Date: August 2005

Unified Modeling Language: Superstructure

version 2.0
formal/05-07-04

OBJECT MANAGEMENT GROUP

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 *UML 2.0: Superstructure* is the second of two complementary specifications that represent a major revision to the Object Management Group's Unified Modeling Language (UML), for which the most current version is UML v1.4. The first specification, which serves as the architectural foundation for this specification, is the *UML 2.0: Infrastructure*.

This *UML 2.0: Superstructure* defines the user level constructs required for UML 2.0. It is complemented by *UML 2.0: Infrastructure* which defines the foundational language constructs required for UML 2.0. The two complementary specifications constitute a complete specification for the UML 2.0 modeling language.

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 107). 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 4.

For this reason, all compliance levels are defined as extensions to a single core "UML" package that defines the common namespace shared by all the compliance levels. Level 0 is defined by the top-level metamodel shown in Figure 2.1. In this model, "UML" is originally an empty package that simply merges in the contents of the Basic package from the UML Infrastructure. This package, contains elementary concepts such as Class, Package, DataType, Operation, etc. (see the *UML 2.0 Infrastructure* specification for the complete list of contents).

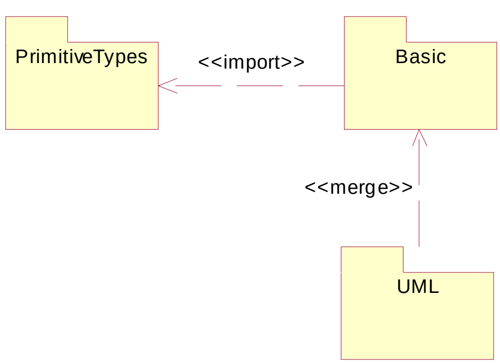


Figure 2.1 - Level 0 package diagram

At the next level (Level 1), the contents of the "UML" package, now including the packages merged into Level 0 and their contents, are extended with additional packages as shown in Figure 2.2 on page 3. 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 6.

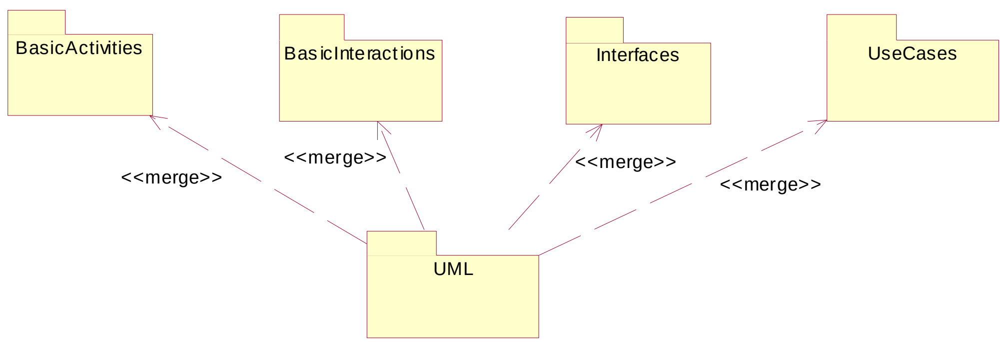


Figure 2.2 - Level 1 top-level package merges

Level 2 adds further language units and extensions to those provided by the Level 1. Once again, the package “UML” now incorporates the complete Level 1 shown in Figure 2.3 on page 3. The actual language units and packages included at this level of compliance are listed in Table 2.4 on page 6.

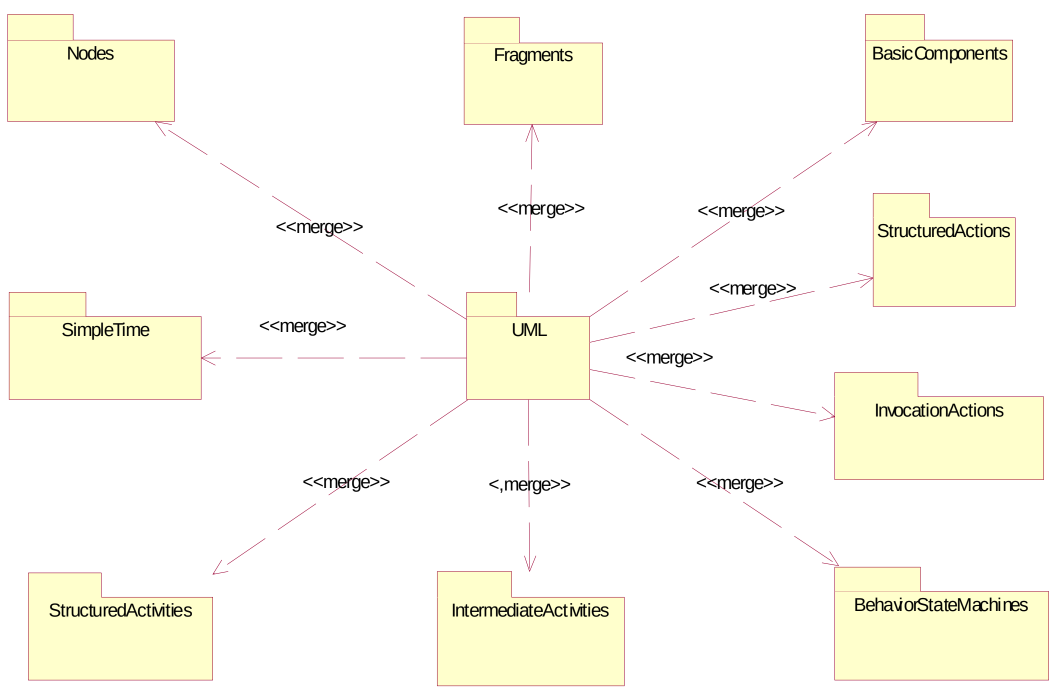


Figure 2.3 - Level 2 top-level package merges

Finally, Level3, incorporating the full UML definition, is shown in Figure 2.4 on page 4. Its contents are described in Table 2.5 on page 7.

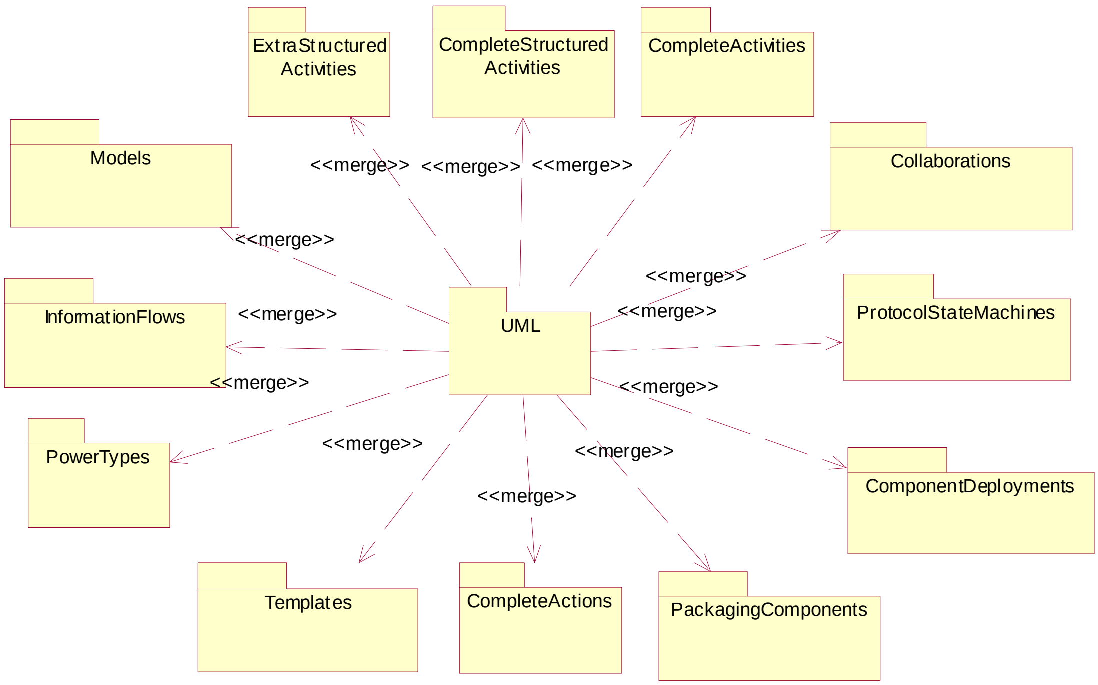


Figure 2.4 - Level 3 top-level package merges

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:

- *Abstract syntax compliance*. For a given compliance level, this entails:
  - compliance with the metaclasses, their structural relationships, and any constraints defined as part of the merged UML metamodel for that compliance level and,
  - the ability to output models and to read in models based on the XMI schema corresponding to that compliance level.
- *Concrete syntax compliance*. For a given compliance level, this entails
  - Compliance to the notation defined in the "Notation" sections 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” subsections 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)**

Table 2.2 Example feature support statement

Feature Support Statement					
Language Unit	**Packages**	**Abstract Syntax**	**Concrete Syntax**	**Semantics**	**Presentation Options**
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 chapters 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
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

Table 2.4 Metamodel packages added in Level 2

Language Unit	Metamodel Packages
General Behavior	CommonBehaviors::Communications
	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 Changes to Adopted OMG Specifications

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

6.2 Architectural Alignment and MDA Support

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

It is the intent that the unified MOF 2.0 Core specification must be architecturally aligned with the *UML 2.0: Infrastructure* part of this specification. Similarly, the unified UML 2.0 Diagram Interchange specification must be architecturally aligned with the *UML 2.0: 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 *UML 2.0: Infrastructure* Appendix B, “Support for Model Driven Architecture.”

6.3 On the Run-Time Semantics of UML

The purpose of this section 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).

6.3.1 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 423). 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.

6.3.2 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.)

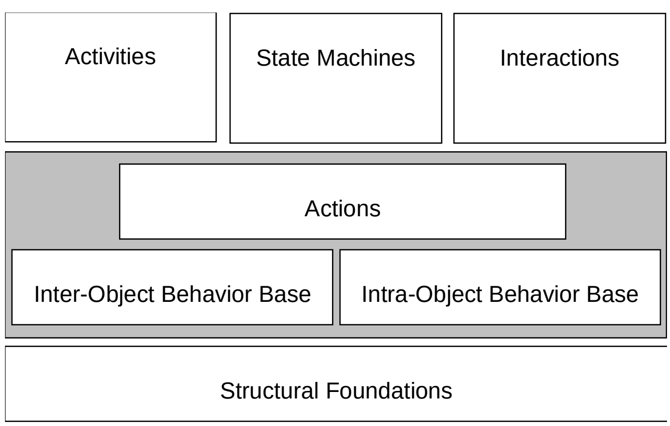


Figure 6.1 - A schematic of the UML semantic areas and their dependencies

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.

6.3.3 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 chapter on page 407. 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).

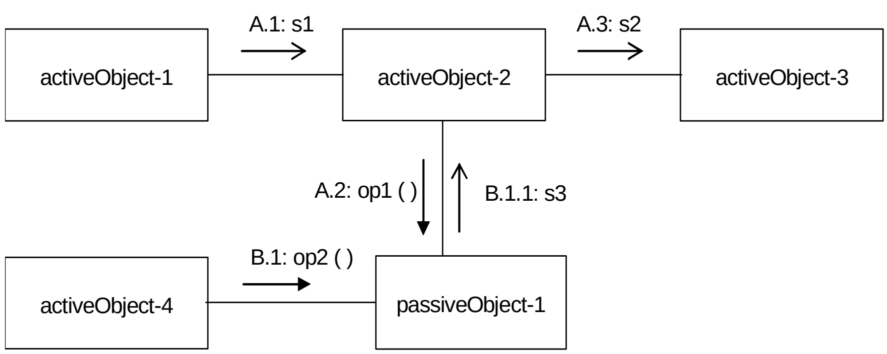


Figure 6.2 - Example illustrating the basic causality model of UML

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 237). This purely "procedural" or "process" model can be used by itself or in conjunction with the object-oriented model of the previous example.

6.3.4 Semantics Descriptions in the Specification

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

The structural foundations are mostly covered in two chapters. The elementary level is mostly covered in Chapter 7, where the root concepts of UML are specified. In particular, the sections 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 Chapter 9 (Composite Structures), with most of the information related to semantics contained in sections 9.3.12 (Property concept) and 9.3.13 (StructuredClassifier). In addition, the introduction to this chapter 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 Chapter 13 and in section 13.3.2 in particular).

Inter-object behavior is covered in three separate chapters. The basic semantics of communications actions are described in the introduction to Chapter F (Actions) and, in more detail, in the sections 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 chapter (sections 14.3.20 and 14.3.21), while the concepts of events and triggers are defined in the Communications package of CommonBehaviors (Chapter 13). Occurrence specifications are defined in section 14.3.25 of the Interactions chapter. 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 Chapter 13.

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

6.4 The UML Metamodel

6.4.1 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.

6.4.2 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 chapter, 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.”

6.5 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 Chapter 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 I. Structure” 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 chapters 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.

6.5.1 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 chapters 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 chapter and all concepts related to the activities modeling capability are in the Activities chapter. The Capability chapters in each part are presented in alphabetical order.

Within each chapter, there is first a brief informal description of the capability described in that chapter. This is followed by a section 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” section follows the abstract syntax section. This section 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 section in most chapters 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 sections corresponding to different aspects. In cases where a given aspect does not apply, its sub section may be omitted entirely from the class description. The following sub sections 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 section. 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 section 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 section. 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 section or until the next sub package increment heading is encountered.

- This convention for describing sub package increments is applied to all other sub sections related to the concept.
- The “Attributes” sub section 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 section 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 section 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 section 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 section 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 section supplements the “Notation” section by providing alternative representations for the concept or its parts. Users have the choice to use either the forms described in this sub section or the forms described in the “Notation” sub section.
- “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 section, 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.

6.5.2 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.
- 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.
- 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.

6.6 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

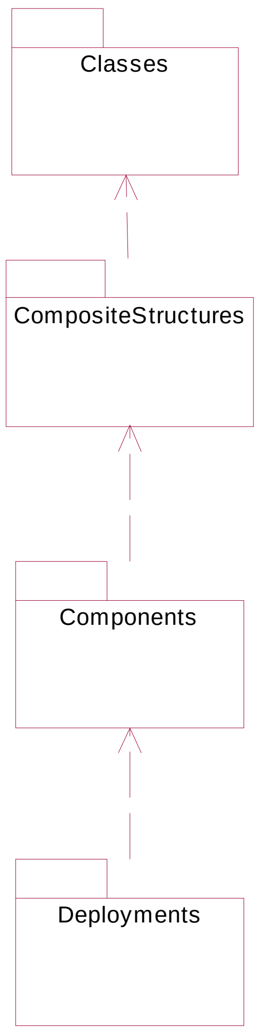
- Lockheed Martin
- MEGA International
- Mercury Computer
- Motorola
- MSC.Software
- Northeastern University
- Oracle
- Popkin Software
- Proforma
- Project Technology
- Sims Associates
- SOFTEAM
- Sun Microsystems
- Syntropy Ltd.
- Telelogic
- Thales Group
- TNI-Valiosys
- Unisys
- University of Kaiserslautern
- 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, and Ian Wilkie.

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 chapters, 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 2.0 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 primarily reuses the *Constructs* and *Abstractions* packages of the InfrastructureLibrary.

The reuse is accomplished by merging *Constructs* with the relevant subpackages of *Abstractions*. 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 chapter; the text from the infrastructure library is repeated here.

It should also be noted that while *Abstractions* contained several subpackages, *Kernel* is a flat structure that like *Constructs* only contains metaclasses. 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::Core are the following:

- Abstractions::Instances
- Abstractions::MultiplicityExpressions
- Abstractions::Literals
- Abstractions::Generalizations
- Constructs

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

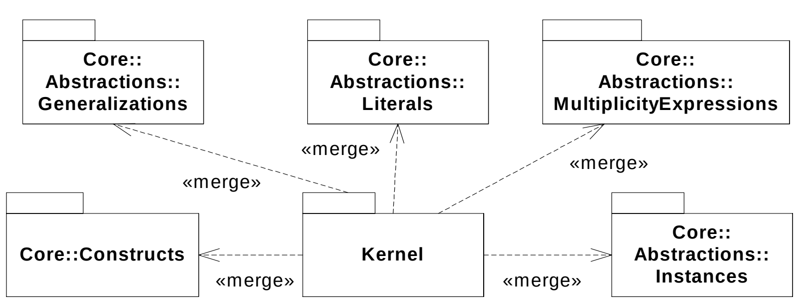


Figure 7.1 - InfrastructureLibrary packages that are merged by Kernel (all dependencies in the picture represent package merges)

7.2 Abstract Syntax

Figure 7.2 shows the package dependencies of the Kernel packages.

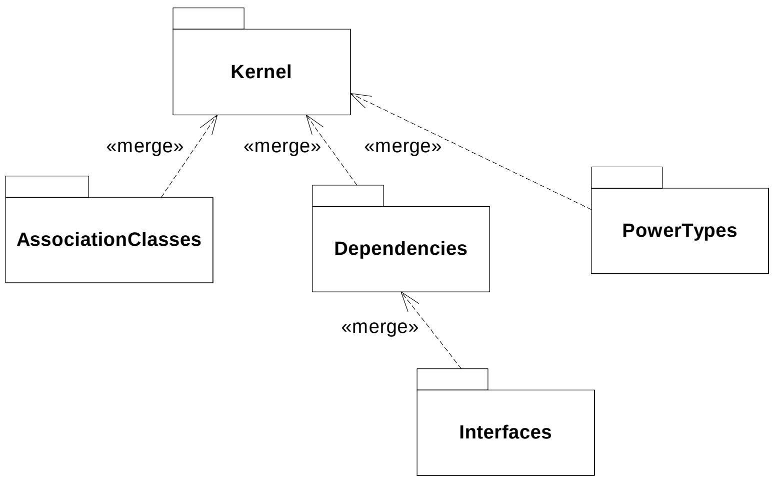


Figure 7.2 - Subpackages of the Classes package and their dependencies

Package Kernel

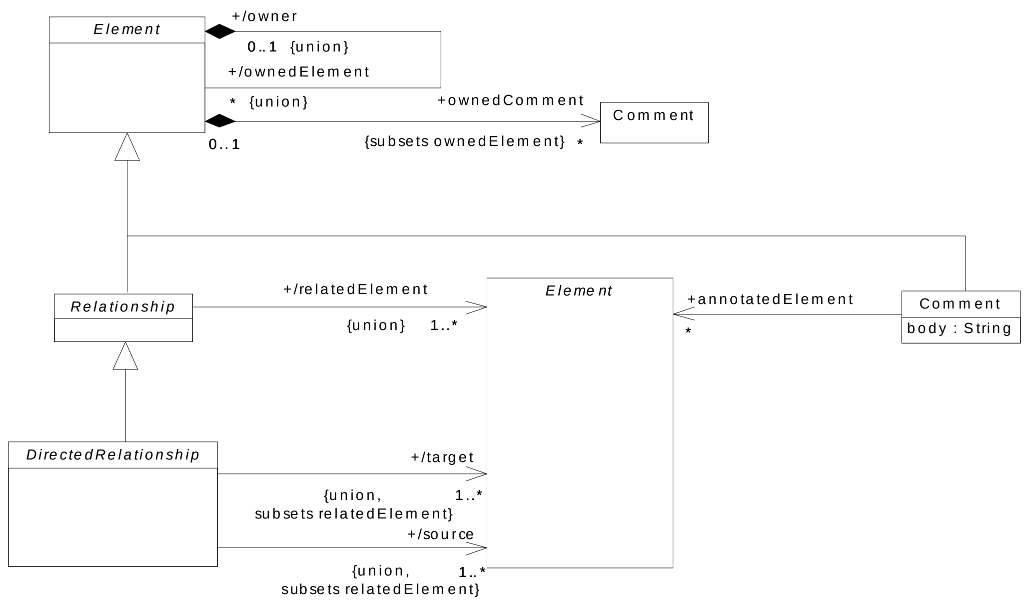


Figure 7.3 - Root diagram of the Kernel package

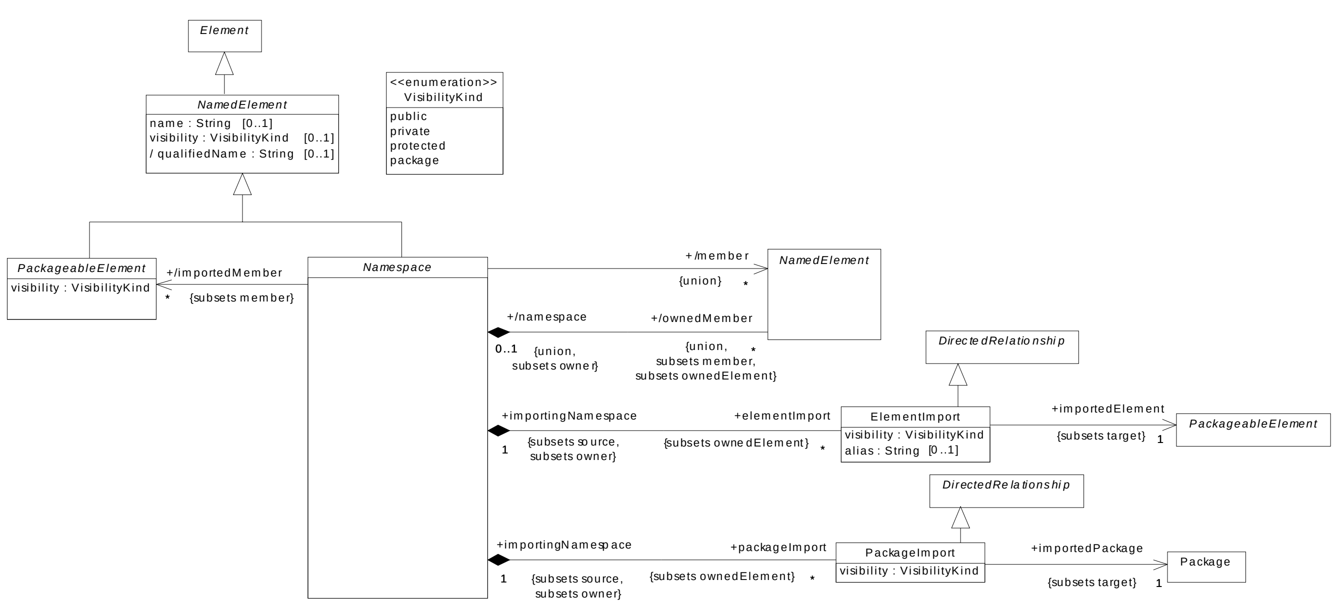


Figure 7.4 - Namespaces diagram of the Kernel package

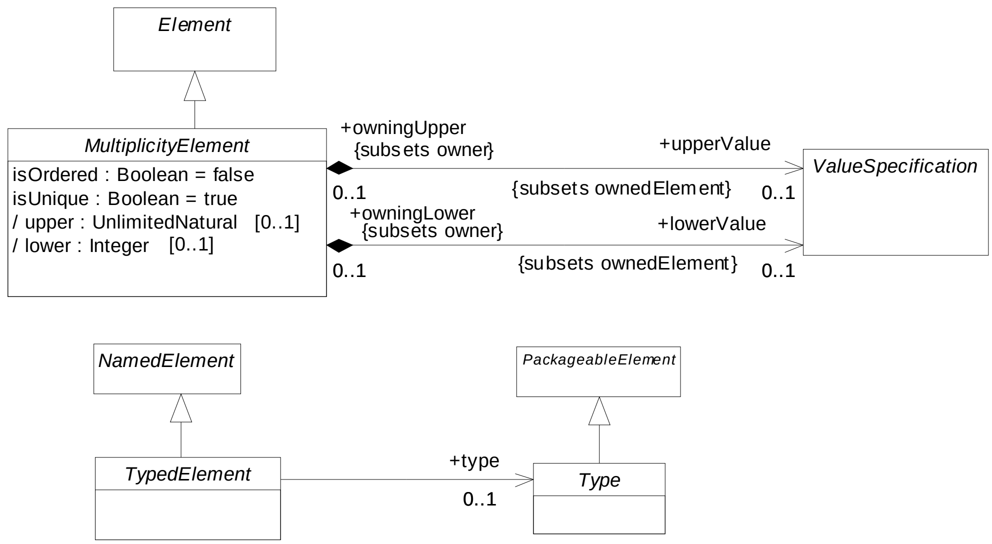


Figure 7.5 - Multiplicities diagram of the Kernel package

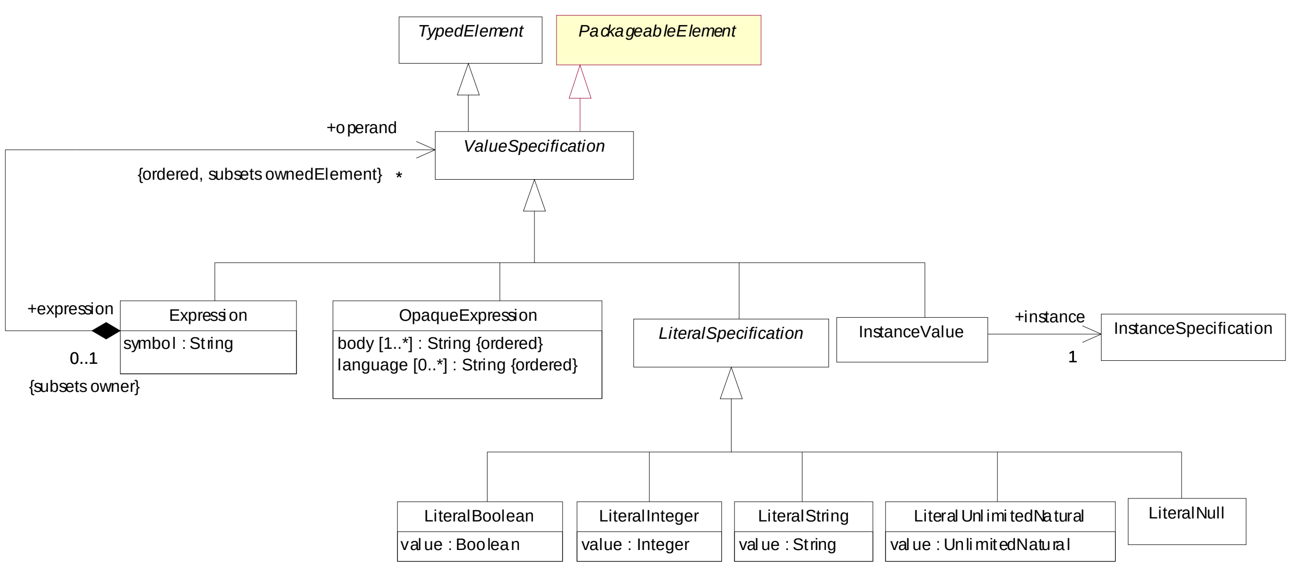


Figure 7.6 - Expressions diagram of the Kernel package

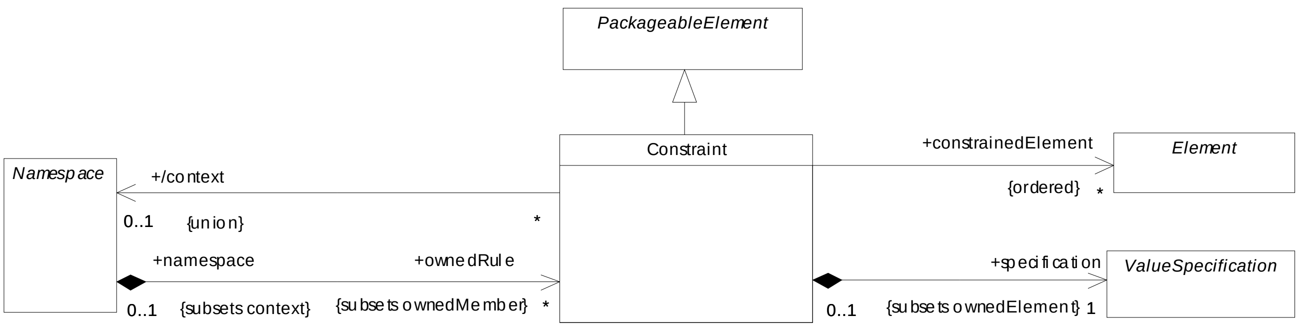


Figure 7.7 - Constraints diagram of the Kernel package

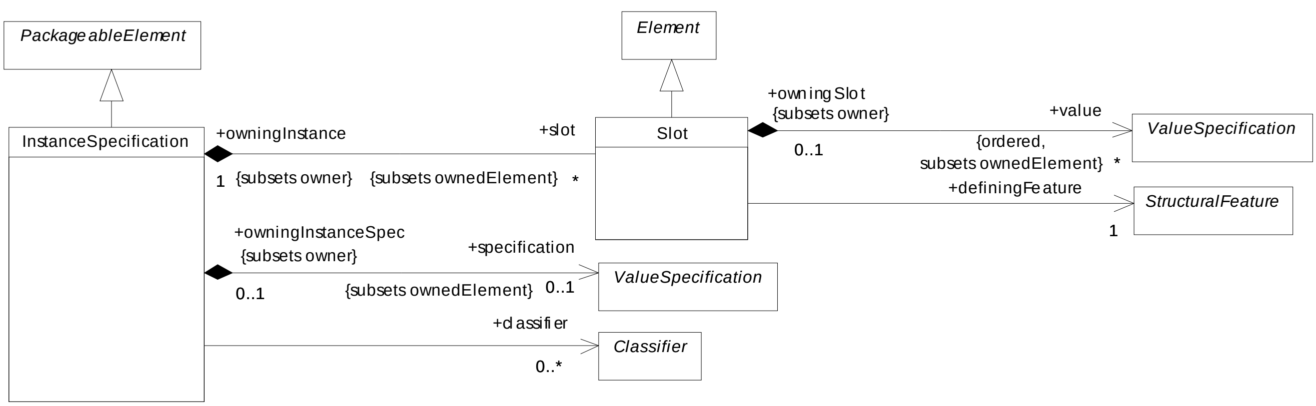


Figure 7.8 - Instances diagram of the Kernel package

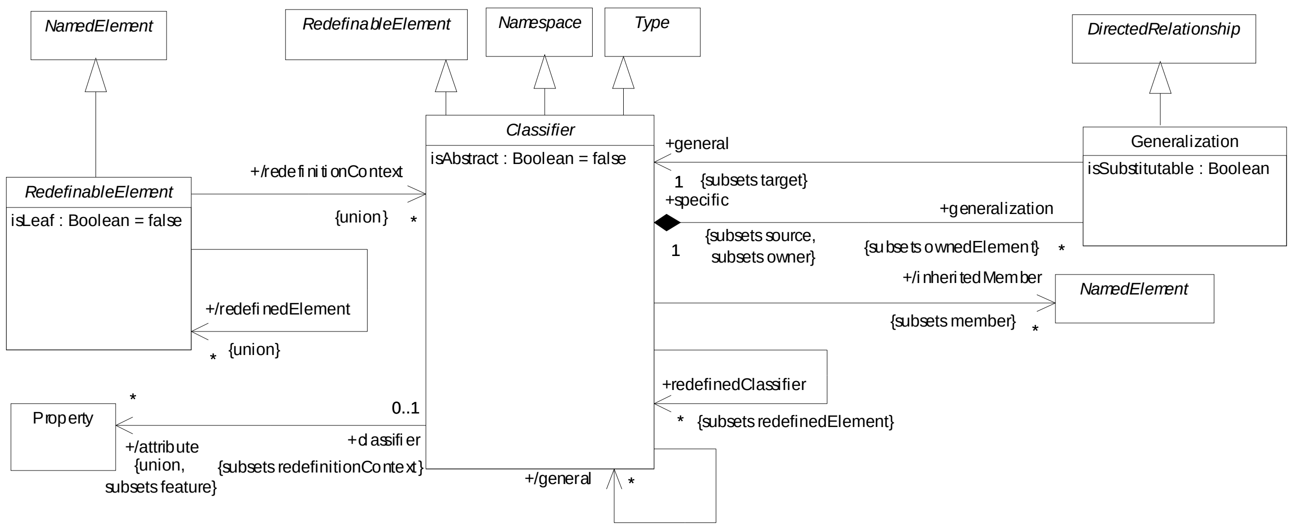


Figure 7.9 - Classifiers diagram of the Kernel package

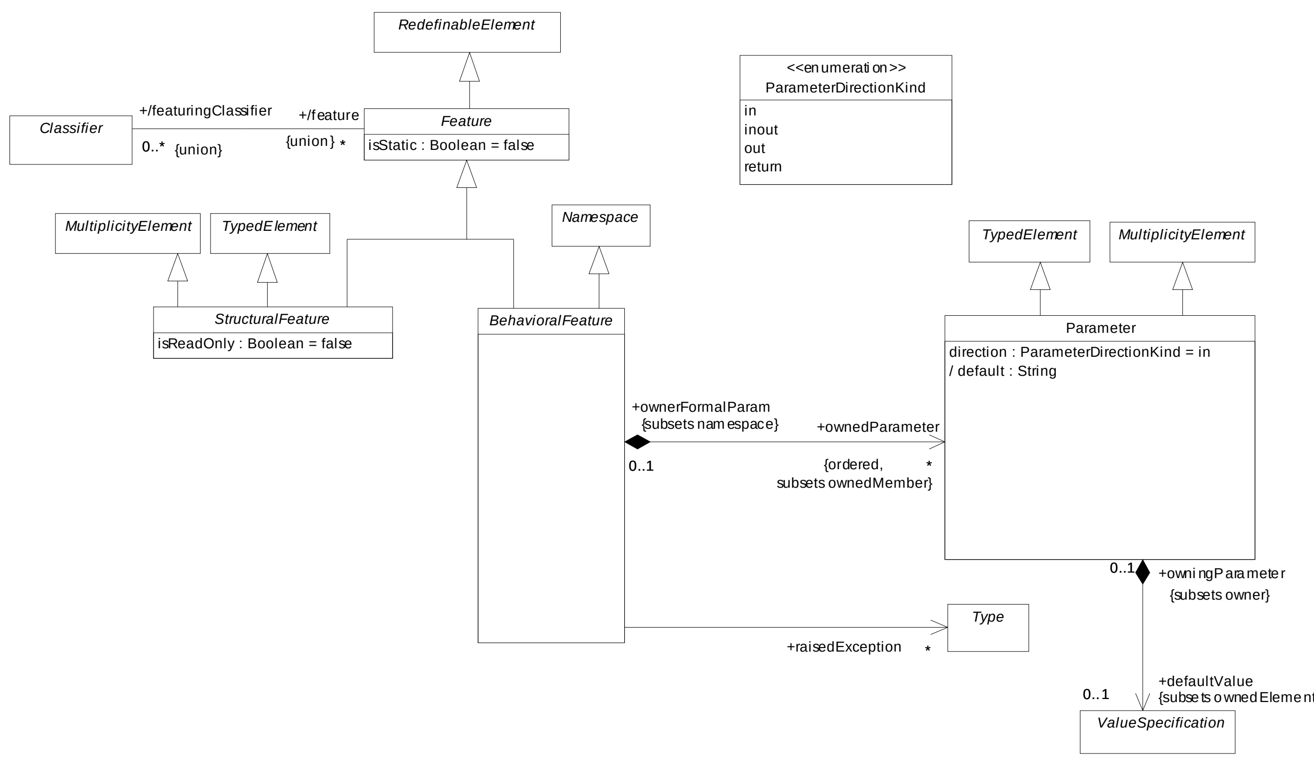


Figure 7.10 - Features diagram of the Kernel package

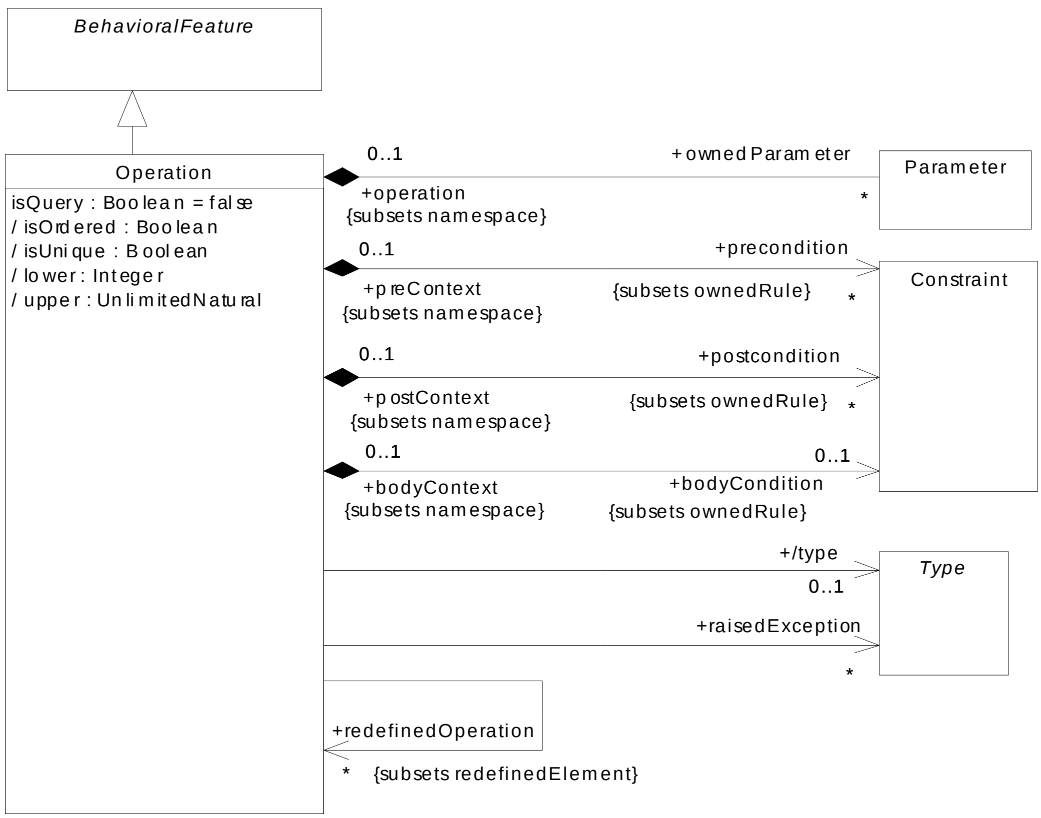


Figure 7.11 - Operations diagram of the Kernel package

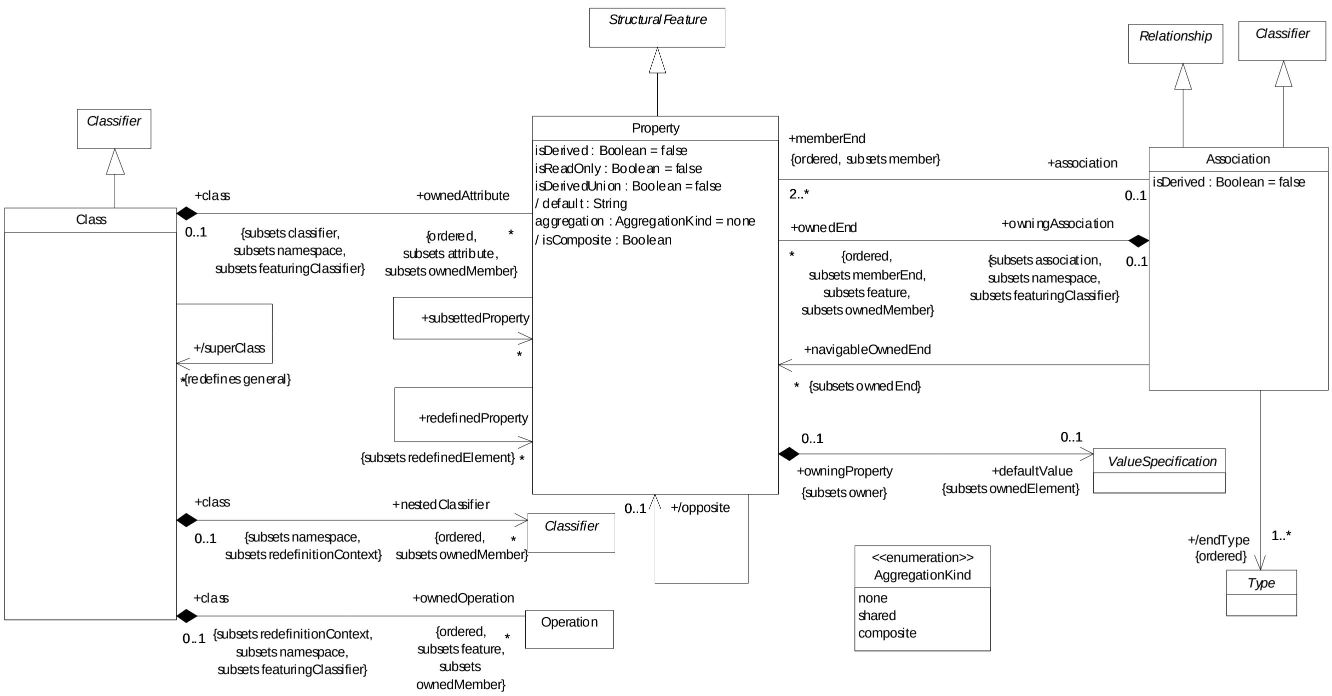


Figure 7.12 - Classes diagram of the Kernel package

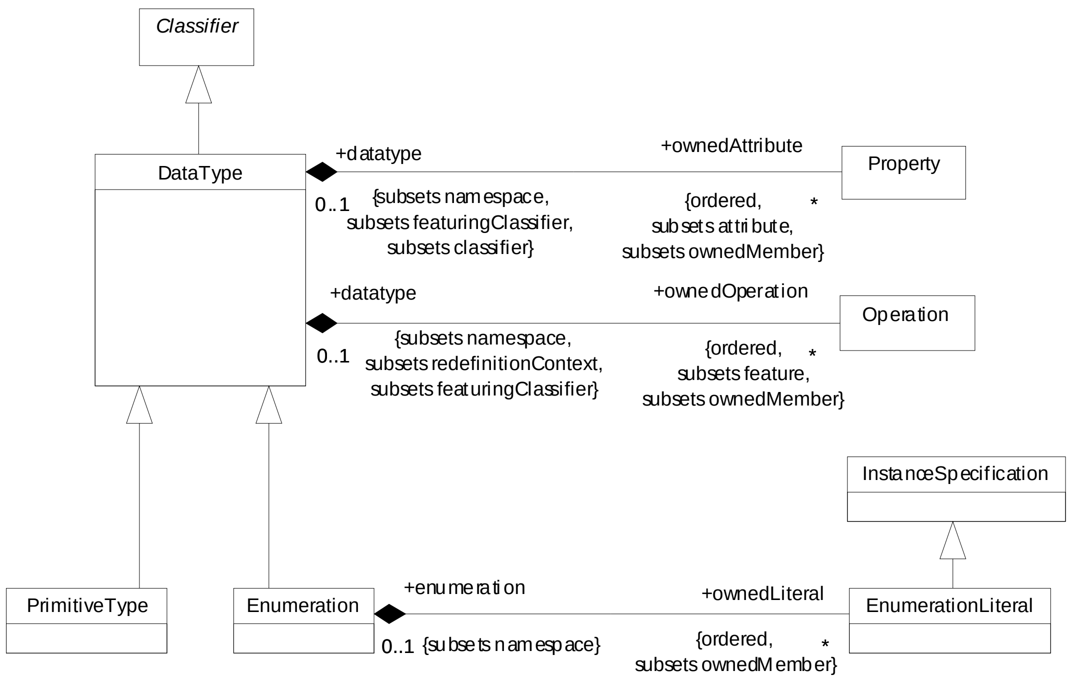


Figure 7.13 - DataTypes diagram of the Kernel package

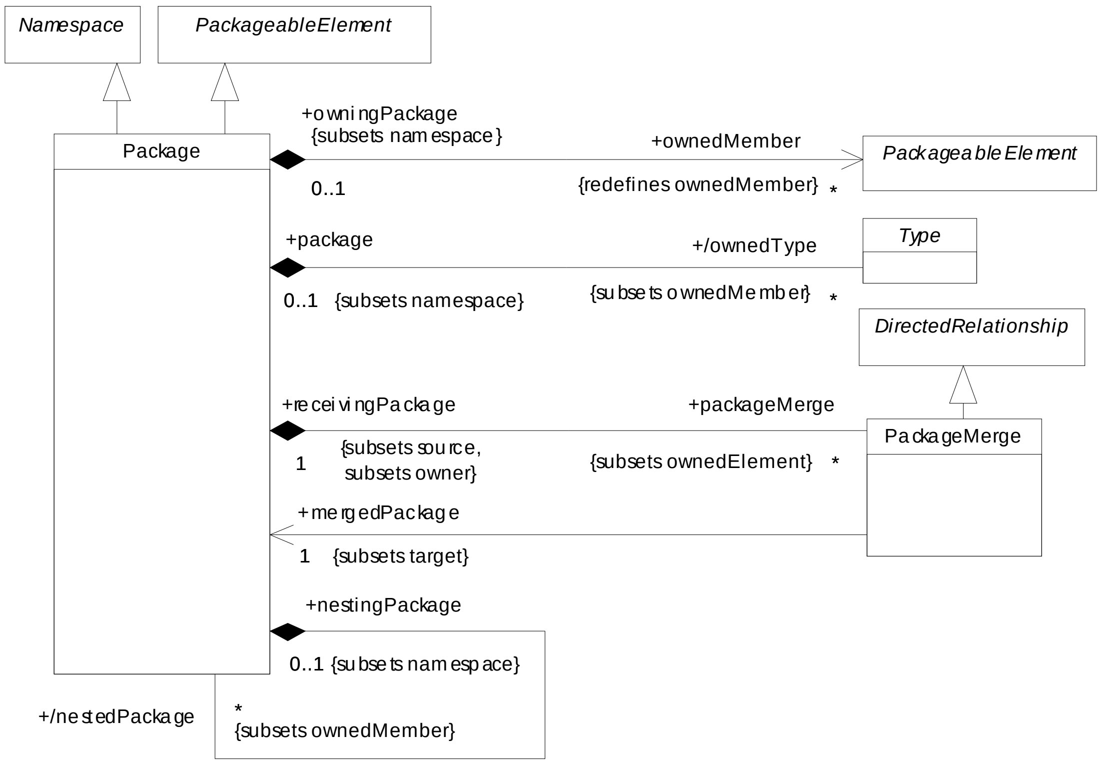


Figure 7.14 - The Packages diagram of the Kernel package

Package Dependencies

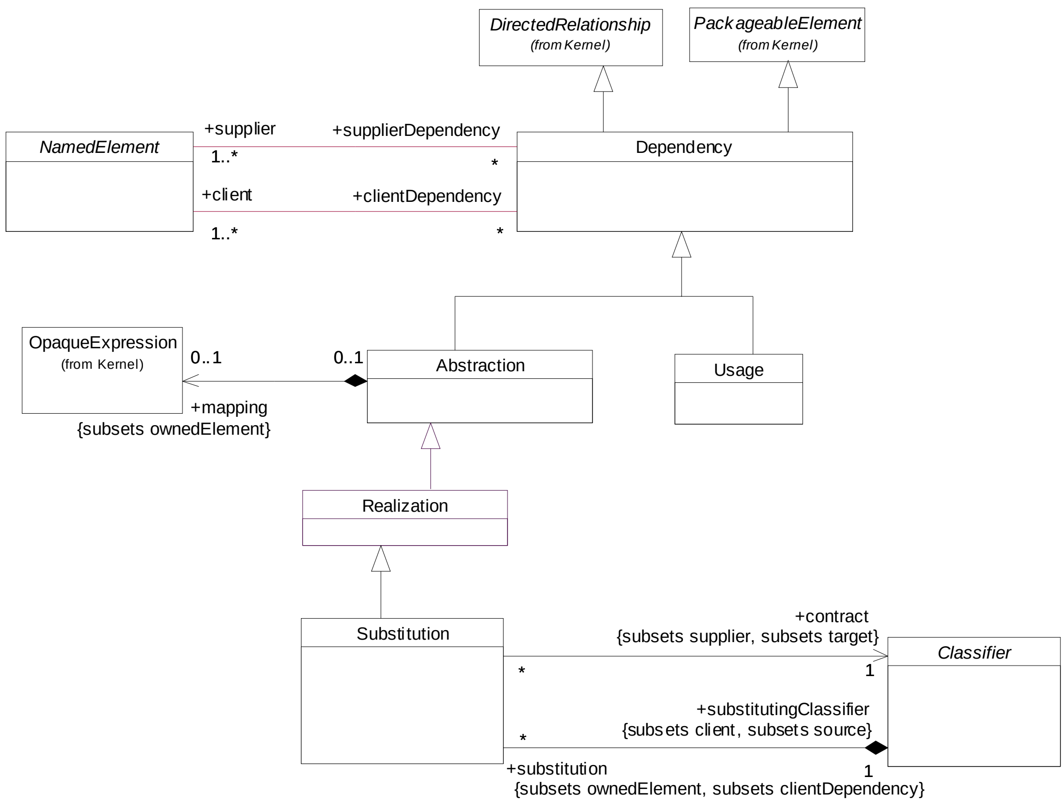


Figure 7.15 - Contents of Dependencies package

Package Interfaces

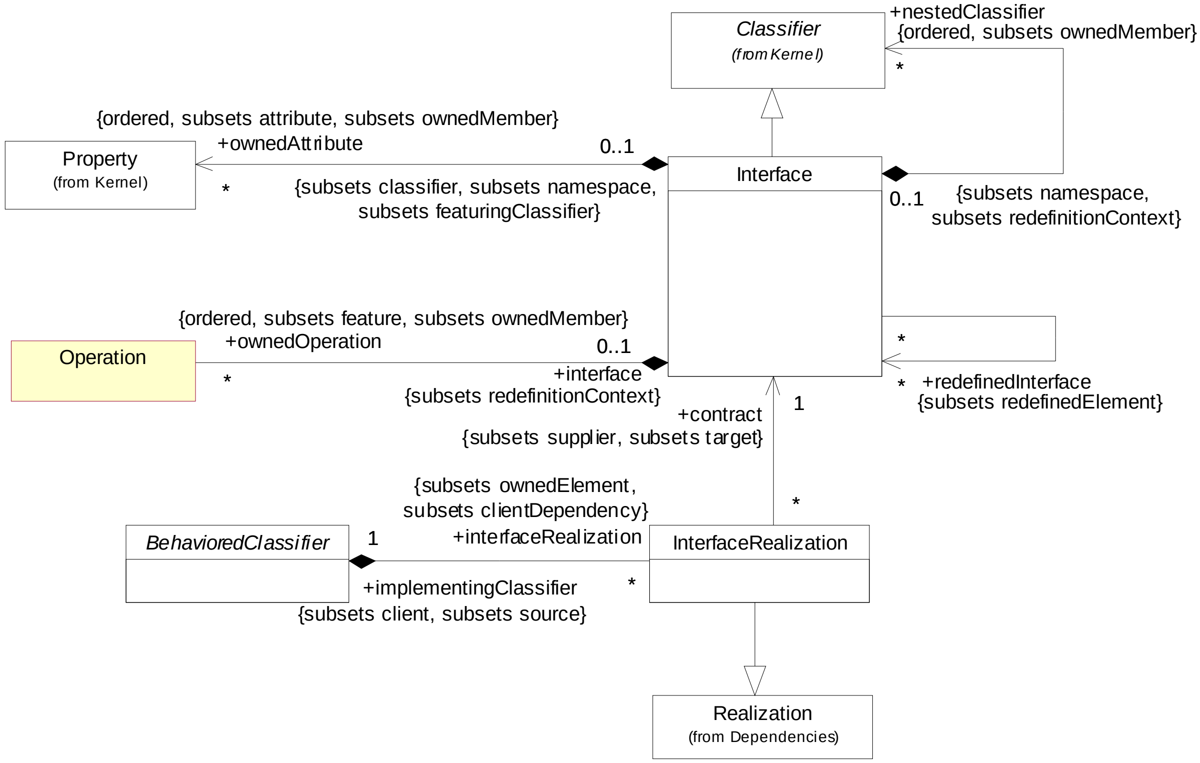


Figure 7.16 - Contents of Interfaces package

Package AssociationClasses

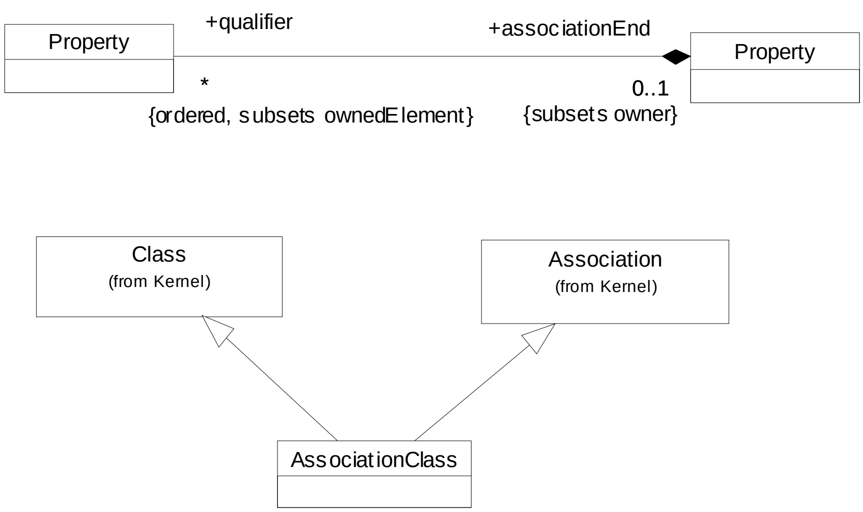


Figure 7.17 - Contents of AssociationClasses package

Package PowerTypes

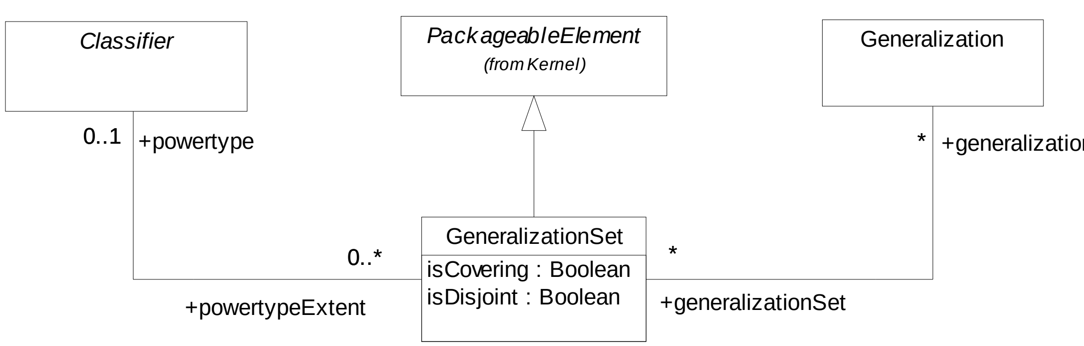


Figure 7.18 - Contents of PowerTypes package

7.3 Class Descriptions

7.3.1 Abstraction (from Dependencies)

Generalizations

- "Dependency (from Dependencies)" on page 58

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 chapter. 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 48
- "Relationship (from Kernel)" on page 126

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.

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 the association itself. It may additionally apply to the extents of classifiers generally. The collection represented by one association end may be a subset of the collection represented by another association end without being a proper subset. That is to say, for A to be a subset of B, it is not required that collection B has a member NOT in A. Proper subsetting implies that the superset is not empty and that the subset has fewer members; subsetting does not have this implication. 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. The lower multiplicity for an end of an n-ary association of 1 (or more) implies that one link (or more) must exist for every possible combination of values for the other ends.

An association may represent a composite aggregation (i.e., a whole/part relationship). Only binary associations can be aggregations. Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it. Note that a part can (where allowed) be removed from a composite before the composite is deleted, and thus not be deleted as part of the composite. Compositions define transitive asymmetric relationships—their links form a directed, acyclic graph. Composition is represented by the isComposite attribute on the part end of the association being set to true.

Semantic Variation Points

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

Notation

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

A binary 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 7.19).

- 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>.
    - {redefined <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.
    - {bag} to show that the end represents a collection that permits the same element to appear more than once.
    - {sequence} or {seq} to show that the end represents a sequence (an ordered bag).
    - If the end is navigable, any property strings that apply to an attribute.

Note that by default an association end represents a set.

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 subsection of "Property (from Kernel, AssociationClasses)" on page 118.

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.

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

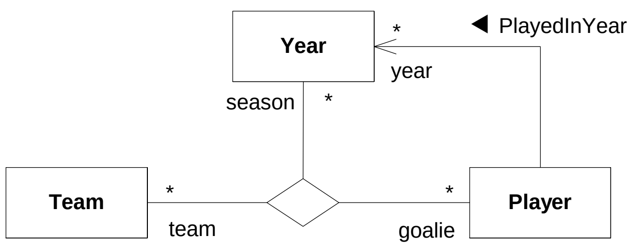


Figure 7.19 - Binary and ternary associations

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.

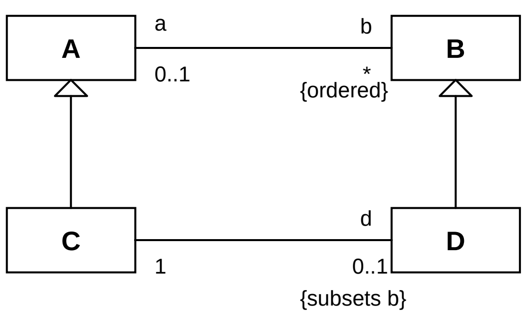


Figure 7.20 - Association ends with various adornments

The following adornments are shown on the four association ends in Figure 7.20.

- 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.

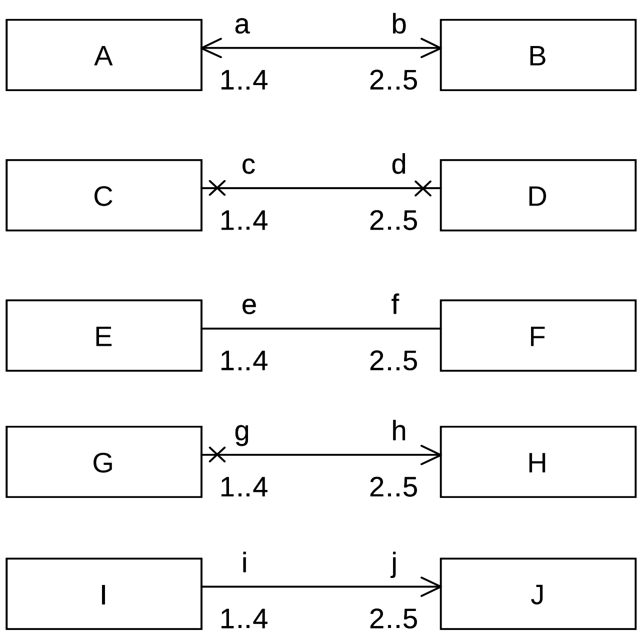


Figure 7.21 - Examples of navigable ends

In Figure 7.21:

- 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 7.22 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.

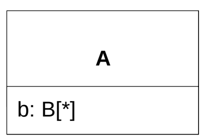


Figure 7.22 - Example of attribute notation for navigable end owned by an end class

Figure 7.23 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.

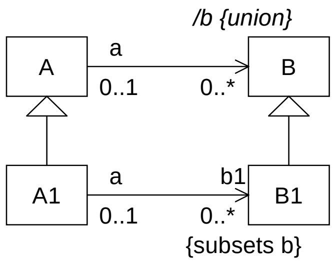


Figure 7.23 - Derived supersets (union)

Figure 7.24 shows the black diamond notation for composite aggregation.

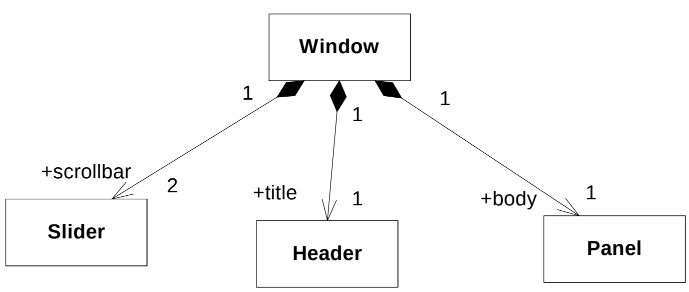


Figure 7.24 - Composite aggregation is depicted as a black diamond

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 36
- “Class (from Kernel)” on page 45

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.

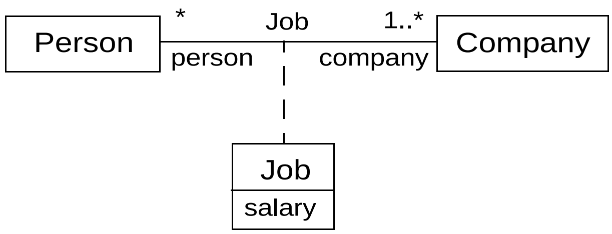


Figure 7.25 - An AssociationClass is depicted by an association symbol (a line) and a class symbol (a box) connected with a dashed line. The diagram shows the association class Job, which is defined between the two classes Person and Company.

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 66
- "Namespace (from Kernel)" on page 95

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 419 *(merge increment)*

Description

A BehavioredClassifier may have an interface realization.

Associations

- interfaceRealization: InterfaceRealization [*] (Specializes *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 48

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.

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

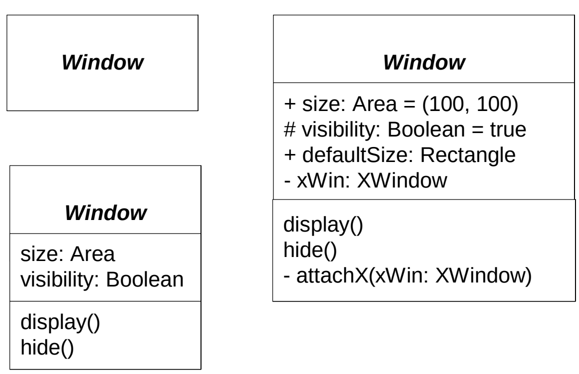


Figure 7.26 - Class notation: details suppressed, analysis-level details, implementation-level details

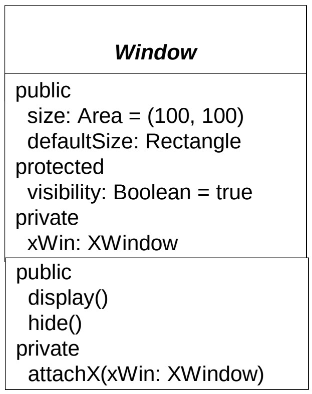


Figure 7.27 - Class notation: attributes and operations grouped according to visibility

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 95
- “RedefinableElement (from Kernel)” on page 125
- “Type (from Kernel)” on page 130

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.
- package: Package [0..1] Specifies the owning package of this classifier, if any. Subsets *NamedElement::namespace*.
- 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 subsection of "Property (from Kernel, AssociationClasses)" on page 118.

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

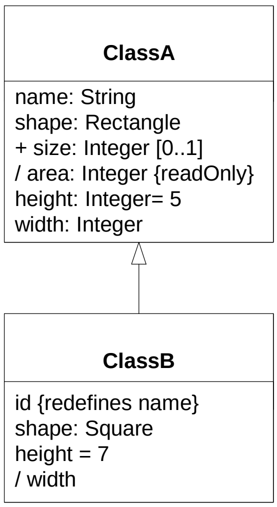


Figure 7.28 - Examples of attributes

The attributes in *Figure 7.28* 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 7.29.

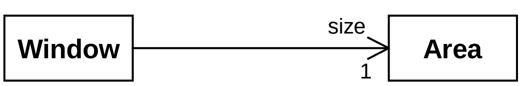


Figure 7.29 - Association-like notation for attribute

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 section, 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 60.

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 Specifies a string that is the comment.

Associations

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

Constraints

No additional constraints.

Semantics

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

Notation

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

Presentation Options

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

Examples

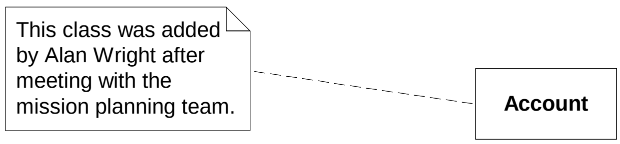


Figure 7.30 - Comment notation

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 105

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. This is a derived union.
- specification: ValueSpecification[0..1] A condition that must be true when evaluated in order for the constraint to be satisfied. Subsets *Element::ownedElement*.

Constraints

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

[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 7.31 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 7.32 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 7.33 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

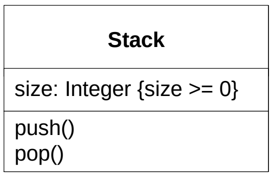


Figure 7.31 - Constraint attached to an attribute

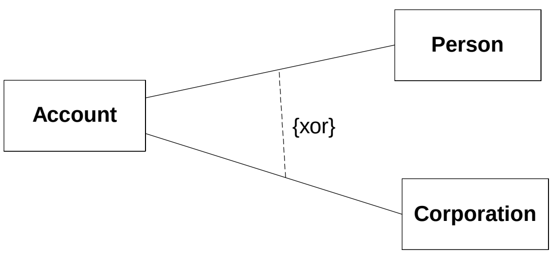


Figure 7.32 - {xor} constraint

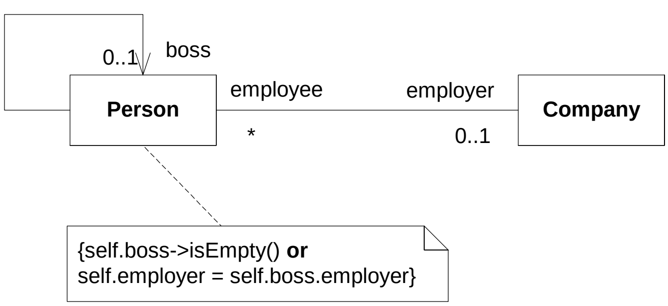


Figure 7.33 - Constraint in a note symbol

7.3.11 DataType (from Kernel)

Generalizations

- "Classifier (from Kernel, Dependencies, PowerTypes)" on page 48.

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

Constraints

No additional constraints

Semantics

A data type is a special kind of classifier, similar to a class. 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

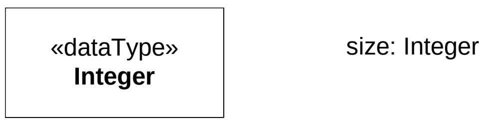


Figure 7.34 - Notation of data type: to the left is an icon denoting a data type and to the right is a reference to a data type that is used in an attribute.

7.3.12 Dependency (from Dependencies)

Generalizations

- "DirectedRelationship (from Kernel)" on page 59
- "PackageableElement (from Kernel)" on page 105

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.
- 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.

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.

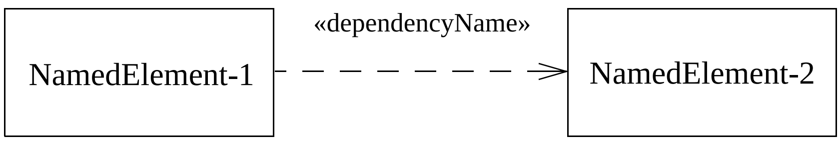


Figure 7.35 - Notation for a dependency between two elements

Examples

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

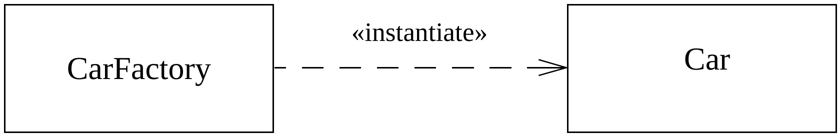


Figure 7.36 - An example of an instantiate dependency

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 126

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 59

Description

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

Attributes

- visibility: VisibilityKind — Specifies the visibility of the imported PackageableElement within the importing Package. The default visibility is the same as that of the imported element. If the imported element does not have a visibility, it is possible to add visibility to the element import.
- alias: String [0..1] — Specifies the name that should be added to the namespace of the importing Package in lieu of the name of the imported PackagableElement. The aliased name must not clash with any other member name in the importing Package. By default, no alias is used.

Associations

- importedElement: PackageableElement [1] — Specifies the PackageableElement whose name is to be added to a Namespace. Subsets *DirectedRelationship::target.*
- importingNamespace: Namespace [1] — Specifies the Namespace that imports a PackageableElement from another Package. Subsets *DirectedRelationship::source* and *Element::owner.*

Constraints

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

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

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

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

Additional Operations

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

ElementImport::getName(): String;

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

Semantics

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

In case of a 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 names of the imported elements must be qualified in order to be used and the elements are not added to the importing namespace. If the name of an imported element is the same as the name of an element owned by the importing namespace, the name of the imported element must be qualified in order to be used and is not added to the importing namespace.

An imported element can be further imported by other namespaces using either element or 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 7.37 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.

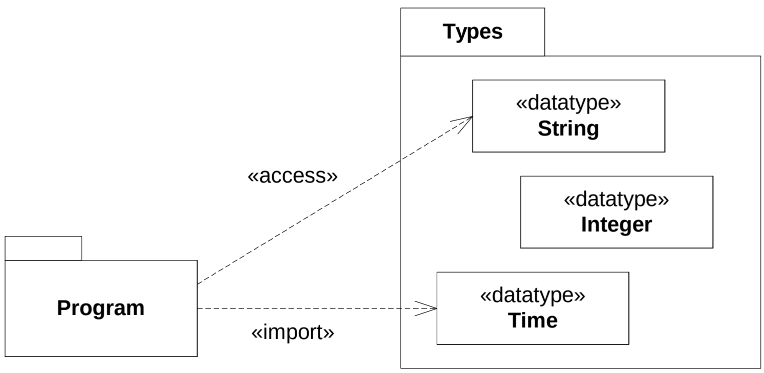


Figure 7.37 - Example of element import

In Figure 7.38, the element import is combined with aliasing, meaning that the type Types::Real will be referred to as Double in the package Shapes.

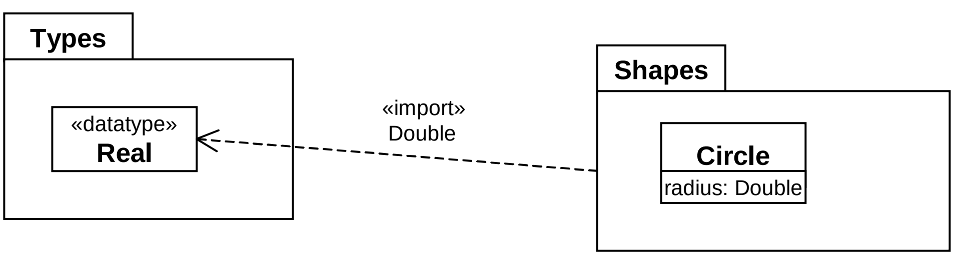


Figure 7.38 - Example of element import with aliasing

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 56

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

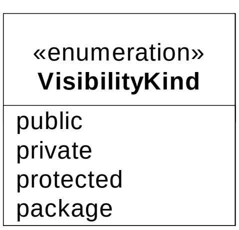


Figure 7.39 - Example of an enumeration

7.3.17 EnumerationLiteral (from Kernel)

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

Generalizations

- “InstanceSpecification (from Kernel)” on page 78

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 132

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 [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 125

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 59

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.

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 section 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 7.40.

When two or more lines are drawn to the same arrowhead, as illustrated in b), Figure 7.40, 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 7.40, 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 7.40. 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.

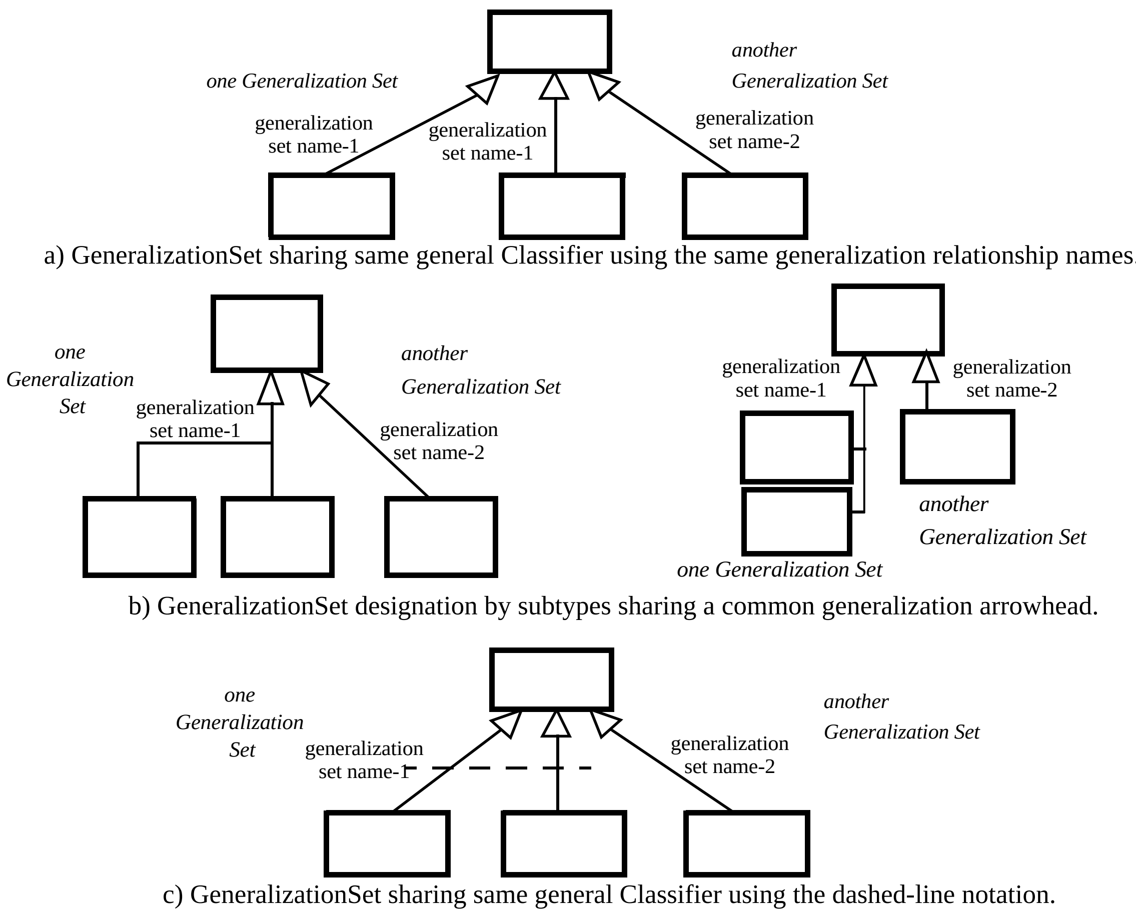


Figure 7.40 - GeneralizationSet designation notations

Presentation Options

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

Examples

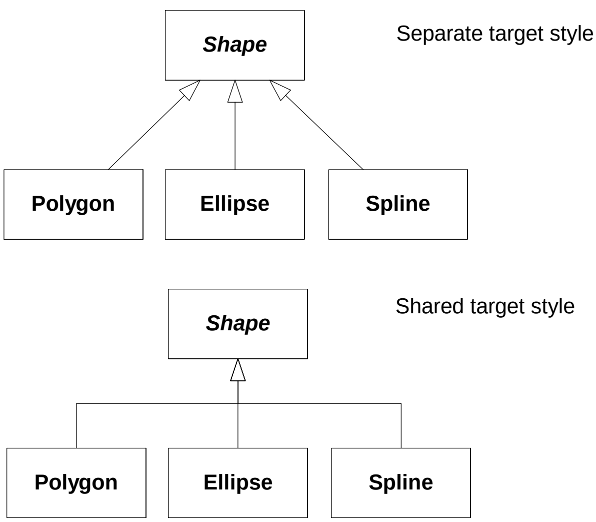


Figure 7.41 - Examples of generalizations between classes

Package PowerTypes

In Figure 7.42, 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 Manager another. This illustration employs the notation forms depicted in the diagram above.

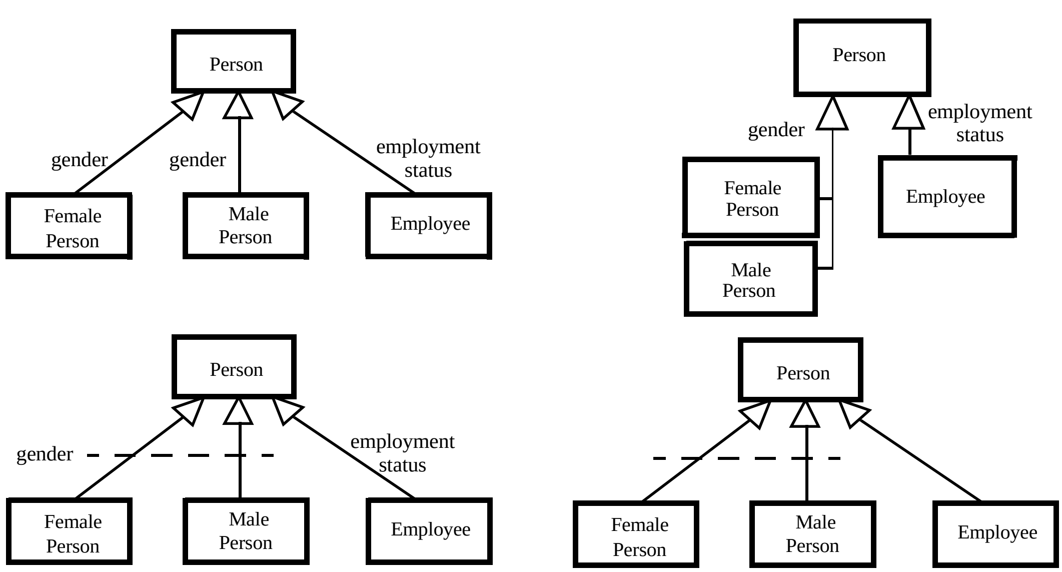


Figure 7.42 - Multiple subtype partitions (GeneralizationSets) example

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 105

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.
- 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.

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 74.)

Notation

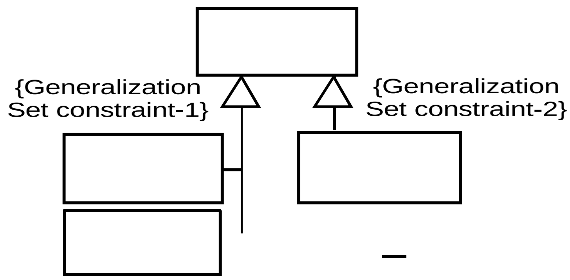
The notation to express the grouping of Generalizations into GeneralizationSets was presented in the Notation section of Generalization, above. To indicate whether or not a generalization set is covering and disjoint, each set should be labeled with one of the constraints indicated below.

{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.

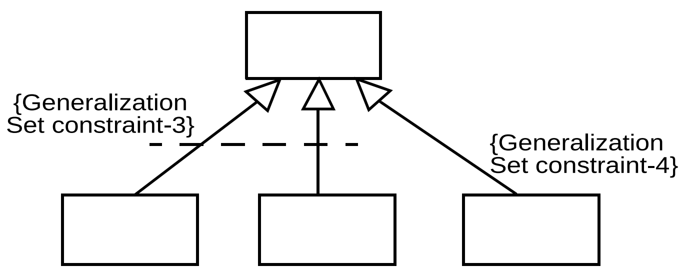
* default is {incomplete, disjoint}

Figure 7.43 - 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..



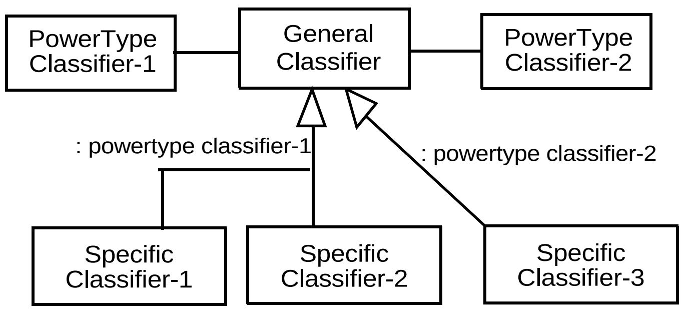
(a) GeneralizationSet constraint when sharing common generalization arrowhead.



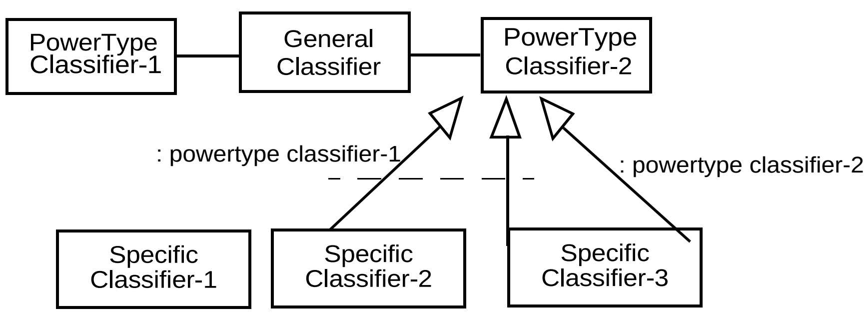
(b) GeneralizationSet constraint using dashed-line notation.

Figure 7.44 - 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 7.45 - 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).

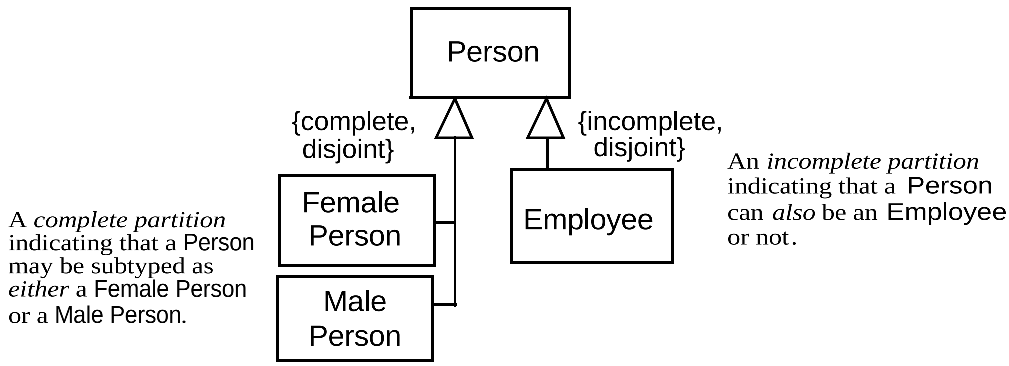


Figure 7.46 - Multiple ways of dividing subtypes (generalization sets) and constraint examples

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.

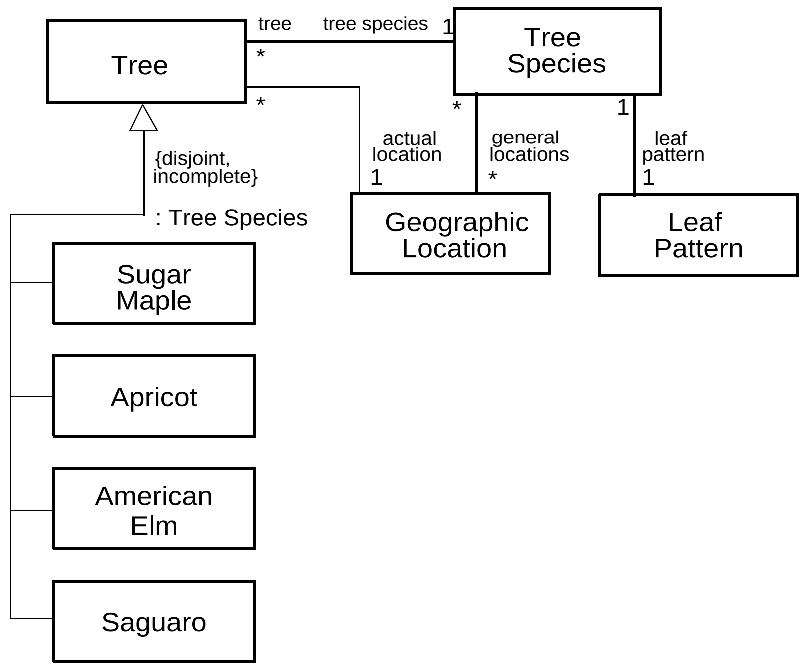
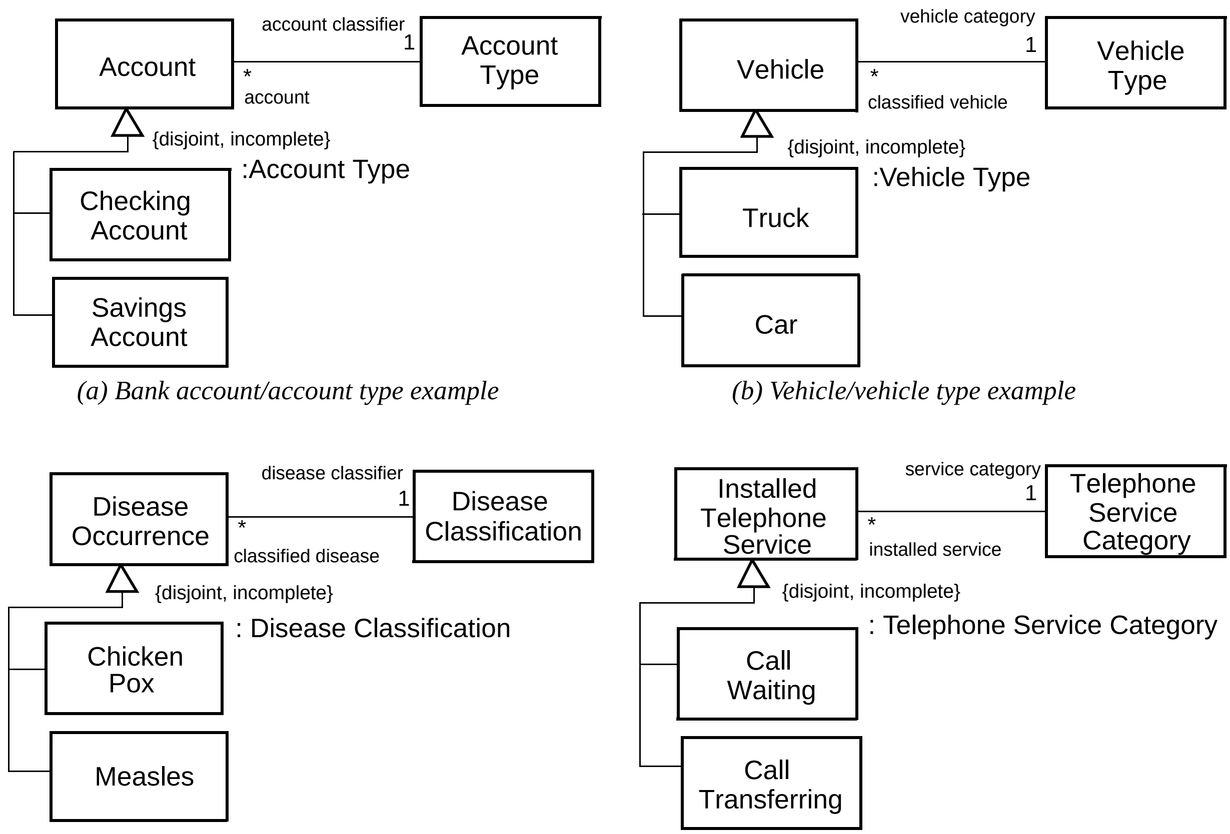


Figure 7.47 - Power type example and notation

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 7.48 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 7.48 - 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.

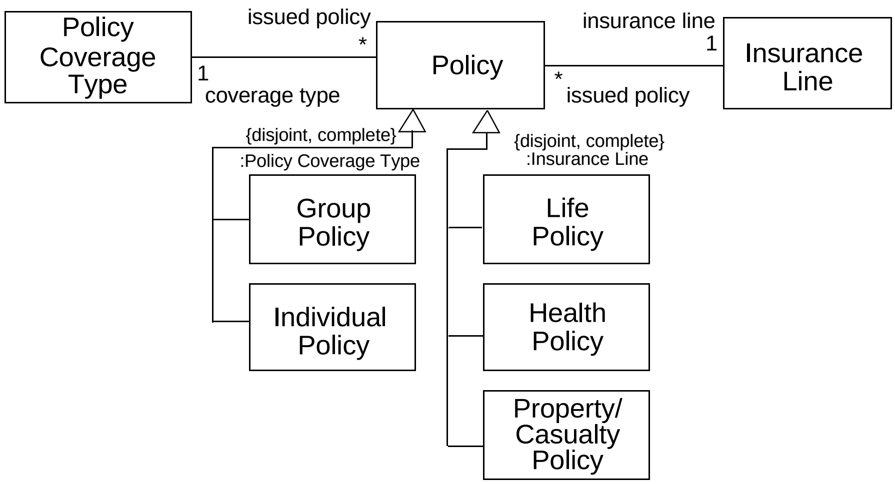


Figure 7.49 - Other power type examples

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 105

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.

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.

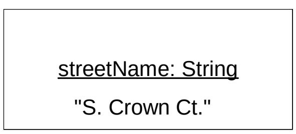


Figure 7.50 - Specification of an instance of String

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.

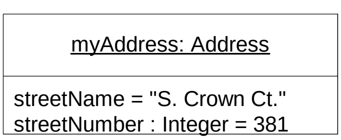


Figure 7.51 - Slots with values

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 7.52 - 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 132

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 48

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 85).

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 7.53).

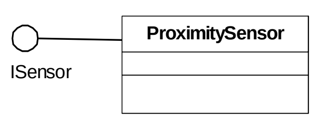


Figure 7.53 - Isensor is the provided interface of ProximitySensor

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 7.54).

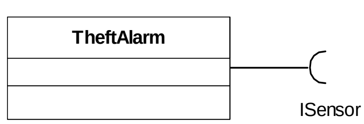


Figure 7.54 - Isensor is the required interface of TheftAlarm

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 7.55). The classifier at the tail of the arrow implements the interface at the head of the arrow or uses that interface, respectively.

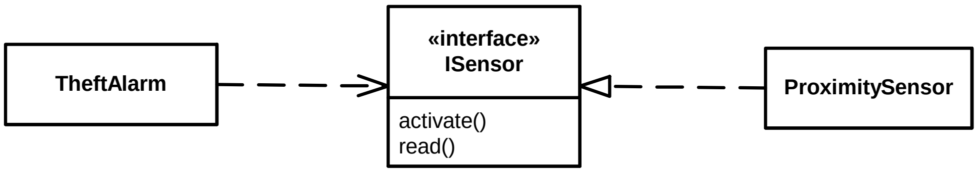


Figure 7.55 - Alternative notation for the situation depicted in Figure 7.53 and Figure 7.54

A set of interfaces constituting a protocol may be depicted as interfaces with associations between them (see Figure 7.56).

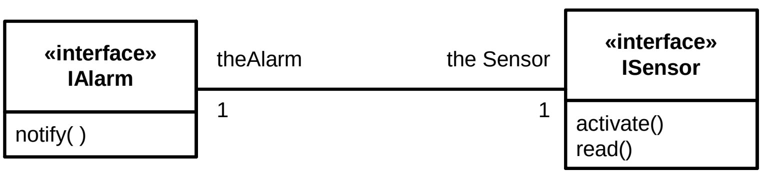


Figure 7.56 - Alarm is the required interface for any classifier implementing Isensor; conversely, Isensor is the required interface for any classifier implementing IAlarm.

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 7.57 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.

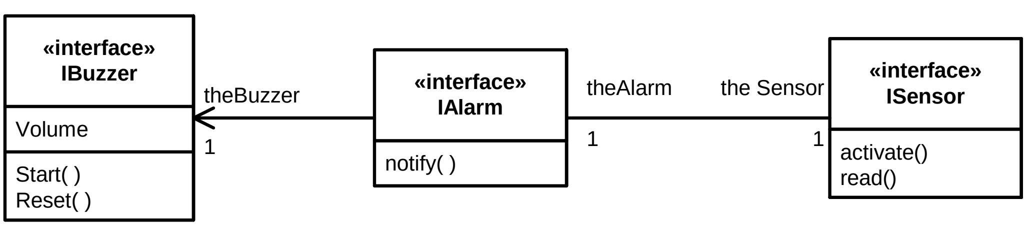


Figure 7.57 - A set of collaborating interfaces.

Three classes: *DoorSensor*, *DoorAlarm*, and *DoorBell* implement the above interfaces (see Figure 7.58). 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.

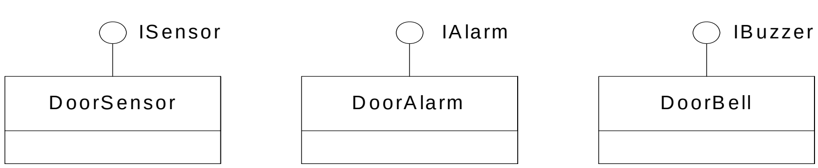


Figure 7.58 - Classifiers implementing the above interfaces

7.3.25 InterfaceRealization (from Interfaces)

Generalizations

- “Realization (from Dependencies)” on page 124

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. (Specializes *Dependency.supplier* and *Relationship.target*).
- implementingClassifier: BehavioredClassifier [1]
  References the BehavioredClassifier that owns this Interfacerealization (i.e., the classifier that realizes the Interface to which it points). (Specializes *Dependency.client, Element.owner,* and *Relationship.source*.)

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 88

Description

A literal boolean contains a Boolean-valued attribute.

Attributes

- value: Boolean The specified Boolean value.

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

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

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

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

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

Semantics

A LiteralBoolean specifies a constant Boolean value.

Notation

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

7.3.27 LiteralInteger (from Kernel)

A literal integer is a specification of an integer value.

Generalizations

- "LiteralSpecification (from Kernel)" on page 88

Description

A literal integer contains an Integer-valued attribute.

Attributes

- value: Integer The specified Integer value.

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

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

[2] The query integerValue() gives the value.
LiteralInteger::integerValue() : [Integer];
integerValue = value

Semantics

A LiteralInteger specifies a constant Integer value.

Notation

A LiteralInteger is 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 88

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 132

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 88.

Description

A literal string contains a String-valued attribute.

Attributes

- value: String The specified String value

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

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

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

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

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

Semantics

A LiteralString specifies a constant String value.

Notation

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

The character set used is unspecified.

7.3.31 LiteralUnlimitedNatural (from Kernel)

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

Generalizations

- “LiteralSpecification (from Kernel)” on page 88

Description

A literal unlimited natural contains an UnlimitedNatural-valued attribute.

Attributes

- value: UnlimitedNatural The specified UnlimitedNatural value.

Associations

No additional associations

Constraints

No additional constraints

Additional Operations

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

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

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

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

Semantics

A LiteralUnlimitedNatural specifies a constant UnlimitedNatural value.

Notation

A LiteralUnlimitedNatural is shown either as a sequence of digits or as an asterisk (*), where 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 60

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

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

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

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

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

[4] 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.

[5] 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.

[6] The derived lower attribute must equal the lowerBound.

lower = lowerBound()

[7] 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.

Notation

The specific notation for a MultiplicityElement is defined by the concrete subclasses. In general, the notation will include a multiplicity specification 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 7.59 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 7.60 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>] ‘}’]
<multiplicity-range> ::= [<lower> ‘..’] <upper>
<lower> ::= <integer> | *<value-specification>*
<upper> ::= ‘’* | *<value-specification>*
<order-designator> ::= ‘ordered’ | *‘unordered’*
<uniqueness-designator> ::= ‘unique’ | *‘nonunique’*

Examples

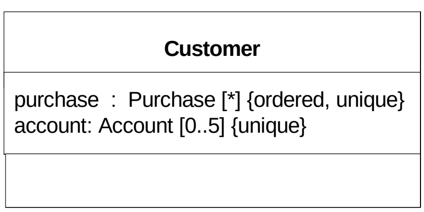


Figure 7.59 - Multiplicity within a textual specification

Figure 7.60 - 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 60

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 the visibility of the NamedElement within different Namespaces within the overall model.

Package Dependencies

- supplierDependency: Dependency [*] — Indicates the dependencies that reference the supplier.
- 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 in different namespaces within a model. It is intended for use in conjunction with import and generalization 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 93.

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. The constraint does not necessarily apply to the namespace itself, but may instead 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 132

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 [1..*] 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

No additional constraints

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 interpretation of the expression body depends on the languages. 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, is often not shown on a diagram. Some modeling tools may impose a particular language or assume a particular default language. The language is often implicit under the assumption that the form of the expression makes its purpose clear. If the language name is shown, it should be displayed in braces ({}) before the expression string 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 44

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[*] — 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

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

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

where:

- *<visibility>* is the visibility of the operation (See “VisibilityKind (from Kernel)” on page 133).

  <visibility> ::= ‘+’ | ‘-‘ | ‘#’ | ‘~’

- *<name>* is the name of the operation.

- *<return-type>* is the type of the return result parameter if the operation has one defined.
- *<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 90).
    - *<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 95
- “PackageableElement (from Kernel)” on page 105

Description

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

In addition a package can be merged with other packages.

Attributes

No additional attributes

Associations

• /nestedPackage: Package [*]	References the owned members that are Packages. Subsets *Package::ownedMember*
• ownedMember: PackageableElement [*]	Specifies the members that are owned by this Package. Redefines *Namespace::ownedMember.*
• ownedType: Type [*]	References the owned members that are Types. Subsets *Package::ownedMember*
• package: Package [0..1]	References the owning package of a package. Subsets *NamedElement::namespace*
• 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 7.61. 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.

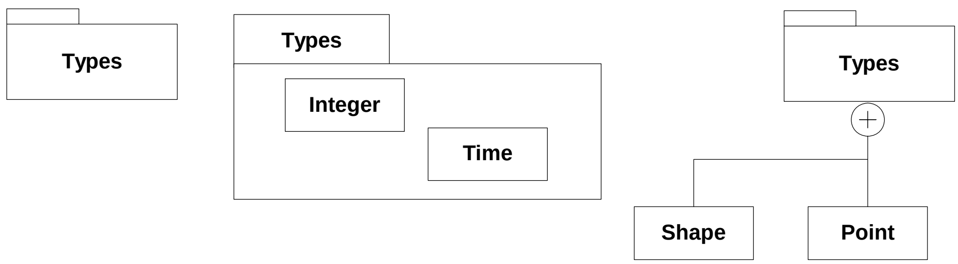


Figure 7.61 - Examples of a package with members

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 93

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.*

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 59

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 7.62, 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.

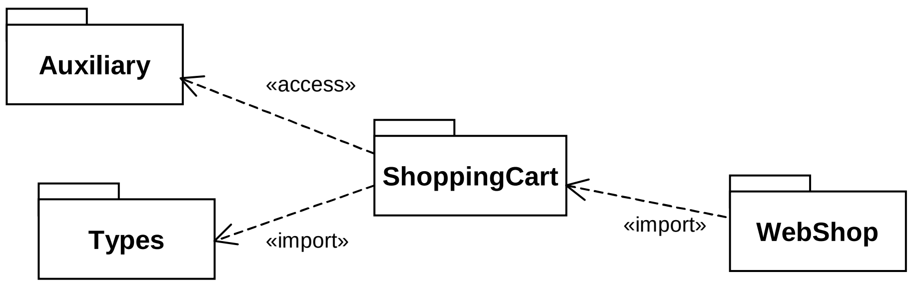


Figure 7.62 - Examples of public and private package imports

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 59

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 7.63 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.

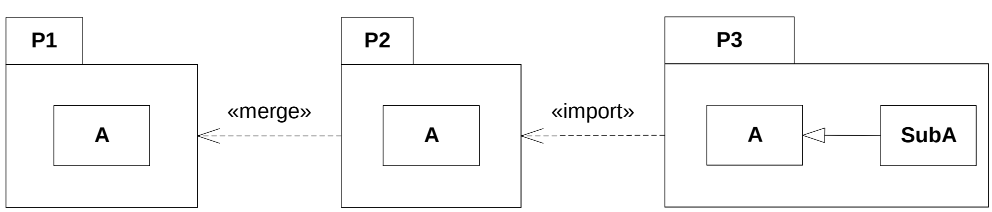


Figure 7.63 - Illustration of the meaning of package merge

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 7.63), 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 7.63, 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 7.64 (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.

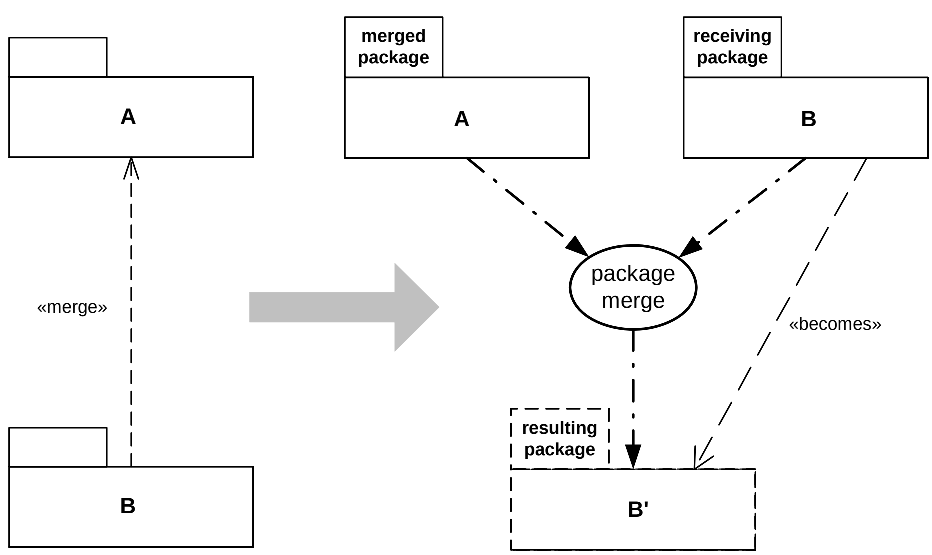


Figure 7.64 - Conceptual view of the package merge semantics

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.

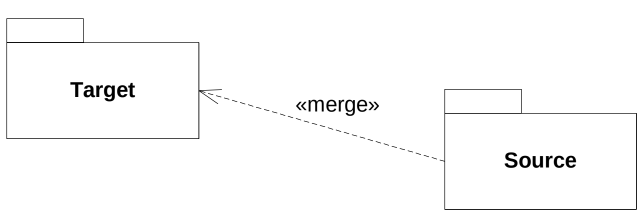


Figure 7.65 - Notation for package merge

Examples

In Figure 7.66, packages P and Q are being merged by package R, while package S merges only package Q.

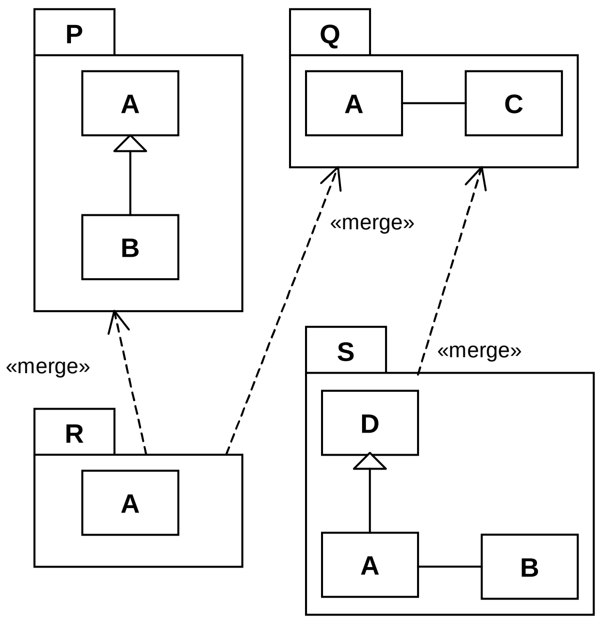


Figure 7.66 - Simple example of package merges

The transformed packages R and S are shown in Figure 7.67. 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).

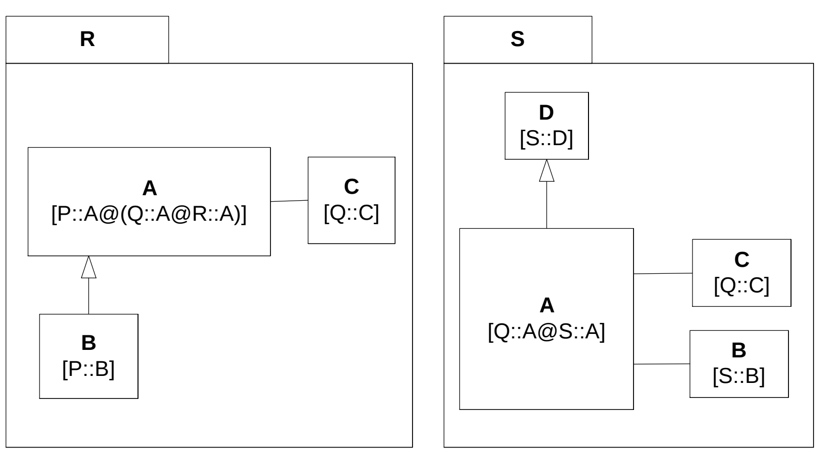


Figure 7.67 - Simple example of transformed packages following the merges in Figure 7.66

In Figure 7.68, 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.

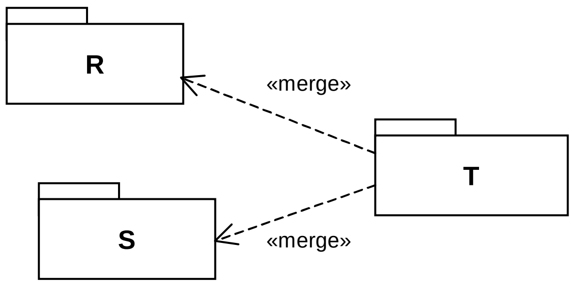


Figure 7.68 - Introducing additional package merges

In Figure 7.69, 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.

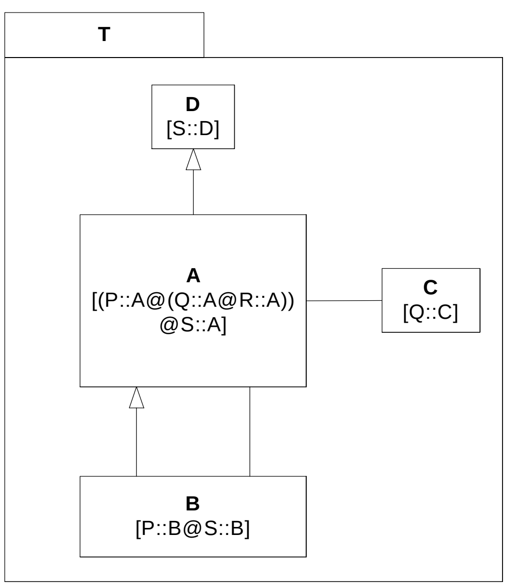


Figure 7.69 - The result of the additional package merges in Figure 7.68

7.3.41 Parameter (from Kernel, AssociationClasses)

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 90.
- "TypedElement (from Kernel)" on page 131.

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). A primitive datatype may have an algebra and operations defined outside of UML, for example, mathematically.

Generalizations

- “DataType (from Kernel)” on page 56.

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 128

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.

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

[2] A 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 navigable property can only be redefined or subsetted by a navigable property.

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

[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)
```

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
                    type.conformsTo(prop.type) and
                    (lowerBound()->notEmpty and prop.lowerBound()->notEmpty() implies
                        lowerBound() >= prop.lowerBound()) and
                    (upperBound()->notEmpty and prop.upperBound()->notEmpty() implies
                        upperBound() <= prop.upperBound()) and
                    (prop.isDerived implies isDerived)
                )
```

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

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

[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)

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 non-derived property. Derived properties are often specified to be read-only (i.e., clients cannot directly change values). But where a derived property is changeable, an implementation is expected to appropriately change the source information of the derivation. The derivation for a derived property may be specified by a constraint.

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

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

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

If a navigable property 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 sections of those classes.

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

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

where:

- <*visibility*> is the visibility of the property. (See “VisibilityKind (from Kernel)” on page 133.)

  <*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 90.)
- *<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’ | <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.

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

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 7.70 - Qualified associations

7.3.45 Realization (from Dependencies)

Generalizations

- "Abstraction (from Dependencies)" on page 35

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

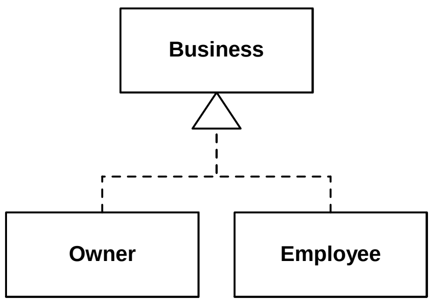


Figure 7.71 - An example of a realization dependency

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 93

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 60

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 60

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*