UML Semantics 2

The UML Semantics section is primarily intended as a comprehensive and precise specification of the UML's semantic constructs.

Contents

2 UML Semantics

Part 1 - Background

2.1 Introduction

2.1.1 Purpose and Scope

The primary audience for this detailed description consists of the OMG, other standards organizations, tool builders, metamodelers, methodologists, and expert modelers. The authors assume familiarity with metamodeling and advanced object modeling. Readers looking for an introduction to the UML or object modeling should consider another source.

Although the document is meant for advanced readers, it is also meant to be easily understood by its intended audience. Consequently, it is structured and written to increase readability. The structure of the document, like the language, builds on previous concepts to refine and extend the semantics. In addition, the document is written in a ‘semi-formal’ style that combines natural and formal languages in a complementary manner.

This section specifies semantics for structural and behavioral object models. Structural models (also known as static models) emphasize the structure of objects in a system, including their classes, interfaces, attributes and relations. Behavioral models (also known as dynamic models) emphasize the behavior of objects in a system, including their methods, interactions, collaborations, and state histories.

This section provides complete semantics for all modeling notations described in the UML Notation Guide (Chapter 3). This includes support for a wide range of diagram techniques: class diagram, object diagram, use case diagram, sequence diagram, collaboration diagram, state diagram, activity diagram, and deployment diagram. The UML Notation Guide includes a summary of the semantics sections that are relevant to each diagram technique.

2.1.2 Approach

This section emphasizes language architecture and formal rigor. The architecture of the UML is based on a four-layer metamodel structure, which consists of the following layers: user objects, model, metamodel, and meta-metamodel. This document is primarily concerned with the metamodel layer, which is an instance of the meta-metamodel layer. For example, Class in the metamodel is an instance of MetaClass in the meta-metamodel. The metamodel architecture of UML is discussed further in “Language Architecture” on page 2-4.

The UML metamodel is a logical model and not a physical (or implementation) model. The advantage of a logical metamodel is that it emphasizes declarative semantics, and suppresses implementation details. Implementations that use the logical metamodel must conform to its semantics, and must be able to import and export full as well as partial models. However, tool vendors may construct the logical metamodel in various ways, so they can tune their implementations for reliability and performance. The disadvantage of a logical model is that it lacks the imperative semantics required for accurate and efficient implementation. Consequently, the metamodel is accompanied with implementation notes for tool builders.

UML is also structured within the metamodel layer. The language is decomposed into several logical packages: Foundation, Behavioral Elements, and Model Management. These packages in turn are decomposed into subpackages. For example, the Foundation package consists of the Core, Extension Mechanisms, and Data Types subpackages. The structure of the language is fully described in “Language Architecture” on page 2-4.

The metamodel is described in a semi-formal manner using these views:

- Abstract syntax
- Well-formedness rules
- Semantics

The abstract syntax is provided as a model described in a subset of UML, consisting of a UML class diagram and a supporting natural language description. (In this way the UML bootstraps itself in a manner similar to how a compiler is used to compile itself.) The well-formedness rules are provided using a formal language (Object Constraint Language) and natural language (English). Finally, the semantics are described primarily in natural language, but may include some additional notation, depending on the part of the model being described. The adaptation of formal techniques to specify the language is fully described in “Language Formalism” on page 2-8.

In summary, the UML metamodel is described in a combination of graphic notation, natural language and formal language. We recognize that there are theoretical limits to what one can express about a metamodel using the metamodel itself. However, our experience suggests that this combination strikes a reasonable balance between expressiveness and readability.

2.2 Language Architecture

2.2.1 Four-Layer Metamodel Architecture

The UML metamodel is defined as one of the layers of a four-layer metamodeling architecture. This architecture is a proven infrastructure for defining the precise semantics required by complex models. There are several other advantages associated with this approach:

- It refines semantic constructs by recursively applying them to successive metalayers.
- It provides an architectural basis for defining future UML metamodel extensions.
- It furnishes an architectural basis for aligning the UML metamodel with other standards based on a four-layer metamodeling architecture, in particular the OMG Meta-Object Facility (MOF).

The generally accepted framework for metamodeling is based on an architecture with four layers:

- meta-metamodel
- metamodel
- model
- user objects

The functions of these layers are summarized in the following table.

Table 2-1 Four Layer Metamodeling Architecture

Layer	Description	Example
meta-metamodel	The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.	*MetaClass, MetaAttribute, MetaOperation*
metamodel	An instance of a meta-metamodel. Defines the language for specifying a model.	*Class, Attribute, Operation, Component*
model	An instance of a metamodel. Defines a language to describe an information domain.	*StockShare, askPrice, sellLimitOrder, StockQuoteServer*
user objects (user data)	An instance of a model. Defines a specific information domain.	*<Acme_SW_Share_98789>, 654.56, sell_limit_order, <Stock_Quote_Svr_32123>*

The meta-metamodeling layer forms the foundation for the metamodeling architecture. The primary responsibility of this layer is to define the language for specifying a metamodel. A meta-metamodel defines a model at a higher level of abstraction than a metamodel, and is typically more compact than the metamodel that it describes. A meta-metamodel can define multiple metamodels, and there can be multiple meta-metamodels associated with each metamodel.

While it is generally desirable that related metamodels and meta-metamodels share common design philosophies and constructs, this is not a strict rule. Each layer needs to maintain its own design integrity. Examples of meta-metaobjects in the meta-metamodeling layer are: MetaClass, MetaAttribute, and MetaOperation.

A metamodel is an instance of a meta-metamodel. The primary responsibility of the metamodel layer is to define a language for specifying models. Metamodels are typically more elaborate than the meta-metamodels that describe them, especially when they define dynamic semantics. Examples of metaobjects in the metamodeling layer are: Class, Attribute, Operation, and Component.

A model is an instance of a metamodel. The primary responsibility of the model layer is to define a language that describes an information domain. Examples of objects in the modeling layer are: StockShare, askPrice, sellLimitOrder, and StockQuoteServer.

User objects (a.k.a. user data) are an instance of a model. The primary responsibility of the user objects layer is to describe a specific information domain. Examples of objects in the user objects layer are: <Acme_Software_Share_98789>, 654.56, sell_limit_order, and <Stock_Quote_Svr_32123>.

Architectural Alignment with the MOF Meta-Metamodel

Both the UML and the MOF are based on a four-layer metamodel architecture, where the MOF meta-metamodel is the meta-metamodel for the UML metamodel. Since the MOF and UML have different scopes and differ in their abstraction levels (the UML metamodel tends to be more of a logical model than the MOF meta-metamodel), they are related by loose metamodeling rather than strict metamodeling.[1] As a result, the UML metamodel is an instance of the MOF meta-metamodel.

Consequently, there is not a strict isomorphic instance-of mapping between all the MOF meta-metamodel elements and the UML metamodel elements. In spite of this, since the two models were designed to be interoperable, the UML Core package metamodel and the MOF meta-metamodel are structurally quite similar.

2.2.2 Package Structure

The UML metamodel is moderately complex. It is composed of approximately 90 metaclasses and over 100 metaassociations, and includes almost 50 stereotypes. The complexity of the metamodel is managed by organizing it into logical packages. These packages group metaclasses that show strong cohesion with each other and loose coupling with metaclasses in other packages. The UML metamodel is decomposed into the top-level packages shown in Figure 2-1 on page -6.

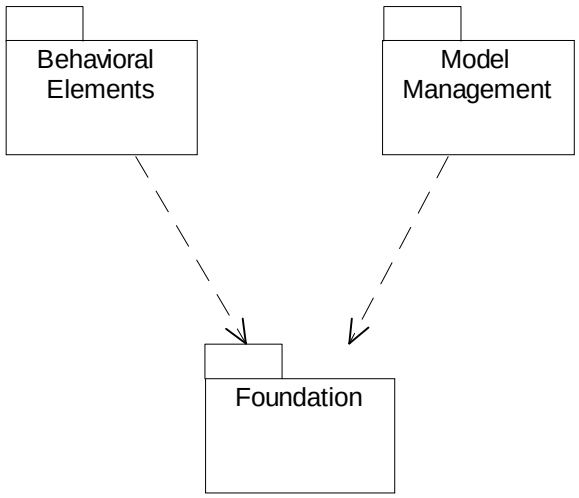


Figure 2-1 Top-Level Packages

1. In loose (or “non-strict”) metamodeling a M_n level model is an instance of a M_{n+1} level model. In strict metamodeling, every element of a M_n level model is an instance of exactly one element of M_{n+1} level model.

The Foundation and Behavioral Elements packages are further decomposed as shown in Figure 2-2 and Figure 2-3 on page -7.

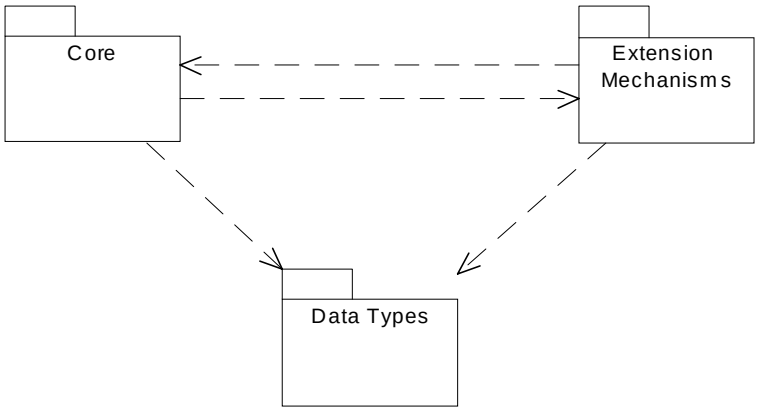


Figure 2-2 Foundation Packages

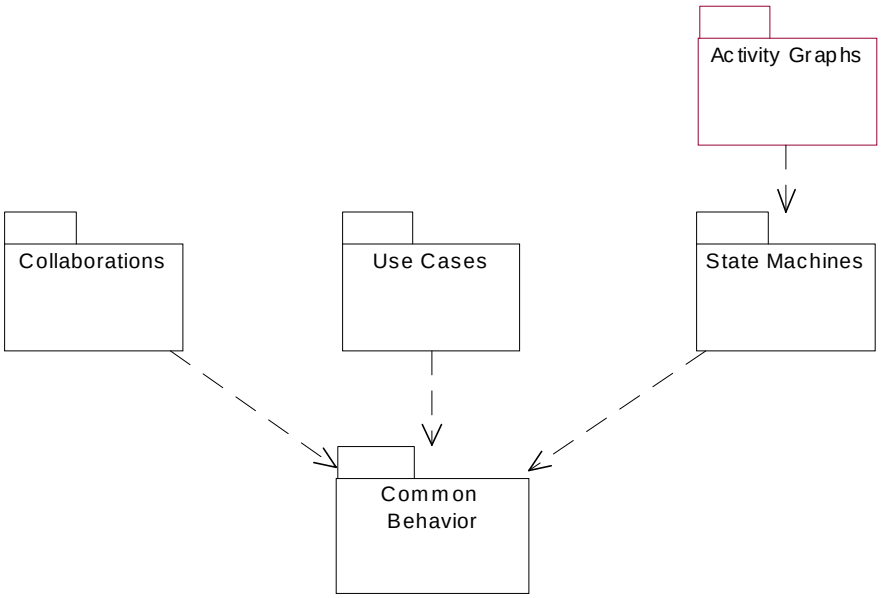


Figure 2-3 Behavioral Elements Packages

The functions and contents of these packages are described in this chapter's Part 3, Behavioral Elements.

2.3 Language Formalism

This section contains a description of the techniques used to describe UML. The specification adapts formal techniques to improve precision while maintaining readability. The technique describes the UML metamodel in three views using both text and graphic presentations. The benefits of adapting formal techniques include:

- the correctness of the description is improved,
- ambiguities and inconsistencies are reduced,
- the architecture of the metamodel is validated by a complementary technique, and
- the readability of the description is increased.

It is important to note that the current description is not a completely formal specification of the language because to do so would have added significant complexity without clear benefit. In addition, the state of the practice in formal specifications does not yet address some of the more difficult language issues that UML introduces.

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

2.3.1 Levels of Formalism

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

The static semantics of a language define how an instance of a construct should be connected to other instances to be meaningful, and the dynamic semantics define the meaning of a well-formed construct. The meaning of a description written in the language is defined only if the description is well formed (i.e., if it fulfills the rules defined in the static semantics). The static semantics are found in sections headed Well-Formedness Rules. The dynamic semantics are described under the heading Semantics. In some cases, parts of the static semantics are also explained in the Semantics section for completeness.

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

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

2.3.2 Package Specification Structure

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

Abstract Syntax

The abstract syntax is presented in a UML class diagram showing the metaclasses defining the constructs and their relationships. The diagram also presents some of the well-formedness rules, mainly the multiplicity requirements of the relationships, and whether or not the instances of a particular sub-construct must be ordered. Finally, a short informal description in natural language describing each construct is supplied. The first paragraph of each of these descriptions is a general presentation of the construct which sets the context, while the following paragraphs give the informal definition of the metaclass specifying the construct in UML. For each metaclass, its attributes are enumerated together with a short explanation. Furthermore, the opposite role names of associations connected to the metaclass are also listed in the same way.

Well-Formedness Rules

The static semantics of each construct in UML, except for multiplicity and ordering constraints, are defined as a set of invariants of an instance of the metaclass. These invariants have to be satisfied for the construct to be meaningful. The rules thus specify constraints over attributes and associations defined in the metamodel. Each invariant is defined by an OCL expression together with an informal explanation of the expression. In many cases, additional operations on the metaclasses are needed for the OCL expressions. These are then defined in a separate subsection after the well-formedness rules for the construct, using the same approach as the abstract syntax: an informal explanation followed by the OCL expression defining the operation.

The statement ‘No extra well-formedness rules’ means that all current static semantics are expressed in the superclasses together with the multiplicity and type information expressed in the diagrams.

---

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

Semantics

The meanings of the constructs are defined using natural language. The constructs are grouped into logical chunks that are defined together. Since only concrete metaclasses have a true meaning in the language, only these are described in this section.

Standard Elements

Stereotypes of the metaclasses defined previously in the section are listed, with an informal definition in natural language. Well-formedness rules, if any, for the stereotypes are also defined in the same manner as in the Well-Formedness Rules subsection.

Other kinds of standard elements (constraints and tagged-values) are listed, and are defined in the Standard Elements appendix.

Notes

This subsection may contain rationales for metamodeling decisions, pragmatics for the use of the constructs, and examples, all written in natural language.

2.3.3 Use of a Constraint Language

The specification uses the Object Constraint Language (OCL), as defined in Object Constraint Language Specification (Chapter 4), for expressing well-formedness rules. The following conventions are used to promote readability:

- Self - which can be omitted as a reference to the metaclass defining the context of the invariant, has been kept for clarity.
- In expressions where a collection is iterated, an iterator is used for clarity, even when formally unnecessary. The type of the iterator is usually omitted, but included when it adds to understanding.
- The ‘collect’ operation is left implicit where this is practical.

2.3.4 Use of Natural Language

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

The following general rules apply:

- When referring to an instance of some metaclass, we often omit the word "instance." For example, instead of saying "a Class instance" or "an Association instance," we just say "a Class" or "an Association." By prefixing it with an "a" or "an," assume that we mean "an instance of." In the same way, by saying something like "Elements" we mean "a set (or the set) of instances of the metaclass Element."

- Every time a word coinciding with the name of some construct in UML is used, that construct is referenced.
- Terms including one of the prefixes sub, super, or meta are written as one word (e.g., metamodel, subclass).

2.3.5 Naming Conventions and Typography

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

- When referring to constructs in UML, not their representation in the metamodel, normal text is used.
- Metaclass names that consist of appended nouns/adjectives, initial embedded capitals are used (e.g., ‘ModelElement,’ ‘StructuralFeature’).
- Names of metaassociations/association classes are written in the same manner as metaclasses (e.g., ‘ElementReference’).
- Initial embedded capital is used for names that consist of appended nouns/adjectives (e.g., ‘ownedElement,’ ‘allContents’).
- Boolean metaattribute names always start with ‘is’ (e.g., ‘isAbstract’).
- Enumeration types always end with “Kind” (e.g., ‘AggregationKind’).
- While referring to metaclasses, metaassociations, metaattributes, etc. in the text, the exact names as they appear in the model are always used.
- Names of stereotypes are delimited by guillemets and begin with lowercase (e.g., «type»).

2 UML Semantics

Part 2 - Foundation

The Foundation package is the infrastructure for UML. The Foundation package is decomposed into several subpackages: Core, Extension Mechanisms, and Data Types.

2.4 Foundation Package

Figure 2-4 illustrates the Foundation Packages. The Core package specifies the basic concepts required for an elementary metamodel and defines an architectural backbone for attaching additional language constructs, such as metaclasses, metaassociations, and metaattributes. The Auxiliary Elements package defines additional constructs that extend the Core to support advanced concepts such as dependencies, templates, physical structures and view elements. The Extension Mechanisms package specifies how model elements are customized and extended with new semantics. The Data Types package defines basic data structures for the language.

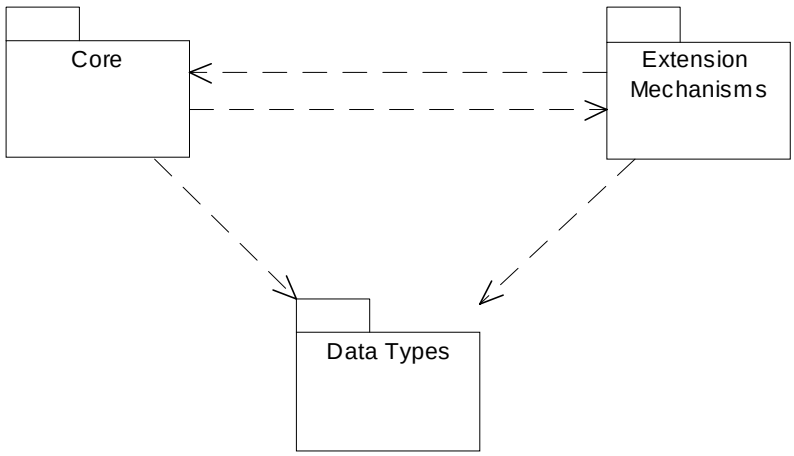


Figure 2-4 Foundation Packages

2.5 Core

2.5.1 Overview

The Core package is the most fundamental of the subpackages that compose the UML Foundation package. It defines the basic abstract and concrete constructs needed for the development of object models. Abstract metamodel constructs are not instantiable and are commonly used to reify key constructs, share structure, and organize the model. Concrete metamodel constructs are instantiable and reflect the modeling constructs used by object modelers (cf. metamodelers). Abstract constructs defined in the Core include ModelElement, GeneralizableElement, and Classifier. Concrete constructs specified in the Core include Class, Attribute, Operation, and Association.

The Core package specifies the core constructs required for a basic metamodel and defines an architectural backbone (“skeleton”) for attaching additional language constructs such as metaclasses, metaassociations, and metaattributes. Although the Core package contains sufficient semantics to define the remainder of UML, it is not the UML meta-metamodel. It is the underlying base for the Foundation package, which in turn serves as the infrastructure for the rest of language. In other packages, the Core is extended by adding metaclasses to the backbone using generalizations and associations.

The following sections describe the abstract syntax, well-formedness rules, and semantics of the Core package.

2.5.2 Abstract Syntax

The abstract syntax for the Core package is expressed in graphic notation in the following figures. Figure 2-5 on page 2-14 shows the model elements that form the structural backbone of the metamodel. Figure 2-6 on page 2-15 shows the model elements that define relationships. Figure 2-7 on page 2-16 shows the model elements that define dependencies. Figure 2-8 on page 2-17 shows the various kinds of classifiers. Figure 2-9 on page 2-18 shows auxiliary elements for bindings, presentation and comments.

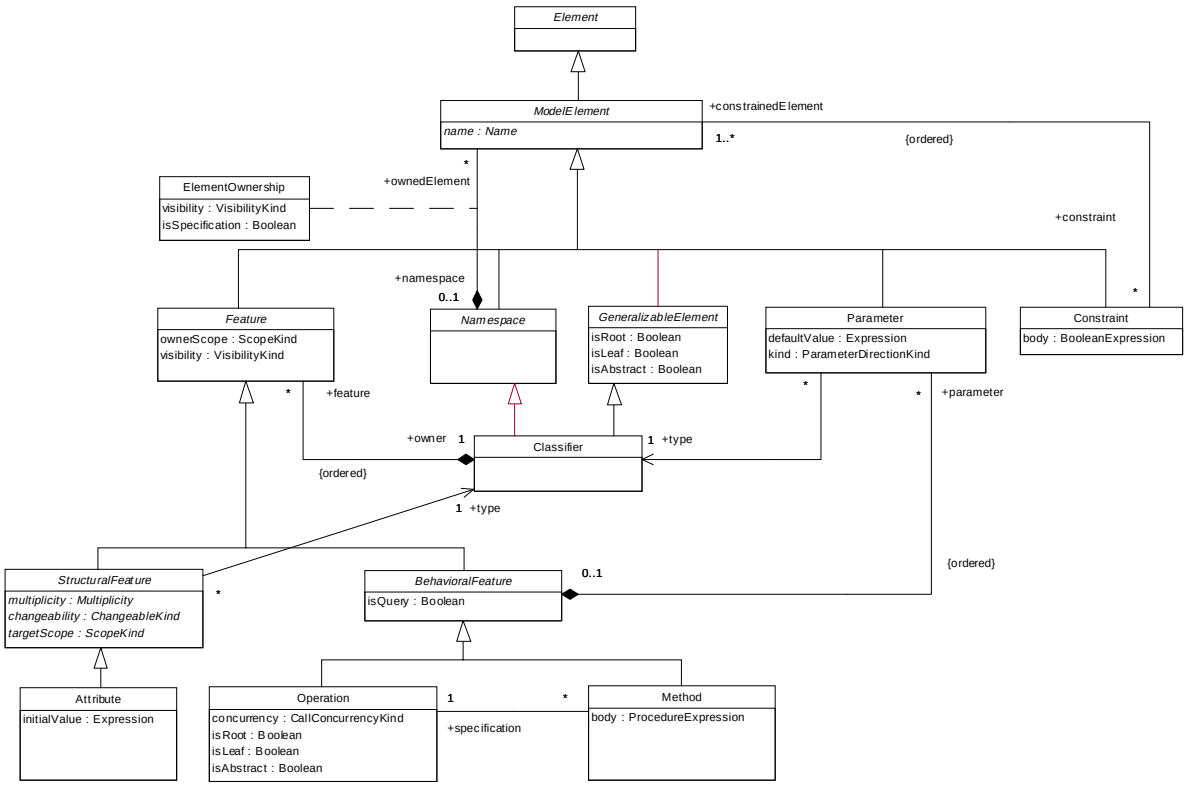


Figure 2-5 Core Package - Backbone

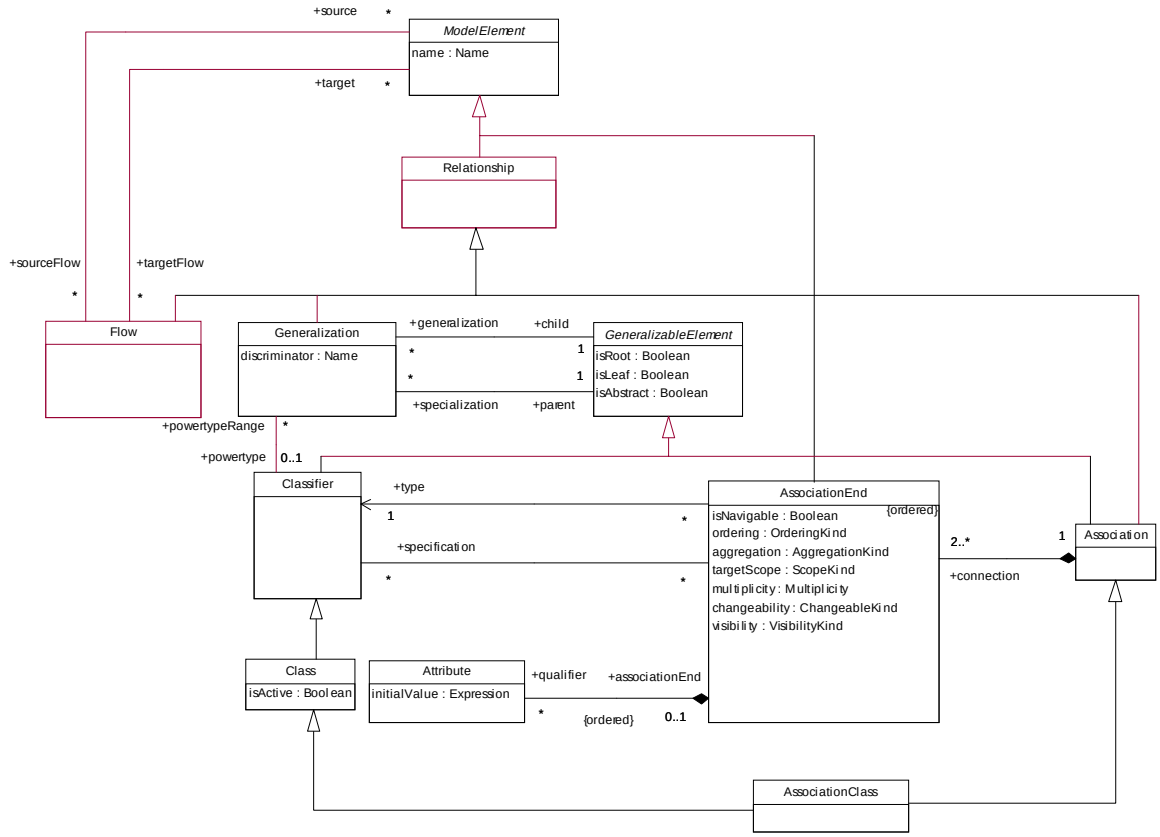


Figure 2-6 Core Package - Relationships

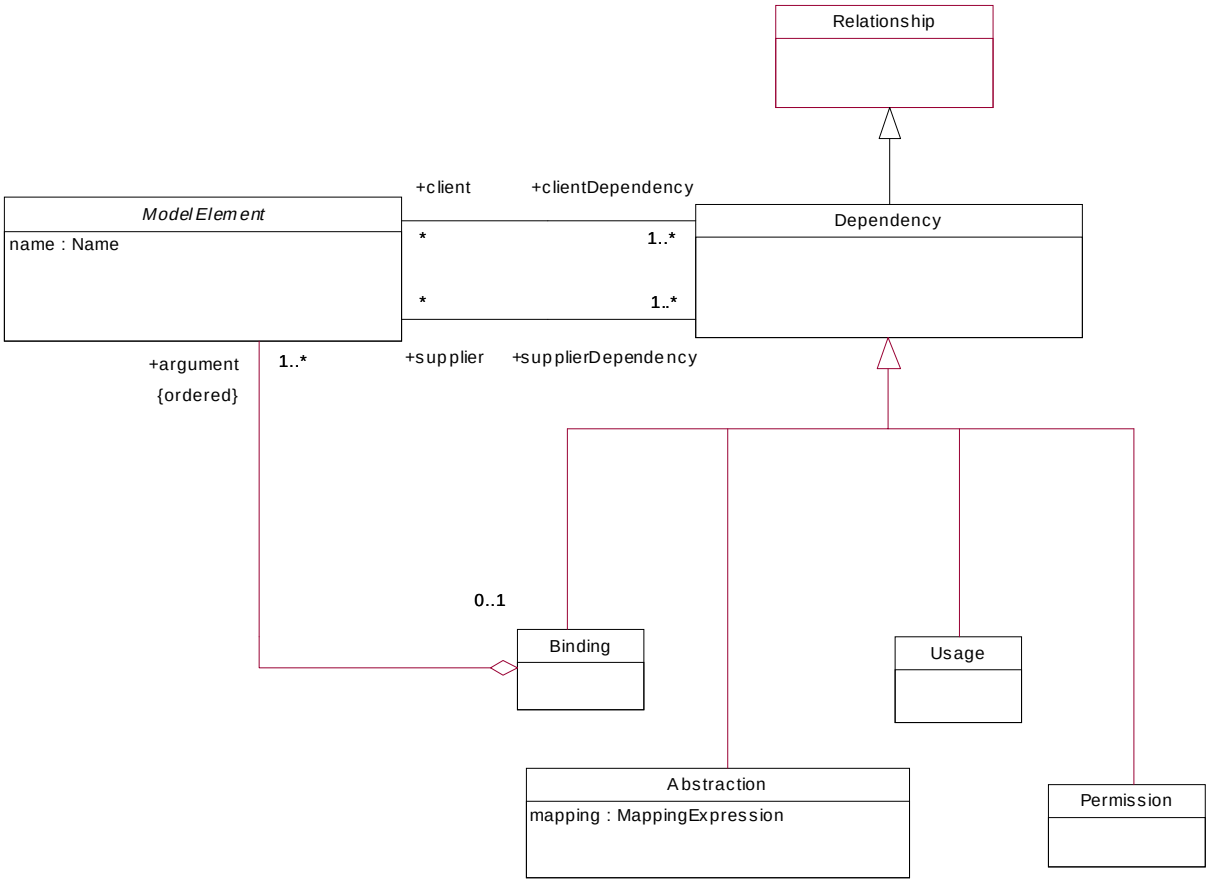


Figure 2-7 Core Package - Dependencies

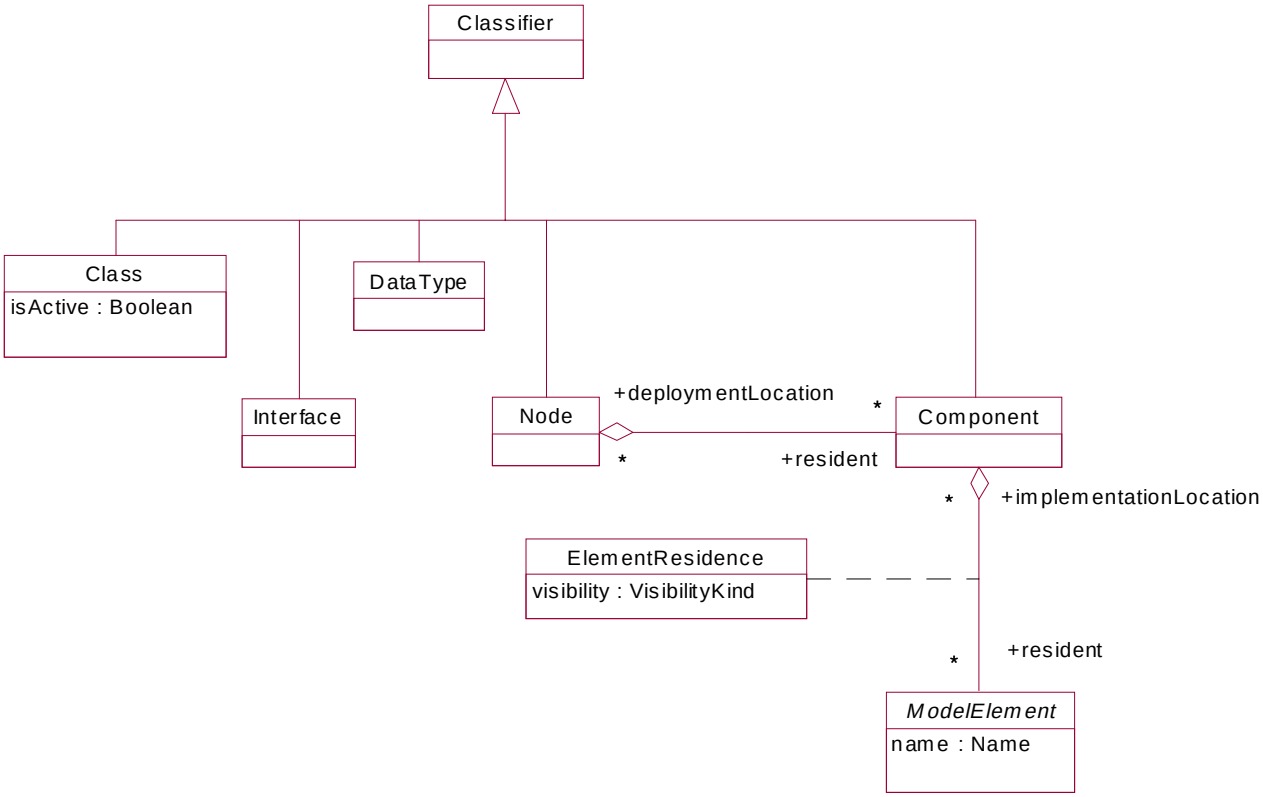


Figure 2-8 Core Package - Classifiers

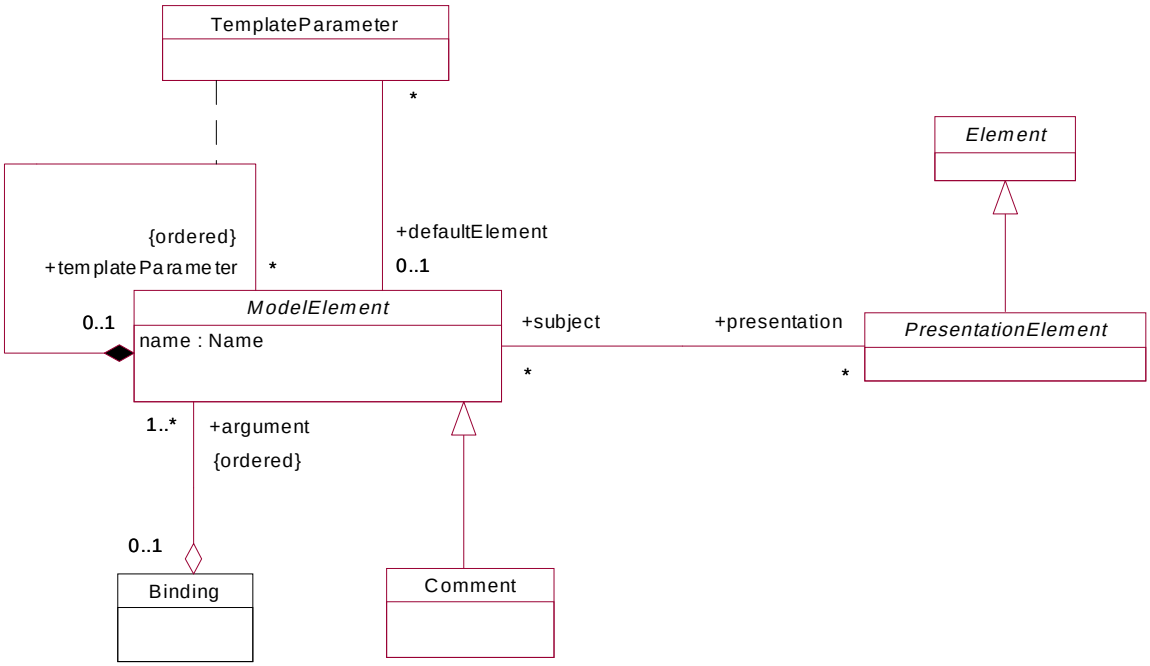


Figure 2-9 Core Package - Auxiliary elements

Abstraction

An abstraction is a Dependency relationship that relates two 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 in a mapping between the supplier and the client. Depending on the specific stereotype of Abstraction, the mapping may be formal or informal, and it may be unidirectional or bidirectional.

The stereotypes of Abstraction are Derivation, Refinement, and Trace.

Attributes

mapping	A MappingExpression that states the 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 may be omitted if the precise relationship between the elements is not specified.

Stereotypes

«derive» Derivation	Derived is a stereotyped abstraction dependency whose client and supplier are both elements, usually but not necessarily of the same type. A derived dependency specifies that the client may be computed from the supplier. The mapping specifies the computation. The client may be implemented for design reasons, such as efficiency, even though it is logically redundant.
«realize» Realization	A realization is a relationship between a specification model element (the supplier) and a model element that implements it (the client). The implementation model element is required to support all of the operations or received signals that the specification model element declares. The implementation model element must make or inherit its own declarations of the operations and signal receptions. The mapping specifies the relationship between the two. The mapping may or may not be computable. Realization can be used to model stepwise refinement, optimizations, transformations, templates, model synthesis, framework composition, etc.
«refine» Refinement	A refinement is a relationship between model elements at different semantic levels, such as analysis and design. The mapping specifies the relationship between the two. The mapping may or may not be computable, and it may be unidirectional or bidirectional. Refinement can be used to model transformations from analysis to design and other such changes.
«trace»	Trace is a stereotyped abstraction dependency that denotes that the client and supplier represent the same concept in different models. Traces are mainly used for tracking requirements and changes across models. Since model changes can occur in both directions, the directionality of the dependency can often be ignored. The mapping specifies the relationship between the two, but it is rarely computable and is usually informal.

Association

An association defines a semantic relationship between classifiers. The instances of an association are a set of tuples relating instances of the classifiers. Each tuple value may appear at most once.

In the metamodel, an Association is a declaration of a semantic relationship between Classifiers, such as Classes. An Association has at least two AssociationEnds. Each end is connected to a Classifier - the same Classifier may be connected to more than one AssociationEnd in the same Association. The Association represents a set of connections among instances of the Classifiers. An instance of an Association is a Link, which is a tuple of Instances drawn from the corresponding Classifiers.

Attributes

name	The name of the Association which, in combination with its associated Classifiers, must be unique within the enclosing namespace (usually a Package).

Associations

connection	An Association consists of at least two AssociationEnds, each of which represents a connection of the association to a Classifier. Each AssociationEnd specifies a set of properties that must be fulfilled for the relationship to be valid. The bulk of the structure of an Association is defined by its AssociationEnds.

Stereotypes

implicit	Implicit is a stereotype applied to an association, specifying that the association is not manifest, but rather is only conceptual.

Standard Constraints

xor	Xor is a constraint applied to a set of associations, specifying that over that set, only one is manifest for each associated instance. Xor is an exclusive or (not inclusive or) constraint.

Tagged Values

persistence	Persistence denotes the permanence of the state of the association, marking it as transitory (its state is destroyed when the instance is destroyed) or persistent (its state is not destroyed when the instance is destroyed).

AssociationClass

An association class is an association that is also a class. It not only connects a set of classifiers but also defines a set of features that belong to the relationship itself and not any of the classifiers.

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 a subclass of both Association and Class (i.e., each AssociationClass is both an Association and a Class); therefore, an AssociationClass has both AssociationEnds and Features.

AssociationEnd

An association end is an endpoint of an association, which connects the association to a classifier. Each association end is part of one association. The association-ends of each association are ordered.

In the metamodel an AssociationEnd is part of an Association and specifies the connection of an Association to a Classifier. It has a name and defines a set of properties of the connection (e.g., which Classifier the Instances must conform to, their multiplicity, and if they may be reached from another Instance via this connection).

In the following descriptions when referring to an association end for a binary association, the source end is the other end. The target end is the one whose properties are being discussed.

Attributes

aggregation — When placed on a target end, specifies whether the target end is an aggregation with respect to the source end. Only one end can be an aggregation. Possibilities are:

- none - The end is not an aggregate.
- aggregate - The end is an aggregate; therefore, the other end is a part and must have the aggregation value of none. The part may be contained in other aggregates.
- composite - The end is a composite; therefore, the other end is a part and must have the aggregation value of none. The part is strongly owned by the composite and may not be part of any other composite.

changeability — When placed on a target end, specifies whether an instance of the Association may be modified from the source end. Possibilities are:

- changeable - No restrictions on modification.
- frozen - No links may be added after the creation of the source object.
- addOnly - Links may be added at any time from the source object, but once created a link may not be removed from the source end.

ordering — When placed on a target end, specifies whether the set of links from the source instance to the target instance is ordered. The ordering must be determined and maintained by Operations that add links. It represents additional information not inherent in the objects or links themselves. Possibilities are:

- unordered - The links form a set with no inherent ordering.
- ordered - A set of ordered links can be scanned in order.
- Other possibilities (such as sorted) may be defined later by declaring additional keywords. As with user-defined stereotypes, this would be a private extension supported by particular editing tools.

isNavigable	When placed on a target end, specifies whether traversal from a source instance to its associated target instances is possible. Specification of each direction across the Association is independent. A value of true means that the association can be navigated by the source class and the target rolename can be used in navigation expressions.
multiplicity	When placed on a target end, specifies the number of target instances that may be associated with a single source instance across the given Association.
name	(Inherited from ModelElement) The rolename of the end. When placed on a target end, provides a name for traversing from a source instance across the association to the target instance or set of target instances. It represents a pseudo-attribute of the source classifier (i.e., it may be used in the same way as an Attribute) and must be unique with respect to Attributes and other pseudo-attributes of the source Classifier.
targetScope	Specifies whether the target value is an instance or a classifier. Possibilities are: • instance. An instance value is part of each link. This is the default. • classifier. A classifier itself is part of each link. Normally this would be fixed at modeling time and need not be stored separately at run time.
visibility	Specifies the visibility of the association end from the viewpoint of the classifier on the other end. Possibilities are: • public - Other classifiers may navigate the association and use the rolename in expressions, similar to the use of a public attribute. • protected - Descendants of the source classifier may navigate the association and use the rolename in expressions, similar to the use of a protected attribute. • private - Only the source classifier may navigate the association and use the rolename in expressions, similar to the use of a private attribute.

Associations

qualifier	An optional list of qualifier Attributes for the end. If the list is empty, then the Association is not qualified.

specification	Designates zero or more Classifiers that specify the Operations that may be applied to an Instance accessed by the AssociationEnd across the Association. These determine the minimum interface that must be realized by the actual Classifier attached to the end to support the intent of the Association. May be an Interface or another Classifier.
type	Designates the Classifier connected to the end of the Association. In a link, the actual class may be a descendant of the nominal class or (for an Interface) a Class that realizes the declared type.

Stereotypes

«association»	Specifies a real association (default and redundant, but may be included for emphasis).
«global»	Specifies that the target is a global value that is known to all elements rather than an actual association.
«local»	Specifies that the relationship represents a local variable within a procedure rather than an actual association.
«parameter»	Specifies that the relationship represents a procedure parameter rather than an actual association.
«self»	Specifies that the relationship represents a reference to the object that owns an operation or action rather than an actual association.

Attribute

An attribute is a named slot within a classifier that describes a range of values that instances of the classifier may hold.

In the metamodel an Attribute is a named piece of the declared state of a Classifier, particularly the range of values that Instances of the Classifier may hold.

(The following list includes properties from StructuralFeature which has no other subclasses in the current metamodel.)

Attributes

changeability — Whether the value may be modified after the object is created. Possibilities are:

- changeable - No restrictions on modification.
- frozen - The value may not be altered after the object is instantiated and its values initialized. No additional values may be added to a set.
- AddOnly - Meaningful only if the multiplicity is not fixed to a single value. Additional values may be added to the set of values, but once created a value may not be removed or altered.

initial value — An Expression specifying the value of the attribute upon initialization. It is meant to be evaluated at the time the object is initialized. (Note that an explicit constructor may supersede an initial value.)

multiplicity — The possible number of data values for the attribute that may be held by an instance. The cardinality of the set of values is an implicit part of the attribute. In the common case in which the multiplicity is 1..1, then the attribute is a scalar (i.e., it holds exactly one value).

targetScope — Specifies whether the targets are ordinary Instances or are Classifiers. Possibilities are:

- instance - Each value contains a reference to an Instance of the target Classifier. This is the setting for a normal Attribute.
- classifier - Each value contains a reference to the target Classifier itself. This represents a way to store meta-information.

Associations

type — Designates the classifier whose instances are values of the attribute. Must be a Class, Interface, or DataType. The actual type may be a descendant of the declared type or (for an Interface) a Class that realizes the declared type.

Tagged Values

persistence — Persistence denotes the permanence of the state of the attribute, marking it as transitory (its state is destroyed when the instance is destroyed) or persistent (its state is not destroyed when the instance is destroyed).

BehavioralFeature

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

In the metamodel a BehavioralFeature specifies a behavioral aspect of a Classifier. All different kinds of behavioral aspects of a Classifier, such as Operation and Method, are subclasses of BehavioralFeature. BehavioralFeature is an abstract metaclass.

Attributes

isQuery	Specifies whether an execution of the Feature leaves the state of the system unchanged. True indicates that the state is unchanged; false indicates that side-effects may occur.
name	(Inherited from ModelElement) The name of the Feature. The entire signature of the Feature (name and parameter list) must be unique within its containing Classifier.

Associations

parameter	An ordered list of Parameters for the Operation. To call the Operation, the caller must supply a list of values compatible with the types of the Parameters.

Stereotypes

«create»	Create is a stereotyped behavioral feature denoting that the designated feature creates an instance of the classifier to which the feature is attached.
«destroy»	Delete is a stereotyped behavioral feature denoting that the designated feature destroys an instance of the classifier to which the feature is attached.

Binding

A binding is a relationship between a template and a model element generated from the template. It includes a list of arguments matching the template parameters. The template is a form that is cloned and modified by substitution to yield an implicit model fragment that behaves as if it were a direct part of the model.

In the metamodel a Binding is a Dependency where the supplier is the template and the client is the instantiation of the template that performs the substitution of parameters of a template. A Binding has a list of arguments that replace the parameters of the supplier to yield the client. The client is fully specified by the binding of the supplier’s parameters and does not add any information of its own.

Associations

argument	An ordered list of arguments. Each argument replaces the corresponding supplier parameter in the supplier definition, and the result represents the definition of the client as if it had been defined directly.

Class

A class is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment.

In the metamodel a Class describes a set of Objects sharing a collection of Features, including Operations, Attributes and Methods, that are common to the set of Objects. Furthermore, a Class may realize zero or more Interfaces; this means that its full descriptor (see “Inheritance” on page 2-59 for the definition) must contain every Operation from every realized Interface (it may contain additional operations as well).

A Class defines the data structure of Objects, although some Classes may be abstract (i.e., no Objects can be created directly from them). Each Object instantiated from a Class contains its own set of values corresponding to the StructuralFeatures declared in the full descriptor. Objects do not contain values corresponding to BehavioralFeatures or class-scope Attributes; all Objects of a Class share the definitions of the BehavioralFeatures from the Class, and they all have access to the single value stored for each class-scope attribute.

Attributes

isActive	Specifies whether an Object of the Class maintains its own thread of control. If true, then an Object has its own thread of control and runs concurrently with other active Objects. If false, then Operations run in the address space and under the control of the active Object that controls the caller.

Stereotypes

«implementationClass»	Implementation class is a stereotyped class that is not a type and that represents the implementation of a class in some programming language. An instance may have zero or one implementation classes. This is in contrast to plain general classes, wherein an instance may statically have multiple classes at one time and may gain or lose classes over time and an object (a child of instance) may dynamically have multiple classes.
«type»	Type is a stereotype of Class, meaning that the class is used for specification of a domain of instances (objects) together with the operations applicable to the objects. A type may not contain any methods, but it may have attributes and associations.

Classifier

A classifier is an element that describes behavioral and structural features; it comes in several specific forms, including class, data type, interface, component, and others that are defined in other metamodel packages.

In the metamodel, a Classifier declares a collection of Features, such as Attributes, Methods, and Operations. It has a name, which is unique in the Namespace enclosing the Classifier. Classifier is an abstract metaclass.

Classifier is a child of GeneralizableElement and Namespace. As a GeneralizableElement, it may inherit Features and participation in Associations (in addition to things inherited as a ModelElement). It also inherits ownership of StateMachines, Collaborations, etc.

As a Namespace, a Classifier may declare other Classifiers nested in its scope. Nested Classifiers may be accessed by other Classifiers only if the nested Classifiers have adequate visibility. There are no data value or state consequences of nested Classifiers, i.e., it is not an aggregation or composition.

Associations

feature	An ordered list of Features, like Attribute, Operation, Method, owned by the Classifier.
participant	Inverse of specification on association to AssociationEnd. Denotes that the Classifier participates in an Association.
powertypeRange	Designates zero or more Generalizations for which the Classifier is a powertype. If the cardinality is zero, then the Classifier is not a powertype; if the cardinality is gerater than zero, then the Classifier is a powertype over the set of Generalizations designated by the association, and the child elements of the Generalizations are the instances of the Classifier as a powertype. A Classifier that is a powertype can be marked with the «powertype» stereotype.

Stereotypes

«metaclass»	Metaclass is a stereotyped classifier whose instances are classes.
«powertype»	Powertype is a stereotyped classifier denoting that the classifier is a metatype, whose instances are children marked by the same discriminator.
«process»	Process is a stereotyped classifier that is also an active class, representing a heavy-weight flow of control.
«thread»	Thread is a stereotyped classifier that is also an active class, representing a light-weight flow of control.
«utility»	Utility is a stereotyped classifier representing a classifier that has no instances, but rather denotes a named collection of non-member attributes and operations, all of which are class-scoped.

Tagged Values

persistence	Persistence denotes the permanence of the state of the classifier, marking it as transitory (its state is destroyed when the instance is destroyed) or persistent (its state is not destroyed when the instance is destroyed).
semantics	Semantics is the specification of the meaning of the classifier.

Comment

A comment is an annotation attached to a model element or a set of model elements. It has no semantic force but may contain information useful to the modeler.

Stereotypes

«requirement»	Requirement is a stereotyped comment that states a responsibility or obligation.
«responsibility»	Responsibility is a stereotyped comment that describes a contract or an obligation of a classifier.

Component

A component is a physical, replaceable part of a system that packages implementation and conforms to and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files. As such, a Component may itself conform to and provide the realization of a set of interfaces, which represent services implemented by the elements resident in the component. These services define behavior offered by instances of the Component as a whole to other client Component instances.

In the metamodel a Component is a subclass of Classifier. It provides the physical packaging of its associated specification elements. As a Classifier, it may also have its own Features, such as Attributes and Operations, and realize Interfaces.

Associations

deploymentLocation	The set of Nodes the Component is residing on.
resident	(Association class ElementResidence) The set of model elements that the component supports. The visibility attribute shows the external visibility of the element outside the component.

Stereotypes

«document»	Document is a stereotyped component representing a document.
«executable»	Executable is a stereotyped component denoting a program that may be run on a node.
«file»	File is a stereotyped component representing a document containing source code or data.
«library»	Library is a stereotyped component representing a static or dynamic library.
«table»	Table is a stereotyped component representing a data base table.

Constraint

A constraint is a semantic condition or restriction expressed in text.

In the metamodel a Constraint is a BooleanExpression on an associated ModelElement(s) which must be true for the model to be well formed. This restriction can be stated in natural language, or in different kinds of languages with a well-defined semantics. Certain Constraints are predefined in the UML, others may be user defined. Note that a Constraint is an assertion, not an executable mechanism. It indicates a restriction that must be enforced by correct design of a system.

Attributes

body	A BooleanExpression that must be true when evaluated for an instance of a system to be well-formed.

Associations

constrainedElement	A ModelElement or list of ModelElements affected by the Constraint. If the constrained element is a Stereotype, then the constraint applies to all ModelElements that use the stereotype.

Stereotypes

«invariant»	Invariant is a stereotyped constraint that must be attached to a set of classifiers or relationships, and denotes that the conditions of the constraint must hold for the classifiers or relationships and their instances.
«postcondition»	Postcondition is a stereotyped constraint that must be attached to an operation, and denotes that the conditions of the constraint must hold after the invocation of the operation.
«precondition»	Precondition is a stereotyped constraint that must be attached to an operation, and denotes that the conditions of the constraint must hold for the invocation of the operation.

DataType

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

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

Dependency

A term of convenience for a Relationship other than Association, Generalization, Flow, or metarelationship (such as the relationship between a Classifier and one of its Instances).

A dependency states that the implementation or functioning of one or more elements requires the presence of one or more other elements. All of the elements must exist at the same level of meaning (i.e., they do not involve a shift in the level of abstraction or realization).

In the metamodel, a Dependency is a directed relationship from a client (or clients) to a supplier (or suppliers) stating that the client is dependent on the supplier (i.e., the client element requires the presence and knowledge of the supplier element).

The kinds of Dependency are Abstraction, Binding, Permission, and Usage. Various stereotypes of those elements are predefined.

Associations

client	The element that is affected by the supplier element. In some cases (such as a trace Abstraction) the direction is unimportant and serves only to distinguish the two elements.
supplier	Inverse of client. Designates the element that is unaffected by a change. In a two-way relationship (such as some refinement Abstractions) this would be the more general element. In an undirected situation, such as a trace Abstraction, the choice of client and supplier may be irrelevant.

Element

An element is an atomic constituent of a model.

In the metamodel, an Element is the top metaclass in the metaclass hierarchy. It has two subclasses: ModelElement and PresentationElement. Element is an abstract metaclass.

Tagged Values

documentation	Documentation is a comment, description, or explanation of the element to which it is attached.

ElementOwnership

Element ownership defines the visibility of a ModelElement contained in a Namespace.

In the metamodel, ElementOwnership reifies the relationship between ModelElement and Namespace denoting the ownership of a ModelElement by a Namespace and its visibility outside the Namespace. See “ModelElement” on page 2-36.

Attributes

isSpecification	Specifies whether the ownedElement is part of the specification for the containing namespace (in cases where specification is distinguished from the realization). Otherwise the ownedElement is part of the realization. In cases in which the distinction is not made, the value is false by default.
visibility	Specifies whether the ModelElement can be seen and referenced by other ModelElements. Possibilities: • public - Any outside ModelElement can see the ModelElement. • protected - Any descendent of the ModelElement can see the ModelElement. • private - Only the ModelElement itself, its constituent parts, or elements nested within it can see the ModelElement. Note that use of an element in another Package may also be subject to access or import of its Package as described in Model Management; see Permission.

ElementResidence

Association class between Component and ModelElement. See Component::resident. Shows that the component supports the element.

Attributes

visibility	Specifies whether the ModelElement can be used by other Components. Possibilities: • public - Any outside Component can use the ModelElement. • protected - Any descendent of the Component can use the ModelElement. • private - Only the Component itself can use the ModelElement.

Feature

A feature is a property, like operation or attribute, which is encapsulated within a Classifier.

In the metamodel a Feature declares a behavioral or structural characteristic of an Instance of a Classifier or of the Classifier itself. Feature is an abstract metaclass.

Attributes

name	(Inherited from ModelElement) The name used to identify the Feature within the Classifier or Instance. It must be unique across inheritance of names from ancestors including names of outgoing AssociationEnds.
ownerScope	Specifies whether Feature appears in each Instance of the Classifier or whether there is just a single instance of the Feature for the entire Classifier. Possibilities are: • instance - Each Instance of the Classifier holds its own value for the Feature. • classifier - There is just one value of the Feature for the entire Classifier.
visibility	Specifies whether the Feature can be used by other Classifiers. Visibilities of nested Classifiers combine so that the most restrictive visibility is the result. Possibilities: • public - Any outside Classifier with visibility to the Classifier can use the Feature. • protected - Any descendent of the Classifier can use the Feature. • private - Only the Classifier itself can use the Feature.

Associations

owner	The Classifier declaring the Feature.

Flow

A flow is a relationship between two versions of an object or between an object and a copy of it.

In the metamodel a Flow is a child of Relationship. A Flow is a directed relationship from a source or sources to a target or targets. It usually connects an activity to or from an object flow state, or tow object flow states. It can also connect from a fork or to a branch.

Predefined stereotypes of Flow are «become» and «copy». Become relates one version of an object to another with a different value, state, or location. Copy relates an object to another object that starts as a copy of it.

Stereotypes

«become»	Become is a stereotyped flow dependency whose source and target represent the same instance at different points in time, but each with potentially different values, state instance, and roles. A become dependency from A to B means that instance A becomes B with possibly new values, state instance, and roles at a different moment in time/space.
«copy»	Copy is a stereotyped flow dependency whose source and target are different instances, but each with the same values, state instance, and roles (but a distinct identity). A copy dependency from A to B means that B is an exact copy of A. Future changes in A are not necessarily reflected in B.

GeneralizableElement

A generalizable element is a model element that may participate in a generalization relationship.

In the metamodel a GeneralizableElement can be a generalization of other GeneralizableElements (i.e., all Features defined in and all ModelElements contained in the ancestors are also present in the GeneralizableElement). GeneralizableElement is an abstract metaclass.

Attributes

isAbstract	Specifies whether the GeneralizableElement is an incomplete declaration or not. True indicates that the GeneralizableElement is an incomplete declaration (abstract), false indicates that it is complete (concrete). An abstract GeneralizableElement is not instantiable since it does not contain all necessary information.
isLeaf	Specifies whether the GeneralizableElement is a GeneralizableElement with no descendents. True indicates that it may not have descendents, false indicates that it may have descendents (whether or not it actually has any descendents at the moment).
isRoot	Specifies whether the GeneralizableElement is a root GeneralizableElement with no ancestors. True indicates that it may not have ancestors, false indicates that it may have ancestors (whether or not it actually has any ancestors at the moment).

Associations

generalization	Designates a Generalization whose parent GeneralizableElement is the immediate ancestor of the current GeneralizableElement.

specialization	Designates a Generalization whose child GeneralizableElement is the immediate descendent of the current GeneralizableElement.

Generalization

A generalization is a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element (it has all of its properties, members, and relationships) and may contain additional information.

In the metamodel a Generalization is a directed inheritance relationship, uniting a GeneralizableElement with a more general GeneralizableElement in a hierarchy. Generalization is a subtyping relationship (i.e., an Instance of the more general GeneralizableElement may be substituted by an Instance of the more specific GeneralizableElement). See Inheritance for the consequences of Generalization relationships.

Attributes

discriminator	Designates the partition to which the Generalization link belongs. All of the Generalization links that share a given parent GeneralizableElement are divided into groups by their discriminator names. Each group of links sharing a discriminator name represents an orthogonal dimension of specialization of the parent GeneralizableElement. The discriminator need not be unique. The empty string is considered just another name. If all of the Generalization below a given GeneralizableElement have the same name (including the empty name), then it is a plain set of subelements. Otherwise the subelements form two or more groups, each of which must be represented by one of its members as an ancestor in a concrete descendent element.

Associations

child	Designates a GeneralizableElement that is the specialized version of the parent GeneralizableElement.
parent	Designates a GeneralizableElement that is the generalized version of the child GeneralizableElement.
powertype	Designates a Classifier that serves as a powertype for the child element along the dimension of generalization expressed by the Generalization. The child element is therefore an instance of the powertype element.

Stereotypes

«implementation»	Implementation is a stereotyped generalization, denoting that the client inherits the implementation of the supplier (its attributes, operations and methods) but does not make public the supplier's interfaces nor guarantee to support them, thereby violating substitutability. This is private inheritance.

Standard Constraints

complete	Complete is a constraint applied to a set of generalizations with the same discriminator and the same parent, specifying that all children have been specified (although some may be elided) and that additional children are not permitted with the same discriminator (in other words, the set of generalizations is closed).
disjoint	Disjoint is a constraint applied to a set of generalizations, specifying that instance may have no more than one of the given children as a type of the instance. This is the default semantics of generalization.
incomplete	Incomplete is a constraint applied to a set of generalizations with the same discriminator, specifying that not all children have been specified (even if some are elided) and that additional children are permitted with the same discriminator. This is the default semantics of generalizations.
overlapping	Overlapping is a constraint applied to a set of generalizations, specifying that instances may have more than one of the given children as a type of the instance.

Interface

An interface is a named set of operations that characterize the behavior of an element.

In the metamodel an Interface contains a set of Operations that together define a service offered by a Classifier realizing the Interface. A Classifier may offer several services, which means that it may realize several Interfaces, and several Classifiers may realize the same Interface.

Interfaces are GeneralizableElements.

Interfaces may not have Attributes, Associations, or Methods. An Interface may participate in an Association provided the Interface cannot see the Association; that is, a Classifier (other than an Interface) may have an Association to an Interface that is navigable from the Classifier but not from the Interface.

Method

A method is the implementation of an operation. It specifies the algorithm or procedure that effects the results of an operation.

In the metamodel a Method is a declaration of a named piece of behavior in a Classifier and realizes one or a set of Operations of the Classifier.

Attributes

body	The implementation of the Method as a ProcedureExpression.

Associations

specification	Designates an Operation that the Method implements. The Operation must be owned by the Classifier that owns the Method or be inherited by it. The signatures of the Operation and Method must match.

ModelElement

A model element is an element that is an abstraction drawn from the system being modeled. Contrast with view element, which is an element whose purpose is to provide a presentation of information for human comprehension.

In the metamodel a ModelElement is a named entity in a Model. It is the base for all modeling metaclasses in the UML. All other modeling metaclasses are either direct or indirect subclasses of ModelElement.

Each ModelElement can be regarded as a template. A template has a set of templateParameters that denotes which of the parts of a ModelElement are the template parameters. A ModelElement is a template when there is at least one template parameter. If it is not a template, a ModelElement cannot have template parameters. However, such embedded parameters are not usually complete and need not satisfy well-formedness rules. It is the arguments supplied when the template is instantiated that must be well-formed.

Partially instantiated templates are allowed. This is the case when there are arguments provided for some, but not all templateParameters. A partially instantiated template is still a template, since it still has parameters.

Attributes

name	An identifier for the ModelElement within its containing Namespace.

Associations

clientDependency	Inverse of client. Designates a set of Dependency in which the ModelElement is a client.
constraint	A set of Constraints affecting the element.
implementationLocation	The component that an implemented model element resides in.
namespace	Designates the Namespace that contains the ModelElement. Every ModelElement except a root element must belong to exactly one Namespace or else be a composite part of another ModelElement (which is a kind of virtual namespace). The pathname of Namespace or ModelElement names starting from the system provides a unique designation for every ModelElement. The association attribute visibility specifies the visibility of the element outside its namespace (see ElementOwnership).

presentation	A set of PresentationElements that present a view of the ModelElement.
supplierDependency	Inverse of supplier. Designates a set of Dependency in which the ModelElement is a supplier.
templateParameter	(association class TemplateParameter) A composite aggregation ordered list of parameters. Each parameter is a dummy ModelElement designated as a placeholder for a real ModelElement to be substituted during a binding of the template (see Binding). The real model element must be of the same kind (or a descendant kind) as the dummy ModelElement. The properties of the dummy ModelElement are ignored, except the name of the dummy element is used as the name of the template parameter. The association class TemplateParameter may be associated with a default ModelElement of the same kind as the dummy ModelElement. In the case of a Binding that does not supply an argument corresponding to the parameter, the value of the default ModelElement is used. If a Binding lacks an argument and there is no default ModelElement, the construct is invalid. Note that the template parameter element lacks structure. For example, a parameter that is a Class lacks Features; they are found in the actual argument.

Note that iff a ModelElement has at least one templateParameter, then it is a template, otherwise it is an ordinary element.

Namespace

A namespace is a part of a model that contains a set of ModelElements each of whose names designates a unique element within the namespace.

In the metamodel a Namespace is a ModelElement that can own other ModelElements, like Associations and Classifiers. The name of each owned ModelElement must be unique within the Namespace. Moreover, each contained ModelElement is owned by at most one Namespace. The concrete subclasses of Namespace have additional constraints on which kind of elements may be contained. Namespace is an abstract metaclass.

Associations

ownedElement	(association class ElementOwnership) A set of ModelElements owned by the Namespace. Its visibility attribute states whether the element is visible outside the namespace.

Node

A node is a run-time physical object that represents a computational resource, generally having at least a memory and often processing capability as well, and upon which components may be deployed.

In the metamodel a Node is a subclass of Classifier. It is associated with a set of Components residing on the Node.

Associations

resident	The set of Components residing on the Node.

Operation

An operation is a service that can be requested from an object to effect behavior. An operation has a signature, which describes the actual parameters that are possible (including possible return values).

In the metamodel an Operation is a BehavioralFeature that can be applied to the Instances of the Classifier that contains the Operation.

Attributes

concurrency	Specifies the semantics of concurrent calls to the same passive instance (i.e., an Instance originating from a Classifier with isActive=false). Active instances control access to their own Operations so this property is usually (although not required in UML) set to sequential. Possibilities include: • sequential - Callers must coordinate so that only one call to an Instance (on any sequential Operation) may be outstanding at once. If simultaneous calls occur, then the semantics and integrity of the system cannot be guaranteed. • guarded - Multiple calls from concurrent threads may occur simultaneously to one Instance (on any guarded Operation), but only one is allowed to commence. The others are blocked until the performance of the first Operation is complete. It is the responsibility of the system designer to ensure that deadlocks do not occur due to simultaneous blocks. Guarded Operations must perform correctly (or block themselves) in the case of a simultaneous sequential Operation or guarded semantics cannot be claimed. • concurrent - Multiple calls from concurrent threads may occur simultaneously to one Instance (on any concurrent Operation). All of them may proceed concurrently with correct semantics. Concurrent Operations must perform correctly in the case of a simultaneous sequential or guarded Operation or concurrent semantics cannot be claimed.

isAbstract	If true, then the operation does not have an implementation, and one must be supplied by a descendant. If false, the operation must have an implementation in the class or inherited from an ancestor.
isLeaf	If true, then the implementation of the operation may not be overriden by a descendant class. If false, then the implementation of the operation may be overridden by a descendant class (but it need not be overridden).
isRoot	If true, then the class must not inherit a declaration of the same operation. If false, then the class may (but need not) inherit a declaration of the same operation. (But the declaration must match in any case; a class may not modify an inherited operation declaration.)

Tagged Values

semantics	Semantics is the specification of the meaning of the operation.

Parameter

A parameter is an unbound variable that can be changed, passed, or returned. A parameter may include a name, type, and direction of communication. Parameters are used in the specification of operations, messages and events, templates, etc.

In the metamodel a Parameter is a declaration of an argument to be passed to, or returned from, an Operation, a Signal, etc.

Attributes

defaultValue	An Expression whose evaluation yields a value to be used when no argument is supplied for the Parameter.
kind	Specifies what kind of a Parameter is required. Possibilities are: • in - An input Parameter (may not be modified). • out - An output Parameter (may be modified to communicate information to the caller). • inout - An input Parameter that may be modified. • return -A return value of a call.
name	(Inherited from ModelElement) The name of the Parameter, which must be unique within its containing Parameter list.

Associations

type	Designates a Classifier to which an argument value must conform.

Permission

Permission is a kind of dependency. It grants a model element permission to access elements in another namespace.

In the metamodel Permission in a Dependency between a client ModelElement and a supplier ModelElement. The client receives permission to reference the supplier's contents. The supplier must be a Namespace.

The predefined stereotypes of Permission are access, import, and friend.

In the case of the access and import stereotypes, the client is granted permission to reference elements in the supplier namespace with public visibility. In the case of the import stereotype, the public names in the supplier namespace are added to the client namespace. An element may also access any protected contents of an ancestor namespace. An element may also access any contents (public, protected, or private) of its own namespace or a containing namespace.

In the case of the friend stereotype, the client is granted permission to reference elements in the supplier namespace, regardless of visibility.

Stereotypes

«access»	Access is a stereotyped permission dependency between two namespaces, denoting that the public contents of the target namespace are accessible to the namespace of the source package.
«friend»	Friend is a stereotyped permission dependency whose source is a model element, such as an operation, class, or package, and whose target is a model element in a different package, such as an operation, class or package. A friend relationship grants the source access to the target regardless of the declared visibility. It extends the visibility of the supplier so that the client can see into the supplier.
«import»	Import is a stereotyped permission dependency between two namespaces, denoting that the public contents of the target package are added to the namespace of the source package.

PresentationElement

A presentation element is a textual or graphical presentation of one or more model elements.

In the metamodel a PresentationElement is an Element which presents a set of ModelElements to a reader. It is the base for all metaclasses used for presentation. All other metaclasses with this purpose are either direct or indirect subclasses of PresentationElement. PresentationElement is an abstract metaclass. The subclasses of this class are proper to a graphic editor tool and are not specified here. It is a stub for their future definition.

Relationship

A relationship is a connection among model elements.

In the metamodel Relationship is a term of convenience without any specific semantics. It is abstract.

Children of Relationship are Association, Dependency, Flow, and Generalization.

StructuralFeature

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

In the metamodel a StructuralFeature declares a structural aspect of an Instance of a Classifier, such as an Attribute. For example, it specifies the multiplicity and changeability of the StructuralFeature. StructuralFeature is an abstract metaclass.

See Attribute for the descriptions of the attributes and associations, as it is the only subclass of StructuralFeature in the current metamodel.

TemplateParameter

Defines the relationship between a template (a ModelElement) and its parameter (a ModelElement). A ModelElement with at least one templateParameter association is a template (by definition).

In the metamodel, TemplateParameter refies the relationship between a ModelElement that is a template and a ModelElement that is a dummy placeholder for a template argment. See ModelElement on page 2-36, association templateParameter, for details.

Associations

defaultElement	An optional default value ModelElement. In case of a Binding of the template ModelElement in the reified TemplateParameter class association, the defaultElement is used as the argument of the bound element if no argument is supplied for the corresponding template parameter. If no argument is supplied and there is no default value, the model is ill formed.

Usage

A usage is a relationship in which one element requires another element (or set of elements) for its full implementation or operation. The relationship is not a mere historical artifact, but an ongoing need; therefore, two elements related by usage must be in the same model.

In the metamodel a Usage is a Dependency in which the client requires the presence of the supplier. How the client uses the supplier, such as a class calling an operation of another class, a method having an argument of another class, and a method from a class instantiating another class, is defined in the description of the particular Usage stereotype.

Various stereotypes of Usage are predefined, but the set is open-ended and may be added to.

Stereotypes

«call»	Call is a stereotyped usage dependency whose source is an operation and whose target is an operation. The relationship may also be subsumed to the class containing an operation, with the meaning that there exists an operation in the class to which the dependency applies. A call dependency specifies that the source operation or an operation in the source class invokes the target operation or an operation in the target class. A call dependency may connect a source operation to any target operation that is within scope including, but not limited to, operations of the enclosing classifier and operations of other visible classifiers.
«create»	Create is a stereotyped usage dependency denoting that the client classifier creates instances of the supplier classifier.
«instantiate»	A stereotyped usage dependency among classifiers indicating that operations on the client create instances of the supplier.
«send»	Send is a stereotyped usage dependency whose source is an operation and whose target is a signal, specifying that the source sends the target signal.

2.5.3 Well-Formedness Rules

The following well-formedness rules apply to the Core package.

Association

[1] The AssociationEnds must have a unique name within the Association.

```
self.allConnections->forAll( r1, r2 | r1.name = r2.name implies r1 = r2
)
```

[2] At most one AssociationEnd may be an aggregation or composition.

```
self.allConnections->select(aggregation <#none)->size <= 1
```

[3] If an Association has three or more AssociationEnds, then no AssociationEnd may be an aggregation or composition.

```
self.allConnections->size >=3 implies
        self.allConnections->forall(aggregation = #none)
```

[4] The connected Classifiers of the AssociationEnds should be included in the Namespace of the Association.

```
self.allConnections->forAll (r |
        self.namespace.allContents->includes (r.type) )
```

Additional operations

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

```
allConnections : Set(AssociationEnd);
```

```
allConnections = self.connection
```

AssociationClass

[1] The names of the AssociationEnds and the StructuralFeatures do not overlap.

```
self.allConnections->forAll( ar |
       self.allFeatures->forAll( f |
       f.oclIsKindOf(StructuralFeature) implies ar.name <f.name ))
```

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

```
self.allConnections->forAll(ar | ar.type <self)
```

Additional operations

[1] The operation allConnections results in the set of all AssociationEnds of the AssociationClass, including all connections defined by its parent (transitive closure).

```
allConnections : Set(AssociationEnd);
allConnections = self.connection->union(self.parent->select
       (s | s.oclIsKindOf(Association))->collect (a : Association |
           a.allConnections))->asSet
```

AssociationEnd

[1] The Classifier of an AssociationEnd cannot be an Interface or a DataType if the association is navigable away from that end.

```
(self.type.oclIsKindOf (Interface) or
self.type.oclIsKingOf (DataType)) implies
       self.association.connection->select
           (ae | ae <self)->forAll(ae | ae.isNavigable = #false)
```

[2] An Instance may not belong by composition to more than one composite Instance.

```
self.aggregation = #composite implies self.multiplicity.max <= 1
```

Attribute

No extra well-formedness rules.

BehavioralFeature

[1] All Parameters should have a unique name.

```
self.parameter->forAll(p1, p2 | p1.name = p2.name implies p1 = p2)
```

[2] The type of the Parameters should be included in the Namespace of the Classifier.

```
self.parameter->forAll( p |
        self.owner.namespace.allContents->includes (p.type) )
```

Additional operations

[1] The operation hasSameSignature checks if the argument has the same signature as the instance itself.

```
hasSameSignature ( b : BehavioralFeature ) : Boolean;
hasSameSignature (b) =
        (self.name = b.name) and
        (self.parameter->size = b.parameter->size) and
        Sequence{ 1..(self.parameter->size) }->forAll( index : Integer |
            b.parameter->at(index).type =
                self.parameter->at(index).type and
            b.parameter->at(index).kind =
                self.parameter->at(index).kind
        )
```

Binding

[1] The argument ModelElement must conform to the parameter ModelElement in a Binding. In an instantiation it must be of the same kind.

```
-- not described in OCL
```

Class

[1] If a Class is concrete, all the Operations of the Class should have a realizing Method in the full descriptor.

```
not self.isAbstract implies self.allOperations->forAll (op |
self.allMethods->exists (m | m.specification->includes(op)))
```

[2] A Class can only contain Classes, Associations, Generalizations, UseCases, Constraints, Dependencies, Collaborations, DataTypes, and Interfaces as a Namespace.

```
self.allContents->forAll->(c |
        c.oclIsKindOf(Class         ) or
        c.oclIsKindOf(Association   ) or
        c.oclIsKindOf(Generalization) or
        c.oclIsKindOf(UseCase       ) or
        c.oclIsKindOf(Constraint    ) or
        c.oclIsKindOf(Dependency    ) or
```

```
        c.oclIsKindOf(Collaboration ) or
        c.oclIsKindOf(DataType ) or
        c.oclIsKindOf(Interface     )
```

Classifier

[1] No BehavioralFeature of the same kind may have the same signature in a Classifier.

```
self.feature->forAll(f, g |
(
        (
            (f.oclIsKindOf(Operation) and g.oclIsKindOf(Operation)) or
            (f.oclIsKindOf(Method   ) and g.oclIsKindOf(Method   )) or
            (f.oclIsKindOf(Reception) and g.oclIsKindOf(Reception))
        ) and
        f.oclAsType(BehavioralFeature).hasSameSignature(g)
)
implies f = g)
```

[2] No Attributes may have the same name within a Classifier.

```
self.feature->select ( a | a.oclIsKindOf (Attribute) )->forAll ( p, q |
        p.name = q.name implies p = q )
```

[3] No opposite AssociationEnds may have the same name within a Classifier.

```
self.oppositeEnds->forAll ( p, q | p.name = q.name implies p = q )
```

[4] The name of an Attribute may not be the same as the name of an opposite AssociationEnd or a ModelElement contained in the Classifier.

```
self.feature->select ( a | a.oclIsKindOf (Attribute) )->forAll ( a |
        not self.allOppositeAssociationEnds->union (self.allContents)-
>collect ( q |
            q.name )->includes (a.name) )
```

[5] The name of an opposite AssociationEnd may not be the same as the name of an Attribute or a ModelElement contained in the Classifier.

```
self.oppositeAssociationEnds->forAll ( o |
        not self.allAttributes->union (self.allContents)->collect ( q |
            q.name )->includes (o.name) )
```

[6] For each Operation in an specification realized by the Classifier, the Classifier must have a matching Operation.

```
self.specification.allOperations->forAll (interOp |
```

```
        self.allOperations->exists( op | op.hasMatchingSignature
(interOp) ) )
```

[7] All of the generalizations in the range of a powertype have the same discriminator.

```
self.powertypeRange->forAll
        (g1, g2 | g1.discriminator = g2.discriminator)
```

Additional operations

[1] The operation allFeatures results in a Set containing all Features of the Classifier itself and all its inherited Features.

```
allFeatures : Set(Feature);

allFeatures = self.feature->union(

self.parent.oclAsType(Classifier).allFeatures)
```

[2] The operation allOperations results in a Set containing all Operations of the Classifier itself and all its inherited Operations.

```
allOperations : Set(Operation);

allOperations = self.allFeatures->select(f | f.oclIsKindOf(Operation))
```

[3] The operation allMethods results in a Set containing all Methods of the Classifier itself and all its inherited Methods.

```
allMethods : set(Method);

allMethods = self.allFeatures->select(f | f.oclIsKindOf(Method))
```

[4] The operation allAttributes results in a Set containing all Attributes of the Classifier itself and all its inherited Attributes.

```
allAttributes : set(Attribute);

allAttributes = self.allFeatures->select(f | f.oclIsKindOf(Attribute))
```

[5] The operation associations results in a Set containing all Associations of the Classifier itself.

```
associations : set(Association);

associations = self.associationEnd.association->asSet
```

[6] The operation allAssociations results in a Set containing all Associations of the Classifier itself and all its inherited Associations.

```
allAssociations : set(Association);

allAssociations = self.associations->union (

self.parent.oclAsType(Classifier).allAssociations)
```

[7] The operation oppositeAssociationEnds results in a set of all AssociationEnds that are opposite to the Classifier.

```
oppositeAssociationEnds : Set (AssociationEnd);

oppositeAssociationEnds =
```

```
        self.association->select ( a | a.associationEnd->select ( ae |
            ae.type = self ).size = 1 )->collect ( a |
                a.associationEnd->select ( ae | ae.type <self ) )->union
(
        self.association->select ( a | a.associationEnd->select ( ae |
            ae.type = self ).size 1 )->collect ( a |
                a.associationEnd) )
```

[8] The operation allOppositeAssociationEnds results in a set of all AssociationEnds, including the inherited ones, that are opposite to the Classifier.

```
allOppositeAssociationEnds : Set (AssociationEnd);
allOppositeAssociationEnds = self.oppositeAssociationEnds->union (
            self.parent.allOppositeAssociationEnds )
```

[9] The operation specification yields the set of Classifiers that the current Classifier realizes.

```
specification: Set(Classifier)
specification = self.clientDependency->
        select(d |
            d.oclIsKindOf(Abstraction)
            and d.stereotype.name = "realization"
            and d.supplier.oclIsKindOf(Classifier))
        .supplier.oclAsType(Classifier)
```

[10] The operation allContents returns a Set containing all ModelElements contained in the Classifier together with the contents inherited from its parents.

```
allContents : Set(ModelElement);
allContents = self.contents->union(
        self.parent.allContents->select(e |
            e.elementOwnership.visibility = #public or
            e.elementOwnership.visibility = #protected))
```

Comment

No extra well-formedness rules.

Component

[1] A Component may only contain other Components.

```
self.allContents-forAll( c | c.oclIsKindOf(Component))
```

[2] A Component may only implement DataTypes, Interfaces, Classes, Associations, Dependencies, Constraints, Signals, DataValues and Objects.

```
self.allResidentElements -forAll( re |
```

```
        re.oclIsKindOf(DataType) or
        re.oclIsKindOf(Interface) or
        re.oclIsKindOf(Class) or
        re.oclIsKindOf(Association) or
        re.oclIsKindOf(Dependency) or
        re.oclIsKindOf(Constraint) or
        re.oclIsKindOf(Signal) or
        re.oclIsKindOf(DataValue) or
        re.oclIsKindOf(Object) )
```

Additional operations

[1] The operation allResidentElements results in a Set containing all ModelElements resident in a Component or one of its ancestors.

```
allResidentElements : set(ModelElement)
        allResidentElements = self.resident->union(
        self.parent.oclAsType(Component).allResidentElements->select(
re |
        re.elementResidence.visibility = #public or
re.elementResidence.visibility = #protected))
```

[2] The operation allVisibleElements results in a Set containing all ModelElements visible outside the Component.

```
allVisibleElements : Set(ModelElement)
allVisibleElements = self.allContents -select( e |
        e.elementOwnership.visibility = #public) -union (
            self.allResidentElements -select ( re |
            re.elementResidence.visibility = #public)))
```

Constraint

[1] A Constraint cannot be applied to itself.

```
not self.constrainedElement->includes (self)
```

DataType

[1] A DataType can only contain Operations, which all must be queries.

```
self.allFeatures->forAll(f |
                        f.oclIsKindOf(Operation) and
f.oclAsType(Operation).isQuery)
```

[2] A DataType cannot contain any other ModelElements.

```
self.allContents->isEmpty
```

Dependency

No extra well-formedness rules.

Element

No extra well-formedness rules.

ElementOwnership

No additional well-formedness rules.

ElementResidence

No additional well-formedness rules.

Feature

No extra well-formedness rules.

GeneralizableElement

[1] A root cannot have any Generalizations.

```
self.isRoot implies self.generalization->isEmpty
```

[2] No GeneralizableElement can have a parent Generalization to an element which is a leaf.

```
self.parent->forAll(s | not s.isLeaf)
```

[3] Circular inheritance is not allowed.

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

[4] The parent must be included in the Namespace of the GeneralizableElement.

```
self.generalization->forAll(g |
        self.namespace.allContents->includes(g.parent) )
```

Additional Operations

[1] The operation parent returns a Set containing all direct parents.

```
parent : Set(GeneralizableElement);
```

```
parent = self.generalization.parent
```

[2] The operation allParents returns a Set containing all the Generalizable Elements inherited by this GeneralizableElement (the transitive closure), excluding the GeneralizableElement itself.

```
allParents : Set(GeneralizableElement);
allParents = self.parent->union(self.parent.allParents)
```

Generalization

[1] A GeneralizableElement may only be a child of GeneralizableElement of the same kind.

```
self.child.oclType = self.parent.oclType
```

ImplementationClass (stereotype of Class)

[1] All direct instances of an implementation class must not have any other Classifiers that are implementation classes.

```
self.instance.forall(i | i.classifier.forall(c |
        c.stereotype.name = "implementationClass" implies c = self))
```

[2] A parent of an implementation class must be an implementation class.

```
self.parent->forAll(stereotype.name="implementationClass")
```

Interface

[1] An Interface can only contain Operations.

```
self.allFeatures->forAll(f | f.oclIsKindOf(Operation))
```

[2] An Interface cannot contain any ModelElements.

```
self.allContents->isEmpty
```

[3] All Features defined in an Interface are public.

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

Method

[1] If the realized Operation is a query, then so is the Method.

```
self.specification->isQuery implies self.isQuery
```

[2] The signature of the Method should be the same as the signature of the realized Operation.

```
self.hasSameSignature (self. specification)
```

[3] The visibility of the Method should be the same as for the realized Operation.

```
self.visibility = self.specification.visibility
```

[4] The realized Operation must be a feature (possibly inherited) of the same Classifier as the Method.

```
self.owner.allOperations->includes(self.specification)
```

[5] If the realized Operation has been overridden one or more times in the ancestors of the owner of the Method, then the Method must realize the latest overriding (that is, all other Operations with the same signature must be owned by ancestors of the owner of the realized Operation).

```
self.specification.owner.allOperations->includesAll(
       (self.owner.allOperations->select(op |
           self.hasSameSignature(op)))
```

ModelElement

That part of the model owned by a template is not subject to all well-formedness rules. A template is not directly usable in a well-formed model. The results of binding a template are subject to well-formedness rules.

```
(not expressed in OCL)
```

Additional operations

[1] The operation supplier results in a Set containing all direct suppliers of the ModelElement.

```
supplier : Set(ModelElement);
```

```
supplier = self.clientDependency.supplier
```

[2] The operation allSuppliers results in a Set containing all the ModelElements that are suppliers of this ModelElement, including the suppliers of these Model Elements. This is the transitive closure.

```
allSuppliers : Set(ModelElement);
```

```
allSuppliers = self.supplier->union(self.supplier.allSuppliers)
```

[3] The operation “model” results in the set of Models to which a ModelElement belongs.

```
model : Set(Model);
```

```
model = self.namespace->union(self.namespace.allSurroundingNamespaces)
              ->select( ns|
                     ns.oclIsKindOf (Model))
```

[4] A ModelElement is a template when it has parameters.

```
isTemplate : Boolean;
```

```
isTemplate = (self.templateParameter->notEmpty)
```

[5] A ModelElement is an instantiated template when it is related to a template by a Binding relationship.

```
isInstantiated : Boolean;
```

```
isInstantiated = self.clientDependency->select(
       oclIsKindOf(Binding))->notEmpty
```

[6] The templateArguments are the arguments of an instantiated template, which substitute for template parameters.

```
templateArguments : Set(ModelElement);
templateArguments = self.clientDependency->

select(oclIsKindOf(Binding)).oclAsType(Binding).argument
```

Namespace

[1] If a contained element, which is not an Association or Generalization has a name, then the name must be unique in the Namespace.

```
self.allContents->forAll(me1, me2 : ModelElement |
       (   not me1.oclIsKindOf (Association) and not me2.oclIsKindOf
(Association) and
           me1.name <'' and me2.name <'' and me1.name = me2.name
       ) implies
           me1 = me2 )
```

[2] All Associations must have a unique combination of name and associated Classifiers in the Namespace.

```
self.allContents -> select(oclIsKindOf(Association))->
       forAll(a1, a2 |
           a1.name = a2.name and
           a1.connection.type = a2.connection.type
           implies a1 = a2)
```

Additional operations

[1] The operation contents results in a Set containing all ModelElements contained by the Namespace.

```
contents : Set(ModelElement)
contents = self.ownedElement
```

[2] The operation allContents results in a Set containing all ModelElements contained by the Namespace.

```
allContents : Set(ModelElement);
allContents = self.contents
```

[3] The operation allVisibleElements results in a Set containing all ModelElements visible outside of the Namespace.

```
allVisibleElements : Set(ModelElement)
allVisibleElements = self.allContents->select(e |
       e.elementOwnership.visibility = #public)
```

[4] The operation allSurroundingNamespaces results in a Set containing all surrounding Namespaces.

```
allSurroundingNamespaces : Set(Namespace)
```

```
allSurroundingNamespaces =
self.namespace->union(self.namespace.allSurroundingNamespaces)
```

Node

No extra well-formedness rules.

Operation

No additional well-formedness rules.

Parameter

No additional well-formedness rules.

PresentationElement

No extra well-formedness rules.

StructuralFeature

[1] The connected type should be included in the owner's Namespace.

```
self.owner.namespace.allContents->includes(self.type)
```

[2] The type of a StructuralFeature must be a Class, DataType or Interface.

```
self.type.oclIsKindOf(Class) or
self.type.oclIsKindOf(DataType) or
self.type.oclIsKindOf(Interface)
```

Trace

A trace is an Abstraction with the «trace» stereotype. These are the additional constraints due to the stereotype.

[1] The client ModelElements of a Trace must all be from the same Model.

```
self.client->forAll(e1, e2 | e1.model = e2.model)
```

[2] The supplier ModelElements of a Trace must all be from the same Model.

```
self.supplier->forAll(e1, e2 | e1.model = e2.model)
```

[3] The client and supplier ModelElements must be from two different Models.

```
self.client.model <self.supplier.model
```

[4] The client and supplier ModelElements must all be from models of the same system.

```
self.client.model.intersection(self.supplier.model) <Set{}
```

Type (stereotype of Class)

[1] A Type may not have any Methods.

```
not self.feature->exists(oclIsKindOf(Method))
```

[2] The parent of a type must be a type.

```
self.parent->forAll(stereotype.name = "type")
```

Usage

No extra well-formedness rules.

2.5.4 Semantics

This section provides a description of the dynamic semantics of the elements in the Core. It is structured based on the major constructs in the core, such as interface, class, and association.

Association

Figure 2-10 Association Illustration

An association declares a connection (link) between instances of the associated classifiers (e.g., classes). It consists of at least two association-ends, each specifying a connected classifier and a set of properties which must be fulfilled for the relationship to be valid. The multiplicity property of an association-end specifies how many instances of the classifier at a given end (the one bearing the multiplicity value) may be associated with a single instance of the classifier at the other end. A multiplicity is a range of nonnegative integers. The association-end also states whether or not the connection may be traversed towards the instance playing that role in the connection (isNavigable), for instance, if the instance is directly reachable via the association. An association-end also specifies whether or not an instance playing that role in a connection may be replaced by another instance. It may state

- that no constraints exist (none),
- that the link cannot be modified once it has been initialized (frozen), or
- that new links of the association may be added but not removed or altered (addOnly).

These constraints do not affect the modifiability of the objects themselves that are attached to the links. Moreover, the targetScope specifies if the association-end should be connected to an instance of (a child of) the classifier, or (a child of) the classifier itself. The isOrdered attribute of association-end states that if the instances related to a single instance at the other end have an ordering that must be preserved, the order of insertion of new links must be specified by

operations that add or modify links. Note that sorting is a performance optimization and is not an example of a logically ordered association, because the ordering information in a sort does not add any information.

In UML, Associations can be of three different kinds: 1) ordinary association, 2) composite aggregate, and 3) shared aggregate. Since the aggregate construct can have several different meanings depending on the application area, UML gives a more precise meaning to two of these constructs (i.e., association and composite aggregate) and leaves the shared aggregate more loosely defined in between.

An association may represent an aggregation (i.e., a whole/part relationship). In this case, the association-end attached to the whole element is designated, and the other association-end of the association represents the parts of the aggregation. Only binary associations may be aggregations. Composite aggregation is a strong form of aggregation which requires that a part instance be included in at most one composite at a time and that the composite object has sole responsibility for the disposition of its parts. This means that the composite object is responsible for the creation and destruction of the parts. In implementation terms, it is responsible for their memory allocation. If a composite object is destroyed, it must destroy all of its parts. It may remove a part and give it to another composite object, which then assumes responsibility for it. If the multiplicity from a part to composite is zero-to-one, the composite may remove the part and the part may assume responsibility for itself, otherwise it may not live apart from a composite.

A consequence of these rules is that a composite implies propagation semantics (i.e., some of the dynamic semantics of the whole is propagated to its parts). For example, if the whole is copied or deleted, then so are the parts as well (because a part may belong to at most one composite).

A shared aggregation denotes weak ownership (i.e., the part may be included in several aggregates) and its owner may also change over time. However, the semantics of a shared aggregation does not imply deletion of the parts when one of its containers is deleted. Both kinds of aggregations define a transitive, antisymmetric relationship (i.e., the instances form a directed, non-cyclic graph). Composition instances form a strict tree (or rather a forest).

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 that 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 also that 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.

AssociationClass

Figure 2-11 AssociationClass Illustration

An association may be refined to have its own set of features (i.e., 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 such an association is a combination of the semantics of an ordinary association and of a class.

The AssociationClass construct can be expressed in a few different ways in the metamodel (e.g., as a subclass of Class, as a subclass of Association, or as a subclass of Classifier). Since an AssociationClass is a construct being both an association (having a set of association-ends) and a class (declaring a set of features), the most accurate way of expressing it is as a subclass of both Association and Class. In this way, AssociationClass will have all the properties of the other two constructs. Moreover, if new kinds of associations containing features (e.g., AssociationDataType) are to be included in UML, these are easily added as subclasses of Association and the other Classifier.

The terms child, subtype, and subclass are synonyms and mean that an instance of a classifier being a subtype of another classifier can always be used where an instance of the latter classifier is expected. The neutral terms *parent* and *child*, with the transitive closures *ancestor* and *descendant*, are the preferred terms in this document.

Class

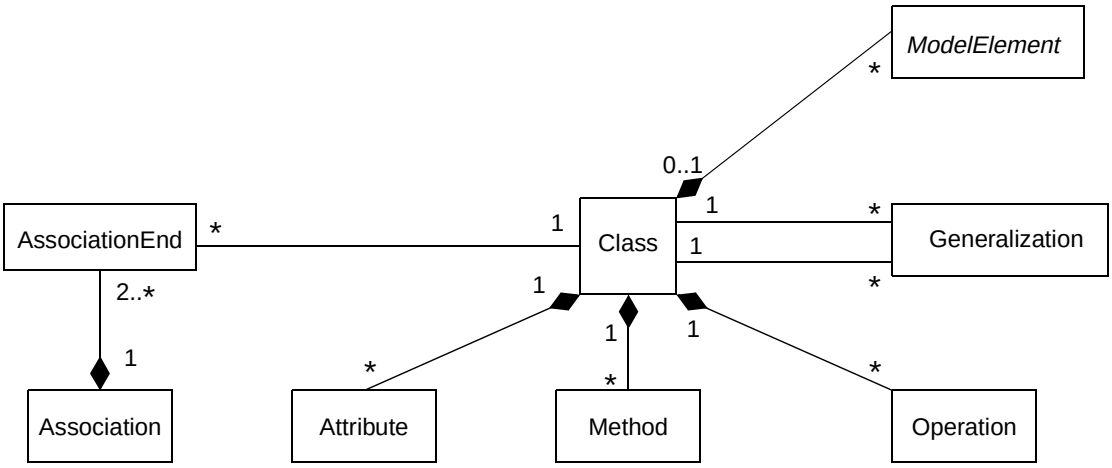


Figure 2-12 Class Illustration

The purpose of a class is to declare a collection of methods, operations, and attributes that fully describe the structure and behavior of objects. All objects instantiated from a class will have attribute values matching the attributes of the full class descriptor and support the operations found in the full class descriptor. Some classes may not be directly instantiated. These classes are said to be abstract and exist only for other classes to inherit and reuse the features declared by them. No object may be a direct instance of an abstract class, although an object may be an indirect instance of one through a subclass that is non-abstract.

When a class is instantiated to create a new object, a new instance is created, which is initialized containing an attribute value for each attribute found in the full class descriptor. The object is also initialized with a connection to the list of methods in the full class descriptor.

> Note – An actual implementation behaves as if there were a full class descriptor, but many clever optimizations are possible in practice.

Finally, the identity of the new object is returned to the creator. The identity of every instance in a well-formed system is unique and automatic.

A class can have generalizations to other classes. This means that the full class descriptor of a class is derived by inheritance from its own segment declaration and those of its ancestors. Generalization between classes implies substitutability (i.e., an instance of a class may be used whenever an instance of a superclass is expected). If the class is specified as a root, it cannot be a subclass of other classes. Similarly, if it is specified as a leaf, no other class can be a subclass of the class.

Each attribute declared in a class has a visibility and a type. The visibility defines if the attribute is publicly available to any class, if it is only available inside the class and its subclasses (protected), or if it can only be used inside the class (private). The targetScope of the attribute declares whether its value should be an instance (of a child) of that type or if it should be (a child of) the type itself. There are two alternatives for the ownerScope of an attribute:

- it may state that each object created by the class (or by its subclasses) has its own value of the attribute, or
- that the value is owned by the class itself.

An attribute also declares how many attribute values should be connected to each owner (multiplicity), what the initial values should be, and if these attribute values may be changed to:

- none - no constraints exists,
- frozen - the value cannot be replaced or added to once it has been initialized, or
- addOnly - new values may be added to a set but not removed or altered.

For each operation, the operation name, the types of the parameters, and the return type(s) are specified, as well as its visibility (see above). An operation may also include a specification of the effects of its invocation. The specification can be done in several different ways (e.g., with pre- and post-conditions, pseudo-code, or just plain text). Each operation declares if it is applicable to the instances, the class, or to the class itself (ownerScope). Furthermore, the operation states whether or not its application will modify the state of the object (isQuery). The operation also states whether or not the operation may be realized by a different method in a subclass (isPolymorphic). A method realizing an operation has the same signature as the operation and a body implementing the specification of the operation. Methods in descendents override and replace methods inherited from ancestors (see “Inheritance” on page 2-59). Each method implements an operation declared in the class or inherited from an ancestor. The same operation may be declared more than once in a full class descriptor, but their descriptions must all match, except that the generalization properties (isRoot, IsAbstract, isLeaf) may vary, and a child operation may strengthen query properties (the child may be a query even though the parent is not). The specification of the method must match the specification of its matching operation, as defined above for operations. Furthermore, if the isQuery attribute of an operation is true, then it must also be true in any realizing method. However, if it is false in the operation, it may still be true in the method if the method does not actually modify the state to carry out the behavior required by the operation (this can only be true if the operation does not inherently modify state). The concept of visibility is not relevant for methods.

Classes may have associations to each other. This implies that objects created by the associated classes are semantically connected (i.e., that links exist between the objects, according to the requirements of the associations). See *Association* on the next page. Associations are inherited by subclasses.

A class may realize a set of interfaces. This means that each operation found in the full descriptor for any realized interface must be present in the full class descriptor with the same specification (see Semantics section Inheritance on page 2-59). The relationship between interface and class is not necessarily one-to-one; a class may offer several interfaces and one interface may be offered by more than one class. The same operation may be defined in

multiple interfaces that a class supports; if their specifications are identical then there is no conflict; otherwise, the model is ill-formed. Moreover, a class may contain additional operations besides those found in its interfaces.

A class acts as the namespace for various kinds of contained elements defined within its scope, including classes, interfaces and associations (note that this is purely a scoping construction and does not imply anything about aggregation), the contained classifiers can be used as ordinary classifiers in the container class. If a class inherits another class, the contents of the ancestor are available to its descendents if the visibility of an element is public or protected; however, if the visibility is private, then the element is not visible and therefore not available in the descendant.

Inheritance

To understand inheritance it is first necessary to understand the concept of a full descriptor and a segment descriptor. A full descriptor is the full description needed to describe an object or other instance (see “Instantiation” on page 2-60). It contains a description of all of the attributes, associations, and operations that the object contains. In a pre-object-oriented language, the full descriptor of a data structure was declared directly in its entirety. In an object-oriented language, the description of an object is built out of incremental segments that are combined using inheritance to produce a full descriptor for an object. The segments are the modeling elements that are actually declared in a model. They include elements such as class and other generalizable elements. Each generalizable element contains a list of features and other relationships that it adds to what it inherits from its ancestors. The mechanism of inheritance defines how full descriptors are produced from a set of segments connected by generalization. The full descriptors are implicit, but they define the structure of actual instances.

Each kind of generalizable element has a set of inheritable features. For any model element, these include constraints. For classifiers, these include features (attributes, operations, signal receptions, and methods) and participation in associations. The ancestors of a generalizable element are its parents (if any) together with all of their ancestors (with duplicates removed). For a Namespace (such as a Package or a Class with nested declarations), the public or protected contents of the Namespace are available to descendants of the Namespace.

If a generalizable element has no parent, then its full descriptor is the same as its segment descriptor. If a generalizable element has one or more parents, then its full descriptor contains the union of the features from its own segment descriptor and the segment descriptors of all of its ancestors. For a classifier, no attribute, operation, or signal with the same signature may be declared in more than one of the segments (in other words, they may not be redefined). A method may be declared in more than one segment. A method declared in any segment supersedes and replaces a method with the same signature declared in any ancestor. If two or more methods nevertheless remain, then they conflict and the model is ill-formed. The constraints on the full descriptor are the union of the constraints on the segment itself and all of its ancestors. If any of them are inconsistent, then the model is ill-formed.

In any full descriptor for a classifier, each method must have a corresponding operation. In a concrete classifier, each operation in its full descriptor must have a corresponding method in the full descriptor.

The purpose of the full descriptor is explained under “Instantiation” on page 2-60.

Instantiation

The purpose of a model is to describe the possible states of a system and their behavior. The state of a system comprises objects, values, and links. Each object is described by a full class descriptor. The class corresponding to this descriptor is the direct class of the object. If an object is not completely described by a single class (multiple classification), then any class in the minimal set of unrelated (by generalization) classes whose union completely describes the object is a direct class of the object. Similarly each link has a direct association and each value has a direct data type. Each of these instances is said to be a direct instance of the classifier from which its full descriptor was derived. An instance is an indirect instance of the classifier or any of its ancestors.

The data content of an object comprises one value for each attribute in its full class descriptor (and nothing more). The value must be consistent with the type of the attribute. The data content of a link comprises a tuple containing a list of instances, one that is an indirect instance of each participant classifier in the full association descriptor. The instances and links must obey any constraints on the full descriptors of which they are instances (including both explicit constraints and built-in constraints such as multiplicity).

The state of a system is a valid system instance if every instance in it is a direct instance of some element in the system model and if all of the constraints imposed by the model are satisfied by the instances.

The behavioral parts of UML describe the valid sequences of valid system instances that may occur as a result of both external and internal behavioral effects.

Interface

Figure 2-13 Interface Illustration

The purpose of an interface is to collect a set of operations that constitute a coherent service offered by classifiers. Interfaces provide a way to partition and characterize groups of operations. An interface is only a collection of operations with a name. It cannot be directly instantiated. Instantiable classifiers, such as class or use case, may use interfaces for specifying different services offered by their instances. Several classifiers may realize the same interface. All of them must contain at least the operations matching those contained in the interface. The specification of an operation contains the signature of the operation (i.e., its name, the types of the parameters and the return type). An interface does not imply any internal structure of the realizing classifier. For example, it does not define which algorithm to use for realizing an operation. An operation may, however, include a specification of the effects of its invocation. The specification can be done in several different ways (e.g., with pre and post-conditions, pseudo-code, or just plain text).

Each operation declares if it applies to the instances of the classifier declaring it or to the classifier itself (e.g., a constructor on a class (ownerScope)). Furthermore, the operation states whether or not its application will modify the state of the instance (isQuery). The operation also states whether or not all the classes must have the same realization of the operation (isPolymorphic).

An interface can be a child of other interfaces denoted by generalizations. This means that a classifier offering the interface must provide not only the operations declared in the interface but also those declared in the ancestors of the interface. If the interface is specified as a root, it cannot be a child of other interfaces. Similarly, if it is specified as a leaf, no other interface can be a child of the interface.

Operation

Operation is a conceptual construct, while Method is the implementation construct. Their common features, such as having a signature, are expressed in the BehavioralFeature metaclass, and the specific semantics of the Operation. The Method constructs are defined in the corresponding subclasses of BehavioralFeature.

PresentationElement

The responsibility of presentation element is to provide a textual and graphical projection of a collection of model elements. In this context, projection means that the presentation element represents a human readable notation for the corresponding model elements. The notation for UML can be found in Chapter 3 of this document.

Presentation elements and model elements must be kept in agreement, but the mechanisms for doing this are design issues for model editing tools.

Template

A template is a parameterized model element that cannot be used directly in a model. Instead, it may be used to generate other model elements using the Binding relationship; those generated model elements can be used in normal relationships with other elements.

A template rcpresents the parameterization of a model element, such as a class or an operation, although conceptutally any model element may be used (but not all may be useful). The template element is attached by composite aggregation to an ordered list of parameter elements. Each parameter element has a name that represents an parameter name within the template element. Any use of the name within the scope of the template element represents an unbound parameter that is to be replaced by an actual value in a Binding of the template. For example, a parameter may represent the type of an attribute of a class (for a class template). The corresponding attribute would have an association to the template parameter as its type.

Note that the scope of the template includes all of the elements recursively owned by it by composition aggregation. For example, a parameterized class template owns its attributes, operations, and so on. Neither the parameterized elements nor its contents may be used directly in a model without binding.

A template element has the templateParameter association to a list of ModelElements that serve as its parameters. To avoid introducing metamodel (M2) elements in an ordinary (M1) model, the model contains a representative of each parameter element, rather than the type of the parameter element. For example, a frequent kind of parameter is a class. Instead of including the metaclass Class in the (M1) ordinary model, a dummy class must be declared whose name is the name of the parameter. This dummy element is meaningful only within the template (it may not be used within the wider model) and it has no features (such as attributes and operations), because the features are part of an actual element that is supplied when the template is bound. Because a template parameter is only a dummy that lacks internal structure, it may violate well-formedness constraints of elements of its kind; the actual elements supplied during binding must satisfy ordinary well-formedness constraints.

Note also that when the template is bound, the bound element does not show the explicit structure of a element of its kind; it is a stub. Its semantics and well-formedness rules must be evaluated as if the actual substitutions of actual elements for parameters had been made; but the expansions are not explicitly shown in a canonical model as they are regarded as derived.

A template element is therefore effectively isolated from the directly-usale part of the model and is indirectly connected to its ultimate instances through Binding associations to bound elements. The bound elements may be used in ordinary models in places where the model element underlying the template could be used.

Miscellaneous

Figure 2-14 Miscellaneous Illustration

A constraint is a Boolean expression over one or several elements which must always be true. A constraint can be specified in several different ways (e.g., using natural language or a constraint language).

A dependency specifies that the semantics of a set of model elements requires the presence of another set of model elements. This implies that if the source is somehow modified, the dependents probably must be modified. The reason for the dependency can be specified in several different ways (e.g., using natural language or an algorithm) but is often implicit.

A Usage or Binding dependency can be established only between elements in the same model, since the semantics of a model cannot be dependent on the semantics of another model. If a connection is to be established between elements in different models, a Trace or Refinement should be used. Refinement can connect elements in different or same models.

Whenever the supplier element of a dependency changes, the client element is potentially invalidated. After such invalidation, a check should be performed followed by possible changes to the derived client element. Such a check should be performed after which action can be taken to change the derived element to validate it again. The semantics of this validation and change is outside the scope of UML.

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

2.6 Extension Mechanisms

2.6.1 Overview

The Extension Mechanisms package is the subpackage that specifies how model elements are customized and extended with new semantics. It defines the semantics for stereotypes, constraints, and tagged values.

The UML provides a rich set of modeling concepts and notations that have been carefully designed to meet the needs of typical software modeling projects. However, users may sometimes require additional features and/or notations beyond those defined in the UML standard. In addition, users often need to attach non-semantic information to models. These needs are met in UML by three built-in extension mechanisms that enable new kinds of modeling elements to be added to the modeler's repertoire as well as to attach free-form information to modeling elements. These three extension mechanisms can be used separately or together to define new modeling elements that can have distinct semantics, characteristics, and notation relative to the built in UML modeling elements specified by the UML metamodel. Concrete constructs defined in Extension Mechanisms include Constraint, Stereotype, and TaggedValue.

The UML extension mechanisms are intended for several purposes:

- To add new modeling elements for use in creating UML models.
- To define standard items that are not considered interesting or complex enough to be defined directly as UML metamodel elements.
- To define process-specific or implementation language-specific extensions.
- To attach arbitrary semantic and non-semantic information to model elements.

Although it is beyond the scope and intent of this document, it is also possible to extend the UML metamodel by explicitly adding new metaclasses and other meta constructs. This capability depends on unique features of certain UML-compatible modeling tools, or direct use of a meta-metamodel facility, such as the CORBA Meta Object Facility (MOF).

The most important of the built-in extension mechanisms is based on the concept of Stereotype. Stereotypes provide a way of classifying model elements at the object model level and facilitate the addition of "virtual" UML metaclasses with new metaattributes and semantics. The other built in extension mechanisms are based on the notion of property lists consisting of tags and values, and constraints. These allow users to attach additional properties and semantics directly to individual model elements, as well as to model elements classified by a Stereotype.

A stereotype is a UML model element that is used to classify (or mark) other UML elements so that they behave in some respects as if they were instances of new "virtual" or "pseudo" metamodel classes whose form is based on existing "base" classes. Stereotypes augment the classification mechanism based on the built in UML metamodel class hierarchy; therefore, names of new stereotypes must not clash with the names of predefined metamodel elements or other stereotypes. Any model element can be marked by at most one stereotype, but any stereotype can be constructed as a specialization of numerous other stereotypes.

A stereotype may introduce additional values, additional constraints, and a new graphical representation. All model elements that are classified by a particular stereotype ("stereotyped") receive these values, constraints, and representation. By allowing stereotypes to have associated graphical representations users can introduce new ways of graphically distinguishing model elements classified by a particular stereotype.

A stereotype shares the attributes, associations, and operations of its base class but it may have additional well-formedness constraints as well as a different meaning and attached values. The intent is that a tool or repository be able to manipulate a stereotyped element the same as the ordinary element for most editing and storage purposes, while differentiating it for certain semantic operations, such as well-formedness checking, code generation, or report writing.

Any modeling element may have arbitrary attached information in the form of a property list consisting of tag-value pairs. A tag is a name string that is unique for a given element that selects an associated arbitrary value. Values may be arbitrary but for uniform information exchange they should be represented as strings. The tag represents the name of an arbitrary property with the given value. Tags may be used to represent management information (author, due date, status), code generation information (optimizationLevel, containerClass), or additional semantic information required by a given stereotype.

It is possible to specify a list of tags (with default values, if desired) that are required by a particular stereotype. Such required tags serve as "pseudoattributes" of the stereotype to supplement the real attributes supplied by the base element class. The values permitted to such tags can also be constrained.

It is not necessary to stereotype a model element in order to give it individually distinct constraints or tagged values. Constraints can be directly attached to a model element (stereotyped or not) to change its semantics. Likewise, a property list consisting of tag-value pairs can be directly attached to any model element. The tagged values of a property list allow characteristics to be assigned to model elements on a flexible, individual basis. Tags are user-definable, certain ones are predefined and are listed in the Standard Elements appendix.

Constraints or tagged values associated with a particular stereotype are used to extend the semantics of model elements classified by that stereotype. The constraints must be observed by all model elements marked with that stereotype.

The following sections describe the abstract syntax, well-formedness rules, and semantics of the Extension Mechanisms package.

2.6.2 Abstract Syntax

The abstract syntax for the Extension Mechanisms package is expressed in graphic notation in Figure 2-15 on page 2-67.

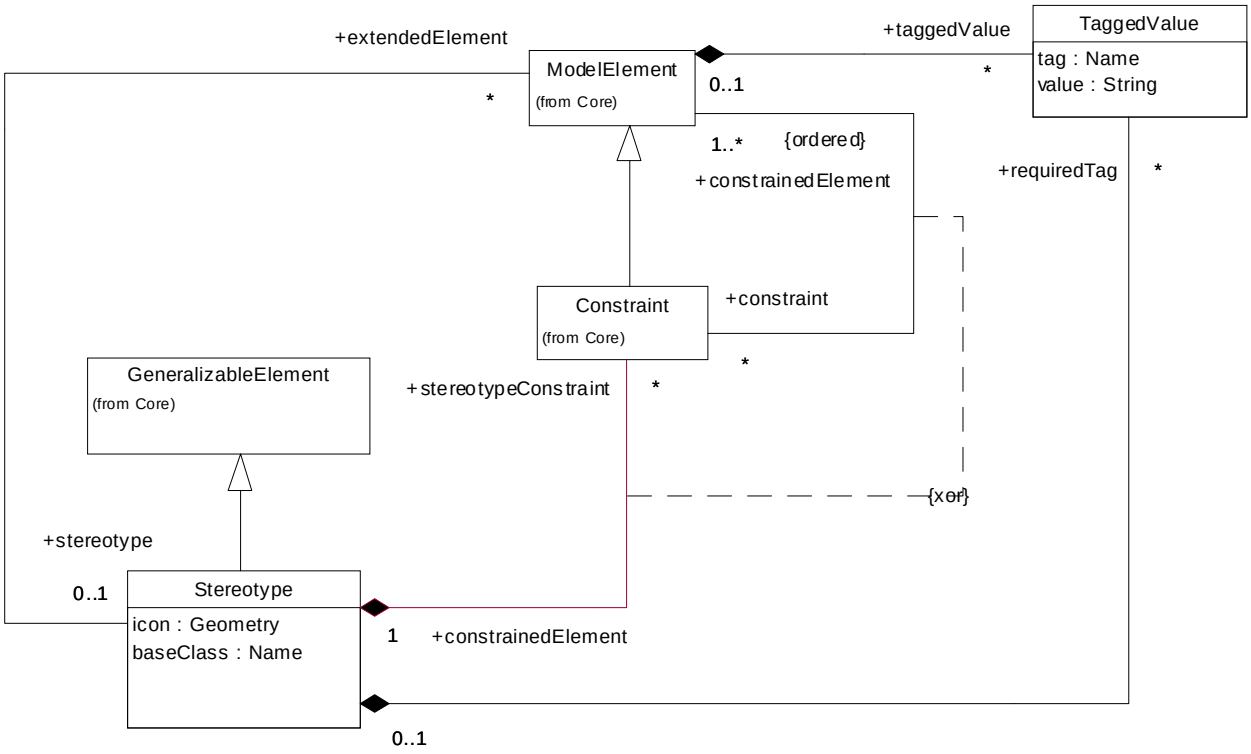


Figure 2-15 Extension Mechanisms

Constraint

The constraint concept allows new semantics to be specified linguistically for a model element. The specification is written as an expression in a designated constraint language. The language can be specially designed for writing constraints (such as OCL), a programming language, mathematical notation, or natural language. If constraints are to be enforced by a model editor tool, then the tool must understand the syntax and semantics of the constraint language. Because the choice of language is arbitrary, constraints are an extension mechanism.

In the metamodel a Constraint directly attached to a ModelElement describes semantic restrictions that this ModelElement must obey. Also, any Constraints attached to a Stereotype apply to each ModelElement that bears the given Stereotype.

Attributes

body	A boolean expression defining the constraint. Expressions are written as strings in a designated language. For the model to be well formed, the expression must always yield a true value when evaluated for instances of the constrained elements at any time when the system is stable (i.e., not during the execution of an atomic operation).

Associations

constrainedElement	An ordered list of elements subject to the constraint. The constraint applies to their instances. If the element is a stereotype, then the constraint applies to the elements classified using it.

ModelElement (as extended)

Any model element may have arbitrary tagged values and constraints (subject to these making sense). A model element may have at most one stereotype whose base class must match the UML class of the modeling element (such as Class, Association, Dependency, etc.). The presence of a stereotype may impose implicit constraints on the modeling element and may require the presence of specific tagged values.

Associations

constraint	A constraint that must be satisfied for instances of the model element. A model element may have a set of constraints. The constraint is to be evaluated when the system is stable (i.e., not in the middle of an atomic operation).
stereotype	Designates at most one stereotype that further qualifies the UML class (the base class) of the modeling element. The stereotype does not alter the structure of the base class but it may specify additional constraints and tagged values. All constraints and tagged values on a stereotype apply to the model elements that are classified by the stereotype. The stereotype acts as a "pseudo metaclass" describing the model element.
taggedValue	An arbitrary property attached to the model element. The tag is the name of the property and the value is an arbitrary value. The interpretation of the tagged value is outside the scope of the UML metamodel. A model element may have a set of tagged values, but a single model element may have at most one tagged value with a given tag name. If the model element has a stereotype, then it may specify that certain tags must be present, providing default values.

Stereotype

The stereotype concept provides a way of classifying (marking) elements so that they behave in some respects as if they were instances of new "virtual" metamodel constructs. Instances have the same structure (attributes, associations, operations) as a similar non-stereotyped instance of the same kind. The stereotype may specify additional constraints and required tagged values that apply to instances. In addition, a stereotype may be used to indicate a difference in meaning or usage between two elements with identical structure.

In the metamodel the Stereotype metaclass is a subtype of GeneralizableElement. TaggedValues and Constraints attached to a Stereotype apply to all ModelElements classified by that Stereotype. A stereotype may also specify a geometrical icon to be used for presenting elements with the stereotype.

Stereotypes are GeneralizableElements. If a stereotype is a subtype of another stereotype, then it inherits all of the constraints and tagged values from its stereotype supertype and it must apply to the same kind of base class. A stereotype keeps track of the base class to which it may be applied.

Attributes

baseClass	Specifies the name of a UML modeling element to which the stereotype applies, such as Class, Association, Refinement, Constraint, etc. This is the name of a metaclass, that is, a class from the UML metamodel itself rather than a user model class.
icon	The geometrical description for an icon to be used to present an image of a model element classified by the stereotype.

Associations

extendedElement	Designates the model elements affected by the stereotype. Each one must be a model element of the kind specified by the baseClass attribute.
constraint	(Inherited from ModelElement) Designates constraints that apply to the stereotype itself.
requiredTag	Specifies a set of tagged values, each of which specifies a tag that an element classified by the stereotype is required to have. The value part indicates the default value for the tagged value, that is, the tagged value that an element will be presumed to have if it is not overridden by an explicit tagged value on the element bearing the stereotype. If the value is unspecified, then the element must explicitly specify a tagged value with the given tag.
stereotypeConstraint	Designates constraints that apply to elements bearing the stereotype.

TaggedValue

A tagged value is a (Tag, Value) pair that permits arbitrary information to be attached to any model element. A tag is an arbitrary name, some tag names are predefined as Standard Elements. At most, one tagged value pair with a given tag name may be attached to a given model element. In other words, there is a lookup table of values selected by tag strings that may be attached to any model element.

The interpretation of a tag is (intentionally) beyond the scope of UML, it must be determined by user or tool convention. It is expected that various model analysis tools will define tags to supply information needed for their operation beyond the basic semantics of UML. Such information could include code generation options, model management information, or user-specified additional semantics.

Attributes

tag	A name that indicates an extensible property to be attached to ModelElements. There is a single, flat space of tag names. UML does not define a mechanism for name registry but model editing tools are expected to provide this kind of service. A model element may have at most one tagged value with a given name. A tag is, in effect, a pseudoattribute that may be attached to model elements.
value	An arbitrary value. The value must be expressible as a string for uniform manipulation. The range of permissible values depends on the interpretation applied to the tag by the user or tool; its specification is outside the scope of UML.

Associations

modelElement	A model element that the tag belongs to
stereotype	A tag that applies to elements bearing the stereotype.

2.6.3 Well-Formedness Rules

The following well-formedness rules apply to the Extension Mechanisms package.

Constraint

[1] A Constraint attached to a Stereotype must not conflict with Constraints on any inherited Stereotype, or associated with the baseClass.

```
-- cannot be specified with OCL
```

[2] A Constraint attached to a stereotyped ModelElement must not conflict with any constraints on the attached classifying Stereotype, nor with the Class (the baseClass) of the ModelElement.

```
-- cannot be specified with OCL
```

[3] A Constraint attached to a Stereotype will apply to all ModelElements classified by that Stereotype and must not conflict with any constraints on the attached classifying Stereotype, nor with the Class (the baseClass) of the ModelElement.

```
-- cannot be specified with OCL
```

Stereotype

[1] Stereotype names must not clash with any baseClass names.

```
Stereotype.oclAllInstances->forAll(st | st.baseClass <> self.name)
```

[2] Stereotype names must not clash with the names of any inherited Stereotype.

```
self.allSupertypes->forAll(st : Stereotype | st.name <> self.name)
```

[3] Stereotype names must not clash in the (M2) meta-class namespace, nor with the names of any inherited Stereotype, nor with any baseClass names.

```
--  M2 level not accessible
```

[4] The baseClass name must be provided; icon is optional and is specified in an implementation specific way.

```
self.baseClass <> ''
```

[5] Tag names attached to a Stereotype must not clash with M2 meta-attribute namespace of the appropriate baseClass element, nor with Tag names of any inherited Stereotype.

```
--  M2 level not accessible
```

ModelElement

[1] Tags associated with a ModelElement (directly via a property list or indirectly via a Stereotype) must not clash with any metaattributes associated with the Model Element.

```
--  not specified in OCL
```

[2] A model element must have at most one tagged value with a given tag name.

```
self.taggedValue->forAll(t1, t2 : TaggedValue |
    t1.tag = t2.tag implies t1 = t2)
```

[3] (Required tags because of stereotypes) If T in modelElement.stereotype.require Tag.such that T.value = unspecified, then the modelElement must have a tagged value with name = T.name.

```
self.stereotype.requiredTag->forAll(tag |
        tag.value = Undefined implies self.taggedValue->exists(t |
                t.tag = tag.tag))
```

TaggedValue

No extra well-formedness rules.

2.6.4 Semantics

Constraints, stereotypes, and tagged values apply to model elements, not to instances. They represent extensions to the modeling language itself, not extensions to the run-time environment. They affect the structure and semantics of models. These concepts represent metalevel extensions to UML. However, they do not contain the full power of a heavyweight metamodel extension language and they are designed such that tools need not implement metalevel semantics to implement them.

Within a model, any user-level model element may have a set of constraints and a set of tagged values. The constraints specify restrictions on the instantiation of the model. An instance of a user-level model element must satisfy all of the constraints on its model element for the model to be well-formed. Evaluation of constraints is to be performed when the system is "stable," that is, after the completion of any internal operations when it is waiting for external events. Constraints are written in a designated constraint language, such as OCL, C++, or natural language. The interpretation of the constraints must be specified by the constraint language.

A user-level model element may have at most one tagged value with a given tag name. Each tag name represents a user-defined property applicable to model elements with a unique value for any single model element. The meaning of a tag is outside the scope of UML and must be determined by convention among users and model analysis tools.

It is intended that both constraints and tagged values be represented as strings so that they can be edited, stored, and transferred by tools that may not understand their semantics. The idea is that the understanding of the semantics can be localized into a few modules that make use of the values. For example, a code generator could use tagged values to tailor the code generation process and a process planning tool could use tagged values to denote model element ownership and status. Other modules would simply preserve the uninterpreted values (as strings) unchanged.

A stereotype refers to a baseClass, which is a class in the UML metamodel (not a user-level modeling element) such as Class, Association, Refinement, etc. A stereotype may be a subtype of one or more existing stereotypes (which must all refer the same baseClass, or baseClasses that derive from the same baseClass), in which case it inherits their constraints and required tags and may add additional ones of its own. As appropriate, a stereotype may add new constraints, a new icon for visual display, and a list of default tagged values.

If a user-level model element is classified by an attached stereotype, then the UML base class of the model element must match the base class specified by the stereotype. Any constraints on the stereotype are implicitly attached to the model element. Any tagged values on the stereotype are implicitly attached to the model element. If any of the values are unspecified, then the model element must explicitly define tagged values with the same tag name or the model is ill-formed. (This behaves as if a copy of the tagged values from the stereotype is attached to the model element, so that the default values can be changed). If the stereotype is a subtype of one or more other stereotypes, then any constraints or tagged values from those stereotypes also

apply to the model element (because they are inherited by this stereotype). If there are any conflicts among multiple constraints or tagged values (inherited or directly specified), then the model is ill-formed.

2.6.5 Notes

From an implementation point of view, instances of a stereotyped class are stored as instances of the base class with the stereotype name as a property. Tagged values can and should be implemented as a lookup table (qualified association) of values (expressed as strings) selected by tag names (represented as strings).

Attributes of UML metamodel classes and tag names should be accessible using a single uniform string-based selection mechanism. This allows tags to be treated as pseudo-attributes of the metamodel and stereotypes to be treated as pseudo-classes of the metamodel, permitting a smooth transition to a full metamodeling capability, if desired. See Section 5.2.2, “Mapping of Interface Model into MOF” for a discussion of the relationship of the UML to the OMG Meta Object Facility (MOF).

2.7 Data Types

2.7.1 Overview

The Data Types package is the subpackage that specifies the different data types used by UML. This chapter has a simpler structure than the other packages, since it is assumed that the semantics of these basic concepts are well known.

2.7.2 Abstract Syntax

The abstract syntax for the Data Types package is expressed in graphic notation in Figure 2-16 on page 2-75 and Figure 2-17 on page 2-76.

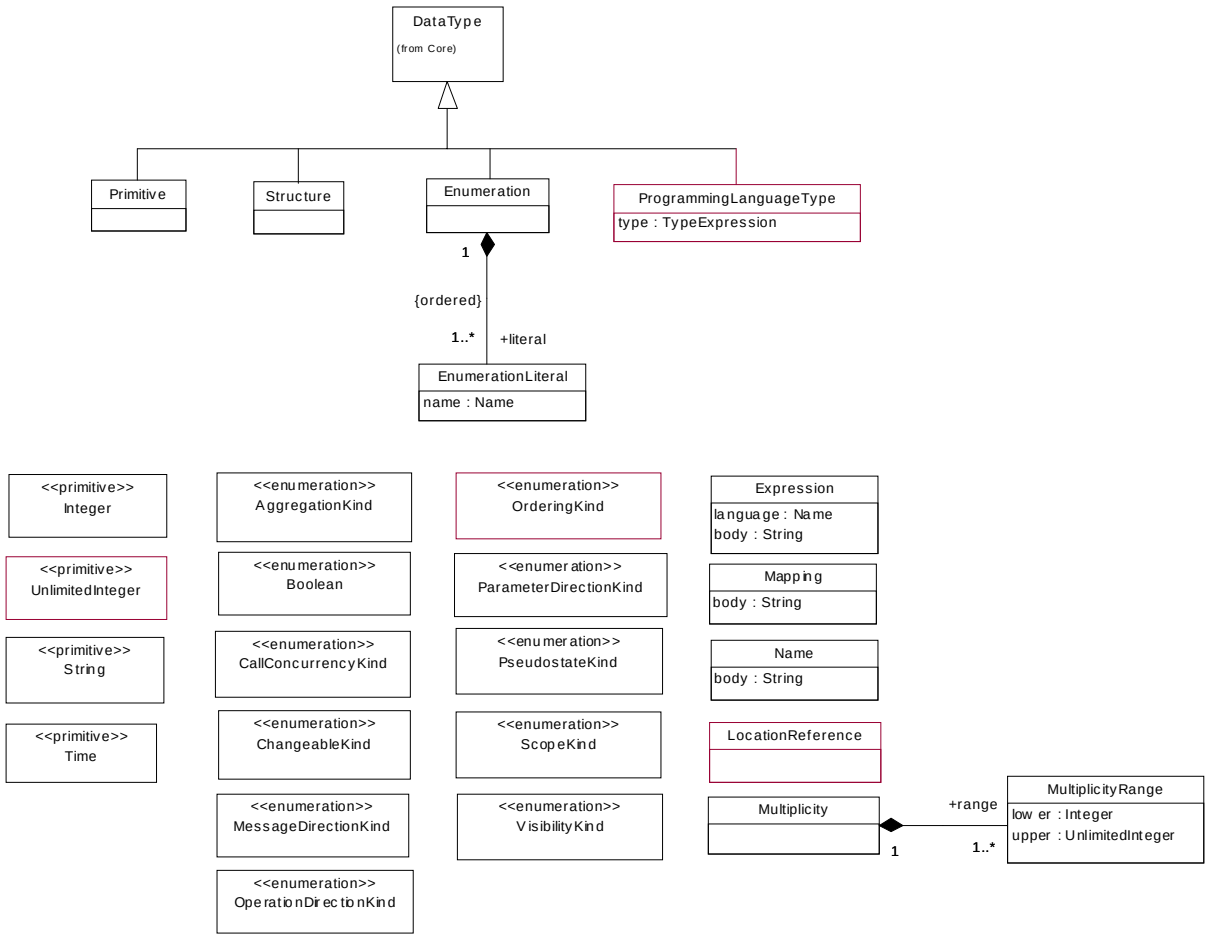


Figure 2-16 Data Types Package - Main

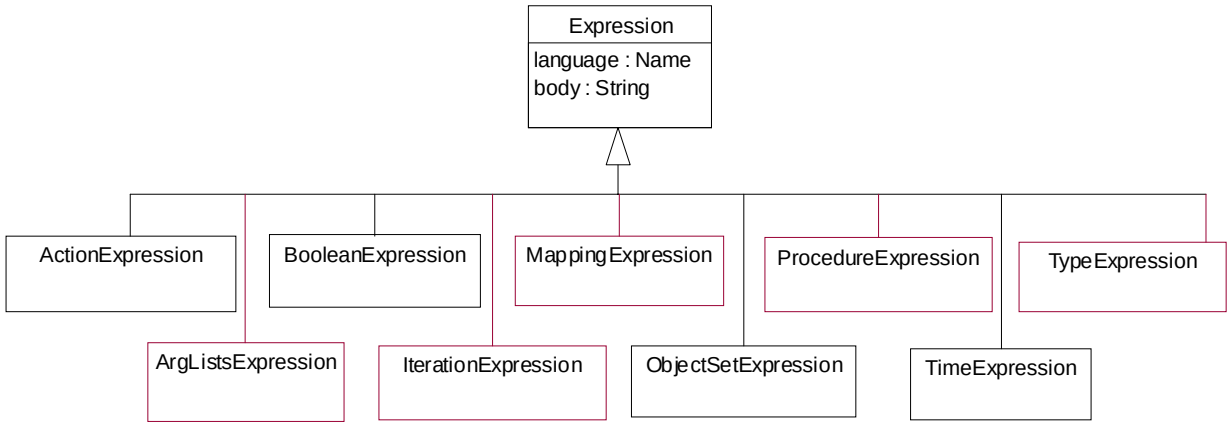


Figure 2-17 Data Types Package - Expressions

In the metamodel the data types are used for declaring the types of the classes' attributes. They appear as strings in the diagrams and not with a separate 'data type' icon. In this way, the sizes of the diagrams are reduced. However, each occurrence of a particular name of a data type denotes the same data type.

Note that these data types are the data types used for defining UML and not the data types to be used by a user of UML. The latter data types will be instances of the DataType metaclass defined in the metamodel.

ActionExpression

An expression that whose evaluation results in the performance of an action.

AggregationKind

In the metamodel AggregationKind defines an enumeration whose values are none, aggregate, and composite. Its value denotes what kind of aggregation an Association is.

ArgListsExpression

In the metamodel ArgListsExpression defines a statement which will result in a set of object lists when it is evaluated.

Boolean

In the metamodel Boolean defines an enumeration whose values are false and true.

BooleanExpression

In the metamodel BooleanExpression defines a statement which will evaluate to an instance of Boolean when it is evaluated.

CallConcurrencyKind

Indicates the concurrency semantics of an operation. It is an enumeration with the choices sequential, guarded, or concurrent.

ChangeableKind

In the metamodel ChangeableKind defines an enumeration whose values are none, frozen, and addOnly. Its value denotes how an AttributeLink or LinkEnd may be modified.

Enumeration

In the metamodel Enumeration defines a special kind of DataType whose range is a list of definable values, called EnumerationLiterals.

EnumerationLiteral

An EnumerationLiteral defines an atom (i.e., with no relevant substructure) that can be compared for equality.

Expression

In the metamodel an Expression defines a statement which will evaluate to a (possibly empty) set of instances when executed in a context. An Expression does not modify the environment in which it is evaluated. An expression contains an expression string and the name of an interpretation language with which to evaluate the string.

Integer

In the metamodel an Integer is an element in the (infinite) set of integers (…-2, -1, 0, 1, 2…).

IterationExpression

In the metamodel IterationExpression defines a string which will evaluate to a iteration control construct in the interpretation language.

LocationReference

Designates a position within a behavior sequences for the insertion of an extension use case. May be a line or range of lines in code, or a state or set of states in a state machine, or some other means in a different kind of specification.

Mapping

In the metamodel a Mapping is an expression that is used for mapping ModelElements. For exchange purposes, it should be represented as a String.

MappingExpression

An expression that evaluates to a mapping.

MessageDirectionKind

In the metamodel MessageDirectionKind defines an enumeration whose values are activation and return. Its value denotes the direction of a Message.

Multiplicity

In the metamodel a Multiplicity defines a non-empty set of non-negative integers. A set which only contains zero ({0}) is not considered a valid Multiplicity. Every Multiplicity has at least one corresponding String representation.

MultiplicityRange

In the metamodel a MultiplicityRange defines a range of integers. The upper bound of the range cannot be below the lower bound. The lower bound must be a nonnegative integer. The upper bound must be a nonnegative integer or the special value *unlimited*, which indicates there is no upper bound on the range.

Name

In the metamodel a Name defines a token which is used for naming ModelElements. Each Name has a corresponding String representation.

ObjectSetExpression

In the metamodel ObjectSetExpression defines a statement which will evaluate to a set of instances when it is evaluated. ObjectSetExpressions are commonly used to designate the target instances in an Action. The expression may be the reserved word “all” when used as the target of a SendAction. It evaluates to all the instances that can receive the signal, as determined by the underlying runtime system.

OperationDirectionKind

In the metamodel OperationDirectionKind defines an enumeration whose values are provide and require. Its value denotes if an Operation is required or provided by a Classifier.

ParameterDirectionKind

In the metamodel ParameterDirectionKind defines an enumeration whose values are in, inout, out, and return. Its value denotes if a Parameter is used for supplying an argument and/or for returning a value.

Primitive

A Primitive defines a special kind of simple DataType, without any relevant substructure.

ProcedureExpression

In the metamodel ProcedureExpression defines a statement which will result in an instance of Procedure when it is evaluated.

ProgrammingLanguageType

Designates a type in the syntax of a particular programming language. Such a type may not correspond exactly to any UML construct. It is defined as a TypeExpression in the given language. A ProgrammingLanguageType can be used for declaring attributes, parameters, and local variables, all of which are to be mapped into programming language code.

PseudostateKind

In the metamodel PseudostateKind defines an enumeration whose values are initial, deepHistory, shallowHistory, join, fork, branch, junction, and final. Its value denotes the possible pseudo states in a state machine.

ScopeKind

In the metamodel ScopeKind defines an enumeration whose values are classifier and instance. Its value denotes if the stored value should be an instance of the associated Classifier or the Classifier itself.

String

In the metamodel a String defines a stream of text.

Structure

A Structure defines a special kind of DataType, that has a fixed number of named parts (a record). It structure is similar to a class.

Time

In the metamodel a Time defines a value representing an absolute or relative moment in time and space. A Time has a corresponding string representation.

TimeExpression

In the metamodel TimeExpression defines a statement which will evaluate to an instance of Time when it is evaluated.

TypeExpression

In the metamodel TypeExpression defines a string that is a programming language type in the interpretation language.

UnlimitedInteger

In the metamodel UnlimitedInteger defines a data type whose range is the nonnegative integers augmented by the special value “unlimited”. It is used for the upper bound of multiplicities.

Uninterpreted

In the metamodel an Uninterpreted is a blob, the meaning of which is domain-specific and therefore not defined in UML.

VisibilityKind

In the metamodel VisibilityKind defines an enumeration whose values are public, protected, and private. Its value denotes how the element to which it refers is seen outside the enclosing name space.

Part 3 - Behavioral Elements

This section defines the superstructure for behavioral modeling in UML, the Behavioral Elements package. The Behavioral Elements package consists of four lower-level packages: Common Behavior, Collaborations, Use Cases, and State Machines.

2.8 Behavioral Elements Package

Common Behavior specifies the core concepts required for behavioral elements. The Collaborations package specifies a behavioral context for using model elements to accomplish a particular task. The Use Case package specifies behavior using actors and use cases. The State Machines package defines behavior using finite-state transition systems. The Activity Graphs package defines a special case of a state machine that is used to model proocesses.

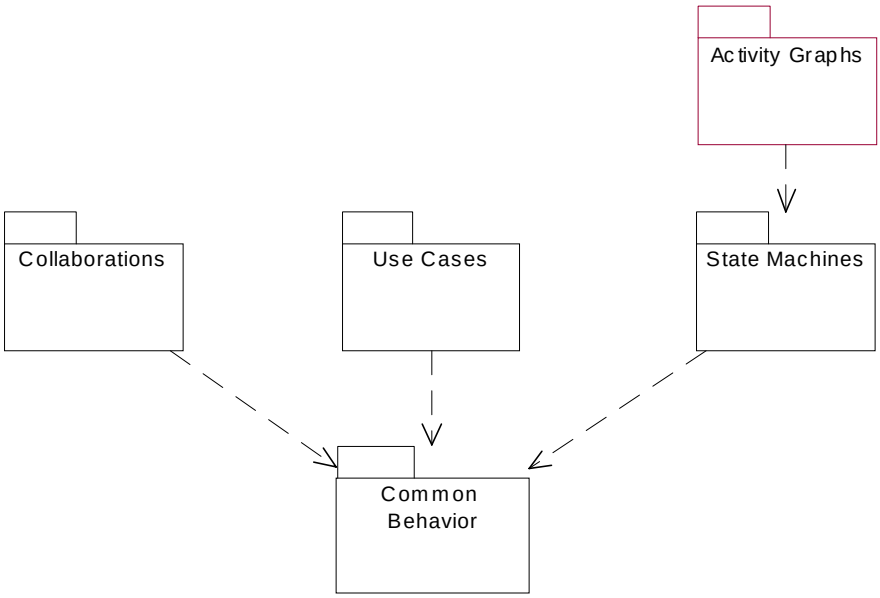


Figure 2-18 Behavioral Elements Package

2.9 Common Behavior

2.9.1 Overview

The Common Behavior package is the most fundamental of the subpackages that compose the Behavioral Elements package. It specifies the core concepts required for dynamic elements and provides the infrastructure to support Collaborations, State Machines and Use Cases.

The following sections describe the abstract syntax, well-formedness rules and semantics of the Common Behavior package.

2.9.2 Abstract Syntax

The abstract syntax for the Common Behavior package is expressed in graphic notation in the following figures. Figure 2-19 on page 2-82 shows the model elements that define Signals and Receptions.

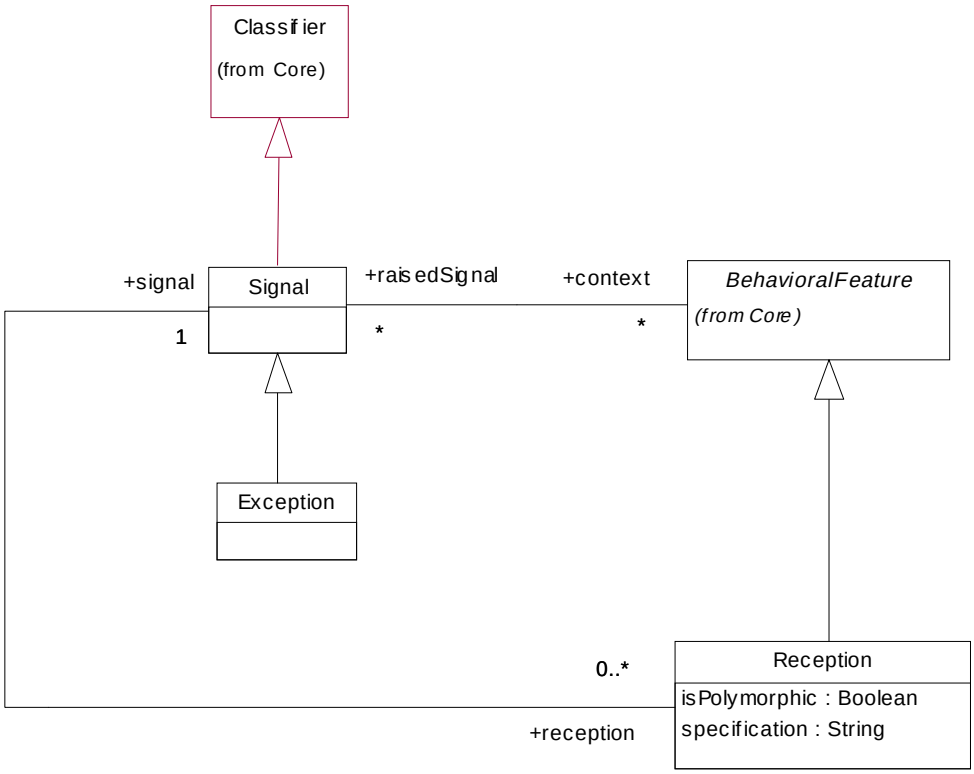


Figure 2-19 Common Behavior - Signals

Figure 2-20 on page 2-83 illustrates the model elements that specify various actions, such as CreateAction, CallAction and SendAction.

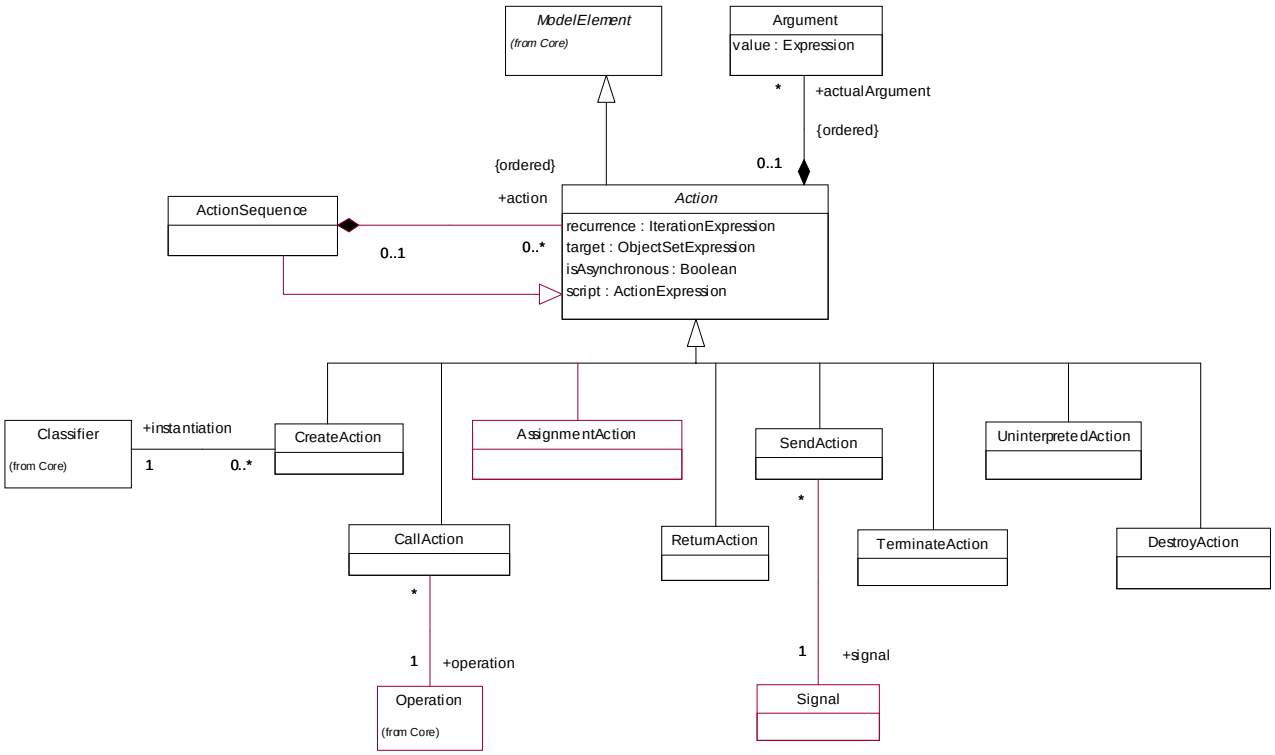


Figure 2-20 Common Behavior - Actions

Figure 2-21 on page 2-84 shows the model elements that define Instances and Links.

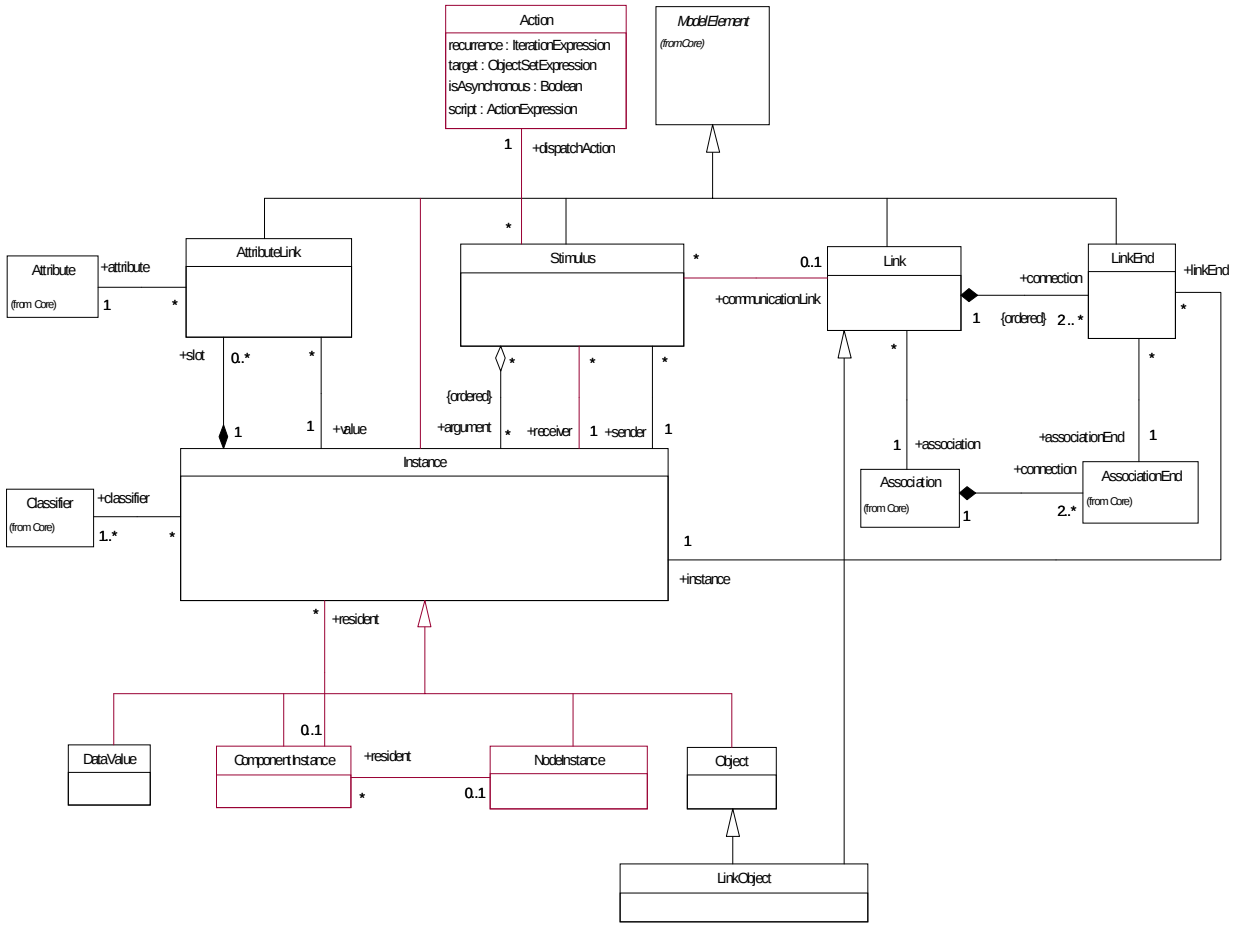


Figure 2-21 Common Behavior - Instances and Links

The following metaclasses are contained in the Common Behavior package.

Action

An action is a specification of an executable statement that forms an abstraction of a computational procedure that results in a change in the state of the model, and can be realized by sending a message to an object or modifying a link or a value of an attribute.

In the metamodel an Action may be part of an ActionSequence and may contain a specification of a target as well as a specification of the actual arguments, i.e. a list of Arguments containing expressions for determining the actual Instances to be used when the Action is performed (or executed).

The target metaattribute is of type ObjectSetExpression which, when executed, resolves into zero or more specific Instances that are the intended target of the Action, like a receiver of a dispatched Signal. The recurrence metaattribute specifies how the target set is iterated when the action is executed. It is not defined within UML if the action is applied sequentially or in parallel to the target instances.

Action is an abstract metaclass.

Attributes

isAsynchronous	Indicates if a dispatched Stimulus is asynchronous or not.
recurrence	An Expression stating how many times the Action should be performed.
script	An ActionExpression describing the effects of the Action.
target	An ObjectSetExpression which determines the target of the Action.

Associations

actualArgument	A sequence of Expressions which determines the actual arguments needed when evaluating the Action.

ActionSequence

An action sequence is a collection of actions.

In the metamodel an ActionSequence is an Action, which is an aggregation of other Actions. It describes the behavior of the owning State or Transition.

Associations

action	A sequence of Actions performed sequentially as an atomic unit.

Argument

An argument is an expression describing how to determine the actual values passed in a dispatched request. It is aggregated within an action.

In the metamodel an Argument is a part of an Action and contains a metaattribute, value, of type Expression. It states how the actual argument is determined when the owning Action is executed.

Attributes

value	An Expression determining the actual Instance when evaluated.

AssignmentAction

An assignment action is an action which assigns a new value to a link or an attribute link.

In the metamodel the AssignmentAction is an Action. The Instance to be assigned to the Link or the AttributeLink is designated by the Argument of the AssignmentAction.

Associations

association	The Association which will be assigned when the AssignmentAction is performed.
attribute	The Attribute which will be assigned when the AssignmentAction is performed.

AttributeLink

An attribute link is a named slot in an instance, which holds the value of an attribute.

In the metamodel AttributeLink is a piece of the state of an Instance and holds the value of an Attribute.

Associations

value	The Instance which is the value of the AttributeLink.
attribute	The Attribute from which the AttributeLink originates.

CallAction

A call action is an action resulting in an invocation of an operation on an instance. A call action can be synchronous or asynchronous, indicating whether the operation is invoked synchronously or asynchronously.

In the metamodel the CallAction is an Action. The designated Instance or set of Instances is specified via the target expression, and the actual arguments are designated via the argument association inherited from Action. The Operation to be invoked is specified by the associated Operation.

Attributes

isAsynchronous	(inherited from Action) Indicates if a dispatched operation is asynchronous or not. • False - indicates that the caller waits for the completion of the execution of the operation. • True - Indicates that the caller does not wait for the completion of the execution of the operation but continues immediately.

Associations

operation	The operation which will be invoked when the Action is executed.

ComponentInstance

A component instance is an instance of a component that resides on a node instance. A component instance may have a state.

In the metamodel, a ComponentInstance is an Instance that originates from a Component. It may be associated with a set of Instance, and may reside on a NodeInstance.

Associations

resident	A collection of Instances that exist inside the ComponentInstance.

CreateAction

A create action is an action resulting in the creation of an instance of some classifier.

In the metamodel the CreateAction is an Action. The Classifier to be instantiated is designated by the instantiation association of the CreateAction. A CreateAction has no target instance.

Associations

instantiation	The Classifier of which an Instance will be created of when the CreateAction is performed.

DestroyAction

A destroy action is an action results in the destruction of an object specified in the action.

In the metamodel a DestroyAction is an Action. The designated object is specified by the target association of the Action.

DataValue

A data value is an instance with no identity.

In the metamodel DataValue is a child of Instance that cannot change its state, i.e. all Operations that are applicable to it are pure functions or queries. DataValues are typically used as attribute values.

Exception

An exception is a signal raised by behavioral features typically in case of execution faults.

In the metamodel, Exception is derived from Signal. An Exception is associated with the BehavioralFeatures that raise it.

Associations

context	(Inherited from Signal) The set of BehavioralFeatures that raise the exception.

Instance

The instance construct defines an entity to which a set of operations can be applied and which has a state that stores the effects of the operations.

In the metamodel Instance is connected to at least one Classifier which declares its structure and behavior. It has a set of attribute values and is connected to a set of Links, both sets matching the definitions of its Classifiers. The two sets implement the current state of the Instance. Instance is an abstract metaclass.

Associations

attributeLink	The set of AttributeLinks that holds the attribute values of the Instance.
linkEnd	The set of LinkEnds of the connected Links that are attached to the Instance.
classifier	The set of Classifiers that declare the structure of the Instance.

Tagged Values

persistent	Persistence denotes the permanence of the state of the instance, marking it as *transitory* (its state is destroyed when the instance is destroyed) or *persistent* (its state is not destroyed when the instance is destroyed).

Link

The link construct is a connection between instances.

In the metamodel Link is an instance of an Association. It has a set of LinkEnds that matches the set of AssociationEnds of the Association. A Link defines a connection between Instances.

Associations

association	The Association that is the declaration of the link.
linkRole	The sequence of LinkEnds that constitute the Link.

LinkEnd

A link end is an end point of a link.

In the metamodel LinkEnd is the part of a Link that connects to an Instance. It corresponds to an AssociationEnd of the Link's Association.

Associations

instance	The Instance connected to the LinkEnd.
associationEnd	The AssociationEnd that is the declaration of the LinkEnd..

Standard Constraints

association	Association is a constraint applied to a link-end, specifying that the corresponding instance is visible via association.
destroyed	Destroyed is a constraint applied to a link-end, specifying that the corresponding instance is destroyed during the execution.
global	Global is a constraint applied to a link-end, specifying that the corresponding instance is visible because it is in a global scope relative to the link.
local	Local is a constraint applied to link-end, specifying that the corresponding instance is visible because it is in a local scope relative to the link.
new	New is a constraint applied to a link-end, specifying that the corresponding instance is created during the execution.
parameter	Parameter is a constraint applied to a link-end, specifying that the corresponding instance is visible because it is a parameter relative to the link.
self	Self is a constraint applied to a link-end, specifying that the corresponding instance is visible because it is the dispatcher of a request.
transient	Transient is a constraint applied to a link-end, specifying that the corresponding instance is created and destroyed during the execution.

LinkObject

A link object is a link with its own set of attribute values and to which a set of operations may be applied.

In the metamodel LinkObject is a connection between a set of Instances, where the connection itself may have a set of attribute values and to which a set of Operations may be applied. It is a child of both Object and Link.

NodeInstance

A node instance is an instance of a node. A collection of component instances may reside on the node instance.

In the metamodel NodeInstance is an Instance that originates from a Node. Each ComponentInstance that resides on a NodeInstance must be an instance of a Component that resides on the corresponding Node.

Associations

resident	A collection of ComponentInstances that reside on the NodeInstances.

Object

An object is an instance that originates from a class.

In the metamodel Object is a subclass of Instance and it originates from at least one Class. The set of Classes may be modified dynamically, which means that the set of features of the Object may change during its life-time.

Reception

A reception is a declaration stating that a classifier is prepared to react to the receipt of a signal. The reception designates a signal and specifies the expected behavioral response. A reception is a summary of expected behavior. The details of handling a signal are specified by a state machine.

In the metamodel Reception is a child of BehavioralFeature and declares that the Classifier containing the feature reacts to the signal designated by the reception feature. The isPolymorphic attribute specifies whether the behavior is polymorphic or not; a true value indicates that the behavior is not always the same and may be affected by state or subclassing. The specification indicates the expected response to the Signal.

Attributes

isAbstract	If true, then the reception does not have an implementation, and one must be supplied by a descendant. If false, the reception must have an implementation in the classifier or inherited from an ancestor.
isLeaf	If true, then the implementation of the reception may not be overriden by a descendant classifier. If false, then the implementation of the reception may be overridden by a descendant classifier (but it need not be overridden).
isRoot	If true, then the classifier must not inherit a declaration of the same reception. If false, then the classifier may (but need not) inherit a declaration of the same reception. (But the declaration must match in any case; a classifier may not modify an inherited declaration of a reception.)
specification	A description of the effects of the classifier receiving a Signal, stated by a String.

Associations

signal	The Signal that the Classifier is prepared to handle.

ReturnAction

A return action is an action that results in returning a value to a caller.

In the metamodel ReturnAction is an Action, which causes values to be passed back to the activator. The values are represented by the arguments inherited from Action. A ReturnAction has no explicit target.

SendAction

A send action is an action that results in the (asynchronous) sending of a signal. The signal can be directed to a set of receivers via an objectSetExpression, or sent implicitly to an unspecified set of receivers, defined by some external mechanism. For example, if the signal is an exception, the receiver is determined by the underlying runtime system mechanisms.

In the metamodel SendAction is an Action. It is associated with the Signal to be raised, and its actual arguments are specified by the argument association, inherited from Action.

Associations

signal	The signal which will be invoked when the Action is executed.

Signal

A signal is a specification of an asynchronous stimulus communicated between instances. The receiving instance handles the signal by a state machine. Signal is a generalizable element and is defined independently of the classes handling the signal. A reception is a declaration that a class handles a signal, but the actual handling is specified by a state machine.

In the metamodel Signal is a child to Classifier, with the parameters expressed as Attributes. A Signal is always asynchronous. A Signal is associated with the BehavioralFeatures that raise it.

Associations

context	The set of BehavioralFeatures that raise the signal.
reception	A set of Receptions that indicates Classes prepared to handle the signal.

Stimulus

A stimulus reifies a communication between two instances.

In the metamodel Stimulus is a communication, i.e. a Signal sent to an Instance, or an invocation of an Operation. It can also be a request to create an Instance, or to destroy an Instance It has a sender, a receiver, and may have a set of actual arguments, all being Instances.

Associations

arguments	The sequence of Instances being the arguments of the MessageInstance.
communicationLink	The Link, which is used for communication.
dispatchAction	The Action which caused the Stimulus to be dispatched when it was executed.
receiver	The Instance which receives the Stimulus.
sender	The Instance which sends the Stimulus.

TerminateAction

A terminate action results in self-destruction of an object.

In the metamodel TerminateAction is a child of Action. The target of a TerminateAction is implicitly the Instance executing the action, so there is no explicit target.

UninterpretedAction

An uninterpreted action represents an action that is not explicitly reified in the UML

Taken to the extreme, any action is a call or raise on some instance, like in Smalltalk. However, in more practical terms, uninterpreted actions can be used to model language-specific actions that are neither call actions nor send actions, and are not easily categorized under the other types of actions.

2.9.3 Well-Formedness Rules

The following well-formedness rules apply to the Common Behavior package.

AttributeLink

[1] The type of the Instance must match the type of the Attribute.

```
self.value.classifier->union (
    self.value.classifier.allParents)->includes (
        self.attribute.type)
```

CallAction

[1] The number of arguments be the same as the number of the Operation.

```
self.actualArgument->size = self.operation.parameter->size
```

ComponentInstance

[1] A ComponentInstance originates from exactly one Component.

```
self.classifier->size = 1
and
self.classifier.oclIsKindOf (Component)
```

CreateAction

[1] A CreateAction does not have a target expression.

```
self.target->isEmpty
```

DestroyAction

[1] A DestroyAction should not have arguments

```
self.actualArgument->size = 0
```

DataValue

[1] A DataValue originates from exactly one Classifier, which is a DataType.

```
(self.classifier->size = 1)
and
self.classifier.oclIsKindOf(DataType)
```

[2] A DataValue has no AttributeLinks.

```
self.slot->isEmpty
```

Instance

[1] The AttributeLinks match the declarations in the Classifiers.

```
self.slot->forAll ( al |
    self.classifier->exists ( c |
        c.allAttributes->includes ( al.attribute ) ) )
```

[2] The Links matches the declarations in the Classifiers.

```
self.allLinks->forAll ( l |
    self.classifier->exists ( c |
        c.allAssociations->includes ( l.association ) ) )
```

[3] If two Operations have the same signature they must be the same.

```
self.classifier->forAll ( c1, c2 |
    c1.allOperations->forAll ( op1 |
        c2.allOperations->forAll ( op2 |
            op1.hasSameSignature (op2)  implies op1 = op2 ) ) )
```

[4] There are no name conflicts between the AttributeLinks and opposite LinkEnds.

```
self.slot->forAll( al |
    not self.allOppositeLinkEnds->exists( le | le.name = al.name ) )
and
self.allOppositeLinkEnds->forAll( le |
    not self.slot->exists( al | le.name = al.name ) )
```

[6] The number of associated Instances in one opposite LinkEnds must match the multiplicity of that AssociationEnd.

```
self.slot->forAll( al |
    not self.allOppositeLinkEnds->exists( le | le.name = al.name ) )
and
self.allOppositeLinkEnds->forAll( le |
    not self.slot->exists( al | le.name = al.name ) )
```

Additional operations

[1] The operation allLinks results in a set containing all Links of the Instance itself.

```
allLinks : set(Link);
allLinks = self.linkEnd.link
```

[2] The operation allOppositeLinkEnds results in a set containing all LinkEnds of Links connected to the Instance with another LinkEnd.

```
allOppositeLinkEnds : set(Link);
allOppositeLinkEnds = self.allLinks.linkEnd->select (le |
    le.instance <> self)
```

Link

[1] The set of LinkEnds must match the set of AssociationEnds of the Association.

```
Sequence {1..self.connection->size}->forAll ( i |
    self.connection->at (i).associationEnd =
    self.association.connection->at (i) )
```

[2] There are not two Links of the same Association which connects the same set of Instances in the same way.

```
self.association.link->forAll ( l |
    Sequence {1..self.connection->size}->forAll ( i |
        self.connection->at (i).instance =
        l.connection->at (i).instance )
            implies self = l )
```

LinkEnd

[1] The type of the Instance must match the type of the AssociationEnd.

```
self.instance.classifier->union (
    self.instance.classifier.allParents)->includes (
        self.associationEnd.type)
```

LinkObject

[1] One of the Classifiers must be the same as the Association.

```
self.classifier->includes(self.association)
```

[2] The Association must be a kind of AssociationClass.

```
self.association.oclIsKindOf(AssociationClass)
```

NodeInstance

[1] Each of the Classifiers must be a kind of Node.

```
self.classifier->forAll ( c | c.oclIsKindOf(Node))
```

Object

[1] Each of the Classifiers must be a kind of Class.

```
self.classifier->forAll ( c | c.oclIsKindOf(Class))
```

Reception

[1] A Reception can not be a query.

```
not self.isQuery
```

SendAction

[1] The number of arguments is the same as the number of parameters of the Signal.

```
self.actualArgument->size = self.signal.allAttributes->size
```

[2] A Signal is always asynchronous.

```
self.isAsynchronous
```

Signal

No extra well-formedness rules.

Stimulus

[1] The number of arguments must match the number of Arguments of the Action.

```
self.dispatchAction.actualArgument->size = self.argument->size
```

[2] The Action must be a SendAction, a CallAction, a CreateAction, or a DestroyAction.

```
self.dispatchAction.oclIsKindOf (SendAction) or
```

```
self.dispatchAction.oclIsKindOf (CallAction) or
self.dispatchAction.oclIsKindOf (CreateAction) or
self.dispatchAction.oclIsKindOf (DestroyAction)
```

TerminateAction

[1] A TerminateAction has no arguments.

```
self.actualArguments->size = 0
```

[2] A TerminateAction has no target expression.

```
self.target->isEmpty
```

2.9.4 Semantics

This section provides a description of the semantics of the elements in the Common Behavior package.

Object and DataValue

An object is an instance that originates from a class, it is structured and behaves according to its class. All objects originating from the same class are structured in the same way, although each of them has its own set of attribute links. Each attribute link references an instance, usually a data value. The number of attribute links with the same name fulfills the multiplicity of the corresponding attribute in the class. The set may be modified according to the specification in the corresponding attribute, e.g. each referenced instance must originate from (a specialization of) the type of the attribute, and attribute links may be added or removed according to the changeable property of the attribute.

An object may have multiple classes (i.e., it may originate from several classes). In this case, the object will have all the features declared in all of these classes, both the structural and the behavioral ones. Moreover, the set of classes (i.e., the set of features that the object conforms to) may vary over time. New classes may be added to the object and old ones may be detached. This means that the features of the new classes are dynamically added to the object, and the features declared in a class which is removed from the object are dynamically removed from the object. No name clashes between attributes links and opposite link ends are allowed, and each operation which is applicable to the object should have a unique signature.

Another kind of instance is data value, which is an instance with no identity. Moreover, a data value cannot change its state; all operations that are applicable to a data value are queries and do not cause any side effects. Since it is not possible to differentiate between two data values that appear to be the same, it becomes more of a philosophical issue whether there are several data values representing the same value or just one for each value-it is not possible to tell. In addition, a data value cannot change its data type.

Link

A link is a connection between instances. Each link is an instance of an association, i.e. a link connects instances of (specializations of) the associated classifiers. In the context of an instance, an opposite end defines the set of instances connected to the instance via links of the same association and each instance is attached to its link via a link-end originating from the same association-end. However, to be able to use a particular opposite end, the corresponding link end attached to the instance must be navigable. An instance may use its opposite ends to access the associated instances. An instance can communicate with the instances of its opposite ends and also use references to them as arguments or reply values in communications.

A link object is a special kind of link, it is at the same time also an object. Since an object may change it classes this is also true for a link object. However, one of the classes must always be an association class.

Signal, Exception and Stimulus

Several kinds of requests exist between instances, e.g. sending a signal and invoking an operation. The former is used to trigger a reaction in the receiver in an asynchronous way and without a reply, while the latter is applies an operation to an instance, which can be either done synchronously or asynchronously and may require a reply from the receiver to the sender. Other kinds of requests are: create a new instance, or deleting an already existing instance. When an instance communicates with another instance a stimulus is passed between the two instances. Each stimulus has a sender instance and a receiver instance, and possibly a sequence of arguments according to the specifying signal or operation. The stimulus uses a link between the sender and the receiver for communication. This link may be missing if the receiver is an argument inside the current activation, a local or global variable, or if the stimulus is sent to the sender instance, itself. Moreover, a stimulus is dispatched by an action, e.g. a call action or a send action. The action specifies the request made by the stimulus, like the operation to be invoked or the signal event to be raised, as well as how the actual arguments of the stimulus are determined.

A signal may be attached to a classifier, which means that instances of the classifier will be able to receive that signal. This is facilitated by declaring a reception by the classifier. An exception is a special kind of signal, typically used to signal fault situations. The sender of the exception aborts execution and execution resumes with the receiver of the exception, which may be the sender itself. Unlike other signals, the receiver of an exception is determined implicitly by the interaction sequence during execution; it is not explicitly specified as the target of the send action.

The reception of a stimulus originating from a call action by an instance causes the invocation of an operation on the receiver. The receiver executes the method that is found in the full descriptor of the class that corresponds to the operation. The reception of a stimulus originating from a signal by an instance may cause a transition and subsequent effects as specified by the state machine for the classifier of the recipient. This form of behavior is described in the State Machines package. Note that the invoked behavior is described by methods and state machine transitions. Operations and receptions merely declare that a classifier accepts a given operation invocation or signal but they do not specify the implementation.

Action

An action is a specification of a computable statement. Each kind of action is defined as a subclass of action. The following kinds of actions are defined:

- send action is an action in which a stimulus is created that causes a signal event for the receiver(s).
- call action is an action in which a stimulus is created that causes an operation to be invoked on the receiver.
- create action is an action in which an instance is created based on the definitions of the specified set of classifiers.
- terminate action is an action in which an instance causes itself to cease to exist.
- destroy action is an action in which an instance causes another instance to cease to exist.
- return action is an action that returns a value to a caller.
- assignment action is an action that assigns an instance to an attribute link or a link.
- uninterpreted action is an action that has no interpretation in UML.

Each action specifies the target of the action and the arguments of the action. The target of an action is an object set expression which resolves into zero or more instances when the action is executed, e.g the receiver of a stimulus or the instance to be destroyed. The action also specifies if it should iterate over the set of target instances (recurrence). Note, however, that UML does not define if the action is applied to the target instances sequentially or in parallel. The recurrence can also (in the degenerated case) be used for specification of a condition, which must be fulfilled if the action is to be applied to the target; otherwise, the request is neglected.

The arguments of the action resolve into a sequence of instances when the action is executed. These instances are the actual arguments of e.g. the stimulus being dispatched by the action, i.e. the instances passed with a signal or the instances used in an operation invocation. The argument sequence may be dependent on the recurrence, i.e. the arguments may vary dependent on the actual target.

An action is always executed within the context of an instance, so the target set expression and the argument expressions are evaluated within an instance.

2.10 Collaborations

2.10.1 Overview

The Collaborations package is a subpackage of the Behavioral Elements package. It specifies the concepts needed to express how different elements of a model interact with each other from a structural point of view. The package uses constructs defined in the Foundation package of UML as well as in the Common Behavior package.

A Collaboration defines a specific way to use the Model Elements in a Model. It describes how different kinds of Classifiers and their Associations are to be used in accomplishing a particular task. The Collaboration defines a restriction of, or a projection of, a collection of Classifiers, i.e. what properties Instances of the participating Classifiers must have when performing a particular collaboration. The same Classifier or Association can appear in several Collaborations, and several times in one Collaboration, each time in a different role. In each appearance it is specified which of the properties of the Classifier or the Association are needed in that particular usage. These properties are a subset of all the properties of that Classifier or Association. A set of Instances and Links conforming to the participants specified in the Collaboration cooperate when the specified task is performed. Hence, the Classifier structure implies the possible collaboration structures of conforming Instances. A Collaboration is a GeneralizableElement. This implies that one Collaboration may specify a task which is a specialization of another Collaboration's task. A Collaboration may be presented in a diagram, either showing the restricted views of the participating Classifiers and Associations, or by showing Instances and Links conforming to the restricted views.

Collaborations can be used for expressing several different things, like how use cases are realized, actor structures of ROOM, OORam role models, and collaborations as defined in Catalysis. They are also used for setting up the context of Interactions and for defining the mapping between the specification part and the realization part of a Subsystem.

An Interaction defined in the context of a Collaboration specifies the details of the communications that should take place in accomplishing a particular task. A communication is specified with a Message, which defines the roles of the sender and the receiver Instances, as well as the Action that will cause the communication. The order of the communications is also specified by the Interaction.

The following sections describe the abstract syntax, well-formedness rules and semantics of the Collaborations package.

2.10.2 Abstract Syntax

The abstract syntax for the Collaborations package is expressed in graphic notation in Figure 2-22.

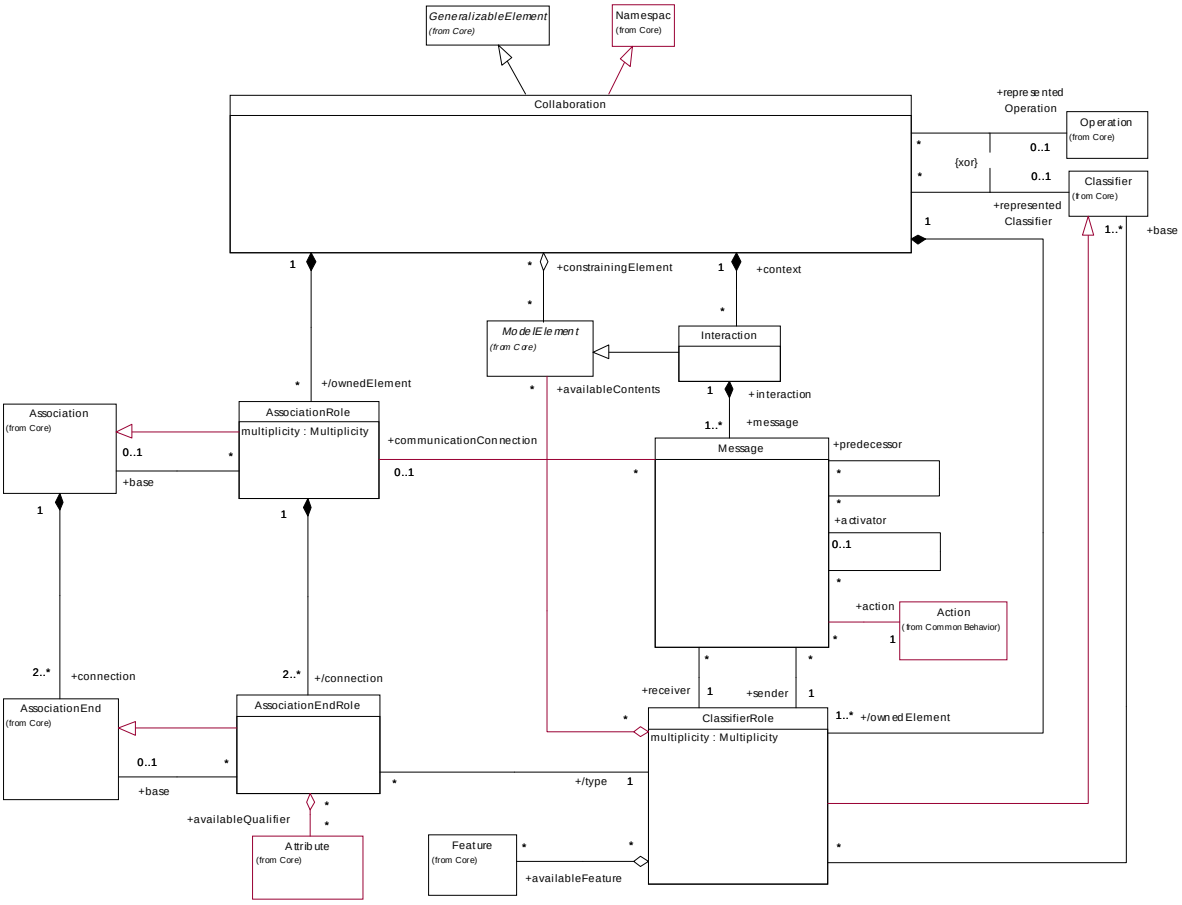


Figure 2-22 Collaborations

AssociationEndRole

An association-end role specifies an endpoint of an association as used in a collaboration.

In the metamodel an AssociationEndRole is part of an AssociationRole and specifies the connection of an AssociationRole to a ClassifierRole. It is related to the AssociationEnd, declaring the corresponding part in an Association.

Attributes

collaboration-Multiplicity	The number of LinkEnds playing this role in a Collaboration.

Associations

availableQualifier	The subset of Qualifiers that are used in the Collaboration.
base	The AssociationEnd which the AssociationEndRole is a projection of.

AssociationRole

An association role is a specific usage of an association needed in a collaboration.

In the metamodel an AssociationRole specifies a restricted view of an Association used in a Collaboration. An AssociationRole is a composition of a set of AssociationEndRoles corresponding to the AssociationEnds of its base Association.

Attributes

multiplicity	The number of Links playing this role in a Collaboration.

Associations

base	The Association which the AssociationRole is a view of.

ClassifierRole

A classifier role is a specific role played by a participant in a collaboration. It specifies a restricted view of a classifier, defined by what is required in the collaboration.

In the metamodel a ClassifierRole specifies one participant of a Collaboration, i.e. a role Instances conform to. A ClassifierRole defines a set of Features, which is a subset of those available in the base Classifiers, as well as a subset of ModelElements contained in the base Classifiers, that are used in the role. The ClassifierRole may be connected to a set of AssociationRoles via AssociationEndRoles. As ClassifierRole is a kind of Classifier, a Generalization relationship may be defined between two ClassifierRoles. The child role is a specialization of the parent, i.e. the Features and the contents of the child includes the (possibly specialized) Features and contents of the parent.

Attributes

multiplicity	The number of Instances playing this role in a Collaboration.

Associations

availableContents	The subset of ModelElements contained in the base Classifier which is used in the Collaboration.
availableFeature	The subset of Features of the base Classifier which is used in the Collaboration.
base	The Classifiers which the ClassifierRole is a view of.

Collaboration

A collaboration describes how an operation or a classifier, like a use case, is realized by a set of classifiers and associations used in a specific way. The collaboration defines a set of roles to be played by instances and links, as well as a set of interactions that define the communication between the instances when they play the roles.

In the metamodel a Collaboration contains a set of ClassifierRoles and AssociationRoles, which represent the Classifiers and Associations that take part in the realization of the associated Classifier or Operation. The Collaboration may also contain a set of Interactions that are used for describing the behavior performed by Instances conforming to the participating ClassifierRoles.

A Collaboration specifies a view (restriction, slice, projection) of a model of Classifiers. The projection describes the required relationships between Instances that conform to the participating ClassifierRoles, as well as the required subsets of the Features and contained ModelElements of these Classifiers. Several Collaborations may describe different projections of the same set of Classifiers. Hence, a Classifier can be a base for several ClassifierRoles.

A Collaboration may also reference a set of ModelElements, usually Classifiers and Generalizations, needed for expressing structural requirements, such as Generalizations required between the Classifiers themselves to fulfill the intent of the Collaboration.

A Collaboration is a GeneralizableElement which implies that one Collaboration may specify a task which is a specialization of the task of another Collaboration.

Associations

constrainingElement	The ModelElements that add extra constraints, like Generalization and Constraint, on the ModelElements participating in the Collaboration.
interaction	The set of Interactions that are defined within the Collaboration.
ownedElement	(Inherited from Namespace) The set of roles defined by the Collaboration. These are ClassifierRoles and AssociationRoles.
representedClassifier	The Classifier the Collaboration is a realization of. (Used if the Collaboration represents a Classifier.)
representedOperation	The Operation the Collaboration is a realization of. Used if the Collaboration represents an Operation.)

Interaction

An interaction specifies the communication between instances performing a specific task. Each interaction is defined in the context of a collaboration.

In the metamodel an Interaction contains a set of Messages specifying the communication between a set of Instances conforming to the ClassifierRoles of the owning Collaboration.

Associations

context	The Collaboration which defines the context of the Interaction.
message	The Messages that specify the communication in the Interaction.

Message

A message defines a particular communication between instances that is specified in an interaction.

In the metamodel a Message defines one specific kind of communication in an Interaction. A communication can be e.g. raising a Signal, invoking an Operation, creating or destroying an Instance. The Message specifies not only the kind of communication, but also the roles of the sender and the receiver, the dispatching Action, and the role played by the communication Link. Furthermore, the Message defines the relative sequencing of Messages within the Interaction.

Associations

action	The Action which causes a Stimulus to be sent according to the Message.
activator	The Message which invokes the behavior causing the dispatching of the current Message.
communication-Connection	The AssociationRole played by the Links used in the communications specified by the Message.
receiver	The role of the Instance that receives the communication and reacts to it.
predecessor	The set of Messages whose completion enables the execution of the current Message. All of them must be completed before execution begins.
sender	The role of the Instance that invokes the communication and possibly receives a response.

2.10.3 Well-Formedness Rules

The following well-formedness rules apply to the Collaborations package.

AssociationEndRole

[1] The type of the ClassifierRole must conform to the type of the base AssociationEnd.

```
self.type.base = self.base.type
or
self.type.base.allParents->includes (self.base.type)
```

[2] The type must be a kind of ClassifierRole.

```
self.type.oclIsKindOf (ClassifierRole)
```

[3] The qualifiers used in the AssociationEndRole must be a subset of those in the base AssociationEnd.

```
self.base.qualifier->includesAll (self.availableQualifier)
```

[4] In a collaboration an association may only be used for traversal if it is allowed by the base association.

```
self.isNavigable implies self.base.isNavigable
```

AssociationRole

[1] The AssociationEndRoles must conform to the AssociationEnds of the base Association.

```
Sequence{ 1..(self.connection->size) }->forAll (index |
    self.connection->at(index).base =
    self.base.connection->at(index))
```

[2] The endpoints must be a kind of AssociationEndRoles.

```
self.connection->forAll( r | r.oclIsKindOf (AssociationEndRole) )
```

ClassifierRole

[1] The AssociationRoles connected to the ClassifierRole must match a subset of the Associations connected to the base Classifiers.

```
self.allAssociations->forAll( ar |
    self.base.allAssociations->exists ( a | ar.base = a ) )
```

[2] The Features and contents of the ClassifierRole must be subsets of those of the base Classifiers.

```
self.base.allFeatures->includesAll (self.allAvailableFeatures)
and
self.base.allContents->includesAll (self.allAvailableContents)
```

[3] A ClassifierRole does not have any Features of its own.

```
self.allFeatures->isEmpty
```

Additional operations

[1] The operation *allAvailableFeatures* results in the set of all Features contained in the ClassifierRole together with those contained in the parents.

```
allAvailableFeatures : Set(Feature);
allAvailableFeatures = self.availableFeature->union
    (self.parent.allAvailableFeatures)
```

[2] The operation *allAvailableContents* results in the set of all ModelElements contained in the ClassifierRole together with those contained in the parents.

```
allAvailableContents : Set(ModelElement);
allAvailableContents = self.availableContents->union
    (self.parent.allAvailableContents)
```

Collaboration

[1] All Classifiers and Associations of the ClassifierRoles and AssociationRoles in the Collaboration must be included in the namespace owning the Collaboration.

```
self.allContents->forAll ( e |
    (e.oclIsKindOf (ClassifierRole) implies
        self.namespace.allContents->includes (
            e.oclAsType(ClassifierRole).base) )
    and
    (e.oclIsKindOf (AssociationRole) implies
        self.namespace.allContents->includes (
            e.oclAsType(AssociationRole).base) ))
```

[2] All the constraining ModelElements must be included in the namespace owning the Collaboration.

```
self.constrainingElement->forAll ( ce |
    self.namespace.allContents->includes (ce) )
```

[3] If a ClassifierRole or an AssociationRole does not have a name then it should be the only one with a particular base.

```
self.allContents->forAll ( p |
    (p.oclIsKindOf (ClassifierRole) implies
        p.name = '' implies
            self.allContents->forAll ( q |
                q.oclIsKindOf(ClassifierRole) implies
                    (p.oclAsType(ClassifierRole).base =
                        q.oclAsType(ClassifierRole).base implies
                            p = q) ) )
    and
    (p.oclIsKindOf (AssociationRole) implies
        p.name = '' implies
            self.allContents->forAll ( q |
                q.oclIsKindOf(AssociationRole) implies
                    (p.oclAsType(AssociationRole).base =
```

```
                                    q.oclAsType(AssociationRole).base implies
                                        p = q) ) )
        )
```

[4] A Collaboration may only contain ClassifierRoles and AssociationRoles, and the Generalizations and the Constraints between them.

```
self.allContents->forAll ( p |
    p.oclIsKindOf (ClassifierRole) or
    p.oclIsKindOf (AssociationRole) or
    p.oclIsKindOf (Generalization) or
    p.oclIsKindOf (Constraint) )
```

[5] A role with the same name as one of the roles in a parent of the Collaboration must be a child (a specialization) of that role.

```
self.contents->forAll ( c |
    self.parent.allContents->forall ( p |
        c.name = p.name implies c.allParents->include (p) ))
```

Additional operations

[1] The operation *allContents* results in the set of all ModelElements contained in the Collaboration together with those contained in the parents except those that have been specialized.

```
allContents : Set(ModelElement);
allContents = self.contents->union (
    self.parent.allContents->reject ( e |
        self.contents.name->include (e.name) ))
```

Interaction

[1] All Signals being sent must be included in the namespace owning the Collaboration in which the Interaction is defined.

```
self.message->forAll ( m |
    m.action.oclIsKindOf(SendAction) implies
        self.context.namespace.allContents->includes (
            m.action->oclAsType (SendAction).signal) )
```

Message

[1] The sender and the receiver must participate in the Collaboration which defines the context of the Interaction.

```
self.interaction.context.ownedElement->includes (self.sender)
and
self.interaction.context.ownedElement->includes (self.receiver)
```

[2] The predecessors and the activator must be contained in the same Interaction.

```
self.predecessor->forAll ( p | p.interaction = self.interaction )
and
self.activator->forAll ( a | a.interaction = self.interaction )
```

[3] The predecessors must have the same activator as the Message.

```
self.allPredecessors->forAll ( p | p.activator = self.activator )
```

[4] A Message cannot be the predecessor of itself.

```
not self.allPredecessors->includes (self)
```

[5] The communicationLink of the Message must be an AssociationRole in the context of the Message's Interaction

```
self.interaction.context.ownedElement->includes (
    self.communicationConnection)
```

[6] The sender and the receiver roles must be connected by the AssociationRole which acts as the communication connection.

```
self.communicationConnection->size > 0 implies
    self.communicationConnection.connection->exists (ar |
        ar.type = self.sender)
    and
    self.communicationConnection.connection->exists (ar |
        ar.type = self.receiver)
```

Additional operations

[1] The operation allPredecessors results in the set of all Messages that precede the current one.

```
allPredecessors : Set(Message);
allPredecessors = self.predecessor->union
    (self.predecessor.allPredecessors)
```

2.10.4 Semantics

This section provides a description of the semantics of the elements in the Collaborations package. It is divided into two parts: Collaboration and Interaction.

Collaboration

In the following text the term instance of a collaboration denotes the set of instances that together participate in and perform one specific collaboration.

The purpose of a collaboration is to specify how an operation or a classifier, like a use case, is realized by a set of classifiers and associations. Together, the classifiers and their associations participating in the collaboration meet the requirements of the realized operation or classifier. The collaboration defines a context in which the behavior of the realized element can be

specified in terms of interactions between the participants of the collaboration. Thus, while a model describes a whole system, a collaboration is a slice, or a projection, of that model. A collaboration defines a usage of a subset of the model's contents.

A collaboration may be presented at two different levels: specification level or instance level. A diagram presenting the collaboration at the specification level will show classifier roles and association roles, while a diagram at the instance level will show instances and links conforming to the roles in the collaboration.

In a collaboration it is specified what properties instances must have to be able to take part in the collaboration, i.e. each participant specifies the required set of features a conforming instance must have. Furthermore, the collaboration also states what associations must exist between the participants, as well as what classifiers a participant, like a subsystem, must contain. Neither all features nor all contents of the participating classifiers, and not all associations between these classifiers are always required in a particular collaboration. Because of this, a collaboration is not actually defined in terms of classifiers, but classifier roles. Thus, while a classifier is a complete description of instances, a classifier role is a description of the features required in a particular collaboration, i.e. a classifier role is a projection of, or a view of, a classifier. The classifier so represented is referred to as the base classifier of that particular classifier role. In fact, since an instance may originate from several classifiers (multiple classification), a classifier role may have several base classifiers. Several classifier roles may have the same base classifier, even in the same collaboration, but their features and contained elements may be different subsets of the features and contained elements of the classifier, respectively. These classifier roles then specify different roles played by (usually different) instances of the same classifier. When the collaboration represents a classifier, its base classifiers can be classifiers of any kind, like classes or subsystems, while in a collaboration specifying the realization of an operation, the base classifiers are the operation's parameter types together with the attribute types and contained classifiers of the classifier owning the operation.

In a collaboration the association roles define what associations are needed between the classifiers in this context. Each association role represents the usage of an association in the collaboration, and it is defined between the classifier roles that represent the associated classifiers. The represented association is called the base association of the association role. As the association roles specifies a particular usage of an association in a specific collaboration, all constraints expressed by the association ends are not necessarily required to be fulfilled in the specified usage. The multiplicity of the association end may be reduced in the collaboration, i.e. the upper and the lower bounds of the association end roles may be lower than those of the corresponding base association end, as it might be that only a subset of the associated instances participate in the collaboration instance. Similarly, an association may be traversed in some, but perhaps not all, of the allowed directions in the specific collaboration, i.e. the isNavigable property of an association end role may be false even if that property of the base association end is true. (However, the opposite is not true, i.e. an association may not be used for traversal in a direction which is not allowed according to the isNavigable properties of the association ends.) The changeability and ordering of an association end may be strengthened in an association-end role, i.e. in a particular usage the end is used in a more restricted way than is defined by the association. Furthermore, if an association has a collection of qualifiers (see the Core), some of them may be used in a specific collaboration. An association end role may therefore include a subset of the qualifiers defined by the corresponding association end of the base association.

An instance participating in a collaboration instance plays a specific role, i.e. conforms to a classifier role, in the collaboration. The number of instances that should play one specific role in one instance of a collaboration is specified by the classifier role's multiplicity. Different instances may play the same role but in different instances of the collaboration. Since all these instances play the same role, they must all conform to the classifier role specifying the role. Thus, they are often instances of the base classifier of the classifier role, or one of its descendants. However, since the only requirement on conforming instances is that they must have attribute values corresponding to the attributes specified by the classifier role, and must participate in links corresponding to the association roles connected to the classifier role, they may be instances of any classifier meeting this requirement. The instances may, of course, have more attribute values than required by the classifier role, which for example would be the case if they originate from a classifier being a child of the base classifier. Moreover, a conforming instance may also have more attribute values than required if it originates from multiple classifiers. Finally, one instance may play different roles in different instances of one collaboration. In fact, the instance may play multiple roles in the same instance of a collaboration.

Collaborations may have generalization relationships to other collaborations, which means that one collaboration specifies a specialization of another collaboration's task. This implies that the child collaboration not only contains all the roles of the parent collaboration but may also contain new roles; the former roles may possibly be specialized with new features (as classifier roles are also generalizable elements). In this way it is possible to specialize a collaboration both by adding new features to existing roles and by adding new roles. Note that the base classifiers of the specialized roles are not necessarily specializations of the base classifiers of the parent's roles; it is enough that they contain all the required features.

How the instances conforming to the roles of a collaboration should interact to jointly perform the behavior of the realized classifier is specified with a set of interactions (see below). The collaboration thus specifies the context in which these interactions are performed. If the collaboration represents an operation, the context includes things like parameters, attributes and classifiers contained in the classifier owning the operation. The interactions then specify how the arguments, the attribute values, the instances etc. will cooperate to perform the behavior specified by the operation. If the collaboration is a specialization of another collaboration, all communications specified by the parent collaboration are also included in the child, as the child collaboration includes all the roles of the parent. However, new messages may be inserted into these sequences of communication, since the child may include specializations of the parent's roles as well as new roles. The child may of course also include completely new interactions that do not exist in the parent.

Two or more collaborations may be composed in order to refine a superordinate collaboration. For example, when refining a superordinate use case into a set of subordinate use cases, the collaborations specifying each of the subordinate use cases may be composed into one collaboration, which will be a (simple) refinement of the superordinate collaboration. The composition is done by observing that at least one instance must participate in both sets of collaborating instances. This instance has to conform to one classifier role in each collaboration. In the composite collaboration these two classifier roles are merged into a new one, which will contain all features included in either of the two original classifier roles. The new classifier role will, of course, be able to fulfill the requirements of both of the previous collaborations, so the instance participating in both of the two sets of collaborating instances will conform to the new classifier role.

A collaboration may be a specification of a template. There will not be any instances of such a template collaboration, but it can be used for generating ordinary collaborations, which may be instantiated. Template collaborations may have parameters that act like placeholders in the template. Usually, these parameters would be used as base classifiers and associations, but other kinds of model elements can also be defined as parameters in the collaboration, like operation or signal. In a collaboration generated from the template these parameters are refined by other model elements that make the collaboration instantiable.

Moreover, a collaboration may also contain a set of constraining model elements, like constraints and generalizations perhaps together with some extra classifiers. These constraining model elements do not participate in the collaboration themselves, but are used for expressing the extra constraints on the participating elements in the collaboration that cannot be covered by the participating roles themselves. For example, in a template it might be required that the base classifiers of two roles must have a common ancestor, or one role must be a subclass of another one. These kinds of requirements cannot be expressed with association roles, as the association roles express the required links between participating instances. An extra set of model elements may therefore be included in the collaboration.

Interaction

The purpose of an interaction is to specify the communication between a set of interacting instances performing a specific task. An interaction is defined within a collaboration, i.e. the collaboration defines the context in which the interaction takes place. The instances performing the communication specified by the interaction conform to the classifier roles of the collaboration.

An interaction specifies the sending of a set of stimuli. These are partially ordered based on which execution thread they belong to. Within each thread the stimuli are sent in a sequential order while stimuli of different threads may be sent in parallel or in an arbitrary order.

A message is a specification of a communication. It specifies the roles of the sender and the receiver instances, as well as which association role specifies the communication link. The message is connected to an action, which specifies the statement that, when executed, causes the communication specified by the message to take place. If the action is a call action or a raise action, the signal to be sent or the operation to be invoked in the communication is stated by the action. The action also contains the argument expressions that, when executed, will determine the actual arguments being transmitted in the communication. Moreover, any conditions or iterations of the communication are also specified by the action. Apart from raise action and call action, the action connected to a message can also be of other kinds, like create action and destroy action. In these cases, the communication will not raise a signal or invoke an operation, but cause a new instance to be created or an already existing instance to be destroyed. In the case of a create action, the receiver specified by the message is the role to be played by the instance which is created when the action is perfromed.

The stimuli being sent when an action is executed conforms to a message, implying that the sender and receiver instances of the stimuli are in conformance with the sender and the receiver roles specified by the message. Furthermore, the action dispatching the stimulus is the same as the action attached to the message. If the action connected to the message is a create action or destroy action, the receiver role of the message specify the role to be played by the instance, or was played by the instance, respectively.

The interaction specifies the activator and predecessors of each message. The activator is the message that invoked the procedure of which in turn invokes the current message. Every message except the initial message of an interaction thus has an activator. The predecessors are the set of messages that must be completed before the current message may be executed. The first message in a procedure of course has no predecessors. If a message has more than one predecessor, it represents the joining of two threads of control. If a message has more than one successor (the inverse of predecessor), it indicates a fork of control into multiple threads. Thus, the predecessors relationship imposes a partial ordering on the messages within a procedure, whereas the activator relationship imposes a tree on the activation of operations. Messages may be executed concurrently subject to the sequential constraints imposed by the predecessors and activator relationship.

2.10.5 Notes

Pattern is a synonym for a template collaboration that describes the structure of a design pattern. Design patterns involve many nonstructural aspects, such as heuristics for their use and lists of advantages and disadvantages. Such aspects are not modeled by UML and may be represented as text or tables.

2 UML Semantics

2.11 Use Cases

2.11.1 Overview

The Use Cases package is a subpackage of the Behavioral Elements package. It specifies the concepts used for definition of the functionality of an entity like a system. The package uses constructs defined in the Foundation package of UML as well as in the Common Behavior package.

The elements in the Use Cases package are primarily used to define the behavior of an entity, like a system or a subsystem, without specifying its internal structure. The key elements in this package are UseCase and Actor. Instances of use cases and instances of actors interact when the services of the entity are used. How a use case is realized in terms of cooperating objects, defined by classes inside the entity, can be specified with a Collaboration. A use case of an entity may be refined to a set of use cases of the elements contained in the entity. How these subordinate use cases interact can also be expressed in a Collaboration. The specification of the functionality of the system itself is usually expressed in a separate use-case model, i.e. a Model stereotyped «useCaseModel». The use cases and actors in the use-case model are equivalent to those of the top-level package.

The following sections describe the abstract syntax, well-formedness rules and semantics of the Use Cases package.

2.11.2 Abstract Syntax

The abstract syntax for the Use Cases package is expressed in graphic notation in Figure 2-23 on page 2-114.

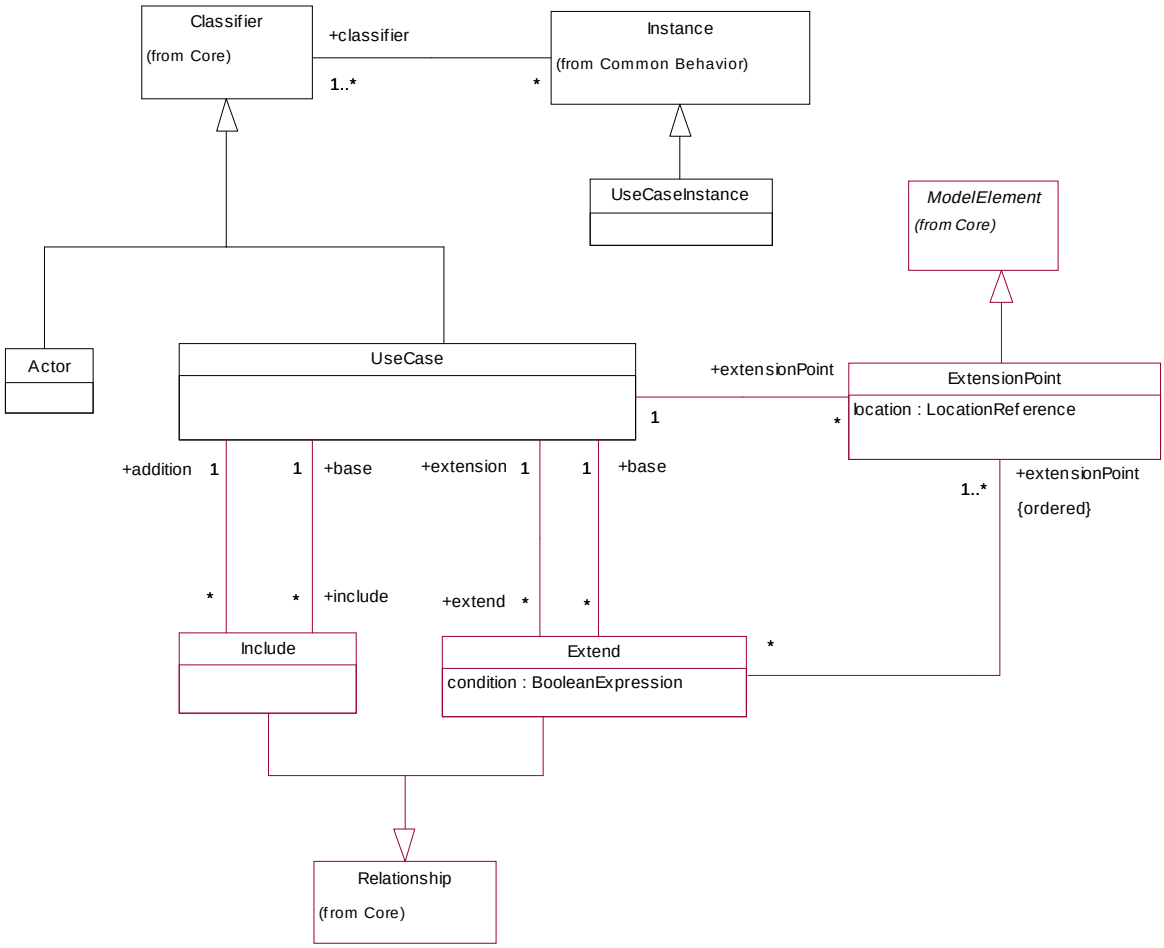


Figure 2-23 Use Cases

The following metaclasses are contained in the Use Cases package.

Actor

An *actor* defines a coherent set of roles that users of an entity can play when interacting with the entity. An actor has one role for each use case it communicates with.

In the metamodel Actor is a subclass of Classifier. An Actor has a Name and may communicate with a set of UseCases, and, at realization level, with Classifiers taking part in the realization of these UseCases. An Actor may also have a set of Interfaces, each describing how other elements may communicate with the Actor.

An Actor may have generalization relationships to other Actors. This means that the child Actor will be able to play the same roles as the parent Actor, i.e. communicate with the same set of UseCases, as the parent Actor.

Extend

An *extend* relationship defines that instances of a use case may be extended with some additional behavior defined in an extending use case.

In the metamodel an Extend relationship is a directed relationship implying that a UseCaseInstance of the base UseCase may be extended with the structure and behavior defined in the extending UseCase. The relationship consists of a condition, which must be fulfilled if the extension is to take place, and a sequence of references to extension points in the base UseCase where the additional behavior fragments are to be inserted.

Attributes

condition	an expression specifying the condition which must be fulfilled if the extension is to take place

Associations

base	the UseCase to be extended
extension	the UseCase specifying the extending behavior
location	a sequence of extension-points in the base UseCase specifying where the additions are to be inserted

ExtensionPoint

An extension point references one or a collection of locations in a use case where the use case may be extended.

In the metamodel an ExtensionPoint has a name and one or a collection of descriptions of locations in the behavior of the owning use case, where a piece of behavior may be inserted into the owning use case.

Attributes

location	a reference to one location or a collection of locations where an extension to the behavior of the use case may be inserted

Include

An include relationship defines that a use case includes the behavior defined in another use case.

In the metamodel an Include relationship is a directed relationship between two UseCases implying that the behavior in the addition UseCase is inserted into the behavior of the base UseCase. The base UseCase may only depend on the result of performing the behavior defined in the addition UseCase, but not on the structure, i.e. on the existence of specific attributes and operations, of the addition UseCase.

Associations

addition	the UseCase specifying the additional behavior
base	the UseCase which is to include the addition

UseCase

The use case construct is used to define the behavior of a system or other semantic entity without revealing the entity's internal structure. Each use case specifies a sequence of actions, including variants, that the entity can perform, interacting with actors of the entity.

In the metamodel UseCase is a subclass of Classifier, containing a set of Operations and Attributes specifying the sequences of actions performed by an instance of the UseCase. The actions include changes of the state and communications with the environment of the UseCase.

There may be Associations between UseCases and the Actors of the UseCases. Such an Association states that an instance of the UseCase and a user playing one of the roles of the Actor communicate. UseCases may be related to other UseCases by Extend, Include, and Generalization relationships. An Include relationship means that a UseCase includes the behavior described in another UseCase, while an Extend relationship implies that a UseCase may extend the behavior described in another UseCase, ruled by a condition. Generalization between UseCases means that the child is a more specific form of the parent. The child inherits all Features and Associations of the parent, and may add new Features and Associations.

The realization of a UseCase may be specified by a set of Collaborations, i.e. the Collaborations define how Instances in the system interact to perform the sequences of the UseCase.

Associations

extend	a collection of Extend relationships to UseCases that the UseCase extends
extensionPoint	defines a collection of ExtensionPoints where the UseCase may be extended.
include	a collection of Include relationships to UseCases that the UseCase includes

UseCaseInstance

A use case instance is the performance of a sequence of actions specified in a use case.

In the metamodel UseCaseInstance is a subclass of Instance. Each method performed by a UseCaseInstance is performed as an atomic transaction, i.e. it is not interrupted by any other UseCaseInstance.

An explicitly described UseCaseInstance is called a scenario.

2.11.3 Well-FormednessRules

The following well-formedness rules apply to the Use Cases package.

Actor

[1] Actors can only have Associations to UseCases, Subsystems, and Classes and these Associations
are binary.

```
self.associations->forAll(a |
    a.connection->size = 2 and
    a.allConnections->exists(r | r.type.oclIsKindOf(Actor)) and
    a.allConnections->exists(r |
        r.type.oclIsKindOf(UseCase) or
        r.type.oclIsKindOf(Subsystem) or
        r.type.oclIsKindOf(Class)))
```

[2] Actors cannot contain any Classifiers.

```
self.contents->isEmpty
```

Extend

[1] The referenced ExtensionPoints must be included in set of ExtensionPoint in the target UseCase.

```
self.base.allExtensionPoints -> includesAll (self.location)
```

ExtensionPoint

[1] The name must not be the empty string

not self.name = ‘’

Include

No extra well-formedness rules.

UseCase

[1] UseCases can only have binary Associations.

```
self.associations->forAll(a | a.connection->size = 2)
```

[2] UseCases can not have Associations to UseCases specifying the same entity.

```
self.associations->forAll(a |
    a.allConnections->forAll(s, o|
        s.type.specificationPath->isEmpty and
        o.type.specificationPath->isEmpty
    or
    (not s.type.specificationPath->includesAll(
        o.type.specificationPath) and
        not o.type.specificationPath->includesAll(
            s.type.specificationPath))
    ))
```

[3] A UseCase cannot contain any Classifiers.

```
self.contents->isEmpty
```

[4] For each Operation in an offered Interface the UseCase must have a matching Operation.

```
self.specification.allOperations->forAll (interOp |
    self.allOperations->exists ( op | op.hasSameSignature (interOp)
) )
```

[5] The names of the ExtensionPoints must be unique within the UseCase.

```
self.allExtensionPoints -> forAll (x, y |
    x.name = y.name implies x = y )
```

Additional operations

[1] The operation specificationPath results in a set containing all surrounding Namespaces that are not instances of Package.

```
specificationPath : Set(Namespace)
specificationPath = self.allSurroundingNamespaces->select(n |
    n.oclIsKindOf(Subsystem) or n.oclIsKindOf(Class))
```

[2] The operation allExtensionPoints results in a set containing all ExtensionPoints of the UseCase.

```
allExtensionPoints : Set(ExtensionPoint)
allExtensionPoints = self.allSupertypes.extensionPoint -> union (
                        self.extensionPoint)
```

UseCaseInstance

[1] The Classifier of a UseCaseInstance must be a UseCase.

```
self.classifier->forAll ( c | c.oclIsKindOf (UseCase) )
```

2.11.4 Semantics

This section provides a description of the semantics of the elements in the Use Cases package, and its relationship to other elements in the Behavioral Elements package.

Actor

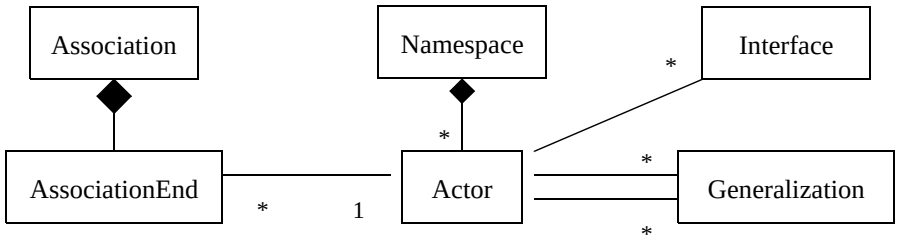


Figure 2-24 Actor Illustration

Actors model parties outside an entity, such as a system, a subsystem, or a class, which interact with the entity. Each actor defines a coherent set of roles users of the entity can play when interacting with the entity. Every time a specific user interacts with the entity, it is playing one such role. An instance of an actor is a specific user interacting with the entity. Any instance that conforms to an actor can act as an instance of the actor. If the entity is a system the actors represent both human users and other systems. Some of the actors of a lower level subsystem or a class may coincide with actors of the system, while others appear inside the system. The roles defined by the latter kind of actors are played by instances of classifiers in other packages or subsystems; in the latter case the classifier may belong to either the specification part or the contents part of the subsystem.

Since an actor is outside the entity, its internal structure is not defined but only its external view as seen from the entity. Actor instances communicate with the entity by sending and receiving message instances to and from use case instances and, at realization level, to and from objects. This is expressed by associations between the actor and the use case or the class. Furthermore, interfaces can be connected to an actor, defining how other elements may interact with the actor.

Two or more actors may have commonalities, i.e. communicate with the same set of use cases in the same way. The commonality is expressed with generalizations to another (possibly abstract) actor, which models the common role(s). An instance of a child can always be used where an instance of the parent is expected.

UseCase

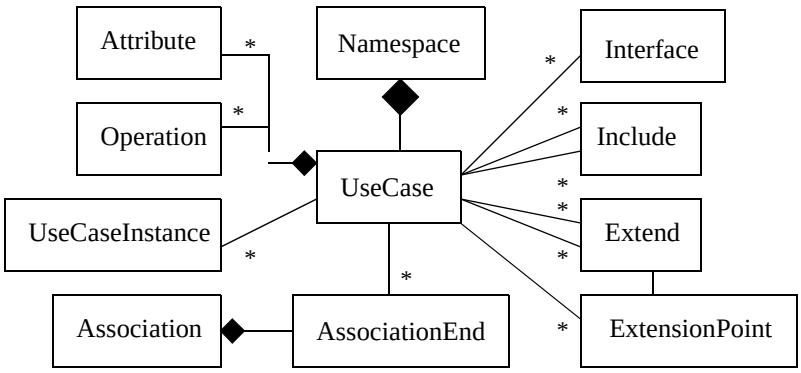


Figure 2-25 UseCase Illustration

In the following text the term entity is used when referring to a system, a subsystem, or a class and the terms model element and element denote a subsystem or a class.

The purpose of a use case is to define a piece of behavior of an entity without revealing the internal structure of the entity. The entity specified in this way may be a system or any model element that contains behavior, like a subsystem or a class, in a model of a system. Each use case specifies a service the entity provides to its users, i.e. a specific way of using the entity. The service, which is initiated by a user, is a complete sequence. This implies that after its performance the entity will in general resume a state in which the sequence can be initiated again. A use case describes the interactions between the users and the entity as well as the responses performed by the entity, as these responses are perceived from the outside of the entity. A use case also includes possible variants of this sequence, e.g. alternative sequences, exceptional behavior, error handling etc. The complete set of use cases specifies all different ways to use the entity, i.e. all behavior of the entity is expressed by its use cases. These use cases can be grouped into packages for convenience.

From a pragmatic point of view, use cases can be used both for specification of the (external) requirements on an entity and for specification of the functionality offered by an (already realized) entity. Moreover, the use cases also indirectly state the requirements the specified entity poses on its users, i.e. how they should interact so the entity will be able to perform its services.

Since users of use cases always are external to the specified entity, they are represented by actors of the entity. Thus, if the specified entity is a system or a subsystem at the topmost level, the users of its use cases are modeled by the actors of the system. Those actors of a lower level subsystem or a class that are internal to the system are often not explicitly defined. Instead, the use cases relate directly to model elements conforming to these implicit actors, i.e. whose instances play the roles of these actors in interaction with the use cases. These model elements are contained in other packages or subsystems, where in the subsystem case they may be contained in the specification part or the contents part. The distinction between actor and conforming element like this is often neglected; thus, they are both referred to by the term actor.

There may be associations between use cases and actors, meaning that the instances of the use case and the actor communicates with each other. One actor may communicate with several use cases of an entity, i.e. the actor may request several services of the entity, and one use case communicates with one or several actors when providing its service. Note that two use cases specifying the same entity cannot communicate with each other since each of them individually describes a complete usage of the entity. Moreover, use cases always use signals when communicating with actors outside the system, while they may use other communication semantics when communicating with elements inside the system.

The interaction between actors and use cases can be defined with interfaces. An interface of a use case defines a subset of the entire interaction defined in the use case. Different interfaces offered by the same use case need not be disjoint.

A use case can be described in plain text, using operations and methods, in activity diagrams, by a state machine, or by other behavior description techniques, such as pre- and post conditions. The interaction between a use case and its actors can also be presented in collaboration diagrams for specification of the interactions between the entity containing the use case and the entity's environment.

A use-case instance is a performance of a use case, initiated by a message instance from an instance of an actor. As a response the use-case instance performs a sequence of actions as specified by the use case, like communicating with actor instances, not necessarily only the initiating one. The actor instances may send new message instances to the use-case instance and the interaction continues until the instance has responded to all input and does not expect any more input, when it ends. Each method performed by a use-case instance is performed as an atomic transaction, i.e. it is not interrupted by any other use-case instance.

In the case where subsystems are used to model the system's containment hierarchy, the system can be specified with use cases at all levels, as use cases can be used to specify subsystems and classes. A use case specifying one model element is then refined into a set of smaller use cases, each specifying a service of a model element contained in the first one. The use case of the whole may be referred to as superordinate to its refining use cases, which, correspondingly, may be called subordinate in relation to the first one. The functionality specified by each superordinate use case is completely traceable to its subordinate use cases. Note, though, that the structure of the container element is not revealed by the use cases, since they only specify the functionality offered by the element. The subordinate use cases of a specific superordinate use case cooperate to perform the superordinate one. Their cooperation is specified by collaborations and may be presented in collaboration diagrams. A specific subordinate use case may appear in several collaborations, i.e. play a role in the performances of several superordinate use cases. In each such collaboration, other roles specify the cooperation with this specific subordinate use case. These roles are the roles played by the actors of that subordinate use case. Some of these actors may be the actors of the superordinate use case, as each actor of a superordinate use case appears as an actor of at least one of the subordinate use cases. Furthermore, the interfaces of a superordinate use case are traceable to the interfaces of those subordinate use cases that communicate with actors that are also actors of the superordinate use case.

The environment of subordinate use cases is the model element containing the model elements specified by these use cases. Thus, from a bottom-up perspective, an interaction between subordinate use cases results in a superordinate use case, i.e. a use case of the container element.

Use cases of classes are mapped onto operations of the classes, since a service of a class in essence is the invocation of the operations of the class. Some use cases may consist of the application of only one operation, while others may involve a set of operations, usually in a well-defined sequence. One operation may be needed in several of the services of the class, and will therefore appear in several use cases of the class.

The realization of a use case depends on the kind of model element it specifies. For example, since the use cases of a class are specified by means of operations of the class, they are realized by the corresponding methods, while the use cases of a subsystem are realized by the elements contained in the subsystem. Since a subsystem does not have any behavior of its own, all services offered by a subsystem must be a composition of services offered by elements contained in the subsystem, i.e. eventually by classes. These elements will collaborate and jointly perform the behavior of the specified use case. One or a set of collaborations describes how the realization of a use case is made. Hence, collaborations are used for specification of both the refinement and the realization of a use case in terms of subordinate use cases.

The usage of use cases at all levels imply not only a uniform way of specification of functionality at all levels, but also a powerful technique for tracing requirements at the system package level down to operations of the classes. The propagation of the effect of modifying a single operation at the class level all the way up to the behavior of the system package is managed in the same way.

Commonalities between use cases can be expressed in two different ways: with generalization relationships or include relationships. A generalization relationship between use cases implies that the child use case contains all the attributes, sequences of behavior, and extension points defined in the parent use case, and participate in all relationships of the parent use case. The child use case may also define new behavior sequences, as well as add additional behavior into and specialize existing behavior of the inherited ones. One use case may have several parent use cases and one use case may be a parent to several other use cases.

An include relationship between two use cases means that the behavior defined in the target use case is included at one location in the sequence of behavior performed by an instance of the base use case. When a use-case instance reaches the location where the behavior of an another use case is to be included, it performs all the behavior described by the included use case and then continues according to its original use case. This means that although there may be several paths through the included use case due to e.g. conditional statements, all of them must end in such a way that the use-case instance can continue according to the original use case. One use case may be included in several other use cases and one use case may include several other use cases. The included use case may not be dependent on the base use case. In that sense the included use case represents encapsulated behavior which may easily be reused in several use cases. Moreover, the base use case may only be dependent on the results of performing the included behavior and not on structure, like Attributes and Associations, of the included use case.

An extend relationship defines that a use case may be extended with some additional behavior defined in another use case. One use case may extend several use cases and one use case may be extended by several use cases. The base use case may not be dependent of the addition of the extending use case. The extend relationship contains a condition and references a sequence of extension points in the target use case. The condition must be satisfied if the extension is to take place, and the references to the extension points define the locations in the base use case where the additions are to be made. Once an instance of a use case is to perform some behavior

referenced by an extension point of its use case, and the extension point is the first one in an extends relationship's sequence of references to extension points, the condition of the relationship is evaluated. If the condition is fulfilled, the sequence obeyed by the use-case instance is extended to include the sequence of the extending use case. The different parts of the extending use case are inserted at the locations defined by the sequence of extension points in the relationship -- one part at each referenced extension point. Note that the condition is only evaluated once: at the first referenced extension point, and if it is fulfilled all of the extending use case is inserted in the original sequence. An extension point may reference one location or a set of locations in the behavior defined by the use case. However, an extension point which references a set of locations may only be used as the first one in the sequence of extension points referenced by an extend relationship. The addition will be made at the location where the condition is fulfilled. All other extension points referenced by the extend relationship must define precisely where the additions are to be made, i.e. each of them must reference single locations and not collection of locations. The description of the location references by an extension point can be made in several different ways, like textual description of where in the behavior the addition should be made, pre-or post conditions, or using the name of a state in a state machine.

Note that the three kinds of relationships described above can only exist between use cases specifying the same entity. The reason for this is that the use cases of one entity specify the behavior of that entity alone, i.e. all use-case instances are performed entirely within that entity. If a use case would have a generalization, include, or extend relationship to a use case of another entity, the resulting use-case instances would involve both entities, resulting in a contradiction. However, generalization, include, and extend relationships can be defined from use cases specifying one entity to use cases of another one if the first entity has a generalization to the second one, since the contents of both entities are available in the first entity. However, the contents of the second entity must be at least protected, so they become available inside the child entity.

As a first step when developing a system, the dynamic requirements of the system as a whole can be expressed with use cases. The entity being specified is then the whole system, and the result is a separate model called a use-case model, i.e. a model with the stereotype «useCaseModel». Next, the realization of the requirements is expressed with a model containing a system package, probably a package hierarchy, and at the bottom a set of classes. If the system package, i.e. a package with the stereotype «topLevelPackage», is a subsystem, its abstract behavior is naturally the same as that of the system. Thus, if use cases are used for the specification part of the system package, these use cases are equivalent to those in the use-case model of the system, i.e. they express the same behavior but possibly slightly differently structured. In other words, all services specified by the use cases of a system package, and only those, define the services offered by the system. Furthermore, if several models are used for modeling the realization of a system, e.g. an analysis model and a design model, the set of use cases of all system packages and the use cases of the use-case model must be equivalent.

2.11.5 Notes

A pragmatic rule of use when defining use cases is that each use case should yield some kind of observable result of value to (at least) one of its actors. This ensures that the use cases are complete specifications and not just fragments.

2 UML Semantics

2.12 State Machines

2.12.1 Overview

The State Machine package is a subpackage of the Behavioral Elements package. It specifies a set of concepts that can be used for modeling discrete behavior through finite state-transition systems. These concepts are based on concepts defined in the Foundation package as well as concepts defined in the Common Behavior package. This enables integration with the other subpackages in Behavioral Elements.

The state machine formalism described in this section is an object-based variant of Harel statecharts. It incorporates several concepts similar to those defined in ROOMcharts, a variant of statechart defined in the ROOM modeling language. The major differences relative to classical Harel statecharts are described on page 152.

State machines can be used to specify behavior of various elements that are being modeled. For example, they can be used to model the behavior of individual entities (e.g., class instances) or to define the interactions (e.g., collaborations) between entities.

In addition, the state machine formalism provides the semantic foundation for activity graphs. This means that activity graphs are simply a special form of state machines.

The following sections describe the abstract syntax, well-formedness rules, and semantics of the State Machines package. Activity graphs are described in section 2.13.

2.12.2 Abstract Syntax

The abstract syntax for state machines is expressed graphically in figure 2-26, which covers all the basic concepts of state machine graphs such as states and transitions. Figure 2-27 describes the abstract syntax of events that can trigger state machine behavior.

The specifications of the concepts defined in these two diagrams are listed in alphabetical order following the figures.

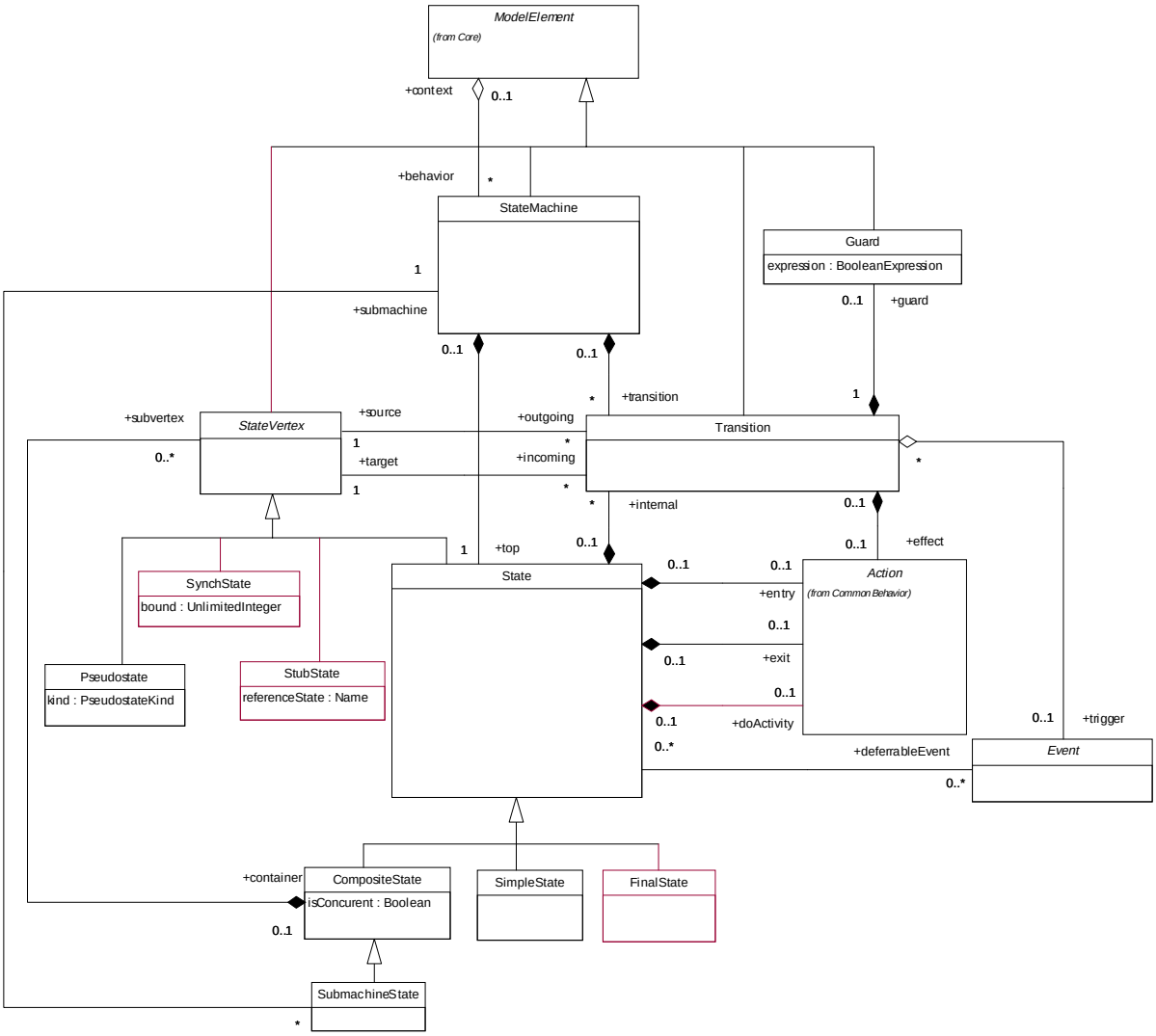


Figure 2-26 State Machines - Main

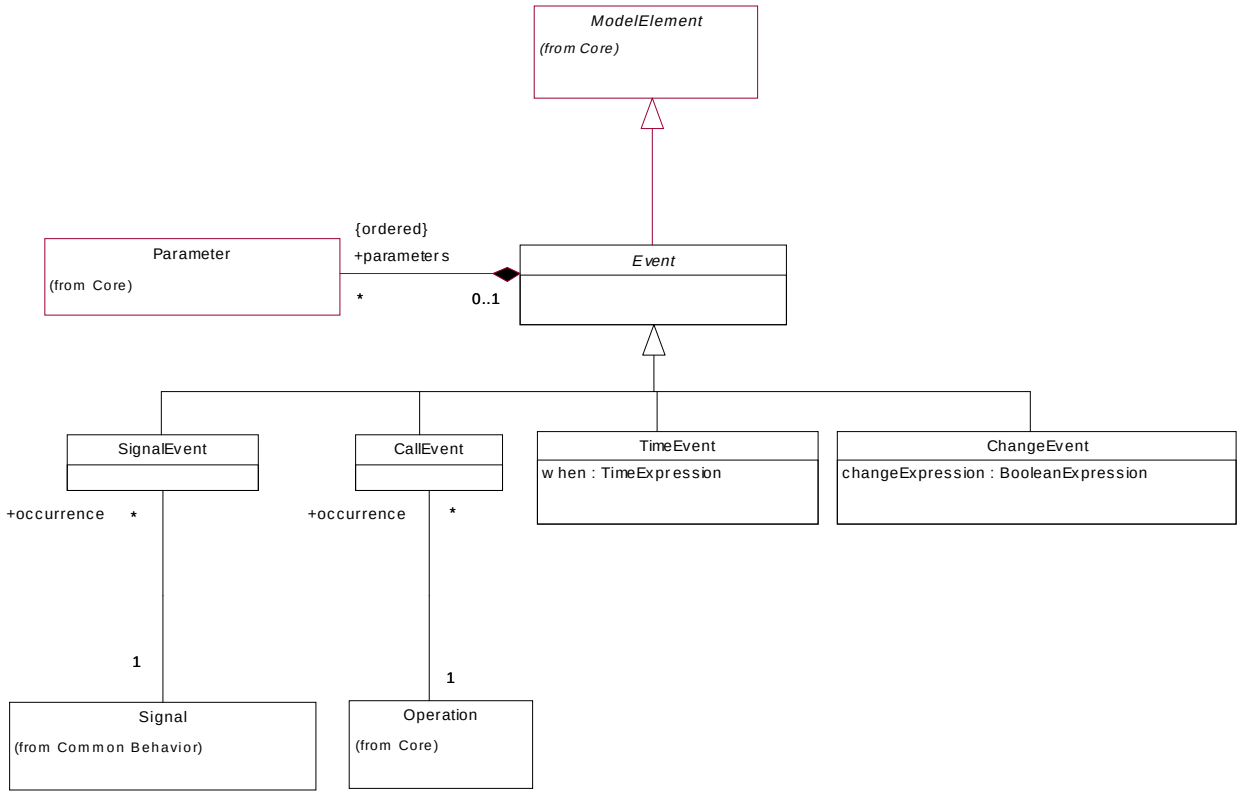


Figure 2-27 State Machines - Events

CallEvent

A call event represents the *reception* of a request to synchronously invoke a specific operation. (Note that a call event instance is distinct from the call action that caused it.) The expected result is the execution of a sequence of actions which characterize the operation behavior at a particular state.

Two special cases of CallEvent are the object creation event and the object destruction event.

Associations

operation	Designates the operation whose invocation raised the call event

Stereotypes

«create»	Create is a stereotyped call event denoting that the instance receiving that event has just been created. For state machines, it triggers the initial transition at the topmost level of the state machine (and is the only kind of trigger that may be applied to an initial transition).
«destroy»	Destroy is a stereotyped call event denoting that the instance receiving the event is being destroyed.

ChangeEvent

A change event models an event that occurs when an explicit boolean expression becomes true as a result of a change in value of one or more attributes or associations. A change event is raised implicitly and is *not* the result of some explicit change event action.

The change event should not be confused with a guard. A guard is only evaluated at the time an event is dispatched whereas, conceptually, the boolean expression associated with a change event is evaluated continuously until it becomes true. The event that is generated remains until it is consumed even if the boolean expression changes to false after that.

Attributes

changeExpression	The boolean expression that specifies the change event.

CompositeState

A composite state is a state that contains other state vertices (states, pseudostates, etc.). The association between the composite and the contained vertices is a composition association. Hence, a state vertex can be a part of at most one composite state.

Any state enclosed within a composite state is called a *substate* of that composite state. It is called a *direct substate* when it is not contained by any other state; otherwise it is referred to as a *transitively nested substate*.

CompositeState is a child of State.

Associations

subvertex	The set of state vertices that are owned by this composite state.

Attributes

isConcurrent	A boolean value that specifies the decomposition semantics. If this attribute is true, then the composite state is decomposed directly into two or more orthogonal conjunctive components called *regions* (usually associated with concurrent execution). If this attribute is false, then there are no direct orthogonal components in the composite.
isRegion	A derived boolean value that indicates whether a CompositeState is a substate of a concurrent state. If it is true, then this composite state is a direct substate of a concurrent state.

Event

An event is a specification of a type of observable occurrence. The occurrence that generates an event instance is assumed to take place at an instant in time with no duration.

Strictly speaking, the term “event” is used to refer to the type and not to an instance of the type. However, on occasion, where the meaning is clear from the context, the term is also used to refer to an event instance.

Event is a child of ModelElement.

Associations

parameters	The list of parameters defined by the event.

FinalState

A special kind of state signifying that the enclosing composite state is completed. If the enclosing state is the top state, then it means that the entire state machine has completed.

A final state cannot have any outgoing transitions.

FinalState is a child of State.

Guard

A guard is a boolean expression that is attached to a transition as a fine-grained control over its firing. The guard is evaluated when an event instance is dispatched by the state machine. If the guard is true at that time, the transition is enabled, otherwise, it is disabled.

Guards should be pure expressions without side effects. Guard expressions with side effects are ill formed.

Guard is a child of ModelElement.

Attributes

expression	The boolean expression that specifies the guard.

PseudoState

A pseudostate is an abstraction that encompasses different types of transient vertices in the state machine graph. They are used, typically, to connect multiple transitions into more complex state transitions paths. For example, by combining a transition entering a fork pseudostate with a set of transitions exiting the fork pseudostate, we get a complex transition that leads to a set of concurrent target states.

The following pseudostate kinds are defined:

- An *initial* pseudostate represents a default vertex that is the source for a single transition to the *default* state of a composite state. There can be at most one initial vertex in a composite state.
- *deepHistory* is used as a shorthand notation that represents the most recent active configuration of the composite state that directly contains this pseudostate; that is, the state configuration that was active when the composite state was last exited. A composite state can have at most one deep history vertex. A transition may originate from the history connector to the *default* deep history state. This transition is taken in case the composite state had never been active before.
- *shallowHistory* is a shorthand notation that represents the most recent active substate of its containing state (but *not* the substates of that substate). A composite state can have at most one shallow history vertex. A transition coming into the shallow history vertex is equivalent to a transition coming into the most recent active substate of a state. A transition may originate from the history connector to the *initial* shallow history state. This transition is taken in case the composite state had never been active before.
- *join* vertices serve to merge several transitions emanating from source vertices in different orthogonal regions. The transitions entering a join vertex cannot have guards.
- *fork* vertices serve to split an incoming transition into two or more transitions terminating on orthogonal target vertices. The segments outgoing from a fork vertex must not have guards.
- *junction* vertices are semantic-free vertices that are used to chain together multiple transitions. They are used to construct complex transition paths between states. For example, a junction can be used to converge multiple incoming transitions into a single outgoing transition representing a shared transition path (this is known as an *merge*). Conversely, they can be used to split an incoming transition into multiple outgoing transition segments with different guard conditions. This realizes a *conditional branch*. (In the latter case, outgoing transitions whose guard conditions evaluate to false are disabled. A predefined guard denoted “else” may be defined for at most one outgoing transition. This transition is enabled if all the guards labeling the other transitions are false.)

PseudoState is a child of StateVertex.

kind	Determines the precise type of the PseudoState and can be one of: *initial, deepHistory, shallowHistory, join, fork, junction.*

SignalEvent

A signal event represents the *reception* of a particular (asynchronous) signal. A signal event instance should not be confused with the action (e.g., send action) that generated it.

SignalEvent is a child of Event.

Associations

signal	The specific signal that is associated with this event.

SimpleState

A SimpleState is a state that does not have substates.

It is a child of State.

State

A state is an abstract metaclass that models a situation during which some (usually implicit) invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some activity (i.e., the model element under consideration enters the state when the activity commences and leaves it as soon as the activity is completed).

State is a child of StateVertex.

Associations

deferrableEvent	A list of events that are candidates to be retained by the state machine if they trigger no transitions out of the state (not consumed). A deferred event is retained until the statemachine reaches a state configuration where it is no longer deferred.
entry	An optional action that is executed whenever this state is entered regardless of the transition taken to reach the state. If defined, entry actions are always executed to completion prior to any internal activity or transitions performed within the state.
exit	An optional action that is executed whenever this state is exited regardless of which transition was taken out of the state. If defined, entry actions are always executed to completion only after all internal activities and transition actions have completed execution.
doActivity	An optional activity that is executed while being in the state. The execution starts when this state is entered, and stops either by itself, or when the state is exited, whichever comes first.

internalTransition	A set of transitions that, if triggered, occur without exiting or entering this state. Thus, they do not cause a state change. This means that the entry or exit condition of the State will not be invoked. These transitions can be taken even if the state machine is in one or more regions nested within this state.

StateMachine

A state machine is a specification that describes all possible behaviors of some dynamic model element. Behavior is modeled as a traversal of a graph of state nodes interconnected by one or more joined transition arcs that are triggered by the dispatching of series of event instances. During this traversal, the state machine executes a series of actions associated with various elements of the state machine.

StateMachine has a composition relationship to State, which represents the top-level state, and a set of transitions. This means that a state machine owns its transitions and its top state. All remaining states are transitively owned through the state containment hierarchy rooted in the top state. The association to ModelElement provides the context of the state machine. A common case of the context relation is where a state machine is used to specify the lifecycle of a classifier.

Associations

context	An association to the model element that whose behavior is specified by this state machine. A model element may have more than one state machine (although one is sufficient for most purposes). Each state machine is owned by exactly one model element.
top	Designates the top-level state that is the root of the state containment hierarchy. There is exactly one state in every state machine that is the top state.
transition	The set of transitions owned by the state machine. Note that internal transitions are owned by their containing states and not by the state machine.

StateVertex

A StateVertex is an abstraction of a node in a statechart graph. In general, it can be the source or destination of any number of transitions.

StateVertex is a child of ModelElement.

Associations

outgoing	Specifies the transitions departing from the vertex.

incoming	Specifies the transitions entering the vertex.
container	The composite state that contains this state vertex.

StubState

A stub state can appear within a submachine state and represents an actual subvertex contained within the referenced state machine. It can serve as a source or destination of transitions that connect a state vertex in the containing state machine with a subvertex in the referenced state machine.

StubState is a child of State.

Associations

referenceState	Designates the referenced state as a pathname (a name formed by the concatenation of the name of a state and the successive names of all states that contain it, up to the top state).

SubmachineState

A submachine state is a syntactical convenience that facilitates reuse and modularity. It is a shorthand that implies a macro-like expansion by another state machine and is semantically equivalent to a composite state. The state machine that is inserted is called the *referenced* state machine while the state machine that contains the submachine state is called the *containing* state machine. The same state machine may be referenced more than once in the context of a single containing state machine. In effect, a submachine state represents a “call” to a state machine “subroutine” with one or more entry and exit points.

The entry and exit points are specified by stub states.

SubmachineState is a child of State.

Associations

submachine	The state machine that is to be substituted in place of the submachine state.

SynchState

A synch state is a vertex used for synchronizing the concurrent regions of a state machine. It is different from a state in the sense that it is not mapped to a boolean value (active, not active), but an integer. A synch sate is used in conjunction with forks and joins to insure that one region leaves a particular state or states before another region can enter a particular state or states.

SynchState is a child of StateVertex.

Attributes

bound	A positive integer or the value “unlimited” specifying the maximal count of the synchState. The count is the difference between the number of times the incoming and outgoing transitions of the synch state are fired

TimeEvent

A TimeEvent models the expiration of a specific deadline. Note that the time of occurrence of a time event instance (i.e., the expiration of the deadline) is the same as the time of its reception. However, it is important to note that there may be a variable delay between the time of reception and the time of dispatching (e.g., due to queueing delays).

The expression specifying the deadline may be relative or absolute. If the time expression is relative and no explicit starting time is defined, then it is relative to the time of entry into the source state of the transition triggered by the event. In the latter case, the time event instance is generated only if the state machine is still in that state when the deadline expires.

Attributes

when	Specifies the corresponding time deadline

Transition

A transition is a directed relationship between a source state vertex and a target state vertex. It may be part of a compound transition, which takes the state machine from one state configuration to another, representing the complete response of the state machine to a particular event instance.

Transition is a child of ModelElement.

Associations

trigger	Specifies the event that fires the transition. There can be at most one trigger per transition
guard	A boolean predicate that provides a fine-grained control over the firing of the transition. It must be true for the transition to be fired. It is evaluated at the time the event is dispatched. There can be at most one guard per transition.
effect	Specifies an optional action to be performed when the transition fires.
source	Designates the originating state vertex (state or pseudostate) of the transition.

target	Designates the target state vertex that is reached when the transition is taken.

2.12.3 Well-FormednessRules

The following well-formedness rules apply to the State Machines package.

CompositeState

[1] A composite state can have at most one initial vertex

```
self.subvertex->select (v | v.oclType = Pseudostate)->
       select(p : Pseudostate | p.kind = #initial)->size <= 1
```

[2] A composite state can have at most one deep history vertex

```
self.subvertex->select (v | v.oclType = Pseudostate)->
       select(p : Pseudostate | p.kind = #deepHistory)->size <= 1
```

[3] A composite state can have at most one shallow history vertex

```
self.subvertex->select(v | v.oclType = Pseudostate)->
       select(p : Pseudostate | p.kind = #shallowHistory)->size <= 1
```

[4] There have to be at least two composite substates in a concurrent composite state

```
(self.isConcurrent) implies
       (self.subvertex->select
           (v | v.oclIsKindOf(CompositeState))->size >= 2)
```

[5] A concurrent state can only have composite states as substates

```
(self.isConcurrent) implies
       self.subvertex->forAll(s | (s.oclIsKindOf(CompositeState))
```

[6] The substates of a composite state are part of only that composite state

```
self.subvertex->forAll(s | (s.container->size = 1) and (s.container =
self))
```

FinalState

[1] A final state cannot have any outgoing transitions

```
self.outgoing->size = 0
```

Guard

[1] A guard should not have side effects

```
self.transition->stateMachine->notEmpty implies
       post: (self.transition.stateMachine->context =
       self.transition.stateMachine->context@pre)
```

PseudoState

[1] An initial vertex can have at most one outgoing transition and no incoming transitions

```
(self.kind = #initial) implies
        ((self.outgoing->size <= 1) and (self.incoming->isEmpty))
```

[2] History vertices can have at most one outgoing transition

```
((self.kind = #deepHistory) or (self.kind = #shallowHistory)) implies
        (self.outgoing->size <= 1)
```

[3] A join vertex must have at least two incoming transitions and exactly one outgoing transition.

```
(self.kind = #join) implies
        ((self.outgoing->size = 1) and (self.incoming->size >= 2))
```

[4] A fork vertex must have at least two outgoing transitions and exactly one incoming transition.

```
(self.kind = #fork) implies
        ((self.incoming->size = 1) and (self.outgoing->size >= 2))
```

StateMachine

[1] A StateMachine is aggregated within either a classifier or a behavioral feature.

```
self.context.oclIsKindOf(BehavioralFeature) or
self.context.oclIsKindOf(Classifier)
```

[2] A top state is always a composite.

```
self.top.oclIsTypeOf(CompositeState)
```

[3] A top state cannot have any containing states

```
self.top.container->isEmpty
```

[4] The top state cannot be the source of a transition.

```
(self.top.outgoing->isEmpty)
```

[5] If a StateMachine describes a behavioral feature, it contains no triggers of type CallEvent, apart from the trigger on the initial transition (see OCL for Transition [8]).

```
self.context.oclIsKindOf(BehavioralFeature) implies

self.transitions->reject(
        source.oclIsKindOf(Pseudostate) and
                source.oclAsType(Pseudostate).kind= #initial).trigger-
>isEmpty
```

SynchState

[1] The value of the bound attribute must be a positive integer, or unlimited.

```
(self.bound > 0) or (self.bound = unlimited)
```

[3] All incoming transitions to a SynchState must come from the same region and all outgoing transitions from a SynchState must go to the same region.

SubmachineState

[1] Only stub states allowed as substates of a submachine state.

```
self.subvertex->forAll (s | s.oclIsTypeOf(StubState))
```

[2] Submachine states are never concurrent.

```
self.isConcurrent = false
```

Transition

[1] A fork segment should not have guards or triggers.

```
self.source.oclIsKindOf(Pseudostate) implies
       ((self.source.oclAsType(Pseudostate).kind = #fork) implies
           ((self.guard->isEmpty) and (self.trigger->isEmpty)))
```

[2] A join segment should not have guards or triggers.

```
self.target.oclIsKindOf(Pseudostate) implies
       ((self.target.oclAsType(Pseudostate).kind = #join) implies
           ((self.guard->isEmpty) and (self.trigger->isEmpty)))
```

[3] A fork segment should always target a state.

```
(self.stateMachine->notEmpty) implies
self.source.oclIsKindOf(Pseudostate) implies
       ((self.source.oclAsType(Pseudostate).kind = #fork) implies
                                 (self.target.oclIsKindOf(State)))
```

[4] A join segment should always originate from a state.

```
(self.stateMachine->notEmpty) implies
self.target.oclIsKindOf(Pseudostate) implies
       ((self.target.oclAsType(Pseudostate).kind = #join) implies
              (self.source.oclIsKindOf(State)))
```

[5] Transitions outgoing pseudostates may not have a trigger.

```
self.source.oclIsKindOf(Pseudostate)
       implies (self.trigger->isEmpty))
```

[6] Join segments should originate from orthogonal states.

```
self.target.oclIsKindOf(Pseudostate) implies
        ((self.target.oclAsType(Pseudostate).kind = #join) implies
                    (self.source.container.isConcurrent))
```

[7] Fork segments should target orthogonal states.

```
self.source.oclIsKindOf(Pseudostate) implies
        ((self.source.oclAsType(Pseudostate).kind = #fork) implies
                (self.target.container.isComposite))
```

[8] An initial transition at the topmost level may have a trigger with the stereotype "create." An initial transition of a StateMachine modeling a behavioral feature has a CallEvent trigger associated with that BehavioralFeature. Apart from these cases, an initial transition never has a trigger.

```
self.source.oclIsKindOf(Pseudostate) implies
        ((self.source.oclAsType(Pseudostate).kind = #initial) implies
            (self.trigger->isEmpty or
            ((self.source.container = self.stateMachine.top) and
                        (self.trigger.stereotype.name = 'create')) or
            (self.stateMachine.context.oclIsKindOf(BehavioralFeature)
and
                        self.trigger.oclIsKindOf(CallEvent) and
                        (self.trigger.oclAsType(CallEvent).operation =
                                        self.stateMachine.context))
            ))
self.source.oclIsKindOf(Pseudostate) implies
        ((self.source.kind = #initial) implies
            (self.trigger.isEmpty or
            ((self.source.container = self.StateMachine.top) and
                        (self.trigger.stereotype.name = 'create')) or
            (self.StateMachine.context.oclIsKindOf(BehaviouralFeature)
and
                        self.trigger.oclIsKindOf(CallEvent) and
                        (self.trigger.operation =
self.StateMachine.context))
            ))
```

2.12.4 Semantics

This section describes the execution semantics of state machines. For convenience, the semantics are described in terms of the operations of a hypothetical machine that implements a state machine specification. This is for reference purposes only. Individual realizations are free to choose any form that achieves the same semantics.

In the general case, the key components of this hypothetical machine are:

- an *event queue* which holds incoming event instances until they are dispatched
- an *event dispatcher mechanism* that selects and de-queues event instances from the event queue for processing
- an *event processor* which processes dispatched event instances according to the general semantics of UML state machines and the specific form of the state machine in question. Because of that, this component is simply referred to as the "state machine" in the following text.

Although the intent is to define the semantics of state machines very precisely, there are a number of semantic variation points to allow for different semantic interpretations that might be required in different domains of application. These are clearly identified in the text.

The basic semantics of events, states, transitions, etc. are discussed first in separate subsections under the appropriate headings. The operation of the state machine as a whole are then described in the state machine subsection.

Event

Event instances are generated as a result of some action either within the system or in the environment surrounding the system. An event is then conveyed to one or more targets. The means by which event instances are transported to their destination depend on the type of action, the target, the properties of the communication medium, and numerous other factors. In some cases, this is practically instantaneous and completely reliable while in others it may involve variable transmission delays, loss of events, reordering, or duplication. No specific assumptions are made in this regard. This provides full flexibility for modeling different types of communication facilities.

An event is *received* when it is placed on the event queue of its target. An event is *dispatched* when it is dequeued from the event queue and delivered to the state machine for processing. At this point, it is referred to as the *current event*. Finally, it is *consumed* when event processing is completed. A consumed event is no longer available for processing. No assumptions are made about the time intervals between event reception, event dispatching, and consumption. This leaves open the possibility of different semantic models such as zero-time semantics.

Any parameter values associated with the current event are available to all actions directly caused by that event (transition actions, entry actions, etc.).

Event generalization may be defined explicitly by a signal taxonomy in the case of signal events, or implicitly defined by event expressions, as in time events.

State

Active states

A state can be active or inactive during execution. A state becomes *active* when it is entered as a result of some transition, and becomes *inactive* if it is exited as a result of a transition. A state can be exited and entered as a result of the same transition (e.g., self transition).

State entry and exit

Whenever a state is entered, it executes its entry action *before* any other action is executed. Conversely, whenever a state is exited, it executes its exit action as the final step prior to leaving the state.

If defined, the activity associated with a state is forked as a concurrent activity at the instant when the entry action of the state is completed. Upon exit, the activity is terminated before the exit action is executed.

Activity in state (do-activity)

The activity represents the execution of a sequence of actions, that occurs while the state machine is in the corresponding state. The activity starts executing upon entering the state, following the entry action. If the activity completes while the state is still active, it raises a completion event. In case where there is an outgoing completion transition (see below) the state will be exited. If the state is exited as a result of the firing of an outgoing transition before the completion of the activity, the activity is aborted prior to its completion.

Deferred events

A state may specify a set of event types that may be *deferred* in that state. An event instance that does not trigger any transitions in the current state, will not be dispatched if its type matches one of the types in the deferred event set of that state. Instead, it remains in the event queue while another non-deferred message is dispatched instead. This situation persists until a state is reached where either the event is no longer deferred or where the event triggers a transition.

CompositeState

Active state configurations

When dealing with composite and concurrent states, the simple term “current state” can be quite confusing. In a hierarchical state machine more than one state can be active at once. If the state machine is in a simple state that is contained in a composite state, then all the composite states that either directly or transitively contain the simple state are also active. Furthermore, since some of the composite states in this hierarchy may be concurrent, the current active “state” is actually represented by a tree of states starting with the single top state at the root down to individual simple states at the leaves. We refer to such a state tree as a *state configuration*.

Except during transition execution, the following invariants always apply to state configurations:

- If a composite state is active and not concurrent, exactly one of its substates is active.
- If the composite state is active and concurrent, all of its substates (regions) are active.

Entering a non-concurrent composite state

Upon entering a composite state, the following cases are differentiated:

- *Default entry:* Graphically, this is indicated by an incoming transition that terminates on the outside edge of the composite state. In this case, the default transition is taken. If there is a guard on the transition it must be enabled (true). (A disabled initial transition is an ill-defined execution state and its handling is not defined.) The entry action of the state is executed before the action associated with the initial transition.
- *Explicit entry:* If the transition goes to a substate of the composite state, then that substate becomes active and its entry code is executed after the execution of the entry code of the composite state. This rule applies recursively if the transition terminates on a transitively nested substate.
- *Shallow history entry:* If the transition terminates on a shallow history pseudostate, the active substate becomes the most recently active substate prior to this entry, unless the most recently active substate is the final state or if this is the first entry into this state. In the latter two cases, the *default history state* is entered. This is the substate that is target of the transition originating from the history pseudostate. (If no such transition is specified, the situation is illegal and its handling is not defined.) If the active substate determined by history is a composite state, then it proceeds with its default entry.
- *Deep history entry:* The rule here is the same as for shallow history except that the rule is applied recursively to all levels in the active state configuration below this one.

Entering a concurrent composite state

Whenever a concurrent composite state is entered, each one of its concurrent substates (regions) is also entered, either by default or explicitly. If the transition terminates on the edge of the composite state, then all the regions are entered using default entry. If the transition explicitly enters one or more regions (in case of a fork), these regions are entered explicitly and the others by default.

Exiting non-concurrent state

When exiting from a composite state, the active substate is exited recursively. This means that the exit actions are executed in sequence starting with the innermost active state in the current state configuration.

Exiting a concurrent state

When exiting from a concurrent state, each of its regions is exited. After that, the exit actions of the regions are executed.

Deferred events

An event that is deferred in a composite state is automatically deferred in all directly or transitively nested substates.

FinalState

When the final state is entered, its containing composite state is *completed*, which means that it satisfies the completion condition. If the containing state is the top state, the entire state machine terminates, implying the termination of the entity associated with the state machine. If the state machine specifies the behavior of a classifier, it implies the “termination” of that instance.

SubmachineState

A submachine state is a convenience that does not introduce any additional dynamic semantics. It is semantically equivalent to a composite state and may have entry and exit actions, internal transitions, and activities.

Transitions

High-level transitions

Transitions originating from the boundary of composite states are called *high-level* or *group* transitions. If triggered, they result in exiting of all the substates of the composite state executing their exit actions starting with the innermost states in the active state configuration. Note that in terms of execution semantics, a high-level transition does not add specialized semantics, but rather reflects the semantics of exiting a composite state.

Compound transitions

A *compound transition* is a derived semantic concept, represents a “semantically complete” path made of one or more transitions, originating from a set of states (as opposed to pseudo-state) and targeting a set of states. The transition execution semantics described below, refer to compound transitions.

In general, a compound transition is an acyclical unbroken chain of transitions joined via join, junction, or fork pseudostates that define path from a set of source states (possibly a singleton) to a set of destination states, (possibly a singleton). For self-transitions, the same state acts as both the source and the destination set. A (simple) transition connecting two states is therefore a special common case of a compound transition.

The tail of a compound transition may have multiple transitions originating from a set of mutually orthogonal concurrent regions that are joined by a join point.

The head of a compound transition may have multiple transitions originating from a fork pseudostate targeted to a set of mutually orthogonal concurrent regions.

In a compound transition multiple outgoing transitions may emanate from a common junction point. In that case, only one of the outgoing transition whose guard is true is taken. If multiple transitions have guards that are true, a transition from this set is chosen. The algorithm for selecting such a transition is not specified.

Internal transitions

An internal transition executes without exiting or re-entering the state in which it is defined. This is true even if the state machine is in a nested state within this state.

Completion transitions and completion events

A *completion transition* is a transition without an explicit trigger, although it may have a guard defined. When all transition and entry actions and activities in the currently active state are completed, a *completion event* instance is generated. This event is the implicit trigger for a completion transition. The completion event is dispatched before any other queued events and has no associated parameters. For instance, a completion transition emanating from a concurrent composite state will be taken automatically as soon as all the concurrent regions have reached their final state.

If multiple completion transitions are defined for a state, then they should have mutually exclusive guard conditions.

Enabled (compound) transitions

A transition is *enabled* if and only if:

- All of its source states are in the active state configuration.
- The trigger of the transition is satisfied by the current event. An event instance *satisfies* a trigger if it matches the event specified by the trigger. In case of signal events, since signals are generalized concepts, a signal event satisfies a signal event associated with the same signal or a generalization of thereof.
- There exists at least one full path from the source state configuration to the target state configuration in which all guard conditions are true (transitions without guards are treated as if their guards are always true).

Since more than one transition may be enabled by the same event instance, being enabled is a necessary but not sufficient condition for the firing of a transition.

Guards

In a transition with guards, all guards are evaluated before a transition is triggered. The order in which the guards of a compound transition are evaluated is not defined.

Guards may include expressions causing side effects. This is considered bad practice, since such expressions are executed even if the associated transition is not taken.

Transition execution sequence

Every transition, except for internal transitions, causes exiting of a source state, and entering of the target state. These two states, which may be composite, are designated as the *main source* and the *main target* of a transition.

The *least common ancestor* state of a transition is the lowest composite state that contains all the explicit source states and explicit target states of the compound transition. In case of junction segments, only the states related to the dynamically selected path are considered explicit targets (bypassed branches are not considered).

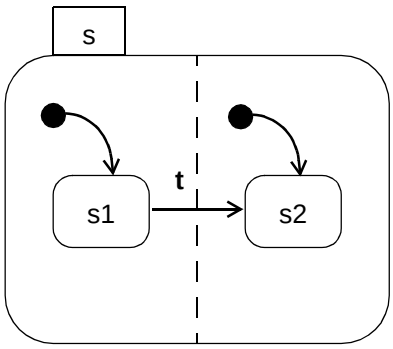
The main source is a direct substate of the least common ancestor that contains the explicit source states. The main target is a substate of the least common ancestor that contains the explicit target states.

Examples:

1. The common simple case: A transition t between two simple states s1 and s2, in a composite state s.

   Here least common ancestor of t is s, the main source is s1 and the main target is s2.

2. A more esoteric case: An unstructured transition from one region to another.



   Here least common ancestor of t is the container of s, the main source is s1 and the main target is s2

Once a transition is enabled and is selected to fire, the following steps are carried out in order:

- The main source state is properly exited.
- Actions are executed in sequence following their linear order along the segments of the transition: The closer the action to the source state, the earlier it is executed.
- The main target state is properly entered.

StateMachine

Event processing - run-to-completion step

Events are dispatched and processed by the state machine, one at a time. The order of dequeuing is not defined, leaving open the possibility of modeling different priority-based schemes.

The semantics of event processing is based on the *run-to-completion* assumption, interpreted as run-to-completion processing. Run-to-completion processing means that an event can only be dequeued and dispatched if the processing of the previous current event is fully completed.

Run-to-completion may be implemented in various ways. For active classes, it may be realized by an event-loop running in its own concurrent thread, and that reads events from a queue. For passive classes it may be implemented as a monitor.

The processing of a single event by a state machine is known as an *run-to-completion step*. Before commencing on a run-to-completion step, a state machine is in a stable state configuration with all actions (but not necessarily activities) completed. The same conditions apply after the run-to-completion step is completed. Thus, an event will never be processed while the state machine is in some intermediate and inconsistent situation. The *run-to-completion* step *is* the passage between two state configurations of the state machine.

The run-to-completion assumption simplifies the transition function of the state machine, since concurrency conflicts are avoided during the processing of event, allowing the state machine to safely complete its run-to-completion step.

When an event instance is dispatched, it may result in one or more transitions being enabled for firing. If no transition is enabled and the event is not in the deferred event list of the current state configuration, the event is discarded and the run-to-completion step is completed.

In the presence of concurrent states it is possible to fire multiple transitions as a result of the same event — as many as one transition in each concurrent state in the current state configuration. In case where one or more transitions are enabled, the state machine selects a subset and fires them. Which of the enabled transitions actually fire is determined by the transition selection algorithm described below. The order in which selected transitions fire is not defined.

Each orthogonal region in the active state configuration that is not decomposed into concurrent regions (i.e., “bottom-level” region) regions can fire at most one transition as a result of the current event. When all orthogonal regions have finished executing the transition, the current event instance is fully consumed, and the run-to-completion step is completed.

During a transition, a number of actions may be executed. If these actions are synchronous, then the transition step is not completed until the invoked objects complete their own run-to-completion steps.

An event instance can arrive at a state machine that is blocked in the middle of a run-to-completion step from some other object within the same thread, in a circular fashion. This event instance can be treated by orthogonal components of the state machine that are not frozen along transitions at that time.

Run-to-completion and concurrency

It is possible to define state machine semantics by allowing the run-to-completion steps to be applied concurrently to the orthogonal regions of a composite state, rather than to the whole state machine. This would allow the event serialization constraint to be relaxed. However, such semantics are quite subtle and difficult to implement. Therefore, the dynamic semantics defined in this document are based on the premise that a single run-to-completion step applies to the entire state machine and includes the concurrent steps taken by concurrent regions in the active state configuration.

In case of active objects, where each object has its own thread of execution, it is very important to clearly distinguish the notion of run to completion from the concept of thread pre-emption. Namely, run-to-completion event handling is performed by a thread that, in principle, *can* be pre-empted and its execution suspended in favor of another thread executing on the same processing node. (This is determined by the scheduling policy of the underlying thread

environment — no assumptions are made about this policy.) When the suspended thread is assigned processor time again, it resumes its event processing from the point of pre-emption and, eventually, completes its event processing.

Conflicting transitions

It was already noted that it is possible for more than one transition to be enabled within a state machine. If that happens, then such transitions may be in *conflict* with each other. For example, consider the case of two transitions originating from the same state, triggered by the same event, but with different guards. If that event occurs and both guard conditions are true, then only one transition will fire. In other words, in case of conflicting transitions, only one of them will fire in a single run-to-completion step.

Two transitions are said to conflict if they both exit the same state, or, more precisely, that the intersection of the set of states they exit is non-empty. Only transitions that occur in mutually orthogonal regions may be fired simultaneously. This constraint guarantees that the new active state configuration resulting from executing the set of transitions is well formed.

An internal transition in a state conflicts only with transitions that cause an exit from that state.

Firing priorities

In situations where there are conflicting transitions, the selection of which transitions will fire is based in part on an *implicit* priority. These priorities resolve some transition conflicts, but not all of them. The priorities of conflicting transitions are based on their relative position in the state hierarchy. By definition, a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states.

The priority of a transition is defined based on its source state. The priority of joined transitions is based on the priority of the transition with the most transitively nested source state.

In general, if t1 is a transition whose source state is s1, and t2 has source s2, then:

- If s1 is a direct or transitively nested substate of s2, then t1 has higher priority than t2.
- If s1 and s2 are not in the same state configuration, then there is no priority difference between t1 and t2.

Transition selection algorithm

The set of transitions that will fire is the maximal set transitions that satisfies the following conditions:

- All transitions in the set are enabled.
- There are no conflicting transitions within the set.
- There is no transition outside the set that has higher priority than a transition in the set (that is, enabled transitions with highest priorities are in the set while conflicting transitions with lower priorities are left out).

This can be easily implemented by a greedy selection algorithm, with a straightforward traversal of the active state configuration. States in the active state configuration are traversed starting with the innermost nested simple states and working outwards toward the top state. For

each state at a given level, all originating transitions are evaluated to determine if they are enabled. This traversal guarantees that the priority principle is not violated. The only non-trivial issue is resolving transition conflicts across orthogonal states on all levels. This is resolved by terminating the search in each orthogonal state once a transition inside any one of its components is fired.

Synch States

Synch states provide a means of synchronizing the execution of two concurrent regions. Specifically, a synch state has incoming transitions from a fork (or forks) in one region, the *source* region, and outgoing transitions to a join (or joins) in another, the *target* region. These forks and joins are called *synchronization* forks and joins. The synch state itself is contained by the least common ancestor of the two regions being synchronized. The synchronized regions do not need to be siblings in state decomposition, but they must have a common ancestor state.

When the source region reaches a synchronization fork, the target states of that fork become active, including the synch state. Activation of the synch state is an indication that the source region has completed some activity. This region can continue performing its remaining activities in parallel. When the target region reaches the corresponding synchronization join, it is prevented from continuing unless all the states leading into the synchronization join are active, including the synch states.

A synch state may have multiple incoming and outgoing transitions, used for multiple synchronization points in each region. Alternatively, it may have single incoming and outgoing transitions where the incoming transition is fired multiple times before the outgoing one is fired. To support these applications, synch states keep count of the difference between the number of times their incoming and outgoing transitions are fired. When an incoming transition is fired, the count is incremented by one, unless its value is equal to the value defined in the *bound* attribute. In that case, the count is not incremented. When an outgoing transition is fired, the count is decremented by one. An outgoing transition may fire only if the count is greater than zero, which prevents the count from becoming negative. The count is automatically set to zero when its container state is exited.

The bound attribute is for limiting the number of times outgoing transitions fire from a synch state. For example, to realize the equivalent of a binary semaphore, the bound should be set to one. In this case multiple incoming transitions may fire before the outgoing transition does, whereupon the outgoing transition can only fire once.

StubStates

Stub states are pseudostates signifying either entry points to or exit points from a submachine. Since a submachine is encapsulated and represented as a submachine state, multi-level (“deep”) transitions may logically connect states in separate state machines. This is facilitated by stub state, representing real states in a referenced machine to or form transitions in the referencing machine are incoming/outgoing. stub states are therefore can only be defined within a submachine state, and are the only potential subvertices of a submachine state.

2.12.5 Notes

Protocol State Machines

One application area of state machines is in specifying object protocols, also known as object life cycles. A 'protocol state machine' for a class defines the order (i.e. sequence) in which the operations of that Class can be invoked. The behaviour of each of these operations is defined by an associated method, rather than through action expressions on transitions.

A transition in a protocol state machine has as its trigger a call event that references an operation of the class, and an empty action sequence. Such a transition indicates that if the call event occurs when an object of the class is in the source state of the transition and the guard on the transition is true, then the method associated with the operation of the call event will be executed (if one exists), and the object will enter the target state. Semantically, the invocation of the method does not lead to a new call event being raised.

If a call event arrives when the state machine is not in an appropriate state to handle the event, the event is discarded, conform the general RTC semantics. Strictly speaking, from the caller's point of view this means that the call is completed. If instead the semantics are required that the caller should 'hang' (potentially infinitely) if the receiver's state machine is not able to process the call event immediately, then the event must be deffered explicitly. This can be done for all call events in a protocol state machine by deferring them at a superstate level.

In any practical application, a protocol state machine is made up exclusively of 'protocol' transitions, and the entry and exit actions of its states are empty (i.e. no action specifications exist other than for the methods). However, formally it is not prohibited to mix this kind of transition with transitions with explicit actions (as it does not seem worth the effort to prohibit this, and there may be some applications that might benefit from 'mixing').

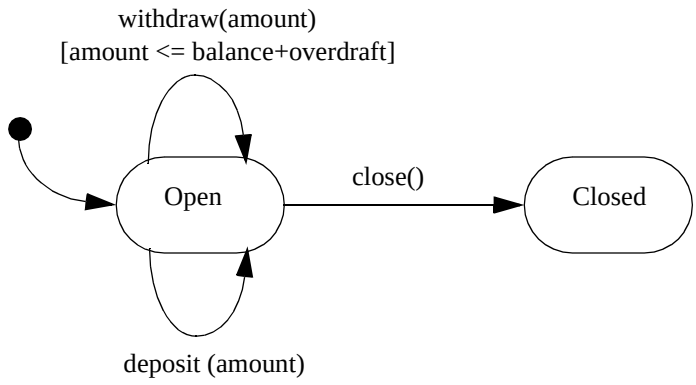


Figure 2-28 Example of a Protocol State Machine for a Class ‘Account’.

Example: Modeling Class Behavior

In the software that is implemented as a result of a state modeling design, the state machine may or may not be actually visible in the (generated or hand-crafted) code. The state machine will not be visible if there is some kind of run-time system that supports state machine behavior. In the more general case, however, the software code will contain specific statements that implement the state machine behavior.

A C++ example is shown below:

```
class bankAccount {
        private:
            int balance;
        public:
            void deposit (amount) {
                if (balance > 0)
                    balance = balance + amount' // no change
                else
                    balance = balance + amount - 1; // transaction fee
                }
            void withdrawal (amount) {
                if (balance>0)
                    balance = balance - amount;
                }
        }
```

In the above example, the class has an abstract state manifested by the balance attribute, controlling the behavior of the class. This is modeled by the state machine in Figure 2-29.

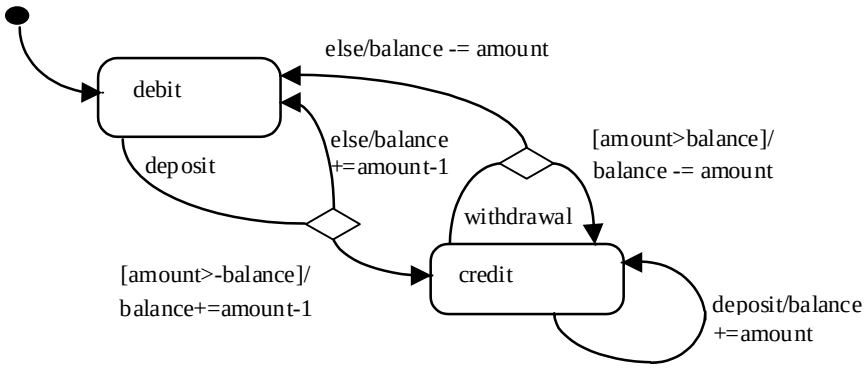


Figure 2-29 State Machine for Modeling Class Behavior

Example: State machine refinement

Note – The following discussion provides some potentially useful heuristics on how state machines can be refined. These techniques are all based on practical experience. However, readers are reminded that this topic is still the subject of research, and that it is likely that other approaches may be defined either now or in the future.

Since state machines describe behaviors of generalizable elements, primarily classes, state machine refinement is used capture the relationships between the corresponding state machines. State machines use refinement in three different mappings, specified by the mapping attribute of the refinement meta-class. The mappings are refinement, substitution, and deletion.

To illustrate state machine refinement, consider the following example where one state machine attached to a class denoted ‘Supplier,’ is refined by another state machine attached to a class denoted as ‘Client.’

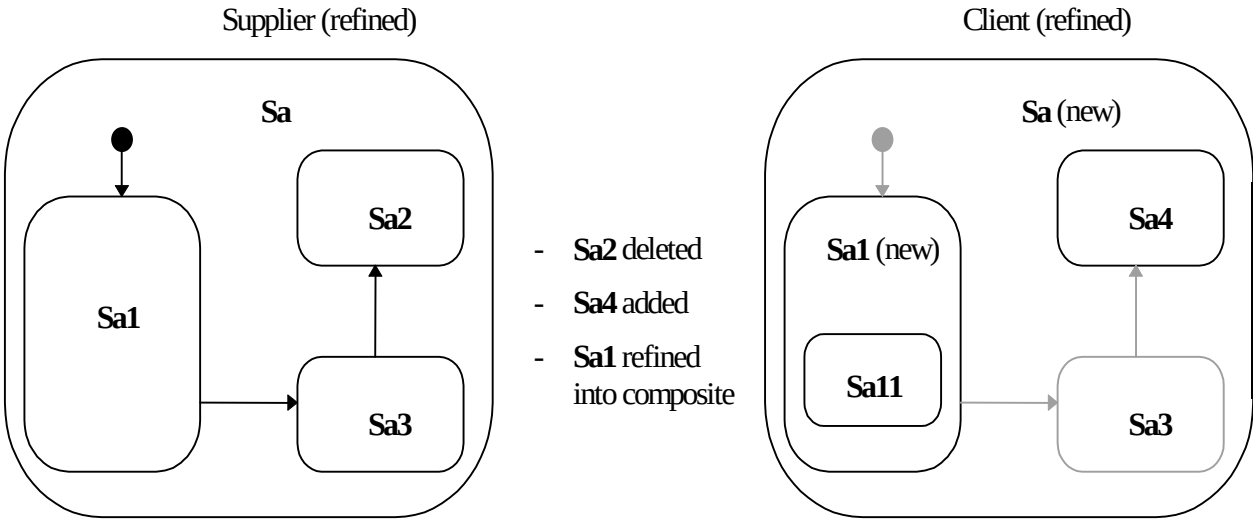


Figure 2-30 State Machine Refinement Example

In the example above, the client state (Sa(new)) in the subclass substitutes the simple substate (Sa1) by a composite substate (Sa1(new)). This new composite substate has a component substate (Sa11). Furthermore, the new version of Sa1 deletes the substate Sa2 and also adds a new substate Sa4. Substate Sa3 is inherited and is therefore common to both versions of Sa. For clarity, we have used a gray shading to identify components that have been inherited from the original. (This is for illustration purposes and is not intended as a notational recommendation.)

It is important to note that state machine refinement as defined here does not specify or favor any specific policy of state machine refinement. Instead, it simply provides a flexible mechanism that allows subtyping, (behavioral compatibility), inheritance (implementation reuse), or general refinement policies.

We provide a brief discussion of potentially useful policies that can be implemented with the state machine refinement mechanism.

Subtyping

The refinement policy for subtyping is based on the rationale that the subtype preserves the pre/post condition relationships of applying events/operations on the type, as specified by the state machine. The pre/post conditions are realized by the states, and the relationships are realized by the transitions. Preserving pre/post conditions guarantee the substitutability principle.

States and transitions are only added, not deleted. Refinement is interpreted as follows:

- A refined state has the same outgoing transitions, but may add others, and a different set of incoming transitions. It may have a bigger set of substates, and it may change its concurrency property from false to true.
- A refined transition may go to a new target state which is a substate of the state specified in the base class. This comes to guarantee the post condition specified by the base class.
- A refined guard has the same guard condition, but may add disjunctions. This guarantees that pre-conditions are weakened rather than strengthened.
- A refined action sequence contains the same actions (in the same sequence), but may have additional actions. The added actions should not hinder the invariant represented by the target state of the transition.

Strict Inheritance

The rationale behind this policy is to encourage reuse of implementation rather than preserving behavior. Since most implementation environment utilize strict inheritance (i.e. features can be replaced or added, but not deleted), the inheritance policy follows this line by disabling refinements which may lead to non-strict inheritance once the state machine is implemented.

States and transitions can be added. Refinement is interpreted as follows:

- A refined state has some of the same incoming transitions (i.e., drop some, add some) but a greater or bigger set of outgoing transitions. It may have more substates, and may change its concurrency attribute.
- A refined transition may go to a new target state but should have the same source.
- A refined guard may have a different guard condition.
- A refined action sequence contains some of the same actions (in the same sequence), and may have additional actions.

General Refinement

In this most general case, states and transitions can be added and deleted (i.e., ‘null’ refinements). Refinement is interpreted without constraints (i.e., there are no formal requirements on the properties and relationships of the refined state machine element and the refining element):

- A refined state may have different outgoing and incoming transitions (i.e., drop all, add some).
- A refined transition may leave from a different source and go to a new target state.
- A refined guard has may have a different guard condition.

- A refined action sequence need not contain the same actions (or it may change their sequence), and may have additional actions.

The refinement of the composite state in the example above is an illustration of general refinement.

It should be noted that if a type has multiple supertype relationships in the structural model, then the default state machine for the type consists of all the state machines of its supertypes as orthogonal state machine regions. This may be explicitly overridden through refinement if required.

Comparison to classical statecharts

The major difference between classical (Harel) statecharts and object state machines result from the external context of the state machine. Object state machines, such as ROOMcharts, primarily come to represent behavior of a type. Classical statechart specify behaviors of processes. The following list of differences result from the above rationale:

- Events carry parameters, rather than being primitive signals.
- Call events (operation triggers) are supported to model behaviors of types.
- Event conjunction is not supported, and the semantics is given in respect to a single event dispatch, to better match the type context as opposed to a general system context.
- Classical statecharts have an elaborated set of predefined actions, conditions and events which are not mandated by object state machines, such as entered(s), exited(s), true(condition), tr!(c) (make true), fs!(c).
- Operations are not broadcast but can be directed to an object-set.
- The notion of activities (processes) does not exist in object state machines. Therefore all predefined actions and events that deal with activities are not supported, as well as the relationships between states and activities.
- Transition compositions are constrained for practical reasons. In classical statecharts any composition of pseudostates, simple transitions, guards and labels is allowed.
- Object state machine support the notion of synchronous communication between state machines.
- Actions on transitions are executed in their given order.
- Classical statecharts are based on the zero-time assumption, meaning transitions take zero time to execute. The whole system execution is based on synchronous steps where each step produces new events that will be processed at the next step. In object-oriented state machines, these assumptions are relaxed and replaced with these of software execution model, based on threads of execution and that execution of actions may take time.

2.13 Activity Graphs

2.13.1 Overview

Activity graphs define an extended view of the State Machine package. State machines and activity graphs are both essentially state transition systems, and share many metamodel elements. This section describes the concepts in the State Machine package that are specific to activity graphs. It should be noted that the activity graphs extension has few semantics of its own. It should be understood in the context of the State Machine package, including its dependencies on the Foundation package and the Common Behavior package.

An activity graph is a special case of a state machine that is used to model processes involving one or more classifiers. Its primary focus is on the sequence and conditions for the actions that are taken, rather than on which classifiers perform those actions. Most of the states in such a graph are action states that represent atomic actions (i.e., states that invoke actions and then wait for their completion). Transitions into action states are triggered by events, which can be

- the completion of a previous action state (completion events),
- the availability of an object in a certain state,
- the occurrence of a signal, or
- the satisfaction of some condition.

By defining a small set of additional subtypes to the basic state machine concepts, the well-formedness of activity graphs can be defined formally, and subsequently mapped to the dynamic semantics of state machines. In addition, the activity specific subtypes eliminate ambiguities that might otherwise arise in the interchange of activity graphs between tools.

2.13.2 Abstract Syntax

The abstract syntax for activity graphs is expressed in graphic notation in Figure 2-28 on page 2-154.

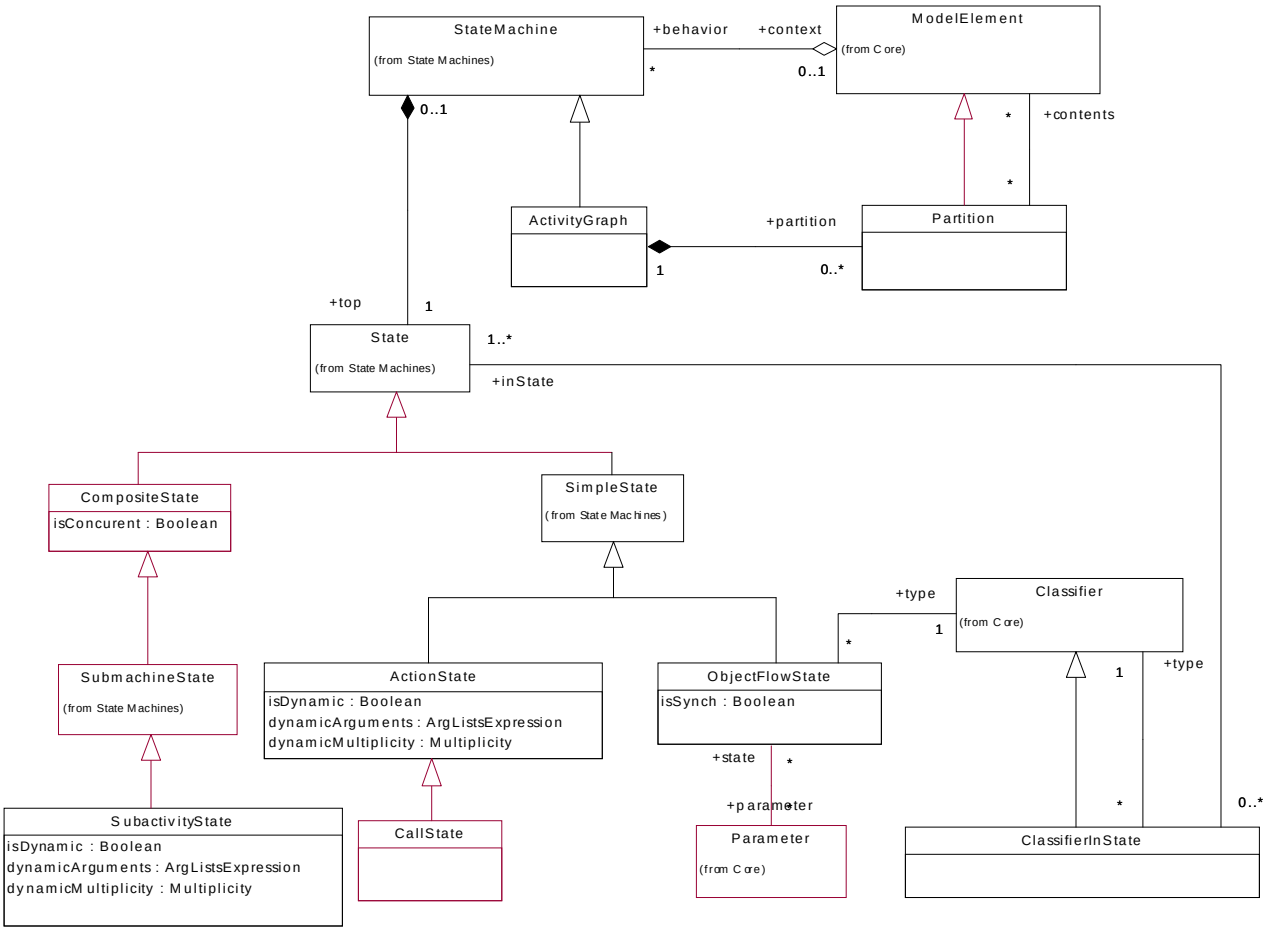


Figure 2-28 Activity Graphs

ActivityGraph

An activity graph is a special case of a state machine that defines a computational process in terms of the control-flow and object-flow among its constituent actions. It does not extend the semantics of state machines in a major way but it does define shorthand forms that are convenient for modeling control-flow and object-flow in computational and organizational processes.

The primary basis for activity graphs is to describe the states of an activity or process involving one or more classifiers. Activity graphs can be attached to packages, classifiers (including use cases) and behavioral features. As in any state machine, if an outgoing transition is not explicitly triggered by an event then it is implicitly triggered by the completion of the contained actions. A subactivity state represents a nested activity that has some duration and internally consists of a set of actions or more subactivities. That is, a subactivity state is a “hierarchical action” with an embedded activity subgraph that ultimately resolves to individual actions.

Junctions, forks, joins, and synchs may be included to model decisions and concurrent activity.

Activity graphs include the concept of Partitions to organize states according to various criteria, such as the real-world organization responsible for their performance.

Activity graphing can be applied to organizational modeling for business process engineering and workflow modeling. In this context, events often originate from inside the system, such as the finishing of a task, but also from outside the system, such as a customer call. Activity graphs can also be applied to system modeling to specify the dynamics of operations and system level processes when a full interaction model is not needed.

Associations

partition	A set of Partitions each of which contains some of the model elements of the model.

ActionState

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

An action state is a simple state with an entry action whose only exit transition is triggered by the implicit event of completing the execution of the entry action. The state therefore corresponds to the execution of the entry action itself and the outgoing transition is activated as soon as the action has completed its execution.

An ActionState may perform more than one action as part of its entry action. An action state may not have an exit action, do activity, or internal transitions.

Attributes

dynamicArguments	An ArgListsExpression that determines at runtime the number of parallel executions of the actions of the state. The value must be a set of lists of objects, each list serving as arguments for one execution. This attribute is ignored if the isDynamic attribute is false.
dynamicMultiplicity	A Multiplicity limiting the number of parallel executions of the actions of state. This attribute is ignored if the isDynamic attribute is false.
isDynamic	A boolean value specifying whether the state's actions might be executed concurrently. It is used in conjunction with the dynamicArguments attribute.

Associations

entry	(Inherited from State) Specifies the invoked actions.

CallState

A call state is an action state that has exactly one call action as its entry action. It is useful in object flow modeling to reduce notational ambiguity over which action is taking input or providing output.

ClassifierInState

A classifier-in-state characterizes instances of a given classifier that are in a particular state or states. In an activity graph, it may be used to specify input and/or output to an action through an object flow state.

ClassifierInState is a child of Classifier and may be used in static structural models and collaborations (e.g., it can be used to show associations that are only relevant when objects of a class are in a given state).

Associations

type	Designates a classifier that characterizes instances.
inState	Designates a state that characterizes instances. The state must be a valid state of the corresponding classifier. This may have multiple states when referring to an object in orthogonal states.

ObjectFlowState

An object flow state defines an object flow between actions in an activity graph. It signifies the availability of an instance of a classifier, possibly in a particular state, usually as the result of an operation. An instance of a particular class, possibly in a particular state, is available when an object flow state is occupied.

The generation of an object by an action in an action state may be modeled by an object flow state that is triggered by the completion of the action state. The use of the object in a subsequent action state may be modeled by connecting the output transition of the object flow state as an input transition to the action state. Generally each action places the object in a different state that is modeled as a distinct object flow state.

Attributes

isSynch	A boolean value indicating whether an object flow state is used as a synch state.

Associations

type	Designates a classifier that specifies the classifier of the object. It may be a classifier-in-state to specify the state and classifier of the object.
parameter	Designates parameters which provide the object as output or take it as input.

Stereotypes

«signalflow»	Signalflow is a stereotype of ObjectFlowState with a Signal as its type.

Partition

A partition is a mechanism for dividing the states of an activity graph into groups. Partitions often correspond to organizational units in a business model. They may be used to allocate characteristics or resources among the states of an activity graph.

Associations

contents	Specifies the states that belong to the partition. They need not constitute a nested region.

It should be noted that Partitions do not impact the dynamic semantics of the model but they help to allocate properties and actions for various purposes.

SubactivityState

A subactivity state represents the execution of a non-atomic sequence of steps that has some duration (i.e., internally it consists of a set of actions and possibly waiting for events). That is, a subactivity state is a “hierarchical action,” where an associated subactivity graph is executed.

A subactivity state is a submachine state that executes a nested activity graph. When an input transition to the subactivity state is triggered, execution begins with the initial state of the nested activity graph. The outgoing transitions of a subactivity state are enabled when the final state of the nested activity graph is reached (i.e., when it completes its execution), or when the trigger events occur on the transitions.

The semantics of a subactivity state are equivalent to the model obtained by statically substituting the contents of the nested graph as a composite state replacing the subactivity state.

Attributes

dynamicArguments	An ArgListsExpression that determines the number of parallel executions of the submachines of the state. The value must be a set of lists of objects, each list serving as arguments for one execution. This attribute is ignored if the isDynamic attribute is false.
dynamicMultiplicity	A Multiplicity limiting the number of parallel executions of the actions of state. This attribute is ignored if the isDynamic attribute is false.
isDynamic	A boolean value specifying whether the state's submachines might be executed concurrently. It is used in conjunction with the dynamicArguments attribute.

Associations

submachine	(Inherited from SubmachineState) This designates an activity graph that is conceptually nested within the subactivity state. The subactivity state is conceptually equivalent to a composite state whose contents are the states of the nested activity graph. The nested activity graph must have an initial state and a final state.

2.13.3 Well-Formedness Rules

ActivityGraph

[1] An ActivityGraph specifies the dynamics of

(i) a Package, or

(ii) a Classifier (including UseCase), or

(iii) a BehavioralFeature.

```
(self.context.oclIsTypeOf(Package)    xor
 self.context.oclIsKindOf(Classifier) xor
 self.context.oclIsKindOf(BehavioralFeature))
```

ActionState

[1] An action state has a non-empty entry action.

```
self.entry->size > 0
```

[2] An action state does not have an internal transition, exit action, or a do activity.

```
self.internalTransition->size = 0 and self.exit->size = 0 and
self.doActivity->size = 0
```

[3] Transitions originating from an action state have no trigger event.

```
self.outgoing->forAll(trigger->size = 0)
```

CallState

[1] The entry action of a call state is a single call action.

```
self.entry->size = 1 and self.entry.oclIsKindOf(CallAction)
```

ObjectFlowState

[1] Parameters of an object flow state must have a type and direction compatible with classifier or classifier-in-state of the object flow state.

```
let osftype : Classifier =
        (if self.type.IsKindOf(ClassifierInState)
```

```
            then self.type.type else self.type);
    self.parameter.forAll(
            type = osftype
            or (parameter.kind = #in
                    and osftype.allSupertypes->includes(type))
            or ((parameter.kind = #out or parameter.kind = #return)
                    and type.allSupertypes->includes(osftype))
            or (parameter.kind = #inout
                    and (   osftype.allSupertypes->includes(type)
                    or type.allSupertypes->includes(osftype))))
```

[2] Downstream states have entry actions that accept input conforming to the type of the classifier or classifier-in-state. The entry actions use the input parameters of the object flow state. Valid downstream states are calculated by traversing outgoing transitions transitively, skipping pseudo states, and entering and exiting subactivity states, looking for regular states. If the object flow state has no parameters, then the target of downstream actions must conform to the type of the classifier or classifier-in-state.

```
self.allnextleafstates.size > 0 and
        self.allnextleafstates.forAll(self.isinputaction(entry))
```

[3] Upstream states have entry actions that provide output or return values conforming to the type of the classifier or classifier-in-state. The entry actions use the output or return parameters of the object flow state. Valid upstream states are calculated by traversing incoming transitions transitively, skipping pseudo states, entering and exiting subactivity states, looking for regular states.

```
self.allpreviousleafstates.size > 0 and
        self.allpreviousleafstates.forAll(self.isoutputaction(entry))
```

PseudoState

[1] In activity graphs, transitions incoming to (and outgoing from) join and fork pseudostates have as sources (targets) any state vertex. That is, joins and forks are syntactically not restricted to be used in combination with composite states, as is the case in state machines.

```
self.stateMachine.oclIsTypeOf(ActivityGraph) implies
        ((self.kind = #join or self.kind = #fork) implies
            (self.incoming->forAll(source.oclIsKindOf(State) or
              source.oclIsTypeOf(PseudoState)) and
            (self.outgoing->forAll(source.oclIsKindOf(State) or
              source.oclIsTypeOf(PseudoState)))))
```

[2] All of the paths leaving a fork must eventually merge in a subsequent join in the model. Furthermore, multiple layers of forks and joins must be well nested, with the exception of forks and joins leading to or from synch state. Therefore the concurrency structure of an activity graph is in fact equally restrictive as that of an ordinary state machine, even though the composite states need not be explicit.

SubactivityState

[1] A subactivity state is a submachine state that is linked to an activity graph.

```
self.submachine.oclIsKindOf(ActivityGraph)
```

2.13.4 Semantics

ActivityGraph

The dynamic semantics of activity graphs can be expressed in terms of state machines. This means that the process structure of activities formally must be equivalent to orthogonal regions (in composite states). That is, transitions crossing between parallel paths (or threads) are not allowed, except for transitions used with synch states. As such, an activity specification that contains ‘unconstrained parallelism’ as is used in general activity graphs is considered ‘incomplete’ in terms of UML.

All events that are not relevant in a state must be deferred so they are consumed when become relevant. This is facilitated by the general deferral mechanism of state machines.

ActionState

As soon as the incoming transition of an ActionState is triggered, its entry action starts executing. Once the entry action has finished executing, the action is considered completed. When the action is complete then the outgoing transition is enabled.

The isDynamic attribute of an action state determines whether multiple invocations of state might be executed concurrently, depending on runtime information. This means that the normal activities of an action state, namely its actions may execute multiple times in parallel. If isDynamic is true, then the dynamicArguments attribute is evaluated at the time the state is entered. The size of the resulting set determines the number of parallel executions of the state. Each element of the set is a list, which is used as arguments for an execution. These arguments can be referred to within actions (e.g. by “object[i]” denoting the *i*th object in a list). If the isDynamic attribute is false, dynamicArguments is ignored. If the dynamicArguments expression evaluates to the empty set, then the state behaves as if it had no actions. It is an error if the dynamicArguments expression evaluates to a set with fewer or more elements than the number allowed by the dynamicMultiplicity attribute. The behavior is not defined in this case.

Dynamic states may be nested. In this case, you can't access the outer set of arguments in the inner nesting. If this should be necessary, arguments can be passed explicitly from the outer to the inner dynamic state.

ObjectFlowState

The activation of an object flow state signifies that an instance of the associated classifier is available, perhaps in a specified state (i.e., a state change has occurred as a result of a previous operation). This may enable a subsequent action state that requires the instance as input. As with all states in activity graphs, if the object flow state leads into a join pseudostate, then the object flow state remains activated until the other predecessors of the join have completed.

Unless there is an explicit ‘fork’ that creates orthogonal object states, only one of an object flow state’s outgoing transitions will fire as determined by the guards of the transitions. The invocation of the action state may result in a state change of the object, resulting in a new object flow state.

An object flow state may specify the parameter of an operation that provides its object as output, and the parameter of an operation that takes its object as input. The operations must be called in actions of states immediately preceding and succeeding the object flow state, respectively, although pseduostates, final states, synch states, and stub states may be interposed between the object flow state and the acting state. For example, an object flow state may transition to a subactivity state, which means at runtime the object is passed as input to the first state after the initial state of the subactivity graph. If no parameter is specified to take the flowing object as input, then it is used as an action target instead. Call actions are particularly suited to be used in conjunction with this technique because they invoke exactly one operation.

Object flow states may be used as synch states, indicated by the isSynch attribute being set to true. In this case, outgoing transitions can fire only if an object has arrived on the incoming transitions. Instead of a count, the state keeps a queue of objects as they arrive on the incoming transitions. These objects are pulled from the queue in FIFO fashion as outgoing transitions are fired. No outgoing transitions can fire if the queue is empty. All objects in the queue conform to the classifier and state specified by the object flow state. The queue is not bounded as the count may be in synch states.

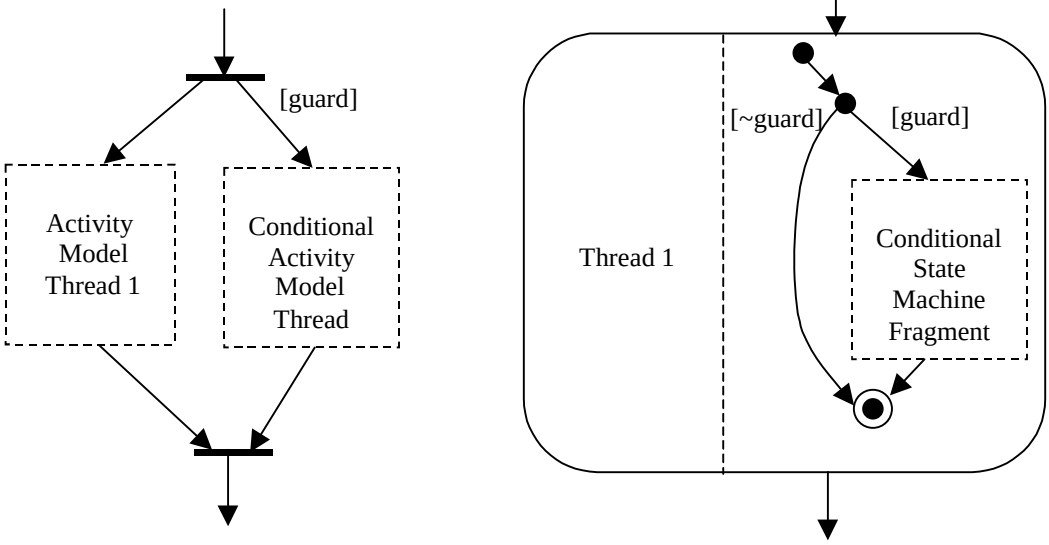
For applications requiring that actions or activities bring about an event as their result, use an object flow state with a signal as a classifier. This means the action or activity must return an instance of a signal. For multiple resulting events, transition the action or activity to a fork, and target the fork transitions at multiple object flow states.

SubactivityState

The isDynamic, dynamicArguments, and dynamicMultiplicity attributes of a subactivity state have a similar meaning to the same attributes of action states. They provide for executing the submachine of the subactivity state multiple times in parallel. See semantics of ActionState.

Transition

In activity graphs, transitions outgoing from forks may have guards. This means the region initiated by a fork transition might not start, and therefore not be required to complete at the corresponding join. Forks and joins must be well-nested in the model to use this feature (see rule #2 for PseudoState in Activity Graphs). The following mapping shows the state machine meaning for such an activity graph:



Activity diagram notation

Equivalent state machine notation

If a conditional region synchronizes with another region using a synch state, and the condition fails, then these synch states have their counts set to infinity to prevent other regions from deadlocking.

2.13.5 Notes

Object flow states in activity graphs are a specialization of the general dataflow aspect of process models. Object-flow activity graphs extend the semantics of standard dataflow relationships in three areas:

1. The operations in action states in activity graphs are operations of classes or types (e.g., ‘Trade’ or ‘OrderEntryClerk’). They are not hierarchical ‘functions’ operating on a dataflow.

2. The ‘contents’ of object flow states are typed. They are not unstructured data definitions as in data stores.

3. The state of the object flowing as input and output between operations may be defined explicitly. The event of the availability of an object in a specific state may form a trigger for the operation that requires the object as input. Object flow states are not necessarily stateless as are data stores.

Part 4 - General Mechanisms

This section defines the mechanisms of general applicability to models. This version of UML contains one general mechanisms package, Model Management. The Model Management package specifies how model elements are organized into models, packages, and subsystems.

2.14 Model Management

2.14.1 Overview

The Model Management package is dependent on the Foundation package. It defines Model, Package, and Subsystem elements that serve mainly as grouping units for other ModelElements.

Packages are used within a Model to group ModelElements. A Subsystem is a special kind of Package that represents a behavioral unit in the physical system, and hence in the model.

In this section it is necessary to clearly distinguish between the *physical system* being modeled (i.e., the subject of the model) and the system and subsystems elements that represent the physical system in the model. For this reason, we consistently use the term *physical system* when we want to indicate the former, and the terms system and subsystem when we want to indicate the latter. An example of a physical system is a credit card service, which includes software, hardware and wetware (people). The UML model for this system might consist of a top-level subsystem called CreditCardService which is decomposed into subsystems for Authorization, Credit, and Billing. An analogy with the construction of houses would be that the house would correspond to the physical system, while a blueprint would correspond to model, and an element used in a blue print would correspond to a model element.

The following sections describe the abstract syntax, well-formedness rules, and semantics of the Model Management package.

2.14.2 Abstract Syntax

The abstract syntax for the Model Management package is expressed in graphic notation in Figure 2-29.

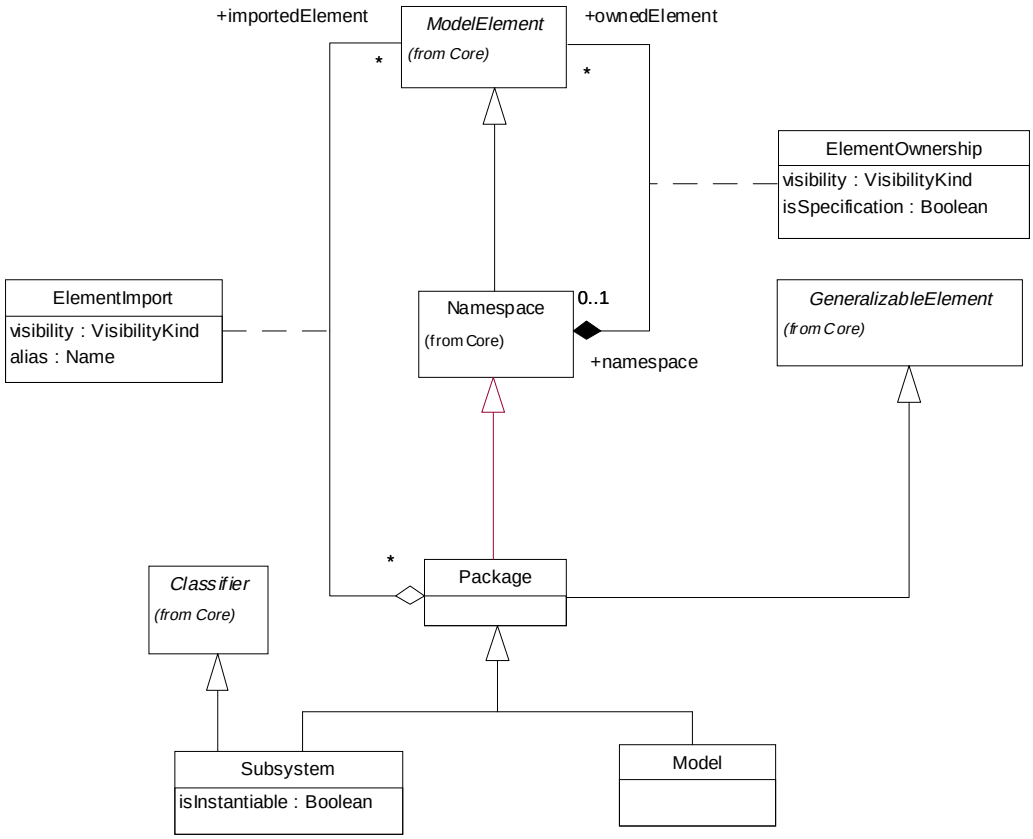


Figure 2-29 Model Management

ElementImport

An element import defines the visibility and alias of a model element included in the namespace of a package, as a result of the package importing another package.

In the metamodel an ElementImport reifies the relationship between a Package and an imported ModelElement. It allows redefinition of the name and the visibility for the imported ModelElement, i.e. the ModelElement may be given another name (an alias) and/or a new visibility to be used within the importing Package. The default is no alias, i.e. the original name will be used, and private visibility relative to the importing Package.

Attributes

alias	The alias defines a local name of the imported ModelElement, to be used within the Package.
visibility	An imported ModelElement is either public, protected, or private relative to the importing Package.

Model

A model is an abstraction of a physical system, with a certain purpose. This purpose determines what is to be included in the model and what is irrelevant. Thus the model completely describes those aspects of the physical system that are relevant to the purpose of the model, at the level of detail that is given by the purpose.

In the metamodel, Model is a subclass of Package. It contains a containment hierarchy of ModelElements that together describe the physical system. A Model also contains a set of ModelElements which represents the environment of the system, such as Actors, together with their interrelationships, such as Dependencies, Generalizations, and Constraints.

Different Models can be defined for the same physical system, specifying it from different viewpoints, like a logical model, a design model, a use-case model, etc. Each Model is self-contained within its viewpoint of the physical system and within its level of abstraction, i.e. within its purpose. Models may be nested, i.e. a Model may contain other Models.

Stereotypes

«systemModel»	A systemModel is a stereotyped model that contains a collection of models of the same physical system. A systemModel also contains all relationships and constraints between model elements contained in different models.
«metamodel»	A metamodel is a stereotyped model denoting that the model is an abstraction of another model, i.e., it is a model of a model. Hence, if M2 is a model of the model M1, then M2 is a metamodel of M1. It follows then that classes in M1 are instances of metaclasses in M2. The stereotype can be recursively applied, as in the case of a 4-layer metamodel architecture.

Package

A package is a grouping of model elements.

In the metamodel, Package is a subclass of Namespace and GeneralizableElement. A Package contains ModelElements like Packages, Classifiers, and Associations. A Package may also contain Constraints and Dependencies between ModelElements of the Package.

Each ModelElement of a Package has a visibility relative to the Package stating if the ModelElement is visible outside the Package or to a child of the Package. A Package may have «access» and «import» Permission dependencies to other Packages, allowing public

ModelElements in the other Packages to be referenced by ModelElements in the first Package. They differ in that all public ModelElements in imported Packages are added to the Namespace of the importing Package, whereas the Namespace of an accessing Package is not affected at all. The ModelElements available in a Package are those in the contents of the Namespace of the Package, which consists of owned and imported ModelElements, together with public ModelElements in accessed Packages.

Associations

importedElement	A Package references ModelElements in other, imported Packages.

Stereotypes

«facade»	A facade is a stereotyped package containing nothing but references to model elements owned by another package. It is used to provide a ‘public view’ of some of the contents of a package. A facade does not contain any model elements of its own.
«framework»	A framework is a stereotyped package consisting mainly of patterns, where patterns are defined as template collaborations.
«stub»	A stub is a stereotyped package representing a package that is incompletely transferred; specifically, a stub provides the public parts of the package, but nothing more.
«topLevel»	TopLevel is a stereotype of package denoting the top-most package in a containment hierarchy. The topLevel stereotype defines the outer limit for looking up names, as namespaces “see” outwards. A topLevel subsystem represents the top of the subsystem containment hierarchy, i.e., the system that corresponds to the physical system being modeled.

Subsystem

A subsystem is a grouping of model elements that represents a behavioral unit in a physical system. A subsytem offers interfaces and has operations. The model elements of a subsystem can be partitioned into specification and realization elements, where the former are realized by (i.e., implemented by) the latter.

In the metamodel, Subsystem is a subclass of both Package and Classifier. As such it may have a set of Features, which are constrained to be Operations and Receptions.

The contents of a Subsystem is divided into two subsets: specification elements and realization elements. The former subset provides, together with the Operations of the Subsystem, a specification of the behavior contained in the Subsystem, while the ModelElements in the latter subset jointly provide a realization of the specification. Any kind of ModelElement can be a specification element or a realization element. The relationships between the specification elements and the realization elements can be defined in different ways, e.g. with Collaborations or «realize» dependencies.

Attributes

isInstantiable	States whether a Subsystem is instantiable or not. If true, then the instances of the model elements within the subsystem form an implicit composition to an implicit subsystem instance, whether or not it is actually implemented.

2.14.3 Well-Formedness Rules

The following well-formedness rules apply to the Model Management package.

ElementImport

No extra well-formedness rules.

Model

No extra well-formedness rules.

Package

[1] A Package may only own or reference Packages, Classifiers, Associations, Generalizations, Dependencies, Constraints, Collaborations, StateMachines, and Stereotypes.

```
self.contents->forAll ( c |
        c.oclIsKindOf(Package) or
        c.oclIsKindOf(Classifier) or
        c.oclIsKindOf(Association) or
        c.oclIsKindOf(Generalization) or
        c.oclIsKindOf(Dependency) or
        c.oclIsKindOf(Constraint) or
        c.oclIsKindOf(Collaboration) or
        c.oclIsKindOf(StateMachine) or
        c.oclIsKindOf(Stereotype) )
```

[2] No imported element (excluding Association) may have the same name or alias as any element owned by the Package or one of its supertypes.

```
self.allImportedElements->reject( re |
    re.oclIsKindOf(Association) )->forAll( re |
        (re.elementImport.alias <> '' implies
            not (self.allContents - self.allImportedElements)->
                reject( ve |
                    ve.oclIsKindOf (Association) )->exists ( ve |
                        ve.name = re.elementImport.alias))
        and
```

```
        (re.elementImport.alias = '' implies
            not (self.allContents - self.allImportedElements)->
                reject ( ve |
                    ve.oclIsKindOf (Association) )->exists ( ve |
                        ve.name = re.name) ) )
```

[3] Imported elements (excluding Association) may not have the same name or alias.

```
self.allImportedElements->reject( re |
    not re.oclIsKindOf (Association) )->forAll( r1, r2 |
        (r1.elementImport.alias <> '' and
            r2.elementImport.alias <> '' and
            r1.elementImport.alias = r2.elementImport.alias
            implies r1 = r2)
        and
        (r1.elementImport.alias = '' and
            r2.elementImport.alias = '' and
            r1.name = r2.name implies r1 = r2)
        and
        (r1.elementImport.alias <> '' and
            r2.elementImport.alias = '' implies
                r1.elementImport.alias <> r2.name))
```

[4] No imported element (Association) may have the same name or alias combined with the same set of associated Classifiers as any Association owned by the Package or one of its supertypes.

```
    self.allImportedElements->select( re |
        re.oclIsKindOf(Association) )->forAll( re |
            (re.elementImport.alias <> '' implies
                not (self.allContents - self.allImportedElements)->
                select( ve |
                    ve.oclIsKindOf(Association) )->exists(
                    ve : Association |
                        ve.name = re.elementImport.alias
                        and
                        ve.connection->size = re.connection->size and
                        Sequence {1..re.connection->size}->forAll( i |
                            re.connection->at(i).type =
                            ve.connection->at(i).type ) ) )
            and
            (re.elementImport.alias = '' implies
                not (self.allContents - self.allImportedElements)->
                select( ve |
```

```
                    not ve.oclIsKindOf(Association) )->exists( ve :
                    Association |
                        ve.name = re.name
                        and
                        ve.connection->size = re.connection->size and
                        Sequence {1..re.connection->size}->forAll( i |
                            re.connection->at(i).type =
                            ve.connection->at(i).type ) ) ) )
```

[5] Imported elements (Association) may not have the same name or alias combined with the same set of associated Classifiers.

```
self.allImportedElements->select ( re |
    re.oclIsKindOf (Association) )->forAll ( r1, r2 : Association |
        (r1.connection->size = r2.connection->size and
        Sequence {1..r1.connection->size}->forAll ( i |
            r1.connection->at (i).type =
                r2.connection->at (i).type  and
            r1.elementImport.alias <> '' and
            r2.elementImport.alias <> '' and
            r1.elementImport.alias = r2.elementImport.alias
            implies r1 = r2))
        and
        (r1.connection->size = r2.connection->size and
          Sequence {1..r1.connection->size}->forAll ( i |
            r1.connection->at (i).type =
                r2.connection->at (i).type and
            r1.elementImport.alias = '' and
            r2.elementImport.alias = '' and
            r1.name = r2.name
            implies r1 = r2))
        and
        (r1.connection->size = r2.connection->size and
        Sequence {1..r1.connection->size}->forAll ( i |
            r1.connection->at (i).type =
                r2.connection->at (i).type and
            r1.elementImport.alias <> '' and
            r2.elementImport.alias = ''
            implies r1.elementImport.alias <> r2.name)))
```

Additional Operations

[1] The operation contents results in a Set containing the ModelElements owned by or imported by the Package.

```
contents : Set(ModelElement)
contents = self.ownedElement->union(self.importedElement)
```

[2] The operation allImportedElements results in a Set containing the Model Elements imported by the Package or one of its supertypes.

```
allImportedElements : Set(ModelElement)
allImportedElements = self.importedElement->union(
self.supertype.oclAsType(Package).allImportedElements->select( re |
    re.elementImport.visibility = #public or
    re.elementImport.visibility = #protected))
```

Subsystem

[1] For each Operation in an Interface offered by a Subsystem, the Subsystem itself or at least one contained specification element must have a matching Operation.

```
self.specification.allOperations->forAll(interOp |
    self.allOperations->union
        (self.allSpecificationElements->select(specEl|
            specEl.oclIsKindOf(Classifier))->forAll(c|
                c.allOperations))->exists
                    ( op | op.hasSameSignature(interOp) ) )
```

[2] The Features of a Subsystem may only be Operations or Receptions.

```
self.feature->forAll(f |  f.oclIsKindOf(Operation) or
                          f.oclIsKindOf(Reception))
```

Additional Operations

[1] The operation allSpecificationElements results in a Set containing the Model Elements specifying the behavior of the Subsystem.

```
allSpecificationElements : Set(ModelElement)
allSpecificationElements = self.allContents->select(c |
c.elementOwnership.isSpecification )
```

2.14.4 Semantics

Package

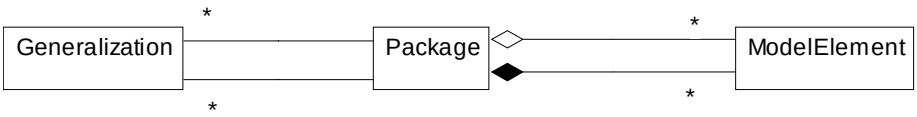


Figure 2-30 Package Illustration

The purpose of the *package* construct is to provide a general grouping mechanism. A package cannot be instantiated, thus it has no runtime semantics. In fact, its only semantics is to define a namespace for its contents. The package construct can be used for element organization of any purpose; the criteria to use for grouping elements together into one package are not defined within UML.

A package owns a set of model elements, with the implication that if the package is removed from the model, so are the elements owned by the package. Elements owned by the same package must have unique names within the package, although elements in different packages may have the same name.

There may be relationships between elements contained in the same package, and between an element in one package and an element in a surrounding package at any level. In other words, elements “see” all the way out through nested levels of packages. (A package with the stereotype «topLevel» defines the outer limit of this outward visibility, though.) Elements in peer packages, however, are encapsulated and not a priori visible to each other. The same goes for elements in contained packages, i.e. packages do not see “inwards”. There are two ways of making elements in other packages available: by importing/accessing these other packages, and by defining generalizations to them.

An *import* dependency (a Permission dependency with the stereotype «import») from one package to another means that the first package imports all the elements with sufficient visibility in the second package. Imported elements are not owned by the package; however, they may be used in associations, generalizations, attribute types, and other relationships. A package defines the *visibility* of its contained elements to be private, protected, or public. Private elements are not available at all outside the containing package. Protected elements are available only to packages with generalizations to the package owning the elements, and public elements are available also to importing and accessing packages. Note that the visibility mechanism does not restrict the availability of an element to peer elements in the same package.

When an element is imported by a package it extends the namespace of that package. It is possible to give an imported element an alias to avoid name conflicts with the names of the other elements in the namespace, including other imported elements. The alias will then be the name of that element in the namespace; the element will not appear under both the alias and its original name. An imported element is by default private to the importing package. It may, however, be given a more permissive visibility relative to the importing package, i.e. the local visibility may be defined as protected or public.

A package with an import dependency to another package imports all the public contents of the namespace defined by the supplier package, including elements of packages imported by the supplier package that are given public visibility in the supplier.

The *access* dependency (a Permission dependency with the stereotype «access») is similar to the import dependency in that it makes elements in the supplier package available to the client package. However, in this case no elements in the supplier package are included in the namespace of the client. They are simply referred to by their full pathname when referenced in the accessing package. Clearly, they are not visible to packages in turn accessing or importing this package.

A package can have *generalizations* to other packages. This means that the public and protected elements owned or imported by a package are also available to its children, and can be used in the same way as any element owned or imported by the children themselves. Elements made

available to another package by the use of a generalization are referred to by the same name in the child as they are in the parent. Moreover, they have the same visibility in the child as they have in the parent package.

A package can be used to define a framework, consisting of patterns in the form of collaborations where (some of) the base elements are the parameters of the patterns. Apart from that, a framework package is described as an ordinary package.

Subsystem

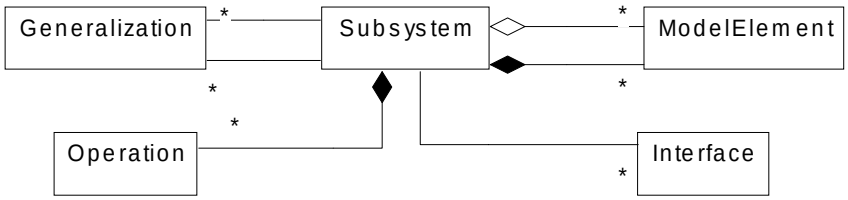


Figure 2-31 Subsystem Illustration

The purpose of the *subsystem* construct is to provide a grouping mechanism with the possibility to specify it as a behavioral unit. A subsystem may or may not be instantiable. Apart from defining a namespace for its contents, a non-instantiable subsystem serves merely as a specification unit for the behavior of its contained model elements.

The contents of a *non-instantiable* subsystem have the same semantics as that of a package, thus it consists of owned elements and imported elements, with unique names or aliases within the subsystem. The contents of a subsystem are divided into two subsets: 1) *specification elements* and 2) *realization elements*. The specification elements, together with the operations of the subsystem, are used for giving an abstract specification of the behavior offered by the realization elements.

The *specification* of a subsystem thus consists of the specification subset of the contents together with the subsystem's features (operations). It specifies the behavior performed jointly by instances of classifiers in the realization subset, without revealing anything about the contents of this subset. The specification is made in terms of e.g. use cases and/or operations, where use cases are used to specify complete sequences performed by the subsystem (i.e., by instances of its contents) interacting with its surroundings, while operations only specify fragments. However, the specification subset may include model elements of all kinds, e.g. classes, interfaces, constraints, relationships between model elements, state machines, etc.

A subsystem has no behavior of its own. All behavior defined in the specification of the subsystem is jointly offered by the elements in the realization subset of the contents. In general, since subsystems are classifiers, they can appear anywhere a classifier is expected. The general interpretation of this is that since the subsystem itself cannot be instantiated or have any behavior of its own, the requirements posed on the subsystem in the context where it occurs is fulfilled by its contents. The same is true for associations (i.e., any association connected to a subsystem is actually connected to one of the classifiers it contains).

The correspondence between the specification and the realization of a subsystem can be specified in several ways, including collaborations and «realize» dependencies. A collaboration specifies how instances of the realization elements cooperate to jointly perform the behavior

specified by a use case, an operation, etc in the subsystem specification (i.e., how the higher level of abstraction is transformed into the lower level of abstraction). A stimulus received by an instance of a use case (higher level of abstraction) corresponds to an instance conforming to one of the classifier roles in the collaboration receiving that stimulus (lower level of abstraction). This instance communicates with other instances conforming to other classifier roles in the collaboration, and together they perform the behavior specified by the use case. All stimuli that can be received and sent by instances of the use cases are also received and sent by the conforming instances, although at a lower level of abstraction. Similarly, application of an operation of the subsystem actually means that a stimulus is sent to a contained instance which then performs a method.

When subsystems are used for modeling a containment hierarchy, subordinate subsystems appear in the realization part to represent the lower level in the hierarchy.

Importing and *accessing* subsystems is done in the same way as with packages, using the *visibility* property to define whether elements are public, protected, or private to the subsystem. Both the specification part and the realization part of a subsystem may include imported elements.

A subsystem can have *generalizations* to other subsystems. This means that the public and protected elements in the contents of a subsystem are also available to its heirs. In a concrete (i.e., non-abstract) subsystem all elements in the specification, including elements from ancestors, are completely realized by cooperating realization elements, as specified with e.g. a set of collaborations. This may not be true for abstract subsystems.

Subsystems may offer a set of *interfaces*. This means that for each operation defined in an interface, the subsystem offering the interface must have a matching operation, either as a feature of the subsystem itself or of a specification element. The relationship between interface and subsystem is not necessarily one-to-one. A subsystem may realize several interfaces and one interface may be realized by more than one subsystem.

The semantics of an *instantiable* subsystem is similar to the semantics of a composite class. However, there are no explicit instances of a subsystem; instead, the instances of the model elements within the subsystem form an implicit composition to an implicit subsystem instance, whether or not it is actually implemented.

A subsystem can be used to define a framework, consisting of patterns in the form of collaborations where (some of) the base elements are the parameters of the patterns. Furthermore, the specification of a framework subsystem may also be parameterized.

Model

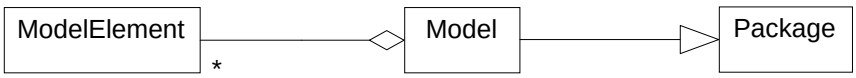


Figure 2-32 Model Illustration

A *model* is a description of a physical system with a certain purpose, such as to give a logical or a behavioral view of the physical system to a certain category of readers. Examples of different kinds of models are ‘use case’, ‘analysis’, ‘design’, and ‘implementation’, or ‘computational’, ‘engineering’, and ‘organizational’.

Thus, a model is an abstraction of a physical system, specifying the physical system from a certain viewpoint and at a certain level of abstraction, both given by the purpose of the model. A model is complete in the sense that it covers the whole physical system, although only those aspects relevant to its purpose, i.e. within the given level of abstraction and viewpoint, are represented in the model. Furthermore, it describes the physical system only once, i.e. there is no overlapping; no part of the physical system is captured more than once in a model.

A model consists of a containment hierarchy where the top-most package or subsystem represents the boundary of the physical system. This package/subsystem may be given the stereotype «topLevel» to emphasize its role within the model.

The model may also contain model elements describing relevant parts of the system's environment. The environment may e.g. be modeled by actors and their interfaces. As these are external to the physical system, they reside outside the package/subsystem hierarchy. They may be collected in a separate package, or owned directly by the model. These model elements and the model elements representing the physical system may be associated with each other.

A model may be a *specialization* of another model. This implies that all elements in the ancestor are also available in the specialized model under the same name and interrelated as in the ancestor.

A model may *import* or *access* another model. The semantics is the same as for packages. However, some of the actors of the supplier model may be internal to the client, e.g. when the imported model represents a lower layer of the physical system than the client model represents. The conformance requirement is that there must be classifiers in the client whose instances may play the roles of such actors.

The contents of a model is the transitive closure of its owned model elements, like packages, classifiers, and relationships, together with inherited and imported elements.

There may be relationships between model elements in different models, such as refinement and trace. A *trace*, i.e. an abstraction dependency with the stereotype «trace», indicates that the connected (sets of) model elements represent the same concept. Trace is used for tracing requirements between models, or tracing the impact on other models of a change to a model element in one model. Thus traces are usually non-directional dependencies. Relationships between model elements in different models have no impact on the model elements' meaning in their containing models because of the self-containment of models. Note, though, that even if inter-model relationships do not express any semantics in relation to the models, they may have semantics in relation to the reader or in deriving model elements as part of the overall development process.

Models may be nested, e.g. several models of the same physical system may be collected in a model with the stereotype «systemModel». The models contained in the «systemModel» all describe the physical system from different viewpoints, the viewpoints not necessarily disjoint. The «systemModel» also contains all inter-model relationships. A «systemModel» makes up a comprehensive specification of the physical system.

A physical system may be a composition of a set of subordinate physical systems, each described by its own set of models collected in a separate «systemModel».

2.14.5 Notes

In UML, there are three different ways to model a group of elements contained in another element; by using a package, a subsystem, or a class. Some pragmatics on their use include:

- Packages are used when nothing but a plain grouping of elements is required.
- Subsystems provide grouping suitable for top-down development, since the requirements on the behavior of their contents can be expressed before the realization of this behavior is defined. Furthermore, from a bottom-up perspective, the specification of a subsystem may also be seen as a provider of “high level APIs” of the subsystem.
- Classes are used when the container itself should be instantiable, so that it is possible to define composite objects.

As Subsystem and Model both are Packages in the metamodel, all three constructs can be combined arbitrarily to organize a containment hierarchy.

It is a tool issue to decide how many of the imported elements that must be explicitly referenced by the importing package, i.e. how many ElementImport links to actually implement.. For example, if all elements have the default visibility (private) and their original names in the importing package, the information can be retrieved directly from the imported package.

Because this is a logical model of the UML, distribution or sharing of models between tools is not described.

It is expected by tools to handle presentation elements, in particular diagrams, that are attached to model elements.

2 UML Semantics

Index