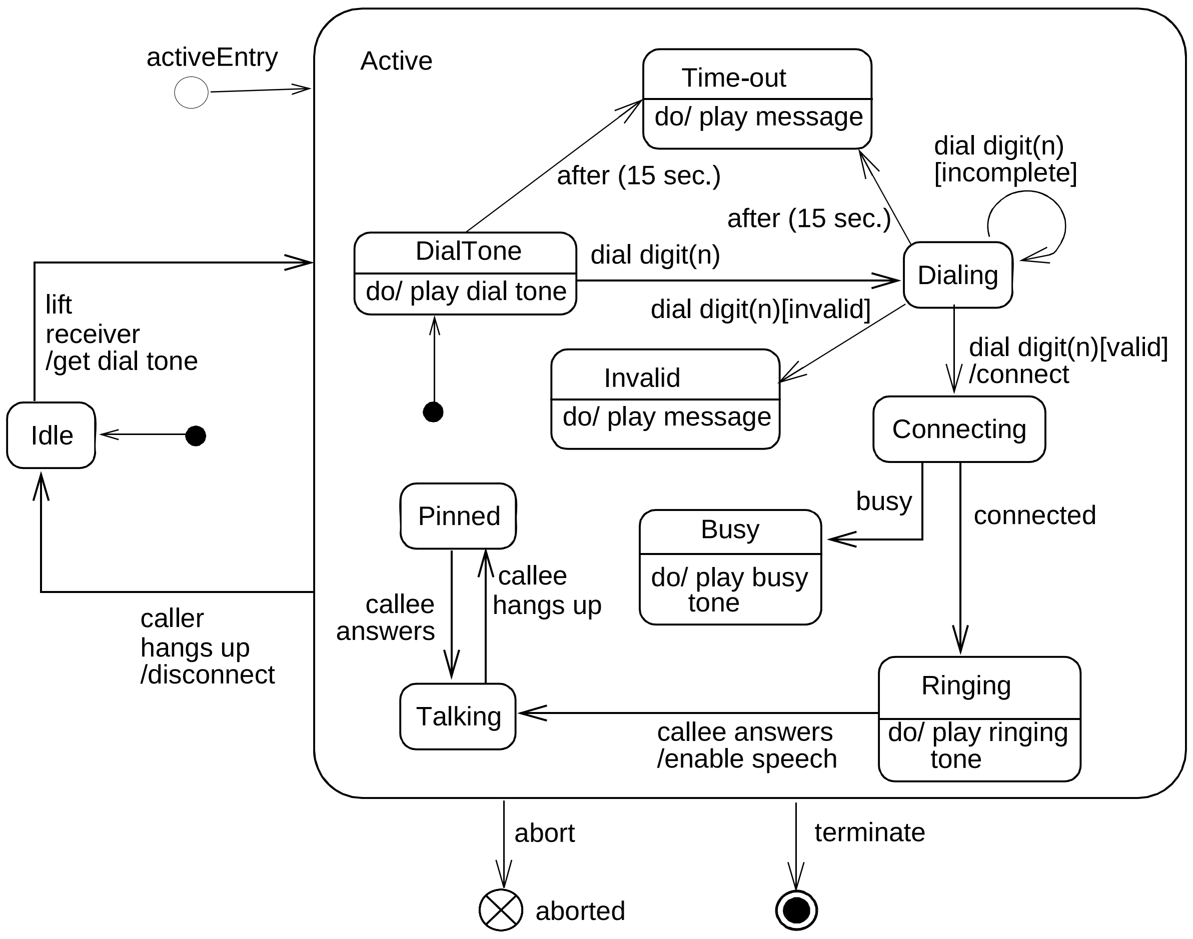


Figure 15.41 - State machine diagram representing a state machine

As an example of state machine specialization, the states VerifyCard, OutOfService, and VerifyTransaction in the ATM state machine in Figure 15.42 have been specified as {final}, which means that they cannot be redefined (i.e., extended) in specializations of ATM. The other states can be redefined. The (verifyTransaction, releaseCard) transition has also been specified as {final}, meaning that the effect behavior and the target state cannot be redefined.

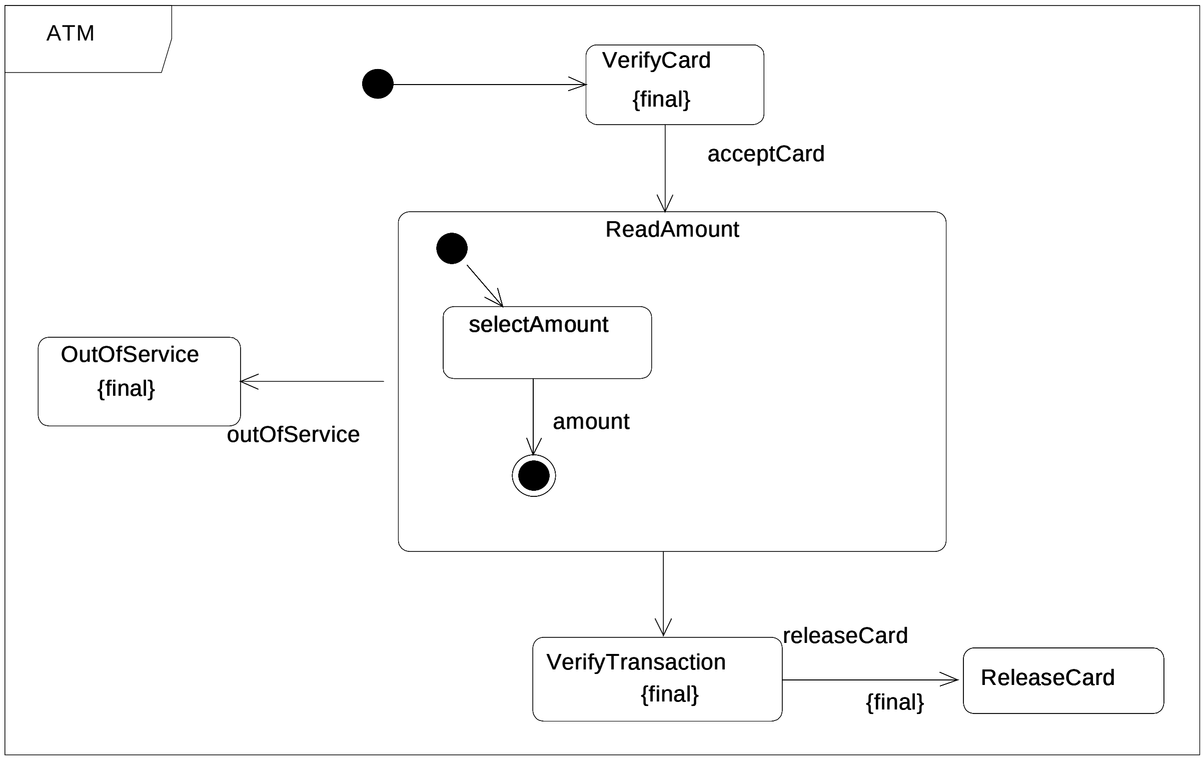


Figure 15.42 - A general state machine

In Figure 15.43 a specialized ATM (which is the statemachine of a class that is a specialization of the class with the ATM statemachine of Figure 15.42) 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.

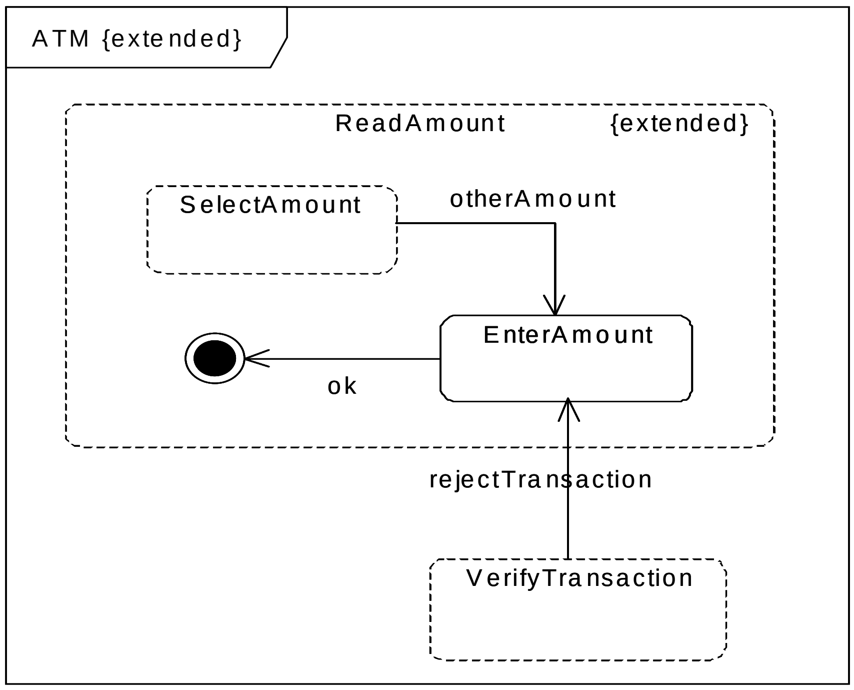


Figure 15.43 - An extended state machine

Rationale

The rationale for statemachine extension is that it shall be possible to define the redefined behavior of a special classifier as an extension of the behavior of the general classifier.

Changes from previous UML

State machine extension is an extension of 1.x. In 1.x, state machine generalization is only explained informally discussed as a non-normative practice.

Rationale

State machines are used for the definition of behavior (for example, classes that are generalizable). As part of the specialization of a class it is desirable also to specialize the behavior definition.

15.3.13 TimeEvent (from BehaviorStateMachines)

Generalizations

- "TimeEvent (from Communications, SimpleTime)" on page 453 *(merge increment)*

Description

Extends TimeEvent to be defined relative to entering the current state of the executing state machine.

Constraints

[1] The starting time for a relative time event may only be omitted for a time event that is the trigger of a state machine.

Semantics

If the deadline expression is relative and no explicit starting time is defined, then it is relative to the time of entry into the source state of the transition triggered by the event. In that case, the time event occurrence is generated only if the state machine is still in that state when the deadline expires.

Notation

If no starting point is indicated, then it is the time since the entry to the current state.

15.3.14 Transition (from BehaviorStateMachines)

Generalizations

- “Namespace (from Kernel)” on page 99
- “RedefinableElement (from Kernel)” on page 130

Description

A transition is a directed relationship between a source vertex and a target vertex. It may be part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to an occurrence of an event of a particular type.

Attributes

- kind: TransitionKind [1]
  See definition of TransitionKind. Default value is *external.*

Associations

- trigger: Trigger[0..*]
  Specifies the triggers that may fire the transition.
- guard: Constraint[0..1]
  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 state machine. 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. {Subsets *Namespace::ownedRule*}
- effect: Behavior[0..1]
  Specifies an optional behavior to be performed when the transition fires.
- source: Vertex[1]
  Designates the originating vertex (state or pseudostate) of the transition.
- target: Vertex[1]
  Designates the target vertex that is reached when the transition is taken.
- redefinedTransition: Transition[0..1]
  The transition of which this is a replacement. {Subsets *RedefinableElement::redefinedElement*}

- /redefinitionContext: Classifier[1]
  References the classifier in which context this element may be redefined. {Redefines *RedefinableElement::redefinitionContext*}
- container [1]
  Designates the region that owns this transition. (Subsets *Namespace.namespace*)

Constraints

[1] A fork segment must not have guards or triggers.

```
(source.oclIsKindOf(Pseudostate) and source.kind = #fork) implies (guard->isEmpty() and trigger->isEmpty())
```

[2] A join segment must not have guards or triggers.

```
((target.oclIsKindOf(Pseudostate)) and (target.kind = #join)) implies ((guard->isEmpty() and (trigger->isEmpty()))
```

[3] A fork segment must always target a state.

```
(source.oclIsKindOf(Pseudostate) and source.kind = #fork) implies (target.oclIsKindOf(State))
```

[4] A join segment must always originate from a state.

```
(target.oclIsKindOf(Pseudostate) and target.kind = #join) implies (source.oclIsKindOf(State))
```

[5] Transitions outgoing pseudostates may not have a trigger (except for those coming out of the initial pseudostate).

```
(source.oclIsKindOf(Pseudostate) and (source.kind <> #initial))  implies trigger->isEmpty()
```

[6] An initial transition at the topmost level (region of a statemachine) either has no trigger or it has a trigger with the stereotype “create.”

```
self.source.oclIsKindOf(Pseudostate) implies
    (self.source.oclAsType(Pseudostate).kind = #initial) implies
        (self.source.container = self.stateMachine.top) implies
            ((self.trigger->isEmpty) or
             (self.trigger.stereotype.name = 'create'))
```

[7] In case of more than one trigger, the signatures of these must be compatible in case the parameters of the signal are assigned to local variables/attributes.

[8] The redefinition context of a transition is the nearest containing statemachine.

```
redefinitionContext =
    let sm = containingStateMachine() in
    if sm.context->isEmpty() or sm.general->notEmpty() then
        sm
    else
        sm.context
    endif
```

Additional operations

[1] 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.

[2] The query containingStateMachine() returns the state machine that contains the transition either directly or transitively.

```
context Transition::containingStateMachine() : StateMachine
    post: result =  container.containingStateMachine()
```

Semantics

High-level transitions

Transitions originating from composite states themselves are called *high-level* or *group* transitions. If triggered, they result in exiting of all the substates of the composite state executing their exit activities starting with the innermost states in the active state configuration. Note that in terms of execution semantics, a high-level transition does not add specialized semantics, but rather reflects the semantics of exiting a composite state. A high-level transition with a target outside the composite state will imply the execution of the exit action of the composite state, while a high-level transition with a target inside the composite state will not imply execution of the exit action of the composite state.

Compound transitions

A *compound transition* is a derived semantic concept, represents a "semantically complete" path made of one or more transitions, originating from a set of states (as opposed to pseudo-state) and targeting a set of states. The transition execution semantics described below refer to compound transitions.

In general, a compound transition is an acyclical unbroken chain of transitions joined via join, junction, choice, or fork pseudostates that define path from a set of source states (possibly a singleton) to a set of destination states, (possibly a singleton). For self-transitions, the same state acts as both the source and the destination set. A (simple) transition connecting two states is therefore a special common case of a compound transition.

The tail of a compound transition may have multiple transitions originating from a set of mutually orthogonal regions that are joined by a join point.

The head of a compound transition may have multiple transitions originating from a fork pseudostate targeted to a set of mutually orthogonal regions.

In a compound transition multiple outgoing transitions may emanate from a common *junction* point. In that case, only one of the outgoing transitions whose guard is true is taken. If multiple transitions have guards that are true, a transition from this set is chosen. The algorithm for selecting such a transition is not specified. Note that in this case, the guards are evaluated before the compound transition is taken.

In a compound transition where multiple outgoing transitions emanate from a common *choice* point, the outgoing transition whose guard is true *at the time the choice point is reached*, will be taken. If multiple transitions have guards that are true, one transition from this set is chosen. The algorithm for selecting this transition is not specified. If no guards are true after the choice point has been reached, the model is ill formed.

Internal transitions

An internal transition executes without exiting or re-entering the state in which it is defined. This is true even if the state machine is in a nested state within this state.

Completion transitions and completion events

A *completion transition* is a transition originating from a state or an exit point but which does not have an explicit trigger, although it may have a guard defined. A completion transition is implicitly triggered by a completion event. In case of a leaf state, a completion event is generated once the entry actions and the internal activities ("do" activities) have been completed. If no actions or activities exist, the completion event is generated upon entering the state. If the state is a composite state or a submachine state, a completion event is generated if either the submachine or the contained region has reached a final state and the state's internal activities have been completed. This event is the implicit trigger for a completion transition. The completion event is dispatched before any other events in the pool and has no associated parameters. For instance, a completion transition emanating from an orthogonal composite state will be taken automatically as soon as all the orthogonal regions have reached their final state.

If multiple completion transitions are defined for a state, then they should have mutually exclusive guard conditions.

Enabled (compound) transitions

A transition is *enabled* if and only if:

- All of its source states are in the active state configuration.
- One of the triggers of the transition is satisfied by the event (type) of the current occurrence. An event *satisfies* a trigger if it matches the event specified by the trigger. In case of signal events, since signals are generalized concepts, a signal event satisfies a signal event associated with the same signal or a generalization thereof.
- If there exists at least one full path from the source state configuration to either the target state configuration or to a dynamic choice point in which all guard conditions are true (transitions without guards are treated as if their guards are always true).

Since more than one transition may be enabled by the same event, being enabled is a necessary but not sufficient condition for the firing of a transition.

Guards

In a simple transition with a guard, the guard is evaluated before the transition is triggered.

In compound transitions involving multiple guards, all guards are evaluated before a transition is triggered, unless there are choice points along one or more of the paths. The order in which the guards are evaluated is not defined.

If there are choice points in a compound transition, only guards that precede the choice point are evaluated according to the above rule. Guards downstream of a choice point are evaluated if and when the choice point is reached (using the same rule as above). In other words, for guard evaluation, a choice point has the same effect as a state.

Guards should not include expressions causing side effects. Models that violate this are considered ill formed.

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.

The *least common ancestor* (LCA) state of a (compound) transition is a region or an orthogonal state that is the LCA of the source and target states of the (compound) transition. The LCA operation is an operation defined for the StateMachine class.

If the LCA is a Region, then the main source is a direct subvertex of the region that contains the source states, and the main target is the subvertex of the region that contains the target states. In the case where the LCA is an orthogonal state, the main source and the main target are both represented by the orthogonal state itself. The reason is that a transition crossing regions of an orthogonal state forces exit from the entire orthogonal state and re-entering of all of its regions.

Examples

- The common simple case: A transition t between two simple states s1 and s2, in a composite state. Here the least common ancestor of t is s, the main source is s1, and the main target is s2.

Note that a transition from one region to another in the same immediate enclosing composite state is not allowed: the two regions must be part of two different composite states. Here least common ancestor of t is s, the main source is s and the main target is s, since s is an orthogonal state as specified above.

Once a transition is enabled and is selected to fire, the following steps are carried out in order:

- The main source state is properly 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 properly entered.

Transition redefinition

A transition of an extended state machine may in the state machine extension be redefined. A redefinition transition redefines all properties of the corresponding replaced transition in the extended state machine, except the source state and the trigger.

Notation

The default notation for a transition is defined by the following BNF expression:

<transition> ::= [<trigger> [‘,’ <trigger>] [‘[‘ <guard-constraint>’]’] [‘/’ <behavior-expression>]]*

However, relative to its use for signal events (see “SignalEvent (from Communications)” on page 450) and change events (see “ChangeEvent (from Communications)” on page 437), the *<assignment-specification>* when used in transitions is extended as follows:

> *<assignment-specification> ::= <attr-spec> [‘,’ <attr-spec>]**
> *<attr-spec> ::= <attr-name> [‘:’ <type-name>]*

Note that *<attr-name>* is the name of an attribute to which the corresponding parameter value of the event is assigned. If a *<type-name>* is included with the attribute name, then it represents an implicit declaration of a local attribute of that type in the context of the effect activity to which the corresponding parameter value of the event is assigned.

The *<guard-constraint>* is a Boolean expression written in terms of parameters of the triggering event and attributes and links of the context object. The guard constraint may also involve tests of orthogonal states of the current state machine, 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 “(RegionOrState1::RegionOrState2::State3.” This may be used in case the same state name occurs in different composite state regions.

The *behavior-expression* is executed if and when the transition fires. It may be written in terms of operations, attributes, and links of the context object and the parameters of the triggering event, or any other features visible in its scope. The behavior expression may be an action sequence comprising a number of distinct actions including actions that explicitly generate events, such as sending signals or invoking operations. The details of this expression are dependent on the action language chosen for the model.

Both in the behavior-expression and in actions of the effect transition specified graphically, the values of the signal instance (in case of a signal trigger) are denoted by *signal-name* '.'*attribute-name* in case of just one signal trigger, and by 'msg.'*attr-name* in case of more than one trigger.

Internal transitions are specified in a special compartment of the source state, see Figure 15.32.

Presentation options

The triggers and the subsequent effect of a transition may be notated either textually or as a presentation option, using graphical symbols on a transition. This sub clause describes the graphical presentation option.

The graphical presentation of triggers and effect of a transition consists of one or more graphical symbols attached to a line connecting the symbols for source and target of the transition each representing a trigger or an action of the transition effect. The action symbols split one logical transition into several graphical line segments with an arrowhead on one end (see Figure 15.44).

The sequence of symbols may consist of a single trigger symbol (or none), followed by a sequence of zero or more *action* symbols. The trigger symbol maps to the set of Triggers. The sequence of action symbols maps to the Behavior representing the effect of the transition. Each action symbol is mapped to an action contained within the single SequenceNode comprising an Activity that is the effect Behavior. The SequenceNode orders the actions according to their graphical order on the transition path.

All line segments connecting the symbols representing source and target of the transition as well as all action symbols represent a single transition, and map to a single behavior. This behavior owns the actions that are represented by the action symbols.

Signal receipt

The trigger 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 represents the trigger of the transition. The textual trigger specification is denoted within the symbol. If the transition owns a guard, the guard is also described within the signal receipt icon. The textual notation:

<trigger> [',' <trigger>] ['[' <guard> ']']*

The trigger symbol is always first in the path of symbols and a transition path can only have at most one such symbol.

Signal sending

Signal sending is a common action that has a special notation described in SendSignalAction. The actual parameters of the signal are shown within the symbol. On a given path, the signal sending symbol must follow the trigger symbol if the latter exists.

The signal sending symbol maps to a SendSignalAction. If a tool does not use the actions package, the details of the sent signal may be captured within the body of the behavior instead of the SendSignalAction instance. It is possible to have multiple signal sending nodes on a transition path.

Other actions

An action symbol is a rectangle that contains a textual representation of the action represented by this symbol. Alternatively, the graphical notation defined for this action, if any, may be used instead. The action symbol must follow the signal receipt symbol if one exists for that transition. The action sequence symbol is mapped to an opaque action, or to a sequence node containing instances of actions, depending on whether it represents one or more actions, respectively.

Note that this presentation option combines only the above symbols into a single transition. Choice and junction symbols (see Figure 15.44) for example, are not mapped to actions on transition but to choice and junction pseudo-states. Therefore, Figure 15.44 shows these 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.

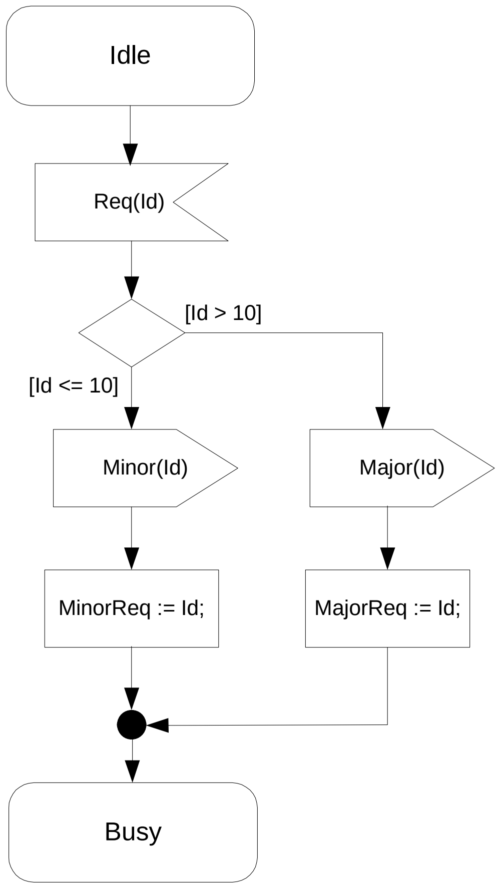


Figure 15.44 - Symbols for Signal Receipt, Sending and Actions on transition

Deferred triggers

A deferrable trigger is shown by listing it within the state followed by a slash and the special operation *defer*. If the event occurs, it is saved and it recurs when the object transitions to another state, where it may be deferred again. When the object reaches a state in which the event is not deferred, it must be accepted or lost. The indication may be placed on a composite state or its equivalents, submachine states, in which case it remains deferrable throughout the composite state. A contained transition may still be triggered by a deferrable event, whereupon it is removed from the pool.

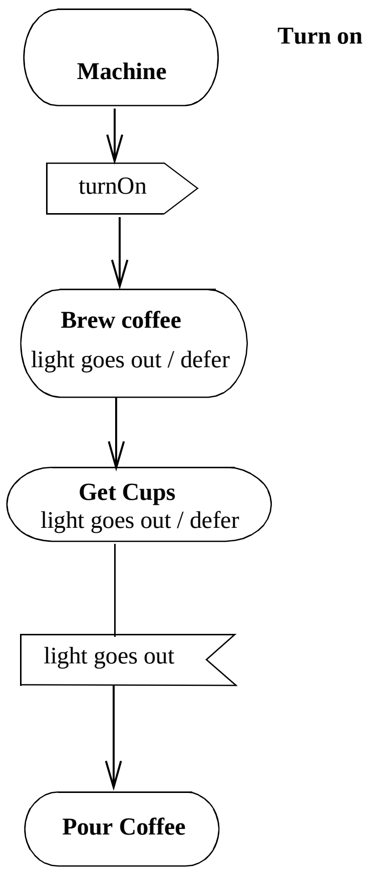


Figure 15.45 - Deferred Trigger Notation

Figure 15.46 shows an example of adding transitions in a specialized state machine.

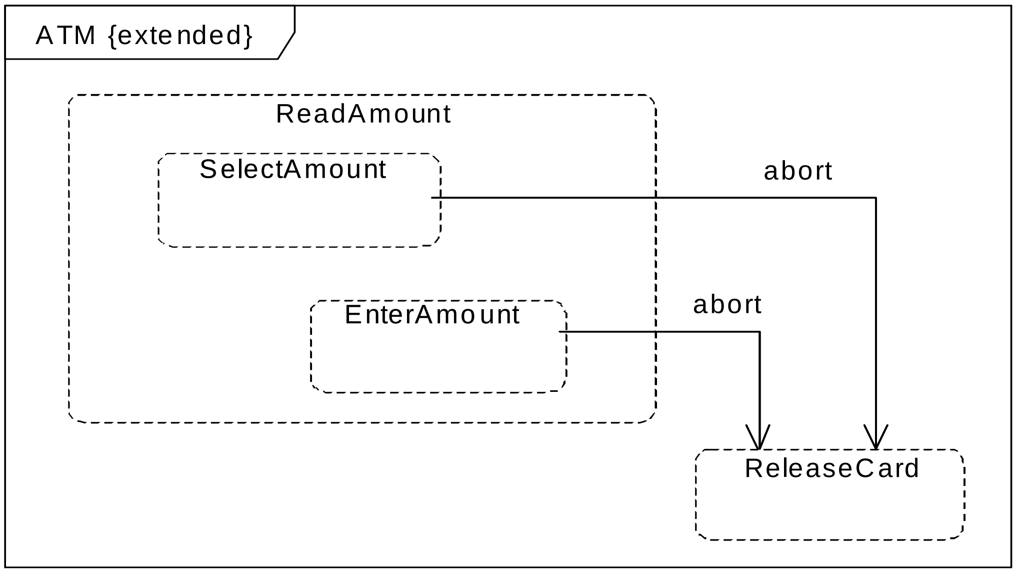


Figure 15.46 - Adding Transitions

Example

Transition with guard constraint and transition string:

right-mouse-down (location) [location in window] / object := pick-object (location);object.highlight ()

The trigger may be any of the standard trigger types. Selecting the type depends on the syntax of the name (for time triggers, for example); however, signal event triggers and call event triggers are not distinguishable by syntax and must be discriminated by their declaration elsewhere.

Changes from previous UML

- Transition redefinition has been added, in order to express if a transition of a general state machine can be redefined or not.
- Along with this, an association to the redefined transition has been added.
- Guard has been replaced by a Constraint.

15.3.15 TransitionKind (from BehaviorStateMachines)

TransitionKind is an enumeration type.

Generalizations

None

Description

TransitionKind is an enumeration of the following literal values:

- external
- internal
- local

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] The source state of a transition with transition kind *local* must be a composite state.

context Transition **inv:**

(kind = TransitionKind::local) **implies** (source.oclIsKindOf (State) and source.isComposite)

[2] The source state of a transition with transition kind *external* must be a composite state.

context Transition **inv:**

(kind = TransitionKind::external) **implies** (source.oclIsKindOf (State) and source.isComposite)

Semantics

- kind=internal implies that the transition, if triggered, occurs without exiting or entering the source state. Thus, 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 state machine is in one or more regions nested within this state.
- kind=local implies that the transition, if triggered, will not exit the composite (source) state, but it will apply to any state within the composite state, and these will be exited and entered.
- kind=external implies that the transition, if triggered, will exit the composite (source) state.

Notation

- Transitions of kind *local* will be on the inside of the frame of the composite state, leaving the border of the composite state and end at a vertex *inside* the composite 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.
- Transitions of kind *external* will leave the border of the composite state and end at either a vertex *outside* the composite state or the composite state itself.

Changes from previous UML

The semantics implied by *local* is new.

15.3.16 Vertex (from BehaviorStateMachines)

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 98

Description

A vertex is an abstraction of a node in a state machine graph. In general, it can be the source or destination of any number of transitions.

Attributes

No additional attributes

Associations

- /outgoing: Transition[0..*]
  Specifies the transitions departing from this vertex. Derived in the following way:

```
context Vertex::outgoing derive:
            Transition.allInstances() -> select(t | t.source = self)
```

- /incoming: Transition[0..*]
  Specifies the transitions entering this vertex. Derived in the following way:

```
context Vertex::incoming derive:
            Transition.allInstances() -> select(t | t.target = self)
```

- container: Region[0..1]
  The region that contains this vertex. {Subsets *Element::owner*}

Additional operations

[1] The operation containingStatemachine() returns the state machine in which this Vertex is defined.

```
context Region::containingStatemachine() : StateMachine
  post: result = if not container->isEmpty() then
    -- the container is a region
    container.containingStateMachine()
    else if (oclIsKindOf(Pseudostate)) then
    -- entry or exit point?
    if (kind = #entryPoint) or (kind = #exitPoint) then
        stateMachine
    else if (oclIsKindOf(ConnectionPointReference)) then
        state.containingStateMachine() -- no other valid cases possible
```

15.4 Diagrams

State machine diagrams specify state machines. This clause outlines the graphic elements that may be shown in state machine 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.

Graphic Nodes

The graphic nodes that can be included in state machine diagrams are shown in Table15.1.

Table15.1 - Graphic nodes included in state machine diagrams

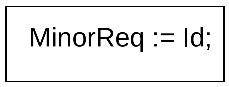
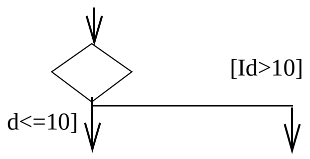
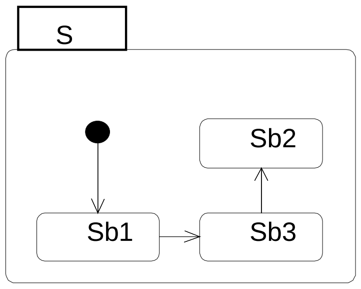
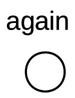
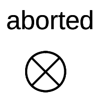
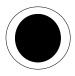
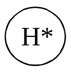
Node Type	Notation	Reference
Action		See “Transition (from BehaviorStateMachines)” on page 572.
Choice pseudostate		See “Pseudostate (from BehaviorStateMachines)” on page 540.
Composite state		See “State (from BehaviorStateMachines, ProtocolStateMachines)” on page 550.
Entry point		See “Pseudostate (from BehaviorStateMachines)” on page 540.
Exit point		See “Pseudostate (from BehaviorStateMachines)” on page 540.
Final state		See “FinalState (from BehaviorStateMachines)” on page 532.
History, Deep pseudostate		See “Pseudostate (from BehaviorStateMachines)” on page 540.

Table15.1 - Graphic nodes included in state machine diagrams

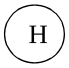
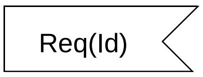
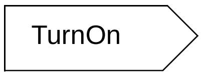
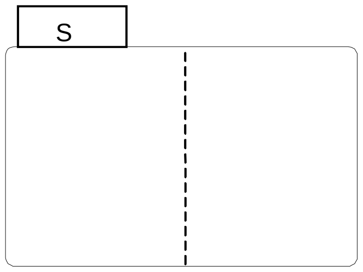
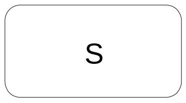
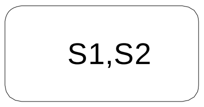
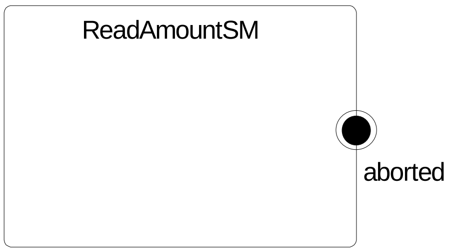
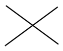
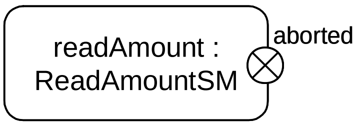
Node Type	Notation	Reference
History, Shallow pseudostate		See "Pseudostate (from BehaviorStateMachines)" on page 540.
Initial pseudostate	●	See "Pseudostate (from BehaviorStateMachines)" on page 540.
Junction pseudostate	●	See "Pseudostate (from BehaviorStateMachines)" on page 540.
Receive signal action		See "Transition (from BehaviorStateMachines)" on page 572.
Send signal action		See "Transition (from BehaviorStateMachines)" on page 572.
Region		See "Region (from BehaviorStateMachines)" on page 548.
Simple state		See "State (from BehaviorStateMachines, ProtocolStateMachines)" on page 550.
State list		See "State (from BehaviorStateMachines, ProtocolStateMachines)" on page 550.

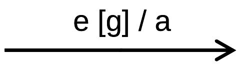
Table15.1 - Graphic nodes included in state machine diagrams

Node Type	Notation	Reference
State Machine		See "StateMachine (from BehaviorStateMachines)" on page 564.
Terminate node		See "Pseudostate (from BehaviorStateMachines)" on page 540.
Submachine state		See "State (from BehaviorStateMachines, ProtocolStateMachines)" on page 550.

Graphic Paths

The graphic paths that can be included in state machine diagrams are shown in Table15.2.

Table15.2 - Graphic paths included in state machine diagrams

Path Type	Notation	Reference
Transition		See "Transition (from BehaviorStateMachines)" on page 572.

Examples

The following are examples of state machine diagrams.

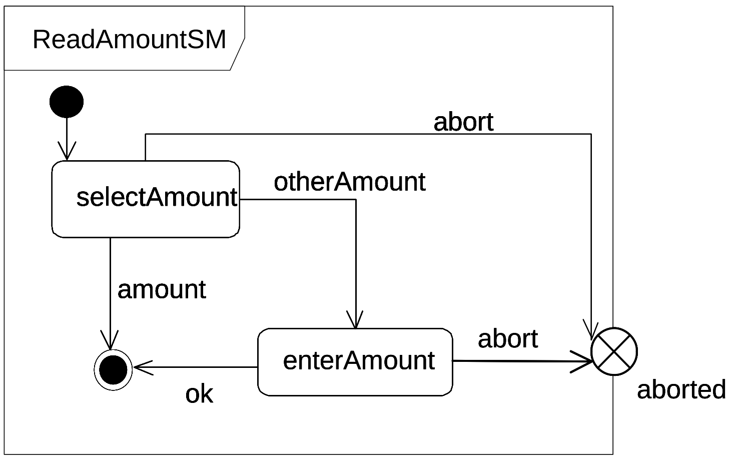


Figure 15.47 - State machine with exit point definition

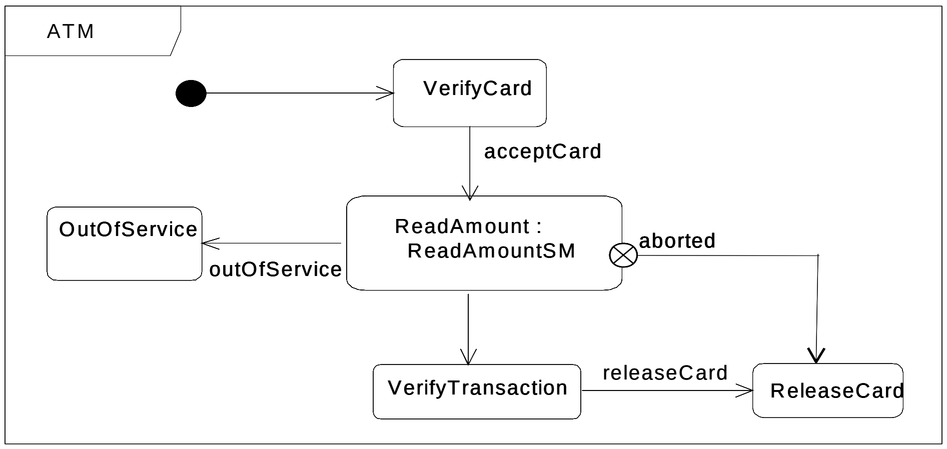


Figure 15.48 - SubmachineState with usage of exit point

16 Use Cases

16.1 Overview

Use cases are a means for specifying required usages of a system. Typically, they are used to capture the requirements of a system, that is, what a system is supposed to do. The key concepts associated with use cases are *actors*, *use cases*, and the *subject*. The subject is the system under consideration to which the use cases apply. The users and any other systems that may interact with the subject are represented as actors. Actors always model entities that are outside the system. The required behavior of the subject is specified by one or more use cases, which are defined according to the needs of actors.

Strictly speaking, the term "use case" refers to a use case type. An instance of a use case refers to an occurrence of the emergent behavior that conforms to the corresponding use case type. Such instances are often described by interaction specifications.

Use cases, actors, and systems are described using use case diagrams.

16.2 Abstract Syntax

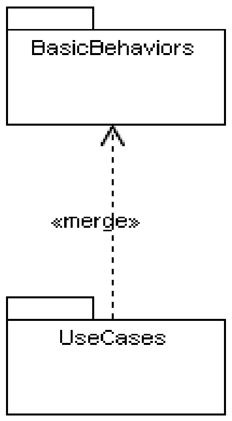


Figure 16.1 - Dependencies of the UseCases package

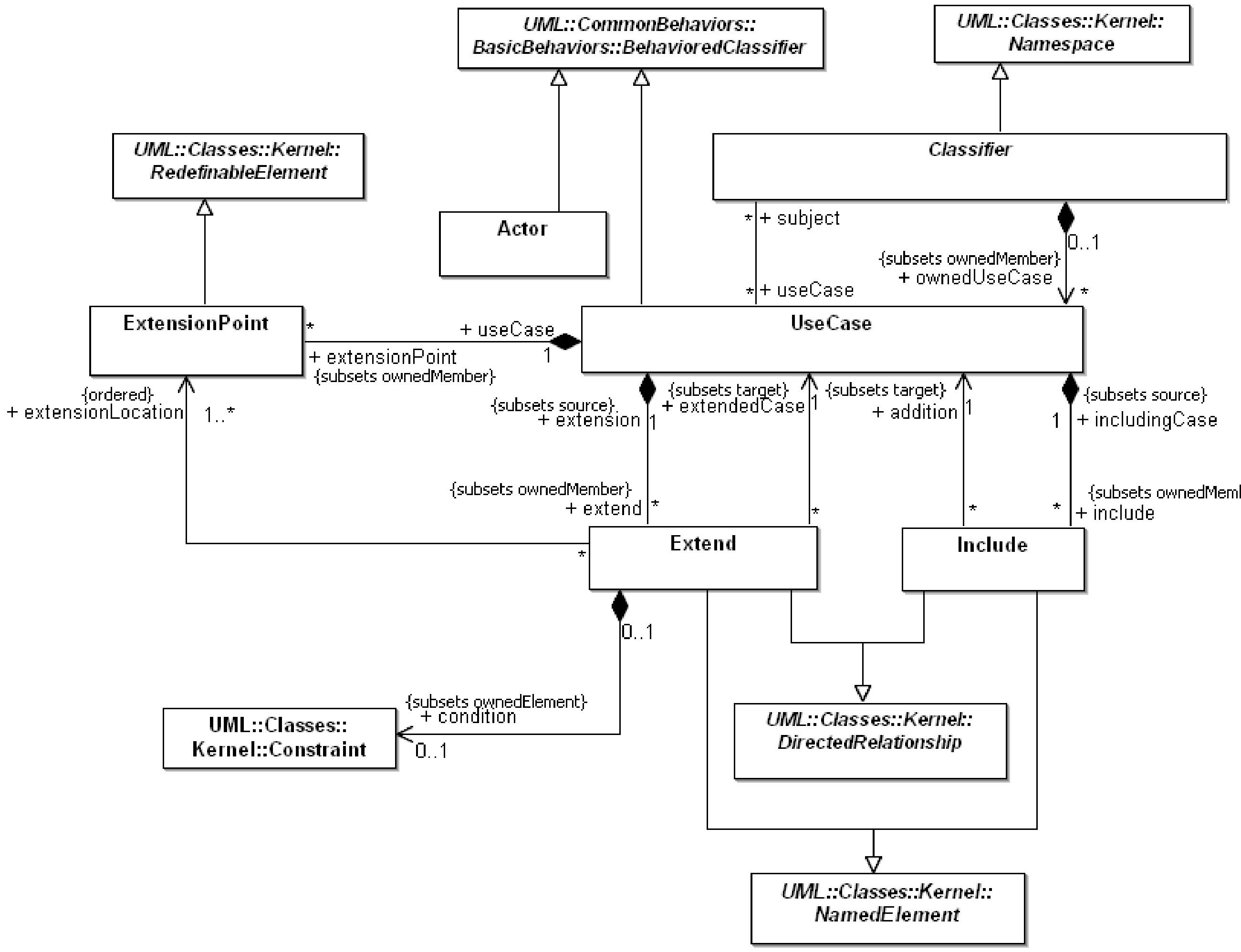

UML::CommonBehaviors::BasicBehaviors::BehavioredClassifier
UML::Classes::Kernel::Namespace
UML::Classes::Kernel::RedefinableElement
Actor
Classifier
* + subject
0..1
{subsets ownedMember} + ownedUseCase
* + useCase
ExtensionPoint
UseCase
+ useCase
+ extensionPoint {subsets ownedMember}
{ordered} + extensionLocation 1..*
{subsets source} + extension 1
{subsets target} + extendedCase 1
{subsets target} + addition 1
{subsets source} + includingCase 1
{subsets ownedMember} + extend *
{subsets ownedMember} + include *
Extend
Include
0..1
{subsets ownedElement} + condition 0..1
UML::Classes::Kernel::Constraint
UML::Classes::Kernel::DirectedRelationship
UML::Classes::Kernel::NamedElement

Figure 16.2 - The concepts used for modeling use cases

16.3 Class Descriptions

16.3.1 Actor (from UseCases)

An actor specifies a role played by a user or any other system that interacts with the subject. (The term “role” is used informally here and does not necessarily imply the technical definition of that term found elsewhere in this specification.)

Generalizations

- “BehavioredClassifier (from BasicBehaviors, Communications)” on page 434

Description

An Actor models a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data), but which is *external* to the subject (i.e., in the sense that an instance of an actor is not a part of the instance of its corresponding subject). Actors may represent roles played by human users, external hardware, or other subjects. Note that an actor does not necessarily represent a specific physical entity but merely a particular facet (i.e., “role”) of some entity that is relevant to the specification of its associated use cases. 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.

Since an actor is external to the subject, it is typically defined in the same classifier or package that incorporates the subject classifier.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] An actor can only have associations to use cases, components, and classes. Furthermore these associations must be binary.

```
self.ownedAttribute->forAll ( a |
    (a.association->notEmpty()) implies
        ((a.association.memberEnd.size() = 2) and
         (a.opposite.class.oclIsKindOf(UseCase) or
          (a.opposite.class.oclIsKindOf(Class) and not a.opposite.class.oclIsKindOf(Behavior))))
```

[2] An actor must have a name.

```
name->notEmpty()
```

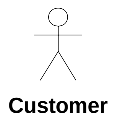
Semantics

Actors model entities external to the subject. When an external entity interacts with the subject, it plays the role of a specific actor.

When an actor has an association to a use case with a multiplicity that is greater than one at the use case end, it means that a given actor can be involved in multiple use cases 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 use cases in parallel (concurrently) or they may be mutually exclusive in time. For example, a computer user may activate a given software application multiple times concurrently or at different points in time.

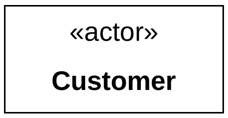
Notation

An actor is represented by “stick man” icon with the name of the actor in the vicinity (usually above or below) the icon.

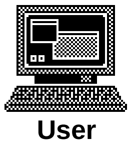


Presentation Options

An actor may also be shown as a class 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.



Style Guidelines

Actor names should follow the capitalization and punctuation guidelines used for classes in the model. The names of abstract actors should be shown in italics.

Changes from previous UML

There are no changes to the Actor concept except for the addition of a constraint that requires that all actors must have names.

16.3.2 Classifier (from UseCases)

Generalizations

- "Classifier (from Kernel, Dependencies, PowerTypes)" on page 52 *(merge increment)*

Description

Extends a classifier with the capability to own use cases. Although the owning classifier typically represents the subject to which the owned use cases apply, this is not necessarily the case. In principle, the same use case can be applied to multiple subjects, as identified by the *subject* association role of a UseCase (see "UseCase (from UseCases)" on page 596).

Attributes

No additional attributes

Associations

- ownedUseCase: UseCase[*]
  References the use cases owned by this classifier. (Subsets *Namespace.ownedMember*)
- useCase : UseCase [*]
  The set of use cases for which this Classifier is the subject.

Constraints

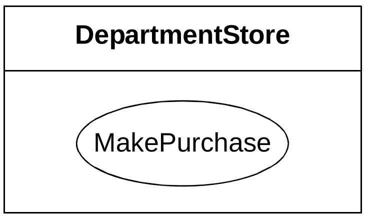
No additional constraints

Semantics

See “UseCase (from UseCases)” on page 596.

Notation

The nesting (owning) of a use case by a classifier is represented using the standard notation for nested classifiers.



Rationale

This extension to the Classifier concept was added to allow classifiers in general to own use cases.

Changes from previous UML

No changes

16.3.3 Extend (from UseCases)

A relationship from an extending use case to an extended use case that specifies how and when the behavior defined in the extending use case can be inserted into the behavior defined in the extended use case.

Generalizations

- “DirectedRelationship (from Kernel)” on page 63
- “NamedElement (from Kernel, Dependencies)” on page 98

Description

This relationship specifies that the behavior of a use case may be extended by the behavior of another (usually supplementary) use case. The extension takes place at one or more specific extension points defined in the extended use case. Note, however, that the extended use case is defined independently of the extending use case and is meaningful independently of the extending use case. On the other hand, the extending use case typically defines behavior that may not necessarily be meaningful by itself. Instead, the extending use case defines a set of modular behavior increments that augment an execution of the extended use case under specific conditions.

Note that the same extending use case can extend more than one use case. Furthermore, an extending use case may itself be extended.

It is a kind of DirectedRelationship, such that the source is the extending use case and the destination is the extended use case. It is also a kind of NamedElement so that it can have a name in the context of its owning use case. The extend relationship itself is owned by the extending use case.

Attributes

No additional attributes

Associations

- extendedCase : UseCase [1]
  References the use case that is being extended. (Subsets *DirectedRelationship.target*)
- extension : UseCase [1]
  References the use case that represents the extension and owns the extend relationship. (Subsets *DirectedRelationship.source*)
- condition : Constraint [0..1]
  References the condition that must hold when the first extension point is reached for the extension to take place. If no constraint is associated with the extend relationship, the extension is unconditional. (Subsets *Element.ownedElement*)
- extensionLocation: ExtensionPoint [1..*]
  An ordered list of extension points belonging to the extended use case, specifying where the respective behavioral fragments of the extending use case are to be inserted. The first fragment in the extending use case 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 use case has just a single behavior fragment, so that the list of extension points is trivial.)

Constraints

[1] The extension points referenced by the extend relationship must belong to the use case that is being extended.

```
extensionLocation->forAll (xp | extendedCase.extensionPoint->includes(xp))
```

Semantics

The concept of an "extension location" is intentionally left underspecified because use cases are typically specified in various idiosyncratic 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 extension location is best explained through the example of a textually described use case: Usually, a use case with extension points consists of a set of finer-grained behavioral fragment descriptions, which are most often executed in sequence. This segmented structuring of the use case 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 use case typically consists of one or more behavior fragment descriptions that are to be inserted into the appropriate spots of the extended use case. An extension location, therefore, is a specification of all the various (extension) points in a use case where supplementary behavioral increments can be merged.

If the condition of the extension is true at the time the first extension point is reached during the execution of the extended use case, then all of the appropriate behavior fragments of the extending use case will also be executed. If the condition is false, the extension does not occur. The individual fragments are executed as the corresponding extension points of the extending use case are reached. Once a given fragment is completed, execution continues with the behavior of the extended use case following the extension point. Note that even though there are multiple use cases involved, there is just a single behavior execution.

Notation

An extend relationship between use cases is shown by a dashed arrow with an open arrowhead from the use case providing the extension to the base use case. The arrow is labeled with the keyword «extend». The condition of the relationship as well as the references to the extension points are optionally shown in a Note attached to the corresponding extend relationship.(See Figure 16.3.)

Examples

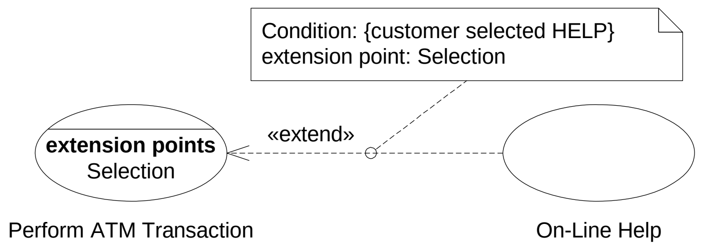


Figure 16.3 - Example of an extend relationship between use cases

In the use case diagram above, the use case “Perform ATM Transaction” has an extension point “Selection.” This use case is extended via that extension point by the use case “On-Line Help” whenever execution of the “Perform ATM Transaction” use case occurrence is at the location referenced by the “Selection” extension point and the customer selects the HELP key. Note that the “Perform ATM Transaction” use case is defined independently of the “On-Line Help” use case.

Rationale

This relationship is intended to be used when there is some additional behavior that should be added, possibly conditionally, to the behavior defined in another use case (which is meaningful independently of the extending use case).

Changes from previous UML

The notation for conditions has been changed such that the condition and the referenced extension points may now be included in a Note attached to the extend relationship, instead of merely being a textual comment that is located in the vicinity of the relationship.

16.3.4 ExtensionPoint (from UseCases)

An extension point identifies a point in the behavior of a use case where that behavior can be extended by the behavior of some other (extending) use case, as specified by an extend relationship.

Generalizations

- “RedefinableElement (from Kernel)” on page 130

Description

An ExtensionPoint is a feature of a use case that identifies a point where the behavior of a use case can be augmented with elements of another (extending) use case.

Attributes

No additional attributes

Associations

- useCase : UseCase [1]
  References the use case that owns this extension point. {Subsets *NamedElement::namespace*}

Constraints

[1] An ExtensionPoint must have a name.

```
self.name->notEmpty ()
```

Semantics

An extension point is a reference to a location within a use case at which parts of the behavior of other use cases may be inserted. Each extension point has a unique name within a use case.

Semantic Variation Points

The specific manner in which the location of an extension point is defined is left as a semantic variation point.

Notation

Extension points are indicated by a text string within in the use case oval symbol or use case rectangle 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 use case where the extension point occurs.

Examples

See Figure 16.3 on page 593 and Figure 16.9 on page 600.

Rationale

ExtensionPoint supports the use case extension mechanism (see "Extend (from UseCases)" on page 591).

Changes from previous UML

In 1.x, ExtensionPoint was modeled as a kind of ModelElement, which due to a multiplicity constraint, was always associated with a specific use case. This relationship is now modeled as an owned feature of a use case. Semantically, this is equivalent and the change will not be visible to users.

ExtensionPoints in 1.x had an attribute called *location*, which was a kind of LocationReference. Since the latter had no specific semantics it was relegated to a semantic variation point. When converting to UML 2.0, models in which ExtensionPoints had a *location* attribute defined, the contents of the attribute should be included in a note attached to the ExtensionPoint.

16.3.5 Include (from UseCases)

An include relationship defines that a use case contains the behavior defined in another use case.

Generalizations

- "DirectedRelationship (from Kernel)" on page 63
- "NamedElement (from Kernel, Dependencies)" on page 98

Description

Include is a DirectedRelationship between two use cases, implying that the behavior of the included use case is inserted into the behavior of the including use case. It is also a kind of NamedElement so that it can have a name in the context of its owning use case. The including use case may only depend on the result (value) of the included use case. This value is obtained as a result of the execution of the included use case.

Note that the included use case is not optional, and is always required for the including use case to execute correctly.

Attributes

No additional attributes

Associations

- addition : UseCase [1]
  References the use case that is to be included. (Subsets *DirectedRelationship.target*)
- including Case : UseCase [1]
  References the use case that will include the addition and owns the include relationship. (Subsets *DirectedRelationship.source*)

Constraints

No additional constraints

Semantics

An include relationship between two use cases means that the behavior defined in the including use case is included in the behavior of the base use case. The include relationship is intended to be used when there are common parts of the behavior of two or more use cases. This common part is then extracted to a separate use case, to be included by all the base use cases having this part in common. Since the primary use of the include relationship is for reuse of common parts, what is left in a base use case 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 use case depends on the addition but not vice versa.

Execution of the included use case is analogous to a subroutine call. All of the behavior of the included use case is executed at a single location in the included use case before execution of the including use case is resumed.

Notation

An include relationship between use cases is shown by a dashed arrow with an open arrowhead from the base use case to the included use case. The arrow is labeled with the keyword «include». (See Figure 16.4.)

Examples

A use case “Withdraw” includes an independently defined use case “Card Identification.”

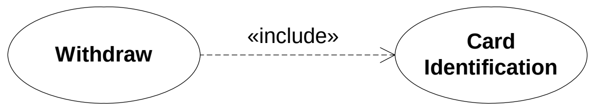


Figure 16.4 - Example of the Include relationship

Rationale

The Include relationship allows hierarchical composition of use cases as well as reuse of use cases.

Changes from previous UML

There are no changes to the semantics or notation of the Include relationship relative to UML 1.x.

16.3.6 UseCase (from UseCases)

A use case is the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system.

Generalizations

- “BehavioredClassifier (from BasicBehaviors, Communications)” on page 434

Description

A UseCase is a kind of behaviored classifier that represents a declaration of an offered behavior. Each use case specifies some behavior, possibly including variants, that the subject can perform in collaboration with one or more actors. Use cases define the offered behavior of the subject without reference to its internal structure. These behaviors, involving interactions between the actor and the subject, may result in changes to the state of the subject and communications with its environment. A use case can include possible variations of its basic behavior, including exceptional behavior and error handling.

The subject of a use case could be a physical system or any other element that may have behavior, such as a component, subsystem, or class. Each use case 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 use case 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 use case can be initiated again or in an error state.

Use cases 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 use cases also state the requirements the specified subject poses on its environment by defining how they should interact with the subject so that it will be able to perform its services.

The behavior of a use case can be described by a specification that is some kind of Behavior (through its ownedBehavior relationship), such as interactions, activities, and state machines, 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 use case and its actors as the classifiers that type its parts. Which of these techniques to use depends on the nature of the use case behavior as well as on the intended reader. These descriptions can be combined. An example of a use case with an associated state machine description is shown in Figure 16.6.

Attributes

No additional attributes

Associations

- subject : Classifier[*]
  References the subjects to which this use case applies. The subject or its parts realize all the use cases that apply to this subject. Use cases need not be attached to any specific subject, however. The subject may, but need not, own the use cases that apply to it.
- include : Include[*]
  References the Include relationships owned by this use case. (Subsets *Namespace.ownedMember*)
- extend : Extend[*]
  References the Extend relationships owned by this use case. (Subsets and *Namespace.ownedMember*)
- extensionPoint: ExtensionPoint[*]
  References the ExtensionPoints owned by the use case. (Subsets *Namespace.ownedMember*)

Constraints

[1] A UseCase must have a name.

```
self.name -> notEmpty ()
```

[2] UseCases can only be involved in binary Associations.

(no OCL available)

[3] UseCases cannot have Associations to UseCases specifying the same subject.

(no OCL available)

[4] A use case cannot include use cases that directly or indirectly include it.

```
not self.allIncludedUseCases()->includes(self)
```

Additional Operations

[1] The query allIncludedUseCases() returns the transitive closure of all use cases (directly or indirectly) included by this use case.

```
UseCase::allIncludedUseCases() : Set(UseCase)
allIncludedUseCases = self.include->union(self.include->collect(in | in.allIncludedUseCases()))
```

Semantics

An execution of a use case is an occurrence of emergent behavior.

Every instance of a classifier realizing a use case must behave in the manner described by the use case.

Use cases may have associated actors, which describes how an instance of the classifier realizing the use case and a user playing one of the roles of the actor interact. Two use cases specifying the same subject cannot be associated since each of them individually describes a complete usage of the subject. It is not possible to state anything about the internal behavior of the actor apart from its communications with the subject.

When a use case 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 use case. The manner in which multiple actors participate in the use case depends on the specific situation on hand and is not defined in this specification. For instance, a particular use case 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).

Notation

A use case is shown as an ellipse, either containing the name of the use case or with the name of the use case placed below the ellipse. An optional stereotype keyword may be placed above the name and a list of properties included below the name. If a subject (or system boundary) is displayed, the use case ellipse is visually located inside the system boundary rectangle. Note that this does not necessarily mean that the subject classifier owns the contained use cases, but merely that the use case applies to that classifier. For example, the use cases shown in Figure 16.5 on page 598 apply to the "ATMsystem" classifier but are owned by various packages as shown in Figure 16.7.

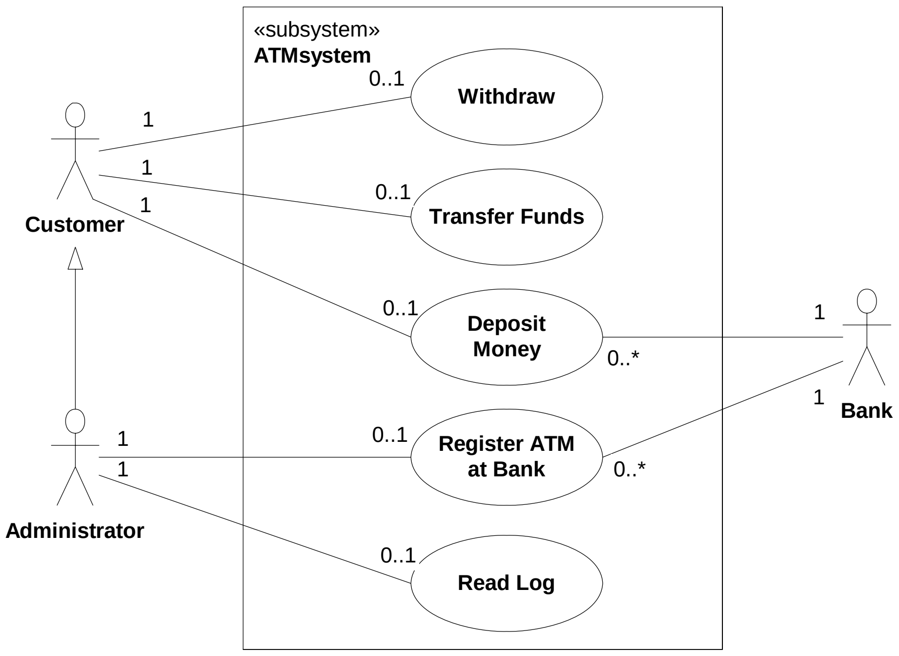


Figure 16.5 - Example of the use cases and actors for an ATM system

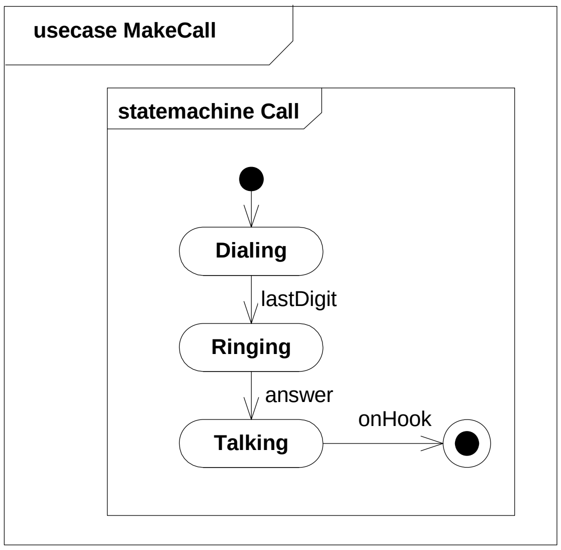


Figure 16.6 - Example of a use case with an associated state machine behavior

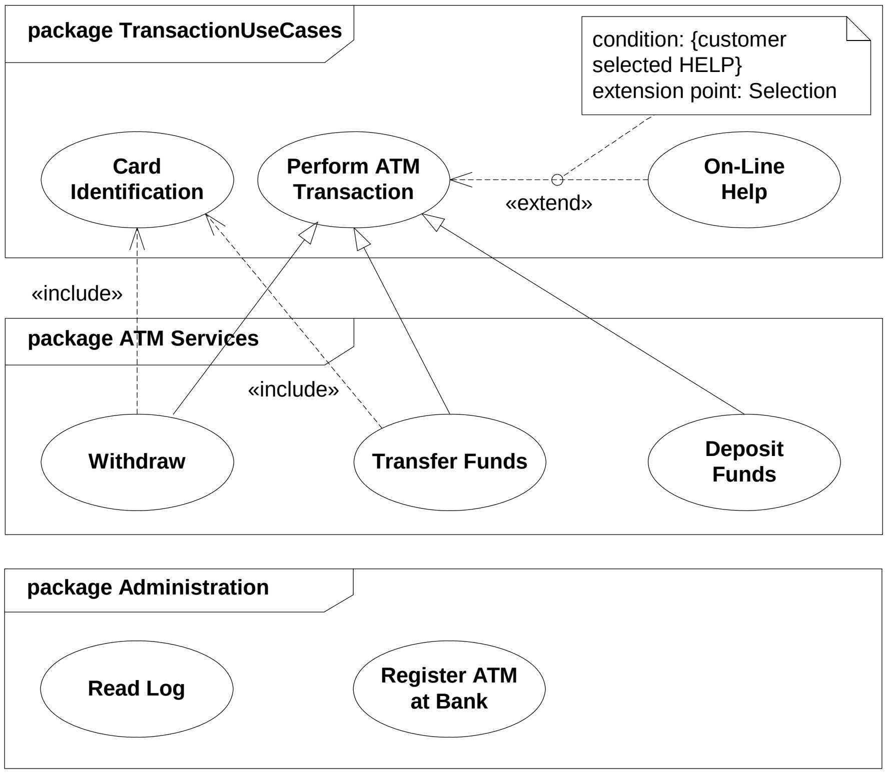


Figure 16.7 - Example of use cases owned by various packages

Extension points may be listed in a compartment of the use case with the heading **extension points**. The description of the locations of the extension point is given in a suitable form, usually as ordinary text, but can also be given in other forms, such as the name of a state in a state machine, an activity in an activity diagram, a precondition, or a postcondition.

Use cases may have other associations and dependencies to other classifiers (e.g., to denote input/output, events, and behaviors).

The detailed behavior defined by a use case is notated according to the chosen description technique, in a separate diagram or textual document. Operations and attributes are shown in a compartment within the use case.

Use cases and actors may represent roles in collaborations as indicated in Figure 16.8.

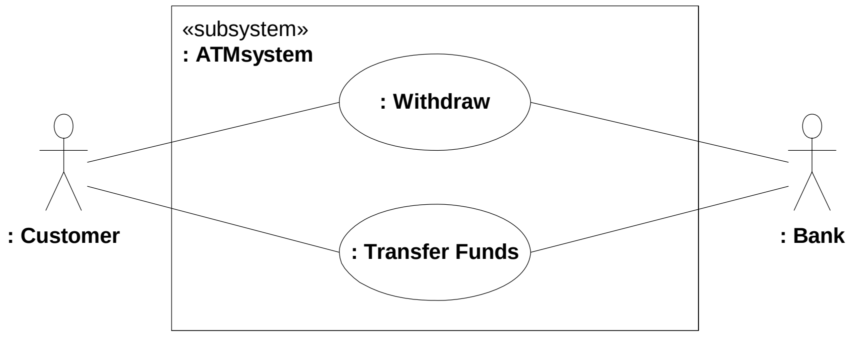


Figure 16.8 - Example of a use case for withdrawal and transfer of funds

Presentation Options

A use case can also be shown using the standard rectangle notation for classifiers with an ellipse icon in the upper-right-hand corner of the rectangle with optional separate list compartments for its features. This rendering is more suitable when there are a large number of extension points.

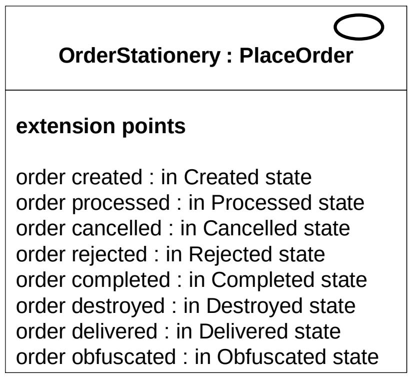

OrderStationery : PlaceOrder
extension points
order created : in Created state
order processed : in Processed state
order cancelled : in Cancelled state
order rejected : in Rejected state
order completed : in Completed state
order destroyed : in Destroyed state
order delivered : in Delivered state
order obfuscated : in Obfuscated state

Figure 16.9 - Example of the classifier based notation for a use case

Examples

See Figure 16.3 through Figure 16.9.

Rationale

The purpose of use cases is to identify the required functionality of a system.

Changes from previous UML

The relationship between a use case and its subject has been made explicit. Also, it is now possible for use cases to be owned by classifiers in general and not just packages.

16.4 Diagrams

Description

Use Case Diagrams are a specialization of Class Diagrams such that the classifiers shown are restricted to being either Actors or Use Cases.

Graphic Nodes

The graphic nodes that can be included in structural diagrams are shown in Table 16.1.

Table 16.1 - Graphic nodes included in use case diagrams

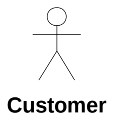
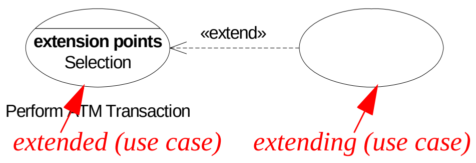
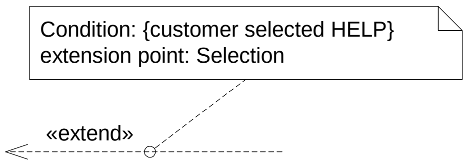
Node Type	Notation	Reference
Actor (default)		See "Actor (from UseCases)" on page 588.
Actor (optional user-defined icon - example)		
Extend	 extension points Selection «extend» Perform ATM Transaction extended (use case) extending (use case)	See "Extend (from UseCases)" on page 591.
Extend (with Condition)	 Condition: {customer selected HELP} extension point: Selection «extend»	

Table 16.1 - Graphic nodes included in use case diagrams

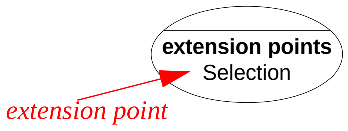
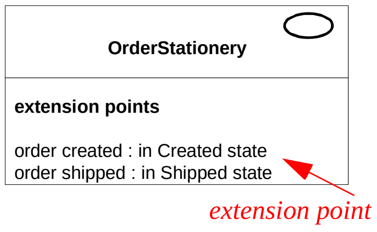
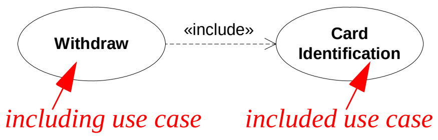
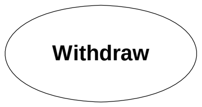
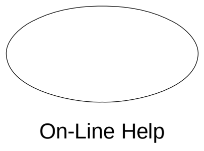
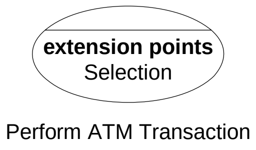
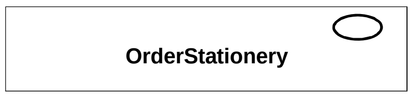
Node Type	Notation	Reference
ExtensionPoint		See “ExtensionPoint (from UseCases)” on page 593.
	 OrderStationery extension points order created : in Created state order shipped : in Shipped state extension point	
Include	 Withdraw «include» Card Identification including use case included use case	See “Include (from UseCases)” on page 594.

Table 16.1 - Graphic nodes included in use case diagrams

Node Type	Notation	Reference
UseCase		See “UseCase (from UseCases)” on page 596.
		
		
		

Examples

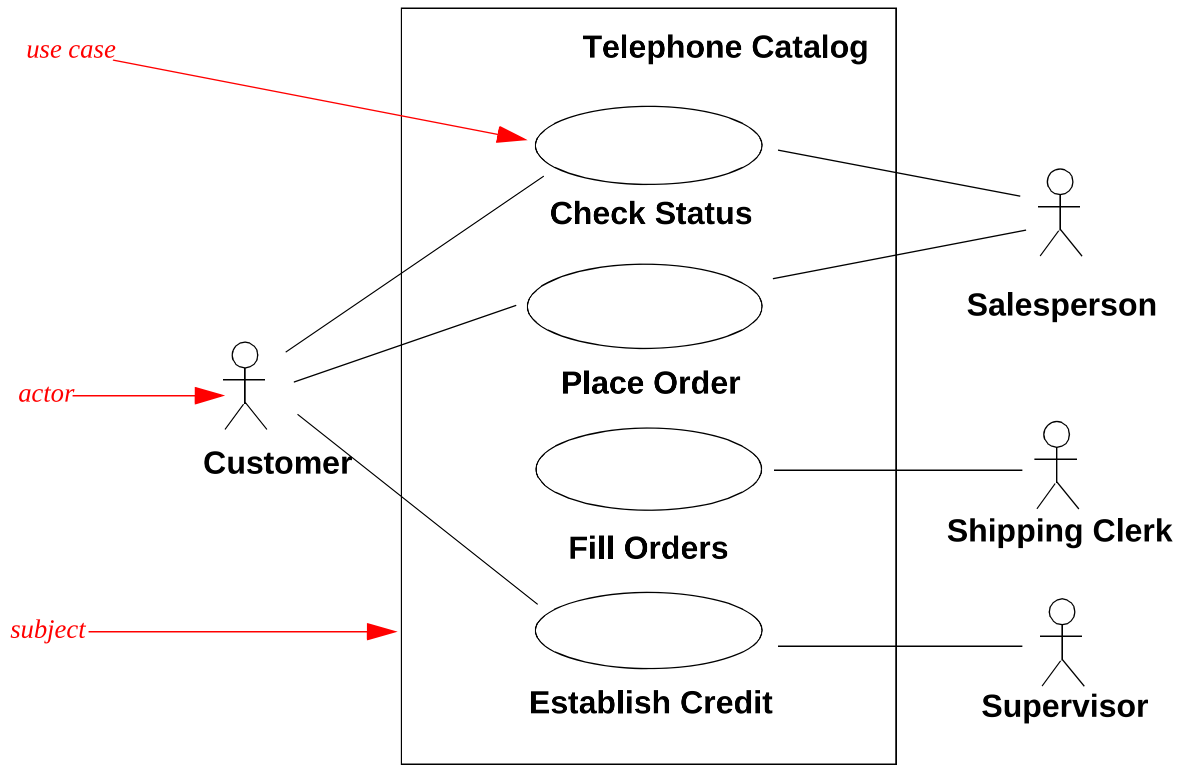


Figure 16.10 - UseCase diagram with a rectangle representing the boundary of the subject

The use case diagram in Figure 16.10 shows a set of use cases used by four actors of a physical system that is the subject of those use cases. The subject can be optionally represented by a rectangle as shown in this example.

Figure 16.11 illustrates a package that owns a set of use cases.

Note – A use case may be owned either by a package or by a classifier (typically the classifier specifying the subject).

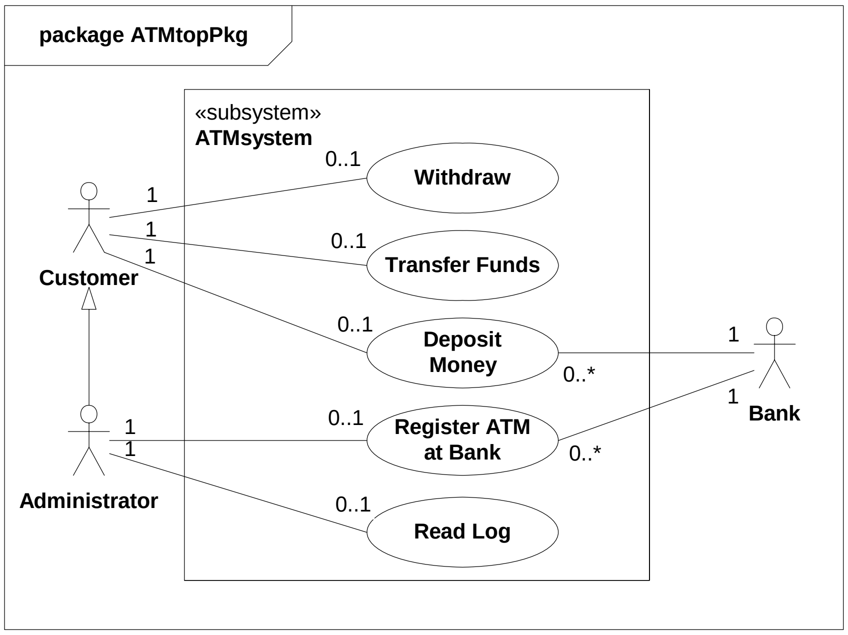


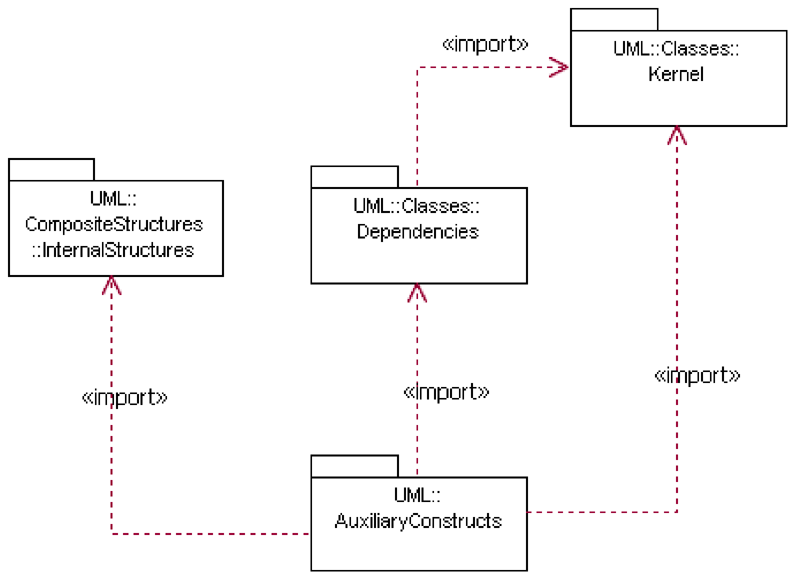
Figure 16.11 - Use cases owned by a package

Changes from previous UML

There are no changes from UML 1.x, although some aspects of notation to model element mapping have been clarified.

Part III - Supplement

This part defines auxiliary constructs (e.g., information flows, models, templates, primitive types) and the profiles used to customize UML for various domains, platforms, and methods. The UML packages that support auxiliary constructs, along with the structure packages they depend upon (InternalStructures, Dependencies, and Kernel) are shown in the figure below.



Part III, Figure 1 - UML packages that support auxiliary constructs

The function and contents of these packages are described in the following clauses, which are organized by major subject areas.

17 Auxiliary Constructs

17.1 Overview

This clause defines mechanisms for information flows, models, primitive types, and templates.

Package structure

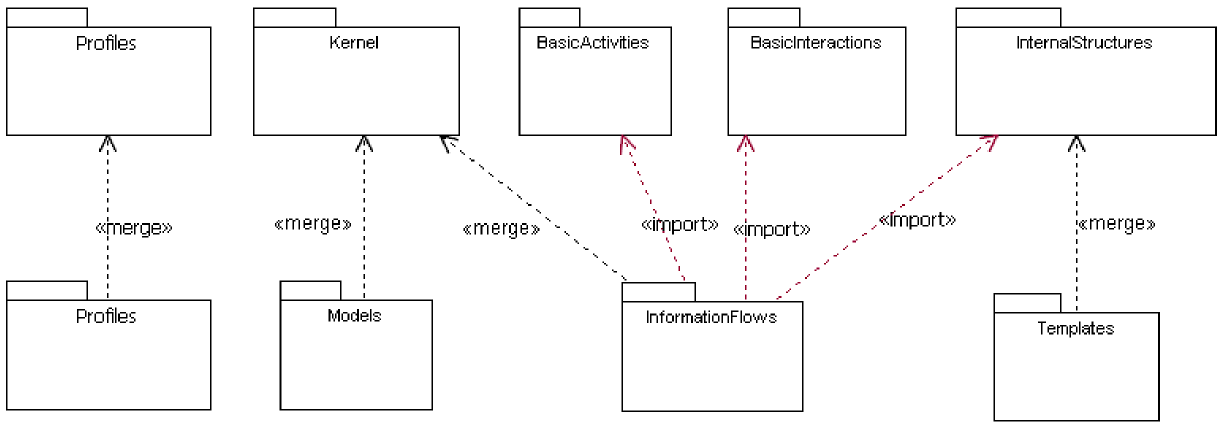


Figure 17.1 - Dependencies between packages described in this clause

17.2 InformationFlows

The InformationFlows package provides mechanisms for specifying the exchange of information between entities of a system at a high level of abstraction. Information flows describe circulation of information in a system in a general manner. They do not specify the nature of the information (type, initial value), nor the mechanisms by which this information is conveyed (message passing, signal, common data store, parameter of operation, etc.). They also do not specify sequences or any control conditions. It is intended that, while modeling in detail, representation and realization links will be able to specify which model element implements the specified information flow, and how the information will be conveyed.

The contents of the InformationFlows package is shown in Figure 17.2. The InformationFlows package is one of the packages of the AuxiliaryConstructs package.

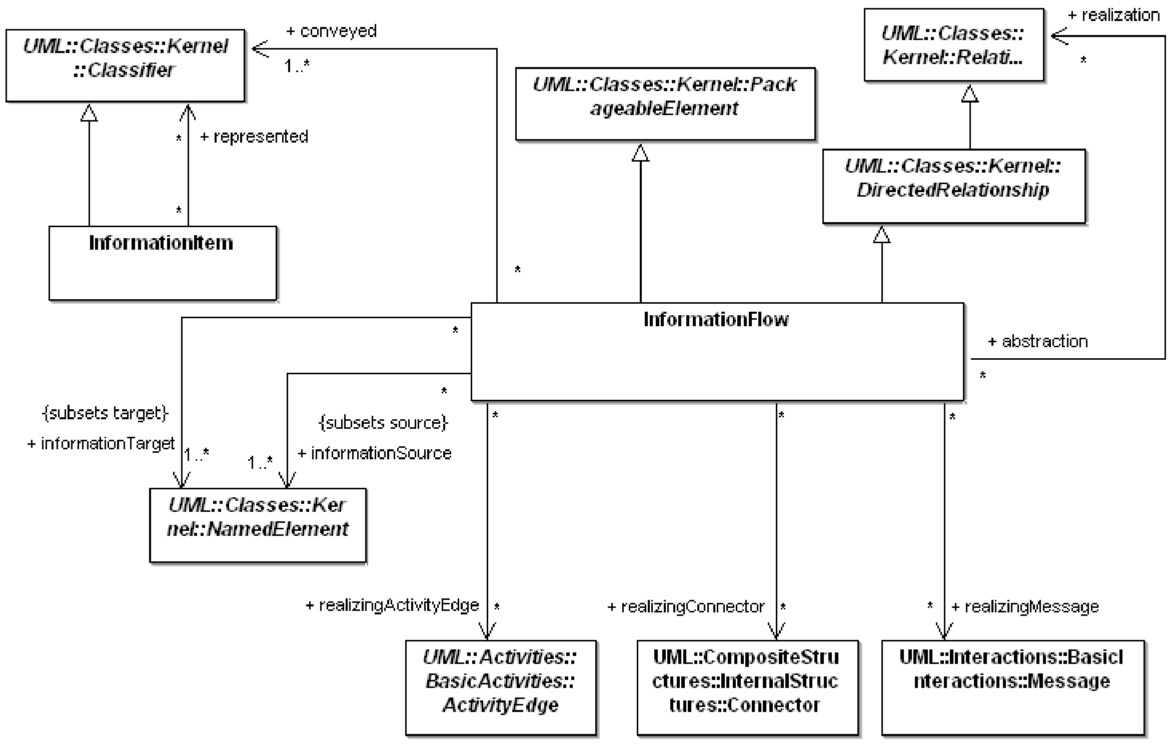

UML::Classes::Kernel ::Classifier
+ conveyed
1..*
+ represented
*
*
InformationItem
UML::Classes::Kernel::Pack ageableElement
UML::Classes:: Kernel::Relati...
+ realization
*
UML::Classes::Kernel:: DirectedRelationship
InformationFlow
+ abstraction
*
{subsets target}
+ informationTarget
1..*
{subsets source}
+ informationSource
1..*
UML::Classes::Ker nel::NamedElement
+ realizingActivityEdge
+ realizingConnector
+ realizingMessage
UML::Activities:: BasicActivities:: ActivityEdge
UML::CompositeStru ctures::InternalStruc tures::Connector
UML::Interactions::BasicI nteractions::Message

Figure 17.2 - The contents of the InformationFlows package

17.2.1 InformationFlow (from InformationFlows)

Generalizations

- “DirectedRelationship (from Kernel)” on page 63
- “PackageableElement (from Kernel)” on page 109

Description

An InformationFlow specifies that one or more information items circulates from its sources to its targets. Information flows require some kind of “information channel” for transmitting information items from the source to the destination. An information channel is represented in various ways depending on the nature of its sources and targets. It may be represented by connectors, links, associations, or even dependencies. For example, if the source and destination are parts in some composite structure such as a collaboration, then the information channel is likely to be represented by a connector between them. Or, if the source and target are objects (which are a kind of InstanceSpecification), they may be represented by a link that joins the two, and so on.

Attributes

No additional attributes

Associations

- realization : Relationship [*]
  Determines which Relationship will realize the specified flow.
- conveyed : Classifier [1..*]
  Specifies the information items that may circulate on this information flow.
- realizingConnector : Connector[*]
  Determines which Connectors will realize the specified flow.
- realizingActivityEdge : ActivityEdge [*]
  Determines which ActivityEdges will realize the specified flow.
- realizingMessage : Message[*]
  Determines which Messages will realize the specified flow.
- target : NamedElement [1..*]
  Defines to which target the conveyed information items are directed. {Subsets *DirectedRelationship::target*}
- source : NamedElement [1..*]
  Defines from which source the conveyed information items are initiated. {Subsets *DirectedRelationship::source*}

Constraints

[1] 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).

```
(self.source->forAll(p | p->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.target-> forAll(p | p->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)))
```

[2] The sources and targets of the information flow must conform with the sources and targets or, conversely, the targets and sources of the realization relationships, if any.

[3] An information flow can only convey classifiers that are allowed to represent an information item (see constraints on “InformationItem (from InformationFlows)” on page 612).

```
self.conveyed.represented->forAll(p |
    p->oclIsKindOf(Class) or  oclIsKindOf(Interface) or  oclIsKindOf(InformationItem) or  oclIsKindOf(Signal)
        or  oclIsKindOf(Component)
```

Semantics

An information flow is an abstraction of the communication of an information item from its sources to its targets. It is used to abstract the communication of information between entities of a system. Sources or targets of an information flow designate sets of objects that can send or receive the conveyed information item. When a source or a target is a classifier, it represents all the potential instances of the classifier; when it is a part, it represents all instances that can play the role specified by the part; when it is a package, it represents all potential instances of the directly or indirectly owned classifiers of the package.

An information flow may directly indicate a concrete classifier, such as a class, that is conveyed instead of using an information item.

Notation

An information flow is represented as a dependency, with the keyword <<flow>>.

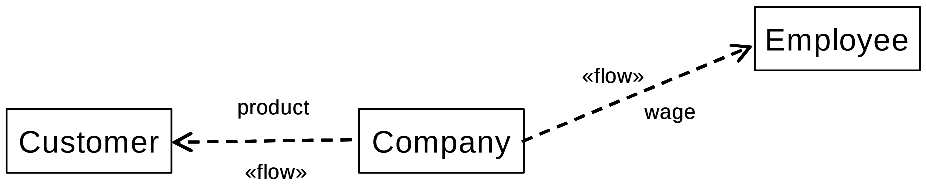


Figure 17.3 - Example of information flows conveying information items

Changes from previous UML

InformationFlow does not exist in UML 1.4.

17.2.2 InformationItem (from InformationFlows)

Generalizations

- "Classifier (from Kernel, Dependencies, PowerTypes)" on page 52

Description

An information item is an abstraction of all kinds of information that can be exchanged between objects. It is a kind of classifier intended for representing information at a very abstract way, one which cannot be instantiated.

One purpose of information items is to be able to define preliminary models, before having made detailed modeling decisions on types or structures. One other purpose of information items and information flows is to abstract complex models by a less precise but more general representation of the information exchanged between entities of a system.

Attributes

No additional attributes

Associations

- represented : Classifier [*]
  Determines the classifiers that will specify the structure and nature of the information. An information item represents all its represented classifiers.

Constraints

[1] 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.

```
(self.represented.select(p | p->oclIsKindOf(InformationItem))->forAll(p |
    p.informationFlow.source->forAll( q |
        self.informationFlow.source->include(q) )
    and p.informationFlow.target->forAll( q |
        self.informationFlow.target->include(q) ) ))
and
(self.represented->forAll(p |
    p->oclIsKindOf(Class) or oclIsKindOf(Interface) or oclIsKindOf(InformationItem) or oclIsKindOf(Signal)
        or oclIsKindOf(Component)))
```

[2] An information item has no feature, no generalization, and no associations.

```
self.generalization->isEmpty() and self.feature->isEmpty()
```

[3] It is not instantiable.

```
isAbstract
```

Semantics

“Information” as represented by an information item encompasses all sorts of data, events, facts that are exploited inside the modeled system. For example, the information item “wage” can represent a Salary Class, or a Bonus Class (see example Figure 17.5). An information item does not specify the structure, the type, or the nature of the represented information. It specifies at an abstract level the elements of information that will be exchanged inside a system. More accurate description will be provided by defining the classifiers represented by information item.

Information items can be decomposed into more specific information items using representation links between them. This gives the ability to express that in specific contexts (specific information flows) a specific information is exchanged.

Information items cannot have properties, or associations. Specifying this detailed information belongs to the represented classifier.

Notation

Being a classifier, an information item can be represented as a name inside a rectangle. The keyword «information» or the black triangle icon on top of this rectangle indicates that it is an information item.

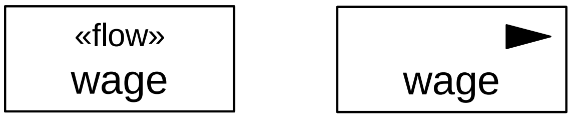


Figure 17.4 - Information Item represented as a classifier

Representation links between a classifier and a representation item are represented as dashed lines with the keyword «representation».

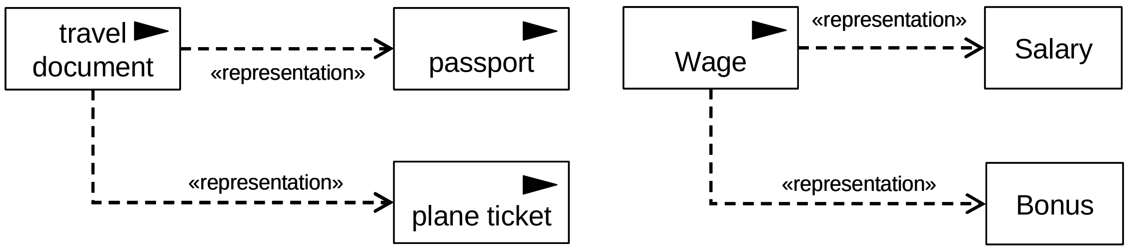


Figure 17.5 - Examples of «representation» notation

An information item is usually represented attached to an information flow, or to a relationship realizing an information flow. When it is attached to an information flow (see Figure 17.3) its name is displayed close to the information flow line. When it is attached to an information channel, a black triangle on the information channel indicates the direction (source and target) of the realized information flow conveying the information item, and its name is displayed close to that triangle. In the example Figure 17.7, two associations are realizing information flows. The black triangle indicates that an information flow is realized, and the information item name is displayed close to the triangle.

The example Figure 17.6 shows the case of information items represented on connectors. When several information items having the same direction are represented, only one triangle is shown, and the list of information item names, separated by a comma is presented.

The name of the information item can be prefixed by the names of the container elements, such as a container information flow, or a container package or classifier, separated by a colon.

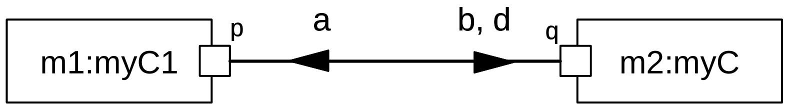


Figure 17.6 - Information item attached to connectors

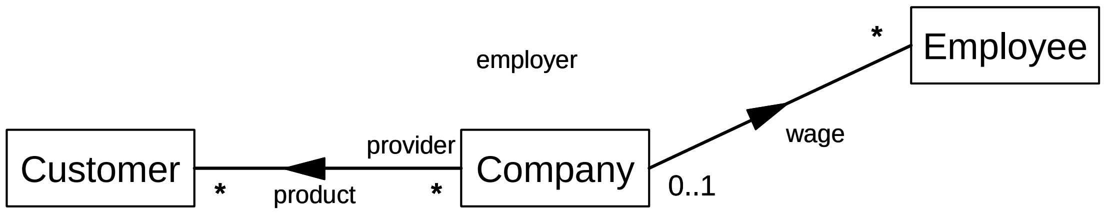


Figure 17.7 - Information Items attached to associations

17.3 Models

The contents of the Models package is shown in Figure 17.8. The Models package is one of the packages of the AuxiliaryConstructs package.

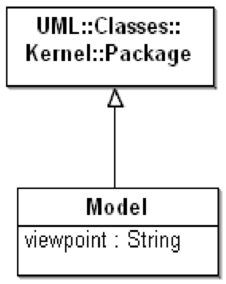


Figure 17.8 - The contents of the Models package

17.3.1 Model (from Models)

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.

Generalizations

- "Package (from Kernel)" on page 107

Description

The Model construct is defined as a Package. It contains a (hierarchical) set of elements that together describe the physical system being modeled. A Model may also contain a set of elements that represents the environment of the system, typically Actors, together with their interrelationships, such as Associations and Dependencies.

Attributes

- viewpoint : String [0..1]
  The name of the viewpoint that is expressed by a model. (This name may refer to a profile definition.)

Associations

No additional associations

Constraints

No additional constraints

Semantics

A model is a description of a physical system with a certain purpose, such as to describe logical or behavioral aspects of the physical system to a certain category of readers.

Thus, a model is an abstraction of a physical system. It specifies the physical system from a certain vantage point (or viewpoint) for a certain category of stakeholders (e.g., designers, users, or customers of the system) and at a certain level of abstraction, both given by the purpose of the model. A model is complete in the sense that it covers the whole physical system, although only those aspects relevant to its purpose (i.e., within the given level of abstraction and vantage point) are represented in the model. Furthermore, it describes the physical system only once (i.e., there is no overlapping; no part of the physical system is captured more than once in a model).

A model owns or imports all the elements needed to represent a physical system completely according to the purpose of this particular model. The elements are organized into a containment hierarchy where the top-most package or subsystem represents the boundary of the physical system. It is possible to have more than one containment hierarchy within a model (i.e., the model contains a set of top-most packages/subsystems each being the root of a containment hierarchy). In this case there is no single package/subsystem that represents the physical system boundary.

The 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 physical system, they reside outside the package/subsystem hierarchy. They may be collected in a separate package, or owned directly by the model. These elements and the elements representing the physical system may be associated with each other.

Different models can be defined for the same physical system, where each model represents a view of the physical system defined by its purpose and abstraction level. Typically different models are complementary and defined from the perspectives (viewpoints) of different system stakeholders. When models are nested, the container model represents the comprehensive view of the physical system given by the different views defined by the contained models.

Models can have refinement or mapping dependencies between them. These are typically decomposed into dependencies between the elements contained in the models. Relationships between elements in different models have no semantic impact on the contents of the models because of the self-containment of models. However, they are useful for tracing refinements and for keeping track of requirements between models.

Notation

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 contents of the model are shown within the large rectangle, the triangle may be drawn to the right of the model name in the tab.

Presentation Options

A model is notated as a package, using the ordinary package symbol with the keyword «model» placed above the name of the model.

Examples

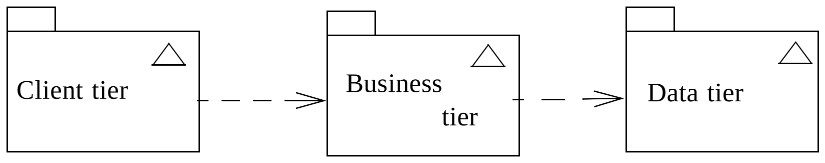


Figure 17.9 - Three models representing parts of a system

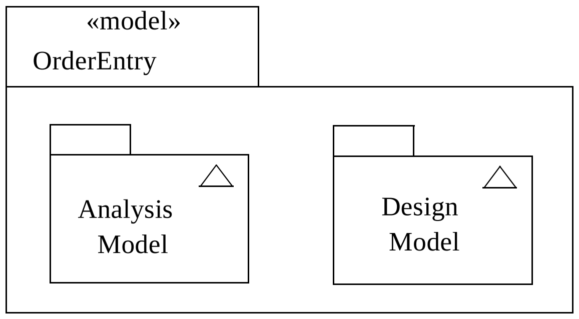


Figure 17.10 - Two views of one and the same physical system collected in a container model

17.4 PrimitiveTypes

A number of primitive types have been defined for use in the specification of the UML metamodel. These include primitive types such as Integer, Boolean, and String. These types are reused by both MOF and UML, and may potentially be reused also in user models. Tool vendors, however, typically provide their own libraries of data types to be used when modeling with UML.

The contents of the PrimitiveTypes package is shown in Figure 17.11. The PrimitiveTypes package is one of the packages of the AuxiliaryConstructs package.

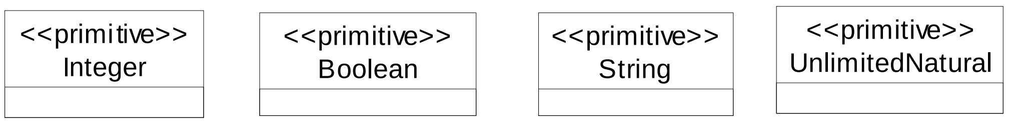


Figure 17.11 - The contents of the PrimitiveTypes package

17.4.1 Boolean (from PrimitiveTypes)

A Boolean type is used for logical expression, consisting of the predefined values true and false.

Generalizations

None

Description

Boolean is an instance of PrimitiveType. In the metamodel, Boolean defines an enumeration that denotes a logical condition. Its enumeration literals are:

- true - The Boolean condition is satisfied.
- false - The Boolean condition is not satisfied.

It is used for Boolean attribute and Boolean expressions in the metamodel, such as OCL expression.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Boolean is an instance of PrimitiveType.

Notation

Boolean will appear as the type of attributes in the metamodel. Boolean instances will be values associated to slots, and can have literally the following values: true or false.

Examples

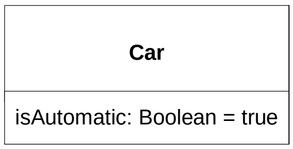


Figure 17.12 - An example of a Boolean attribute

17.4.2 Integer (from PrimitiveTypes)

An integer is a primitive type representing integer values.

Generalizations

None

Description

An instance of Integer is an element in the (infinite) set of integers (…-2, -1, 0, 1, 2…). It is used for integer attributes and integer expressions in the metamodel.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Integer is an instance of PrimitiveType.

Notation

Integer will appear as the type of attributes in the metamodel. Integer instances will be values associated to slots such as 1, -5, 2, 34, 26524, etc.

Examples

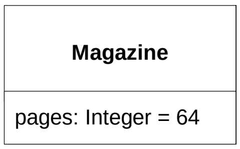


Figure 17.13 - An example of an integer attribute

17.4.3 String (from PrimitiveTypes)

A string is a sequence of characters in some suitable character set used to display information about the model. Character sets may include non-Roman alphabets and characters.

Generalizations

None

Description

An instance of String defines a piece of text. The semantics of the string itself depends on its purpose, it can be a comment, computational language expression, OCL expression, etc. It is used for String attributes and String expressions in the metamodel.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

String is an instance of PrimitiveType.

Notation

String appears as the type of attributes in the metamodel. String instances are values associated to slots. The value is a sequence of characters surrounded by double quotes ("). It is assumed that the underlying character set is sufficient for representing multibyte characters in various human languages; in particular, the traditional 8-bit ASCII character set is insufficient. It is assumed that tools and computers manipulate and store strings correctly, including escape conventions for special characters, and this document will assume that arbitrary strings can be used.

A string is displayed as a text string graphic. Normal printable characters should be displayed directly. The display of nonprintable characters is unspecified and platform-dependent.

Examples

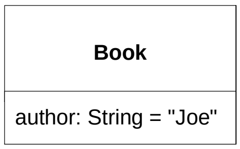


Figure 17.14 - An example of a string attribute

17.4.4 UnlimitedNatural (from PrimitiveTypes)

An unlimited natural is a primitive type representing unlimited natural values.

Generalizations

None

Description

An instance of UnlimitedNatural is an element in the (infinite) set of naturals (0, 1, 2…). The value of infinity is shown using an asterisk (‘*’).

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

UnlimitedNatural is an instance of PrimitiveType.

Notation

UnlimitedNatural will appear as the type of upper bounds of multiplicities in the metamodel. UnlimitedNatural instances will be values associated to slots such as 1, 5, 398475, etc. The value infinity may be shown using an asterisk (‘*’).

Examples

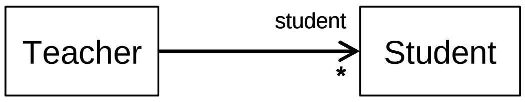


Figure 17.15 - An example of an unlimited natural

17.5 Templates

The Templates package specifies how both Classifiers, Packages, and Operations can be parameterized with Classifier, ValueSpecification, and Feature (Property and Operation) template parameters. The package introduces mechanisms for defining templates, template parameters, and bound elements in general, and the specialization of these for classifiers and packages.

Classifier, package, and operation templates were covered in 1.x in the sense that any model element could be templateable. This new metamodel restricts the templateable elements to those for which it is meaningful to have template parameters.

Templates

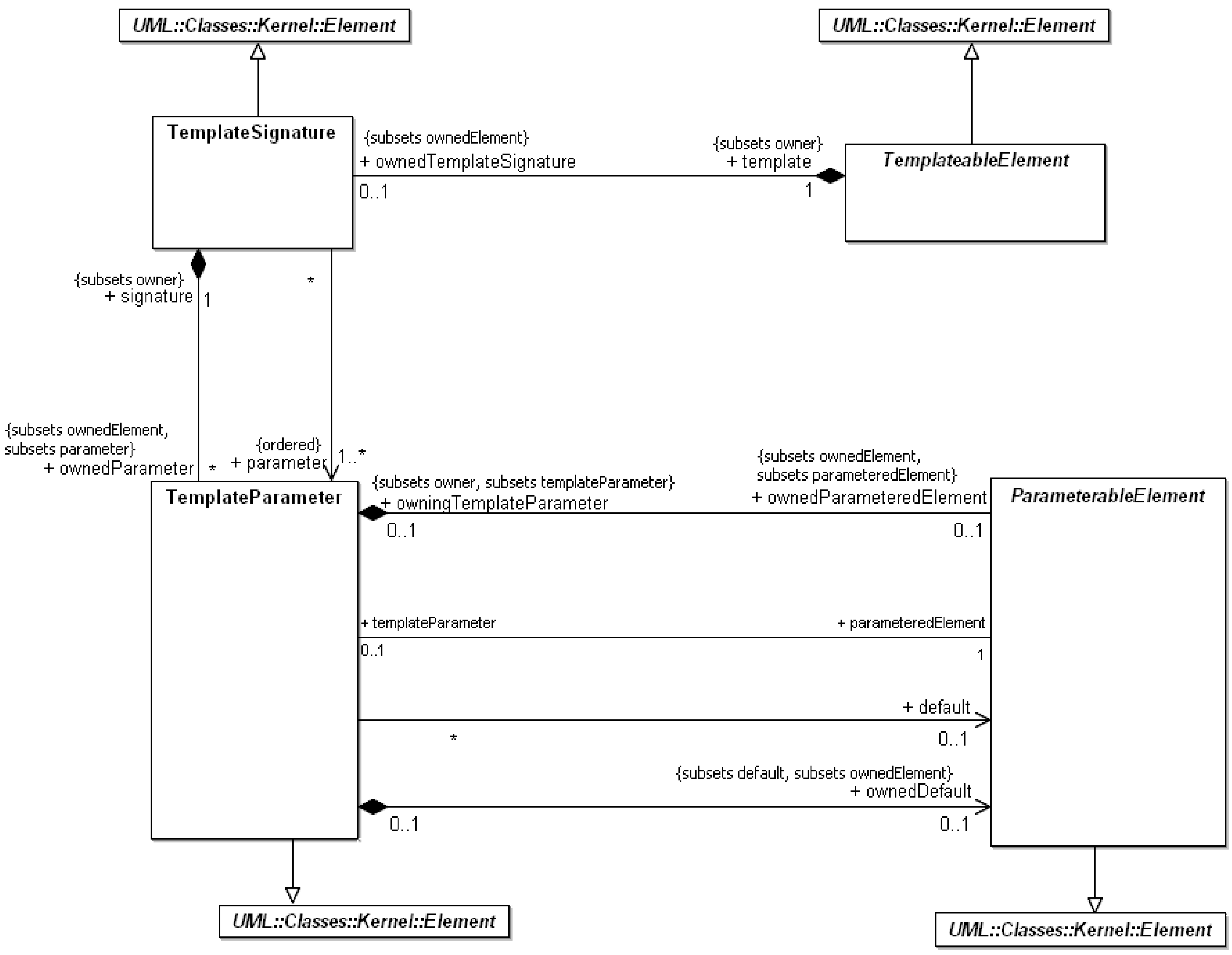


Figure 17.16 - Templates

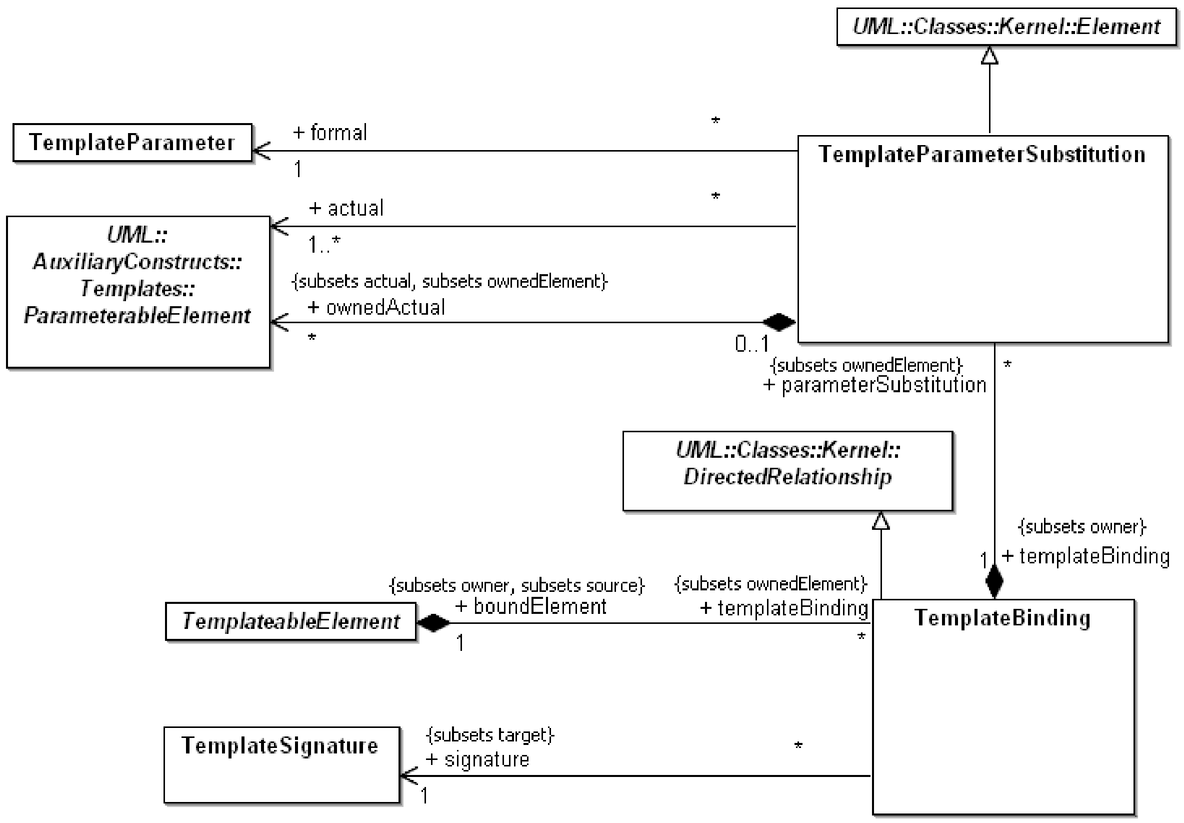


Figure 17.17 - Template parameters

17.5.1 ParameterableElement (from Templates)

A parameterable element is an element that can be exposed as a formal template parameter for a template, or specified as an actual parameter in a binding of a template.

Generalizations

- "Element (from Kernel)" on page 64

Description

A ParameterableElement can be referenced by a TemplateParameter when defining a formal template parameter for a template. A ParameterableElement can be referenced by a TemplateParameterSubstitution when used as an actual parameter in a binding of a template. ParameterableElement is an abstract metaclass.

Attributes

No additional attributes

Associations

- owningTemplateParameter : TemplateParameter[0..1]
  The formal template parameter that owns this element. Subsets *Element::owner* and *ParameterableElement::templateParameter.*
- templateParameter : TemplateParameter [0..1]
  The template parameter that exposes this element as a formal parameter.

Constraints

No additional constraints

Additional Operations

[1] 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. Subclasses should override this operation to specify different compatibility constraints.

```
ParameterableElement::isCompatibleWith(p : ParameterableElement) : Boolean;
isCompatibleWith = p->oclIsKindOf(self.oclType)
```

[2] The query isTemplateParameter() determines if this parameterable element is exposed as a formal template parameter.

```
ParameterableElement::isTemplateParameter() : Boolean;
isTemplateParameter = templateParameter->notEmpty()
```

Semantics

A ParameterableElement may be part of the definition of a template parameter. The ParameterableElement is used to constrain the actual arguments that may be specified for this template parameter in a binding of the template.

A ParameterableElement exposed as a template parameter can be used in the template as any other element of this kind defined in the namespace of the template. For example, a classifier template parameter can be used as the type of typed elements. In an element bound to the template, any use of the template parameter will be substituted by the use of the actual parameter.

If a ParameterableElement is exposed as a template parameter, then the parameterable element is only meaningful within the template (it may not be used in other parts of the model).

Semantic Variation Points

The enforcement of template parameter constraints is a semantic variation point:

- If template parameter constraints apply, then within the template element a template parameter can only be used according to its constraint. For example, an operation template parameter can only be called with actual parameters matching the constraint in terms of the signature constraint of the operation template parameter. Applying constraints will imply that a bound element is well-formed if the template element is well-formed and if actual parameters comply with the formal parameter constraints.
- If template parameter constraints do not apply, then within the template element a template parameter can be used without being constrained. For example, an operation template parameter will have no signature in terms of parameters and it can be called with arbitrary actual parameters. Not applying constraints provides more flexibility, but some actual template parameters may not yield a well-formed bound element.

Notation

See TemplateParameter for a description of the notation for exposing a ParameterableElement as a formal parameter of a template.

See TemplateBinding for a description of the notation for using a ParameterableElement as an actual parameter in a binding of a template.

Within these notations, the parameterable element is typically shown as the name of the parametered element (if that element is a named element).

Examples

See TemplateParameter.

17.5.2 TemplateableElement (from Templates)

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

Generalizations

- "Element (from Kernel)" on page 64

Description

TemplateableElement may contain a template signature that specifies the formal template parameters. A TemplateableElement that contains a template signature is often referred to as a template.

TemplateableElement may contain bindings to templates that describe how the templateable element is constructed by replacing the formal template parameters with actual parameters. A TemplateableElement containing bindings is often referred to as a bound element.

Attributes

No additional attributes

Associations

- ownedTemplateSignature : TemplateSignature[0..1]
  The optional template signature specifying the formal template parameters. {Subsets *Element::ownedElement*}
- templateBinding : TemplateBinding[*]
  The optional bindings from this element to templates. {Subsets *Element::ownedElement*}

Constraints

No additional constraints

Additional Operations

[1] The query parameterableElements() returns the set of elements that may be used as the parametered elements for a template parameter of this templateable element. By default, this set includes all the owned elements. Subclasses may override this operation if they choose to restrict the set of parameterable elements.

```
TemplateableElement::parameterableElements() : Set(ParameterableElement);
parameterableElements = allOwnedElements->select(oclIsKindOf(ParameterableElement))
```

[2] The query isTemplate() returns whether this templateable element is actually a template.

```
TemplateableElement::isTemplate() : Boolean;
isTemplate = ownedSignature->notEmpty()
```

Semantics

A TemplateableElement that has a template signature is a specification of a template. A template is a parameterized element that can be used to generate other model elements using TemplateBinding relationships. The template parameters for the template signature specify the formal parameters that will be substituted by actual parameters (or the default) in a binding.

A template parameter is defined in the namespace of the template, but the template parameter represents a model element that is defined in the context of the binding.

A templateable element can be bound to other templates. This is represented by the bound element having bindings to the template signatures of the target templates. In a canonical model a bound element does not explicitly contain the model elements implied by expanding the templates it binds to, since those expansions are regarded as derived. The semantics and well-formedness rules for the bound element must be evaluated as if the bindings were expanded with the substitutions of actual elements for formal parameters.

The semantics of a binding relationship is equivalent to the model elements that would result from copying the contents of the template into the bound element, replacing any elements exposed as a template parameter with the corresponding element(s) specified as actual parameters in this binding.

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 specific 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 found in 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 templateable element.

A templateable element may contain both a template signature and bindings. Thus a templateable element may be both a template and a bound element.

A template cannot be used in the same manner as a non-template element of the same kind. The template element can only be used to generate bound elements (e.g., a template class cannot be used as the type of a typed element) or as part of the specification of another template (e.g., a template class may specialize another template class).

A bound (non-template) 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 bound class may be used as the type of a typed element.

Notation

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

The formal template parameter list may be shown as a comma-separated list, or it may be one formal template parameter per line. See TemplateParameter for the general syntax of each template parameter.

A bound element has the same graphical notation as other elements of that kind. Each binding is shown using the notation described under TemplateBinding.

Presentation Options

An alternative presentation for the bindings for a bound element is to include the binding information within the notation for the bound element. Typically the name compartment would be extended to contain a string with the following syntax:

<element-name> ‘:’ *<binding-expression> [‘,’ <binding-expression>]**

<binding-expression> ::= <template-element-name> ‘<‘ *<template-parameter-substitution>*

[‘,’<template-parameter-substitution]* ‘>’

and *<template-parameter-substitution>* is defined in TemplateBinding (from Templates).

Examples

For examples of templates, the reader is referred to those sub clauses that deal with specializations of TemplateableElement, in particular “ClassifierTemplates” on page 631 and “PackageTemplates” on page 639.

17.5.3 TemplateBinding (from Templates)

A template binding represents a relationship between a templateable element and a template. A template binding specifies the substitutions of actual parameters for the formal parameters of the template.

Generalizations

- “DirectedRelationship (from Kernel)” on page 63

Description

TemplateBinding is a directed relationship from a bound templateable element to the template signature of the target template. A TemplateBinding owns a set of template parameter substitutions.

Attributes

No additional attributes

Associations

- parameterSubstitution : TemplateParameterSubstitution[*]
  The parameter substitutions owned by this template binding. Subsets *Element::ownedElement*.
- boundElement : TemplateableElement[1]
  The element that is bound by this binding. Subsets *DirectedRelationship::source* and *Element::owner.*
- signature: TemplateSignature[1]
  The template signature for the template that is the target of the binding. Subsets *DirectedRelationship::target*.

Constraints

[1] Each parameter substitution must refer to a formal template parameter of the target template signature.

```
parameterSubstitution->forAll(b | template.parameter->includes(b.formal))
```

[2] A binding contains at most one parameter substitution for each formal template parameter of the target template signature.

template.parameter->forAll(p | parameterSubstitution->select(b | b.formal = p)->size() <= 1)

Semantics

The presence of a TemplateBinding relationship implies the same semantics as if the contents of the template owning the target template signature were copied into the bound element, substituting any elements exposed as formal template parameters by the corresponding elements specified as actual parameters in this binding. If no actual parameter is specified in this binding for a formal parameter, then the default element for that formal template parameter (if specified) is used.

Semantic Variation Points

It is a semantic variation point

- if all formal template parameters must be bound as part of a binding (complete binding), or
- if a subset of the formal template parameters may be bound in a binding (partial binding).

In case of complete binding, the bound element may have its own formal template parameters, and these template parameters can be provided as actual parameters of the binding. In case of partial binding, the unbound formal template parameters are formal template parameters of the bound element.

Notation

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 is generally displayed as a comma-separated list of template parameter substitutions:

<template-param-substitition> ::= <template-param-name> '->' <actual-template-parameter>

where the syntax of <*template-param-name*> depends on the kind of ParameteredElement for this template parameter substitution and the kind of <*actual-template-parameter*> depends upon the kind of element. See "ParameterableElement (from Templates)" on page 623 (and its subclasses).

Examples

For examples of templates, the reader is referred to those sub clauses that deal with specializations of TemplateableElement, in particular "ClassifierTemplates" on page 631 and "PackageTemplates" on page 639.

17.5.4 TemplateParameter (from Templates)

A template parameter exposes a parameterable element as a formal template parameter of a template.

Generalizations

- "Element (from Kernel)" on page 64

Description

TemplateParameter references a ParameterableElement that is exposed as a formal template parameter in the containing template.

Attributes

No additional attributes

Associations

- default : ParameterableElement[0..1]
  The element that is the default for this formal template parameter.
- ownedDefault : ParameterableElement[0..1]
  The element that is owned by this template parameter for the purpose of providing a default. Subsets default and Element::ownedElement.
- ownedParameteredElement : ParameterableElement[0..1]
  The element that is owned by this template parameter. Subsets parameteredElement and Element::ownedElement.
- parameteredElement : ParameterableElement[1]
  The element exposed by this template parameter.
- signature : TemplateSignature[1]
  The template signature that owns this template parameter. Subsets Element::owner.

Constraints

[1] The default must be compatible with the formal template parameter.

```
default->notEmpty() implies default->isCompatibleWith(parameteredElement)
```

Semantics

A TemplateParameter references a ParameterableElement that is exposed as a formal template parameter in the containing template. This parameterable element is meaningful only within the template, or other templates that may have access to its internals (e.g., if the template supports specialization). The exposed parameterable element may not be used in other parts of the model. A TemplateParameter may own the exposed ParameterableElement in situations where that element is only referenced from within the template.

Each exposed element constrains the elements that may be substituted as actual parameters in a binding.

A TemplateParameter may reference a ParameterableElement as the default for this formal parameter in any binding that does not provide an explicit substitution. The TemplateParameter may own this default ParameterableElement in situations where the exposed ParameterableElement is not owned by the TemplateParameter.

Notation

The general notation for a template parameter is a string displayed within the template parameter 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>* depends on the kind of parameteredElement for this template parameter substitution and of *<default>* depends upon the kind of element. See “ParameterableElement (from Templates)” on page 623 (and its subclasses).

Examples

See TemplateableElement.

17.5.5 TemplateParameterSubstitution (from Templates)

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

Generalizations

- "Element (from Kernel)" on page 64

Description

TemplateParameterSubstitution associates an actual parameter with a formal template parameter within the context of a TemplateBinding.

Attributes

No additional attributes

Associations

- actual : ParameterableElement[1]
  The element that is the actual parameter for this substitution.
- templateBinding : TemplateBinding[1]
  The template binding that owns this substitution. Subsets Element::owner.
- formal : TemplateParameter[1]
  The formal template parameter that is associated with this substitution.
- ownedActual : ParameterableElement[0..1]
  The actual parameter that is owned by this substitution. Subsets Element::ownedElement and actual.

Constraints

[1] The actual parameter must be compatible with the formal template parameter (e.g., the actual parameter for a class template parameter must be a class).

```
actual->forAll(a | a.isCompatibleWith(formal.parameteredElement))
```

Semantics

A TemplateParameterSubstitution specifies the actual parameter to be substituted for a formal template parameter within the context of a template binding.

Notation

See TemplateBinding.

Examples

See TemplateBinding.

17.5.6 TemplateSignature (from Templates)

A template signature bundles the set of formal template parameters for a templated element.

Generalizations

- “Element (from Kernel)” on page 64

Description

A TemplateSignature is owned by a TemplateableElement and has one or more TemplateParameters that define the signature for binding this template.

Attributes

No additional attributes

Associations

- ownedParameter : TemplateParameter[*]
  The formal template parameters that are owned by this template signature. Subsets parameter and Element::ownedElement.
- parameter : TemplateParameter[1..*]
  The ordered set of allformal template parameters for this template signature.
- template : TemplateableElement[1]
  The element that owns this template signature. Subsets Element::owner.

Constraints

[1] Parameters must own the elements they parameter or those elements must be owned by the element being templated.
templatedElement.ownedElement->includesAll(parameter.parameteredElement - parameter.ownedParameteredElement)

Semantics

A TemplateSignature specifies the set of formal template parameters for the associated templated element. The formal template parameters specify the elements that may be substituted in a binding of the template.

There are constraints on what may be parametered by a template parameter. Either the signature owns the parametered element, or the element is owned, directly or indirectly, by the template subclasses of TemplateSignature can add additional rules constraining what a parameter can reference in the context of a particular kind of template.

Notation

See TemplateableElement for a description of how the template parameters are shown as part of the notation for the template.

Examples

See TemplateableElement.

ClassifierTemplates

The Classifier templates diagram (Figure 17.18 on page 632) specifies the abstract mechanisms that support defining classifier templates, bound classifiers, and classifier template parameters. Specific subclasses of Classifier must also specialize one or more of the abstract metaclasses defined in this diagram in order to expose these capabilities in a concrete manner.

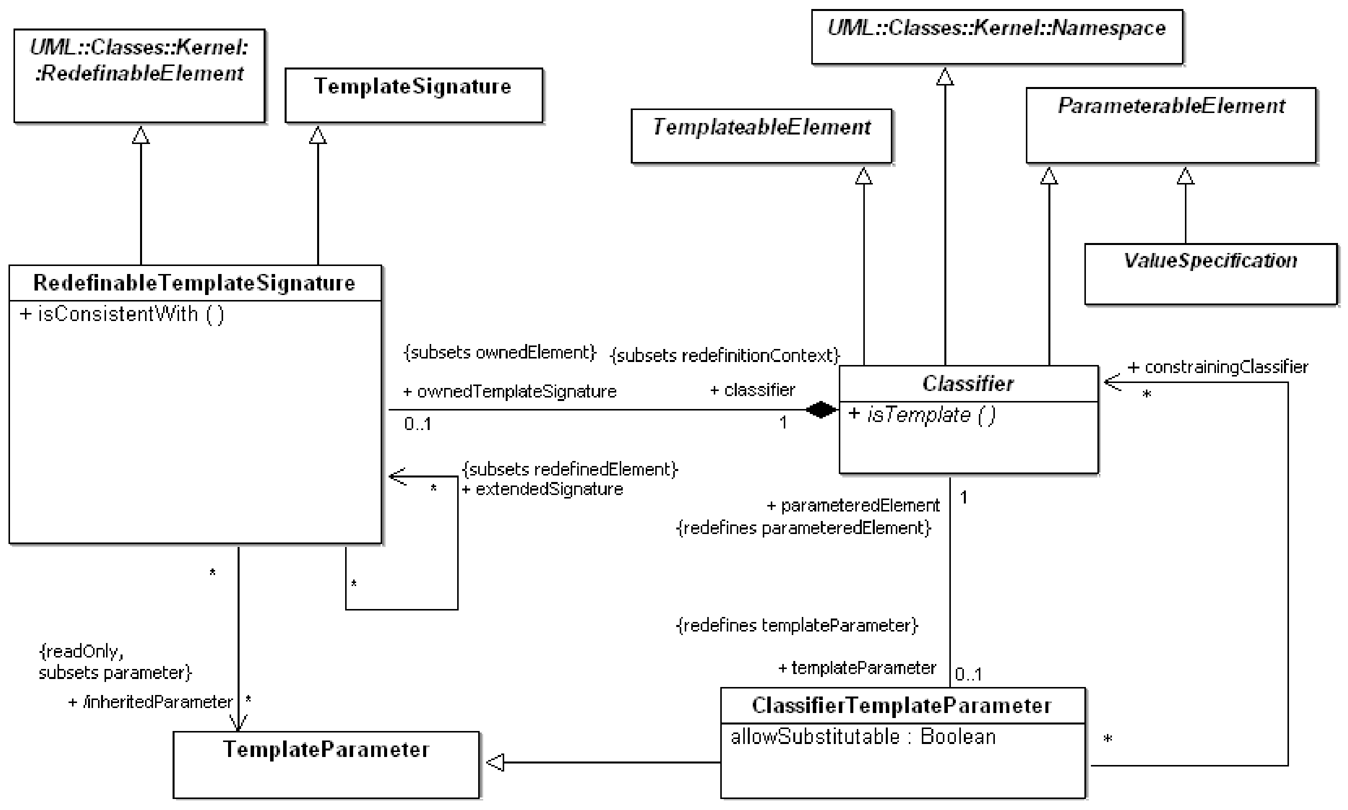


Figure 17.18 - Classifier templates

17.5.7 Classifier (from Templates)

Classifier is defined to be a kind of templateable element so that a classifier can be parameterized. It is also defined to be a kind of parameterable element so that a classifier can be a formal template parameter.

Generalizations

- "ParameterableElement (from Templates)" on page 623
- "TemplateableElement (from Templates)" on page 625
- "Classifier (from Kernel, Dependencies, PowerTypes)" on page 52 *(merge increment)*

Description

Classifier specializes Kernel::Classifier, TemplateableElement, and ParameterableElement to specify that a classifier can be parameterized, be exposed as a formal template parameter, and can be specified as an actual parameter in a binding of a template.

A classifier with template parameters is often called a template classifier, while a classifier with a binding is often called a bound classifier.

By virtue of Classifier being defined here, all subclasses of Classifier (such as Class, Collaboration, Component, Datatype, Interface, Signal, and UseCases) can be parameterized, bound, and used as template parameters. The same holds for Behavior as a subclass of Class, and thereby all subclasses of Behavior (such as Activity, Interaction, StateMachine).

Attributes

No additional attributes

Associations

- ownedTemplateSignature : RedefinableTemplateSignature[0..1]
  The optional template signature specifying the formal template parameters. Subsets *Element::ownedElement*.
- templateParameter : ParameterableElement [0..1]
  The template parameter that exposes this element as a formal parameter. Redefines *ParameterableElement::templateParameter*

Constraints

No additional constraints

Additional Operations

[1] The query isTemplate() returns whether this templateable element is actually a template.

```
Classifier::isTemplate() : Boolean;
isTemplate = oclAsType(TemplatableElement).isTemplate() or general->exists(g | g.isTemplate())
```

Semantics

Classifier in general

Classifier provides the abstract mechanism that can be specialized to support subclass of Classifiers to be templates, exposing subclasses of Classifier as formal template parameters, and as actual parameters in a binding of a template.

Classifier as a kind of templateable element provides the abstract mechanism that can be specialized by subclasses of Classifier to support being defined as a template, or being bound to a template.

A bound classifier may have contents in addition to those of the template classifier. In this case the semantics are equivalent to inserting an anonymous general classifier that contains the contents, and the bound classifier is defined to be a specialization this anonymous general classifier. This supports the use of elements within the bound classifier as actual parameters in a binding.

A bound classifier may have multiple bindings. In this case the semantics are equivalent to inserting an anonymous general bound classifier for each binding, and specializing all these bound classifiers by this (formerly) bound classifier.

The formal template parameters for a classifier include all the formal template parameters of all the templates it specializes. For this reason the classifier may reference elements that are exposed as template parameters of the specializes templates.

Collaboration

A Collaboration supports the ability to be defined as a template. A collaboration may be defined to be bound from template collaboration(s).

A collaboration template will typically have the types of its parts as class template parameters. Consider the Collaboration in Figure 9.11 on page 170. This Collaboration can be bound from a Collaboration template of the form found in Figure 17.23, by means of the binding described in Figure 17.24.

A bound Collaboration is not the same as a CollaborationUse; in fact, parameterized Collaborations (and binding) cannot express what CollaborationUses can. Consider the Sale Collaboration in Figure 9.13 on page 172. It is defined by means of two parts (Buyer and Seller) representing roles in this collaboration. The two CollaborationUses “wholesale” and “retail” in Figure 9.14 on page 173 cannot be defined as bound Collaborations.

A bound Collaboration is a Collaboration, while a CollaborationUse is not. A CollaborationUse is defined by means of RoleBindings, binding parts in a Collaboration (here Buyer and Seller) to parts in another classifier (here broker, producer, and consumer in BrokeredSale) with the semantics that the interaction described in Sale will occur between broker, producer, and consumer. Binding eventual Buyer and Seller part template parameters (of a Sale Collaboration template) to broker and producer would just provide that the parts broker and producer are visible within the bound Sale Collaboration. Even if Sale had two part template parameters Buyer and Seller, it could not use these for defining an internal structure as is the case in Figure 9.14 on page 173. Parameters, by their very nature, represent elements that are defined ‘outside’ a Collaboration template and can therefore not be parts of an internal structure of the Collaboration template.

Semantic Variation Points

If template parameter constraints apply, then the actual classifier is constrained as follows. If the classifier template parameter:

- has a generalization, then an actual classifier must have generalization with the same general classifier.
- has a substitution, then an actual classifier must have a substitution with the same contract.
- has neither a generalization nor a substitution, then an actual classifier can be any classifier.

If template parameter constraints do not apply, then an actual classifier can be any classifier.

Notation

See ClassifierTemplateParameter for a description of how a parameterable classifier is displayed as a formal template parameter.

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 an attribute, then <*classifier expression*> acts as the *type* of the attribute in the notation for an attribute:

[<visibility>] [‘/’] <name> [‘:’ <attr-type>] [‘[‘<multiplicity> ‘]’] [‘=’ <default>]
[‘{‘<attr-property> [‘,’<attr-property>] ‘}’]*

When a bound classifier is used directly as the type of a part, then <*classifier-expression*> acts as the *classname* of the part in the notation for a part:

(([<name>] ‘:’ <classname>) | <name>) [‘[‘<multiplicity>’]’]

Presentation Options

Collaboration extends the presentation option for bound elements described under TemplateableElement so that the binding information can be displayed in the internal structure compartment.

Examples

Class templates

As Classifier is an abstract class, the following example is an example of concrete subclass (Class) of Classifier being a template.

The example shows a class template (named FArray) with two formal template parameters. The first formal template parameter (named T) is an unconstrained class template parameter. The second formal template parameter (named k) is an integer expression template parameter that has a default of 10. There is also a bound class (named AddressList) that substitutes the Address for T and 3 for k.

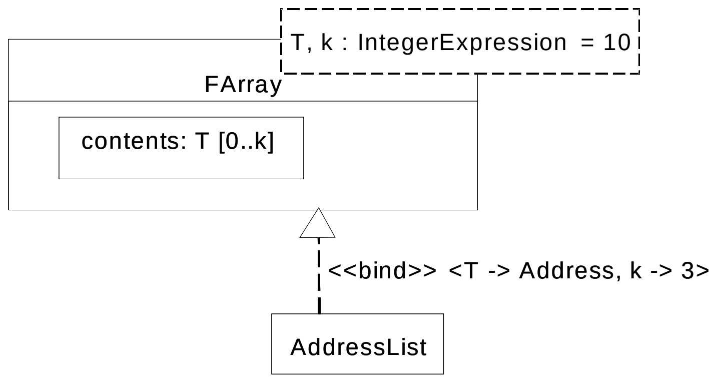

T, k : IntegerExpression = 10
FArray
contents: T [0..k]
<<bind>> <T -> Address, k -> 3>
AddressList

Figure 17.19 - Template Class and Bound Class

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

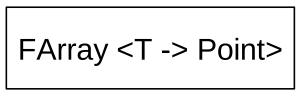


Figure 17.20 - Anonymous Bound Class

The following figure shows a template class (named Car) with two formal template parameters. The first formal template parameter (named CarEngine) is a class template parameter that is constrained to conform to the Engine class. The second formal template parameter (named n) is an integer expression template parameter.

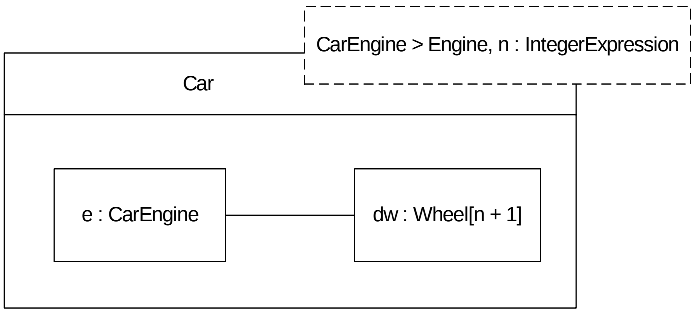


Figure 17.21 - Template Class with a constrained class parameter

The following figure shows a bound class (named DieselCar) that binds CarEngine to DieselEngine and n to 2.

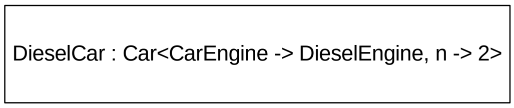


Figure 17.22 - Bound Class

Collaboration templates

The example below shows a collaboration template (named ObserverPattern) with two formal template parameters (named SubjectType and ObserverType). Both formal template parameters are unconstrained class template parameters.

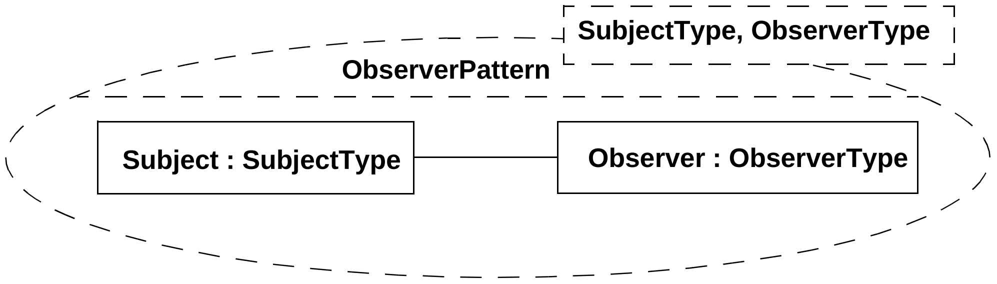


Figure 17.23 - Template Collaboration

The following figure shows a bound collaboration (named Observer) that substitutes CallQueue for SubjectType, and SlidingBarIcon for ObserverType.

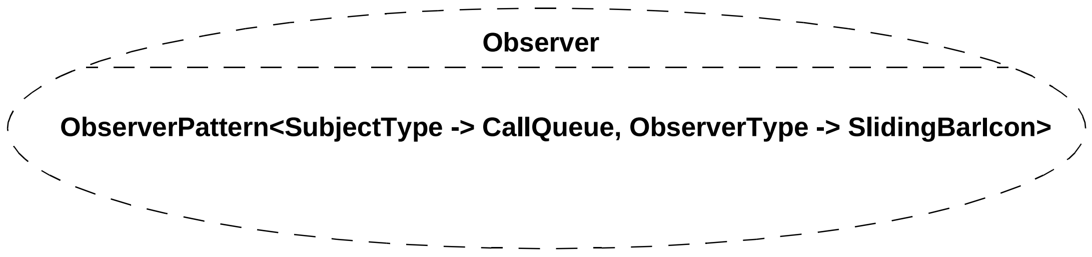

Observer
ObserverPattern<SubjectType -> CallQueue, ObserverType -> SlidingBarIcon>

Figure 17.24 - Bound Collaboration

17.5.8 ClassifierTemplateParameter (from Templates)

A classifier template parameter exposes a classifier as a formal template parameter.

Generalizations

- “TemplateParameter (from Templates)” on page 628

Description

ClassifierTemplateParameter is a template parameter where the parametered element is a Classifier in its capacity of being a kind of ParameterableElement.

Attributes

- allowSubstitutable : Boolean[1]
  Constrains the required relationship between an actual parameter and the parameteredElement for this formal parameter. Default is true.

Associations

- parameteredElement : Classifier[1]
  The parameterable classifier for this template parameter. Redefines TemplateParameter::parameteredElement.
- constrainingClassifier : Classifier [0..*]
  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 these classifier or one of their subclasses. If this property is empty, there are no constraints on the classifier that can be used as an argument.

Constraints

[1] If “allowSubstitutable” is true, then there must be a constrainingClassifier

allowSubstitutable **implies** constrainingClassifier->notEmpty()

Semantics

See Classifier for additional semantics related to the compatibility of actual and formal classifier parameters.

Notation

A classifier template parameter extends the notation for a template parameter 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 parametered element. It may be suppressed if it is ‘class.’

The *classifier-name* of *constraint* designates the type constraint of the parameter, which reflects the general classifier for the parametered element for this template parameter. The ‘contract’ option indicates that allowSubstitutable is true, meaning the actual parameter must be a classifier that may substitute for the classifier designated by the classifier-name. A classifier template parameter with a constraint but without ‘contract’ indicates that the actual classifier must be a specialization of the classifier designated by the *classifier-name*.

Examples

See Classifier.

17.5.9 RedefinableTemplateSignature (from Templates)

A redefinable template signature supports the addition of formal template parameters in a specialization of a template classifier.

Generalizations

- “TemplateSignature (from Templates)” on page 630
- “RedefinableElement (from Kernel)” on page 130

Description

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

Attributes

No additional attributes

Associations

- classifier : Classifier[1]
  The classifier that owns this template signature. Subsets RedefinableElement::redefinitionContext and Template::templatedElement.
- / inheritedParameter : TemplateParameter[*]
  The formal template parameters of the extendedSignature. Subsets Template::parameter.
- extendedSignature : RedefinableTemplateSignature[*]
  The template signature that is extended by this template signature. Subsets RedefinableElement::redefinedElement.

Constraints

[1] The inherited parameters are the parameters of the extended template signature.

```
inheritedParameter = if extendedSignature->isEmpty() then Set{} else extendedSignature.parameter endif
```

Additional Operations

[1] 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, since redefinition only adds new formal parameters.

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

pre: redefinee.isRedefinitionContextValid(self)

isConsistentWith = redefinee.oclIsKindOf(RedefineableTemplateSignature)

Semantics

A RedefinableTemplateSignature may extend an inherited template signature in order to specify additional formal template parameters that apply within the templateable classifier that owns this RedefinableTemplateSignature. All the formal template parameters of the extended signatures are included as formal template parameters of the extending signature, along with any parameters locally specified for the extending signature.

Notation

Notation as for redefinition in general.

PackageTemplates

The Package templates diagram supports the specification of template packages and package template parameters.

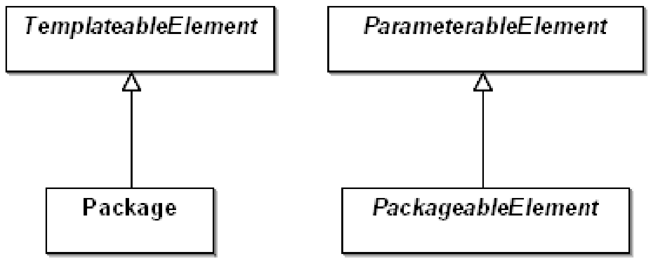


Figure 17.25 - Package templates

17.5.10 Package (from Templates)

Generalizations

- “TemplateableElement (from Templates)” on page 625
- “Package (from Kernel)” on page 107

Description

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.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

A Package supports the ability to be defined as a template and PackageableElements may, therefore, be parametered in a package template. A package template parameter may refer to any element owned or used by the package template, or templates nested within it, and so on recursively. That is, there are no special rules concerning how the parameters of a package template are defined.

A package may be defined to be bound to one or more template packages. The semantics for these bindings is as described in the general case, with the exception that we need to spell out the rules for merging the results of multiple bindings. In that case, the effect is the same as taking the intermediate results and merging them into the eventual result using package merge. This is illustrated by the example below.

Notation

See TemplateableElement for a description of the general notation that is defined to support these added capabilities.

Examples

The example below shows a package template (named ResourceAllocation) with three formal template parameters. All three formal template parameters (named Resource, ResourceKind, and System) are unconstrained class template parameters. There is also a bound package (named CarRental) that substitutes Car for Resource, CarSpec for ResourceKind, and CarRentalSystem for System.

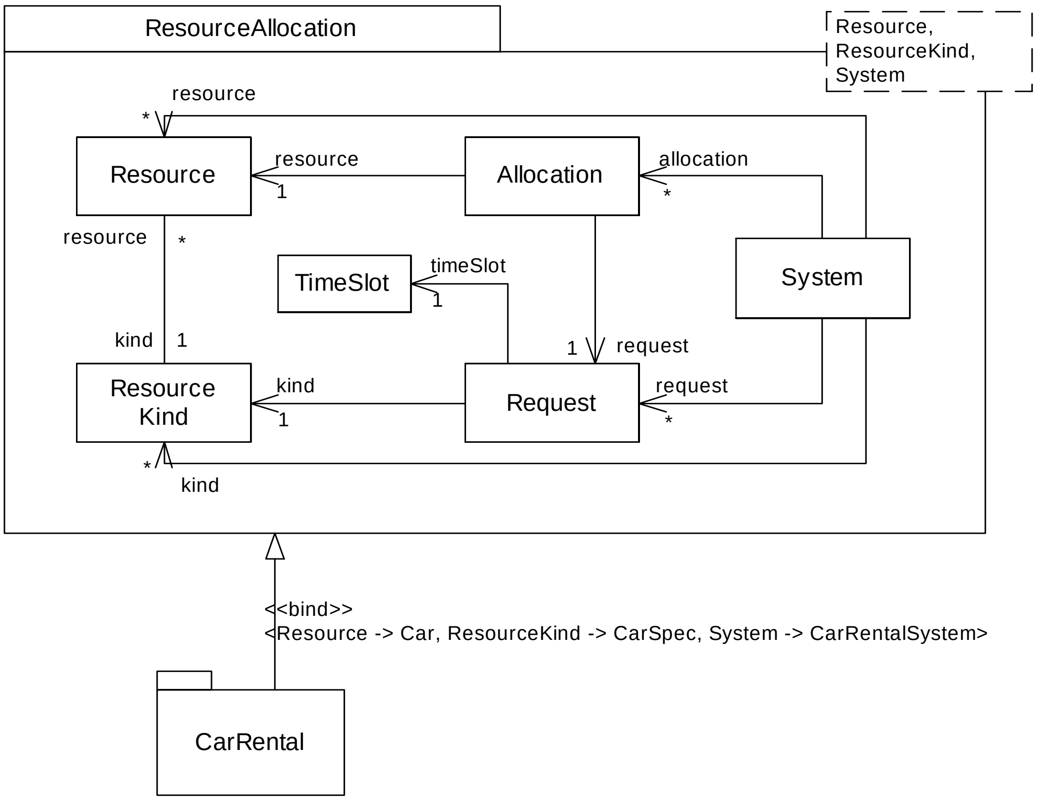


Figure 17.26 - Template Package and Bound Package

17.5.11 PackageableElement (from Templates)

Generalizations

- “ParameterableElement (from Templates)” on page 623
- “PackageableElement (from Kernel)” on page 109

Description

PackageableElements are extended to enable any such element to serve as a template parameter.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

NameExpressions

The NameExpressions diagram supports the use of string expressions to specify the name of a named element.

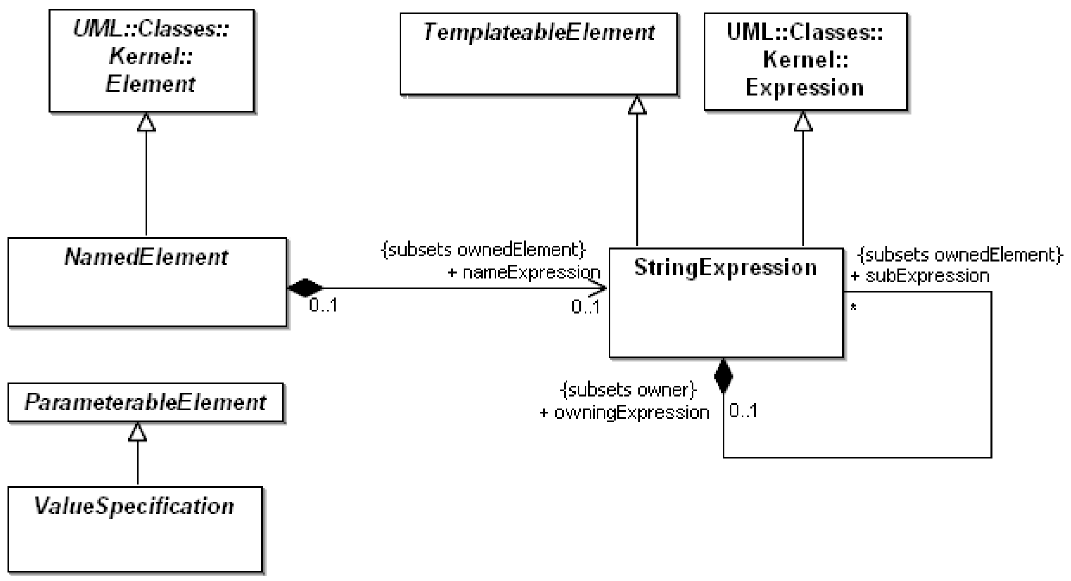

UML::Classes::Kernel::Element
TemplateableElement
UML::Classes::Kernel::Expression
NamedElement
{subsets ownedElement} + nameExpression
0..1
0..1
StringExpression
{subsets ownedElement} + subExpression
*
{subsets owner} + owningExpression
0..1
ParameterableElement
ValueSpecification

Figure 17.27 - Name expression

17.5.12 NamedElement (from Templates)

A named element is extended to support using a string expression to specify its name. This allows names of model elements to involve template parameters. The actual name is evaluated from the string expression only when it is sensible to do so (e.g., when a template is bound).

Generalizations

- "NamedElement (from Kernel, Dependencies)" on page 98 *(merge increment)*

Description

NamedElement specializes Kernel::NamedElement and adds a composition association to Expression.

Attributes

No additional attributes

Associations

- nameExpression : StringExpression [0..1]
  The string expression used to define the name of this named element (Subsets *Element::ownedElement}*

Constraints

No additional constraints

Semantics

A NamedElement may, in addition to a name, be associated with a string expression. This expression is used to calculate the name in the special case when it is a string literal. This allows string expressions, whose sub expressions may be parametered elements, to be associated with named elements in a template. When a template is bound, the sub expressions are substituted with the actuals substituted for the template parameters. In many cases, the resultant expression will be a string literal (if we assume that a concatenation of string literals is itself a string literal), in which case this will provide the name of the named element.

A NamedElement may have both a name and a name expression associated with it. In which case, the name can be used as an alias for the named element, which will surface, for example, in an OCL string. This avoids the need to use string expressions in surface notation, which is often cumbersome, although it doesn't preclude it.

Notation

The expression associated with a named element can be shown in two ways, depending on whether an alias is required or not. Both notations are illustrated in Figure 17.28.

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

In both cases the string expression 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 the sub clause on Expressions. In the context of templates, sub expressions of a string expression (usually string literals) that are parametered in the template are shown between angle brackets (see sub clause on ValueSpecificationTemplateParameters).

Examples

The figure shows a modified version of the ResourceAllocation package template where the first two formal template parameters have been changed to be string expression parameters. These formal template parameters 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 both classes Instructor, Qualification, and InstructorAllocation as well as classes Facility, FacilitySpecification, and FacilityAllocation. The associations are similarly replicated. Note that Request will have two attributes derived from the single <resourceKind> attribute (shown here by an arrow), namely qualification and facilitySpecification.

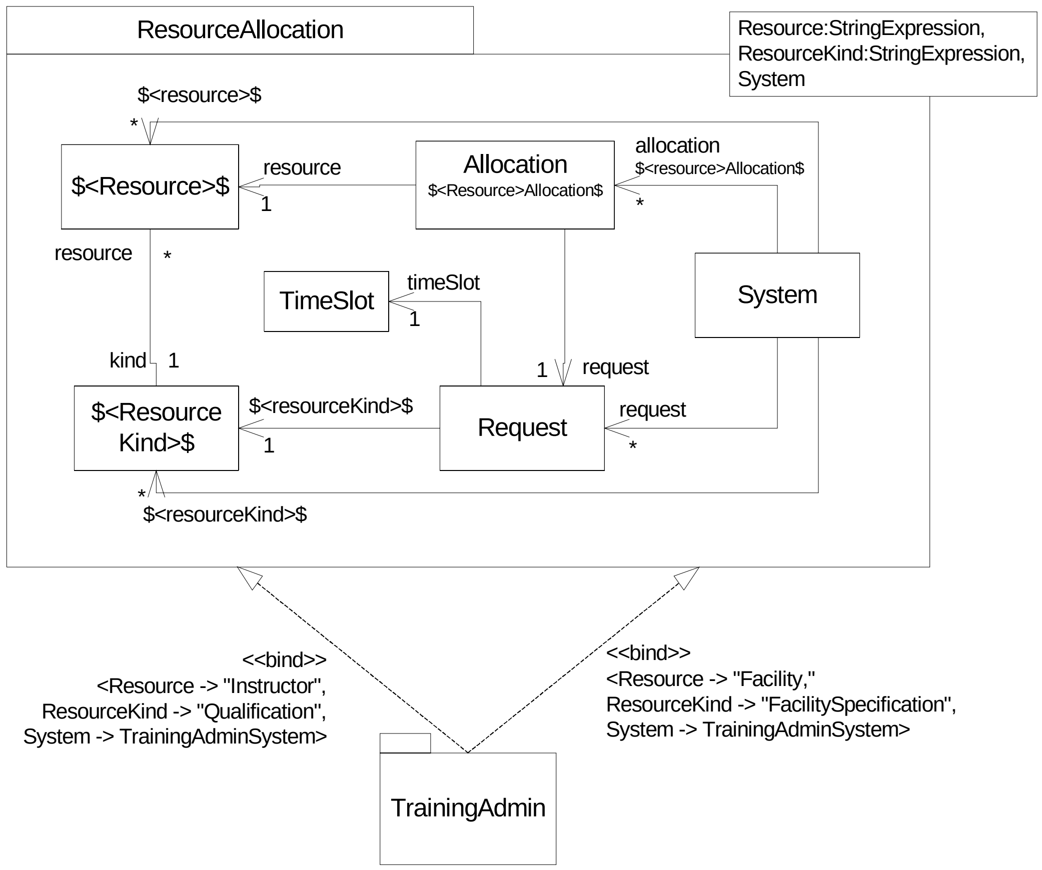


Figure 17.28 - Template Package with string parameters

17.5.13 StringExpression (from Templates)

An expression that specifies a string value that is derived by concatenating a set of sub string expressions, some of which might be template parameters.

Generalizations

- "TemplateableElement (from Templates)" on page 625
- "Expression (from Kernel)" on page 69

Description

StringExpression is a specialization of the general Expression metaclass that adds the ability to contain sub expressions and whose operands are exclusively LiteralStrings.

Attributes

No additional attributes

Associations

- subExpression : StringExpression[0..*]
  The StringExpressions that constitute this StringExpression. Subsets *Element::ownedElement*
- owningExpression : StringExpression [0..1]
  The StringExpression of which this expression is a substring. Subsets *Element::owner*

Constraints

[1] All the operands of a StringExpression must be LiteralStrings.

```
operand->forAll (op | op.oclIsKindOf (LiteralString))
```

[2] 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).

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

Additional Operations

[1] The query stringValue() returns the string that concatenates, in order, all the component string literals of all the subexpressions that are part of the StringExpression.

```
StringExpression::stringValue() : String;
if subExpression->notEmpty()
  then subExpression->iterate(se; stringValue = ‘’| stringValue.concat(se.stringValue()))
  else operand->iterate()(op; stringValue = ‘’ | stringValue.concat(op.value))
```

Semantics

A StringExpression is a composite expression whose elements are either nested StringExpressions or LiteralStrings. The string value of the expression is obtained by concatenating the respective string values of all the subexpressions in order. If the expression has no subexpressions, the string value is obtained by concatenating the LiteralStrings that are the operands of the StringExpression in order.

Notation

See the notation sub clause of “NamedElement (from Templates)” on page 642.

Examples

See Figure 17.28 on page 644.

Operation templates

The Operation templates diagram (Figure 17.29 on page 646) supports the specification of operation templates.

17.5.14 Operation (from Templates)

Generalizations

- “TemplateableElement (from Templates)” on page 625
- “Operation (from Kernel, Interfaces)” on page 103 *(merge increment)*

Description

Operation specializes TemplateableElement in order to support specification of template operations and bound operations.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

An Operation supports the ability to be defined as a template. An operation may be defined to be bound to template operation(s).

Notation

The template parameters and template parameter binding of a template operation are two lists in between the name of the operation and the parameters of the operation.

<visibility< <name> ‘<‘ *<template-parameter-list>* ‘>’ ‘<<‘ *<binding-expression-list>* ‘>>’‘(‘ <parameter> [‘,’<parameter>]* ‘)’ [‘:’ <property-string>]

OperationTemplateParameters

The operation template diagram supports the specification of operation template parameters.

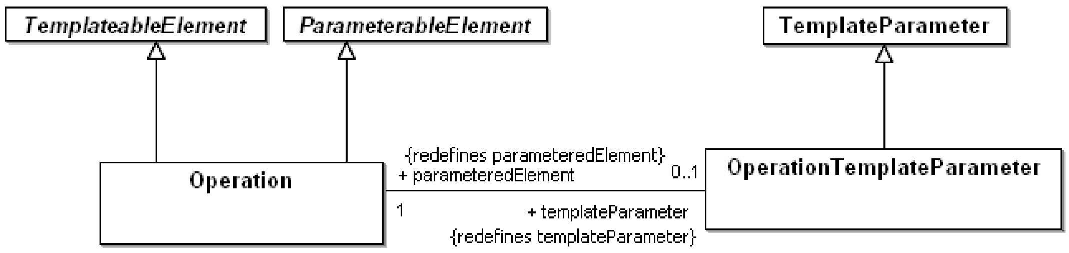


Figure 17.29 - Operation templates and parameters

17.5.15 Operation (from Templates)

Generalizations

- “ParameterableElement (from Templates)” on page 623
- “Operation (from Kernel, Interfaces)” on page 103 *(merge increment)*

Description

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.

Attributes

No additional attributes

Associations

- templateParameter : OperationTemplateParameter [0..1]
  The template parameter that exposes this element as a formal parameter. Redefines *ParameterableElement::templateParameter.*

Constraints

No additional constraints

Semantics

An Operation may be exposed by a template as a formal template parameter. Within a template classifier an operation template parameter may be used as any other operation defined in an enclosing namespace. Any references to the operation template parameter within the template will end up being a reference to the actual operation in the bound classifier. For example, a call to the operation template parameter will be a call to the actual operation.

Notation

See OperationTemplateParameter for a description of the general notation that is defined to support these added capabilities.

Within the notation for formal template parameters and template parameter bindings, an operation is shown as

<operation-name> '(' [*<operation-parameter> [',' <operation-parameter>]*]* ')'.

17.5.16 OperationTemplateParameter (from Templates)

An operation template parameter exposes an operation as a formal parameter for a template.

Generalizations

- "TemplateParameter (from Templates)" on page 628

Description

OperationTemplateParameter is a template parameter where the parametered element is an Operation.

Attributes

No additional attributes

Associations

- parameteredElement : Operation[1]
  The operation for this template parameter. Redefines TemplateParameter::parameteredElement.

Constraints

No additional constraints

Semantics

See Operation for additional semantics related to the compatibility of actual and formal operation parameters.

Notation

An operation template parameter extends the notation for a template parameter to include the parameters for the operation:

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

<parameter> ::= *<operation-name>* ‘(‘ *[<op-parameter> [‘,’ <op-parameter>]* ‘)’*

<default> ::= *<operation-name* ‘(‘ *[<op-parameter> [‘,’ <op-parameter>]* ‘)’*

ConnectableElement template parameters

The Connectable element template parameters package supports the specification of ConnectableElement template parameters.

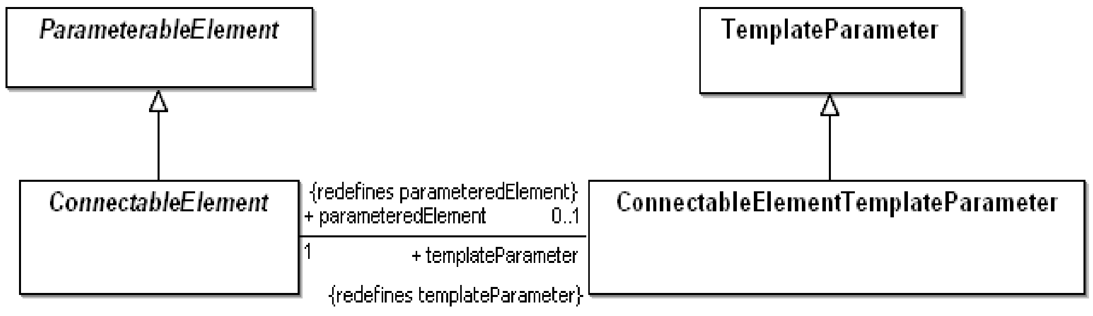


Figure 17.30 - Connectable element template parameters

17.5.17 ConnectableElement (from Templates)

A connectable element may be exposed as a connectable element template parameter.

Generalizations

- “ParameterableElement (from Templates)” on page 623.
- “ConnectableElement (from InternalStructures)” on page 174 *(merge increment)*.

Description

ConnectableElement is the connectable element of a ConnectableElementTemplateParameter.

Attributes

No additional attributes

Associations

- templateParameter : ConnectableElementTemplateParameter [0..1]
  The ConnectableElementTemplateParameter for this ConnectableElement parameter. Redefines *TemplateParameter::templateParameter.*

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

17.5.18 ConnectableElementTemplateParameter (from Templates)

A connectable element template parameter exposes a connectable element as a formal parameter for a template.

Generalizations

- "TemplateParameter (from Templates)" on page 628.

Description

ConnectableElementTemplateParameter is a template parameter where the parametered element is a ConnectableElement.

Attributes

No additional attributes

Associations

- parameteredElement : ConnectableElement[1]
  The ConnectableElement for this template parameter. Redefines TemplateParameter::parameteredElement.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

PropertyTemplateParameters

The Property template parameters diagram supports the specification of property template parameters.

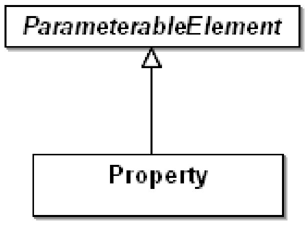


Figure 17.31 - The elements defined in the PropertyTemplates package

17.5.19 Property (from Templates)

Generalizations

- "ParameterableElement (from Templates)" on page 623
- "Property (from Kernel, AssociationClasses)" on page 122

Description

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.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] A binding of a property template parameter representing an attribute must be to an attribute

```
(isAttribute(self) and (templateParameterSubstitution->notEmpty() implies
    (templateParameterSubstitution->forAll(ts | isAttribute(ts.formal)))
```

Additional Operations

[1] 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 properties, the type must be conformant with the type of the specified parameterable element.

```
Property::isCompatibleWith(p : ParameterableElement) : Boolean;
isCompatibleWith = p->oclIsKindOf(self.oclType) and self.type.conformsTo(p.oclAsType(TypedElement).type)
```

Semantics

A Property may be exposed by a template as a formal template parameter. Within a template a property template parameter may be used as any other property defined in an enclosing namespace. Any references to the property template parameter within the template will end up being a reference to the actual property in the bound element.

Notation

See ParameterableElement for a description of the general notation that is defined to support these added capabilities.

ValueSpecificationTemplateParameters

The ValueSpecification template parameters diagram supports the specification of value specification template parameters.

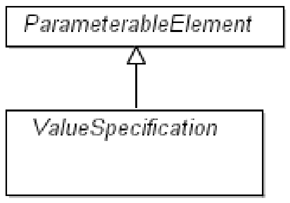


Figure 17.32 - ValueSpecification template parameters

17.5.20 ValueSpecification (from Templates)

Generalizations

- "ParameterableElement (from Templates)" on page 623
- "ValueSpecification (from Kernel)" on page 137 *(merge increment)*

Description

ValueSpecification specializes ParameterableElement to specify that a value specification can be exposed as a formal template parameter, and provided as an actual parameter in a binding of a template.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional attributes

Additional Operations

[1] 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 ValueSpecification, the type must be conformant with the type of the specified parameterable element.

```
Property::isCompatibleWith(p : ParameterableElement) : Boolean;
isCompatibleWith = p->oclIsKindOf(self.oclType) and self.type.conformsTo(p.oclAsType(TypedElement).type)
```

Semantics

The semantics is as in the general case. However, two aspects are worth commenting on. The first is to note that a value specification may be an expression with substructure (i.e., an instance of the Expression class), in which case a template parameter may expose a subexpression, not necessarily the whole expression itself. An example of this is given in Figure 17.21 where the parametered element with label ‘n’ appears within the expression ‘n+1.’ Secondly, to note that by extending NamedElement to optionally own a name expression, strings that are part of these named expressions may be parametered.

Notation

Where a parametered ValueSpecification is used within an expression, the name of the parameter is used where any symbol (in case of an Expression) or value (in case of a Literal) would otherwise appear.

18 Profiles

18.1 Overview

The Profiles package contains mechanisms that allow metaclasses from existing metamodels to be extended to adapt them for different purposes. This includes the ability to tailor the UML metamodel for different platforms (such as J2EE or .NET) or domains (such as real-time or business process modeling). The profiles mechanism is consistent with the OMG Meta Object Facility (MOF).

18.1.1 Positioning profiles versus metamodels, MOF and UML

The infrastructure specification is reused at several meta-levels in various OMG specifications that deal with modeling. For example, MOF uses it to provide the ability to model metamodels, whereas the UML superstructure uses it to model the UML model. This clause deals with use cases comparable to the MOF at the meta-meta-level, which is one level higher than the rest of the superstructure specification. Thus, in this clause, when we mention “Class,” in most cases we are dealing with the meta-metaclass “Class” (used to define every meta class in the UML superstructure specification (Activity, Class, State, Use Case, etc.).

18.1.2 Profiles History and design requirements

The Profile mechanism has been specifically defined for providing a lightweight extension mechanism to the UML standard. In UML 1.1, stereotypes and tagged values were used as string-based extensions that could be attached to UML model elements in a flexible way. In subsequent revisions of UML, the notion of a Profile was defined in order to provide more structure and precision to the definition of Stereotypes and Tagged values. The UML2.0 infrastructure and superstructure specifications have carried this further, by defining it as a specific meta-modeling technique. Stereotypes are specific metaclasses, tagged values are standard metaattributes, and profiles are specific kinds of packages.

The following requirements have driven the definition of profile semantics from inception:

1. A profile must provide mechanisms for specializing a reference metamodel (such as a set of UML packages) in such a way that the specialized semantics do not contradict the semantics of the reference metamodel. That is, profile constraints may typically define well-formedness rules that are more constraining (but consistent with) those specified by the reference metamodel.

2. It must be possible to interchange profiles between tools, together with models to which they have been applied, by using the UML XMI interchange mechanisms. A profile must therefore be defined as an interchangeable UML model. In addition to exchanging profiles together with models between tools, profile application should also be definable “by reference” (e.g., “import by name”); that is, a profile does not need to be interchanged if it is already present in the importing tool.

3. A profile must be able to reference domain-specific UML libraries where certain model elements are pre-defined.

4. It must be possible to specify which profiles are being applied to a given Package (or any specializations of that concept). This is particularly useful during model interchange so that an importing environment can interpret a model correctly.

5. It should be possible to define a UML extension that combines profiles and model libraries (including template libraries) into a single logical unit. However, within such a unit, for definitional clarity and for ease of interchange (e.g., ‘reference by name’), it should still be possible to keep the libraries and the profiles distinct from each other.

6. A profile should be able to specialize the semantics of standard UML metamodel elements. For example, in a model with the profile “Java model,” generalization of classes should be able to be restricted to single inheritance without having to explicitly assign a stereotype «Java class» to each and every class instance.
7. A notational convention for graphical stereotype definitions as part of a profile should be provided.
8. In order to satisfy requirement [1] above, UML Profiles should form a metamodel extension mechanism that imposes certain restrictions on how the UML metamodel can be modified. The reference metamodel is considered as a “read only” model, that is extended without changes by profiles. It is therefore forbidden to insert new metaclasses in the UML metaclass hierarchy (i.e., new super-classes for standard UML metaclasses) or to modify the standard UML metaclass definitions (e.g., by adding meta-associations). Such restrictions do not apply in a MOF context where in principle any metamodel can be reworked in any direction.
9. The vast majority of UML case tools should be able to implement Profiles. The design of UML profiles should therefore not constrain these tools to have an internal implementation based on a meta-metamodel/metamodel architecture.
10. Profiles can be dynamically applied to or retracted from a model. It is possible on an existing model to apply new profiles, or to change the set of applied profiles.
11. Profiles can be dynamically combined. Frequently, several profiles will be applied at the same time on the same model. This profile combination may not be foreseen at profile definition time.
12. Models can be exchanged regardless of the profiles known by the destination target. The destination of the exchange of a model extended by a profile may not know the profile, and is not required to interpret a specific profile description. The destination environment interprets extensions only if it possesses the required profiles.

Extensibility

The profiles mechanism is not a first-class extension mechanism (i.e., it does not allow for modifying existing metamodels). Rather, the intention of profiles is to give a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method. Each such adaptation is grouped in a profile. It is not possible to take away any of the constraints that apply to a metamodel such as UML using a profile, but it is possible to add new constraints that are specific to the profile. The only other restrictions are those inherent in the profiles mechanism; there is nothing else that is intended to limit the way in which a metamodel is customized.

First-class extensibility is handled through MOF, where there are no restrictions on what you are allowed to do with a metamodel: you can add and remove metaclasses and relationships as you find necessary. Of course, it is then possible to impose methodology restrictions that you are not allowed to modify existing metamodels, but only extend them. In this case, the mechanisms for first-class extensibility and profiles start coalescing.

There are several reasons why you may want to customize a metamodel:

- Give a terminology that is adapted to a particular platform or domain (such as capturing EJB terminology like home interfaces, enterprise java beans, and archives).
- Give a syntax for constructs that do not have a notation (such as in the case of actions).
- Give a different notation for already existing symbols (such as being able to use a picture of a computer instead of the ordinary node symbol to represent a computer in a network).
- Add semantics that is left unspecified in the metamodel (such as how to deal with priority when receiving signals in a statemachine).

- Add semantics that does not exist in the metamodel (such as defining a timer, clock, or continuous time).
- Add constraints that restrict the way you may use the metamodel and its constructs (such as disallowing actions from being able to execute in parallel within a single transition).
- Add information that can be used when transforming a model to another model or code (such as defining mapping rules between a model and Java code).

Profiles and Metamodels

There is no simple answer for when you should create a new metamodel and when you instead should create a new profile.

18.2 Abstract Syntax

Package structure

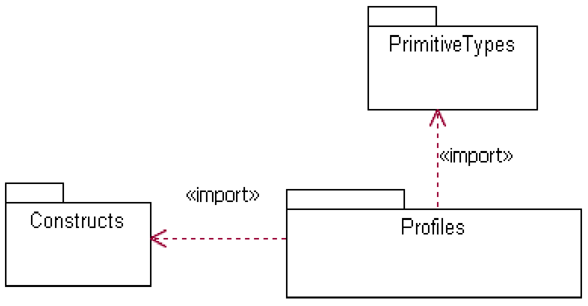


Figure 18.1 - Dependencies between packages described in this clause

The classes of the Profiles package are depicted in Figure 18.2, and subsequently specified textually.

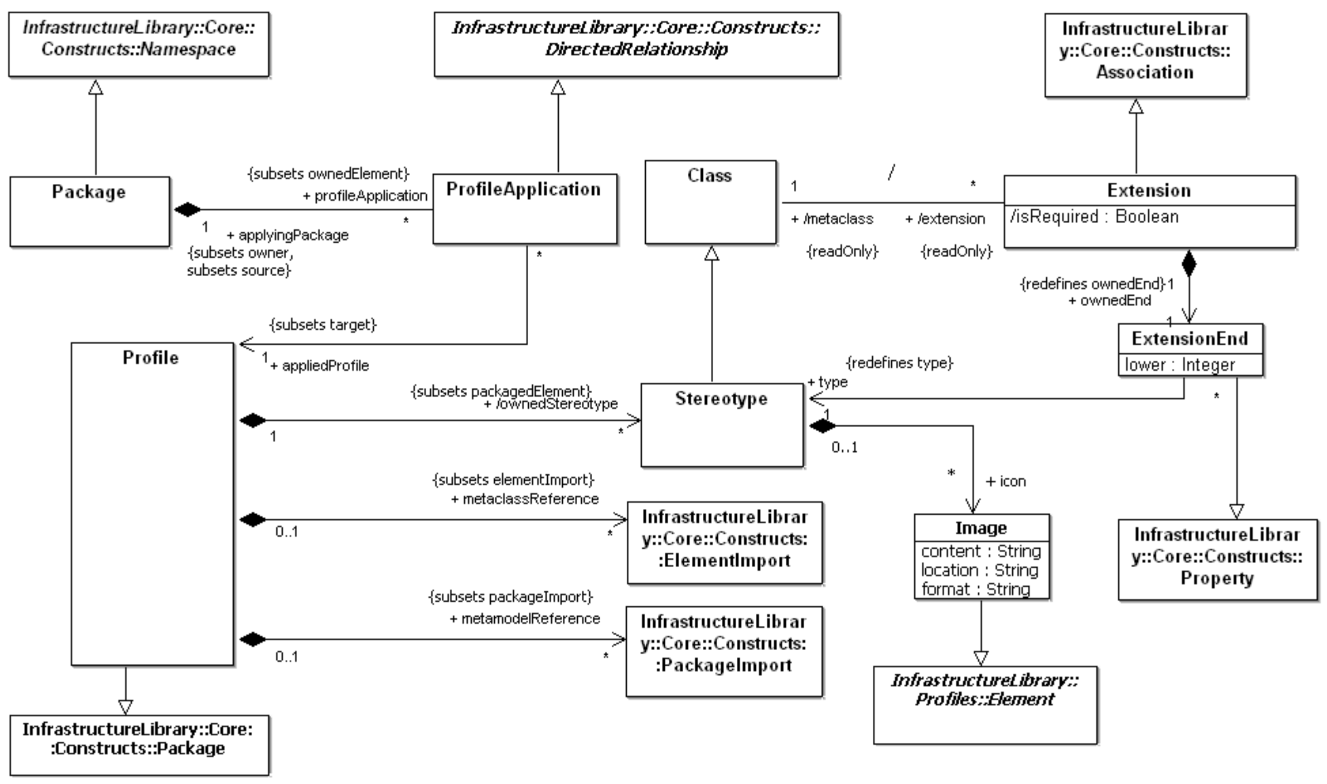

InfrastructureLibrary::Core::Constructs::Namespace
InfrastructureLibrary::Core::Constructs::DirectedRelationship
InfrastructureLibrary::Core::Constructs::Association
Package
ProfileApplication
Class
Extension
/isRequired : Boolean
{subsets ownedElement}
+ profileApplication
1
+ applyingPackage
{subsets owner, subsets source}
+ /metaclass
+ /extension
{readOnly}
{redefines ownedEnd}
+ ownedEnd
ExtensionEnd
lower : Integer
{subsets target}
Profile
+ appliedProfile
{subsets packagedElement}
+ /ownedStereotype
Stereotype
{redefines type}
+ type
0..1
+ icon
{subsets elementImport}
+ metaclassReference
InfrastructureLibrary::Core::Constructs::ElementImport
Image
content : String
location : String
format : String
InfrastructureLibrary::Core::Constructs::Property
{subsets packageImport}
+ metamodelReference
InfrastructureLibrary::Core::Constructs::PackageImport
InfrastructureLibrary::Profiles::Element
InfrastructureLibrary::Core::Constructs::Package

Figure 18.2 - The elements defined in the Profiles package

18.3 Class Descriptions

18.3.1 Class (from Profiles)

Generalizations

- InfrastructureLibrary::Constructs::Class *(merge increment)*

Description

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

Stereotype is the only kind of metaclass that cannot be extended by stereotypes.

Attributes

No additional attributes

Associations

- / extension: Extension [*]
  References the Extensions that specify additional properties of the metaclass. The property is derived from the extensions whose memberEnds are typed by the Class.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

Presentation Options

A Class that is extended by a Stereotype may be extended by the optional stereotype «metaclass» (see Annex C., “Standard Stereotypes”) shown above or before its name.

Examples

In Figure 18.3, an example is given where it is made explicit that the extended class *Interface* is in fact a metaclass (from a reference metamodel).

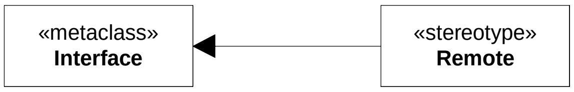


Figure 18.3 - Showing that the extended class is a metaclass

Changes from previous UML

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

18.3.2 Extension (from Profiles)

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

Generalizations

- InfrastructureLibrary::Constructs::Association

Description

Extension is a kind of Association. One end of the Extension is an ordinary Property and the other end is an ExtensionEnd. The former ties the Extension to a Class, while the latter ties the Extension to a Stereotype that extends the Class.

Attributes

- / isRequired: Boolean
  Indicates whether an instance of the extending stereotype must be created when an instance of the extended class is created. The attribute value is derived from the multiplicity of the Property referenced by *Extension::ownedEnd*; a

multiplicity of 1 means that isRequired is *true*, but otherwise it is *false*. Since the default multiplicity of an ExtensionEnd is 0..1, the default value of isRequired is *false*.

Associations

- ownedEnd: ExtensionEnd [1]
  References the end of the extension that is typed by a Stereotype. {Redefines *Association::ownedEnd*}
- / metaclass: Class [1]
  References the Class that is extended through an Extension. The property is derived from the type of the memberEnd that is not the ownedEnd.

Constraints

[1] The non-owned end of an Extension is typed by a Class.
metaclassEnd()->notEmpty() **and** metaclass()->oclIsKindOf(Class)

[2] An Extension is binary (i.e., it has only two memberEnds).
memberEnd->size() = 2

Additional Operations

[1] The query metaclassEnd() returns the Property that is typed by a metaclass (as opposed to a stereotype).
Extension::metaclassEnd(): Property;
metaclassEnd = memberEnd->reject(ownedEnd)

[2] The query metaclass() returns the metaclass that is being extended (as opposed to the extending stereotype).
Extension::metaclass(): Class;
metaclass = metaclassEnd().type

[3] The query isRequired() is true if the owned end has a multiplicity with the lower bound of 1.
Extension::isRequired(): Boolean;
isRequired = (ownedEnd->lowerBound() = 1)

Semantics

A required extension means that an instance of a stereotype must always be linked to an instance of the extended metaclass. The instance of the stereotype is typically deleted only when either the instance of the extended metaclass is deleted, or when the profile defining the stereotype is removed from the applied profiles of the package. The model is not well formed if an instance of the stereotype is not present when isRequired is true. If the extending stereotype has subclasses, then at most one instance of the stereotype or one of its subclasses is required.

A non-required extension means that an instance of a stereotype can be linked to an instance of an extended metaclass at will, and also later deleted at will; however, there is no requirement that each instance of a metaclass be extended. An instance of a stereotype is further deleted when either the instance of the extended metaclass is deleted, or when the profile defining the stereotype is removed from the applied profiles of the package.

The equivalence to a MOF construction is shown in Figure 18.4. This figure illustrates the case shown in Figure 18.6, where the “Home” stereotype extends the “Interface” metaclass. In this figure, Interface is an instance of a CMOF::Class and Home is an instance of a CMOF::Stereotype. The MOF construct equivalent to an extension is an aggregation from the extended metaclass to the extension stereotype, navigable from the extension stereotype to the extended metaclass. When the extension is required, then the cardinality on the extension stereotype is “1.” The role names are provided using the following rule: The name of the role of the extended metaclass is:

'base_' extendedMetaclassName

The name of the role of the extension stereotype is:

'extension$_' stereotypeName

Constraints are frequently added to stereotypes. The role names will be used for expressing OCL navigations. For example, the following OCL expression states that a Home interface shall not have attributes:

```
self.base_Interface.ownedAttributes->size() = 0
```

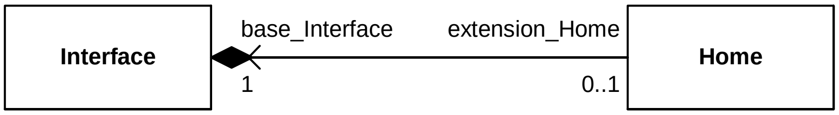


Figure 18.4 - MOF model equivalent to extending "interface" by the "Home" stereotype

Notation

The notation for an Extension is an arrow pointing from a Stereotype to the extended Class, where the arrowhead is shown as a filled triangle. An Extension may have the same adornments as an ordinary association, but navigability arrows are never shown. If isRequired is true, the property {required} is shown near the ExtensionEnd.

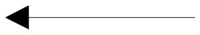


Figure 18.5 - The notation for an Extension

Presentation Options

It is possible to use the multiplicities 0..1 or 1 on the ExtensionEnd as an alternative to the property {required}. Due to how isRequired is derived, the multiplicity 0..1 corresponds to isRequired being *false*.

Style Guidelines

Adornments of an Extension are typically elided.

Examples

In Figure 18.6, a simple example of using an extension is shown, where the stereotype *Home* extends the metaclass *Interface*.

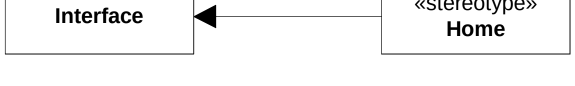


Figure 18.6 - An example of using an Extension

An instance of the stereotype *Home* can be added to and deleted from an instance of the class *Interface* at will, which provides for a flexible approach of dynamically adding (and removing) information specific to a profile to a model.

In Figure 18.7, an instance of the stereotype *Bean* always needs to be linked to an instance of class *Component* since the Extension is defined to be required. (Since the stereotype *Bean* is abstract, this means that an instance of one of its concrete subclasses always has to be linked to an instance of class *Component*.) The model is not well formed unless such a stereotype is applied. This provides a way to express extensions that should always be present for all instances of the base metaclass depending on which profiles are applied.

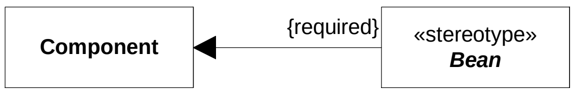


Figure 18.7 - An example of a required Extension

Changes from previous UML

Extension did not exist as a metaclass in UML 1.x.

Occurrences of Stereotype::baseClass of UML 1.4 is mapped to an instance of Extension, where the ownedEnd is typed by Stereotype and the other end is typed by the metaclass that is indicated by the baseClass.

18.3.3 ExtensionEnd (from Profiles)

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

Generalizations

- InfrastructureLibrary::Constructs::Property

Description

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

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

The aggregation of an ExtensionEnd is always composite.

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

Attributes

- lower : integer = 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. {*Redefines MultiplicityElement::lower*}

Associations

- type: Stereotype [1]
  References the type of the ExtensionEnd. Note that this association restricts the possible types of an ExtensionEnd to only be Stereotypes. {Redefines *TypedElement::type*}

Constraints

[1] The multiplicity of ExtensionEnd is 0..1 or 1.

(self->lowerBound() = 0 **or** self->lowerBound() = 1) **and** self->upperBound() = 1

[2] The aggregation of an ExtensionEnd is composite.

self.aggregation = #composite

Additional Operations

[1] The query lowerBound() returns the lower bound of the multiplicity as an Integer. This is a redefinition of the default lower bound, which, if empty, normally evaluates to 1 for MulticplicityElements.

ExtensionEnd::lowerBound() : Integer;

lowerBound = **if** lowerValue->isEmpty() **then** 0 **else** lowerValue->IntegerValue() **endif**

Semantics

No additional semantics

Notation

No additional notation

Examples

See "Class (from Profiles)" on page 656.

Changes from previous UML

ExtensionEnd did not exist as a metaclass in UML 1.4. See "Class (from Profiles)" on page 656 for further details.

18.3.4 Image (from Profiles)

Physical definition of a graphical image.

Generalizations

- "Element (from Kernel)" on page 64

Description

The Image class provides the necessary information to display an Image in a diagram. Icons are typically handled through the Image class.

Attributes

- content : String [0..1]
  This contains the serialization of the image according to the imageFormat. The value could represent a bitmap, image

such as a GIF file, or drawing 'instructions' using a standard such as Scalable Vector Graphics (SVG) (which is XML based).

- format : String [0..1]
  This indicates the format of the imageContent - which is how the string imageContent should be interpreted. The following values are reserved: SVG, GIF, PNG, JPG, WMF, EMF, BMP.

  In addition the prefix 'MIME:' is also reserved: this must be followed by a valid MIME type as defined by RFC3023. This option can be used as an alternative to express the reserved values above, for example "SVG" could instead be expressed "MIME: image/svg+xml."
- location : String [0..1]
  This contains a location that can be used by a tool to locate the image as an alternative to embedding it in the stereotype.

Associations

No additional associations

Constraints

No additional constraints

Semantics

Information such as physical localization or format is provided by the Image class. The Image class provides a generic way of representing images in different formats. Although some predefined values are specified for imageFormat for convenience and interoperability, the set of possible formats is open ended. However there is no requirement for an implementation to be able to interpret and display any specific format, including those predefined values.

18.3.5 Package (from Profiles)

Generalizations

- InfrastructureLibrary::Constructs::Package *(merge increment)*

Description

A package can have one or more ProfileApplications to indicate which profiles have been applied.

Because a profile is a package, it is possible to apply a profile not only to packages, but also to profiles.

Attributes

No additional attributes

Associations

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

Constraints

No additional constraints

Semantics

The association “appliedProfile” between a package and a profile crosses metalevels: It links one element from a model (a kind of package) to an element of its metamodel and represents the set of profiles that define the extensions applicable to the package. Although this kind of situation is rare in the UML metamodel, this only shows that model and metamodel can coexist on the same space, and can have links between them.

Notation

No additional notation

Changes from previous UML

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

18.3.6 Profile (from Profiles)

A profile defines limited extensions to a reference metamodel with the purpose of adapting the metamodel to a specific platform or domain.

Generalizations

- InfrastructureLibrary::Constructs::Package

Description

A Profile is a kind of Package that extends a reference metamodel. The primary extension construct is the Stereotype, which is defined as part of Profiles.

A profile introduces several constraints, or restrictions, on ordinary metamodeling through the use of the metaclasses defined in this package.

A profile is a restricted form of a metamodel that must always be related to a *reference metamodel*, such as UML, as described below. A profile cannot be used without its reference metamodel, and defines a limited capability to extend metaclasses of the reference metamodel. The extensions are defined as stereotypes that apply to existing metaclasses.

Attributes

No additional attributes

Associations

- metaclassReference: ElementImport [*]
  References a metaclass that may be extended. Subsets *Package::elementImport*
- metamodelReference: PackageImport [*]
  References a package containing (directly or indirectly) metaclasses that may be extended. Subsets *Package::packageImport*
- /ownedStereotype: Stereotype [*]
  References the Stereotypes that are owned by the Profile. Subsets *Package::packagedElement*

Constraints

[1] An element imported as a metaclassReference is not specialized or generalized in a Profile.

```
self.metaclassReference.importedElement->
    select(c | c.oclIsKindOf(Classifier) and
    (c.generalization.namespace = self or
    (c.specialization.namespace = self) )->isEmpty()
```

[2] All elements imported either as metaclassReferences or through metamodelReferences are members of the same base reference metamodel.

```
self.metamodelReference.importedPackage.elementImport.importedElement.allOwningPackages())->
    union(self.metaclassReference.importedElement.allOwningPackages() )->notEmpty()
```

Additional Operations

[1] The query allOwningPackages() returns all the directly or indirectly owning packages.

```
NamedElement::allOwningPackages(): Set(Package)
allOwningPackages = self.namespace->select(p | p.oclIsKindOf(Package))->
    union(p.allOwningPackages())
```

Semantics

A profile by definition extends a reference metamodel. It is not possible to define a standalone profile that does not directly or indirectly extend an existing metamodel. The profile mechanism may be used with any metamodel that is created from MOF, including UML and CWM.

A reference metamodel typically consists of metaclasses that are either imported or locally owned. All metaclasses that are extended by a profile have to be members of the same reference metamodel. The “metaclassReference” element imports and “metamodelReference” package imports serve two purposes: (1) they identify the reference metamodel elements that are imported by the profile and (2) they specify the profile’s filtering rules. The filtering rules determine which elements of the metamodel are *visible* when the profile is applied and which ones are *hidden*. Note that applying a profile does not change the underlying model in any way; it merely defines a view of the underlying model.

In general, only model elements that are instances of imported reference metaclasses will be visible when the profile is applied. All other metaclasses will be hidden. By default, model elements whose metaclasses are public and owned by the reference metamodel are visible. This applies transitively to any subpackages of the reference metamodel according to the default rules of package import. If any metaclasses is imported using a *metaclass reference* element import, then model elements whose metaclasses are the same as that metaclass are visible. Note, however, that a metaclass reference overrides a *metamodel reference* whenever an element or package of the referenced metamodel is also referenced by a metaclass reference. In such cases, only the elements that are explicitly referenced by the metaclass reference will be visible, while all other elements of the metamodel package will be hidden.

The following rules are used to determine whether a model element is visible or hidden when a profile has been applied. Model elements are *visible* if they are instances of metaclasses that are:

1. referenced by an explicit MetaclassReference, or
2. contained (directly or transitively) in a package that is referenced by an explicit MetamodelReference; unless there are other elements of subpackages of that package that are explicitly referenced by a MetaclassReference, or
3. extended by a stereotype owned by the applied profile (even if the extended metaclass itself is not visible).

All other model elements are *hidden* (not visible) when the profile is applied.

The most common case is when a profile just imports an entire metamodel using a MetamodelReference. In that case, every element of the metamodel is visible.

In the example in Figure 18.8, MyMetamodel is a metamodel containing two metaclasses: Metaclass1 and Metaclass2. MyProfile is a profile that references MyMetamodel and Metaclass2. However, there is also an explicit metaclass reference to Metaclass2, which overrides the metamodel reference. An application of MyProfile to some model based on MyMetamodel will show instances of Metaclass2 (because it is referenced by an explicit metaclass reference). Also, those instances of Metaclass1 that are extended by an instance of MyStereotype will be visible. However, instances of Metaclass1 that are not extended by MyStereotype remain hidden.

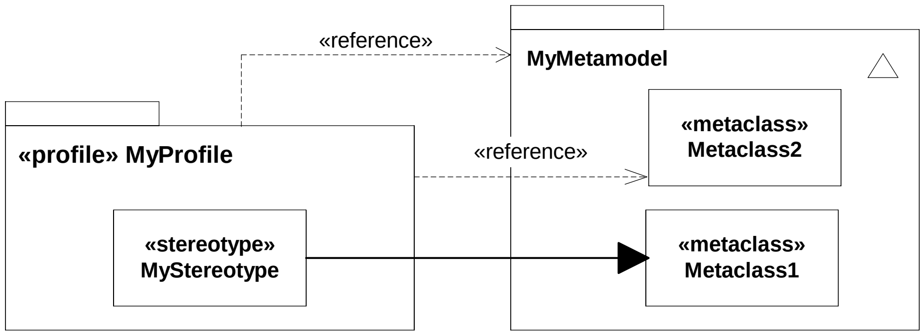


Figure 18.8 - Specification of an accessible metaclass

If a profile P1 imports another profile P2, then all metaclassReference and metamodelReference associations will be combined at the P2 level, and the filtering rules applies 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 “isStric” attribute on a profileApplication specifies that the filtering rules have to be applied strictly. If isStrict is true on a ProfileApplication, then no other metaclasses than the accessible one defined by the profile shall be accessible when the profile is applied on a model. This prohibits the combination of applied profiles that specify different accessible metaclasses.

In addition to these import and filtering mechanisms, profile designers can select the appropriate metamodel by selecting the appropriate subpackages, and using the package merge mechanism. For example, they can build a specific reference metamodel by merging UML2 superstructure packages and classes, and or import packages from one of the UML2 compliance packages. (L0-L4)

Stereotypes can participate in associations. The opposite class can be another stereotype, a non-stereotype class that is owned by a profile, or a metaclass of the reference metamodel. For these associations there must be a property owned by the Stereotype to navigate to the opposite class. The opposite property must be owned by the Association itself rather than the other class/metaclass.

The most direct implementation of the Profile mechanism that a tool can provide is by having a metamodel based implementation, similar to the Profile metamodel. However, this is not a requirement of the current standard, which requires only the support of the specified notions, and the standard XMI based interchange capacities. The profile mechanism has been designed to be implementable by tools that do not have a metamodel-based implementation. Practically any mechanism used to attach new values to model elements can serve as a valid profile implementation. As an example, the UML1.4 profile metamodel could be the basis for implementing a UML2.0-compliant profile tool.

Using XMI to exchange Profiles

A profile is an instance of a UML2 metamodel, not a CMOF metamodel. Therefore the MOF to XMI mapping rules do not directly apply for instances of a profile. Figure 18.4 on page 659 is an example of a mapping between a UML2 Profile and an equivalent CMOF model. This mapping is used as a means to explain and formalize how profiles are serialized and exchanged as XMI. Using this Profile to CMOF mapping, rules for mapping CMOF to XMI can be used indirectly to specify mappings from Profiles to XMI. In the mapping:

- A Profile maps to a CMOF Package.
- The metaclass, Extension::Class, is an instance of a MOF class; Extension::Class maps to itself (that is, a Class in the profile is also the same Class in the corresponding CMOF model; the mapping is the identity transformation).
- A Stereotype maps to a MOF class with the same name and properties.
- An Extension maps to an Association as described in the Semantics sub clause of Profile::Class on page 657.

The Profile to CMOF mapping should also include values for CMOF Tags on the CMOF package corresponding to the Profile in order to control the XMI generation. The exact specification of these tags is a semantic variation point. A recommended convention is:

```
nsURI = http://<profileParentQualifiedName>/schemas/<profileName>.xmi
```

where:

- `<profileParentQualifiedName>` is the qualified name of the package containing the Profile (if any) with . (dot) substituted for ::, and all other illegal XML QName characters removed, and
    - `<profileName>` is the name of the Profile,
    - nsPrefix = <profileName>,
    - all others use the XMI defaults.

A profile can be exchanged just like any model, as an XMI schema definition, and models that are extended by a profile can also be interchanged.

Figure 18.6 on page 659 shows a “Home” stereotype extending the “Interface” UML2 metaclass. Figure 18.4 on page 659 illustrates the MOF correspondence for that example, basically by introducing an association from the “Home” MOF class to the “Interface” MOF class. For illustration purposes, we add a property (tagged value definition in UML1.4) called “magic:String” to the “Home” stereotype.

The first serialization below shows how the model Figure 18.5 on page 659 (instance of the profile and UML2 metamodel) can be exchanged.

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Profile xmi:version="2.1"
    nsURI="http://HomeExample.xml"
    nsPrefix="HomeExample"
    xmlns:uml="http://schema.omg.org/spec/UML/2.0/uml.xml"
    xmi:id="id0" name="HomeExample" metamodelReference="id2">

  <packageImport xmi:id="id2">
    <importedPackage href="http://schema.omg.org/spec/UML/2.0/uml.xml"/>
  </packageImport>

  <ownedMember 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://schema.omg.org/spec/UML/2.0/uml.xml#Interface"/>
    </ownedAttribute>
    <ownedAttribute xmi:type="uml:Property" xmi:id="id4" name="magic">
      <type href="http://schema.omg.org/spec/UML/2.0/uml.xml#String"/>
    </ownedAttribute>
  </ownedMember>

  <ownedMember xmi:type="uml:Extension" xmi:id="id6" name="A_Interface_Home"
    memberEnd="id4 id5">
    <ownedEnd
     xmi:type="uml:ExtensionEnd"
     xmi:id="id7"
     name="extension_Home" type="id3"
     isComposite="true" lower="0" upper="1">
    </ownedEnd>
  </ownedMember>
</uml:Profile>
```

Next, we describe an XMI definition from the «HomeExample» profile. That XMI description will itself define in XML how the models extended by the HomeExample will be exchanged in XMI. We can see here that an XMI schema separated from the standard UML2 schema can be obtained. This XMI definition is stored in a file called “HomeExample.xmi.”

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace =
     "http://www.mycompany.com/schemas/HomeExample.xmi"
     xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
     xmlns:uml="http://schema.omg.org/spec/UML/2.1"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">

     <xsd:import namespace="http://schema.omg.org/spec/XMI/2.1" schemaLocation="XMI.xsd"/>
     <xsd:import namespace="http://schema.omg.org/spec/UML/2.0" schemaLocation="UML21.xsd"/>
     <xsd:complexType name="Home">
          <xsd:choice minOccurs="0" maxOccurs ="unbounded">
               <xsd:element name="base_Interface" type="xmi:Any"/>
               <xsd:element name="magic" type="xsd:string"/>
               <xsd:element ref="xmi:Extension"/>
          </xsd:choice>
          <xsd:attribute ref="xmi:id"/>
          <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
          <xsd:attribute name="base_Interface" type="uml:Interface"
          <xsd:attribute name="magic" type="xsd:string" use="optional"/>
          use="optional"/>
     </xsd:complexType>
     <xsd:element name="Home" type="Home"/>
</xsd:schema>
```

Figure 18.9 is an example model that includes an instance of Interface extended by the Home stereotype.

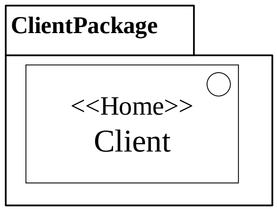


Figure 18.9 - Using the “HomeExample” profile to extend a model

Now the XMI code below shows how this model extended by the profile is serialized. A tool importing that XMI file can filter out the elements related to the “HomeExample” schema, if the tool does not have this schema (profile) definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi xmi:version="2.1"
    xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
    xmlns:uml="http://schema.omg.org/spec/UML/2.1"
    xmlns:HomeExample="http://www.mycompany.com/schemas/HomeExample.xmi">

  <uml:Package xmi:id="id1" name="ClientPackage">
    <profileApplication xmi:id="id3">
        <appliedProfile href="HomeExample.xmi#id0"/>
    </profileApplication>

    <ownedMember xmi:type="uml:Interface" xmi:id="id2" name="Client"/>
  </uml:Package>

  <!-- applied stereotypes -->
  <HomeExample:Home xmi:id= "id4" base_Interface="id2" magic="1234"/>
</xmi>
```

Notation

A Profile uses the same notation as a Package, with the addition that the keyword «profile» is shown before or above the name of the Package. *Profile::metaclassReference* and *Profile::metamodelReference* uses the same notation as *Package::elementImport* and *Package::packageImport*, respectively.

Examples

In Figure 18.10, a simple example of an EJB profile is shown.

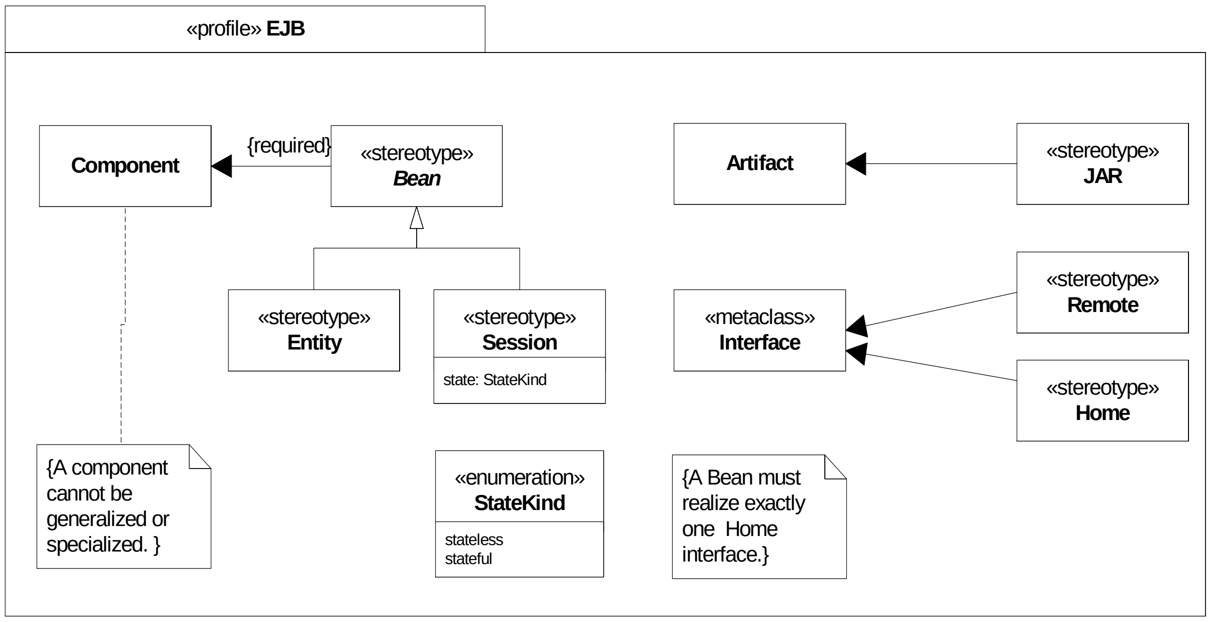


Figure 18.10 - Defining a simple EJB profile

The profile states that the abstract stereotype *Bean* is required to be applied to metaclass *Component*, which means that an instance of either of the concrete subclasses *Entity* and *Session* of *Bean* must be linked to each instance of *Component*. The constraints that are part of the profile are evaluated when the profile has been applied to a package, and need to be satisfied in order for the model to be well formed.

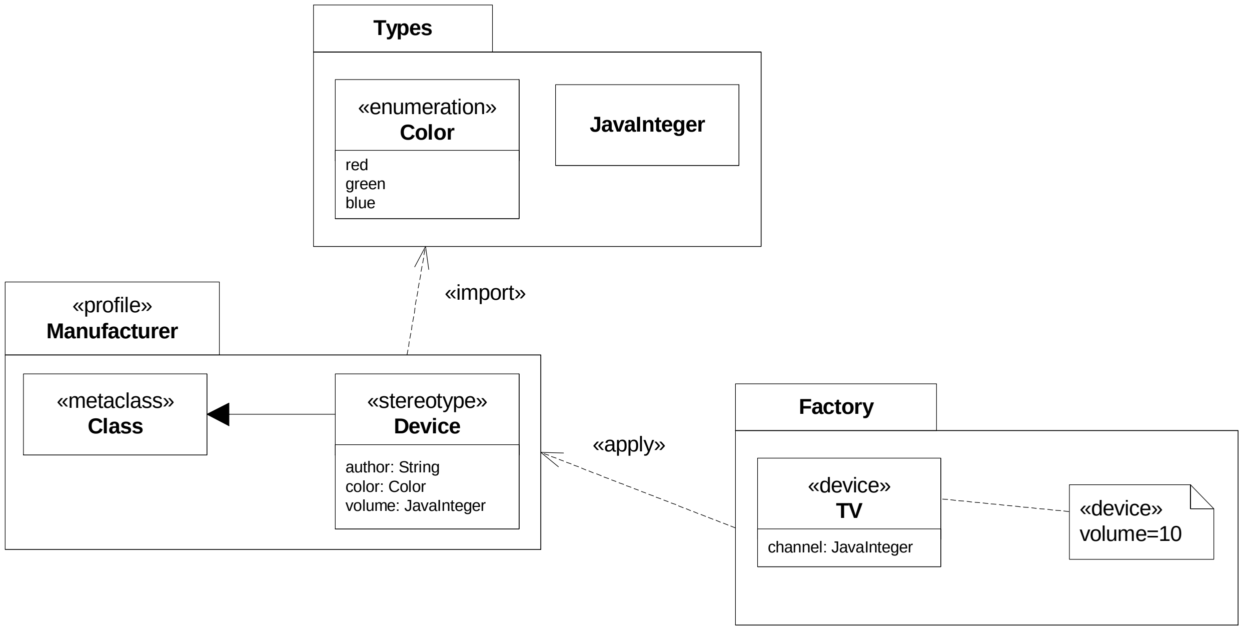


Figure 18.11 - Importing a package from a profile

In Figure 18.11, the package *Types* is imported from the profile *Manufacturer*. The data type *Color* is then used as the type of one of the properties of the stereotype *Device*, just like the predefined type String is also used. Note that the class *JavaInteger* may also be used as the type of a property.

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

18.3.7 ProfileApplication (from Profiles)

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

Generalizations

- InfrastructureLibrary::Constructs::Core::DirectedRelationship

Description

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

Attributes

- isStrict : Boolean [1] = false
  Specifies whether or not the Profile filtering rules for the metaclasses of the referenced metamodel shall be strictly applied. See the semantics sub clause of “Profile (from Profiles)” on page 663 for further details.

Associations

- importedProfile: Profile [1]
  References the Profiles that are applied to a Package through this ProfileApplication. Subsets *PackageImport::importedPackage*
- applyingPackage : Package [1]
  The package that owns the profile application. {*Subsets Element::owner* and *DirectedRelationship::source*}

Constraints

No additional constraints

Semantics

One or more profiles may be applied at will to a package that is created from the same metamodel that is extended by the profile. Applying a profile means that it is allowed, but not necessarily required, to apply the stereotypes that are defined as part of the profile. It is possible to apply multiple profiles to a package as long as they do not have conflicting constraints. If a profile that is being applied depends on other profiles, then those profiles must be applied first.

When a profile is applied, instances of the appropriate stereotypes should be created for those elements that are instances of metaclasses with required extensions. The model is not well formed without these instances.

Once a profile has been applied to a package, it is allowed to remove the applied profile at will. Removing a profile implies that all elements that are instances of elements defined in a profile are deleted. A profile that has been applied cannot be removed unless other applied profiles that depend on it are first removed.

Note – The removal of an applied profile leaves the instances of elements from the referenced metamodel intact. It is only the instances of the elements from the profile that are deleted. This means that for example a profiled UML model can always be interchanged with another tool that does not support the profile and be interpreted as a pure UML model.

Notation

The names of Profiles are shown using a dashed arrow with an open arrowhead from the package to the applied profile. The keyword «apply» is shown near the arrow.

If multiple applied profiles have stereotypes with the same name, it may be necessary to qualify the name of the stereotype (with the profile name).

Examples

Given the profiles *Java* and *EJB*, Figure 18.12 shows how these have been applied to the package *WebShopping*.

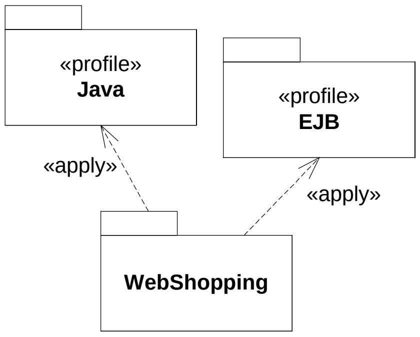


Figure 18.12 - Profiles applied to a package

18.3.8 Stereotype (from Profiles)

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

Generalizations

- InfrastructureLibrary::Constructs::Class

Description

Stereotype is a kind of Class that extends Classes through Extensions.

Just like a class, a stereotype may have properties, which may be referred to as tag definitions. When a stereotype is applied to a model element, the values of the properties may be referred to as tagged values.

Attributes

No additional attributes

Associations

- icon : Image [*]
  Stereotype can change the graphical appearance of the extended model element by using attached icons. When this association is not null, it references the location of the icon content to be displayed within diagrams presenting the extended model elements.

Constraints

[1] A Stereotype may only generalize or specialize another Stereotype.
generalization.general->forAll(e | e.oclIsKindOf(Stereotype)) **and**
generalization.specific->forAll(e | e.oclIsKindOf(Stereotype))

[2] Stereotype names should not clash with keyword names for the extended model element.

Semantics

A stereotype is a limited kind of metaclass that cannot be used by itself, but must always be used in conjunction with one of the metaclasses it extends. Each stereotype may extend one or more classes through extensions as part of a profile. Similarly, a class may be extended by one or more stereotypes.

An instance “S” of Stereotype is a kind of (meta) class. Relating it to a metaclass “C” from the reference metamodel (typically UML) using an “Extension” (which is a specific kind of association), signifies that model elements of type C can be extended by an instance of “S” (see example in Figure 18.13). At the model level (such as in Figure 18.18) instances of “S” are related to “C” model elements (instances of “C”) by links (occurrences of the association/extension from “S’ to “C”).

Any model element from the reference metamodel (any UML model element) can be extended by a stereotype. For example in UML, States, Transitions, Activities, Use cases, Components, Attributes, Dependencies, etc. can all be extended with stereotypes.

Notation

A Stereotype uses the same notation as a Class, with the addition that the keyword «stereotype» is shown before or above the name of the Class.

When a stereotype is applied to a model element (an instance of a stereotype is linked to an instance of a metaclass), the name of the stereotype is shown within a pair of guillemets above or before the name of the model element. If multiple stereotypes are applied, the names of the applied stereotypes are shown as a comma-separated list with a pair of guillemets. When the extended model element has a keyword, then the stereotype name will be displayed close to the keyword, within separate guillemets (example: «interface» «Clock»).

Presentation Options

If multiple stereotypes are applied to an element, it is possible to show this by enclosing each stereotype name within a pair of guillemets and listing them after each other. A tool can choose whether it will display stereotypes or not. In particular, some tools can choose not to display “required stereotypes,” but to display only their attributes (tagged values) if any.

The values of a stereotyped element can be shown in one of the following three ways:

- As part of a comment symbol connected to the graphic node representing the model element.
- In separate compartments of a graphic node representing that model element.
- Above the name string within the graphic node or, else, before the name string.

In the case where a compartment or comment symbol is used, the stereotype name may shown in guillemets before the name string in addition to being included in the compartment or comment.

The values are displayed as name-value pairs:

    <namestring> ‘=’ <valuestring>

If a stereotype property is multi-valued, then the <valuestring> is displayed as a comma-separated list:

    <valuestring> ::= <value> [‘,’ <value>]*

Certain values have special display rules:

- As an alternative to a name-value pair, when displaying the values of Boolean properties, diagrams may use the convention that if the <namestring> is displayed, then the value is True; otherwise, the value is False.
- If the value is the name of a NamedElement, then, optionally, its qualifiedName can be used.

If compartments are used to display stereotype values, then an additional compartment is required for each applied stereotype whose values are to be displayed. Each such compartment is headed by the name of the applied stereotype in guillemets. Any graphic node may have these compartments.

Within a comment symbol, or, if displayed before or above the symbols' <namestring>, the values from a specific stereotype are optionally preceded with the name of the applied stereotype within a pair of guillemets. This is useful if values of more than one applied stereotype should be shown.

When displayed in compartments or in a comment symbol, at most one name-value pair can appear on a single line. When displayed above or before a <namestring>, the name-value pairs are separated by semicolons and all pairs for a given stereotype are enclosed in braces.

If the extension end is given a name, this name can be used in lieu of the stereotype name within the pair of guillemets when the stereotype is applied to a model element.

It is possible to attach a specific notation to a stereotype that can be used in lieu of the notation of a model element to which the stereotype is applied.

Icon presentation

When a stereotype includes the definition of an icon, this icon can be graphically attached to the model elements extended by the stereotype. Every model element that has a graphical presentation can have an attached icon. When model elements are graphically expressed as:

- Boxes (see Figure 18.14 on page 675): the box can be replaced by the icon, and the name of the model element appears below the icon. This presentation option can be used only when a model element is extended by one single stereotype and when properties of the model element (i.e., attributes, operations of a class) are not presented. As another option, the icon can be presented in a reduced shape, inside and on top of the box representing the model element. When several stereotypes are applied, several icons can be presented within the box.
- Links: the icon can be placed close to the link.
- Textual notation: the icon can be presented to the left of the textual notation.

Several icons can be attached to a stereotype. The interpretation of the different attached icons in that case is a semantic variation point. Some tools may use different images for the icon replacing the box, for the reduced icon inside the box, for icons within explorers, etc. Depending on the image format, other tools may choose to display one single icon into different sizes.

Some model elements are already using an icon for their default presentation. A typical example of this is the Actor model element, which uses the "stickman" icon. In that case, when a model element is extended by a stereotype with an icon, the stereotype's icon replaces the default presentation icon within diagrams.

Style Guidelines

The first letter of an applied stereotype should not be capitalized.

Examples

In Figure 18.13, a simple stereotype *Clock* is defined to be applicable at will (dynamically) to instances of the metaclass Class.

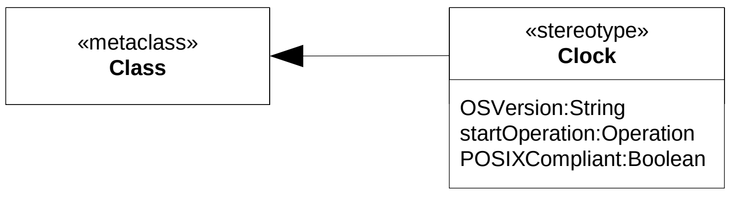


Figure 18.13 - Defining a stereotype

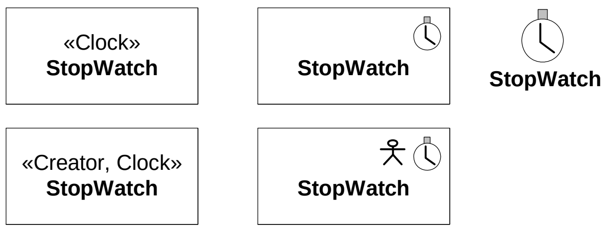


Figure 18.14 - Presentation options for an extended class

In Figure 18.15, an instance specification of the example in Figure 18.13 is shown. Note that the extension end must be composite, and that the derived isRequired" attribute in this case is false. Figure 18.15 shows the repository schema of the stereotype "clock" defined in Figure 18.13. In this schema, the extended instance (:Class; "name = Class") is defined in the UML2.0 (reference metamodel) repository. In a UML modeling tool these extended instances referring to the UML2.0 standard would typically be in a "read only" form, or presented as proxies to the metaclass being extended.

(It is therefore still at the same meta-level as UML, and does not show the instance model of a model extended by the stereotype. An example of this is provided in Figure 18.17 and Figure 18.18.) The Semantics sub clause of the Extension concept explains the MOF equivalent, and how constraints can be attached to stereotypes.

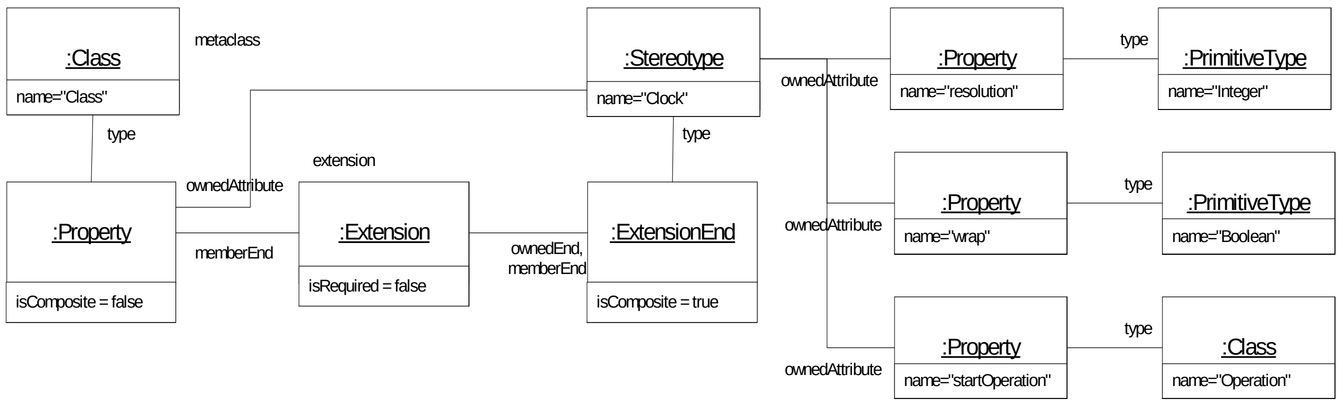


Figure 18.15 - An instance specification when defining a stereotype

Figure 18.16 shows how the same stereotype *Clock* can extend either the metaclass Component or the metaclass Class. It also shows how different stereotypes can extend the same metaclass.

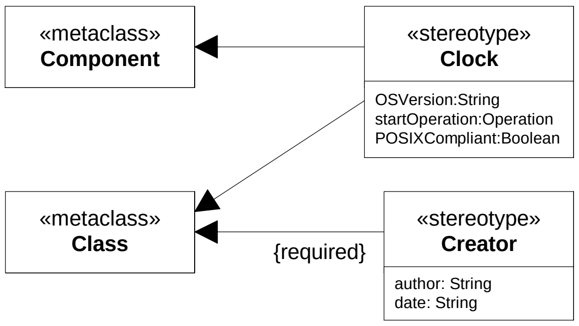


Figure 18.16 - Defining multiple stereotypes on multiple stereotypes

Figure 18.17 shows how the stereotype *Clock*, as defined in Figure 18.16, is applied to a class called *StopWatch*.

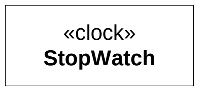


Figure 18.17 - Using a stereotype

Figure 18.18 shows an example instance model for when the stereotype *Clock* is applied to a class called *StopWatch*. The extension between the stereotype and the metaclass results in a link between the instance of stereotype *Clock* and the (user-defined) class *StopWatch*.

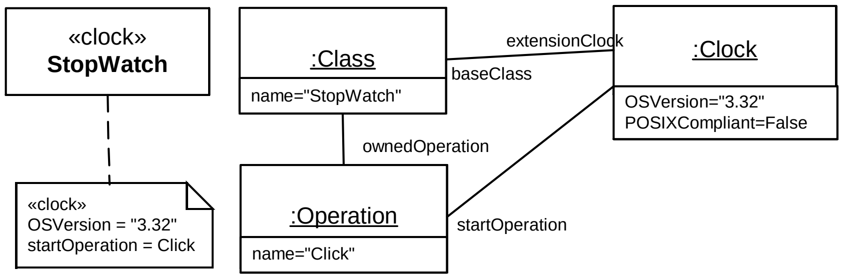


Figure 18.18 - Showing values of stereotypes and a simple instance specification

Next, two stereotypes, *Clock* and *Creator*, are applied to the same model element, as shown in Figure 18.19. Note that the attribute values of each of the applied stereotypes can be shown in a comment symbol attached to the model element.

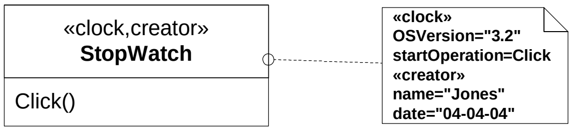


Figure 18.19 - Using stereotypes and showing values

Finally, two more alternative notational forms are shown in Figure 18.20.

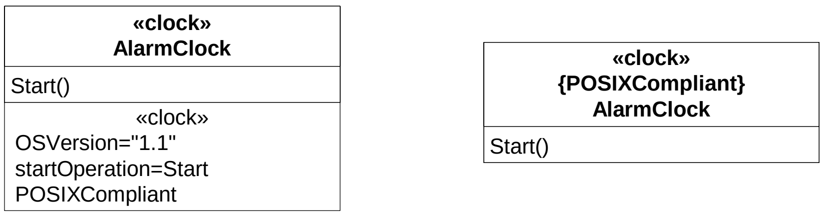


Figure 18.20 - Other notational forms for depicting stereotype values

Changes from previous UML

In UML 1.3, tagged values could extend a model element without requiring the presence of a stereotype. In UML 1.4, this capability, although still supported, was deprecated, to be used only for backward compatibility reasons. In UML 2.0, a tagged value can only be represented as an attribute defined on a stereotype. Therefore, a model element must be extended by a stereotype in order to be extended by tagged values. However, the "required" extension mechanism can, in effect, provide the 1.3 capability, since a tool can in those circumstances automatically define a stereotype to which "unattached" attributes (tagged values) would be attached.

18.4 Diagrams

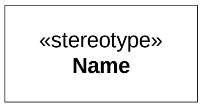
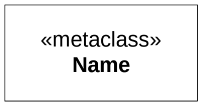
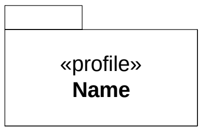
Structure diagrams

This sub clause outlines the graphic elements that may be shown in structure 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.

Graphical nodes

The graphic nodes that can be included in structure diagrams are shown in Table 18.1.

Table 18.1 - Graphic nodes included in structure diagrams

Node Type	Notation	Reference
Stereotype		See “Stereotype (from Profiles)” on page 672.
Metaclass		See “Class (from Profiles)” on page 656.
Profile		See “Profile (from Profiles)” on page 663.

Graphical paths

The graphic paths that can be included in structure diagrams are shown in Table 18.2.

Table 18.2 - Graphic paths included in structure diagrams

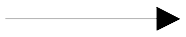
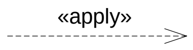
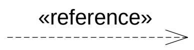
Node Type	Notation	Reference
Extension		See “Class (from Profiles)” on page 656.

Table 18.2 - Graphic paths included in structure diagrams

Node Type	**Notation**	**Reference**
ProfileApplication		See “ProfileApplication (from Profiles)” on page 670.
ElementImport PackageImport		See items “metaclassReference” and “metamodelReference” in “ProfileApplication (from Profiles)” on page 670

Part IV - Annexes

This section contains the annexes for this specification.

- Annex A - Diagrams
- Annex B - UML Keywords
- Annex C - Standard Stereotypes
- Annex D - Component Profile Examples
- Annex E - Tabular Notations
- Annex F - Classifiers Taxonomy
- Annex G - XMI Serialization and Schema
- Annex H - UML Compliance Level XMI Documents

Annex A
(normative)

Diagrams

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.

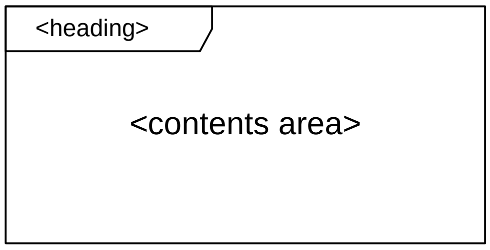


Figure A.1

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.

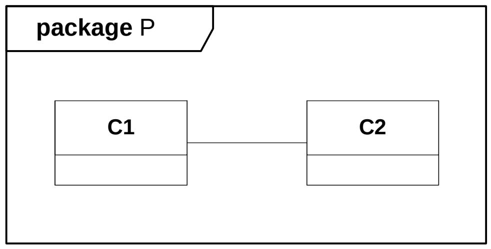


Figure A.2 - Class diagram of package P

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 that 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.

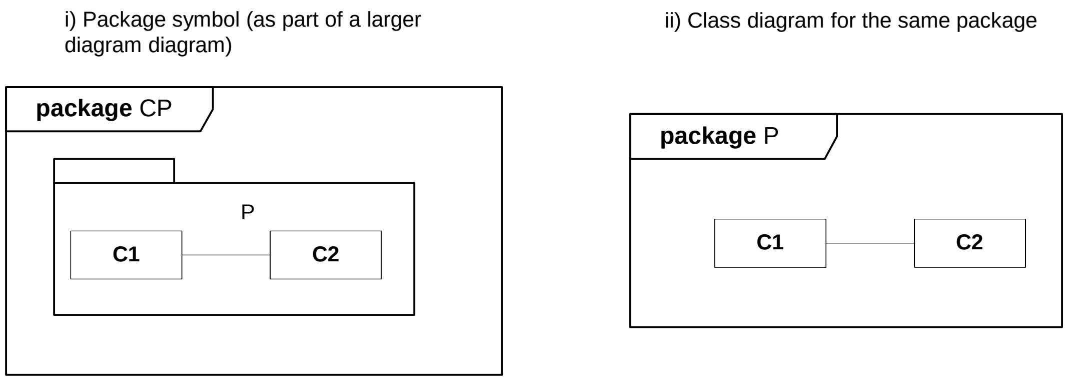

i) Package symbol (as part of a larger diagram diagram)
ii) Class diagram for the same package
package CP
P
C1
C2
package P
C1
C2

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.

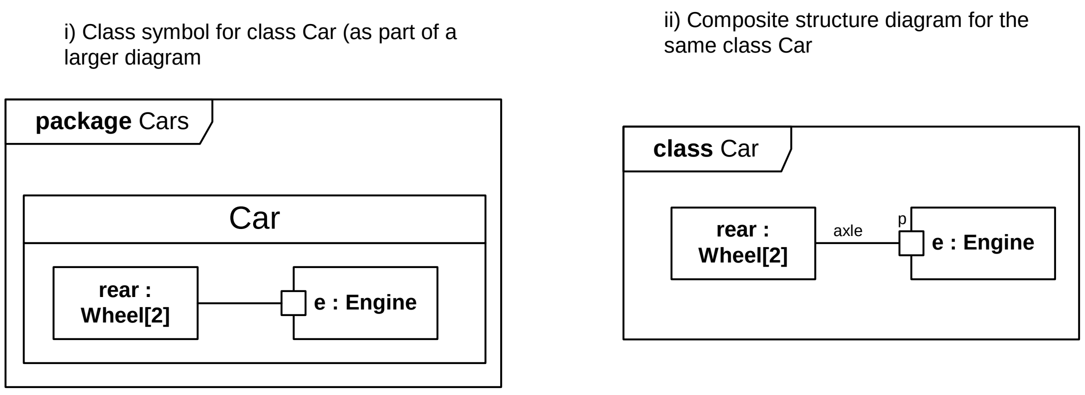

i) Class symbol for class Car (as part of a larger diagram
package Cars
Car
rear : Wheel[2]
e : Engine
ii) Composite structure diagram for the same class Car
class Car
rear : Wheel[2]
axle
p
e : Engine

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.

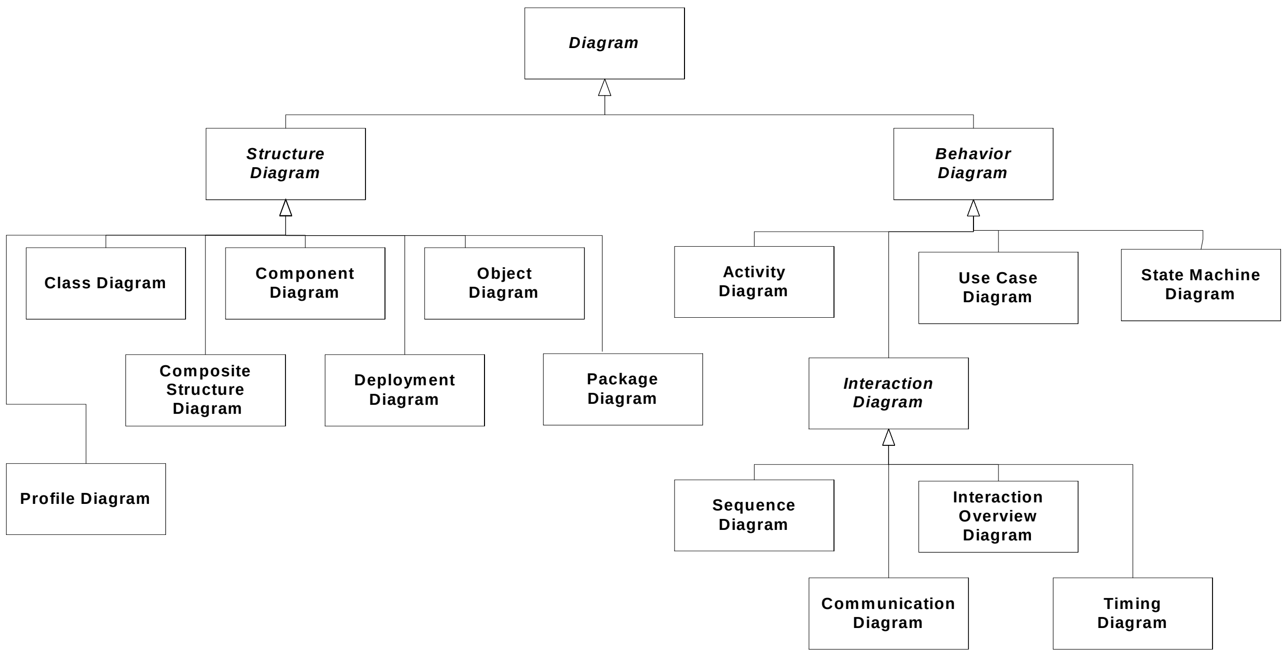


Figure A.5 - The taxonomy of structure and behavior diagram

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.

Please note that 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 is described in the Superstructure clauses as indicated below.

- Activity Diagram - “Activities” on page 295
- Class Diagram - “Classes” on page 23
- Communication Diagram - “Interactions” on page 459
- Component Diagram - “Components” on page 143
- Composite Structure Diagram - “Composite Structures” on page 161
- Deployment diagram - “Deployments” on page 193
- Interaction Overview Diagram - “Interactions” on page 459

- Object Diagram - “Classes” on page 23
- Package Diagram - “Classes” on page 23
- Profile Diagram - “Profiles” on page 653
- State Machine Diagram - “State Machines” on page 525
- Sequence Diagram - “Interactions” on page 459
- Timing Diagram - “Interactions” on page 459
- Use Case Diagram - “Use Cases” on page 587

Annex B
(normative)

Keywords

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 (see Annex C, “Standard Stereotypes”) and, therefore, cannot be used to define any user-defined stereotype.

In UML, keywords are used for four different purposes:

- 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.
- 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.
- 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.
- To indicate a Standard Stereotype (see Annex C, “Standard Stereotypes”). 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 that 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:

1. Not all words appearing between guillemets are necessarily keywords (i.e., reserved words), and
2. words appearing in guillemets do not necessarily represent stereotypes.

If multiple keywords and/or stereotype names apply to the same model element, 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 as 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 since they are likely to lead to confusion.

The keywords currently defined as part of standard UML are specified in Table B.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).
- *Language Unit* identifies the language unit in which the keyword is defined (and, implicitly, the clause in which the keyword is described).
- *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:L<x>," where <x> = 2, or 3, it means that the keyword represents a stereotype that is defined at compliance level. In those cases, the more detailed description of the semantics can be found in Annex C, "Standard Stereotypes."
- *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 B.1 - UML Keywords

Keyword	Language Unit	Metamodel Element	Semantics	Notation Placement
abstraction	Classes	Abstraction	Abstraction	box header
access	Classes	PackageImport	(visibility <> #public)	dashed-line label
activity	Activities	Activity	Activity	box header
actor	Use Cases	Actor	Actor	box header
after	CommonBehaviors	TimeEvent	isRelative = true	Inline label
all	CommonBehaviors	AnyReceiveEvent	Any event	Inline label
apply	Profiles	ProfileApplication	Package::appliedProfile-> collect(ap | ap.importedProfile)	dashed-line label
artifact	Deployments	Artifact	Artifact	box header
artifacts	Deployments	Component		list-box header
at	CommonBehaviors	TimeEvent	isRelative = false	Inline label
attribute	Activities	ActivityPartition:: represents	ActivityPartition::represents ->forAll(r | r.oclIsKindOf(Property))	swimlane header
auxiliary	Classes	Classifier	standard stereotype:L2	box header
build Component	Components	Component	standard stereotype:L3	box header
call	Classes	Usage	standard stereotype:L2	dashed-line label
centralBuffer	Activities	CentralBufferNode	CentralBufferNode	box header
class	Activities	ActivityPartition:: represents	ActivityPartition::represents ->forAll(r | r.oclIsKindOf(Class))	swimlane header
component	Components	Component	Component	box header
create	Classes	Usage	standard stereotype:L2	dashed-line label
create	Classes	BehavioralFeature	standard stereotype:L2 (constructor)	inline label on operation
create	CompositeStructures	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	Activities	DataStoreNode	DataStoreNode	box header
datatype	Classes	DataType	DataType	box header
decisionInput	Activities	Comment	annotatedElement.decisionInput.name = body, or annotatedElement.decisionInput.body = body (for opaque behaviors), or annotatedElement.decisionInput.comment.body = body	note label

Table B.1 - UML Keywords

Keyword	Language Unit	Metamodel Element	Semantics	Notation Placement
decisionInputFlow	Activities	ActivityEdge	target.decisionInputFlow = self	line label
delegate	Components	Connector	Connector::kind = #delegation	connector label
deploy	Deployments	Connector	delegation connector	dashed-line label
deployment spec	Deployments	Deployment Specification	DeploymentSpecification	box header
derive	Classes	Abstraction	standard stereotype:L2	dashed-line label
destroy	Classes	BehavioralFeature	standard stereotype:L2	inline label on operation
device	Deployments	Device	Device	box header
document	Deployments	Artifact	standard stereotype:L2	box header
element access	Classes	ElementImport	not (visibility = #public)	dashed-line label
element import	Classes	ElementImport	visibility = #public	dashed-line label
entity	Components	Component	standard stereotype:L2 (business concept)	box header
enumeration	Classes	Enumeration	Enumeration	box header
executable	Deployments	Artifact	standard stereotype:L2	box header
executionEnvironment	Deployments	ExecutionEnvironment	ExecutionEnvironment	box header
extend	Use Cases	Extend	Extend	dashed-line label
extended	State Machines	Region	Region::extendedRegion ->notEmpty()	between braces
extended	State Machines	StateMachine	StateMachine::redefinedBehavior-> notEmpty()	between braces
external	Activities	ActivityPartition	ActivityPartition::isExternal = true	swimlane header
file	Deployments	Artifact	standard stereotype:L2	box header
focus	Classes	Class	standard stereotype:L2	box header
framework	Classes	Package	standard stereotype:L2	box header
from	CommonBehaviors	Trigger	to show port name	inline label
implement	Components	Component	standard stereotype:L2	box header
implementationClass	Classes	Class	standard stereotype:L2	box header
import	Classes	PackageImport	visibility = #public	dashed-line label
include	Use Cases	Include	Include	dashed-line label
information	Auxiliary::Information-Flows	InformationItem	InformationItem	box header

Table B.1 - UML Keywords

Keyword	Language Unit	Metamodel Element	Semantics	Notation Placement
instantiate	Classes	Dependency	Dependency	dashed-line label
instantiate	Classes	Usage	standard stereotype:L2	dashed-line label
interface	Classes	Interface	Interface	box header
library	Deployments	Artifact	standard stereotype:L2	box header
localPostcondition	Actions	Constraint	Action::localPostcondition	box header
localPrecondition	Actions	Constraint	Action::localPrecondition	box header
manifest	Deployments	Manifestation	Manifestation	dashed-line label
merge	Classes	PackageMerge	PackageMerge	dashed-line label
metaclass	Profiles	Classifier	metaclass being stereotyped	box header
metamodel	Auxiliary::Models	Model	standard stereotype:L3	box header
model	Auxiliary::Models	Model	Model	box header
modelLibrary	Classes	Package	standard stereotype:L2	box header
multicast	Activities	ObjectFlow	ObjectFlow::isMulticast	flow label
multireceive	Activities	ObjectFlow	ObjectFlow::isMultireceive	flow label
occurrence	CompositeStructures	Collaboration	(Behavior::context) from a Collaboration to the owning BehavioredClassifier, indicating that the collaboration represents the use of a classifier.	dashed-line label
postcondition	Activities	Constraint	Behavior::postcondition	box header
precondition	Activities	Constraint	Behavior::precondition	box header
primitive	Classes	PrimitiveType	PrimitiveType	box header
process	Components	Component	standard stereotype:L2	box header
profile	Profiles	Profile	Profile	box header
provided interfaces	Components	Component	Component::provided	list-box header
realization	Classes	Classifier	standard stereotype:L2	box header
realizations	Components	Component	Component::realizations->collect(r | r.realizingClassifier)	list-box header
reference	Profiles	ElementImport	Profile::metaclassReference	dashed-line label
reference	Profiles	PackageImport	Profile::metamodelReference	dashed-line label
refine	Classes	Abstraction	standard stereotype:L2	dashed-line label
representation	Auxiliary::Information-Flows	Classifier	InformationFlow::conveyed	dashed-line label

Table B.1 - UML Keywords

Keyword	Language Unit	Metamodel Element	Semantics	Notation Placement
represents	CompositeStructures	Collaboration	(Behavior::context) from a Collaboration to the owning BehavioredClassifier; collaboration is USED in the classifier	dashed-line label
required interfaces	Components	Component	Component::required	list-box header
responsibility	Classes	Usage	standard stereotype:L2	dashed-line label
script	Deployments	Artifact	standard stereotype:L2	box header
selection	Activities	Behavior	ObjectFlow::selection	box header
selection	Activities	Behavior	ObjectNode::selection	box header
send	Classes	Usage	standard stereotype:L2	dashed-line label
service	Components	Component	standard stereotype:L2	box header
signal	CommonBehaviors	Signal	Signal	box header
singleExecution	Activities	Activity	Activity::isSingleExecution = true	inside box
source	Deployments	Artifact	standard stereotype:L2	box header
specification	Components	Classifier	standard stereotype:L2	box header
statemachine	State Machines	BehavioredClassifier:: ownedBehavior	BehavioredClassifier::ownedBehavior. oclIsKindOf(StateMachine)	box header
stereotype	Profiles	Stereotype	Stereotype	box header
structured	Activities	StructuredActivity Node	StructuredActivityNode	box header
substitute	Classes	Substitution	Substitution	dashed-line label
subsystem	Components	Component	standard stereotype:L2	box header
systemModel	Auxiliary::Models	Model	standard stereotype:L3	box header
trace	Classes	Abstraction	standard stereotype:L2	dashed-line label
transformation	Activities	Behavior	ObjectFlow::transformation	box header
type	Classes	Class	standard stereotype:L2	box header
use	Classes	Usage	Usage	dashed-line label
utility	Classes	Class	standard stereotype:L2	box header
when	CommonBehaviors	ChangeEvent	ChangeEvent::changeExpression	inline label

Annex D

Component Profile Examples

This annex describes example profiles for J2EE/Enterprise Java Beans (EJB), NET, COM and CORBA Component Model (CCM) components. These profiles are not meant to be either normative or complete, but are provided as an illustration of how UML 2.0 can be customized to model component architectures.

D.1 J2EE/EJB Component Profile Example

Table D.1 shows an example profile for Enterprise Java Beans components.

Table D.1 - Example Profile for Enterprise Java Beans

Stereotype	Base Class	Parent	Tags	Constraints	Description
EJBEntityBean «EJBEntityBean»	Component	N/A	N/A	N/A	Indicates that a Component represents an EJB Entity Bean, a component that manages the business logic of an application.
EJBSessionBean «EJBSessionBean»	Component	N/A	N/A	N/A	Indicates that a Component represents an EJB Session Bean, a component that processes transactions.
EJBMessage DrivenBean «EJBMessage DrivenBean»	Component	N/A	N/A	N/A	Indicates that a Component represents an EJB Message-Driven Bean, a component that handles messages, and is invoked on the arrival of a message.
EJBHome «EJBHome»	Interface	N/A	N/A	N/A	Indicates that the Interface is an EJB Home interface that supports lifecycle and class-level operations.
EJBRemote «EJBRemote»	Interface	N/A	N/A	N/A	Indicates that the Interface is an EJB Remote interface that supports business-specific operations.
EJBService «EJBService»	Interface	N/A	N/A	N/A	Indicates that the Interface is an EJB Service interface that is exposed as a webservice definition.
EJBCreate «EJBCreate»	Method	N/A	N/A	N/A	Indicates that the Method is an EJB Create Method that facilitates a create operation.
EJBBusiness «EJBBusiness»	Method	N/A	N/A	N/A	Indicates that the Method is an EJB instance-level method that supports the business logic of the EJB associated with the Remote interface.
EJBSecurityRoleRef «EJBSecurity RoleRef»	Association	N/A	N/A	N/A	Indicates an Association between an EJB client and an EJB Role Name Reference supplier.

Table D.1 - Example Profile for Enterprise Java Beans

Stereotype	Base Class	Parent	Tags	Constraints	Description
EJBRoleName «EJBRoleName»	Actor	N/A	N/A	N/A	Indicates the name of a security role used in the definitions of method permissions, etc.
EJBRoleNameRef «EJBRoleNameRef»	Actor	N/A	N/A	N/A	Indicates the name of a security role reference used programmatically in an EJB's source code and mapped to an EJB Role Name.
JavaSourceFile «JavaSourceFile»	Artifact	«file»	N/A	N/A	Indicates that the Artifact represents a Java source file.
JAR «JAR»	Artifact	«file»	N/A	N/A	Indicates that the Artifact represents a JAR (Java ARchive) file.
EJBQL «EJBQL»	Expression	N/A	N/A	N/A	Indicates that the expression conforms to the EJB Query Language syntax.

D.2 COM Component Profile Example

Table D.2 shows an example profile for COM components.

Table D.2 - Example Profile for COM Components

Stereotype	Base Class	Parent	Tags	Constraints	Description
COMCoClass «COMCoClass»	Component	N/A	N/A	N/A	Indicates a Component specification in COM (as defined in a type library). Specifies all externally visible aspects of a component.
COMAtlClass «COMAtlClass»	Component	N/A	N/A	N/A	Indicates a component implementation class using the ATL framework. An ATL class realizes a CoClass and provides its implementation.
COMInterface «COMInterface»	Interface	N/A	N/A	N/A	Indicates a component interface. COMInterfaces are offered or required by CoClasses and realized by COMAtlClasses.
COMConnectionPoint «COMConnectionPoint»	Dependency	N/A	N/A	N/A	Indicates that a component publishes an interface for subscribers. It is a special kind of a provided interface.
COMTypeLibrary «COMTypeLibrary»	Package	N/A	N/A	N/A	Indicates a packaging of COM types (COMCoClasses, COMInterface) to be deployed as a library.
COMTLB «COMTLB»	Artifact	N/A	N/A	N/A	Indicates a (runtime) component specification file (similar to a deployment descriptor).
COMExe «COMExe»	Artifact	«file»	N/A	N/A	An artifact that deployes an executable COM server application.
COMDLL «COMDLL»	Artifact	«file»	N/A	N/A	An artifact that deploys a component library.

D.3 .NET Component Profile Example

Table D.3 shows an example profile for .NET components.

Table D.3 - Example Profile for .NET Components

Stereotype	Base Class	Parent	Tags	Constraints	Description
NetComponent «NetComponent»	Component	N/A	N/A	N/A	Indicates that a Component represents a component in the .NET framework.
NETProperty «NETProperty»	Property	N/A	N/A	N/A	Indicates a Property offered by a component for public get/set operations.
NETAssembly «NETAssembly»	Package	N/A	N/A	N/A	Indicates a .NET assembly, which is a runtime packaging entitiy for components and other type definitions.
MSI «MSI»	Artifact	N/A	N/A	N/A	Indicates a component self installer file.
DLL «DLL»	Artifact	«file»	N/A	N/A	Indicates a portable executable (PE) of type DLL.
EXE «EXE»	Artifact	«file»	N/A	N/A	Indicates a portable executable (PE) of type EXE.

D.4 CCM Component Profile Example

Table D.4 shows an example profile for CCM components.

Table D.4 - Example Profile for CCM Components

Stereotype	Base Class	Parent	Tags	Constraints	Description
CCMEntity «CCMEntity»	Component	N/A	N/A	N/A	Indicates that a Component represents an Entity CC, a component that manages the business logic of an application.
CCMSession «CCMSession»	Component	N/A	N/A	N/A	Indicates that a Component represents a Session CC, a component that processes transactions.
CCMService «CCMService»	Component	N/A	N/A	N/A	Indicates that a Component represents a Service CC, a component that models single independent execution of an “operation.”
CCMProcess «CCMService»	Component	N/A	N/A	N/A	Indicates that a Component represents a service CC, a component realizing a long-lived business process.
CCMHome «CCMHome»	Interface	N/A		Must have a manages dependency.	Indicates that the Interface is a CCMHome that provides factory and finder methods for a particular CC.
CCMManages «CCMManages»	Dependency	N/A		Always between a CCMHome and some CCM component.	Associates a CCM home with the CC component it manages.
CCMFactory «CCMFactory»	Operation	N/A	N/A	Operation is owned by CCMHome.	Indicates that the Operation is a CCMhome Create Method that facilitates a create operation.

Table D.4 - Example Profile for CCM Components

Stereotype	Base Class	Parent	Tags	Constraints	Description
CCMFinder «CCMFinder»	Operation	N/A	N/A	Operation is owned by CCMHome.	Indicates that the operation is a finder method of a CCMHome.
CCMProvided «CCMProvided»	Port	N/A	N/A	Port owns a single provided interface.	Indicates a port that models a CC facet.
CCMRequired «CCMRequired»	Port	N/A	N/A	Port owns a single provided interface.	Indicates a port that models a CC Receptacle.
CCMPackage «CCMPackage»	Artifact	N/A	N/A	N/A	Indicates an artifact that deploys a set of CCs.
CCMComponentDescriptor «CCMComponent Descriptor»	Artifact	«file»	N/A	N/A	Indicates that the Artifact represents a CCMComponentDescriptor.
CCMSoftPkgDescriptor «CCMComponent Descriptor»	Artifact	«file»	N/A	N/A	Indicates that the Artifact represents a CCM softPkg descriptor.

Annex C

(normative)

Standard Stereotypes

This annex describes the predefined standard stereotypes for UML. The standard stereotypes are specified in two separate system-defined profiles, corresponding to the top two compliance levels of UML (L2 and L3). These profiles can be applied to a user model just like any other profile. However, it is not necessary to include an explicit profile definition in such cases as it is assumed that such definitions are included (implicitly or explicitly) within any tool that is compliant with the standard. Of course, a tool need only support the profile that is consistent with its level of standard compliance.

The stereotypes belonging to the profile are described using a compact tabular form rather than graphically. 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 except that the first letter of each is capitalized. The second column identifies the language unit of the stereotype. The third column identifies the metaclass to which the stereotype applies and the last column provides a description of the meaning of the stereotype.

C.1 StandardProfileL2

Name	Language Unit	Applies to	Description
«auxiliary»	Classes::Kernel	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».
«call»	Classes:: Dependencies	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»	Dependencies	Usage	A usage dependency denoting that the client classifier creates instances of the supplier classifier.
«create»	Classes::Kernel	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»	Classes:: Dependencies	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»	Classes::Kernel	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»	Deployments:: Artifacts	Artifact	A generic file that is not a «source» file or «executable». Subclass of «file».
«entity»	Components:: BasicComponents	Component	A persistent information component representing a business concept.
«executable»	Deployments:: Artifacts	Artifact	A program file that can be executed on a computer system. Subclass of «file».
«file»	Deployments:: Artifacts	Artifact	A physical file in the context of the system developed.

«focus»	Classes::Kernel	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»	Classes::Kernel	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»	Components:: BasicComponents	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»	Classes::Kernel	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. Note that the physical attributes and associations of the Implementation class do not have to be the same as those of any Classifier it realizes and that the Implementation Class may provide methods for its operations in terms of its physical attributes and associations. See also: «type».
«instantiate»	Classes:: Dependencies	Usage	A usage dependency among classifiers indicating that operations on the client create instances of the supplier.
«library»	Deployments:: Artifacts	Artifact	A static or dynamic library file. Subclass of «file».
«metaclass»	Classes::Kernel	Class	A class whose instances are also classes.

«modelLibrary»	Classes::Kernel	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»	Components:: BasicComponents	Component	A transaction based component.
«realization»	Classes::Kernel	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»	Classes:: Dependencies	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»	Classes::Kernel	Usage	A contract or an obligation of an element in its relationship to other elements.
«script»	Deployments:: Artifacts	Artifact	A script file that can be interpreted by a computer system. Subclass of «file».
«send»	Classes:: Dependencies	Usage	A usage dependency whose source is an operation and whose target is a signal, specifying that the source sends the target signal.
«service»	Components:: BascComponents	Component	A stateless, functional component (computes a value).
«source»	Deployments:: Artifacts	Artifact	A source file that can be compiled into an executable file. Subclass of «file».

«specification»	Classes::Kernel	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»	Components:: BasicComponents	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».
«trace»	Classes:: Dependencies	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. Since 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»	Classes::Kernel	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»	Classes::Kernel	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.

C.2 StandardProfileL3

Name	Subpackage	Applies to	Description
«buildComponent»	Components:: BasicComponents	Component	A collection of elements defined for the purpose of system level decelopment activities, such as compilation and versioning.
«metamodel»	AuxilliaryConstructs:: Models	Model	A model that specifies the modeling concepts of some modeling language (e.g., a MOF model). See <<metaclass>>.
«systemModel»	AuxilliaryConstructs:: Models	Model	A SystemModel is a stereotyped model that contains a collection of models of the same physical system. A systemModel also contains all relationships and constraints between model elements contained in different models.

Changes from previous UML

The following table lists predefined standard elements for UML 1.x that are now obsolete.

Standard Element Name	Applies to Base Element
«access»	Permission
«appliedProfile»	Package
«association»	AssociationEnd
«copy»	Flow
«create»	CallEvent
«create»	Usage
«destroy»	CallEvent
«facade»	Package
«friend»	Permission
«invariant»	Constraint
«local»	AssociationEnd
«parameter»	AssociationEnd
«postcondition»	Constraint
«powertype»	Class
«precondition»	Constraint
«profile»	Package
«realize»	Abstraction
«requirement»	Comment
«self»	AssociationEnd
«signalflow»	ObjectFlowState

«stateInvariant»	Constraint
«stub»	Package
«table»	Artifact
«thread»	Classifier
«topLevel»	Package

Annex E

(normative)

Tabular Notations

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 appendix mostly describes tabular notations for sequence diagrams, the approach may also be applied to other kinds of behavioral diagrams.

E.1 Tabular Notation for Sequence Diagrams

This sub clause 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. See definition of weak order (sub clause 34.1 in the U2 partners submission.) 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

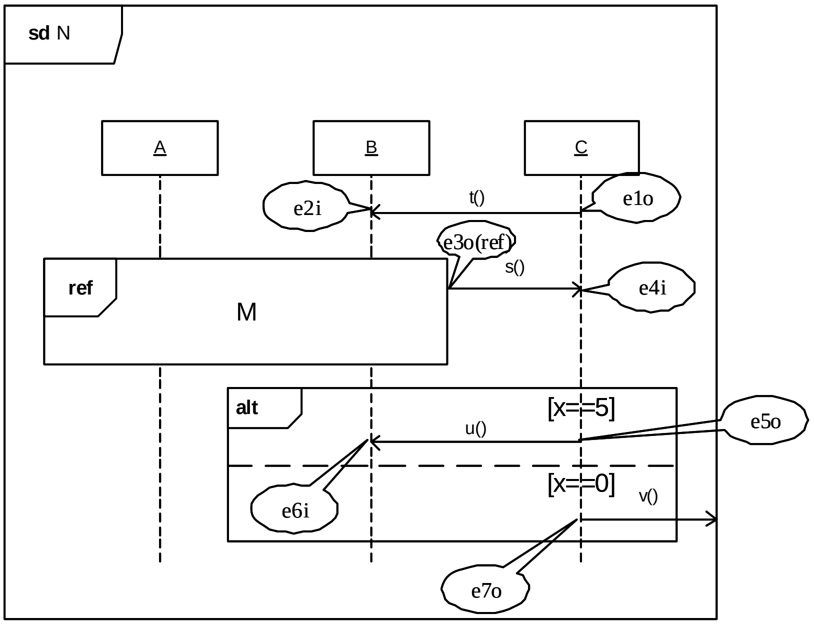


Figure E.1 - Sequence diagram enhanced with identification of the Event occurrences

Table E.1 - Interaction Table describing Figure E-1

Lifeline Class	Lifeline Inst	Constraint	Message Sending Class	Message Sending Instance	Diag ID	Generated Instance Name	Sequence No	Weak Order	Mesg Name	Para meter	Return Value	Message Receiving Class	Message Receive Instance	Othe 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(r
	C	alt [1]x==5			N.sq			e5o	u				B	e6i
	B	alt[1] x==5		B	N.sq			e6i	u					e5o
	C	alt[2] x==o			N.sq			e7o	v				out_v	

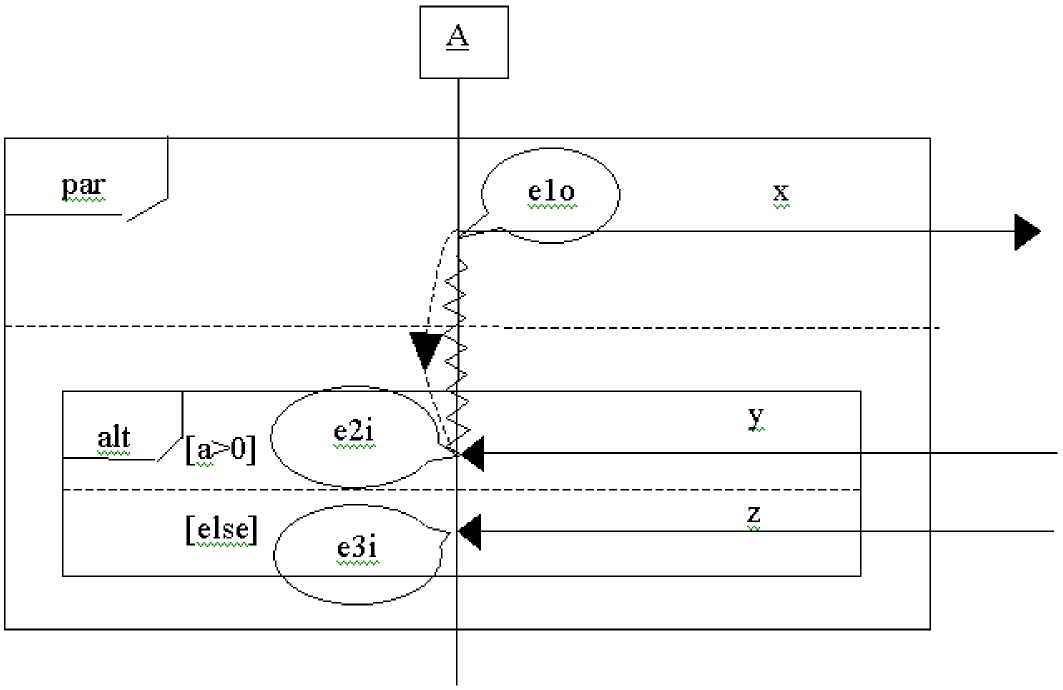
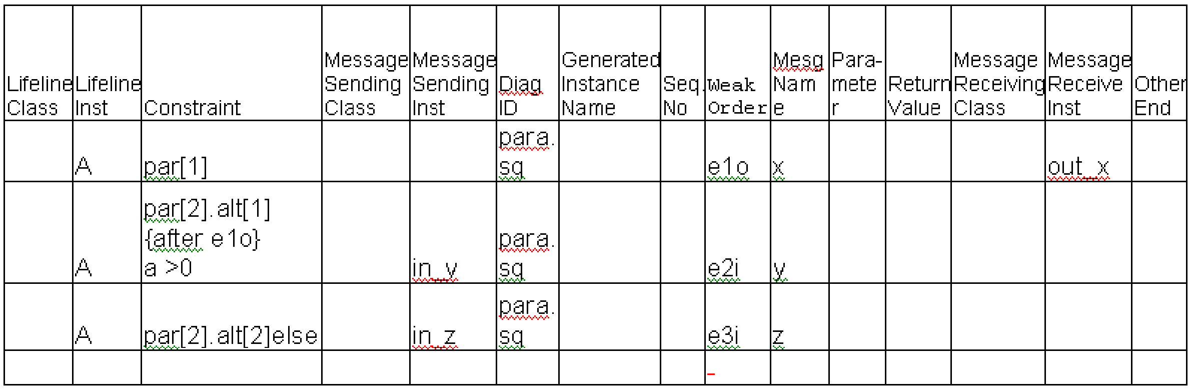


Figure E.2 - Sequence diagram with guards, parallel composition and alternatives

Table E.2 - Interaction Table for Figure E-2



Lifeline Class	Lifeline Inst	Constraint	Message Sending Class	Message Sending Inst	Diag ID	Generated Instance Name	Seq. No	Weak Order	Mesg Name	Para-meter	Return Value	Message Receiving Class	Message Receive Inst	Other End
	A	par[1]			para.sq			e1o	x				out_x	
	A	par[2].alt[1] {after e1o} a >0		in_y	para.sq			e2i	y					
	A	par[2].alt[2]else		in_z	para.sq			e3i	z					

E.2 Tabular Notation for Other Behavioral Diagrams

The approach for defining tabular notation for sequence diagrams should also be applicable to other major behavioral diagram types, such as state machine diagrams and activity diagrams.

Annex F

(normative)

Classifiers Taxonomy

The class inheritance hierarchy of the Classifiers in the UML 2.0 Superstructure is shown in Figure F.1. The root of the Classifier hierarchy is the Classifier defined in the Classes::Kernel package, and includes numerous merge increments and subclasses and their increments. The Classifier hierarchy includes package references, so that readers can refer to their definitions in the appropriate packages.

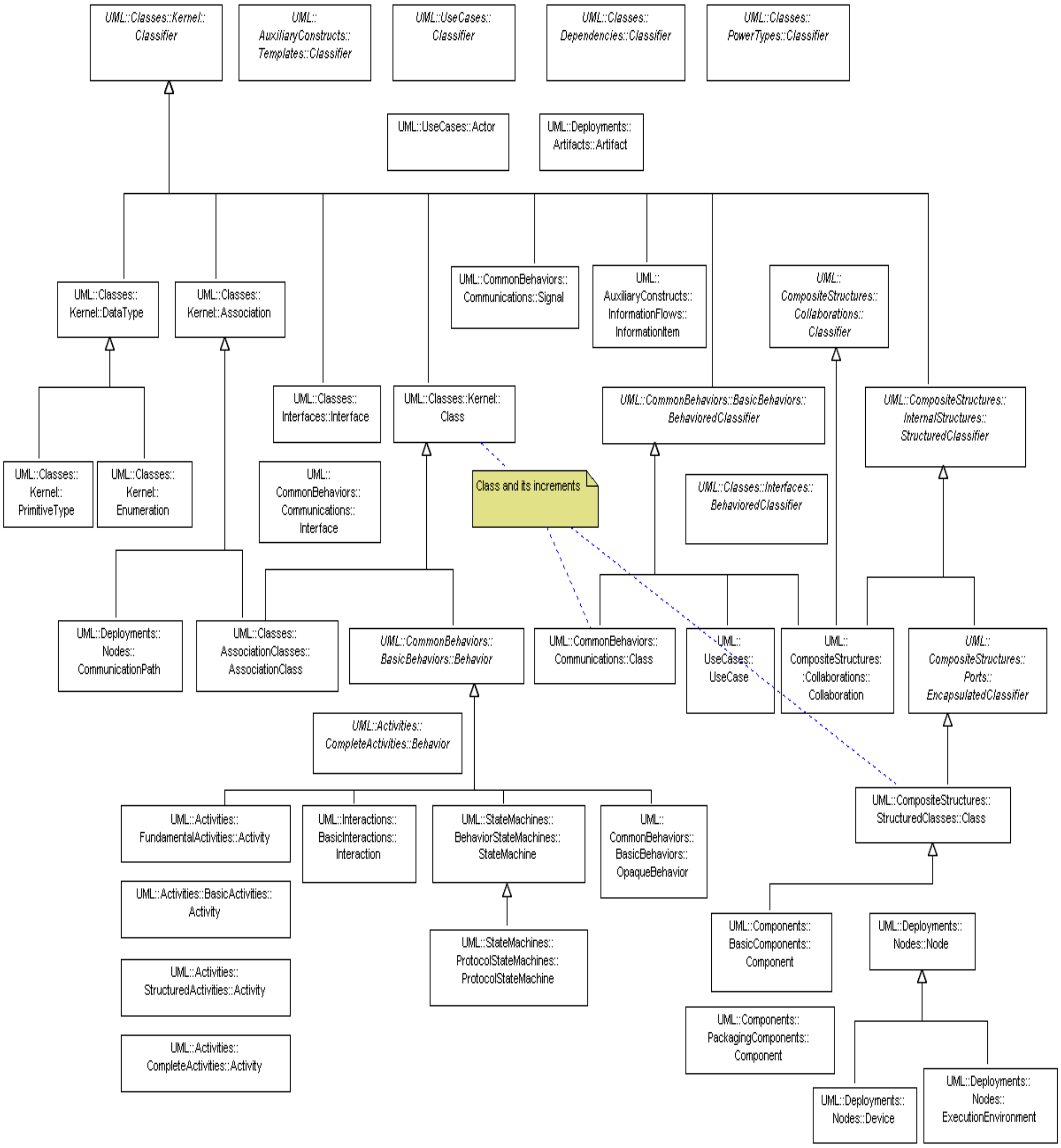

UML::Classes::Kernel:: Classifier
UML:: AuxiliaryConstructs:: Templates::Classifier
UML::UseCases:: Classifier
UML::Classes:: Dependencies::Classifier
UML::Classes:: PowerTypes::Classifier
UML::UseCases::Actor
UML::Deployments:: Artifacts::Artifact
UML::Classes:: Kernel::DataType
UML::Classes:: Kernel::Association
UML::CommonBehaviors:: Communications::Signal
UML:: AuxiliaryConstructs:: InformationFlows:: InformationItem
UML:: CompositeStructures:: Collaborations:: Classifier
UML::Classes:: Interfaces::Interface
UML::Classes::Kernel:: Class
UML::CommonBehaviors::BasicBehaviors:: BehavioredClassifier
UML::CompositeStructures:: InternalStructures:: StructuredClassifier
UML::Classes:: Kernel:: PrimitiveType
UML::Classes:: Kernel:: Enumeration
UML:: CommonBehaviors:: Communications:: Interface
Class and its increments
UML::Classes::Interfaces:: BehavioredClassifier
UML::Deployments:: Nodes:: CommunicationPath
UML::Classes:: AssociationClasses:: AssociationClass
UML::CommonBehaviors:: BasicBehaviors::Behavior
UML::CommonBehaviors:: Communications::Class
UML:: UseCases:: UseCase
UML:: CompositeStructures: :Collaborations:: Collaboration
UML:: CompositeStructures:: Ports:: EncapsulatedClassifier
UML::Activities:: CompleteActivities::Behavior
UML::Activities:: FundamentalActivities::Activity
UML::Interactions:: BasicInteractions:: Interaction
UML::StateMachines:: BehaviorStateMachines:: StateMachine
UML:: CommonBehaviors:: BasicBehaviors:: OpaqueBehavior
UML::CompositeStructures:: StructuredClasses::Class
UML::Activities::BasicActivities:: Activity
UML::StateMachines:: ProtocolStateMachines:: ProtocolStateMachine
UML::Components:: BasicComponents:: Component
UML::Deployments:: Nodes::Node
UML::Activities:: StructuredActivities::Activity
UML::Components:: PackagingComponents:: Component
UML::Activities:: CompleteActivities::Activity
UML::Deployments:: Nodes::Device
UML::Deployments:: Nodes:: ExecutionEnvironment

Figure F.1 - Classifier hierarchy for UML 2.0 Superstructure

Annex G

(normative)

XMI Serialization and Schema

UML 2 Superstructure models are serialized in XMI 2.1 according to the rules specified by the MOF 2.0/XMI Mapping Specification, v2.1 (OMG document formal/05-09-01).

As is common policy for OMG, the normative representation of MOF 2 and UML 2 models is an XMI file since this provides the level of complete and processable definition that is not always possible or practical across a set of diagrams. These XMI documents are all CMOF documents since all the MOF 2 and UML 2 metamodels are described using CMOF. The normative XMI document for Superstructure consists of a single XMI document that contains all of the packages described in the Superstructure specification organized the same as the specification.

According to the UML 2 specification and the definition of PackageMerge semantics, XMI files containing package merges are semantically equivalent to the same XMI files with the package merges merged away. The UML 2 Superstructure normative model is expressed in this annex as an unmerged XMI document matching the rest of the specification. It is assumed that this XMI document will primarily be used to specify the normative packages and their contents, and will generally be used by tool vendors wishing to implement package merge and create either UML 2 compliance levels, or extensions thereof.

Non-normative merged convenience documents and normative xmi files are provided in OMG document ptc/06-10-06.

The non-normative merged XMI documents for each compliance level are identified in Annex H. These documents will be useful for tool vendors wishing to validate instances of UML2 models for specific compliance levels against the XMI defining the contents of that compliance level.

Annex H

(normative)

UML Compliance Level XMI Documents

The normative XMI document for UML2 Superstructure is given in Annex G. However, there are also non-normative convenience documents that will be quite useful to tool vendors. Consider compliance level L2. XMI documents that are representations of L2 UML2 models could be validated against the normative unmerged metamodel document which includes the packages for L2 (as well as the others). But these XMI documents have to be instances of the merged L2 metamodel and, since it is derived from the package merge rules, there is not a normative XMI file for this. To facilitate interoperability, and to eliminate the need for tool vendors to implement package merge in order to produce XMI metamodel documents for the UML2 compliance levels, the XMI documents for each compliance level are included in this annex. An XMI document for a compliance level is the XMI representation of the merged CMOF metamodel for that compliance level.

XML uses XSD to validate instances of XML documents, and the MOF 2.0/XMI Mapping Specification, v2.1 specifies how these schema documents are derived for a given CMOF model. However, XSD cannot capture all of the structure, constraints, or semantics of a UML2 compliance level, so vendors may choose to use MOF reflection or XMI metamodel documents to provide more complete validation.

Non-normative merged convenience documents and normative xmi files for this specification are located at: http://www.omg.org/spec/UML/ and include these files:

- http://www.omg.org/spec/UML/20061001/uml-L0-model.xmi
- http://www.omg.org/spec/UML/20061001/uml-LM-model.xmi
- http://www.omg.org/spec/UML/20061001/uml-L1-model.xmi
- http://www.omg.org/spec/UML/20061001/uml-L2-model.xmi
- http://www.omg.org/spec/UML/20061001/uml-L3-model.xmi
- http://www.omg.org/spec/UML/20061001/uml-L0-model.merged.xmi
- http://www.omg.org/spec/UML/20061001/uml-LM-model.merged.xmi
- http://www.omg.org/spec/UML/20061001/uml-L1-model.merged.xmi
- http://www.omg.org/spec/UML/20061001/uml-L2-model.merged.xmi
- http://www.omg.org/spec/UML/20061001/uml-L3-model.merged.xmi

INDEX