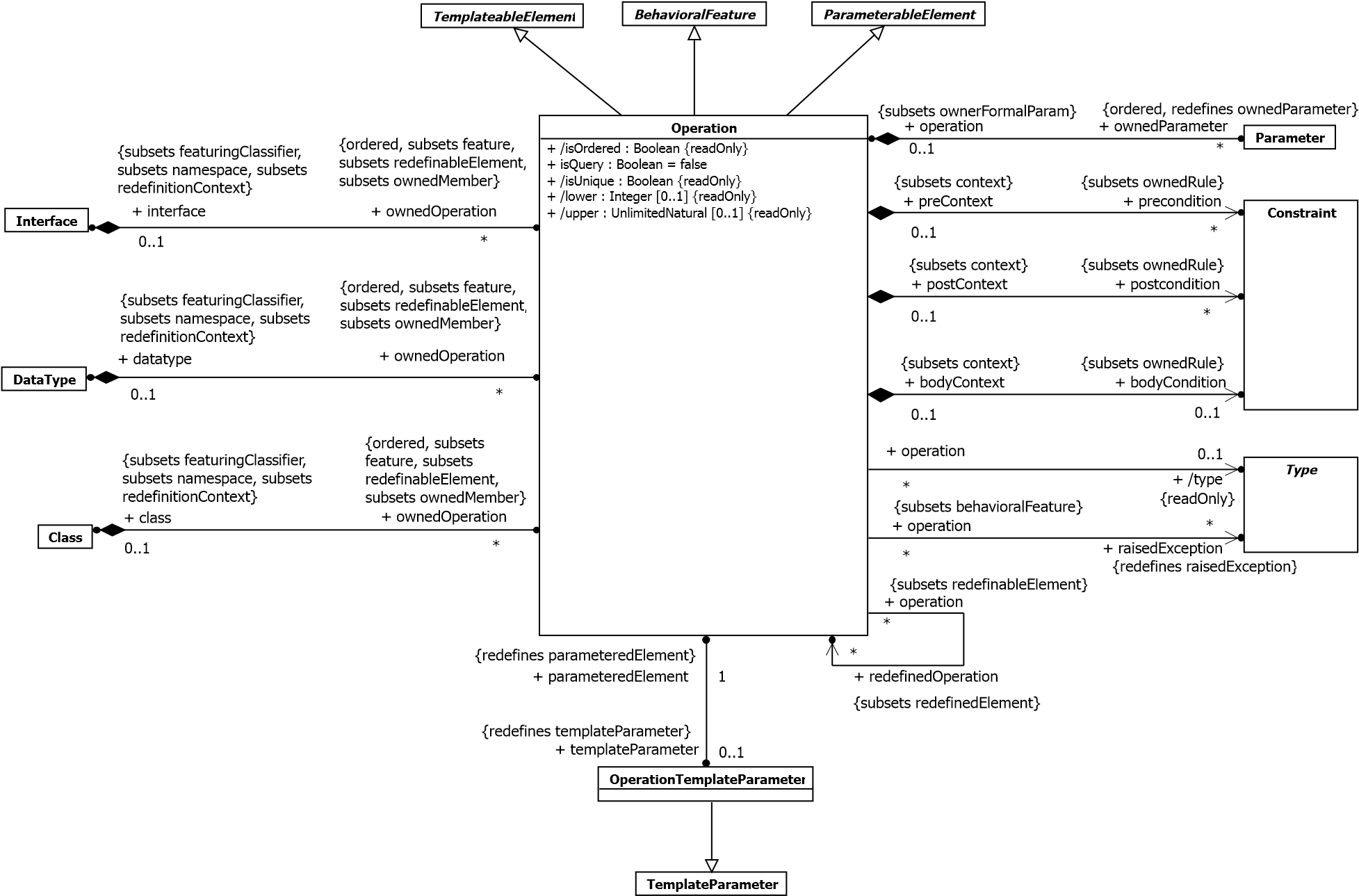
### 9.6 Operations

**9.6.1 Summary**

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

**9.6.2 Abstract Syntax**



**Figure 9.13 Operations**

**9.6.3 Semantics**

**9.6.3.1 Operations**

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

If there is a return Parameter, the type of the Operation is the same as the type of this Parameter. Otherwise the Operation has no type.

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

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

The bodyCondition for an Operation constrains the return result to a value calculated by the specification of the bodyCondition. This value should satisfy the postconditions, if any. The bodyCondition differs from postconditions in that the bodyCondition may be overridden when an Operation is redefined, whereas postconditions may only be added during redefinition.

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

An Operation may be redefined in a specialization of the featuringClassifier. This redefinition may add new preconditions or postconditions, add new raisedExceptions, or otherwise refine the specification of the Operation.

Different type-conformance systems adopt different schemes for how the types of parameters and results may vary when an Operation is redefined in a specialization. When the type may not vary, it is called invariance. When the parameter type may be specialized in a specialized type, it is called covariance. When the parameter type may be generalized in a specialized type, it is called contravariance. In UML, such rules for type conformance are intentionally not specified. Redefined parameters shall have compatible multiplicity, and the same direction, ordering and uniqueness as the redefined parameters.

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

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

**9.6.3.2 Template Operations**

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

**9.6.3.3 Operation Template Parameters**

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

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

**9.6.4 Notation**

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

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

where:

* <*visibility*> is the visibility of the Operation (see 7.4).

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

* <*name*> is the name of the Operation.
* <*parameter-list*> is a list of Parameters of the Operation in the following format: <*parameter-list*> ::= <*parameter*> [‘,’<*parameter*>]\* where <*parameter*> is defined in 9.4.4.
* <*return-type*> is the type of the return result Parameter if the Operation has one defined.
* <*multiplicity-range*> is the multiplicity of the return type (see 7.5).
* <*oper-property*> indicates the properties of the Operation.

<*oper-property*> ::= ‘redefines’ <*oper-name*> | ‘query’ | ‘ordered’ | ‘unordered’ | ‘unique’ | ‘nonunique’ | ‘seq’ | ‘sequence’ | <*oper-constraint*> where:

* ‘redefines’ <*oper-name*> means that the Operation redefines an inherited Operation identified by <*opername*>, where <*oper-name*> may be qualified.
* ‘query’ means that the Operation does not change the state of the system.
* ‘ordered’ applies when there is a multi-valued return Parameter and means that its values are ordered.
* ‘unordered’ applies when there is a multi-valued return Parameter and means that its values are not ordered.
* ‘unique’ applies when there is a multi-valued return Parameter and means that its values have no duplicates.
* ‘nonunique’ applies when there is a multi-valued return Parameter and means that its values may have duplicates.
* ‘seq’ or ‘sequence’ applies when there is a multi-valued return Parameter and means that its values constitute an ordered bag, i.e., isUnique = false and isOrdered = true.
* <*oper-constraint*> is a constraint that applies to the Operation. The parameter list may be suppressed.

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

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

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

[<*visibility*>] <*name*> ‘<<‘ <*binding-expression-list*> ‘>>’ ‘(‘ [<*parameter-list*>] ‘)’ [‘:’ [<*return-type*>] [‘[‘

<*multiplicity*> ‘]’] [‘{‘ <*oper-property*> [‘,’ <*oper-property*>]\* ‘}’]]

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

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

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

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

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

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

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

**9.6.5 Examples**

Normal Operations:

display ()

-hide ()

+createWindow (location: Coordinates, container: Container [0..1]): Window +toString (): String A template Operation:

f <T:Class>(x : T)

A binding of that template Operation.

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

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

### 9.7 Generalization Sets

**9.7.1 Summary**

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

**9.7.2 Abstract Syntax**

**GeneralizationSet**

isCovering : Boolean

+

isDisjoint : Boolean

+

**Generalization**

**PackageableElement**

**Classifier**

0..1

+

powertype

\*

+

powertypeExtent

\*

+

generalization

\*

+

generalizationSet

**Figure 9.14 Generalization Sets**

**9.7.3 Semantics**

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

The isCovering property of GeneralizationSet specifies whether the specific Classifiers of the Generalizations in that set are complete, in the following sense: if isCovering is true, then every instance of the general Classifier is an instance of (at least) one of the specific Classifiers. The isDisjoint property specifies whether the specific Classifiers of the

Generalizations in that set may overlap, in the following sense: if isDisjoint is true, then no instance of any of the specific Classifiers may also be an instance of any other of the specific Classifiers. By default, both properties are false.

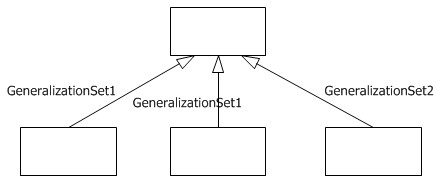
A GeneralizationSet may optionally be associated with a Classifier called its powertype. This means that for every Generalization in the GeneralizationSet, the specializing Classifier is uniquely associated with an instance of the powertype, i.e., there is a 1-1 correspondence between instances of the powertype and specializations in the

GeneralizationSet, so that the powertype instances and the corresponding Classifiers may be treated as semantically equivalent. How this semantic equivalence is implemented and how its integrity is maintained is not defined within the scope of UML.

**9.7.4 Notation**

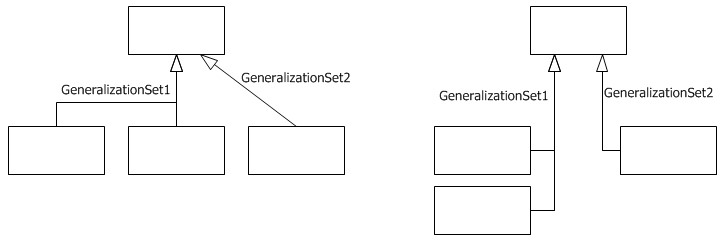
When Generalization relationship lines are named, that name designates a GeneralizationSet to which the

Generalization belongs. All Generalization relationships with the same GeneralizationSet name are part of the same GeneralizationSet. This notation form is depicted in Figure 9.15.



**Figure 9.15 GeneralizationSets designated by name**

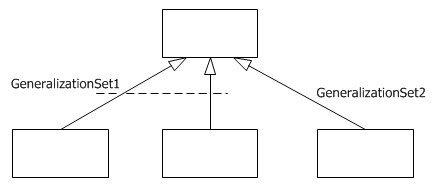
When two or more lines are drawn to the same arrowhead and labeled by a single GeneralizationSet name, i.e., “shared target” style as illustrated in Figure 9.16, the specific Classifiers are part of the same GeneralizationSet.



**Figure 9.16 GeneralizationSets designated by shared target**

With either of the notation forms above, if there are no labels on the Generalization arrows it cannot be determined from the diagram whether there are any GeneralizationSets in the model.

Lastly in Figure 9.17, a GeneralizationSet may be designated by drawing a dashed line across those lines with separate arrowheads that are meant to be part of the same set. Here, as in Figure 9.16, the GeneralizationSet is labeled with a single name, instead of each line labeled separately. This label may be elided.



**Figure 9.17 GeneralizationSet designated by dashed line spanning Generalization arrows**

To indicate whether or not a generalization set is covering and disjoint, each set may be labeled with a constraint consisting of one of the textual annotations indicated below.

**Table 9.1 GeneralizationSet constraints**

{complete, disjoint} Indicates the generalization set is covering and its specific Classifiers have no common instances.

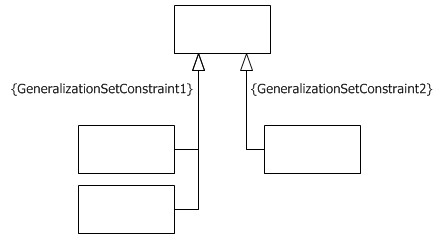
{incomplete, disjoint} Indicates the generalization set is not covering and its specific Classifiers have no common instances.

{complete, Indicates the generalization set is covering and its specific Classifiers do share common overlapping} instances.

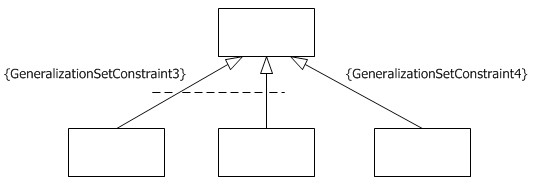
{incomplete, Indicates the generalization set is not covering and its specific Classifiers do share common overlapping} instances.

The constraints may appear in either order: {complete, disjoint} is equivalent to {disjoint, complete}. The default values are {incomplete, overlapping}. If only one constraint is shown, the other takes its default value.

Graphically, the GeneralizationSet constraints are placed next to the sets, whether the common arrowhead notation is employed as illustrated in Figure 9.18 below, or the dashed line notation as shown in Figure 9.19.

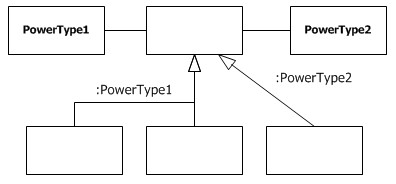


**Figure 9.18 GeneralizationSet constraint notation with shared target style**

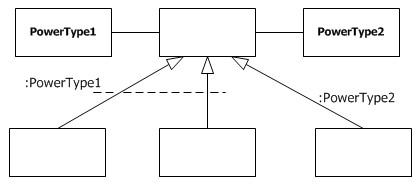


**Figure 9.19 GeneralizationSet constraint notation with dashed line style**

Power type specification is indicated by placing the name of the powertype Classifier—preceded by a colon—next to the corresponding GeneralizationSet. Figure 9.20 below indicates how this would appear for the shared arrowhead notation, and Figure 9.21 shows it for the dashed-line notation.



**Figure 9.20 Power type notation with shared target style**

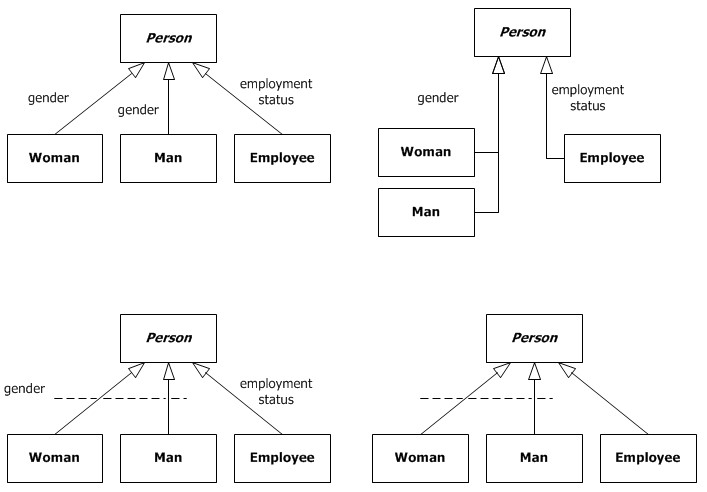


**Figure 9.21 Power type notation with dashed line style**

The labels for GeneralizationSet name, GeneralizationSet constraint and powertype may appear together in any combination on a diagram.

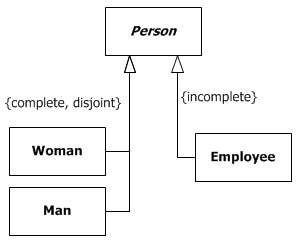
**9.7.5 Examples**

In Figure 9.22, Person (an abstract class) is specialized as Woman and Man. Separately, Person is specialized as Employee. Here, the specializations to Woman and Man constitute one GeneralizationSet and that to Employee another. This example employs the various notation forms.



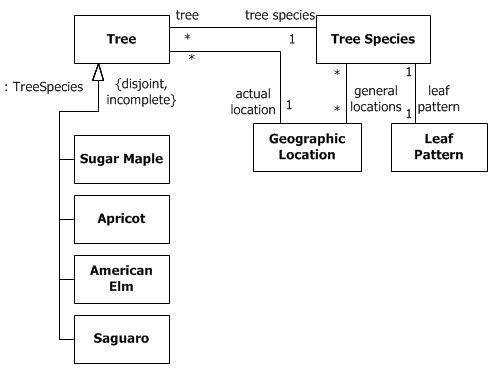
**Figure 9.22 GeneralizationSet notation options**

In Figure 9.23 below, Person (an abstract class) is specialized as Woman and Man. Because this GeneralizationSet is partitioned (i.e., is constrained to be complete and disjoint), each instance of Person shall either be a Woman or a Man; that is, it shall be one or the other and not both. Person is also specialized as Employee, and this single specialization is expressed as {incomplete}, which means that a Person may either be an Employee or not. Taken together, the diagram indicates that a Person may be 1) either a Man or Woman, and 2) an Employee or not (a total of four options).



**Figure 9.23 GeneralizationSets and constraints**

One of the ways botanists organize trees is by species. Each tree we see may be classified as an American elm, sugar maple, apricot, saguaro—or some other species of tree. The class diagram below expresses that each Tree Species classifies zero or more instances of Tree, and each Tree is classified as exactly one Tree Species. For example, one of the instances of Tree could be the tree in your front yard, the tree in your neighbor’s backyard, or trees at your local nursery. Furthermore, this figure indicates the relationships that exist between these two sets of objects. For instance, the tree in your front yard might be classified as a sugar maple, your neighbor’s tree as an apricot, and so on. This class diagram indicates that each Tree Species is identified with a Leaf Pattern and has a general location in any number of Geographic Locations. For example, the saguaro cactus has leaves reduced to large spines and is generally found in southern Arizona and northern Sonora. Additionally, this figure indicates each Tree has an actual location at a particular Geographic Location. In this way, a particular tree could be classified as a saguaro and be located in Phoenix, Arizona.

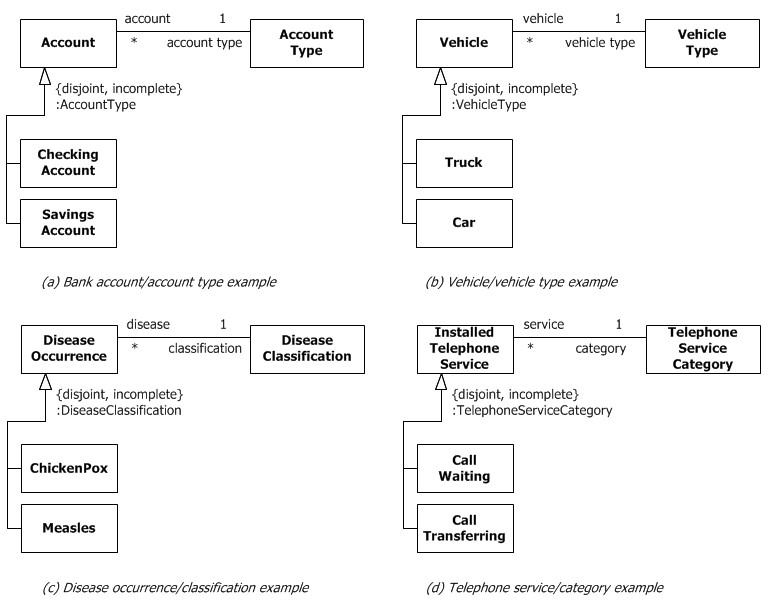


**Figure 9.24 Power type example**

This diagram also illustrates that Tree is subtyped as American Elm, Sugar Maple, Apricot, or Saguaro—or something else. Each subtype, then, may have its own specialized Properties. For instance, each Sugar Maple could have a yearly maple sugar yield of some given quantity, each Saguaro could be inhabited by zero or more instances of a Gila Woodpecker, and so on.

The powertype designation on the Tree GeneralizationSet specifies that the instances of TreeSpecies are in one-to-one correspondence to the subclasses of Tree.

This concept applies to many situations within many lines of business. Figure 9.25 depicts other examples of power types. The name on the GeneralizationSet beginning with a colon indicates the power type.

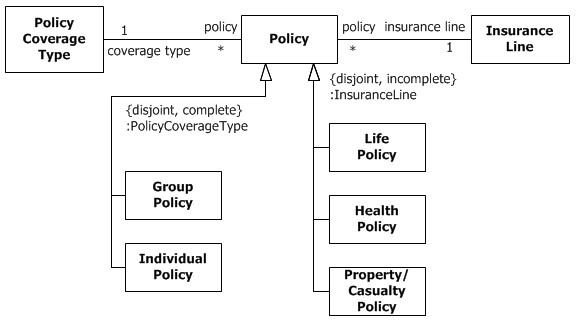


**Figure 9.25 More power type examples**

In diagram (a), each instance of Checking Account could have its own attributes (including those inherited from Account), such as account number and balance. Additionally, the equivalent instance for Checking Account may have attributes, such as interest rate and maximum delay for withdrawal.

The example (b) depicts a vehicle-modeling example. Here, each Vehicle may be classified as either a Truck or a Car or something else. Furthermore, Truck and Car are equivalent to instances of Vehicle Type. In (c), Disease Occurrence classifies each occurrence of disease (e.g., my chicken pox and your measles). Disease Classification is the power type whose instances are equivalent to classes such as Chicken Pox and Measles.

Labeling collections of subtypes with powertypes becomes increasingly important when a type has more than one powertype. Figure 9.26 illustrates one such example, showing which subtype collection contains Policy Coverage Types and which Insurance Lines. For instance, a Policy may be classified as Life, Health, Property/Casualty, or some other Insurance Line. The same Policy may be classified with its Policy Coverage Type as Group or Individual.



**Figure 9.26 More than one powertype**

### 9.8 Instances

**9.8.1 Summary**

InstanceSpecifications represent instances of Classifiers in a modeled system. They are often used to model example configurations of instances. They may be partial or complete representations of the instances that they correspond to.

**9.8.2 Abstract Syntax**

**Classifier**

**Element**

**InstanceSpecification**

**PackageableElement**

**Slot**

**StructuralFeature**

**ValueSpecification**

**DeployedArtifact**

**DeploymentTarget**

**InstanceValue**

1

owningInstance

+

\*

+

slot

}

subsets owner

{

}

subsets ownedElement

{

\*

slot

+

1

+

definingFeature

0..1

+

owningSlot

\*

value

+

}

subsets owner

{

}

ordered, subsets ownedElement

{

0..1

owningInstanceSpec

+

0..1

+

specification

{

subsets owner

}

subsets ownedElement

}

{

\*

instanceSpecification

+

\*

+

classifier

\*

+

instanceValue

1

+

instance

**Figure 9.27 Instances**

**9.8.3 Semantics**

An InstanceSpecification represents the possible or actual existence of instances in a modeled system and completely or partially describes those instances.

A Slot specifies that an instance modeled by an InstanceSpecification has a value or values for a specific

StructuralFeature, which shall be a StructuralFeature that is related to a classifier of the InstanceSpecification owning the Slot by being a direct attribute, inherited attribute, private attribute in a generalization, or a memberEnd if the classifier is an Association, but excluding redefined StructuralFeatures. The values in a Slot shall conform to the defining StructuralFeature of the Slot (in type, multiplicity, etc.). The values in a Slot are specified using ValueSpecifications (see Clause 8).

The InstanceSpecification may represent:

* Classification of the instance by one or more Classifiers, any of which may be abstract.
* The kind of instance, based on its classifiers. For example, an InstanceSpecification whose classifier is a Class describes an instance of that Class, while an InstanceSpecification whose classifier is an Association describes a link of that Association. If no classifiers are given, then the InstanceSpecification does not constrain the kind of instance represented. If classifiers of different kinds are given, then the semantics are not defined.
* Specification of values of StructuralFeatures of the instance, where the values are contained in Slots. Not all StructuralFeatures of all Classifiers of the InstanceSpecification need be represented by Slots, in which case the InstanceSpecification is a partial description.
* An optional specification, by a ValueSpecification, of how to compute, derive, or construct the instance. If such a ValueSpecification is given, then the represented instance is equal to the value resulting from the evaluation of the ValueSpecification. If the InstanceSpecification has one or more classifiers, then the type of the ValueSpecification must conform to at least one of those classifiers.

An InstanceSpecification may specify the actual existence of an instance in a modeled system. Or, an

InstanceSpecification may provide an illustration or example of a possible instance in a modeled system. The purpose of an InstanceSpecification is to show what is of interest about the instance. The instance conforms to each classifier of the InstanceSpecification, and has properties with values indicated by each slot of the InstanceSpecification. Having no slot in an InstanceSpecification for some properties does not mean that the represented instance does not have the property, but merely that the property is not of interest in the model. Similarly, the actual instance might conform to a specialization of a modeled classifier of the InstanceSpecification, but this fact may not be of interest in the model.

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

It is important to keep in mind that InstanceSpecification is a model element and should not be confused with the instance that it is modeling. As an InstanceSpecification may only partially determine the properties of an instance, there may actually be multiple instances in the modeled system that satisfy the requirements of the

InstanceSpecification. On the other hand, an InstanceSpecification may model a situation which is not actually supposed to occur in the modeled system, in which case no instance meeting the requirements of the InstanceSpecification may ever actually occur in the system.

An InstanceValue is a kind of ValueSpecification whose value is specified using an InstanceSpecification. Each evaluation of the InstanceValue is considered to result in a distinct instance conforming to the InstanceSpecification. If the InstanceSpecification has a specification, then that ValueSpecification is evaluated to give the value of the

InstanceValue. Otherwise, an InstanceValue is evaluated by creating a value that is an instance of each of the classifiers identified in the InstanceSpecification. Any slots in the InstanceSpecification then provide values for the corresponding StructuralFeatures of the instance by evaluating the ValueSpecifications associated with those slots. A StructuralFeature for which no slot is given either has the value obtained by evaluating its defaultValue, if it is a Property with a defaultValue, or no value, otherwise.

**NOTE.** An InstanceValue does not own the InstanceSpecification to which it refers; multiple InstanceValues may refer to the same InstanceSpecification.

**9.8.4 Notation**

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

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

**NOTE.** Names are optional for Classifiers and InstanceSpecifications. The absence of a name in a diagram does not necessarily reflect its absence in the underlying model.

The standard notation for an anonymous InstanceSpecification of an unnamed Classifier is an underlined colon (‘:’).

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

Slots are shown using similar notation to that of the corresponding StructuralFeatures. Where a StructuralFeature would be shown textually in a compartment, a Slot for that StructuralFeature may be shown textually as a StructuralFeature

name or qualifiedName followed by an equal sign (‘=’) and a value specification. Other properties of the StructuralFeature, such as its type, may optionally be shown.

An InstanceValue may appear using textual or graphical notation. When textual, as may appear for the value of a Slot, the name of the InstanceSpecification is shown. This may be displayed as a qualified name. When graphical, an InstanceValue is represented using the notation for its InstanceSpecification.

A Slot value that is an InstanceValue may alternatively be shown using a graphical notation similar to that for a link. A solid path runs from the owning InstanceSpecification to the symbol representing the InstanceValue that is the Slot’s value, and the name of the attribute adorns the target end of the path. Navigability, if shown, shall be only in the direction of the target. This notation can give rise to visual ambiguity with the link notation when the only adornments are at the target end; in such cases the model should be inspected to determine the presence or absence of an actual Association instance.

Where an InstanceSpecification is classified by a StructuredClassifier (see 11.2.3) it may contain nested rectangles representing the instances playing its roles. The namestring of such a nested InstanceSpecification obeys the following syntax:

{<*name*> [‘/’ <*rolename*>] | ‘/’ <*rolename*>} [‘:’ <*classifiername*> [‘,’ <*classifiername*>]\*]

The name of the InstanceSpecification may be followed by the name of the role which the instance plays. The role name may only be present if the instance plays a role.

Where an InstanceSpecification contains both Slot values and nested rectangles depicting roles, it is divided into compartments analogous to the attributes and internal structure compartments of its corresponding StructuredClassifier.

Examples of InstanceSpecifications for StructuredClassifiers are shown in 11.4.5.

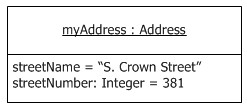
**9.8.5 Examples**

The example in Figure 9.28 below shows an InstanceSpecification called “streetName,” classified as String, and with a specification that is a LiteralString whose value is “S.Crown Street.”



**Figure 9.28 Specification of an Instance of String**

The example in Figure 9.29 below shows an InstanceSpecification with Slots.



**Figure 9.29 Slots with values**

The example in Figure 9.30 below shows a link between two InstanceSpecifications.



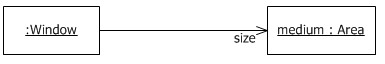
**Figure 9.30 InstanceSpecifications representing two objects connected by a link**

The example in Figure 9.31 below shows an InstanceValue as the value of a Slot represented using textual notation.



**Figure 9.31 InstanceValue represented textually**

The example in Figure 9.32 below shows the same model represented using graphical notation.



**Figure 9.32 InstanceValue represented graphically**

### 9.9 Classifier Descriptions

**9.9.1 AggregationKind [Enumeration]**

**9.9.1.1 Description**

AggregationKind is an Enumeration for specifying the kind of aggregation of a Property.

**9.9.1.2 Diagrams**

* Properties

**9.9.1.3 Literals**

* none

Indicates that the Property has no aggregation.

* shared

Indicates that the Property has shared aggregation.

* composite

Indicates that the Property is aggregated compositely, i.e., the composite object has responsibility for the existence and storage of the composed objects (parts).

**9.9.2 BehavioralFeature [Abstract Class]**

**9.9.2.1 Description**

A BehavioralFeature is a feature of a Classifier that specifies an aspect of the behavior of its instances. A

BehavioralFeature is implemented (realized) by a Behavior. A BehavioralFeature specifies that a Classifier will respond to a designated request by invoking its implementing method.

**9.9.2.2 Diagrams**

Features, Operations, Signals, Behaviors

**9.9.2.3 Generalizations**

Feature, Namespace

**9.9.2.4 Specializations**

Operation, Reception

**9.9.2.5 Attributes**

* concurrency : CallConcurrencyKind [1..1] = sequential

Specifies the semantics of concurrent calls to the same passive instance (i.e., an instance originating from a Class with isActive being false). Active instances control access to their own BehavioralFeatures.

* isAbstract : Boolean [1..1] = false

If true, then the BehavioralFeature does not have an implementation, and one must be supplied by a more specific Classifier. If false, the BehavioralFeature must have an implementation in the Classifier or one must be inherited.

**9.9.2.6 Association Ends**

* method : Behavior [0..\*] (opposite Behavior::specification)

A Behavior that implements the BehavioralFeature. There may be at most one Behavior for a particular pairing of a Classifier (as owner of the Behavior) and a BehavioralFeature (as specification of the Behavior).

* ♦ ownedParameter : Parameter [0..\*]{ordered, subsets Namespace::ownedMember} (opposite

A\_ownedParameter\_ownerFormalParam::ownerFormalParam)

The ordered set of formal Parameters of this BehavioralFeature.

♦ ownedParameterSet : ParameterSet [0..\*]{subsets Namespace::ownedMember} (opposite

A\_ownedParameterSet\_behavioralFeature::behavioralFeature) The ParameterSets owned by this BehavioralFeature.

* raisedException : Type [0..\*] (opposite A\_raisedException\_behavioralFeature::behavioralFeature) The Types representing exceptions that may be raised during an invocation of this BehavioralFeature.

**9.9.2.7 Operations**

* isDistinguishableFrom(n : NamedElement, ns : Namespace) : Boolean {redefines

NamedElement::isDistinguishableFrom()}

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

body: (n.oclIsKindOf(BehavioralFeature) and ns.getNamesOfMember(self)-

>intersection(ns.getNamesOfMember(n))->notEmpty()) implies

Set{self}->including(n.oclAsType(BehavioralFeature))->isUnique(ownedParameter->collect(p|

Tuple { name=p.name,

type=p.type,effect=p.effect,direction=p.direction,isException=p.isException,

isStream=p.isStream,isOrdered=p.isOrdered,isUnique=p.isUnique,lower=p.lower, upper=p.upper }))

* inputParameters() : Parameter [0..\*]{ordered}

The ownedParameters with direction in and inout.

body: ownedParameter->select(direction=ParameterDirectionKind::\_'in' or direction=ParameterDirectionKind::inout)

* outputParameters() : Parameter [0..\*]{ordered}

The ownedParameters with direction out, inout, or return.

body: ownedParameter->select(direction=ParameterDirectionKind::out or

direction=ParameterDirectionKind::inout or direction=ParameterDirectionKind::return)

**9.9.2.8 Constraints**

* abstract\_no\_method

When isAbstract is true there are no methods. inv: isAbstract implies method->isEmpty()

**9.9.3 CallConcurrencyKind [Enumeration]**

**9.9.3.1 Description**

CallConcurrencyKind is an Enumeration used to specify the semantics of concurrent calls to a BehavioralFeature.

**9.9.3.2 Diagrams**

* Features

**9.9.3.3 Literals**

* sequential

No concurrency management mechanism is associated with the BehavioralFeature and, therefore, concurrency

conflicts may occur. Instances that invoke a BehavioralFeature need to coordinate so that only one invocation to a target on any BehavioralFeature occurs at once.

* guarded

Multiple invocations of a BehavioralFeature that overlap in time may occur to one instance, but only one is allowed to commence. The others are blocked until the performance of the currently executing

BehavioralFeature is complete. It is the responsibility of the system designer to ensure that deadlocks do not occur due to simultaneous blocking.

* concurrent

Multiple invocations of a BehavioralFeature that overlap in time may occur to one instance and all of them may proceed concurrently.

**9.9.4 Classifier [Abstract Class]**

**9.9.4.1 Description**

A Classifier represents a classification of instances according to their Features.

**9.9.4.2 Diagrams**

Classifiers, Classifier Templates, Features, Instances, Generalization Sets, Executable Nodes, Use Cases,

Structured Classifiers, Classes, Associations, Components, Collaborations, State Machine Redefinition,

DataTypes, Signals, Interfaces, Information Flows, Artifacts, Actions, Accept Event Actions, Object Actions

**9.9.4.3 Generalizations**

Namespace, Type, TemplateableElement, RedefinableElement

**9.9.4.4 Specializations**

Association, StructuredClassifier, BehavioredClassifier, DataType, Interface, Signal, InformationItem, Artifact

**9.9.4.5 Attributes**

* + - * isAbstract : Boolean [1..1] = false

If true, the Classifier can only be instantiated by instantiating one of its specializations. An abstract Classifier is intended to be used by other Classifiers e.g., as the target of Associations or Generalizations.

* + - * isFinalSpecialization : Boolean [1..1] = false If true, the Classifier cannot be specialized.

**9.9.4.6 Association Ends**

* + - * /attribute : Property [0..\*]{ordered, union, subsets Classifier::feature, subsets

A\_redefinitionContext\_redefinableElement::redefinableElement} (opposite A\_attribute\_classifier::classifier) All of the Properties that are direct (i.e., not inherited or imported) attributes of the Classifier.

* + - * ♦ collaborationUse : CollaborationUse [0..\*]{subsets Element::ownedElement} (opposite

A\_collaborationUse\_classifier::classifier)

The CollaborationUses owned by the Classifier.

/feature : Feature [0..\*]{union, subsets Namespace::member} (opposite Feature::featuringClassifier)

Specifies each Feature directly defined in the classifier. Note that there may be members of the Classifier that are of the type Feature but are not included, e.g., inherited features.

* + - * /general : Classifier [0..\*] (opposite A\_general\_classifier::classifier) The generalizing Classifiers for this Classifier.
      * ♦ generalization : Generalization [0..\*]{subsets Element::ownedElement, subsets A\_source\_directedRelationship::directedRelationship} (opposite Generalization::specific)

The Generalization relationships for this Classifier. These Generalizations navigate to more general Classifiers in the generalization hierarchy.

* + - * /inheritedMember : NamedElement [0..\*]{subsets Namespace::member} (opposite

A\_inheritedMember\_inheritingClassifier::inheritingClassifier) All elements inherited by this Classifier from its general Classifiers.

* + - * ♦ ownedTemplateSignature : RedefinableTemplateSignature [0..1]{subsets

A\_redefinitionContext\_redefinableElement::redefinableElement, redefines

TemplateableElement::ownedTemplateSignature} (opposite RedefinableTemplateSignature::classifier) The optional RedefinableTemplateSignature specifying the formal template parameters.

* + - * ♦ ownedUseCase : UseCase [0..\*]{subsets Namespace::ownedMember} (opposite

A\_ownedUseCase\_classifier::classifier) The UseCases owned by this classifier.

* + - * powertypeExtent : GeneralizationSet [0..\*] (opposite GeneralizationSet::powertype) The GeneralizationSet of which this Classifier is a power type.
      * redefinedClassifier : Classifier [0..\*]{subsets RedefinableElement::redefinedElement} (opposite

A\_redefinedClassifier\_classifier::classifier) The Classifiers redefined by this Classifier.

* + - * representation : CollaborationUse [0..1]{subsets Classifier::collaborationUse} (opposite

A\_representation\_classifier::classifier)

A CollaborationUse which indicates the Collaboration that represents this Classifier.

* + - * ♦ substitution : Substitution [0..\*]{subsets Element::ownedElement, subsets NamedElement::clientDependency}

(opposite Substitution::substitutingClassifier) The Substitutions owned by this Classifier.

* + - * templateParameter : ClassifierTemplateParameter [0..1]{redefines ParameterableElement::templateParameter}

(opposite ClassifierTemplateParameter::parameteredElement)

TheClassifierTemplateParameter that exposes this element as a formal parameter.

* + - * useCase : UseCase [0..\*] (opposite UseCase::subject) The set of UseCases for which this Classifier is the subject.

**9.9.4.7 Operations**

* allFeatures() : Feature [0..\*]

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

body: member->select(oclIsKindOf(Feature))->collect(oclAsType(Feature))->asSet()

* allParents() : Classifier [0..\*]

The query allParents() gives all of the direct and indirect ancestors of a generalized Classifier. body: parents()->union(parents()->collect(allParents())->asSet())

* conformsTo(other : Type) : Boolean {redefines Type::conformsTo()}

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

body: if other.oclIsKindOf(Classifier) then

let otherClassifier : Classifier = other.oclAsType(Classifier) in self = otherClassifier or allParents()->includes(otherClassifier)

else false endif

* general() : Classifier [0..\*]

The general Classifiers are the ones referenced by the Generalization relationships. body: parents()

* hasVisibilityOf(n : NamedElement) : Boolean

The query hasVisibilityOf() determines whether a NamedElement is visible in the classifier. Non-private members are visible. It is only called when the argument is something owned by a parent.

pre: allParents()->including(self)->collect(member)->includes(n) body: n.visibility <> VisibilityKind::private

* inherit(inhs : NamedElement [0..\*]) : NamedElement [0..\*]

The query inherit() defines how to inherit a set of elements passed as its argument. It excludes redefined elements from the result.

body: inhs->reject(inh |

inh.oclIsKindOf(RedefinableElement) and

ownedMember->select(oclIsKindOf(RedefinableElement))->

select(redefinedElement->includes(inh.oclAsType(RedefinableElement))) ->notEmpty())

* inheritableMembers(c : Classifier) : NamedElement [0..\*]

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

pre: c.allParents()->includes(self) body: member->select(m | c.hasVisibilityOf(m))

* inheritedMember() : NamedElement [0..\*]

The inheritedMember association is derived by inheriting the inheritable members of the parents. body: inherit(parents()->collect(inheritableMembers(self))->asSet())

* isTemplate() : Boolean {redefines TemplateableElement::isTemplate()}

The query isTemplate() returns whether this Classifier is actually a template.

body: ownedTemplateSignature <> null or general->exists(g | g.isTemplate())

maySpecializeType(c : Classifier) : Boolean

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

body: self.oclIsKindOf(c.oclType())

* parents() : Classifier [0..\*]

The query parents() gives all of the immediate ancestors of a generalized Classifier. body: generalization.general->asSet()

* directlyRealizedInterfaces() : Interface [0..\*]

The Interfaces directly realized by this Classifier

body: (clientDependency->

select(oclIsKindOf(Realization) and supplier->forAll(oclIsKindOf(Interface))))-> collect(supplier.oclAsType(Interface))->asSet()

* directlyUsedInterfaces() : Interface [0..\*]

The Interfaces directly used by this Classifier

body: (supplierDependency->

select(oclIsKindOf(Usage) and client->forAll(oclIsKindOf(Interface))))-> collect(client.oclAsType(Interface))->asSet()

* allRealizedInterfaces() : Interface [0..\*]

The Interfaces realized by this Classifier and all of its generalizations

body: directlyRealizedInterfaces()->union(self.allParents()>collect(directlyRealizedInterfaces()))->asSet()

* allUsedInterfaces() : Interface [0..\*]

The Interfaces used by this Classifier and all of its generalizations

body: directlyUsedInterfaces()->union(self.allParents()->collect(directlyUsedInterfaces()))>asSet()

* isSubstitutableFor(contract : Classifier) : Boolean

body: substitution.contract->includes(contract)

* allAttributes() : Property [0..\*]{ordered}

The query allAttributes gives an ordered set of all owned and inherited attributes of the Classifier. All owned attributes appear before any inherited attributes, and the attributes inherited from any more specific parent Classifier appear before those of any more general parent Classifier. However, if the Classifier has multiple immediate parents, then the relative ordering of the sets of attributes from those parents is not defined.

body: attribute->asSequence()->union(parents()->asSequence().allAttributes())->select(p | member->includes(p))->asOrderedSet()

* allSlottableFeatures() : StructuralFeature [0..\*]

All StructuralFeatures related to the Classifier that may have Slots, including direct attributes, inherited attributes, private attributes in generalizations, and memberEnds of Associations, but excluding redefined StructuralFeatures.

body: member->select(oclIsKindOf(StructuralFeature))->

collect(oclAsType(StructuralFeature))->

union(self.inherit(self.allParents()->collect(p | p.attribute)->asSet())-> collect(oclAsType(StructuralFeature)))->asSet()

**9.9.4.8 Constraints**

 specialize\_type

A Classifier may only specialize Classifiers of a valid type. inv: parents()->forAll(c | self.maySpecializeType(c))

* + - * maps\_to\_generalization\_set

The Classifier that maps to a GeneralizationSet may neither be a specific nor a general Classifier in any of the Generalization relationships defined for that GeneralizationSet. In other words, a power type may not be an instance of itself nor may its instances also be its subclasses.

inv: powertypeExtent->forAll( gs | gs.generalization->forAll( gen |

not (gen.general = self) and not gen.general.allParents()->includes(self) and not

(gen.specific = self) and not self.allParents()->includes(gen.specific)

))

* + - * non\_final\_parents

The parents of a Classifier must be non-final. inv: parents()->forAll(not isFinalSpecialization)

* + - * no\_cycles\_in\_generalization

Generalization hierarchies must be directed and acyclical. A Classifier can not be both a transitively general and transitively specific Classifier of the same Classifier.

inv: not allParents()->includes(self)

**9.9.5 ClassifierTemplateParameter [Class]**

**9.9.5.1 Description**

A ClassifierTemplateParameter exposes a Classifier as a formal template parameter.

**9.9.5.2 Diagrams**

Classifier Templates

**9.9.5.3 Generalizations**

TemplateParameter

**9.9.5.4 Attributes**

* allowSubstitutable : Boolean [1..1] = true

Constrains the required relationship between an actual parameter and the parameteredElement for this formal parameter.

**9.9.5.5 Association Ends**

* constrainingClassifier : Classifier [0..\*] (opposite

A\_constrainingClassifier\_classifierTemplateParameter::classifierTemplateParameter)

The classifiers that constrain the argument that can be used for the parameter. If the allowSubstitutable attribute is true, then any Classifier that is compatible with this constraining Classifier can be substituted; otherwise, it must be either this Classifier or one of its specializations. If this property is empty, there are no constraints on the Classifier that can be used as an argument.

* parameteredElement : Classifier [1..1]{redefines TemplateParameter::parameteredElement} (opposite

Classifier::templateParameter)

The Classifier exposed by this ClassifierTemplateParameter.

**9.9.5.6 Constraints**

* has\_constraining\_classifier

If allowSubstitutable is true, then there must be a constrainingClassifier. inv: allowSubstitutable implies constrainingClassifier->notEmpty()

* parametered\_element\_no\_features

The parameteredElement has no direct features, and if constrainedElement is empty it has no generalizations.

inv: parameteredElement.feature->isEmpty() and (constrainingClassifier->isEmpty() implies parameteredElement.allParents()->isEmpty())

* matching\_abstract

If the parameteredElement is not abstract, then the Classifier used as an argument shall not be abstract.

inv: (not parameteredElement.isAbstract) implies templateParameterSubstitution.actual>forAll(a | not a.oclAsType(Classifier).isAbstract)

* actual\_is\_classifier

The argument to a ClassifierTemplateParameter is a Classifier. inv: templateParameterSubstitution.actual->forAll(a | a.oclIsKindOf(Classifier))

* constraining\_classifiers\_constrain\_args

If there are any constrainingClassifiers, then every argument must be the same as or a specialization of them, or if allowSubstitutable is true, then it can also be substitutable.

inv: templateParameterSubstitution.actual->forAll( a | let arg : Classifier = a.oclAsType(Classifier) in

constrainingClassifier->forAll(

cc |

arg = cc or arg.conformsTo(cc) or (allowSubstitutable and

arg.isSubstitutableFor(cc))

)

)

* constraining\_classifiers\_constrain\_parametered\_element

If there are any constrainingClassifiers, then the parameteredElement must be the same as or a specialization of them, or if allowSubstitutable is true, then it can also be substitutable.

inv: constrainingClassifier->forAll(

cc | parameteredElement = cc or parameteredElement.conformsTo(cc) or

(allowSubstitutable and parameteredElement.isSubstitutableFor(cc))

)

**9.9.6 Feature [Abstract Class]**

**9.9.6.1 Description**

A Feature declares a behavioral or structural characteristic of Classifiers.

**9.9.6.2 Diagrams**

Classifiers, Features, Structured Classifiers

**9.9.6.3 Generalizations**

RedefinableElement

**9.9.6.4 Specializations**

BehavioralFeature, StructuralFeature, Connector

**9.9.6.5 Attributes**

* isStatic : Boolean [1..1] = false

Specifies whether this Feature characterizes individual instances classified by the Classifier (false) or the Classifier itself (true).

**9.9.6.6 Association Ends**

* /featuringClassifier : Classifier [0..1]{union, subsets A\_member\_memberNamespace::memberNamespace}

(opposite Classifier::feature)

The Classifiers that have this Feature as a feature.

**9.9.7 Generalization [Class]**

**9.9.7.1 Description**

A Generalization is a taxonomic relationship between a more general Classifier and a more specific Classifier. Each instance of the specific Classifier is also an instance of the general Classifier. The specific Classifier inherits the features of the more general Classifier. A Generalization is owned by the specific Classifier.

**9.9.7.2 Diagrams**

Classifiers, Generalization Sets

**9.9.7.3 Generalizations**

DirectedRelationship

**9.9.7.4 Attributes**

* isSubstitutable : Boolean [0..1] = true

Indicates whether the specific Classifier can be used wherever the general Classifier can be used. If true, the execution traces of the specific Classifier shall be a superset of the execution traces of the general Classifier. If false, there is no such constraint on execution traces. If unset, the modeler has not stated whether there is such a constraint or not.

**9.9.7.5 Association Ends**

* general : Classifier [1..1]{subsets DirectedRelationship::target} (opposite

A\_general\_generalization::generalization)

The general classifier in the Generalization relationship.

* generalizationSet : GeneralizationSet [0..\*] (opposite GeneralizationSet::generalization)

Represents a set of instances of Generalization. A Generalization may appear in many GeneralizationSets.

* specific : Classifier [1..1]{subsets DirectedRelationship::source, subsets Element::owner} (opposite

Classifier::generalization)

The specializing Classifier in the Generalization relationship.

**9.9.8 GeneralizationSet [Class]**

**9.9.8.1 Description**

A GeneralizationSet is a PackageableElement whose instances represent sets of Generalization relationships.

**9.9.8.2 Diagrams**

Classifiers, Generalization Sets

**9.9.8.3 Generalizations**

PackageableElement

**9.9.8.4 Attributes**

* isCovering : Boolean [1..1] = false

Indicates (via the associated Generalizations) whether or not the set of specific Classifiers are covering for a particular general classifier. When isCovering is true, every instance of a particular general Classifier is also an instance of at least one of its specific Classifiers for the GeneralizationSet. When isCovering is false, there are one or more instances of the particular general Classifier that are not instances of at least one of its specific Classifiers defined for the GeneralizationSet.

* isDisjoint : Boolean [1..1] = false

Indicates whether or not the set of specific Classifiers in a Generalization relationship have instance in common. If isDisjoint is true, the specific Classifiers for a particular GeneralizationSet have no members in common; that is, their intersection is empty. If isDisjoint is false, the specific Classifiers in a particular GeneralizationSet have one or more members in common; that is, their intersection is not empty.

**9.9.8.5 Association Ends**

* generalization : Generalization [0..\*] (opposite Generalization::generalizationSet) Designates the instances of Generalization that are members of this GeneralizationSet.
* powertype : Classifier [0..1] (opposite Classifier::powertypeExtent)

Designates the Classifier that is defined as the power type for the associated GeneralizationSet, if there is one.

**9.9.8.6 Constraints**

* generalization\_same\_classifier

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

inv: generalization->collect(general)->asSet()->size() <= 1

* maps\_to\_generalization\_set

The Classifier that maps to a GeneralizationSet may neither be a specific nor a general Classifier in any of the Generalization relationships defined for that GeneralizationSet. In other words, a power type may not be an instance of itself nor may its instances be its subclasses.

inv: powertype <> null implies generalization->forAll( gen |

not (gen.general = powertype) and not gen.general.allParents()->includes(powertype) and not (gen.specific = powertype) and not powertype.allParents()->includes(gen.specific) )

**9.9.9 InstanceSpecification [Class]**

**9.9.9.1 Description**

An InstanceSpecification is a model element that represents an instance in a modeled system. An InstanceSpecification can act as a DeploymentTarget in a Deployment relationship, in the case that it represents an instance of a Node. It can also act as a DeployedArtifact, if it represents an instance of an Artifact.

**9.9.9.2 Diagrams**

Instances, DataTypes, Deployments

**9.9.9.3 Generalizations**

DeploymentTarget, PackageableElement, DeployedArtifact

**9.9.9.4 Specializations**

EnumerationLiteral

**9.9.9.5 Association Ends**

* classifier : Classifier [0..\*] (opposite A\_classifier\_instanceSpecification::instanceSpecification)

The Classifier or Classifiers of the represented instance. If multiple Classifiers are specified, the instance is classified by all of them.

* ♦ slot : Slot [0..\*]{subsets Element::ownedElement} (opposite Slot::owningInstance)

A Slot giving the value or values of a StructuralFeature of the instance. An InstanceSpecification can have one Slot per StructuralFeature of its Classifiers, including inherited features. It is not necessary to model a Slot for every StructuralFeature, in which case the InstanceSpecification is a partial description.

* ♦ specification : ValueSpecification [0..1]{subsets Element::ownedElement} (opposite

A\_specification\_owningInstanceSpec::owningInstanceSpec)

A specification of how to compute, derive, or construct the instance.

**9.9.9.6 Constraints**

* deployment\_artifact

An InstanceSpecification can act as a DeployedArtifact if it represents an instance of an Artifact. inv: deploymentForArtifact->notEmpty() implies classifier->exists(oclIsKindOf(Artifact))

* structural\_feature

No more than one slot in an InstanceSpecification may have the same definingFeature.

inv: classifier->forAll(c | (c.allSlottableFeatures()->forAll(f | slot->select(s |

s.definingFeature = f)->size() <= 1)))

* defining\_feature

The definingFeature of each slot is a StructuralFeature related to a classifier of the InstanceSpecification, including direct attributes, inherited attributes, private attributes in generalizations, and memberEnds of Associations, but excluding redefined StructuralFeatures.

inv: slot->forAll(s | classifier->exists (c | c.allSlottableFeatures()->includes (s.definingFeature)))

* deployment\_target

An InstanceSpecification can act as a DeploymentTarget if it represents an instance of a Node and functions as a part in the internal structure of an encompassing Node.

inv: deployment->notEmpty() implies classifier->exists(node | node.oclIsKindOf(Node) and Node.allInstances()->exists(n | n.part->exists(p | p.type = node)))

**9.9.10 InstanceValue [Class]**

**9.9.10.1 Description**

An InstanceValue is a ValueSpecification that identifies an instance.

**9.9.10.2 Diagrams**

Instances

**9.9.10.3 Generalizations**

ValueSpecification

**9.9.10.4 Association Ends**

* instance : InstanceSpecification [1..1] (opposite A\_instance\_instanceValue::instanceValue) The InstanceSpecification that represents the specified value.

**9.9.11 Operation [Class]**

**9.9.11.1 Description**

An Operation is a BehavioralFeature of a Classifier that specifies the name, type, parameters, and constraints for invoking an associated Behavior. An Operation may invoke both the execution of method behaviors as well as other behavioral responses. Operation specializes TemplateableElement in order to support specification of template operations and bound operations. Operation specializes ParameterableElement to specify that an operation can be exposed as a formal template parameter, and provided as an actual parameter in a binding of a template.

**9.9.11.2 Diagrams**

Operations, Classes, Protocol State Machines, DataTypes, Interfaces, Artifacts, Events, Invocation Actions

**9.9.11.3 Generalizations**

TemplateableElement, ParameterableElement, BehavioralFeature

**9.9.11.4 Attributes**

* /isOrdered : Boolean [1..1]

Specifies whether the return parameter is ordered or not, if present. This information is derived from the return result for this Operation.

* isQuery : Boolean [1..1] = false

Specifies whether an execution of the BehavioralFeature leaves the state of the system unchanged (isQuery=true) or whether side effects may occur (isQuery=false).

* /isUnique : Boolean [1..1]

Specifies whether the return parameter is unique or not, if present. This information is derived from the return result for this Operation.

* /lower : Integer [0..1]

Specifies the lower multiplicity of the return parameter, if present. This information is derived from the return result for this Operation.

* /upper : UnlimitedNatural [0..1]

The upper multiplicity of the return parameter, if present. This information is derived from the return result for this Operation.

**9.9.11.5 Association Ends**

* ♦ bodyCondition : Constraint [0..1]{subsets Namespace::ownedRule} (opposite

A\_bodyCondition\_bodyContext::bodyContext)

An optional Constraint on the result values of an invocation of this Operation.

* class : Class [0..1]{subsets Feature::featuringClassifier, subsets NamedElement::namespace, subsets

RedefinableElement::redefinitionContext} (opposite Class::ownedOperation) The Class that owns this operation, if any.

* datatype : DataType [0..1]{subsets Feature::featuringClassifier, subsets NamedElement::namespace, subsets

RedefinableElement::redefinitionContext} (opposite DataType::ownedOperation) The DataType that owns this Operation, if any.

* interface : Interface [0..1]{subsets Feature::featuringClassifier, subsets NamedElement::namespace, subsets

RedefinableElement::redefinitionContext} (opposite Interface::ownedOperation) The Interface that owns this Operation, if any.

* ♦ ownedParameter : Parameter [0..\*]{ordered, redefines BehavioralFeature::ownedParameter} (opposite

Parameter::operation)

The parameters owned by this Operation.

* ♦ postcondition : Constraint [0..\*]{subsets Namespace::ownedRule} (opposite

A\_postcondition\_postContext::postContext)

An optional set of Constraints specifying the state of the system when the Operation is completed.

* ♦ precondition : Constraint [0..\*]{subsets Namespace::ownedRule} (opposite

A\_precondition\_preContext::preContext)

An optional set of Constraints on the state of the system when the Operation is invoked.

* raisedException : Type [0..\*]{redefines BehavioralFeature::raisedException} (opposite

A\_raisedException\_operation::operation)

The Types representing exceptions that may be raised during an invocation of this operation.

* redefinedOperation : Operation [0..\*]{subsets RedefinableElement::redefinedElement} (opposite

A\_redefinedOperation\_operation::operation)

The Operations that are redefined by this Operation.

* templateParameter : OperationTemplateParameter [0..1]{redefines ParameterableElement::templateParameter}

(opposite OperationTemplateParameter::parameteredElement)

The OperationTemplateParameter that exposes this element as a formal parameter.

* /type : Type [0..1]{} (opposite A\_type\_operation::operation)

The return type of the operation, if present. This information is derived from the return result for this Operation.

**9.9.11.6 Operations**

* isConsistentWith(redefiningElement : RedefinableElement) : Boolean {redefines

RedefinableElement::isConsistentWith()}

The query isConsistentWith() specifies, for any two Operations in a context in which redefinition is possible, whether redefinition would be consistent. A redefining operation is consistent with a redefined operation if it has the same number of owned parameters, and for each parameter the following holds: - Direction, ordering and uniqueness are the same. - The corresponding types are covariant, contravariant or invariant. - The multiplicities are compatible, depending on the parameter direction.

pre: redefiningElement.isRedefinitionContextValid(self) body: redefiningElement.oclIsKindOf(Operation) and let op : Operation = redefiningElement.oclAsType(Operation) in

self.ownedParameter->size() = op.ownedParameter->size() and

Sequence{1..self.ownedParameter->size()}-> forAll(i |

let redefiningParam : Parameter = op.ownedParameter->at(i),

redefinedParam : Parameter = self.ownedParameter->at(i) in (redefiningParam.isUnique = redefinedParam.isUnique) and

(redefiningParam.isOrdered = redefinedParam. isOrdered) and

(redefiningParam.direction = redefinedParam.direction) and

(redefiningParam.type.conformsTo(redefinedParam.type) or redefinedParam.type.conformsTo(redefiningParam.type)) and

(redefiningParam.direction = ParameterDirectionKind::inout implies

(redefinedParam.compatibleWith(redefiningParam) and redefiningParam.compatibleWith(redefinedParam))) and

(redefiningParam.direction = ParameterDirectionKind::\_'in' implies redefinedParam.compatibleWith(redefiningParam)) and ((redefiningParam.direction = ParameterDirectionKind::out or redefiningParam.direction = ParameterDirectionKind::return) implies

redefiningParam.compatibleWith(redefinedParam))

)

* isOrdered() : Boolean

If this operation has a return parameter, isOrdered equals the value of isOrdered for that parameter. Otherwise isOrdered is false.

body: if returnResult()->notEmpty() then returnResult()-> exists(isOrdered) else false endif

* isUnique() : Boolean

If this operation has a return parameter, isUnique equals the value of isUnique for that parameter. Otherwise isUnique is true.

body: if returnResult()->notEmpty() then returnResult()->exists(isUnique) else true endif

* lower() : Integer

If this operation has a return parameter, lower equals the value of lower for that parameter. Otherwise lower has no value. body: if returnResult()->notEmpty() then returnResult()->any(true).lower else null endif

* returnResult() : Parameter [0..\*]

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

body: ownedParameter->select (direction = ParameterDirectionKind::return)

* type() : Type

If this operation has a return parameter, type equals the value of type for that parameter. Otherwise type has no value. body: if returnResult()->notEmpty() then returnResult()->any(true).type else null endif

* upper() : UnlimitedNatural

If this operation has a return parameter, upper equals the value of upper for that parameter. Otherwise upper has no value. body: if returnResult()->notEmpty() then returnResult()->any(true).upper else null endif

**9.9.11.7 Constraints**

* at\_most\_one\_return

An Operation can have at most one return parameter; i.e., an owned parameter with the direction set to 'return.'

inv: self.ownedParameter->select(direction = ParameterDirectionKind::return)->size() <= 1

* only\_body\_for\_query

A bodyCondition can only be specified for a query Operation. inv: bodyCondition <> null implies isQuery

**9.9.12 OperationTemplateParameter [Class]**

**9.9.12.1 Description**

An OperationTemplateParameter exposes an Operation as a formal parameter for a template.

**9.9.12.2 Diagrams**

Operations

**9.9.12.3 Generalizations**

TemplateParameter

**9.9.12.4 Association Ends**

* parameteredElement : Operation [1..1]{redefines TemplateParameter::parameteredElement} (opposite

Operation::templateParameter)

The Operation exposed by this OperationTemplateParameter.

**9.9.12.5 Constraints**

* match\_default\_signature

inv: default->notEmpty() implies (default.oclIsKindOf(Operation) and (let defaultOp :

Operation = default.oclAsType(Operation) in

defaultOp.ownedParameter->size() = parameteredElement.ownedParameter->size() and

Sequence{1.. defaultOp.ownedParameter->size()}->forAll( ix |

let p1: Parameter = defaultOp.ownedParameter->at(ix), p2 : Parameter =

parameteredElement.ownedParameter->at(ix) in

p1.type = p2.type and p1.upper = p2.upper and p1.lower = p2.lower and p1.direction = p2.direction and p1.isOrdered = p2.isOrdered and p1.isUnique = p2.isUnique)))

**9.9.13 Parameter [Class]**

**9.9.13.1 Description**

A Parameter is a specification of an argument used to pass information into or out of an invocation of a BehavioralFeature. Parameters can be treated as ConnectableElements within Collaborations.

**9.9.13.2 Diagrams**

Features, Operations, Object Nodes, Expressions, Behaviors

**9.9.13.3 Generalizations**

MultiplicityElement, ConnectableElement

**9.9.13.4 Attributes**

* /default : String [0..1]

A String that represents a value to be used when no argument is supplied for the Parameter.

* direction : ParameterDirectionKind [1..1] = in

Indicates whether a parameter is being sent into or out of a behavioral element.

* effect : ParameterEffectKind [0..1]

Specifies the effect that executions of the owner of the Parameter have on objects passed in or out of the parameter.

* isException : Boolean [1..1] = false

Tells whether an output parameter may emit a value to the exclusion of the other outputs.

* isStream : Boolean [1..1] = false

Tells whether an input parameter may accept values while its behavior is executing, or whether an output parameter may post values while the behavior is executing.

**9.9.13.5 Association Ends**

* ♦ defaultValue : ValueSpecification [0..1]{subsets Element::ownedElement} (opposite

A\_defaultValue\_owningParameter::owningParameter)

Specifies a ValueSpecification that represents a value to be used when no argument is supplied for the Parameter.

* operation : Operation [0..1]{subsets A\_ownedParameter\_ownerFormalParam::ownerFormalParam} (opposite

Operation::ownedParameter)

The Operation owning this parameter.

* parameterSet : ParameterSet [0..\*] (opposite ParameterSet::parameter) The ParameterSets containing the parameter. See ParameterSet.

**9.9.13.6 Operations**

* default() : String [0..1]

Derivation for Parameter::/default

body: if self.type = String then defaultValue.stringValue() else null endif

**9.9.13.7 Constraints**

* in\_and\_out

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

inv: (effect = ParameterEffectKind::delete implies (direction =

ParameterDirectionKind::\_'in' or direction = ParameterDirectionKind::inout))

and

(effect = ParameterEffectKind::create implies (direction = ParameterDirectionKind::out or direction = ParameterDirectionKind::inout or direction = ParameterDirectionKind::return))

* not\_exception

An input Parameter cannot be an exception.

inv: isException implies (direction <> ParameterDirectionKind::\_'in' and direction <> ParameterDirectionKind::inout)

* connector\_end

A Parameter may only be associated with a Connector end within the context of a Collaboration. inv: end->notEmpty() implies collaboration->notEmpty()

* reentrant\_behaviors

Reentrant behaviors cannot have stream Parameters. inv: (isStream and behavior <> null) implies not behavior.isReentrant

* stream\_and\_exception

A Parameter cannot be a stream and exception at the same time. inv: not (isException and isStream)

* object\_effect

Parameters typed by DataTypes cannot have an effect.

inv: (type.oclIsKindOf(DataType)) implies (effect = null)

**9.9.14 ParameterDirectionKind [Enumeration]**

**9.9.14.1 Description**

ParameterDirectionKind is an Enumeration that defines literals used to specify direction of parameters.

**9.9.14.2 Diagrams**

* Features

**9.9.14.3 Literals**

* in

Indicates that Parameter values are passed in by the caller.

* inout

Indicates that Parameter values are passed in by the caller and (possibly different) values passed out to the caller.

* out

Indicates that Parameter values are passed out to the caller.

* return

Indicates that Parameter values are passed as return values back to the caller.

**9.9.15 ParameterEffectKind [Enumeration]**

**9.9.15.1 Description**

ParameterEffectKind is an Enumeration that indicates the effect of a Behavior on values passed in or out of its parameters.

**9.9.15.2 Diagrams**

* Features

**9.9.15.3 Literals**

* create

Indicates that the behavior creates values.

* read

Indicates objects that are values of the parameter have values of their properties, or links in which they participate, or their classifiers retrieved during executions of the behavior.

* update

Indicates objects that are values of the parameter have values of their properties, or links in which they participate, or their classification changed during executions of the behavior.

* delete

Indicates objects that are values of the parameter do not exist after executions of the behavior are finished.

**9.9.16 ParameterSet [Class]**

**9.9.16.1 Description**

A ParameterSet designates alternative sets of inputs or outputs that a Behavior may use.

**9.9.16.2 Diagrams**

Features, Behaviors

**9.9.16.3 Generalizations**

NamedElement

**9.9.16.4 Association Ends**

* ♦ condition : Constraint [0..\*]{subsets Element::ownedElement} (opposite

A\_condition\_parameterSet::parameterSet)

A constraint that should be satisfied for the owner of the Parameters in an input ParameterSet to start execution using the values provided for those Parameters, or the owner of the Parameters in an output ParameterSet to end execution providing the values for those Parameters, if all preconditions and conditions on input ParameterSets were satisfied.

* parameter : Parameter [1..\*] (opposite Parameter::parameterSet) Parameters in the ParameterSet.

**9.9.16.5 Constraints**

* same\_parameterized\_entity

The Parameters in a ParameterSet must all be inputs or all be outputs of the same parameterized entity, and the ParameterSet is owned by that entity.

inv: parameter->forAll(p1, p2 | self.owner = p1.owner and self.owner = p2.owner and p1.direction = p2.direction)

* input

If a parameterized entity has input Parameters that are in a ParameterSet, then any inputs that are not in a ParameterSet must be streaming. Same for output Parameters.

inv: ((parameter->exists(direction = ParameterDirectionKind::\_'in')) implies

behavioralFeature.ownedParameter->select(p | p.direction = ParameterDirectionKind::\_'in'

and p.parameterSet->isEmpty())->forAll(isStream))

and

((parameter->exists(direction = ParameterDirectionKind::out)) implies

behavioralFeature.ownedParameter->select(p | p.direction = ParameterDirectionKind::out and p.parameterSet->isEmpty())->forAll(isStream))

* two\_parameter\_sets

Two ParameterSets cannot have exactly the same set of Parameters.

inv: parameter->forAll(parameterSet->forAll(s1, s2 | s1->size() = s2->size() implies

s1.parameter->exists(p | not s2.parameter->includes(p))))

**9.9.17 Property [Class]**

**9.9.17.1 Description**

A Property is a StructuralFeature. A Property related by ownedAttribute to a Classifier (other than an association) represents an attribute and might also represent an association end. It relates an instance of the Classifier to a value or set of values of the type of the attribute. A Property related by memberEnd to an Association represents an end of the Association. The type of the Property is the type of the end of the Association. A Property has the capability of being a

DeploymentTarget in a Deployment relationship. This enables modeling the deployment to hierarchical nodes that have Properties functioning as internal parts. Property specializes ParameterableElement to specify that a Property can be exposed as a formal template parameter, and provided as an actual parameter in a binding of a template.

**9.9.17.2 Diagrams**

Classifiers, Properties, Encapsulated Classifiers, Structured Classifiers, Classes, Associations, DataTypes, Signals, Interfaces, Profiles, Deployments, Artifacts, Link End Data, Link Object Actions

**9.9.17.3 Generalizations**

ConnectableElement, DeploymentTarget, StructuralFeature

**9.9.17.4 Specializations**

Port, ExtensionEnd

**9.9.17.5 Attributes**

* aggregation : AggregationKind [1..1] = none

Specifies the kind of aggregation that applies to the Property.

* /isComposite : Boolean [1..1] = false

If isComposite is true, the object containing the attribute is a container for the object or value contained in the attribute. This is a derived value, indicating whether the aggregation of the Property is composite or not.

* isDerived : Boolean [1..1] = false

Specifies whether the Property is derived, i.e., whether its value or values can be computed from other information.

* isDerivedUnion : Boolean [1..1] = false

Specifies whether the property is derived as the union of all of the Properties that are constrained to subset it.

* isID : Boolean [1..1] = false

True indicates this property can be used to uniquely identify an instance of the containing Class.

**9.9.17.6 Association Ends**

* association : Association [0..1]{subsets A\_member\_memberNamespace::memberNamespace} (opposite

Association::memberEnd)

The Association of which this Property is a member, if any.

* associationEnd : Property [0..1]{subsets Element::owner} (opposite Property::qualifier) Designates the optional association end that owns a qualifier attribute.
* class : Class [0..1]{subsets NamedElement::namespace, subsets

A\_ownedAttribute\_structuredClassifier::structuredClassifier, subsets A\_attribute\_classifier::classifier} (opposite Class::ownedAttribute)

The Class that owns this Property, if any.

* datatype : DataType [0..1]{subsets NamedElement::namespace, subsets A\_attribute\_classifier::classifier}

(opposite DataType::ownedAttribute)

The DataType that owns this Property, if any.

* ♦ defaultValue : ValueSpecification [0..1]{subsets Element::ownedElement} (opposite

A\_defaultValue\_owningProperty::owningProperty)

A ValueSpecification that is evaluated to give a default value for the Property when an instance of the owning Classifier is instantiated.

* interface : Interface [0..1]{subsets NamedElement::namespace, subsets A\_attribute\_classifier::classifier}

(opposite Interface::ownedAttribute)

The Interface that owns this Property, if any.

* /opposite : Property [0..1] (opposite A\_opposite\_property::property)

In the case where the Property is one end of a binary association this gives the other end.

* owningAssociation : Association [0..1]{subsets Feature::featuringClassifier, subsets

NamedElement::namespace, subsets Property::association, subsets RedefinableElement::redefinitionContext} (opposite Association::ownedEnd)

The owning association of this property, if any.

* ♦ qualifier : Property [0..\*]{ordered, subsets Element::ownedElement} (opposite Property::associationEnd) An optional list of ordered qualifier attributes for the end.
* redefinedProperty : Property [0..\*]{subsets RedefinableElement::redefinedElement} (opposite

A\_redefinedProperty\_property::property)

The properties that are redefined by this property, if any.

* subsettedProperty : Property [0..\*] (opposite A\_subsettedProperty\_property::property) The properties of which this Property is constrained to be a subset, if any.

**9.9.17.7 Operations**

* isAttribute() : Boolean

The query isAttribute() is true if the Property is defined as an attribute of some Classifier.

body: not classifier->isEmpty()

* isCompatibleWith(p : ParameterableElement) : Boolean {redefines

ParameterableElement::isCompatibleWith()}

The query isCompatibleWith() determines if this Property is compatible with the specified

ParameterableElement. This Property is compatible with ParameterableElement p if the kind of this Property is thesame as or a subtype of the kind of p. Further, if p is a TypedElement, then the type of this Property must be conformant with the type of p.

body: self.oclIsKindOf(p.oclType()) and (p.oclIsKindOf(TypeElement) implies

self.type.conformsTo(p.oclAsType(TypedElement).type))

* isComposite() : Boolean

The value of isComposite is true only if aggregation is composite. body: aggregation = AggregationKind::composite

* isConsistentWith(redefiningElement : RedefinableElement) : Boolean {redefines

RedefinableElement::isConsistentWith()}

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

pre: redefiningElement.isRedefinitionContextValid(self) body: redefiningElement.oclIsKindOf(Property) and

let prop : Property = redefiningElement.oclAsType(Property) in

(prop.type.conformsTo(self.type) and

((prop.lowerBound()->notEmpty() and self.lowerBound()->notEmpty()) implies

prop.lowerBound() >= self.lowerBound()) and

((prop.upperBound()->notEmpty() and self.upperBound()->notEmpty()) implies

prop.lowerBound() <= self.lowerBound()) and (self.isComposite implies prop.isComposite))

* isNavigable() : Boolean

The query isNavigable() indicates whether it is possible to navigate across the property. body: not classifier->isEmpty() or association.navigableOwnedEnd->includes(self)

* opposite() : Property

If this property is a memberEnd of a binary association, then opposite gives the other end.

body: if association <> null and association.memberEnd->size() = 2

then

association.memberEnd->any(e | e <> self)

else null endif

* subsettingContext() : Type [0..\*]

The query subsettingContext() gives the context for subsetting a Property. It consists, in the case of an attribute, of the corresponding Classifier, and in the case of an association end, all of the Classifiers at the other ends.

body: if association <> null

then association.memberEnd->excluding(self)->collect(type)->asSet()

else

if classifier<>null then classifier->asSet()

else Set{} endif endif

**9.9.17.8 Constraints**

* subsetting\_context\_conforms

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

inv: subsettedProperty->notEmpty() implies

(subsettingContext()->notEmpty() and subsettingContext()->forAll (sc |

subsettedProperty->forAll(sp |

sp.subsettingContext()->exists(c | sc.conformsTo(c)))))

* derived\_union\_is\_read\_only A derived union is read only. inv: isDerivedUnion implies isReadOnly
* multiplicity\_of\_composite
  1. multiplicity on the composing end of a composite aggregation must not have an upper bound greater than 1.inv: isComposite and association <> null implies opposite.upperBound() <= 1
* redefined\_property\_inherited
  1. redefined Property must be inherited from a more general Classifier.

inv: (redefinedProperty->notEmpty()) implies

(redefinitionContext->notEmpty() and redefinedProperty->forAll(rp|

((redefinitionContext->collect(fc|

fc.allParents()))->asSet())->collect(c| c.allFeatures())->asSet()->includes(rp)))

* subsetting\_rules
  1. subsetting Property may strengthen the type of the subsetted Property, and its upper bound may be less.

inv: subsettedProperty->forAll(sp | self.type.conformsTo(sp.type) and

((self.upperBound()->notEmpty() and sp.upperBound()->notEmpty()) implies self.upperBound() <= sp.upperBound() ))

* binding\_to\_attribute
  1. binding of a PropertyTemplateParameter representing an attribute must be to an attribute.

inv: (self.isAttribute()

and (templateParameterSubstitution->notEmpty()) implies (templateParameterSubstitution->forAll(ts |

ts.formal.oclIsKindOf(Property)

and ts.formal.oclAsType(Property).isAttribute())))

* derived\_union\_is\_derived A derived union is derived. inv: isDerivedUnion implies isDerived
* deployment\_target
  1. Property can be a DeploymentTarget if it is a kind of Node and functions as a part in the internal structure of an encompassing Node.

inv: deployment->notEmpty() implies owner.oclIsKindOf(Node) and Node.allInstances()>exists(n | n.part->exists(p | p = self))

* subsetted\_property\_names
  1. Property may not subset a Property with the same name.inv: subsettedProperty->forAll(sp | sp.name <> name)
* type\_of\_opposite\_end

If a Property is a classifier-owned end of a binary Association, its owner must be the type of the opposite end.

inv: (opposite->notEmpty() and owningAssociation->isEmpty()) implies classifier =

opposite.type

* qualified\_is\_association\_end

All qualified Properties must be Association ends

inv: qualifier->notEmpty() implies association->notEmpty()

**9.9.18 RedefinableElement [Abstract Class]**

**9.9.18.1 Description**

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

**9.9.18.2 Diagrams**

Classifiers, Classifier Templates, Features, Activities, Use Cases, State Machine Redefinition

**9.9.18.3 Generalizations**

NamedElement

**9.9.18.4 Specializations**

Classifier, Feature, RedefinableTemplateSignature, ActivityEdge, ActivityNode, ExtensionPoint, Region, State, Transition

**9.9.18.5 Attributes**

* isLeaf : Boolean [1..1] = false

Indicates whether it is possible to further redefine a RedefinableElement. If the value is true, then it is not possible to further redefine the RedefinableElement.

**9.9.18.6 Association Ends**

* /redefinedElement : RedefinableElement [0..\*]{union} (opposite

A\_redefinedElement\_redefinableElement::redefinableElement) The RedefinableElement that is being redefined by this element.

* /redefinitionContext : Classifier [0..\*]{union} (opposite A\_redefinitionContext\_redefinableElement::redefinableElement) The contexts that this element may be redefined from.

**9.9.18.7 Operations**

* isConsistentWith(redefiningElement : RedefinableElement) : Boolean

The query isConsistentWith() specifies, for any two RedefinableElements in a context in which redefinition is possible, whether redefinition would be logically consistent. By default, this is false; this operation must be overridden for subclasses of RedefinableElement to define the consistency conditions.

pre: redefiningElement.isRedefinitionContextValid(self) body: false

* isRedefinitionContextValid(redefinedElement : RedefinableElement) : Boolean

The query isRedefinitionContextValid() specifies whether the redefinition contexts of this RedefinableElement are properly related to the redefinition contexts of the specified RedefinableElement to allow this element to redefine the other. By default at least one of the redefinition contexts of this element must be a specialization of at least one of the redefinition contexts of the specified element.

body: redefinitionContext->exists(c | c.allParents()>includesAll(redefinedElement.redefinitionContext))

**9.9.18.8 Constraints**

* redefinition\_consistent

A redefining element must be consistent with each redefined element. inv: redefinedElement->forAll(re | re.isConsistentWith(self))

* non\_leaf\_redefinition

A RedefinableElement can only redefine non-leaf RedefinableElements. inv: redefinedElement->forAll(re | not re.isLeaf)

* redefinition\_context\_valid

At least one of the redefinition contexts of the redefining element must be a specialization of at least one of the redefinition contexts for each redefined element. inv: redefinedElement->forAll(re | self.isRedefinitionContextValid(re))

**9.9.19 RedefinableTemplateSignature [Class]**

**9.9.19.1 Description**

A RedefinableTemplateSignature supports the addition of formal template parameters in a specialization of a template classifier.

**9.9.19.2 Diagrams**

Classifier Templates

**9.9.19.3 Generalizations**

RedefinableElement, TemplateSignature

**9.9.19.4 Association Ends**

* classifier : Classifier [1..1]{subsets RedefinableElement::redefinitionContext, redefines

TemplateSignature::template} (opposite Classifier::ownedTemplateSignature) The Classifier that owns this RedefinableTemplateSignature.

* extendedSignature : RedefinableTemplateSignature [0..\*]{subsets RedefinableElement::redefinedElement}

(opposite A\_extendedSignature\_redefinableTemplateSignature::redefinableTemplateSignature) The signatures extended by this RedefinableTemplateSignature.

* /inheritedParameter : TemplateParameter [0..\*]{subsets TemplateSignature::parameter} (opposite

A\_inheritedParameter\_redefinableTemplateSignature::redefinableTemplateSignature) The formal template parameters of the extended signatures.

**9.9.19.5 Operations**

* inheritedParameter() : TemplateParameter [0..\*]

Derivation for RedefinableTemplateSignature::/inheritedParameter

body: if extendedSignature->isEmpty() then Set{} else extendedSignature.parameter->asSet() endif

* isConsistentWith(redefiningElement : RedefinableElement) : Boolean {redefines

RedefinableElement::isConsistentWith()}

The query isConsistentWith() specifies, for any two RedefinableTemplateSignatures in a context in which redefinition is possible, whether redefinition would be logically consistent. A redefining template signature is always consistent with a redefined template signature, as redefinition only adds new formal parameters.

pre: redefiningElement.isRedefinitionContextValid(self) body: redefiningElement.oclIsKindOf(RedefinableTemplateSignature)

**9.9.19.6 Constraints**

* redefines\_parents

If any of the parent Classifiers are a template, then the extendedSignature must include the signature of that Classifier.

inv: classifier.allParents()->forAll(c | c.ownedTemplateSignature->notEmpty() implies self>closure(extendedSignature)->includes(c.ownedTemplateSignature))

**9.9.20 Slot [Class]**

**9.9.20.1 Description**

A Slot designates that an entity modeled by an InstanceSpecification has a value or values for a specific StructuralFeature.

**9.9.20.2 Diagrams**

Instances

**9.9.20.3 Generalizations**

Element

**9.9.20.4 Association Ends**

* definingFeature : StructuralFeature [1..1] (opposite A\_definingFeature\_slot::slot) The StructuralFeature that specifies the values that may be held by the Slot.
* owningInstance : InstanceSpecification [1..1]{subsets Element::owner} (opposite InstanceSpecification::slot) The InstanceSpecification that owns this Slot.
* ♦ value : ValueSpecification [0..\*]{ordered, subsets Element::ownedElement} (opposite

A\_value\_owningSlot::owningSlot)

The value or values held by the Slot.

**9.9.21 StructuralFeature [Abstract Class]**

**9.9.21.1 Description**

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

**9.9.21.2 Diagrams**

Features, Properties, Instances, Structural Feature Actions

**9.9.21.3 Generalizations**

MultiplicityElement, TypedElement, Feature

**9.9.21.4 Specializations**

Property

**9.9.21.5 Attributes**

 isReadOnly : Boolean [1..1] = false

If isReadOnly is true, the StructuralFeature may not be written to after initialization.

**9.9.22 Substitution [Class]**

**9.9.22.1 Description**

A substitution is a relationship between two classifiers signifying that the substituting classifier complies with the contract specified by the contract classifier. This implies that instances of the substituting classifier are runtime substitutable where instances of the contract classifier are expected.

**9.9.22.2 Diagrams**

Classifiers

**9.9.22.3 Generalizations**

Realization

**9.9.22.4 Association Ends**

* contract : Classifier [1..1]{subsets Dependency::supplier} (opposite A\_contract\_substitution::substitution) The contract with which the substituting classifier complies.
* substitutingClassifier : Classifier [1..1]{subsets Dependency::client, subsets Element::owner} (opposite

Classifier::substitution)

Instances of the substituting classifier are runtime substitutable where instances of the contract classifier are expected.

### 9.10 Association Descriptions

**9.10.1 A\_attribute\_classifier [Association]**

**9.10.1.1 Diagrams**

Classifiers

**9.10.1.2 Owned Ends**

* /classifier : Classifier [0..1]{union, subsets Feature::featuringClassifier, subsets RedefinableElement::redefinitionContext} (opposite Classifier::attribute)

**9.10.2 A\_bodyCondition\_bodyContext [Association]**

**9.10.2.1 Diagrams**

Operations

**9.10.2.2 Owned Ends**

* bodyContext : Operation [0..1]{subsets Constraint::context} (opposite Operation::bodyCondition)

**9.10.3 A\_classifier\_instanceSpecification [Association]**

**9.10.3.1 Diagrams**

Instances

**9.10.3.2 Specializations**

A\_classifier\_enumerationLiteral

**9.10.3.3 Owned Ends**

* instanceSpecification : InstanceSpecification [0..\*] (opposite InstanceSpecification::classifier)

**9.10.4 A\_classifier\_templateParameter\_parameteredElement [Association]**

**9.10.4.1 Diagrams**

Classifier Templates

**9.10.4.2 Member Ends**

* Classifier::templateParameter
* ClassifierTemplateParameter::parameteredElement

**9.10.5 A\_collaborationUse\_classifier [Association]**

**9.10.5.1 Diagrams**

Classifiers, Collaborations

**9.10.5.2 Specializations**

A\_representation\_classifier

**9.10.5.3 Owned Ends**

* classifier : Classifier [0..1]{subsets Element::owner} (opposite Classifier::collaborationUse)

**9.10.6 A\_condition\_parameterSet [Association]**

**9.10.6.1 Diagrams**

Features

**9.10.6.2 Owned Ends**

* parameterSet : ParameterSet [0..1]{subsets Element::owner} (opposite ParameterSet::condition)

**9.10.7 A\_constrainingClassifier\_classifierTemplateParameter [Association]**

**9.10.7.1 Diagrams**

Classifier Templates

**9.10.7.2 Owned Ends**

* classifierTemplateParameter : ClassifierTemplateParameter [0..\*] (opposite ClassifierTemplateParameter::constrainingClassifier)

**9.10.8 A\_contract\_substitution [Association]**

**9.10.8.1 Diagrams**

Classifiers

**9.10.8.2 Owned Ends**

* substitution : Substitution [0..\*]{subsets A\_supplier\_supplierDependency::supplierDependency} (opposite Substitution::contract)

**9.10.9 A\_defaultValue\_owningParameter [Association]**

**9.10.9.1 Diagrams**

Features

**9.10.9.2 Owned Ends**

* owningParameter : Parameter [0..1]{subsets Element::owner} (opposite Parameter::defaultValue)

**9.10.10 A\_defaultValue\_owningProperty [Association]**

**9.10.10.1 Diagrams**

Properties

**9.10.10.2 Owned Ends**

* owningProperty : Property [0..1]{subsets Element::owner} (opposite Property::defaultValue)

**9.10.11 A\_definingFeature\_slot [Association]**

**9.10.11.1 Diagrams**

Instances

**9.10.11.2 Owned Ends**

* slot : Slot [0..\*] (opposite Slot::definingFeature)

**9.10.12 A\_extendedSignature\_redefinableTemplateSignature [Association]**

**9.10.12.1 Diagrams**

Classifier Templates

**9.10.12.2 Owned Ends**

* redefinableTemplateSignature : RedefinableTemplateSignature [0..\*]{subsets

A\_redefinedElement\_redefinableElement::redefinableElement} (opposite

RedefinableTemplateSignature::extendedSignature)

**9.10.13 A\_feature\_featuringClassifier [Association]**

**9.10.13.1 Diagrams**

Classifiers, Features

**9.10.13.2 Member Ends**

* Classifier::feature
* Feature::featuringClassifier

**9.10.14 A\_general\_classifier [Association]**

**9.10.14.1 Diagrams**

Classifiers

**9.10.14.2 Owned Ends**

* classifier : Classifier [0..\*] (opposite Classifier::general)

**9.10.15 A\_general\_generalization [Association]**

**9.10.15.1 Diagrams**

Classifiers

**9.10.15.2 Owned Ends**

* generalization : Generalization [0..\*]{subsets A\_target\_directedRelationship::directedRelationship} (opposite Generalization::general)

**9.10.16 A\_generalizationSet\_generalization [Association]**

**9.10.16.1 Diagrams**

Classifiers, Generalization Sets

**9.10.16.2 Member Ends**

* Generalization::generalizationSet  GeneralizationSet::generalization

**9.10.17 A\_generalization\_specific [Association]**

**9.10.17.1 Diagrams**

Classifiers

**9.10.17.2 Member Ends**

* Classifier::generalization
* Generalization::specific

**9.10.18 A\_inheritedMember\_inheritingClassifier [Association]**

**9.10.18.1 Diagrams**

Classifiers

**9.10.18.2 Owned Ends**

* inheritingClassifier : Classifier [0..\*]{subsets A\_member\_memberNamespace::memberNamespace} (opposite Classifier::inheritedMember)

**9.10.19 A\_inheritedParameter\_redefinableTemplateSignature [Association]**

**9.10.19.1 Diagrams**

Classifier Templates

**9.10.19.2**

* redefinableTemplateSignature : RedefinableTemplateSignature [0..\*]{subsets

A\_parameter\_templateSignature::templateSignature} (opposite

RedefinableTemplateSignature::inheritedParameter)

**9.10.20 A\_instance\_instanceValue [Association]**

**9.10.20.1 Diagrams**

Instances

**9.10.20.2 Owned Ends**

* instanceValue : InstanceValue [0..\*] (opposite InstanceValue::instance)

**9.10.21 A\_method\_specification [Association]**

**9.10.21.1 Diagrams**

Features, Behaviors

**9.10.21.2 Member Ends**

* BehavioralFeature::method
* Behavior::specification

**9.10.22 A\_operation\_templateParameter\_parameteredElement [Association]**

**9.10.22.1 Diagrams**

Operations

**9.10.22.2 Member Ends**

* Operation::templateParameter
* OperationTemplateParameter::parameteredElement

**9.10.23 A\_opposite\_property [Association]**

**9.10.23.1 Diagrams**

Properties

**9.10.23.2 Owned Ends**

* property : Property [0..1] (opposite Property::opposite)

**9.10.24 A\_ownedParameterSet\_behavioralFeature [Association]**

**9.10.24.1 Diagrams**

Features

**9.10.24.2 Owned Ends**

* behavioralFeature : BehavioralFeature [0..1]{subsets NamedElement::namespace} (opposite BehavioralFeature::ownedParameterSet)

**9.10.25 A\_ownedParameter\_operation [Association]**

**9.10.25.1 Diagrams**

Operations

**9.10.25.2 Member Ends**

* Operation::ownedParameter
* Parameter::operation

**9.10.26 A\_ownedParameter\_ownerFormalParam [Association]**

**9.10.26.1 Diagrams**

Features

**9.10.26.2 Owned Ends**

* ownerFormalParam : BehavioralFeature [0..1]{subsets NamedElement::namespace} (opposite BehavioralFeature::ownedParameter)

**9.10.27 A\_ownedTemplateSignature\_classifier [Association]**

**9.10.27.1 Diagrams**

Classifier Templates

**9.10.27.2 Member Ends**

* Classifier::ownedTemplateSignature
* RedefinableTemplateSignature::classifier

**9.10.28 A\_ownedUseCase\_classifier [Association]**

**9.10.28.1 Diagrams**

Classifiers, Use Cases

**9.10.28.2**

* classifier : Classifier [0..1]{subsets NamedElement::namespace} (opposite Classifier::ownedUseCase)

**9.10.29 A\_parameterSet\_parameter [Association]**

**9.10.29.1 Diagrams**

Features

**9.10.29.2 Member Ends**

* Parameter::parameterSet  ParameterSet::parameter

**9.10.30 A\_postcondition\_postContext [Association]**

**9.10.30.1 Diagrams**

Operations

**9.10.30.2 Owned Ends**

* postContext : Operation [0..1]{subsets Constraint::context} (opposite Operation::postcondition)

**9.10.31 A\_powertypeExtent\_powertype [Association]**

**9.10.31.1 Diagrams**

Classifiers, Generalization Sets

**9.10.31.2 Member Ends**

* Classifier::powertypeExtent
* GeneralizationSet::powertype

**9.10.32 A\_precondition\_preContext [Association]**

**9.10.32.1 Diagrams**

Operations

**9.10.32.2 Owned Ends**

* preContext : Operation [0..1]{subsets Constraint::context} (opposite Operation::precondition)

**9.10.33 A\_qualifier\_associationEnd [Association]**

**9.10.33.1 Diagrams**

Properties, Associations

**9.10.33.2 Member Ends**

* Property::qualifier
* Property::associationEnd

**9.10.34 A\_raisedException\_behavioralFeature [Association]**

**9.10.34.1 Diagrams**

Features

**9.10.34.2 Owned Ends**

* behavioralFeature : BehavioralFeature [0..\*] (opposite BehavioralFeature::raisedException)

**9.10.35 A\_raisedException\_operation [Association]**

**9.10.35.1 Diagrams**

Operations

**9.10.35.2 Owned Ends**

* operation : Operation [0..\*]{subsets A\_raisedException\_behavioralFeature::behavioralFeature} (opposite Operation::raisedException)

**9.10.36 A\_redefinedClassifier\_classifier [Association]**

**9.10.36.1 Diagrams**

Classifiers

**9.10.36.2 Owned Ends**

* classifier : Classifier [0..\*]{subsets A\_redefinedElement\_redefinableElement::redefinableElement} (opposite Classifier::redefinedClassifier)

**9.10.37 A\_redefinedElement\_redefinableElement [Association]**

**9.10.37.1 Diagrams**

Classifiers

**9.10.37.2 Owned Ends**

* /redefinableElement : RedefinableElement [0..\*]{union} (opposite RedefinableElement::redefinedElement)

**9.10.38 A\_redefinedOperation\_operation [Association]**

**9.10.38.1 Diagrams**

Operations

**9.10.38.2**

* operation : Operation [0..\*]{subsets A\_redefinedElement\_redefinableElement::redefinableElement} (opposite Operation::redefinedOperation)

**9.10.39 A\_redefinedProperty\_property [Association]**

**9.10.39.1 Diagrams**

Properties

**9.10.39.2 Owned Ends**

* property : Property [0..\*]{subsets A\_redefinedElement\_redefinableElement::redefinableElement} (opposite Property::redefinedProperty)

**9.10.40 A\_redefinitionContext\_redefinableElement [Association]**

**9.10.40.1 Diagrams**

Classifiers

**9.10.40.2 Specializations**

A\_redefinitionContext\_transition, A\_redefinitionContext\_state, A\_redefinitionContext\_region

**9.10.40.3 Owned Ends**

* /redefinableElement : RedefinableElement [0..\*]{union} (opposite RedefinableElement::redefinitionContext)

**9.10.41 A\_representation\_classifier [Association]**

**9.10.41.1 Diagrams**

Classifiers, Collaborations

**9.10.41.2 Generalizations**

A\_collaborationUse\_classifier

**9.10.41.3 Owned Ends**

* classifier : Classifier [0..1]{redefines A\_collaborationUse\_classifier::classifier} (opposite Classifier::representation)

**9.10.42 A\_slot\_owningInstance [Association]**

**9.10.42.1 Diagrams**

Instances

**9.10.42.2 Member Ends**

* InstanceSpecification::slot
* Slot::owningInstance

**9.10.43 A\_specification\_owningInstanceSpec [Association]**

**9.10.43.1 Diagrams**

Instances

**9.10.43.2 Owned Ends**

* owningInstanceSpec : InstanceSpecification [0..1]{subsets Element::owner} (opposite InstanceSpecification::specification)

**9.10.44 A\_subsettedProperty\_property [Association]**

**9.10.44.1 Diagrams**

Properties

**9.10.44.2 Owned Ends**

* property : Property [0..\*] (opposite Property::subsettedProperty)

**9.10.45 A\_substitution\_substitutingClassifier [Association]**

**9.10.45.1 Diagrams**

Classifiers

**9.10.45.2 Member Ends**

* Classifier::substitution
* Substitution::substitutingClassifier

**9.10.46 A\_type\_operation [Association]**

**9.10.46.1 Diagrams**

Operations

**9.10.46.2 Owned Ends**

* operation : Operation [0..\*] (opposite Operation::type)

**9.10.47 A\_value\_owningSlot [Association]**

**9.10.47.1 Diagrams**

Instances

**9.10.47.2 Owned Ends**

* owningSlot : Slot [0..1]{subsets Element::owner} (opposite Slot::value)

## 10 Simple Classifiers

**10.1 Summary**

This clause specifies various kinds of Classifier that do not have complex internal structure.

### 10.2 DataTypes

**10.2.1 Summary**

DataTypes model Types whose instances are distinguished only by their value.

**10.2.2 Abstract Syntax**

**DataType**

**Enumeration**

**EnumerationLiteral**

**PrimitiveType**

**InstanceSpecification**

**Property**

**Operation**

**Classifier**

1

enumeration

+

\*

ownedLiteral

+

}

subsets namespace

{

ordered, subsets

{

ownedMember}

0..1

datatype

+

\*

ownedAttribute

+

subsets namespace,

{

subsets classifier}

ordered, subsets attribute,

{

subsets ownedMember}

0..1

+

datatype

\*

+

ownedOperation

subsets featuringClassifier,

{

subsets namespace, subsets

redefinitionContext}

{

ordered, subsets feature, subsets

redefinableElement, subsets

ownedMember}

\*

+

enumerationLiteral

1

+

/classifier

{

redefines instanceSpecification

}

{

readOnly, redefines classifier

}

**Figure 10.1 DataTypes**

**10.2.3 Semantics**

**10.2.3.1 DataTypes**

A DataType is a kind of Classifier. DataType differs from Class in that instances of a DataType are identified only by their value. All instances of a DataType with the same value are considered to be equal instances.

If a DataType has attributes (i.e., Properties owned by it and in its namespace) it is called a *structured* DataType. Instances of a structured DataType contain attribute values matching its attributes. Instances of a structured DataType are considered to be equal if and only if the structure is the same and the values of the corresponding attributes are equal.

A DataType may be parameterized, bound, and used as TemplateParameters.

**10.2.3.2 Primitive Types**

A PrimitiveType defines a predefined DataType, without any substructure. A PrimitiveType may have algebra and operations defined outside of UML, for example, mathematically. The run-time instances of a PrimitiveType are values that correspond to mathematical elements defined outside of UML (for example, the Integers).

**10.2.3.3 Enumerations**

Enumeration is a kind of DataType. Each value of an Enumeration corresponds to one of its user-defined EnumerationLiterals.

As a specialization of Classifier, Enumerations can participate in generalization relationships. An Enumeration that specializes another may define new EnumerationLiterals that are not defined in the generalizing Enumeration; in such a case the set of applicable literals comprises inherited literals plus locally-defined ones.

An EnumerationLiteral defines an element of the run-time extension of an Enumeration. Values corresponding to EnumerationLiterals are immutable and may be compared for equality. EnumerationLiterals may not change during their existence, so any attributes on an Enumeration shall be read-only.

An EnumerationLiteral has a name that shall be used to identify it within its Enumeration. The EnumerationLiteral name is scoped within and shall be unique within its Enumeration. EnumerationLiteral names shall be qualified for general use.

**10.2.4 Notation**

A DataType is designated using the Classifier notation (a rectangle) with keyword «dataType» or, when it is referenced (e.g., by an attribute), by the name of the DataType. A compartment listing the attributes is placed below the name compartment. A compartment listing the Operations is placed below the attribute compartment.

A PrimitiveType is similarly designated with the keyword «primitive» above or before the name of the PrimitiveType.

An Enumeration is similarly designated. The name of the Enumeration is placed in the upper compartment with the keyword «enumeration» above or before the name. A list of EnumerationLiterals may be placed, one to a line, in a compartment named “literals” below the operations compartment. The attributes and operations compartments may be suppressed, and typically are suppressed and empty.

**10.2.5 Examples**

Figure 10.2 illustrates the notation for defining a PrimitiveType.

«primitive» **Integer**

**Figure 10.2 PrimitiveType Notation**

Figure 10.3 illustrates the notation for defining DataTypes. The FullName type defined on the left is used as the type of the fullName attribute in the Person type defined on the right.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | «dataType» **FullName** | | firstName : String secondName : String  initial : String | | |  | | --- | | «dataType» **Person** | | fullName : FullName | |

**Figure 10.3 DataType Notation**

Figure 10.4 illustrates the notation for defining Enumerations.

|  |
| --- |
| «enumeration» **VisibilityKind** |
| public private protected package |

**Figure 10.4 Enumeration Notation**

### 10.3 Signals

**10.3.1 Summary**

Signals and Receptions are used to model asynchronous communication between objects.

**10.3.2 Abstract Syntax**

**Reception**

**BehavioralFeature**

**Signal**

**Classifier**

**Property**

\*

reception

+

1

signal

+

0..1

+

owningSignal

\*

+

ownedAttribute

subsets namespace,

{

subsets classifier}

ordered, subsets attribute,

{

subsets ownedMember}

**Figure 10.5 Signals**

**10.3.3 Semantics**

**10.3.3.1 Signals**

A Signal is a specification of a kind of communication between objects in which a reaction is asynchronously triggered in the receiver without a reply. The receiving object handles Signals as specified by clause 13.3. The data carried by the communication are represented as attributes of the Signal. A Signal is defined independently of the Classifiers handling it.

The sender of a Signal will not block waiting for a reply but continue execution immediately. By declaring a Reception associated to a given Signal, a Classifier specifies that its instances will be able to receive that Signal, or a subtype thereof, and will respond to it with the designated Behavior.

A Signal may be parameterized, bound, and used as TemplateParameters.

**10.3.3.2 Receptions**

A Reception specifies that its owning Class or Interface is prepared to react to the receipt of a Signal. A Reception matches a Signal if the received Signal is a specialization of the Reception’s signal. The details of how the object responds to the received Signal depend on the kind of Behavior associated with the Reception and its owning Class or Interface. See 13.2. The name of the Reception is the same as the name of the Signal. A Reception may only have in Parameters (see 9.4.3) that match the attributes of the Signal by name, type, and multiplicity.

**10.3.4 Notation**

A Signal is depicted by a Classifier symbol with the keyword «signal».

Receptions are shown in the receptions compartment using the same notation as for Operations with the keyword «signal».

**10.3.5 Examples**

Figure 10.6 shows an interface IAlarm that defines two Receptions, each referring to a Signal also shown in the example.

**NOTE.** The name of the Reception matches the name of the Signal, and the parameter of the Reception matches the attribute of the Signal.

|  |
| --- |
| «interface» **IAlarm** |
| «signal» Notify()  «signal» Activate() |

«signal» **Notify**

«signal»

**Activate**

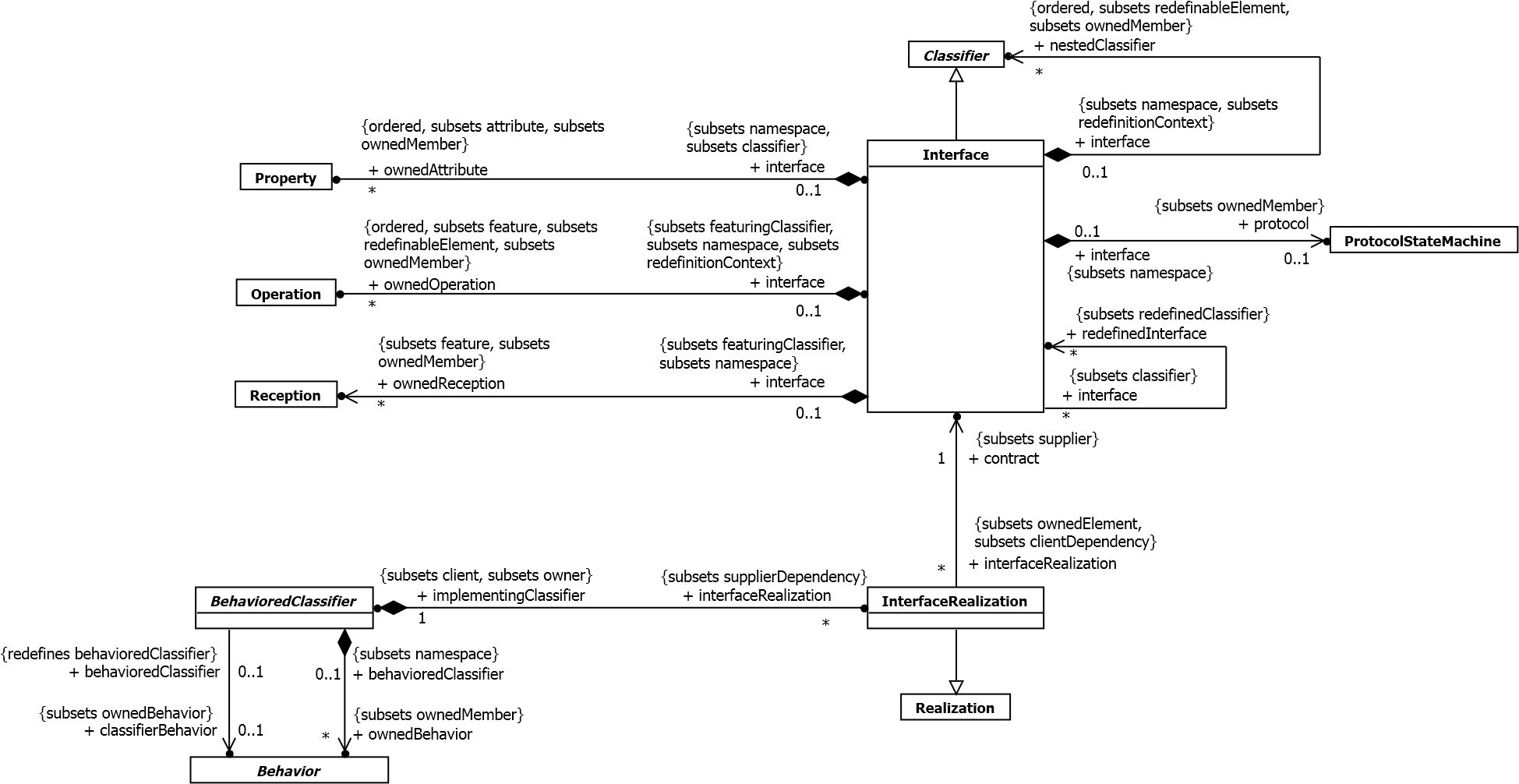
**Figure 10.6 Reception Notation**

### 10.4 Interfaces

**10.4.1 Summary**

Interfaces declare coherent services that are implemented by BehavioredClassifiers that implement the Interfaces via InterfaceRealizations.

**10.4.2 Abstract Syntax**



**Figure 10.7 Interfaces**

**10.4.3 Semantics**

**10.4.3.1 Interfaces**

An Interface is a kind of Classifier that represents a declaration of a set of public Features and obligations that together constitute a coherent service. An Interface specifies a contract; any instance of a Classifier that realizes the Interface shall fulfill that contract. The obligations associated with an Interface are in the form of constraints (such as pre- and postconditions) or protocol specifications, which may impose ordering restrictions on interactions through the Interface.

Interfaces may not be instantiated. Instead, an Interface specification is *implemented* or *realized* by a

BehavioredClassifier, which means that the BehavioredClassifier presents a public facade that conforms to the Interface specification.

**NOTE.** A given BehavioredClassifier may implement more than one Interface and that an Interface may be implemented by a number of different BehavioredClassifiers.

Interfaces provide a way to partition and characterize groups of public Features and obligations that realizing

BehavioredClassifiers shall possess. An Interface does not specify how it is to be implemented, but merely what needs to be supported by realizing BehavioredClassifiers. That is, such BehavioredClassifiers shall provide a public façade consisting of attributes, Operations, and externally observable Behavior that conforms to the Interface.

**NOTE.** If an Interface declares an attribute, this does not necessarily mean that the realizing BehavioredClassifier will necessarily have such an attribute in its implementation, but only that it will appear so to external observers.

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

Properties owned by Interfaces (including Association ends) imply that the realizing BehavioredClassifier should maintain information corresponding to the type and multiplicity of the Property and facilitate retrieval and modification of that information. A Property declared on an Interface does not necessarily imply that there will be such a Property on a realizing BehavioredClassifier (e.g., it may be realized by equivalent get and set Operations). Interfaces may also own constraints that impose constraints on the Features of the implementing BehavioredClassifier.

Interfaces may own a ProtocolStateMachine that specifies event sequences and pre/post conditions for the Operations and Receptions described by the Interface. A BehavioredClassifier realizing an Interface shall comply with the ProtocolStateMachine owned by the Interface.

An Interface may be parameterized, bound, and used as TemplateParameters.

An InterfaceRealization relationship between a BehavioredClassifier and an Interface implies that the

BehavioredClassifier conforms to the contract specified by the Interface by supporting the set of Features owned by the

Interface, and any of its parent Interfaces. For BehavioralFeatures, the implementing BehavioredClassifier will have an Operation or Reception for every Operation or Reception, respectively, defined by the Interface. For Properties, the realizing BehavioredClassifier will provide functionality that maintains the state represented by the Property. While such may be done by direct mapping to a Property of the realizing BehavioredClassifier, it may also be supported by the StateMachine of the BehavioredClassifier or by a pair of Operations that support the retrieval of the state information and an Operation that changes the state information.

**10.4.4 Notation**

An Interface may be designated using the default notation for Classifier (see 9.2.4) with the keyword «interface».

Alternatively an InterfaceRealization dependency from a BehavioredClassifier to an Interface may be shown by representing the Interface by a circle or *ball*, often also called *lollipop*, labeled with the name of the Interface, attached by a solid line to the BehavioredClassifier that realizes this Interface.

The Usage dependency from a Classifier to an Interface is shown by representing the Interface by a half-circle or *socket*, labeled with the name of the Interface, attached by a solid line to the Classifier that requires this Interface.

Interfaces inherited from a generalization of the BehavioredClassifier may be notated on a diagram through a lollipop. These Interfaces are indicated on the diagram by preceding the name of the Interface by a caret symbol. Earlier versions of UML permitted a forward slash preceding the name to indicate inherited Interfaces; this notation is permitted but discouraged.

If a Dependency is wired from a Usage to an InterfaceRealization that are represented using a socket and a lollipop, the dependency arrow may be shown joining the socket to the lollipop

**10.4.5 Examples**

The InterfaceRealization dependency from ProximitySensor to ISensor is shown using *ball* (*lollipop*) notation (see Figure 10.8).

**ProximitySensor**

**ISensor**

**Figure 10.8 ISensor is a provided Interface of ProximitySensor**

Figure 10.9 shows the lollipop notation for an inherited provided interface.

**P**

**r**

**o**

**x**

**i**

**m**

**i**

**t**

**y**

**S**

**e**

**n**

**s**

**o**

**r**

**CapacitiveSensor**

**^**

**ISensor**

**Figure 10.9 ISensor, a provided Interface of ProximitySensor, is shown as inherited by CapacitiveSensor** The Usage dependency from TheftAlarm to ISensor is shown using *socket* notation (see Figure 10.10).

**T**

**h**

**e**

**f**

**t**

**A**

**l**

**a**

**r**

**m**

.

.

.

Text

Text

**Figure 10.10 ISensor is a required Interface of TheftAlarm**

Alternatively, in cases where Interfaces are represented using the rectangle notation, InterfaceRealization and Usage dependencies are denoted with appropriate dependency arrows (see Figure 10.11). The Classifier at the tail of the arrow implements the Interface at the head of the arrow or uses that Interface, respectively.

«interface»

**ISensor**

activate ( )

read ( )

**ProximitySensor**

**TheftAlarm**

«use»

**Figure 10.11 Alternative notation for required and provided Interface**

It is often the case in practice that two or more Interfaces are mutually coupled through application-specific dependencies. In such situations, each Interface represents a specific role in a multi-party “protocol.” These types of protocol role couplings may be captured by Associations between Interfaces as shown in the example in Figure 10.12. This shows the specification of three Interfaces, *IAlarm*, *ISensor*, and *IBuzzer*. *IAlarm* and *ISensor* are shown as engaged in a bidirectional protocol, meaning that any implementation of ISensor must maintain the information needed to realize the theAlarm property, and similarly for IAlarm and theSensor. *IBuzzer* describes an Interface that implementers of *IAlarm* must be able to access.

«interface»

**IAlarm**

notify ( )

«interface»

**ISensor**

activate ( )

read ( )

«interface»

**IBuzzer**

volume : Integer

start ( )

reset ( )

\*

1

+

theBuzzer

1

+

theAlarm

1

+

theSensor

**Figure 10.12 A set of collaborating Interfaces**

### 10.5 Classifier Descriptions

**10.5.1 BehavioredClassifier [Abstract Class]**

**10.5.1.1 Description**

A BehavioredClassifier may have InterfaceRealizations, and owns a set of Behaviors one of which may specify the behavior of the BehavioredClassifier itself.

**10.5.1.2 Diagrams**

Interfaces, Use Cases, Classes, Collaborations, Behaviors

**10.5.1.3 Generalizations**

Classifier

**10.5.1.4 Specializations**

Actor, UseCase, Class, Collaboration

**10.5.1.5 Association Ends**

* classifierBehavior : Behavior [0..1]{subsets BehavioredClassifier::ownedBehavior} (opposite

A\_classifierBehavior\_behavioredClassifier::behavioredClassifier)

A Behavior that specifies the behavior of the BehavioredClassifier itself.

* ♦ interfaceRealization : InterfaceRealization [0..\*]{subsets Element::ownedElement, subsets

NamedElement::clientDependency} (opposite InterfaceRealization::implementingClassifier)

The set of InterfaceRealizations owned by the BehavioredClassifier. Interface realizations reference the Interfaces of which the BehavioredClassifier is an implementation.

* ♦ ownedBehavior : Behavior [0..\*]{subsets Namespace::ownedMember} (opposite

A\_ownedBehavior\_behavioredClassifier::behavioredClassifier) Behaviors owned by a BehavioredClassifier.

**10.5.1.6 Constraints**

* class\_behavior

If a behavior is classifier behavior, it does not have a specification.

inv: classifierBehavior->notEmpty() implies classifierBehavior.specification->isEmpty()

**10.5.2 DataType [Class]**

**10.5.2.1 Description**

A DataType is a type whose instances are identified only by their value.

**10.5.2.2 Diagrams**

DataTypes, Properties, Operations

**10.5.2.3 Generalizations**

Classifier

**10.5.2.4 Specializations**

Enumeration, PrimitiveType

**10.5.2.5 Association Ends**

* ♦ ownedAttribute : Property [0..\*]{ordered, subsets Classifier::attribute, subsets Namespace::ownedMember}

(opposite Property::datatype)

The attributes owned by the DataType.

* ♦ ownedOperation : Operation [0..\*]{ordered, subsets Classifier::feature, subsets

A\_redefinitionContext\_redefinableElement::redefinableElement, subsets Namespace::ownedMember} (opposite Operation::datatype)

The Operations owned by the DataType.

**10.5.3 Enumeration [Class]**

**10.5.3.1 Description**

An Enumeration is a DataType whose values are enumerated in the model as EnumerationLiterals.

**10.5.3.2 Diagrams**

DataTypes

**10.5.3.3 Generalizations**

DataType

**10.5.3.4 Association Ends**

* ♦ ownedLiteral : EnumerationLiteral [0..\*]{ordered, subsets Namespace::ownedMember} (opposite

EnumerationLiteral::enumeration)

The ordered set of literals owned by this Enumeration.

**10.5.3.5 Constraints**

* immutable

inv: ownedAttribute->forAll(isReadOnly)

**10.5.4 EnumerationLiteral [Class]**

**10.5.4.1 Description**

An EnumerationLiteral is a user-defined data value for an Enumeration.

**10.5.4.2 Diagrams**

DataTypes

**10.5.4.3 Generalizations**

InstanceSpecification

**10.5.4.4 Association Ends**

* /classifier : Enumeration [1..1]{redefines InstanceSpecification::classifier} (opposite

A\_classifier\_enumerationLiteral::enumerationLiteral)

The classifier of this EnumerationLiteral derived to be equal to its Enumeration.

* enumeration : Enumeration [1..1]{subsets NamedElement::namespace} (opposite Enumeration::ownedLiteral) The Enumeration that this EnumerationLiteral is a member of.

**10.5.4.5 Operations**

* classifier() : Enumeration

Derivation of Enumeration::/classifier

body: enumeration

**10.5.5 Interface [Class]**

**10.5.5.1 Description**

Interfaces declare coherent services that are implemented by BehavioredClassifiers that implement the Interfaces via InterfaceRealizations.

**10.5.5.2 Diagrams**

Interfaces, Encapsulated Classifiers, Components, Properties, Operations

**10.5.5.3 Generalizations**

Classifier

**10.5.5.4 Association Ends**

* ♦ nestedClassifier : Classifier [0..\*]{ordered, subsets

A\_redefinitionContext\_redefinableElement::redefinableElement, subsets Namespace::ownedMember}

(opposite A\_nestedClassifier\_interface::interface)

References all the Classifiers that are defined (nested) within the Interface.

* ♦ ownedAttribute : Property [0..\*]{ordered, subsets Classifier::attribute, subsets Namespace::ownedMember}

(opposite Property::interface)

The attributes (i.e., the Properties) owned by the Interface.

* ♦ ownedOperation : Operation [0..\*]{ordered, subsets Classifier::feature, subsets

A\_redefinitionContext\_redefinableElement::redefinableElement, subsets Namespace::ownedMember} (opposite Operation::interface)

The Operations owned by the Interface.

* ♦ ownedReception : Reception [0..\*]{subsets Classifier::feature, subsets Namespace::ownedMember}

(opposite A\_ownedReception\_interface::interface)

Receptions that objects providing this Interface are willing to accept.

* ♦ protocol : ProtocolStateMachine [0..1]{subsets Namespace::ownedMember} (opposite A\_protocol\_interface::interface)

References a ProtocolStateMachine specifying the legal sequences of the invocation of the BehavioralFeatures described in the Interface.

* redefinedInterface : Interface [0..\*]{subsets Classifier::redefinedClassifier} (opposite

A\_redefinedInterface\_interface::interface)

References all the Interfaces redefined by this Interface.

**10.5.5.5 Constraints**

* visibility

The visibility of all Features owned by an Interface must be public. inv: feature->forAll(visibility = VisibilityKind::public)

**10.5.6 InterfaceRealization [Class]**

**10.5.6.1 Description**

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

**10.5.6.2 Diagrams**

Interfaces

**10.5.6.3 Generalizations**

Realization

**10.5.6.4 Association Ends**

* contract : Interface [1..1]{subsets Dependency::supplier} (opposite

A\_contract\_interfaceRealization::interfaceRealization)

References the Interface specifying the conformance contract.

* implementingClassifier : BehavioredClassifier [1..1]{subsets Dependency::client, subsets Element::owner}

(opposite BehavioredClassifier::interfaceRealization)

References the BehavioredClassifier that owns this InterfaceRealization, i.e., the BehavioredClassifier that realizes the Interface to which it refers.

**10.5.7 PrimitiveType [Class]**

**10.5.7.1 Description**

A PrimitiveType defines a predefined DataType, without any substructure. A PrimitiveType may have an algebra and operations defined outside of UML, for example, mathematically.

**10.5.7.2 Diagrams**

DataTypes

**10.5.7.3 Generalizations**

DataType

**10.5.8 Reception [Class]**

**10.5.8.1 Description**

A Reception is a declaration stating that a Classifier is prepared to react to the receipt of a Signal.

**10.5.8.2 Diagrams**

Signals, Interfaces, Classes

**10.5.8.3 Generalizations**

BehavioralFeature

**10.5.8.4 Association Ends**

* signal : Signal [1..1] (opposite A\_signal\_reception::reception) The Signal that this Reception handles.

**10.5.8.5 Constraints**

* same\_name\_as\_signal

A Reception has the same name as its signal

inv: name = signal.name

* same\_structure\_as\_signal

A Reception's parameters match the ownedAttributes of its signal by name, type, and multiplicity

inv: signal.ownedAttribute->size() = ownedParameter->size() and

Sequence{1..signal.ownedAttribute->size()}->forAll( i |

ownedParameter->at(i).direction = ParameterDirectionKind::\_'in' and ownedParameter->at(i).name = signal.ownedAttribute->at(i).name and ownedParameter->at(i).type = signal.ownedAttribute->at(i).type and

ownedParameter->at(i).lowerBound() = signal.ownedAttribute->at(i).lowerBound() and ownedParameter->at(i).upperBound() = signal.ownedAttribute->at(i).upperBound() )

**10.5.9 Signal [Class]**

**10.5.9.1 Description**

A Signal is a specification of a kind of communication between objects in which a reaction is asynchronously triggered in the receiver without a reply.

**10.5.9.2 Diagrams**

Signals, Events, Invocation Actions

**10.5.9.3 Generalizations**

Classifier

**10.5.9.4 Association Ends**

 ♦ ownedAttribute : Property [0..\*]{ordered, subsets Classifier::attribute, subsets Namespace::ownedMember}

(opposite A\_ownedAttribute\_owningSignal::owningSignal) The attributes owned by the Signal.

### 10.6 Association Descriptions

**10.6.1 A\_classifierBehavior\_behavioredClassifier [Association]**

**10.6.1.1 Diagrams**

Interfaces, Behaviors

**10.6.1.2 Generalizations**

A\_ownedBehavior\_behavioredClassifier

**10.6.1.3 Owned Ends**

* behavioredClassifier : BehavioredClassifier [0..1]{redefines

A\_ownedBehavior\_behavioredClassifier::behavioredClassifier} (opposite BehavioredClassifier::classifierBehavior)

**10.6.2 A\_classifier\_enumerationLiteral [Association]**

**10.6.2.1 Diagrams**

DataTypes

**10.6.2.2 Generalizations**

A\_classifier\_instanceSpecification

**10.6.2.3 Owned Ends**

* enumerationLiteral : EnumerationLiteral [0..\*]{redefines

A\_classifier\_instanceSpecification::instanceSpecification} (opposite EnumerationLiteral::classifier)

**10.6.3 A\_contract\_interfaceRealization [Association]**

**10.6.3.1 Diagrams**

Interfaces

**10.6.3.2 Owned Ends**

* interfaceRealization : InterfaceRealization [0..\*]{subsets

A\_supplier\_supplierDependency::supplierDependency} (opposite InterfaceRealization::contract)

**10.6.4 A\_interfaceRealization\_implementingClassifier [Association]**

**10.6.4.1 Diagrams**

Interfaces

**10.6.4.2 Member Ends**

* BehavioredClassifier::interfaceRealization
* InterfaceRealization::implementingClassifier

**10.6.5 A\_nestedClassifier\_interface [Association]**

**10.6.5.1 Diagrams**

Interfaces

**10.6.5.2 Owned Ends**

* interface : Interface [0..1]{subsets NamedElement::namespace, subsets

RedefinableElement::redefinitionContext} (opposite Interface::nestedClassifier)

**10.6.6 A\_ownedAttribute\_datatype [Association]**

**10.6.6.1 Diagrams**

DataTypes, Properties

**10.6.6.2 Member Ends**

* DataType::ownedAttribute
* Property::datatype

**10.6.7 A\_ownedAttribute\_interface [Association]**

**10.6.7.1 Diagrams**

Interfaces, Properties

**10.6.7.2 Member Ends**

* Interface::ownedAttribute
* Property::interface

**10.6.8 A\_ownedAttribute\_owningSignal [Association]**

**10.6.8.1 Diagrams**

Signals

**10.6.8.2 Owned Ends**

* owningSignal : Signal [0..1]{subsets NamedElement::namespace, subsets A\_attribute\_classifier::classifier} (opposite Signal::ownedAttribute)

**10.6.9 A\_ownedBehavior\_behavioredClassifier [Association]**

**10.6.9.1 Diagrams**

Interfaces, Behaviors

**10.6.9.2 Specializations**

A\_classifierBehavior\_behavioredClassifier

**10.6.9.3 Owned Ends**

* behavioredClassifier : BehavioredClassifier [0..1]{subsets NamedElement::namespace} (opposite BehavioredClassifier::ownedBehavior)

**10.6.10 A\_ownedLiteral\_enumeration [Association]**

**10.6.10.1 Diagrams**

DataTypes

**10.6.10.2 Member Ends**

* Enumeration::ownedLiteral
* EnumerationLiteral::enumeration

**10.6.11 A\_ownedOperation\_datatype [Association]**

**10.6.11.1 Diagrams**

DataTypes, Operations

**10.6.11.2 Member Ends**

* DataType::ownedOperation
* Operation::datatype

**10.6.12 A\_ownedOperation\_interface [Association]**

**10.6.12.1 Diagrams**

Interfaces, Operations

**10.6.12.2 Member Ends**

* Interface::ownedOperation
* Operation::interface

**10.6.13 A\_ownedReception\_interface [Association]**

**10.6.13.1 Diagrams**

Interfaces

**10.6.13.2 Owned Ends**

* interface : Interface [0..1]{subsets Feature::featuringClassifier, subsets NamedElement::namespace} (opposite Interface::ownedReception)

**10.6.14 A\_protocol\_interface [Association]**

**10.6.14.1 Diagrams**

Interfaces

**10.6.14.2 Owned Ends**

* interface : Interface [0..1]{subsets NamedElement::namespace} (opposite Interface::protocol) Specifies the namespace in which the protocol state machine is defined.

**10.6.15 A\_redefinedInterface\_interface [Association]**

**10.6.15.1 Diagrams**

Interfaces

**10.6.15.2 Owned Ends**

* interface : Interface [0..\*]{subsets A\_redefinedClassifier\_classifier::classifier} (opposite Interface::redefinedInterface)

**10.6.16 A\_signal\_reception [Association]**

**10.6.16.1 Diagrams**

Signals

**10.6.16.2 Owned Ends**

* reception : Reception [0..\*] (opposite Reception::signal)

## 11 Structured Classifiers

### 11.1 Summary

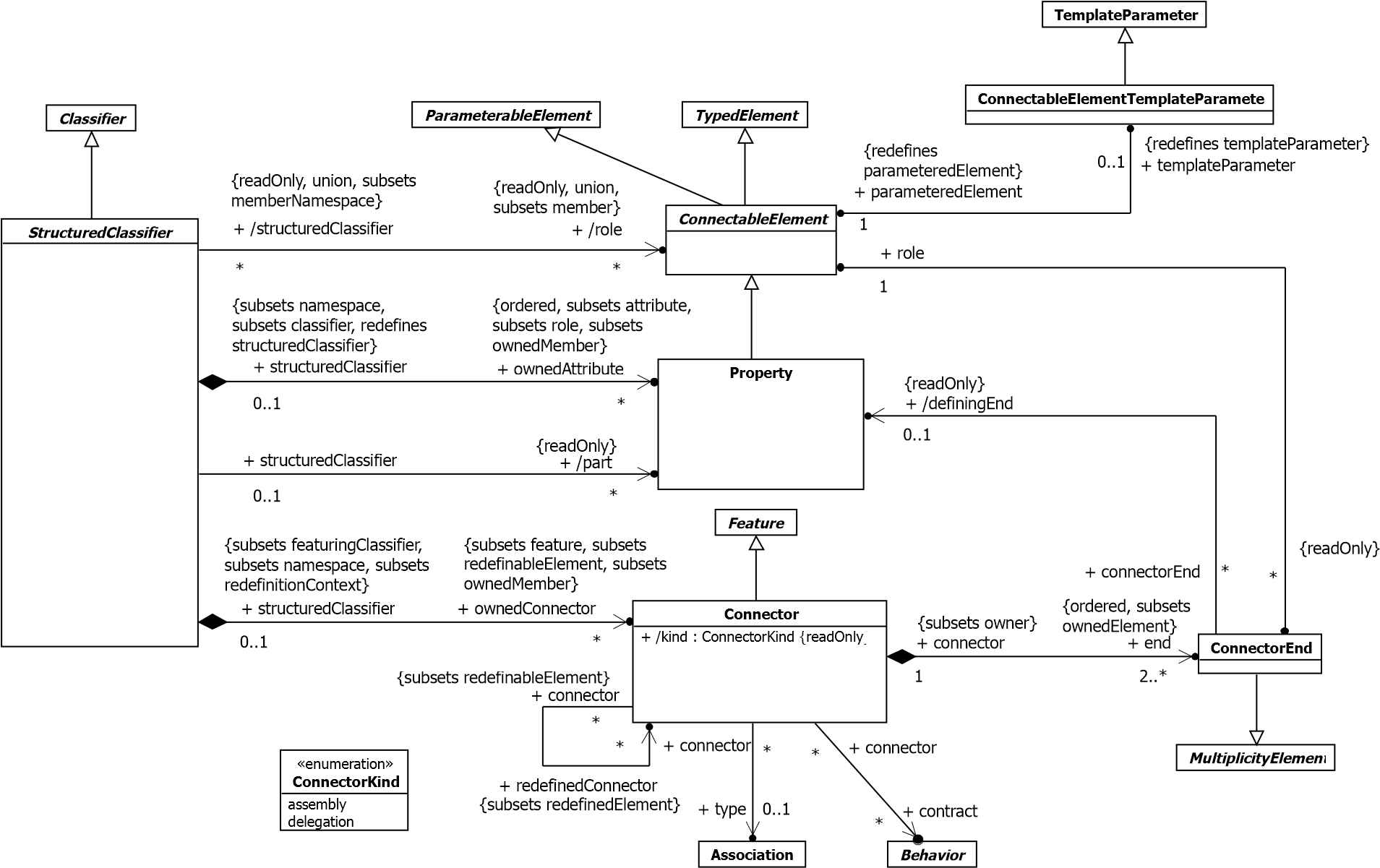
StructuredClassifiers are Classifiers that may have an internal structure comprising a network of linked roles (which can themselves be instances of structured classifiers) and an external structure consisting of one or more Ports. The Ports of EncapsulatedClassifiers act as local agents of remote collaborators, allowing EncapsulatedClassifiers to differentiate between them but without being directly coupled to them. Classes, Components, Associations and Collaborations are concrete metaclasses that use these capabilities.

### 11.2 Structured Classifiers

**11.2.1 Summary**

StructuredClassifiers may contain an internal structure of connected elements each of which plays a role in the overall behavior modeled by the StructuredClassifier. It may be helpful to read this sub clause in conjunction with sub clause 11.5 - Associations.

**11.2.2 Abstract Syntax**

+ /end

**Figure 11.1 Structured Classifiers**

**11.2.3 Semantics**

**11.2.3.1 Connectable Elements**

ConnectableElement is an abstract metaclass. Each ConnectableElement represents a participant within the internal structure of a StructuredClassifier; these participants are called roles. Roles may be joined by Connectors, and specify configurations of linked instances contained or referenced within an instance of the containing StructuredClassifier.

The detailed semantics of ConnectableElement is given by its concrete subtypes. In general, each ConnectableElement exhibits a set of *effective required Interfaces* and a set of *effective provided Interfaces*. These sets are used to determine the connectability of ConnectableElements using Connectors, see below.

For ConnectableElements except delegating Ports (see 11.3.3) the effective required Interfaces are the required Interfaces, and the effective provided Interfaces are the provided Interfaces, derived as follows:

* The provided Interfaces comprises the union of the sets of Interfaces realized by the type of the

ConnectableElement and its supertypes, or the set containing just its type if it is typed by an Interface.

* The required Interfaces comprises the union of the sets of Interfaces used by the type of the ConnectableElement and its supertypes.

A ConnectableElement may be exposed via a ConnectableElementTemplateParameter as a formal parameter for a template. The semantics and notation for this are only defined when the ConnectableElement is a Property (see the semantics and notation for Property in 9.5).

**11.2.3.2 Parts and Roles**

The Properties of a StructuredClassifier obey the semantics of Property specified in 9.5.

Property is a kind of ConnectableElement. All of the ownedAttributes of a StructuredClassifier are roles and can be connected using Connectors.

Those ownedAttributes of a StructuredClassifier that have isComposite = true (see 9.5.3) are called its parts. Hence parts constitute a subset of roles.

**11.2.3.3 Connectors**

A Connector specifies *links* (see 11.5 Associations) between two or more instances playing owned or inherited roles within a StructuredClassifier. Each link may be realized by something as simple as a pointer or by something as complex as a network connection, and may represent the possibility of instances being able to communicate because their identities are known by virtue of being passed in as parameters, held in variables or slots, or even because the communicating instances are the same instance.

In contrast to Associations, which specify links between any suitably-typed instance of the associated Classifiers, Connectors specify links between instances playing the connected roles only.

Each Connector may be attached to two or more ConnectableElements, each representing a set of instances that contribute to the instantiation of the containing StructuredClassifier.

A ConnectorEnd is an endpoint of a Connector, which attaches the Connector to a ConnectableElement.

Links corresponding to Connectors may be created upon the creation of the instance of the containing

StructuredClassifier. All such links are destroyed when the containing StructuredClassifier instance is destroyed.

A Connector may be typed by an Association, in which case the links specified by the Connector are instances of the typing Association.

Each feature of each of the *effective required Interfaces* of each ConnectableElement at the end of a Connector must have at least one compatible feature among the features of the *effective provided Interfaces* of ConnectableElements at the other ends. One feature is compatible with another at least in the cases when the two features are the same or when they are both properties or operations and the second feature is a redefinition of the first. However, conforming tools may allow additional cases of compatible features beyond this.

When there are multiple connectors attached to a single ConnectableElement, the semantics are the same as a single nary Connector connecting the ConnectableElement to all of the ConnectableElements connected via the multiple connectors.

Connectors have a kind, whose value is *assembly* or *delegation*. The semantics of *delegation* connectors are only related to Ports and described under Port (see 11.3). All other Connectors are *assembly* connectors.

ConnectorKind is an enumeration of the following literal values:

assembly Indicates that the Connector is an assembly Connector.

delegation Indicates that the Connector is a delegation Connector.

Behaviors may be associated with Connectors as contracts to specify valid interaction patterns across the Connector.

**11.2.3.4 Multiplicities and topologies**

The multiplicities on ConnectableElements constrain the number of objects that may be created within an instance of the containing StructuredClassifier, according to the semantics of MultiplicityElement (see 7.5.3).

For a binary Connector, the ConnectorEnd’s multiplicity indicates the number of instances that may be linked to each instance of the ConnectableElement on the other end. For an n-ary Connector, the multiplicity of one end constrains the number of links that may refer to a set containing one particular instance for each of the other ends.

When an instance is removed from a role of an instance of a StructuredClassifier, links that exist due to Connectors between that role and others are destroyed.

The topologies that result from matching the multiplicities of ConnectorEnds and those of ConnectableElements they interconnect cannot always be deduced from the model. Specific examples in which the topology can be determined from the multiplicities are shown in Figure 11.6 and Figure 11.7.

**11.2.4 Notation**

The internal structure of a StructuredClassifier is shown in a separate compartment with the name “internal structure.” This compartment is mandatory: all tools that conform to the concrete syntax of UML must implement it. The internal structure compartment contains symbols representing the roles and connectors. The internal structure compartment appears below the attributes and operations compartments.

A part may be shown by graphical nesting of a box symbol with a solid outline representing the part within the internal structure compartment. A role that is not a composition may be shown by graphical nesting of a box symbol with a dashed outline. In either case the box may be called a *part box*, even though strictly-speaking only the compositions are parts. Lollipop and socket symbols may optionally be shown to indicate the provided and required interfaces of the part, using the same notation as for the definition of the part’s type (see 10.4.4).

The part box symbol has a name compartment, which contains a string according to the syntax defined in sub clause 9.5.4. Detail may also be shown within the part box indicating specific values for Properties of the part’s type when instances corresponding to the Property are created.

The multiplicity for a Property may also be shown as a multiplicity mark in the top right corner of the part box.

When a role is typed by an EncapsulatedClassifier (see 11.3), any Ports of the type may also be shown as small square symbols overlapping the boundary of the part box denoting the role. The name of the Port is shown near the Port; the multiplicity follows the name surrounded by square brackets. Name and multiplicity may be elided. Lollipop and socket symbols may optionally be shown to indicate the provided and required interfaces of the Port, using the same notation as for the Port’s definition (see 11.3.4).

If a role is typed by a classifier other than Class, the name compartment of the part box symbol contains the appropriate keyword (e.g., «component») above the name. For some kinds of Classifiers, optionally in the right hand corner an icon denoting the kind of Classifier can be displayed.

A Connector is drawn using similar notation to that for Association (see 11.5.4). The optional name string of the Connector obeys the following syntax:

*<connector> ::= ( [<name> ] ’:’ <associationname> ) | ([<name> ] ’:’ <associationclassname> ) | [<name> ]*

where <*name>* is the name of the Connector, and *<associationname>* or *<associationclassname>* is the name of the Association or AssociationClass, respectively, that is its type. A stereotype keyword within guillemets may be placed above or in front of the Connector name. A property string may be placed after or below the Connector name.

Adornments may be shown on the ConnectorEnd using the same notation as adornments on Association ends. If no multiplicity is shown, the multiplicity matches the multiplicity of the role the end is attached to.

If a ConnectorEnd is attached to a Port on a part or role of the internal structure and no multiplicity is shown, the multiplicity of the ConnectorEnd matches the multiplicity of the Port multiplied by the multiplicity of the role (if any).

The notational specifications in the next three paragraphs are optional: a conforming tool does not need to implement them. They are useful for scalability in complex systems.

If the parts have *simple* Ports (Ports with a single required or provided Interface), then *ball-and-socket notation* may be used to represent assembly Connectors between those Ports. Ball-and-socket notation may not be used to connect complex (i.e., non-simple) Ports or parts without Ports.

When connecting simple Ports, normal Connector notation for assembly or delegation may be shown connected to the ball or socket symbol rather than to the Port symbol itself.

When there is an n-ary Connector connecting more than two simple Ports, and two or more of the Ports provide or require the same or compatible Interfaces, a single symbol representing the Interface can be shown, and lines from the Components can be drawn to that symbol, in a “channeled ball-and-socket” notation.

An internal structure compartment may also contain symbols representing CollaborationUses, following the notation described in 11.7.4.

**11.2.5 Examples**



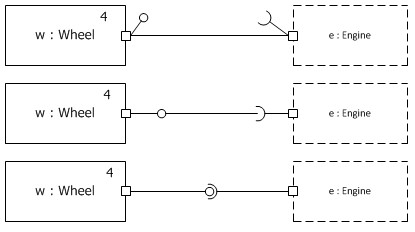
**Figure 11.2 Parts and roles**

Figure 11.2 shows examples of part boxes. On the left, the part box denotes that the containing instance will own four instances of the *Wheel* class by composition. The multiplicity is shown in the corner of the part box. The part box on the right is not composite, and denotes that the containing instance will reference one or two instances of the *Engine* class.



**Figure 11.3 Parts and roles with Ports**

Figure 11.3 shows examples of part boxes for properties typed by EncapsulatedClassifiers with Ports, in this case simple Ports. The notation for more complex Ports can also be used.

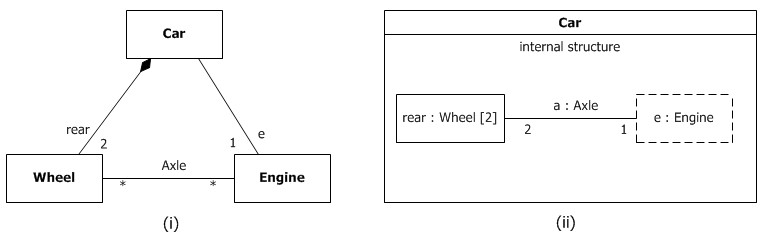


**Figure 11.4 Alternative notations for connecting parts and roles with Ports**

Figure 11.4 shows three alternative notations for connecting simple Ports on the parts and roles within a

StructuredClassifier. In the top example, the connector is joined to the Port symbols themselves. This is the only mandatory notation for connecting Ports in an internal structure. The lollipops and sockets indicate the provided and required interfaces of the Ports; their appearance is optional.

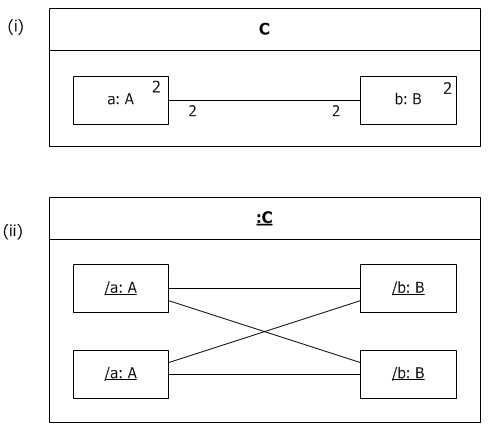
In the second example, the connector line is attached to the ball and socket symbols; in the third example, ball-andsocket notation is used. These notations correspond to the same model as the top example.



**Figure 11.5 Associations compared with Connectors**

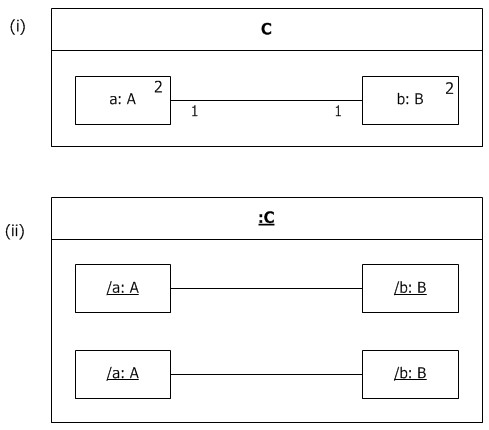
Figure 11.5 shows two possible views of the *Car* Class. In subfigure (i), *Car* is shown as having a composition Association with role name *rear* to a class *Wheel* and an Association with role name *e* to a class *Engine*. In subfigure (ii), the same is specified. However, in addition, in subfigure (ii) it is specified that *rear* and *e* belong to the internal structure of the class *Car*. This allows specification of detail that holds only for instances of the *Wheel* and *Engine* classes within the context of the class *Car*, but which will not hold for wheels and engines in general. For example, subfigure (i) specifies that any instance of class *Engine* can be linked to an arbitrary number of instances of class *Wheel*. Subfigure (ii), however, specifies that within the context of class *Car*, the instance playing the role of *e* may only be connected to two instances playing the role of *rear*. In addition, the instances playing the *e* and *rear* roles may only be linked if they are roles of the same instance of class *Car*. In other words, subfigure (ii) asserts additional constraints on the instances of the classes *Wheel* and *Engine*, when they are playing the respective roles within an instance of class *Car*. These constraints are not true for instances of *Wheel* and *Engine* in general. Other wheels and engines may be arbitrarily linked as specified in subfigure (i).

For each instance playing a role in an internal structure, there will initially be as many links as indicated by the lower multiplicity of the opposite ends of Connectors attached to that role. If the multiplicities of the ends match the multiplicities of the roles they are attached to as defined in Figure 11.6 (i), the initial configuration that will be created when an instance of the containing StructuredClassifier is created consists of the set of instances corresponding to the roles (as specified by the multiplicities on the roles) fully connected by links; see the resultant instance shown in Figure 11.6 (ii).



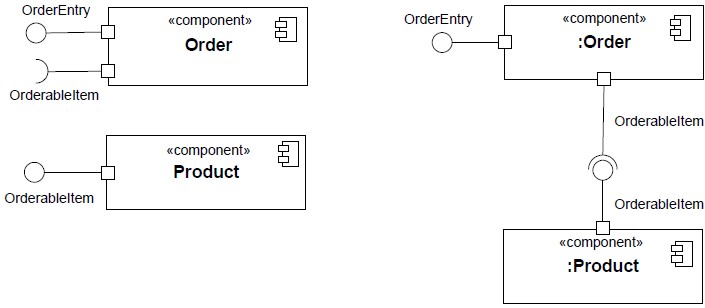
**Figure 11.6 "Star" Connector pattern**

Links will be created for each instance playing the connected roles according to their ordering until the minimum ConnectorEnd multiplicity is reached for both ends of the Connector; see the resultant instance in Figure 11.7 (ii). In this example, only two links are created.



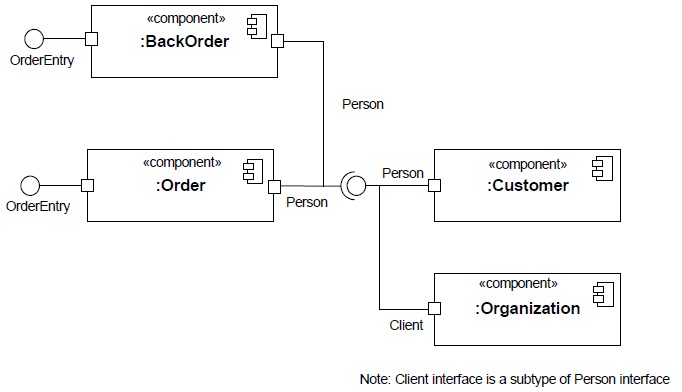
**Figure 11.7 "Array" Connector pattern**

Figure 11.8 shows example notation for parts typed by Components with simple Ports (Ports with only one interface), and the optional ball-and-socket notation to represent an assembly Connector between compatible Ports. The Component definitions are on the left and the corresponding parts on the right.



**Figure 11.8 An assembly Connector maps a simple Port of a Component to a matching simple Port of another Component.**

Figure 11.9 shows “channeled ball-and-socket notation” for a 4-ary Connector. The two simple Ports that require Person have been channeled into a single socket, and the two simple Ports that provide Person (either directly or indirectly) have been channeled into a single ball.



**Figure 11.9 An n-ary Connector that assembles four simple Ports using channeled ball-and-socket notation.**

### 11.3 Encapsulated Classifiers

**11.3.1 Summary**

EncapsulatedClassifier extends StructuredClassifier with the ability to own Ports, a mechanism for isolating an EncapsulatedClassifier from its environment.

**11.3.2 Abstract Syntax**

**EncapsulatedClassifier**

**Port**

+

isBehavior : Boolean = false

+

isConjugated : Boolean = false

+

isService : Boolean = true

**Interface**

**StructuredClassifier**

**Property**

**ConnectorEnd**

**ProtocolStateMachine**

\*

port

+

\*

+

redefinedPort

}

subsets property

{

subsets redefinedProperty

}

{

0..1

encapsulatedClassifier

+

\*

/ownedPort

+

}

subsets structuredClassifier

{

{

readOnly, subsets ownedAttribute

}

\*

+

port

\*

/provided

+

{

readOnly

}

\*

+

port

\*

/required

+

{

readOnly

}

\*

+

connectorEnd

0..1

+

partWithPort

\*

+

port

0..1

protocol

+

**Figure 11.10 Encapsulated Classifiers**

**11.3.3 Semantics**

**11.3.3.1 Ports**

Ports represent interaction points through which an EncapsulatedClassifier communicates with its environment. Multiple Ports can be defined for an EncapsulatedClassifier, enabling different communications to be distinguished based on the Port through which they occur. By decoupling the internals of the EncapsulatedClassifier from its environment, Ports allow an EncapsulatedClassifier to be defined independently of its environment, making it reusable in any environment that conforms to the constraints imposed by its Ports.

A Port is a Property of an EncapsulatedClassifier that specifies a distinct interaction point between that

EncapsulatedClassifier and its environment or between the Behavior of the EncapsulatedClassifier and its internal roles.

Ports are connected by Connectors through which requests can be made to invoke the BehavioralFeatures of an EncapsulatedClassifier. A Port may specify the services an EncapsulatedClassifier provides (offers) to its environment as well as the services that an EncapsulatedClassifier expects (requires) of its environment.

The property isService, when true, indicates that this Port is used to provide the published functionality of an

EncapsulatedClassifier. If false, this Port is used to implement the EncapsulatedClassifier but is not part of the essential externally-visible functionality of the EncapsulatedClassifier and can, therefore, be altered or deleted along with the internal implementation of the EncapsulatedClassifier and other properties that are considered part of its implementation.

The phrase *Port on Part* or more generally *Port on Property* signifies the situation where a Property playing a role in a

StructuredClassifier is typed by an EncapsulatedClassifier that has Ports. A Connector within the containing

StructuredClassifier may be connected to one of these Ports. In such a case, the property partWithPort of the applicable ConnectorEnd references the actual Property being connected: in general, there might be many Properties in the structure typed by the same EncapsulatedClassifier, and partWithPort is used to signify the right one.

The Interfaces associated with a Port specify the nature of the interactions that may occur over it. The required Interfaces of a Port characterize the requests that may be made from the EncapsulatedClassifier to its environment through this Port. Instances of this EncapsulatedClassifier expect that the Features owned by its required Interfaces will be offered by one or more instances in its environment. The provided Interfaces of a Port characterize requests to the

EncapsulatedClassifier that its environment may make through this Port. The owning EncapsulatedClassifier must offer the Features owned by the provided Interfaces.

As a kind of Property, a Port has a type. The provided and required interfaces of the Port are related to its type mediated by the value of isConjugated as follows:

* If isConjugated is false, provided is derived as the union of the sets of Interfaces realized by the type of the Port and its supertypes, or directly from the type of the Port if the Port is typed by an Interface; required is derived as the union of the sets of Interfaces used by the type of the Port and its supertypes.
* If isConjugated is true, provided is derived as the union of the sets of Interfaces used by the type of the Port and its supertypes; required is derived as the union of the sets of Interfaces realized by the type of the Port and its supertypes, or directly from the type of the Port if the Port is typed by an Interface.

The Interfaces do not necessarily establish the exact sequences of interactions across the Port. A Port’s protocol may reference a ProtocolStateMachine that describes valid sequences of Operation and Reception invocations that may occur at this Port.

When an instance of an EncapsulatedClassifier is created, instances corresponding to each of its Ports are created and held in the slots specified by each Port, in accordance with its type and multiplicity. These instances are referred to as “interaction points” and provide unique references. It is, therefore, possible for an EncapsulatedClassifier instance to differentiate between requests for the invocation of a BehavioralFeature targeted at its different Ports. Similarly, it is possible to direct such requests at a Port, and the requests will be routed as specified by the links corresponding to Connectors attached to this Port.

**NOTE.** In the following, “requests arriving at a Port” shall mean “request occurrences arriving at the interaction point of this instance corresponding to this Port.”

A Port has the ability, by setting the property isBehavior to true, to specify that any requests arriving at this Port are handled by the Behavior of the instance of the owning EncapsulatedClassifier, rather than being forwarded to any

contained instances, if any. Such a Port is called a *behavior Port*. If there is no Behavior defined for this EncapsulatedClassifier, any communication arriving at a behavior Port is lost.

A *delegation* Connector is a Connector that links a Port to a role within the owning EncapsulatedClassifier. It represents the forwarding of requests (Operation invocations and Signals). A request that arrives at a Port that has a delegation Connector to one or more Properties or Ports on Properties will be passed on to those targets for handling.

Delegation Connectors can be used to model the hierarchical decomposition of behavior, where services provided by an EncapsulatedClassifier may ultimately be realized by one that is nested multiple levels deep within it.

As a ConnectableElement, the *effective provided Interfaces* (see 11.2.3) of a Port are its provided interfaces, and the *effective required Interfaces* are its required Interfaces. However, for a *delegating Port*, i.e., a Port that is at an end of a delegation Connector and is not on a role and that is not a behavior Port, the *effective provided Interfaces* are its *required* interfaces and its *effective required Interfaces* are its *provided* interfaces. Consequently a delegating Port behaves, for connection, as though it had an internal “face” that is the conjugate of its external “face.”

If several Connectors are attached on one side of a Port, then any request arriving at this Port on a link derived from a Connector on the other side of the Port will be forwarded on links corresponding to these Connectors. It is not defined whether these requests will be forwarded on all links, or on only one of those links.

**11.3.4 Notation**

A Port of an EncapsulatedClassifier is shown as a small square symbol. The name of the Port is placed near the square symbol. The Port symbol may be placed either overlapping the boundary of the rectangle symbol denoting that EncapsulatedClassifier or it may be shown inside the rectangle symbol. When the Port is connected to elements visually contained in a compartment of the EncapsulatedClassifier, such as parts or roles in the internal structure compartment, the Port symbol must be placed within or overlapping the boundary of that compartment.

The type of a Port may be shown following the Port name, separated by colon (“:”). When isConjugated is true for the Port, the type of the Port is shown with a tilde “~” prepended. A provided Interface may be shown using the *lollipop* notation (see Interface – 10.4) attached to the Port. A required Interface may be shown by the *socket* notation attached to the Port.

A behavior Port is indicated by a Port being connected through a line to a small state symbol drawn inside the symbol representing the containing EncapsulatedClassifier. The small state symbol indicates the Behavior of the containing EncapsulatedClassifier.

The name of a Port may be suppressed. Every depiction of an unnamed Port denotes a different Port from any other Port.

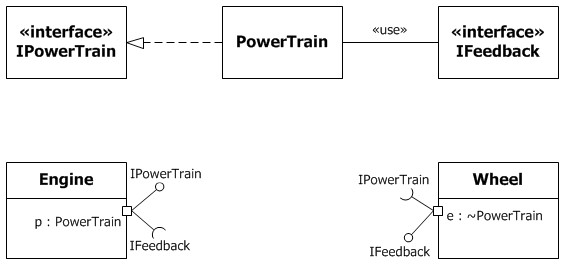
If there are multiple Interfaces associated with a Port, these Interfaces may be listed on the one Interface lollipop, separated by commas.

In the case of a Dependency wired from a simple Port with a required Interface to a simple Port to a provided Interface it is a notational option to show the dependency arrow joining the socket to the lollipop.

**11.3.5 Examples**

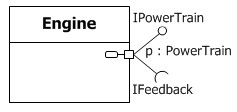
Figure 11.11 illustrates the notation for Ports. At the top of the figure is the definition of a class *PowerTrain*, together with an interface *IPowerTrain* that it realizes, and an interface *IFeedback* that it uses.

On the lower left figure, *p* is a Port on the EngineClass which is typed by *PowerTrain*. As a consequence, the provided Interface of Port *p* is *IPowerTrain* and the required Interface is *IFeedback*. The multiplicity of *p* is 1, and isConjugatedis *false*. On the right figure, *e* is a Port of the Class Wheel, which also has the type *PowerTrain* and isConjugatedset to *true*, which results in the reversal of the provided and required Interfaces.



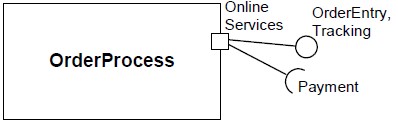
**Figure 11.11 Port notation**

Figure 11.12 illustrates a *behavior port* *p*, as indicated by its connection to the small state symbol representing the Behavior of the *Engine* Class. Its type is *PowerTrain*, as in the earlier example.

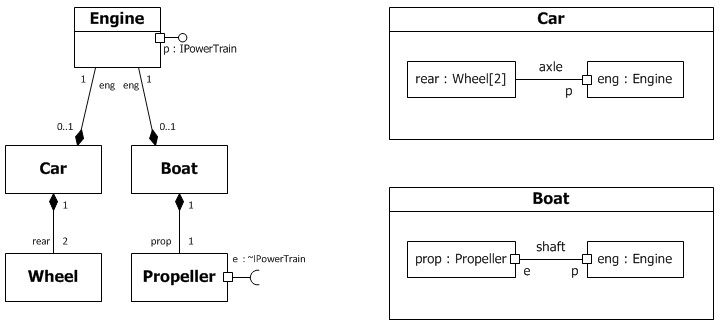


**Figure 11.12 Behavior Port notation**

Figure 11.13 below shows a Port *OnlineServices* on the *OrderProcess* Class with two provided Interfaces, *OrderEntry* and *Tracking* listed on the same interface lollipop, as well as a required Interface *Payment*.



**Figure 11.13 Port notation showing multiple provided Interfaces**



**Figure 11.14 Port examples**

Figure 11.14 shows a Class *Engine* with a Port *p* typed by its provided Interface *IPowerTrain*. This Interface specifies the services that the Engine offers at this Port (i.e., the Operations and Receptions that are accessible by communication arriving at this Port).

Two uses of the *Engine* Class are depicted: Both a Boat and a Car contain a part that is an Engine. The *Car* Class connects Port *p* of the Engine to a pair of Wheels via the *axle*. The *Boat* Class connects Port *p* of the engine to a Propeller via the *shaft*. As long as the interaction between the *Engine* and the part linked to its Port *p* obeys the constraints specified by its Interface, the Engine will function as specified, whether it is in a Car or a Boat. This example also shows that Connectors need not necessarily attach to parts via Ports (as shown in the *Car* Class).

Because the Ports are simple, the depiction of the connector within Boat could have been shown using any of the notational options shown in Figure 11.4.

### 11.4 Classes

**11.4.1 Summary**

Class is the concrete realization of EncapsulatedClassifier and BehavioredClassifier. The purpose of a Class is to specify a classification of objects and to specify the Features that characterize the structure and behavior of those objects.

**11.4.2 Abstract Syntax**

**Class**

+

isAbstract : Boolean = false {redefines isAbstract

}

isActive : Boolean = false

+

**EncapsulatedClassifier**

**BehavioredClassifier**

**Property**

**Classifier**

**Operation**

**Reception**

**Extension**

\*

class

+

\*

+

/superClass

{

subsets classifier

}

redefines general

{

}

0..1

class

+

\*

ownedAttribute

+

subsets namespace,

{

subsets structuredClassifier,

subsets classifier}

{

ordered, subsets attribute, subsets

ownedMember, redefines

ownedAttribute}

0..1

class

+

\*

ownedOperation

+

subsets featuringClassifier,

{

subsets namespace, subsets

redefinitionContext}

ordered, subsets

{

feature, subsets

redefinableElement,

subsets ownedMember}

0..1

nestingClass

+

\*

+

nestedClassifier

subsets namespace, subsets

{

redefinitionContext}

{

ordered, subsets

redefinableElement,

subsets ownedMember}

0..1

+

class

\*

ownedReception

+

subsets featuringClassifier,

{

subsets namespace}

{

subsets feature, subsets

ownedMember}

1

+

/metaclass

\*

+

/extension

{

readOnly

}

{

}

readOnly

**Figure 11.15 Classes**

**11.4.3 Semantics**

**11.4.3.1 Classes**

Class is a kind of EncapsulatedClassifier whose Features are Properties, Operations, Receptions, Ports and Connectors. Attributes of a Class are Properties that are owned by the Class. Some of these attributes may represent the ends of binary Associations.

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

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

Operations of a Class can be invoked on an object, given a particular set of values for the parameters of the Operation, according to the semantics specified in 9.6.3.

A Class cannot access private Features of another Class, or protected Features on another Class that is not its ancestor.

A Class acts as the namespace for various kinds of Classifiers defined within its scope, including Classes. Nested Classifiers are members of the namespace of the containing Class. Classifier nesting is used for reasons of information hiding.

A Class may be designated by setting isActive to true as active (i.e., each of its instances is an active object). When isActive is false the Class is passive (i.e., each of its instances executes within the context of some other object).

An active object is an object that, as a direct consequence of its creation, commences to execute its classifierBehavior, and does not cease until either the complete Behavior is executed or the object is terminated by some external object. (This is sometimes referred to as “the object having its own thread of control.”) The points at which an active object responds to communications from other objects is determined solely by the Behavior of the active object and not by the invoking object. If the classifierBehavior of an active object completes, the object is terminated.

A Class’s Receptions specify which Signals the instances of this Class handle.

An InstanceSpecification may be used to specify the initial value to be created for a Class.

All instances corresponding to parts and ports of a Class are destroyed recursively, when an instance of that Class is deleted.

A Class may act as a metaclass in the definition of Profiles and metamodels. See Profiles in 12.3.

**11.4.4 Notation**

A Class is shown using the Classifier symbol. As Class is the most widely used Classifier, no keyword is needed to indicate that the metaclass is Class.

A Class has four mandatory compartments: attributes, operations, receptions (see 9.2.4) and internal structure (see 11.2.4). A Class may also have optional compartments as described for Classifiers in general (see 9.2.4).

The operations compartment of a Class contains notation for its ownedOperations using the notation specified in 9.6.4. The receptions compartment contains ownedReceptions using the notation specified in 10.3.4.

A usage dependency may relate an InstanceSpecification to a constructor for a Class, describing the single value returned by the constructor Operation. The Operation is the client, the created instance the supplier. The

InstanceSpecification may reference parameters declared by the Operation. A constructor is an Operation having a single return result parameter of the type of the owning Class, and marked with the standard stereotype «Create». The InstanceSpecification that is the supplier of the usage dependency represents the default value of the single return result parameter of a constructor Operation.

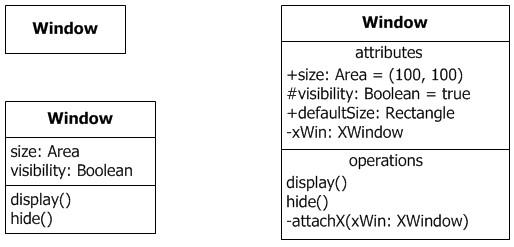
A Class with the Property *isActive* = *true* can be shown by a Class box with an additional vertical bar on either side.

A Class that represents a metaclass may be extended by the optional stereotype «Metaclass» (see StandardProfile in clause 22) shown above or before its name.

**11.4.5 Examples**

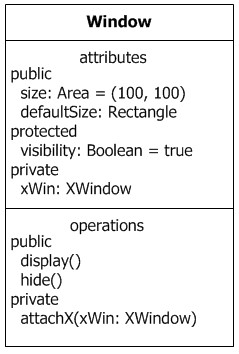
Figure 11.16 shows three ways of displaying the Class Window, according to the options set out for Classifier notation in 9.2.4. The top left symbol shows all compartments suppressed. The lower left symbol shows the attributes and operations compartments, each listing the features but suppressing details such as default values, parameters, and visibility markings. The right symbol shows these details, as well as the optional compartment headers.

**NOTE.** The display() and hide() operations have no visibility specified.



**Figure 11.16 Class notation variants**

Figure 11.17 shows the visibility grouping option (see 9.2.4) applied to the attributes and operations compartments in the Class Window.

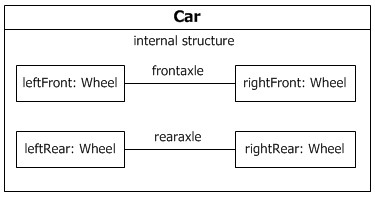


**Figure 11.17 Class notation: attributes and Operations grouped according to visibility** Figure 11.18 shows an example of an active class.



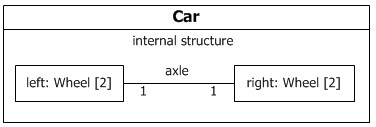
**Figure 11.18 Active Class**

The following example uses two Classes, *Car* and *Wheel*. The *Car* Class has four parts, all of type *Wheel*, representing the four wheels of the car. The front wheels and the rear wheels are linked via Connectors representing the front and rear axle, respectively. Figure 11.19 specifies that whenever an instance of the *Car* Class is created, four instances of the *Wheel* Class are created and held by composition within the car instance. In addition, one link each is created between the front wheel instances and the rear wheel instances.



**Figure 11.19 Connectors and Parts**

Figure 11.20 specifies an equivalent system, but relies on multiplicities to show the replication of the wheel and axle arrangement. This diagram specifies that there will be exactly two instances of the left wheel and exactly two instances of the right wheel, with each matching instance connected by a link deriving from the Connector representing the axle.



**Figure 11.20 Connectors and Parts in a structure diagram using multiplicities**

Figure 11.21 shows an InstanceSpecification (see 9.8) for an instance of the *Car* Class (as specified in Figure 11.19). It describes the internal structure of the *Car* that it creates and how the four contained instances of *Wheel* will be initialized. In this case, every instance of *Wheel* will have the predefined size and use the brand of tire as specified. The left wheel instances are given names, and all wheel instances are shown as playing the respective roles. The types of the wheel instances have been suppressed.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | **: Car** | | |  |  |  | | --- | --- | --- | | l1 / leftFront | frontaxle | / rightFront | | tire = "Michelin" size = "215x95" | tire = "Michelin" size = "215x95" | |  | | l2 / leftRear | rearaxle | l2 / rightRear | | tire = "Firestone" size = "215x95" | tire = "Firestone" size = "215x95" | |  | | | |  | | --- | | **Wheel** | | tire : String size : String | |

**Figure 11.21 An Instance of the Car Class**

Figure 11.22 shows a constructor for the Window class, illustrating how the standard stereotype «Create» is applied to the *makeWindow* Operation to mark it as a constructor.

«Create» makeWindow(...) : Window

**Window**

theW : Window

**Figure 11.22 InstanceSpecification indicating a constructor**

Figure 11.23 shows a constructor for the *Car* Class. This constructor takes a parameter *brand* of type *String*. It describes the internal structure of the *Car* that it creates and how the four contained instances of *Wheel* will be initialized. In this case, every instance of *Wheel* will have the predefined size and use the brand of tire passed as parameter. The left wheel instances are given names, and all wheel instances are shown as playing the parts. The types of the wheel instances have been suppressed.

**:**

**C**

**a**

**r**

t

i

r

e

=

"

M

i

c

h

e

l

i

n

"

size = "215x95"

l1 / leftFront

frontaxle

tire = "Michelin"

size = "215x95"

/ rightFront

tire = "Firestone"

size = "215x95"

l2 / leftRear

rearaxle

tire = "Firestone"

size = "215x95"

l2 / rightRear

«

C

r

e

a

t

e

»

c

r

e

a

t

e

C

a

r

(

b

r

a

n

d

:

S

t

r

i

n

g

)

**Car**

**Figure 11.23 A constructor for the Car Class**

In Figure 11.24, it is made explicit that the extended Class *Interface* is in fact a metaclass (from a reference metamodel).



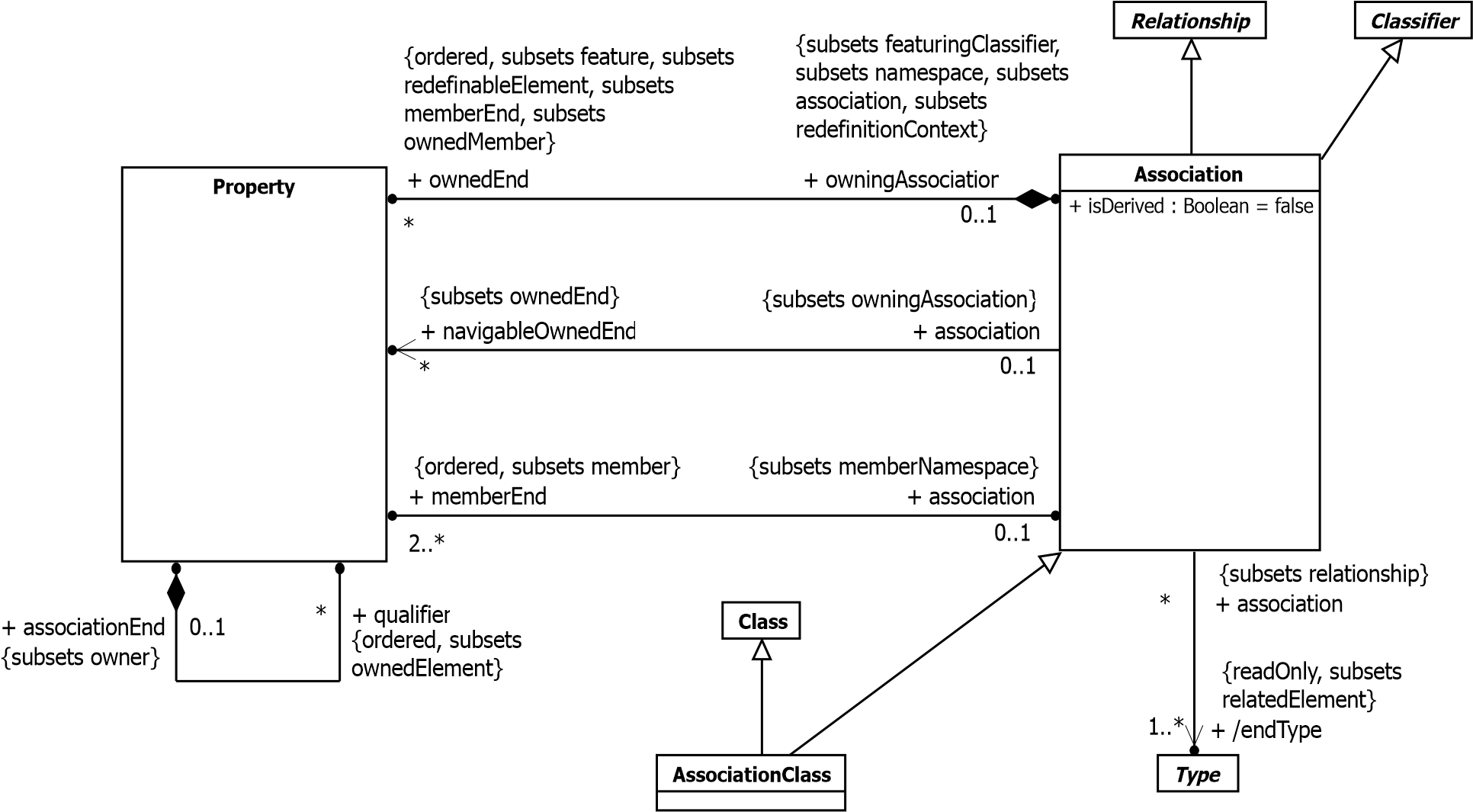
**Figure 11.24 Showing that the extended Class is a metaclass**

### 11.5 Associations

**11.5.1 Summary**

An Association classifies a set of tuples representing links between typed instances. An AssociationClass is both an Association and a Class.

**11.5.2 Abstract Syntax**



**Figure 11.25 Associations**

**11.5.3 Semantics**

**11.5.3.1 Associations**

An Association specifies a semantic relationship that can occur between typed instances. It has at least two memberEnds represented by Properties, each of which has the type of the end. More than one end of the Association may have the same type.

An Association declares that there can be links between instances whose types conform to or implement the associated types. A link is a tuple with one value for each memberEnd of the Association, where each value is an instance whose type conforms to or implements the type at the end.

Not all links need to be classified by an Association.

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

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

For an Association with N memberEnds, choose any N-1 ends. Let the Property that constitutes the other end be called oep, so that the Classifiers at the chosen N-1 ends are the context for oep (see 9.5.3). Associate specific instances with the context ends. Then the collection of links of the Association that refer to these specific instances will identify a set of instances at oep. The value represented by oep (see 9.5.3) is a collection calculated from this set as follows: All of the instances in the set occur in the collection, and nothing else does. If oep is marked as unique, each instance will occur in the collection just once, regardless of how many links connect to it. If oep is marked as nonunique, each instance will occur in the collection once for each link that connects to it. If oep is marked as ordered, the collection will be ordered in accordance with the ordering information in the links. The cardinality of this collection is its size. The multiplicity of oep constrains this cardinality, or in the case of qualified associations, the size of the collection partition that may be associated with a qualifier value.

*Subsetting* of Association ends has the meaning specified for Property (see 9.5.3).

*Specialization* is, in contrast to subsetting, a relationship in the domain of intentional semantics, which is to say it characterizes the criteria whereby membership in the collection is defined, not by the membership. In the case of Associations, specialization means that a link classified by the specializing Association is also classified by the specialized Association. Semantically this implies that sets calculated by eliminating duplicates from the collections representing the ends of the specializing Association are subsets of the corresponding sets calculated by eliminating duplicates from collections representing the ends of the specialized Association; this fact of subsetting may or may not be explicitly declared in a model.

**NOTE.** For n-ary Associations, the lower multiplicity of an end is typically 0. A lower multiplicity for an end of an nary Association of 1 (or more) implies that one link (or more) must exist for every possible combination of values for the other ends.

A binary Association may represent a composite aggregation (i.e., a whole/part relationship). Composition is represented by the isComposite attribute on the part end of the Association being set to true. See the semantics of composition in 9.5.3. An end Property of an Association may only be marked as a shared or composite aggregation if the Association is binary and the other end is not marked as a shared or composite aggregation.

An end Property of an Association that is owned by an end Class or that is a navigableOwnedEnd of the Association indicates that the Association is navigable from the opposite ends; otherwise, the Association is not navigable from the opposite ends. Navigability means that instances participating in links at runtime (instances of an Association) can be accessed efficiently from instances at the other ends of the Association. The precise mechanism by which such efficient access is achieved is implementation specific. If an end is not navigable, access from the other ends may or may not be possible, and if it is, it might not be efficient.

**NOTE.** Tools operating on UML models are not prevented from navigating Associations from non-navigable ends.

A qualified Association end has qualifiers that partition the instances associated with an instance at that end, the qualified instance. Each partition is designated by a qualifier value, which is a tuple comprising one value for each qualifier. The multiplicities at the other ends of the association determine the number of instances in each partition. So, for example, 0..1 means there is at most one instance per qualifier value. If the lower bounds are non-zero, the qualifier values must be a finite set, for example because the qualifiers are typed by enumerations.

The existence of an association may be derived from other information in the model. The logical relationship between the derivation of an Association and the derivation of its ends is model-specific.

**11.5.3.2 Association Classes**

An AssociationClass is a declaration of an Association that has a set of Features of its own. An AssociationClass is both an Association and a Class, and preserves the static and dynamic semantics of both. An AssociationClass describes a set of objects that each share the same specifications of Features, Constraints, and semantics entailed by the

AssociationClass as a kind of Class, and correspond to a unique link instantiating the AssociationClass as a kind of Association.

Both Association and Class are Classifiers and hence have a set of common properties, like being able to have Features, having a name, etc. These properties are multiply inherited from the same construct (Classifier), and are not duplicated. Therefore, an AssociationClass has only one name, and has the set of Features that are defined for Classes and

Associations. The constraints defined for Class and Association also are applicable for AssociationClass, which implies for example that the attributes of the AssociationClass, the memberEnds of the AssociationClass, and the opposite ends of

Associations connected to the AssociationClass must all have distinct names. Moreover, the specialization and refinement rules defined for Class and Association are also applicable to AssociationClass. Redefinition is applicable to an AssociationClass nested in the context of a Classifier just as it is applicable to a nested Class.

An AssociationClass inherits the composite Properties Class::ownedAttribute and Association::ownedEnd. Values of ownedAttribute are Properties that are attributes of the Class, not ends of the AssociationClass owned through

Association::ownedEnd. Values of Association::ownedEnd are the ends of the Association owned by the AssociationClass, not attributes of the AssociationClass. As Association ends, they can be used for navigation between end objects, as in all Associations, depending on whether they are navigable (see Navigability in the semantics of Association).

An instance of an AssociationClass has the characteristics both of a link representing an instantiation of the

AssociationClass as a kind of Association, and of an object representing an instantiation of the AssociationClass as a kind of Class.

**NOTE.** Even when all ends of the AssociationClass have isUnique=true, it is possible to have several instances associating the same set of instances of the end Classes.

An AssociationClass cannot be a generalization of an Association or a Class.

**11.5.4 Notation**

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

A binary Association is normally drawn as a solid line connecting two Classifiers, or a solid line connecting a single Classifier to itself (the two ends are distinct). A line may consist of one or more connected segments. The individual segments of the line itself have no semantic significance, but they may be graphically meaningful to a tool in dragging or resizing an Association symbol.

An Association symbol may be adorned as follows:

* The Association’s name can be shown as a name string near the Association symbol, but not near enough to an end to be confused with the end’s name.
* A slash appearing in front of the name of an Association, or in place of the name if no name is shown, marks the Association as being derived.
* A property string may be placed near the Association symbol, but far enough from any end to not be confused with a property string on an end.

On a binary Association drawn as a solid line, a solid triangular arrowhead next to or in place of the name of the Association and pointing along the line in the direction of one end indicates that end to be the last in the order of the ends of the Association. The arrow indicates that the Association is to be read as associating the end away from the direction of the arrow with the end to which the arrow is pointing (see Figure 11.27). This notation is for documentation purposes only and has no general semantic interpretation. It is used to capture some application-specific detail of the relationship between the associated Classifiers.

Generalizations between Associations can be shown using a generalization arrow between the Association symbols. Other notational options for Generalizations such as “shared target style” (see 9.2.4) and the notations defined in 9.7.4 may be used for Generalizations between Associations, but a conforming tool is not required to support those options.

An Association end is the connection between the line depicting an Association and the icon (often a box) depicting the connected Classifier. A name string may be placed near the end of the line to show the name of the Association end. The name is optional and suppressible.

Various other notations can be placed near the end of the line as follows:

* A multiplicity
* A <*prop-modifier*> enclosed in curly braces, where <*prop-modifier*> is defined in Property (see 9.5.4).
* A <*visibility*> symbol (see 9.5.4).

**NOTE.** If no multiplicity is shown on the diagram, no conclusion may be drawn about the multiplicity in the model.

An open arrowhead on the end of an Association indicates the end is navigable. A small x on the end of an Association indicates the end is not navigable.

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

A binary Association may have one end with aggregation *=* AggregationKind::shared or aggregation *=*

AggregationKind::composite*.* When one end has aggregation *=* AggregationKind::shared a hollow diamond is added as a terminal adornment at the end of the Association line opposite the end marked with aggregation *=*

AggregationKind::shared. The diamond shall be noticeably smaller than the diamond notation for Associations. An Association with aggregation *=* AggregationKind::composite likewise has a diamond at the corresponding end, but differs in having the diamond filled in.

Ownership of Association ends by an associated Classifier may be indicated graphically by a small filled circle, which for brevity we will term a *dot*. The dot is to be drawn integral to the graphic path of the line, at the point where it meets the Classifier, inserted between the end of the line and the side of the node representing the Classifier. The diameter of the dot shall not exceed half the height of the aggregation diamond, and shall be larger than the width of the line. This avoids visual confusion with the filled diamond notation while ensuring that it can be distinguished from the line. The dot shows that the model includes a Property of the type represented by the Classifier touched by the dot. This Property is owned by the Classifier at the other end. In such a case it is normal to suppress the Property from the attributes compartment of the owning Classifier.

The dot may be used in combination with the other graphic line-path notations for Properties of Associations and Association ends. These include aggregation type and navigability.

Explicit end-ownership notation is not mandatory, i.e., a conforming tool may not support it. Where the dot notation is used, it shall be applied consistently throughout each diagram, so that the absence of the dot signifies ownership by the Association. Stated otherwise, when applying this notation to a binary Association in a user model, the dot will be omitted only for ends which are not owned by a Classifier. In this way, in contexts where the notation is used, the absence of the dot on certain ends does not leave the ownership of those ends ambiguous.

The dot is illustrated in Figure 11.26, at the maximum allowed size. The diagram shows endA to be owned by Classifier B, and because the notation must be applied consistently throughout the diagram, this diagram also shows unambiguously that endB is owned by BinaryAssociationAB.



**Figure 11.26 Graphic notation indicating exactly one Association end owned by the Association**

Navigability notation was often used in the past according to an informal convention, whereby non-navigable ends were assumed to be owned by the Association whereas navigable ends were assumed to be owned by the Classifier at the opposite end. This convention is now deprecated. Aggregation type, navigability, and end ownership are separate concepts, each with their own explicit notation. Association ends owned by classes are always navigable, while those owned by associations may be navigable or not.

An AssociationClass is shown as a Class symbol attached to the Association path by a dashed line. The Association path may include a diamond, in which case the Class symbol shall be shown attached to the diamond by a dashed line. The Association path and the AssociationClass symbol represent the same underlying model element, which has a single name. The name may be placed on the path, in the Class symbol, or on both, but they must be the same name.

Association end names appear in the same position as regular Associations, not in the attribute compartment of the AssociationClass.

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

When two association lines cross, a conforming tool may provide the option to show a small semicircular jog to indicate that the lines do not intersect (as in electrical circuit diagrams).

In practice, it is often convenient to suppress some of the arrows and crosses that signify navigability of association ends. A conforming tool may provide various options for showing navigation arrows and crosses. As with dot notation, these options apply at the level of complete diagrams.

* Show all arrows and crosses. Navigation and its absence are made completely explicit.
* Suppress all arrows and crosses. No inference can be drawn about navigation.
* Suppress all crosses. Suppress arrows for Associations with navigability in both directions, and show arrows only for Associations with one-way navigability. In this case, the two-way navigability cannot be distinguished from situations where there is no navigation at all; however, the latter case occurs rarely in practice.

If there are two or more aggregations to the same aggregate, a conforming tool may as a purely presentational option show them as a tree by merging the aggregation ends into a single segment adorned by the solid or hollow aggregation diamond symbol. Any adornments on that single segment apply to all of the aggregation ends. The absence of an adornment on a merged segment does not imply that the properties corresponding to the suppressed adornment have equal values for all of the aggregation ends.

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

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

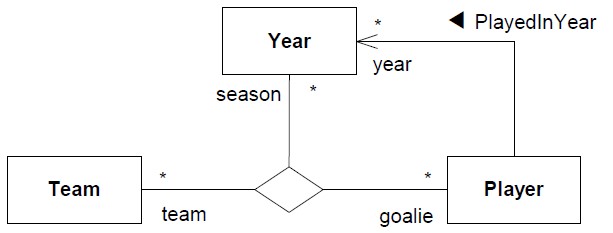
The qualifier attributes are drawn within the qualifier box. There may be one or more attributes, shown one to a line. Qualifier attributes have the same notation as Classifier attributes, except that initial value expressions are not meaningful.

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

A qualifier may not be suppressed.

**11.5.5 Examples**

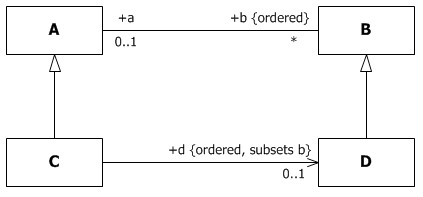
Figure 11.27 shows a binary Association from *Player* to *Year* named *PlayedInYear*.



**Figure 11.27 Binary and ternary Associations**

The solid triangle indicates the order of reading: *Player PlayedInYear Year*. The figure further shows a ternary Association between *Team*, *Year*, and *Player* with ends named *team*, *season*, and *goalie* respectively.

The following example shows Association ends with various adornments.



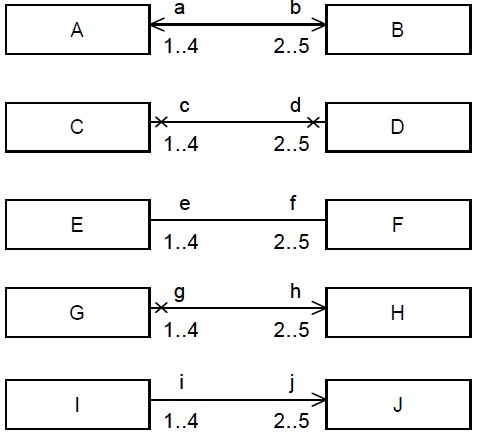
**Figure 11.28 Association ends with various adornments**

The following adornments are shown on the four Association ends in Figure 11.28.

* Names *a*, *b*, and *d* on three of the ends.
* Public visibility marked on the ends *a*, *b* and *d*.
* Multiplicities 0..1 on *a*, \* on *b*, and 0..1 on *d*.
* Specification of ordering on *b* and *d*.
* Subsetting on *d*. For an instance of Class C, the collection d is a subset of the collection b. This is equivalent to the OCL constraint:

context C inv: b->includesAll(d)

The following examples show notation for ends owned by an association (no dots).



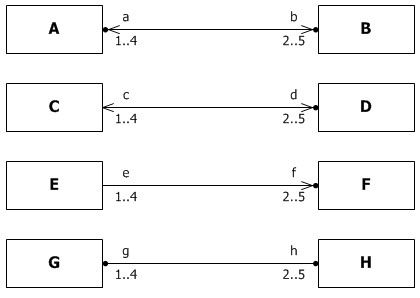
**Figure 11.29 Examples of navigable association-owned ends**

In Figure 11.29:

* The top pair AB shows a binary Association with two navigable ends.
* The second pair CD shows a binary Association with two non-navigable ends.
* The third pair EF shows a binary Association with unspecified navigability. In a diagram where arrows are only shown for one-way navigable associations, this probably signifies bidirectional navigability.
* The fourth pair GH shows a binary Association with one end navigable and the other non-navigable.
* The fifth pair IJ shows a binary Association with one end navigable and the other non-navigable, in a diagram where arrows are only shown for one-way navigable associations, and crosses are suppressed.

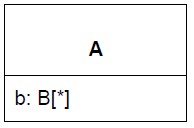
The following examples show some class-owned ends, where class ownership is indicated by the dot. In Figure 11.30:

* In the top pair AB, end b is owned by Class A and end a is owned by Class B. Because the ends are classowned, they are navigable.
* In the second pair CD, end d is owned by Class C, and hence is navigable. End c is owned by the Association, and is marked as navigable.
* In the third pair EF, end f is owned by Class E, and hence is navigable. End e is owned by the Association, and is marked as not navigable, in a diagram where arrows are only shown for one-way navigable associations, and crosses are suppressed.
* In the fourth pair GH, end h is owned by Class G and end g is owned by Class H. Because the ends are classowned, they are navigable. This is in a diagram where arrows are only shown for one-way navigable associations.



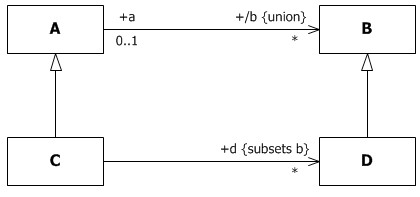
**Figure 11.30 Examples of class-owned ends**

Figure 11.31 shows that the attribute notation can be used for an Association end owned by a Class, because an Association end owned by a Class is also an attribute. Although it would typically be suppressed on grounds of redundancy, this notation may be used in conjunction with the association notation to make it perfectly clear that the attribute is also an Association end.



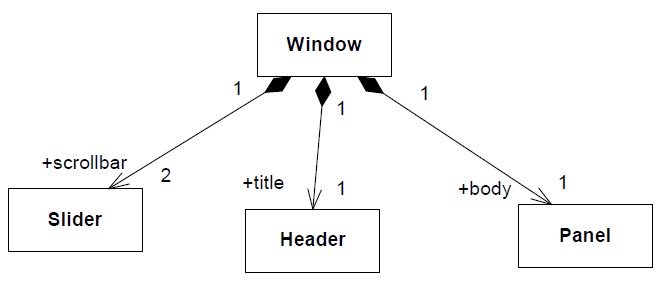
**Figure 11.31 Example of attribute notation for navigable end owned by an end Class**

Figure 11.32 shows the notation for a derived union. The attribute A::b is derived by being the strict union of all of the attributes that subset it. In this case there is just one of these, C::d. So for an instance of the Class C, d is a subset of b, and b is derived from d.



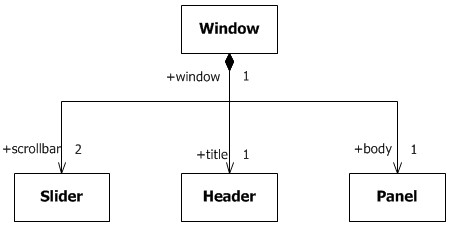
**Figure 11.32 Derived supersets (union)**

Figure 11.33 shows the black diamond notation for composite aggregation. The names of the composite ends have been suppressed in the diagram.



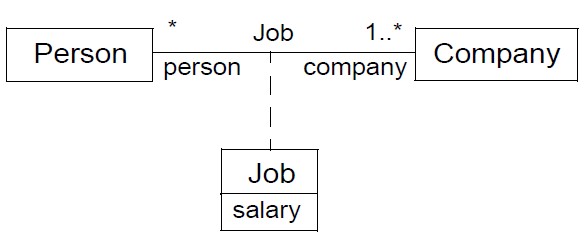
**Figure 11.33 Composite aggregation is depicted as a black diamond**

Figure 11.34 shows a similar model using the notational option of sharing the same source segment between multiple compositions. The multiplicity and name adornments on the shared end apply to all of the compositions. The model values for absent adornments on the merged segment, such as property modifiers or visibility, may differ.



**Figure 11.34 Composite aggregation sharing a source segment**

Figure 11.35 shows the notation for an AssociationClass. In this example the name of the AssociationClass appears twice, once on the Class rectangle and once on the Association. These are both renderings of the same model element.



**Figure 11.35 Example AssociationClass Job, which is defined between the two Classes Person and Company**

Figure 11.36 shows the same model using the diamond notation for the AssociationClass.

**Person**

**Company**

salary

**Job**

**Job**

\*

person

1

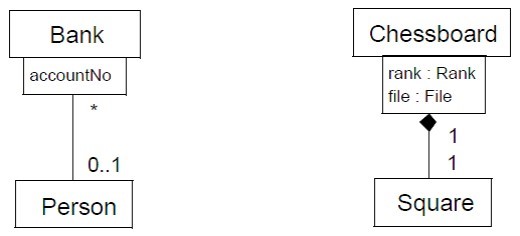
..\*

company

**Figure 11.36 Example AssociationClass using diamond symbol**

Figure 11.37 illustrates some qualified Associations. The left diagram shows that given a Bank, a particular accountNo identifies zero or one Person. The qualifier is the Property accountNo, and the qualified object is the Bank. The qualifier is owned by the unnamed Property at the Bank end of the Association, i.e., the Property whose type is Bank.

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



**Figure 11.37 Qualified associations**

### 11.6 Components

**11.6.1 Summary**

This sub clause specifies a set of constructs that can be used to define software systems of arbitrary size and complexity. In particular, it specifies a Component as a modular unit with well-defined Interfaces that is replaceable within its environment. The Component concept addresses the area of component-based development and component-based system structuring, where a Component is modeled throughout the development life cycle and successively refined into deployment and run-time.

An important aspect of component-based development is the reuse of previously constructed Components. A Component can always be considered an autonomous unit within a system or subsystem. It has one or more provided and/or required Interfaces (potentially exposed via Ports), and its internals are hidden and inaccessible other than as provided by its Interfaces. Although it may be dependent on other elements in terms of Interfaces that are required, a Component is encapsulated and its Dependencies are designed such that it can be treated as independently as possible.

As a result, Components and subsystems can be flexibly reused and replaced by connecting (“wiring”) them together.

The aspects of autonomy and reuse also extend to Components at deployment time. The artifacts that implement Component are intended to be capable of being deployed and re-deployed independently, for instance to update an existing system.

The Components package supports the specification of both logical Components (e.g., business components, process components) and physical Components (e.g., EJB components, CORBA components, COM+ and .NET components, WSDL components, etc.), along with the artifacts that implement them and the nodes on which they are deployed and executed. It is anticipated that profiles based around Components will be developed for specific component technologies and associated hardware and software environments.

**11.6.2 Abstract Syntax**

**C**

**o**

**m**

**p**

**o**

**n**

**e**

**n**

**t**

+

i

s

I

n

d

i

r

e

c

t

l

y

I

n

s

t

a

n

t

i

a

t

e

d

:

B

o

o

l

e

a

n

=

t

r

u

e

**ComponentRealization**

**Class**

**Realization**

**Interface**

**C**

**l**

**a**

**s**

**s**

**i**

**f**

**i**

**e**

**r**

**PackageableElement**

\*

+

component

\*

+

/provided

{

readOnly

}

\*

+

component

\*

+

/required

{

readOnly

}

\*

componentRealization

+

1..\*

realizingClassifier

+

}

subsets clientDependency

{

{

subsets client

}

0..1

+

component

\*

+

packagedElement

{

subsets namespace

}

{

}

subsets ownedMember

0..1

+

abstraction

\*

realization

+

{

subsets supplier,

subsets owner}

subsets ownedElement,

{

subsets supplierDependency}

**Figure 11.38 Components**

**11.6.3 Semantics**

**11.6.3.1 Components**

A Component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.

A Component is a *self-contained* unit that encapsulates the state and behavior of a number of Classifiers. A Component specifies a formal contract of the services that it provides to its clients and those that it requires from other Components or services in the system in terms of its provided and required Interfaces.

A Component is a *substitutable* unit that can be replaced at design time or run-time by a Component that offers equivalent functionality based on compatibility of its Interfaces. As long as the environment is fully compatible with the provided and required Interfaces of a Component, it will be able to interact with this environment. Similarly, a system can be extended by adding new Component types that add new functionality. Larger pieces of a system’s functionality may be assembled by reusing Components as parts in an encompassing Component or assembly of Components, and wiring them together.

A Component is modeled throughout the development life cycle and successively refined into deployment and run-time. A Component may be manifested by one or more Artifacts, and in turn, that Artifact may be deployed to its execution environment. A DeploymentSpecification may define values that parameterize the Component’s execution. (See Deployments – Clause 19).

The required and provided Interfaces of a Component allow for the specification of StructuralFeatures such as attributes and Association ends, as well as BehavioralFeatures such as Operations and Receptions. A Component may implement a provided Interface directly, or its realizing Classifiers may do so, or they may be inherited. The required and provided Interfaces may optionally be organized through Ports; these enable the definition of named sets of provided and required Interfaces that are typically (but not always) addressed at run-time.

A Component has an *external view* (or “black-box” view) by means of its publicly visible Properties and Operations. Optionally, a Behavior such as a ProtocolStateMachine may be attached to an Interface, Port, and to the Component itself, to define the external view more precisely by making dynamic constraints in the sequence of Operation calls explicit.

The wiring between Components in a system or other context can be structurally defined by using Dependencies between compatible simple Ports, or between Usages and matching InterfaceRealizations that are represented by sockets and lollipops (see 10.4.4) on Components on Component diagrams. Creating a wiring Dependency between a Usage and a matching InterfaceRealization, or between compatible simple Ports, means that there may be some additional information, such as performance requirements, transport bindings, or other policies that determine that the Interface is realized in a way that is suitable for consumption by the depending Component. Such additional information could be captured in a profile by means of stereotypes.

A Component also has an *internal view* (or “white-box” view) by means of its private Properties and realizing Classifiers. This view shows how the external Behavior is realized internally. Dependencies on the external view provide a convenient overview of what may happen in the internal view; they do not prescribe what must happen. More detailed behavior specifications such as Interactions and Activities may be used to detail the mapping from external to internal behavior.

The execution time semantics for an assembly Connector in a Component are that requests (signals and operation invocations) travel along an instance of a Connector. The execution semantics for multiple Connectors directed to and from different roles, or n-ary Connectors where n> 2, indicates that the instance that will originate or handle the request will be determined at execution time.

A number of UML standard stereotypes exist that apply to Component. For example, «Subsystem» to model large-scale Components, and «Specification» and «Realization» to model Components with distinct specification and realization definitions, where one specification may have multiple realizations (see the Standard Profiles).

A Component may be realized (or implemented) by a number of Classifiers. In that case, a Component owns a set of ComponentRealizations to these Classifiers.

A component acts like a Package for all model elements that are involved in or related to its definition, which should be either owned or imported explicitly. Typically the Classifiers that realize a Component are owned by it.

The isDirectlyInstantiated property specifies the kind of instantiation that applies to a Component. If false, the Component is instantiated as an addressable object. If true, the Component is defined at design-time, but at run-time (or executiontime) an object specified by the Component does not exist, that is, the Component is instantiated indirectly, through the instantiation of its realizing Classifiers or parts.

**11.6.4 Notation**

A Component is shown as a Classifier rectangle with the keyword «component». Optionally, in the right hand corner a Component icon can be displayed. This is a Classifier rectangle with two smaller rectangles protruding from its left hand side. If the icon symbol is shown, the keyword «component» may be hidden.

The attributes, operations and internal structure compartments all have their normal meaning. The internal structure uses the notation defined in StructuredClassifiers (11.2).

The provided and required Interfaces of a Component may be shown by means of ball (lollipop) and socket notation (see 10.4.4), where the lollipops and sockets stick out of the Component rectangle.

For displaying the full signature of a provided or required Interface of a Component, the Interfaces can also be displayed as normal expandable Classifier rectangles. For this option, the Interface rectangles are connected to the Component rectangle by appropriate dependency arrows, as specified in 7.7.4 and 10.4.4.

A conforming tool may optionally support compartments named “provided interfaces” and “required interfaces” listing the provided and required Interfaces by name. This may be a useful option in scenarios in which a Component has a large number of provided or required Interfaces.

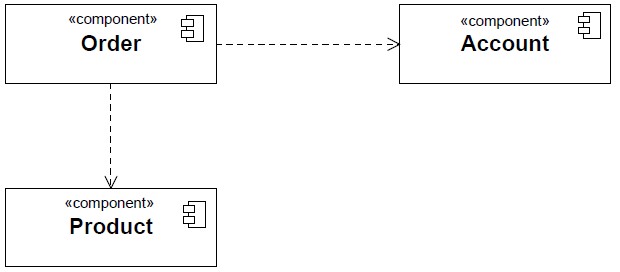
Additional optional compartments “realizations” and “artifacts” may be used to list the realizing Classifiers (Classifiers reached by following the realization property) and manifesting Artifacts (Artifacts that manifest this component – see 19.3).

A ComponentRealization is notated in the same way as a Realization dependency (i.e., as a general dashed line with a hollow triangle as an arrowhead).

The packagedElements of a Component may be displayed in an optional compartment named “packaged elements,” according to the specification for optional compartments for ownedMembers set out in 9.2.4.

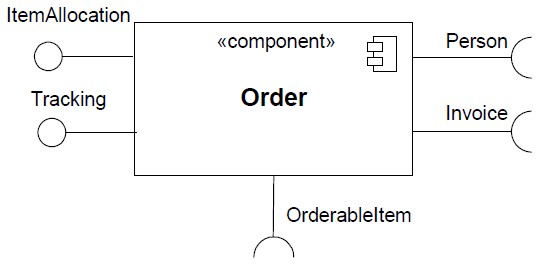
**11.6.5 Examples**

An overview diagram can show Components related by Dependencies, which signify some further unspecified kind of dependency between the components, and by implication a lack of dependency where there are no Dependency arrows.



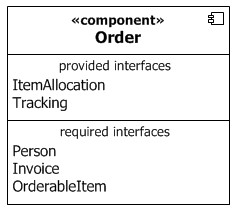
**Figure 11.39 Example of an overview diagram showing Components and their general Dependencies**

Figure 11.40 shows an external (“black-box”) view of a Component by means of interface lollipops and sockets sticking out of the Component rectangle.



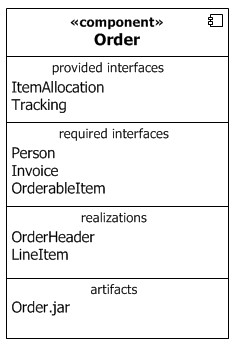
**Figure 11.40 A Component with two provided and three required Interfaces**

Figure 11.41 shows provided and required interfaces listed in optional compartments.



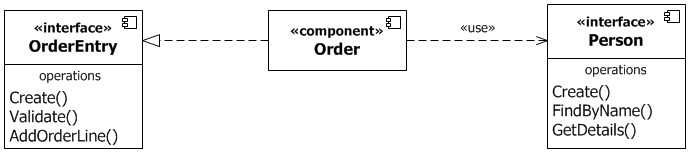
**Figure 11.41 Black box notation showing a listing of provided and required interfaces**

Figure 11.42 shows a “white box” view of a Component listing realizing Classifiers and manifesting Artifacts in additional optional compartments.



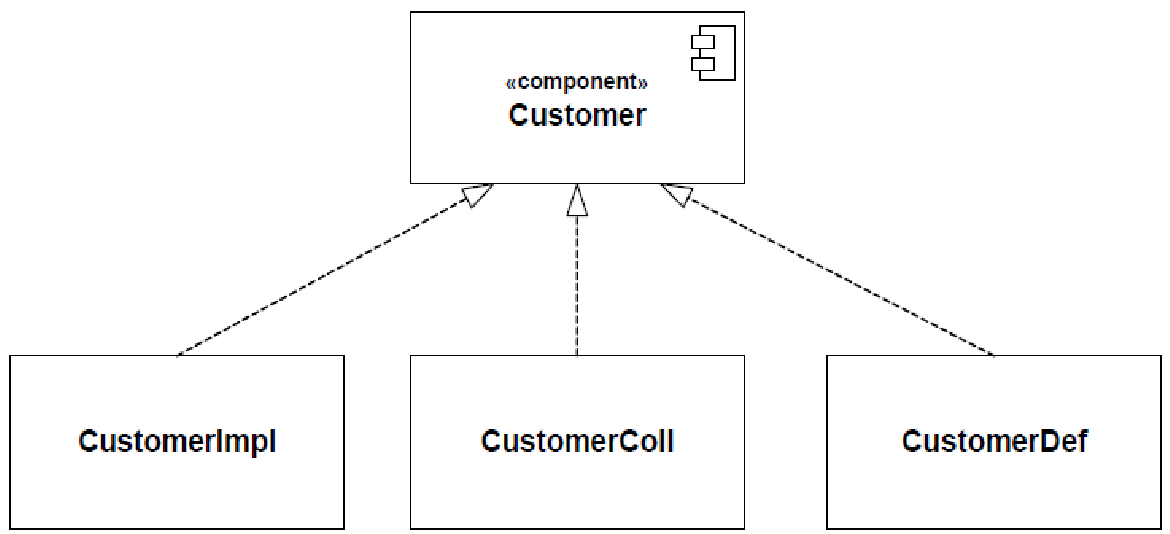
**Figure 11.42 Optional “white-box” representation of a Component**

Figure 11.43 shows explicit representation of the provided and required Interfaces using Dependency notations, allowing Interface details such as Operations to be displayed.



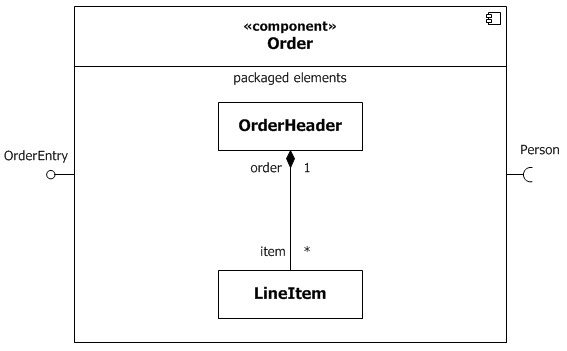
**Figure 11.43 Explicit representation of provided and required Interfaces using Dependency notation.**

Figure 11.44 shows a set of Classifiers that realize a Component with realization arrows representing the ComponentRealizations.



**Figure 11.44 A representation of the realization of a complex Component**

Figure 11.45 shows owned Classes that realize a Component nested within an optional “packaged elements” compartment of the Component shape.



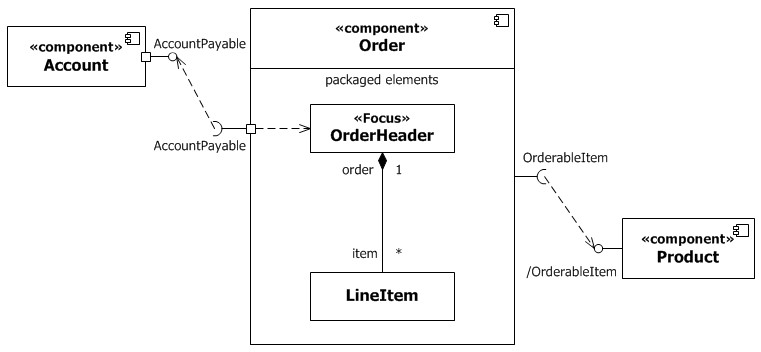
**Figure 11.45 An alternative nested representation of a complex Component**

Figure 11.46 shows various ways of wiring Components using Dependencies.

The Dependency on the right of the figure is from the Usage of OrderableItem to the InterfaceRealization of OrderableItem. This also shows that “/OrderableItem” is an Interface that is implemented by a supertype of Product, following the notation specified in 10.4.4.

The Dependency between the AccountPayable Ports illustrates the notational option of showing the dependency arrow joining the socket to the lollipop, when a Dependency is wired between simple Ports.

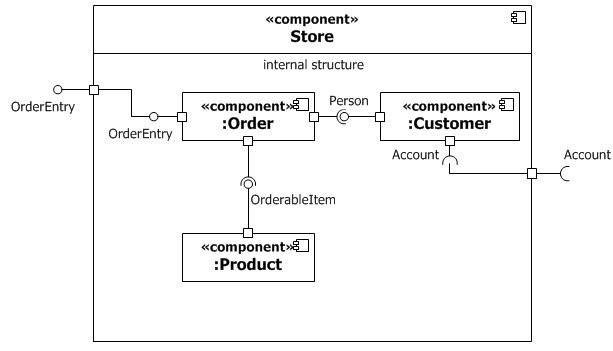
When realizing Classifiers are shown in a packaged elements compartment, a Dependency may be shown from a simple Port to a realizing Classifier to indicate that the Interface provided or required by the Port is dependent in some way upon the Classifier. This is illustrated by the Dependency from AccountPayable to OrderHeader, which indicates that something about the fact that the Component requires AccountPayable is dependent upon OrderHeader.



**Figure 11.46 Example model of a Component, its provided and required Interfaces, and wiring through Dependencies**

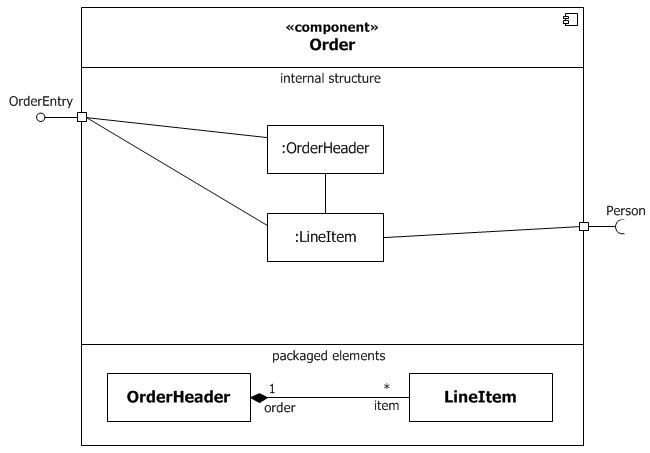
Figure 11.47 shows an internal or white-box view of the internal structure of a Component that contains other

Components with simple Ports as parts of its internal assembly. The assembly Connectors use ball-and-socket notation. The delegation connectors use the notational option that the Connector line can end on the ball or socket, rather than the simple port itself.



**Figure 11.47 Internal structure of a Component**

Figure 11.48 shows delegation Connectors from delegating Ports to handling parts; in this example the parts in the internal structure compartment are typed by Classes shown in the optional packaged elements compartment.



**Figure 11.48 Delegation Connectors connect externally provided Interfaces to the parts that realize or require them.**

### 11.7 Collaborations

**11.7.1 Summary**

The primary purpose of Collaborations is to explain how a system of communicating elements collectively accomplish a specific task or set of tasks without necessarily having to incorporate detail that is irrelevant to the explanation. Collaborations are one way that UML may be used to capture design patterns.

A CollaborationUse represents the application of the pattern described by a Collaboration to a specific situation involving specific elements playing its collaborationRoles.

**11.7.2 Abstract Syntax**

**C**

**o**

**l**

**l**

**a**

**b**

**o**

**r**

**a**

**t**

**i**

**o**

**n**

**U**

**s**

**e**

**N**

**a**

**m**

**e**

**d**

**E**

**l**

**e**

**m**

**e**

**n**

**t**

**Dependency**

**Collaboration**

**StructuredClassifier**

**BehavioredClassifier**

**ConnectableElement**

***C***

***l***

***a***

***s***

***s***

***i***

***f***

***i***

***e***

***r***

0

.

.

1

collaborationUse

+

\*

+

roleBinding

{

subsets owner

}

{

subsets ownedElement

}

\*

+

collaborationUse

1

type

+

\*

+

collaboration

\*

collaborationRole

+

{

subsets structuredClassifier

}

}

subsets role

{

0

.

.

1

0..1

+

representation

{

subsets collaborationUse

}

0..1

classifier

+

\*

+

collaborationUse

{

}

subsets owner

}

subsets ownedElement

{

**Figure 11.49 Collaborations**

**11.7.3 Semantics**

**11.7.3.1 Collaborations**

Collaborations may be used to explain how a collection of cooperating instances achieve a joint task or set of tasks. Therefore, a Collaboration typically incorporates only those aspects that are necessary for its explanation and suppresses everything else. Thus, a given object may be simultaneously playing collaborationRoles in multiple different Collaborations, but each Collaboration would only represent those aspects of that object that are relevant to its purpose.

A Collaboration defines a set of cooperating participants that are needed for a given task. The collaborationRoles of a Collaboration will be played by instances when interacting with each other. Their relationships relevant for the given task are shown as Connectors between the collaborationRoles. CollaborationRoles of Collaborations define a usage of instances, while the Classifiers typing these collaborationRoles specify all required Properties of these instances. Thus, a

Collaboration specifies what Properties instances must have to be able to participate in the Collaboration. The Connectors between the collaborationRoles specify what communication paths must exist between the participating instances.

Neither all Features nor all contents of the participating instances nor all links between these instances are always required in a particular Collaboration. Therefore, a Collaboration is often defined in terms of collaborationRoles typed by Interfaces.

Collaborations may be specialized from other Collaborations. If a collaborationRole is extended in the specialization, its type in the specialized Collaboration must conform to its type in the general Collaboration. The specialization of the types of the collaborationRoles does not imply corresponding specialization of the Classifiers that realize those collaborationRoles. It is sufficient that they conform to the constraints defined by those collaborationRoles.

A Collaboration is not directly instantiable. Instead, the cooperation defined by the Collaboration comes about as a consequence of the actual cooperation between the instances that play the collaborationRoles defined in the Collaboration.

**11.7.3.2 CollaborationUses**

A CollaborationUse represents a particular use of a Collaboration to explain the relationships between a set of elements. A CollaborationUse shows how the pattern described by a Collaboration is applied in a given *context* Classifier, by binding specific ConnectableElements from that context to the collaborationRoles of the Collaboration. There may be multiple CollaborationUses related to a given Collaboration within a Classifier, each bound differently. A given collaborationRole or Connector may be involved in multiple uses of the same or different Collaborations.

The roleBindings are implemented using Dependencies owned by the CollaborationUse. Each collaborationRole in the

Collaboration is bound by a distinct Dependency and is its supplier. The client of the Dependency is a

ConnectableElement that relates in some way to the context Classifier: it may be a direct collaborationRole of the context Classifier, or an element reachable by some set of references from the context Classifier. These roleBindings indicate which ConnectableElement from the context Classifier plays which collaborationRole in the Collaboration.

Connectors in a Collaboration typing a CollaborationUse must have corresponding Connectors between elements bound in the context Classifier, and these corresponding Connectors must have the same or more general type than the Collaboration Connectors.

One of the CollaborationUses owned by a Classifier may be singled out as representing the Behavior of the Classifier as a whole. This is called the Classifier’s representation. The Collaboration that is related to the Classifier by its representation shows how the instances corresponding to the StructuralFeatures of this Classifier (e.g., its attributes and parts) interact to generate the overall Behavior of the Classifier. The representing Collaboration may be used to provide a description of the Behavior of the Classifier at a different level of abstraction than is offered by the internal structure of the Classifier. The Properties of the Classifier are mapped to collaborationRoles in the Collaboration by the roleBindings of the CollaborationUse.

Any Behavior attached to the Collaboration applies to the set of collaborationRoles and Connectors bound within a given CollaborationUse. For example, an interaction among parts of a Collaboration applies to the Classifier parts bound to a single CollaborationUse.

If the same ConnectableElement is used in both the Collaboration and the represented element, no roleBinding is required.

It is not specified further when client and supplier elements in roleBindings are compatible.

**11.7.4 Notation**

A Collaboration is shown as a dashed ellipse shape containing the name of the Collaboration. The internal structure of a Collaboration as comprised by collaborationRoles and Connectors may be shown in a compartment within the dashed ellipse shape. This compartment follows the same notational specification as for the internal structure compartment of a normal Classifier rectangle.

Alternatively, a composite structure diagram can be used, or a normal Classifier rectangle with the keyword «collaboration».

There is no notation defined for a Collaboration whose collaborationRoles are not Properties.

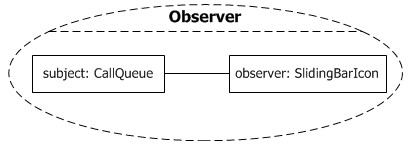
Using an alternative notation for Properties, a line may be drawn from the elliptical Collaboration shape to rectangles denoting Classifiers that are the types of Properties of the Collaboration. Each line is labeled by the name of the Property. In this manner, a diagram can show the definition of a Collaboration together with the actual Classifiers that type the collaborationRoles in that definition

A CollaborationUse is shown within an internal structure compartment of the context Classifier by a dashed ellipse containing the name of the occurrence, a colon, and the name of the Collaboration type. For every roleBinding, there is a dashed line from the ellipse to the client element; the dashed line is labeled on the client end with the name of the supplier element. With this notation the Connectors that must exist in the context Classifier as a consequence of the bindings may be suppressed.

An optional notation for CollaborationUse is as a dashed arrow with the keyword «occurrence» pointing from the using Classifier to the used Collaboration. In conjunction with this the roleBindings are shown as normal Dependency arrows. With this option any Connectors that must exist in the context Classifier as a consequence of the bindings should be shown.

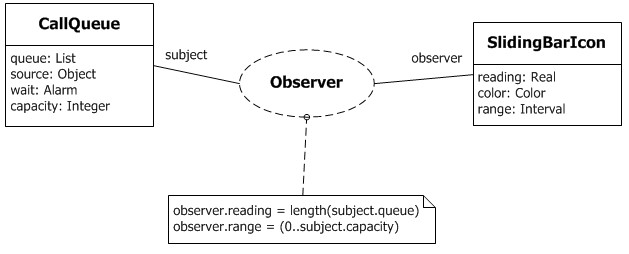
**11.7.5 Examples**

Figure 11.50 shows the internal structure of the Collaboration named Observer, with two parts that are collaborationRoles named subject and observer, and a Connector between them.



**Figure 11.50 The internal structure of the Observer Collaboration**

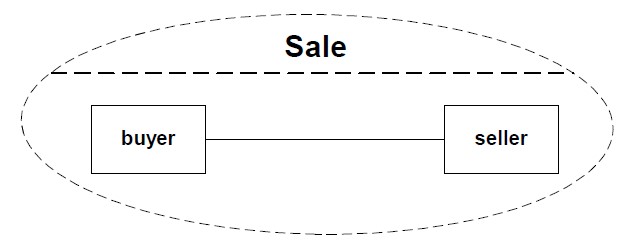
Figure 11.51 shows the alternative notation for definition of the parts of the Observer Collaboration, which allows the details of the Classes CallQueue and SlidingBarIcon to be shown in the same definition. Any instance playing the Subject collaborationRole must possess the Properties specified by CallQueue, and similarly for the Observer collaborationRole. The example also shows a Constraint on Observer.



**Figure 11.51 Alternative notation for the parts of the Observer Collaboration.**

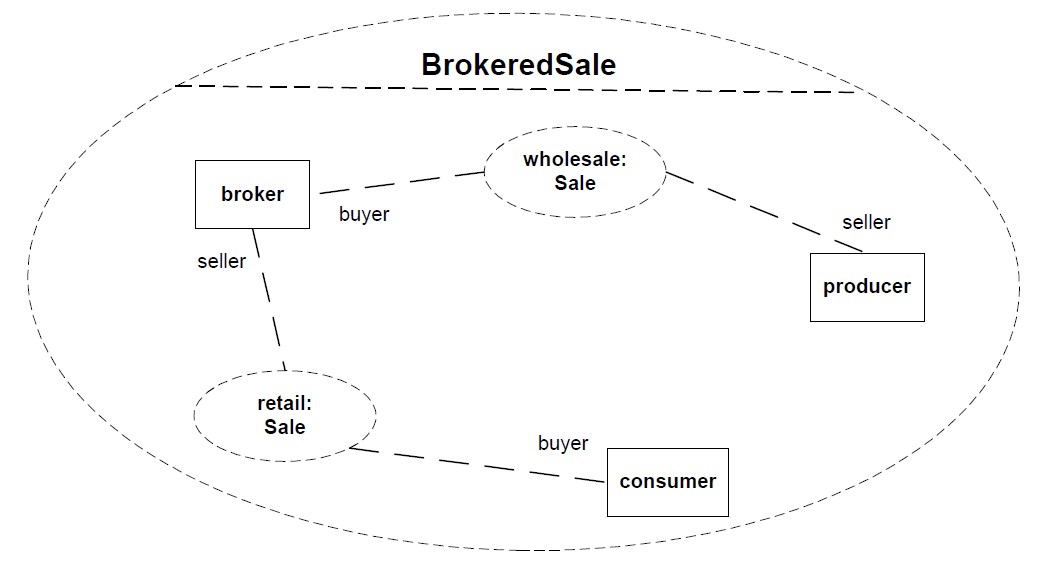
The next example shows the definition of two Collaborations, *Sale* (Figure 11.52) and *BrokeredSale* (Figure 11.53).

*Sale* is used twice as part of the definition of *BrokeredSale*. *Sale* is a Collaboration among two collaborationRoles (actually parts), a *seller* and a *buyer*. An interaction, or other Behavior specification, could be attached to *Sale* to specify the steps involved in making a *Sale*.



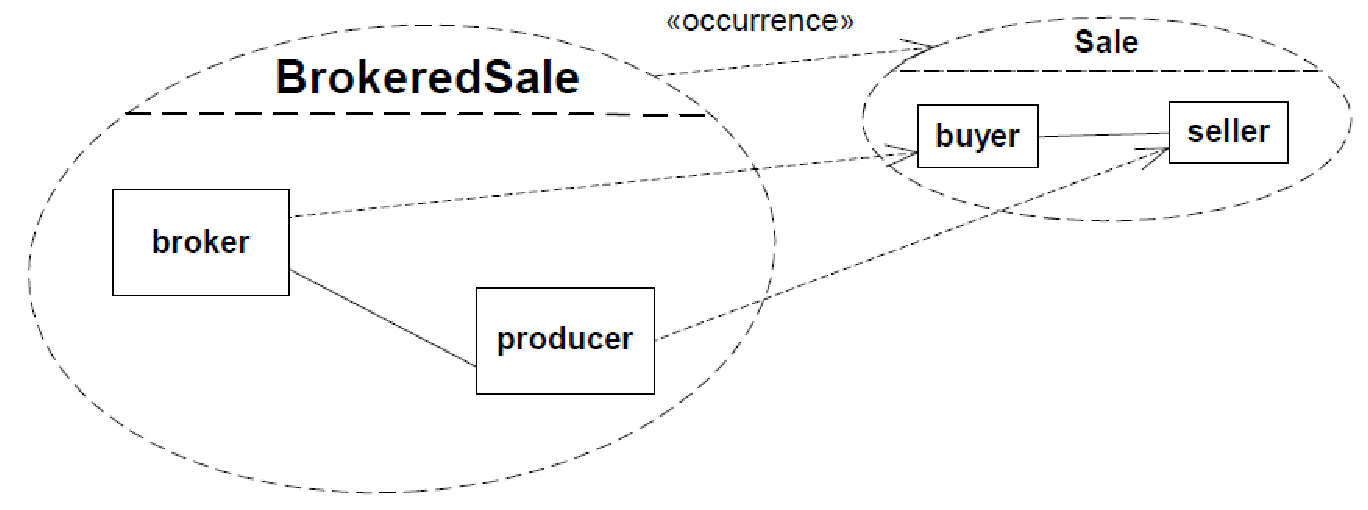
**Figure 11.52 The Sale Collaboration**

*BrokeredSale* is a Collaboration among three collaborationRoles, a *producer*, a *broker*, and a *consumer*. The specification of *BrokeredSale* shows that it consists of two CollaborationUses of the *Sale* Collaboration, indicated by the dashed ellipses. The occurrence *wholesale* indicates a *Sale* in which the *producer* is the *seller* and the *broker* is the *buyer*. The occurrence *retail* indicates a *Sale* in which the *broker* is the *seller* and the *consumer* is the *buyer*. The Connectors between *sellers* and *buyers* are not shown in the two occurrences; these Connectors must exist in the *BrokeredSale* Collaboration as a consequence of the Connector defined in *Sale*. The *BrokeredSale* Collaboration could itself be used as part of a larger Collaboration.



**Figure 11.53 The BrokeredSale Collaboration**

Figure 11.54 shows part of the *BrokeredSale* Collaboration using the optional «occurrence» notation.



**Figure 11.54 A subset of the BrokeredSale Collaboration using «occurrence» and Dependency arrows**

### 11.8 Classifier Descriptions

**11.8.1 Association [Class]**

**11.8.1.1 Description**

A link is a tuple of values that refer to typed objects. An Association classifies a set of links, each of which is an instance of the Association. Each value in the link refers to an instance of the type of the corresponding end of the Association.

**11.8.1.2 Diagrams**

Structured Classifiers, Associations, Profiles, Nodes, Properties, Link Actions

**11.8.1.3 Generalizations**

Relationship, Classifier

**11.8.1.4 Specializations**

AssociationClass, Extension, CommunicationPath

**11.8.1.5 Attributes**

* isDerived : Boolean [1..1] = false

Specifies whether the Association is derived from other model elements such as other Associations.

**11.8.1.6 Association Ends**

* /endType : Type [1..\*]{subsets Relationship::relatedElement} (opposite A\_endType\_association::association) The Classifiers that are used as types of the ends of the Association.
* memberEnd : Property [2..\*]{ordered, subsets Namespace::member} (opposite Property::association)

Each end represents participation of instances of the Classifier connected to the end in links of the Association.

* navigableOwnedEnd : Property [0..\*]{subsets Association::ownedEnd} (opposite

A\_navigableOwnedEnd\_association::association)

The navigable ends that are owned by the Association itself.

* ♦ ownedEnd : Property [0..\*]{ordered, subsets Classifier::feature, subsets

A\_redefinitionContext\_redefinableElement::redefinableElement, subsets Association::memberEnd, subsets Namespace::ownedMember} (opposite Property::owningAssociation) The ends that are owned by the Association itself.

**11.8.1.7 Operations**

* endType() : Type [1..\*] endType is derived from the types of the member ends. body: memberEnd->collect(type)->asSet()

**11.8.1.8 Constraints**

* specialized\_end\_number

An Association specializing another Association has the same number of ends as the other Association.

inv: parents()->select(oclIsKindOf(Association)).oclAsType(Association)->forAll(p |

p.memberEnd->size() = self.memberEnd->size())

* specialized\_end\_types

When an Association specializes another Association, every end of the specific Association corresponds to an end of the general Association, and the specific end reaches the same type or a subtype of the corresponding general end.

inv: Sequence{1..memberEnd->size()}->

forAll(i | general->select(oclIsKindOf(Association)).oclAsType(Association)-> forAll(ga | self.memberEnd->at(i).type.conformsTo(ga.memberEnd->at(i).type)))

* binary\_associations

Only binary Associations can be aggregations.

inv: memberEnd->exists(aggregation <> AggregationKind::none) implies (memberEnd->size() = 2 and memberEnd->exists(aggregation = AggregationKind::none))

* association\_ends

Ends of Associations with more than two ends must be owned by the Association itself. inv: memberEnd->size() > 2 implies ownedEnd->includesAll(memberEnd)

* ends\_must\_be\_typed

inv: memberEnd->forAll(type->notEmpty())

**11.8.2 AssociationClass [Class]**

**11.8.2.1 Description**

A model element that has both Association and Class properties. An AssociationClass can be seen as an Association that also has Class properties, or as a Class that also has Association properties. It not only connects a set of Classifiers but also defines a set of Features that belong to the Association itself and not to any of the associated Classifiers.

**11.8.2.2 Diagrams**

Associations

**11.8.2.3 Generalizations**

Class, Association

**11.8.2.4 Constraints**

* cannot\_be\_defined

An AssociationClass cannot be defined between itself and something else.

inv: self.endType()->excludes(self) and self.endType()->collect(et| et.oclAsType(Classifier).allParents())->flatten()->excludes(self)

* disjoint\_attributes\_ends

The owned attributes and owned ends of an AssociationClass are disjoint. inv: ownedAttribute->intersection(ownedEnd)->isEmpty()

**11.8.3 Class [Class]**

**11.8.3.1 Description**

A Class classifies a set of objects and specifies the features that characterize the structure and behavior of those objects. A Class may have an internal structure and Ports.

**11.8.3.2 Diagrams**

Classes, Associations, Components, Profiles, Nodes, Behaviors, Properties, Operations

**11.8.3.3 Generalizations**

BehavioredClassifier, EncapsulatedClassifier

**11.8.3.4 Specializations**

AssociationClass, Component, Behavior, Stereotype, Node

**11.8.3.5 Attributes**

* isAbstract : Boolean [1..1] = false

If true, the Class does not provide a complete declaration and cannot be instantiated. An abstract Class is typically used as a target of Associations or Generalizations.

* isActive : Boolean [1..1] = false

Determines whether an object specified by this Class is active or not. If true, then the owning Class is referred to as an active Class. If false, then such a Class is referred to as a passive Class.

**11.8.3.6 Association Ends**

* /extension : Extension [0..\*]{} (opposite Extension::metaclass)

This property is used when the Class is acting as a metaclass. It references the Extensions that specify additional properties of the metaclass. The property is derived from the Extensions whose memberEnds are typed by the Class.

* ♦ nestedClassifier : Classifier [0..\*]{ordered, subsets

A\_redefinitionContext\_redefinableElement::redefinableElement, subsets Namespace::ownedMember}

(opposite A\_nestedClassifier\_nestingClass::nestingClass)

The Classifiers owned by the Class that are not ownedBehaviors.

* ♦ ownedAttribute : Property [0..\*]{ordered, subsets Classifier::attribute, subsets Namespace::ownedMember, redefines StructuredClassifier::ownedAttribute} (opposite Property::class) The attributes (i.e., the Properties) owned by the Class.
* ♦ ownedOperation : Operation [0..\*]{ordered, subsets Classifier::feature, subsets

A\_redefinitionContext\_redefinableElement::redefinableElement, subsets Namespace::ownedMember} (opposite Operation::class)

The Operations owned by the Class.

* ♦ ownedReception : Reception [0..\*]{subsets Classifier::feature, subsets Namespace::ownedMember}

(opposite A\_ownedReception\_class::class) The Receptions owned by the Class.

* /superClass : Class [0..\*]{redefines Classifier::general} (opposite A\_superClass\_class::class) The superclasses of a Class, derived from its Generalizations.

**11.8.3.7 Operations**

* extension() : Extension [0..\*]

Derivation for Class::/extension : Extension

body: Extension.allInstances()->select(ext |

let endTypes : Sequence(Classifier) = ext.memberEnd->collect(type.oclAsType(Classifier))

in

endTypes->includes(self) or endTypes.allParents()->includes(self) )

* superClass() : Class [0..\*]

Derivation for Class::/superClass : Class

body: self.general()->select(oclIsKindOf(Class))->collect(oclAsType(Class))->asSet()

**11.8.3.8 Constraints**

* passive\_class

Only an active Class may own Receptions and have a classifierBehavior. inv: not isActive implies (ownedReception->isEmpty() and classifierBehavior = null)

**11.8.4 Collaboration [Class]**

**11.8.4.1 Description**

A Collaboration describes a structure of collaborating elements (roles), each performing a specialized function, which collectively accomplish some desired functionality.

**11.8.4.2 Diagrams**

Collaborations

**11.8.4.3 Generalizations**

StructuredClassifier, BehavioredClassifier

**11.8.4.4 Association Ends**

 collaborationRole : ConnectableElement [0..\*]{subsets StructuredClassifier::role} (opposite

A\_collaborationRole\_collaboration::collaboration) Represents the participants in the Collaboration.

**11.8.5 CollaborationUse [Class]**

**11.8.5.1 Description**

A CollaborationUse is used to specify the application of a pattern specified by a Collaboration to a specific situation.

**11.8.5.2 Diagrams**

Collaborations, Classifiers

**11.8.5.3 Generalizations**

NamedElement

**11.8.5.4 Association Ends**

* ♦ roleBinding : Dependency [0..\*]{subsets Element::ownedElement} (opposite

A\_roleBinding\_collaborationUse::collaborationUse)

A mapping between features of the Collaboration and features of the owning Classifier. This mapping indicates which ConnectableElement of the Classifier plays which role(s) in the Collaboration. A ConnectableElement may be bound to multiple roles in the same CollaborationUse (that is, it may play multiple roles).

* type : Collaboration [1..1] (opposite A\_type\_collaborationUse::collaborationUse)

The Collaboration which is used in this CollaborationUse. The Collaboration defines the cooperation between its roles which are mapped to ConnectableElements relating to the Classifier owning the CollaborationUse.

**11.8.5.5 Constraints**

* client\_elements

All the client elements of a roleBinding are in one Classifier and all supplier elements of a roleBinding are in one Collaboration.

inv: roleBinding->collect(client)->forAll(ne1, ne2 |

ne1.oclIsKindOf(ConnectableElement) and ne2.oclIsKindOf(ConnectableElement) and let ce1 : ConnectableElement = ne1.oclAsType(ConnectableElement), ce2 :

ConnectableElement = ne2.oclAsType(ConnectableElement) in ce1.structuredClassifier = ce2.structuredClassifier)

and

roleBinding->collect(supplier)->forAll(ne1, ne2 |

ne1.oclIsKindOf(ConnectableElement) and ne2.oclIsKindOf(ConnectableElement) and let ce1 : ConnectableElement = ne1.oclAsType(ConnectableElement), ce2 :

ConnectableElement = ne2.oclAsType(ConnectableElement) in ce1.collaboration = ce2.collaboration)

* every\_role

Every collaborationRole in the Collaboration is bound within the CollaborationUse.

inv: type.collaborationRole->forAll(role | roleBinding->exists(rb | rb.supplier>includes(role)))

* connectors

Connectors in a Collaboration typing a CollaborationUse must have corresponding Connectors between

elements bound in the context Classifier, and these corresponding Connectors must have the same or more general type than the Collaboration Connectors.

inv: type.ownedConnector->forAll(connector |

let rolesConnectedInCollab : Set(ConnectableElement) = connector.end.role->asSet(), relevantBindings : Set(Dependency) = roleBinding->select(rb | rb.supplier-

>intersection(rolesConnectedInCollab)->notEmpty()),

boundRoles : Set(ConnectableElement) = relevantBindings-

>collect(client.oclAsType(ConnectableElement))->asSet(), contextClassifier : StructuredClassifier = boundRoles-

>any(true).structuredClassifier->any(true) in

contextClassifier.ownedConnector->exists( correspondingConnector |

correspondingConnector.end.role->forAll( role | boundRoles->includes(role) ) and (connector.type->notEmpty() and correspondingConnector.type->notEmpty())

implies connector.type->forAll(conformsTo(correspondingConnector.type)) ) )

**11.8.6 Component [Class]**

**11.8.6.1 Description**

A Component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.

**11.8.6.2 Diagrams**

Components

**11.8.6.3 Generalizations**

Class

**11.8.6.4 Attributes**

* isIndirectlyInstantiated : Boolean [1..1] = true

If true, the Component is defined at design-time, but at run-time (or execution-time) an object specified by the Component does not exist, that is, the Component is instantiated indirectly, through the instantiation of its realizing Classifiers or parts.

**11.8.6.5 Association Ends**

* ♦ packagedElement : PackageableElement [0..\*]{subsets Namespace::ownedMember} (opposite

A\_packagedElement\_component::component)

The set of PackageableElements that a Component owns. In the namespace of a Component, all model elements that are involved in or related to its definition may be owned or imported explicitly. These may include e.g., Classes, Interfaces, Components, Packages, UseCases, Dependencies (e.g., mappings), and Artifacts.

* /provided : Interface [0..\*]{} (opposite A\_provided\_component::component)

The Interfaces that the Component exposes to its environment. These Interfaces may be Realized by the

Component or any of its realizingClassifiers, or they may be the Interfaces that are provided by its public Ports.

* ♦ realization : ComponentRealization [0..\*]{subsets Element::ownedElement, subsets

A\_supplier\_supplierDependency::supplierDependency} (opposite ComponentRealization::abstraction) The set of Realizations owned by the Component. Realizations reference the Classifiers of which the Component is an abstraction; i.e., that realize its behavior.

* /required : Interface [0..\*]{} (opposite A\_required\_component::component)

The Interfaces that the Component requires from other Components in its environment in order to be able to

offer its full set of provided functionality. These Interfaces may be used by the Component or any of its realizingClassifiers, or they may be the Interfaces that are required by its public Ports.

**11.8.6.6 Operations**

* provided() : Interface [0..\*]

Derivation for Component::/provided

body: let ris : Set(Interface) = allRealizedInterfaces(),

realizingClassifiers : Set(Classifier) = self.realization.realizingClassifier-

>union(self.allParents()->collect(realization.realizingClassifier))->asSet(), allRealizingClassifiers : Set(Classifier) = realizingClassifiers-

>union(realizingClassifiers.allParents())->asSet(),

realizingClassifierInterfaces : Set(Interface) = allRealizingClassifiers->iterate(c;

rci : Set(Interface) = Set{} | rci->union(c.allRealizedInterfaces())),

ports : Set(Port) = self.ownedPort->union(allParents()->collect(ownedPort))-

>asSet(),

providedByPorts : Set(Interface) = ports.provided->asSet()

in ris->union(realizingClassifierInterfaces) ->union(providedByPorts)->asSet()

 required() : Interface [0..\*]

Derivation for Component::/required

body: let uis : Set(Interface) = allUsedInterfaces(),

realizingClassifiers : Set(Classifier) = self.realization.realizingClassifier-

>union(self.allParents()->collect(realization.realizingClassifier))->asSet(), allRealizingClassifiers : Set(Classifier) = realizingClassifiers-

>union(realizingClassifiers.allParents())->asSet(),

realizingClassifierInterfaces : Set(Interface) = allRealizingClassifiers->iterate(c;

rci : Set(Interface) = Set{} | rci->union(c.allUsedInterfaces())),

ports : Set(Port) = self.ownedPort->union(allParents()->collect(ownedPort))-

>asSet(),

usedByPorts : Set(Interface) = ports.required->asSet()

in uis->union(realizingClassifierInterfaces)->union(usedByPorts)->asSet()

**11.8.6.7 Constraints**

* no\_nested\_classifiers
  1. Component cannot nest Classifiers.inv: nestedClassifier->isEmpty()
* no\_packaged\_elements
  1. Component nested in a Class cannot have any packaged elements.inv: nestingClass <> null implies packagedElement->isEmpty()

**11.8.7 ComponentRealization [Class]**

* + - 1. **Description**

Realization is specialized to (optionally) define the Classifiers that realize the contract offered by a Component in terms of its provided and required Interfaces. The Component forms an abstraction from these various Classifiers.

* + - 1. **Diagrams**

Components

* + - 1. **Generalizations**

Realization

* + - 1. **Association Ends**
* abstraction : Component [0..1]{subsets Dependency::supplier, subsets Element::owner} (opposite

Component::realization)

The Component that owns this ComponentRealization and which is implemented by its realizing Classifiers.

* realizingClassifier : Classifier [1..\*]{subsets Dependency::client} (opposite

A\_realizingClassifier\_componentRealization::componentRealization)

The Classifiers that are involved in the implementation of the Component that owns this Realization.

**11.8.8 ConnectableElement [Abstract Class]**

* + - 1. **Description**

ConnectableElement is an abstract metaclass representing a set of instances that play roles of a StructuredClassifier. ConnectableElements may be joined by attached Connectors and specify configurations of linked instances to be created within an instance of the containing StructuredClassifier.

* + - 1. **Diagrams**

Structured Classifiers, Collaborations, Activities, Lifelines, Features, Properties

* + - 1. **Generalizations**

TypedElement, ParameterableElement

* + - 1. **Specializations**

Variable, Parameter, Property

* + - 1. **Association Ends**
* /end : ConnectorEnd [0..\*]{} (opposite ConnectorEnd::role) A set of ConnectorEnds that attach to this ConnectableElement.
* templateParameter : ConnectableElementTemplateParameter [0..1]{redefines

ParameterableElement::templateParameter} (opposite

ConnectableElementTemplateParameter::parameteredElement)

The ConnectableElementTemplateParameter for this ConnectableElement parameter.

**11.8.8.6 Operations**

* end() : ConnectorEnd [0..\*]

Derivation for ConnectableElement::/end : ConnectorEnd

body: ConnectorEnd.allInstances()->select(role = self)

**11.8.9 ConnectableElementTemplateParameter [Class]**

**11.8.9.1 Description**

A ConnectableElementTemplateParameter exposes a ConnectableElement as a formal parameter for a template.

**11.8.9.2 Diagrams**

Structured Classifiers

**11.8.9.3 Generalizations**

TemplateParameter

**11.8.9.4 Association Ends**

 parameteredElement : ConnectableElement [1..1]{redefines TemplateParameter::parameteredElement}

(opposite ConnectableElement::templateParameter)

The ConnectableElement for this ConnectableElementTemplateParameter.

**11.8.10 Connector [Class]**

**11.8.10.1 Description**

A Connector specifies links that enables communication between two or more instances. In contrast to Associations, which specify links between any instance of the associated Classifiers, Connectors specify links between instances playing the connected parts only.

**11.8.10.2 Diagrams**

Structured Classifiers, Messages, Information Flows

**11.8.10.3 Generalizations**

Feature

**11.8.10.4 Attributes**

* /kind : ConnectorKind [1..1]

Indicates the kind of Connector. This is derived: a Connector with one or more ends connected to a Port which is not on a Part and which is not a behavior port is a delegation; otherwise it is an assembly.

**11.8.10.5 Association Ends**

* contract : Behavior [0..\*] (opposite A\_contract\_connector::connector)

The set of Behaviors that specify the valid interaction patterns across the Connector.

* ♦ end : ConnectorEnd [2..\*]{ordered, subsets Element::ownedElement} (opposite

A\_end\_connector::connector)

A Connector has at least two ConnectorEnds, each representing the participation of instances of the Classifiers typing the ConnectableElements attached to the end. The set of ConnectorEnds is ordered.

* redefinedConnector : Connector [0..\*]{subsets RedefinableElement::redefinedElement} (opposite

A\_redefinedConnector\_connector::connector)

A Connector may be redefined when its containing Classifier is specialized. The redefining Connector may have a type that specializes the type of the redefined Connector. The types of the ConnectorEnds of the redefining Connector may specialize the types of the ConnectorEnds of the redefined Connector. The properties of the ConnectorEnds of the redefining Connector may be replaced.

* type : Association [0..1] (opposite A\_type\_connector::connector)

An optional Association that classifies links corresponding to this Connector.

**11.8.10.6 Operations**

* kind() : ConnectorKind

Derivation for Connector::/kind : ConnectorKind

body: if end->exists( role.oclIsKindOf(Port) and partWithPort->isEmpty() and not role.oclAsType(Port).isBehavior)

then ConnectorKind::delegation else ConnectorKind::assembly endif

**11.8.10.7 Constraints**

* types

The types of the ConnectableElements that the ends of a Connector are attached to must conform to the types of the ends of the Association that types the Connector, if any.

inv: type<>null implies

let noOfEnds : Integer = end->size() in

(type.memberEnd->size() = noOfEnds) and Sequence{1..noOfEnds}->forAll(i | end-

>at(i).role.type.conformsTo(type.memberEnd->at(i).type))

* roles

The ConnectableElements attached as roles to each ConnectorEnd owned by a Connector must be owned or inherited roles of the Classifier that owned the Connector, or they must be Ports of such roles.

inv: structuredClassifier <> null

and

end->forAll( e | structuredClassifier.allRoles()->includes(e.role)

or

e.role.oclIsKindOf(Port) and structuredClassifier.allRoles()->includes(e.partWithPort))

**11.8.11 ConnectorEnd [Class]**

**11.8.11.1 Description**

A ConnectorEnd is an endpoint of a Connector, which attaches the Connector to a ConnectableElement.

**11.8.11.2 Diagrams**

Encapsulated Classifiers, Structured Classifiers

**11.8.11.3 Generalizations**

MultiplicityElement

**11.8.11.4 Association Ends**

* /definingEnd : Property [0..1]{} (opposite A\_definingEnd\_connectorEnd::connectorEnd)

A derived property referencing the corresponding end on the Association which types the Connector owing this ConnectorEnd, if any. It is derived by selecting the end at the same place in the ordering of Association ends as this ConnectorEnd.

* partWithPort : Property [0..1] (opposite A\_partWithPort\_connectorEnd::connectorEnd)

Indicates the role of the internal structure of a Classifier with the Port to which the ConnectorEnd is attached.

* role : ConnectableElement [1..1] (opposite ConnectableElement::end)

The ConnectableElement attached at this ConnectorEnd. When an instance of the containing Classifier is created, a link may (depending on the multiplicities) be created to an instance of the Classifier that types this ConnectableElement.

**11.8.11.5 Operations**

* definingEnd() : Property [0..1]

Derivation for ConnectorEnd::/definingEnd : Property

body: if connector.type = null

then null else

let index : Integer = connector.end->indexOf(self) in

connector.type.memberEnd->at(index) endif

**11.8.11.6 Constraints**

* role\_and\_part\_with\_port

If a ConnectorEnd references a partWithPort, then the role must be a Port that is defined or inherited by the type of the partWithPort.

inv: partWithPort->notEmpty() implies (role.oclIsKindOf(Port) and partWithPort.type.oclAsType(Namespace).member->includes(role))

* part\_with\_port\_empty

If a ConnectorEnd is attached to a Port of the containing Classifier, partWithPort will be empty.

inv: (role.oclIsKindOf(Port) and role.owner = connector.owner) implies partWithPort>isEmpty()

* multiplicity

The multiplicity of the ConnectorEnd may not be more general than the multiplicity of the corresponding end of the Association typing the owning Connector, if any. inv: self.compatibleWith(definingEnd)

* self\_part\_with\_port

The Property held in self.partWithPort must not be a Port. inv: partWithPort->notEmpty() implies not partWithPort.oclIsKindOf(Port)

**11.8.12 ConnectorKind [Enumeration]**

**11.8.12.1 Description**

ConnectorKind is an enumeration that defines whether a Connector is an assembly or a delegation.

**11.8.12.2 Diagrams**

* Structured Classifiers

**11.8.12.3 Literals**

* assembly

Indicates that the Connector is an assembly Connector.

* delegation

Indicates that the Connector is a delegation Connector.

**11.8.13 EncapsulatedClassifier [Abstract Class]**

**11.8.13.1 Description**

An EncapsulatedClassifier may own Ports to specify typed interaction points.

**11.8.13.2 Diagrams**

Encapsulated Classifiers, Classes

**11.8.13.3 Generalizations**

StructuredClassifier

**11.8.13.4 Specializations**

Class

**11.8.13.5 Association Ends**

* ♦ /ownedPort : Port [0..\*]{subsets StructuredClassifier::ownedAttribute} (opposite

A\_ownedPort\_encapsulatedClassifier::encapsulatedClassifier) The Ports owned by the EncapsulatedClassifier.

**11.8.13.6 Operations**

* ownedPort() : Port [0..\*]{ordered}

Derivation for EncapsulatedClassifier::/ownedPort : Port

body: ownedAttribute->select(oclIsKindOf(Port))->collect(oclAsType(Port))->asOrderedSet()

**11.8.14 Port [Class]**

**11.8.14.1 Description**

A Port is a property of an EncapsulatedClassifier that specifies a distinct interaction point between that

EncapsulatedClassifier and its environment or between the (behavior of the) EncapsulatedClassifier and its internal parts. Ports are connected to Properties of the EncapsulatedClassifier by Connectors through which requests can be made to invoke BehavioralFeatures. A Port may specify the services an EncapsulatedClassifier provides (offers) to its environment as well as the services that an EncapsulatedClassifier expects (requires) of its environment. A Port may have an associated ProtocolStateMachine.

**11.8.14.2 Diagrams**

Encapsulated Classifiers, Events, Invocation Actions

**11.8.14.3 Generalizations**

Property

**11.8.14.4 Attributes**

* isBehavior : Boolean [1..1] = false

Specifies whether requests arriving at this Port are sent to the classifier behavior of this EncapsulatedClassifier. Such a Port is referred to as a behavior Port. Any invocation of a BehavioralFeature targeted at a behavior Port will be handled by the instance of the owning EncapsulatedClassifier itself, rather than by any instances that it may contain.

* isConjugated : Boolean [1..1] = false

Specifies the way that the provided and required Interfaces are derived from the Port’s Type.

* isService : Boolean [1..1] = true

If true, indicates that this Port is used to provide the published functionality of an EncapsulatedClassifier. If false, this Port is used to implement the EncapsulatedClassifier but is not part of the essential externally-visible functionality of the EncapsulatedClassifier and can, therefore, be altered or deleted along with the internal implementation of the EncapsulatedClassifier and other properties that are considered part of its implementation.

**11.8.14.5 Association Ends**

* protocol : ProtocolStateMachine [0..1] (opposite A\_protocol\_port::port)

An optional ProtocolStateMachine which describes valid interactions at this interaction point.

* /provided : Interface [0..\*]{} (opposite A\_provided\_port::port)

The Interfaces specifying the set of Operations and Receptions that the EncapsulatedClassifier offers to its environment via this Port, and which it will handle either directly or by forwarding it to a part of its internal structure. This association is derived according to the value of isConjugated. If isConjugated is false, provided is derived as the union of the sets of Interfaces realized by the type of the port and its supertypes, or directly from the type of the Port if the Port is typed by an Interface. If isConjugated is true, it is derived as the union of the sets of Interfaces used by the type of the Port and its supertypes.

* redefinedPort : Port [0..\*]{subsets Property::redefinedProperty} (opposite A\_redefinedPort\_port::port) A Port may be redefined when its containing EncapsulatedClassifier is specialized. The redefining Port may have additional Interfaces to those that are associated with the redefined Port or it may replace an Interface by one of its subtypes.
* /required : Interface [0..\*]{} (opposite A\_required\_port::port)

The Interfaces specifying the set of Operations and Receptions that the EncapsulatedCassifier expects its environment to handle via this port. This association is derived according to the value of isConjugated. If isConjugated is false, required is derived as the union of the sets of Interfaces used by the type of the Port and its supertypes. If isConjugated is true, it is derived as the union of the sets of Interfaces realized by the type of the Port and its supertypes, or directly from the type of the Port if the Port is typed by an Interface.

**11.8.14.6 Operations**

* provided() : Interface [0..\*]

Derivation for Port::/provided

body: if isConjugated then basicRequired() else basicProvided() endif

* required() : Interface [0..\*]

Derivation for Port::/required

body: if isConjugated then basicProvided() else basicRequired() endif

* basicProvided() : Interface [0..\*]

The union of the sets of Interfaces realized by the type of the Port and its supertypes, or directly the type of the Port if the Port is typed by an Interface.

body: if type.oclIsKindOf(Interface) then type.oclAsType(Interface)->asSet()

else type.oclAsType(Classifier).allRealizedInterfaces() endif

* basicRequired() : Interface [0..\*]

The union of the sets of Interfaces used by the type of the Port and its supertypes. body: type.oclAsType(Classifier).allUsedInterfaces()

**11.8.14.7 Constraints**

* port\_aggregation

Port.aggregation must be composite. inv: aggregation = AggregationKind::composite

* default\_value

A defaultValue for port cannot be specified when the type of the Port is an Interface. inv: type.oclIsKindOf(Interface) implies defaultValue->isEmpty()

* encapsulated\_owner

All Ports are owned by an EncapsulatedClassifier. inv: owner = encapsulatedClassifier

**11.8.15 StructuredClassifier [Abstract Class]**

**11.8.15.1 Description**

StructuredClassifiers may contain an internal structure of connected elements each of which plays a role in the overall Behavior modeled by the StructuredClassifier.

**11.8.15.2 Diagrams**

Encapsulated Classifiers, Structured Classifiers, Collaborations

**11.8.15.3 Generalizations**

Classifier

**11.8.15.4 Specializations**

Collaboration, EncapsulatedClassifier

**11.8.15.5 Association Ends**

* ♦ ownedAttribute : Property [0..\*]{ordered, subsets Classifier::attribute, subsets StructuredClassifier::role, subsets Namespace::ownedMember} (opposite A\_ownedAttribute\_structuredClassifier::structuredClassifier) The Properties owned by the StructuredClassifier.
* ♦ ownedConnector : Connector [0..\*]{subsets Classifier::feature, subsets

A\_redefinitionContext\_redefinableElement::redefinableElement, subsets Namespace::ownedMember} (opposite A\_ownedConnector\_structuredClassifier::structuredClassifier) The connectors owned by the StructuredClassifier.

* /part : Property [0..\*]{} (opposite A\_part\_structuredClassifier::structuredClassifier)

The Properties specifying instances that the StructuredClassifier owns by composition. This collection is derived, selecting those owned Properties where isComposite is true.

* /role : ConnectableElement [0..\*]{union, subsets Namespace::member} (opposite

A\_role\_structuredClassifier::structuredClassifier)

The roles that instances may play in this StructuredClassifier.

**11.8.15.6 Operations**

* part() : Property [0..\*]

Derivation for StructuredClassifier::/part

body: ownedAttribute->select(isComposite)

* allRoles() : ConnectableElement [0..\*]

All features of type ConnectableElement, equivalent to all direct and inherited roles.

body: allFeatures()->select(oclIsKindOf(ConnectableElement))>collect(oclAsType(ConnectableElement))->asSet()

### 11.9 Association Descriptions

**11.9.1 A\_collaborationRole\_collaboration [Association]**

**11.9.1.1 Diagrams**

Collaborations

**11.9.1.2 Owned Ends**

* collaboration : Collaboration [0..\*]{subsets A\_role\_structuredClassifier::structuredClassifier} (opposite Collaboration::collaborationRole)

**11.9.2 A\_connectableElement\_templateParameter\_parameteredElement [Association]**

**11.9.2.1 Diagrams**

Structured Classifiers

**11.9.2.2 Member Ends**

* ConnectableElement::templateParameter
* ConnectableElementTemplateParameter::parameteredElement

**11.9.3 A\_contract\_connector [Association]**

**11.9.3.1 Diagrams**

Structured Classifiers

**11.9.3.2 Owned Ends**

* connector : Connector [0..\*] (opposite Connector::contract)

**11.9.4 A\_definingEnd\_connectorEnd [Association]**

**11.9.4.1 Diagrams**

Structured Classifiers

**11.9.4.2 Owned Ends**

* connectorEnd : ConnectorEnd [0..\*] (opposite ConnectorEnd::definingEnd)

**11.9.5 A\_endType\_association [Association]**

**11.9.5.1 Diagrams**

Associations

**11.9.5.2 Owned Ends**

* association : Association [0..\*]{subsets A\_relatedElement\_relationship::relationship} (opposite Association::endType)

**11.9.6 A\_end\_connector [Association]**

**11.9.6.1 Diagrams**

Structured Classifiers

**11.9.6.2 Owned Ends**

* connector : Connector [1..1]{subsets Element::owner} (opposite Connector::end)

**11.9.7 A\_end\_role [Association]**

**11.9.7.1 Diagrams**

Structured Classifiers

**11.9.7.2 Member Ends**

* ConnectableElement::end
* ConnectorEnd::role

**11.9.8 A\_extension\_metaclass [Association]**

**11.9.8.1 Diagrams**

Classes, Profiles

**11.9.8.2 Member Ends**

* Class::extension
* Extension::metaclass

**11.9.9 A\_memberEnd\_association [Association]**

**11.9.9.1 Diagrams**

Associations, Properties

**11.9.9.2 Member Ends**

* Association::memberEnd
* Property::association

**11.9.10 A\_navigableOwnedEnd\_association [Association]**

**11.9.10.1 Diagrams**

Associations

**11.9.10.2 Owned Ends**

* association : Association [0..1]{subsets Property::owningAssociation} (opposite Association::navigableOwnedEnd)

**11.9.11 A\_nestedClassifier\_nestingClass [Association]**

**11.9.11.1 Diagrams**

Classes

**11.9.11.2 Owned Ends**

* nestingClass : Class [0..1]{subsets NamedElement::namespace, subsets

RedefinableElement::redefinitionContext} (opposite Class::nestedClassifier)

**11.9.12 A\_ownedAttribute\_class [Association]**

**11.9.12.1 Diagrams**

Classes, Properties

**11.9.12.2 Member Ends**

* Class::ownedAttribute
* Property::class

**11.9.13 A\_ownedAttribute\_structuredClassifier [Association]**

**11.9.13.1 Diagrams**

Structured Classifiers

**11.9.13.2 Generalizations**

A\_role\_structuredClassifier

**11.9.13.3 Owned Ends**

* structuredClassifier : StructuredClassifier [0..1]{subsets NamedElement::namespace, subsets

A\_attribute\_classifier::classifier, redefines A\_role\_structuredClassifier::structuredClassifier} (opposite StructuredClassifier::ownedAttribute)

**11.9.14 A\_ownedConnector\_structuredClassifier [Association]**

**11.9.14.1 Diagrams**

Structured Classifiers

**11.9.14.2 Owned Ends**

* structuredClassifier : StructuredClassifier [0..1]{subsets Feature::featuringClassifier, subsets

NamedElement::namespace, subsets RedefinableElement::redefinitionContext} (opposite StructuredClassifier::ownedConnector)

**11.9.15 A\_ownedEnd\_owningAssociation [Association]**

**11.9.15.1 Diagrams**

Associations, Properties

**11.9.15.2 Member Ends**

* Association::ownedEnd
* Property::owningAssociation

**11.9.16 A\_ownedOperation\_class [Association]**

**11.9.16.1 Diagrams**

Classes, Operations

**11.9.16.2 Member Ends**

* Class::ownedOperation
* Operation::class

**11.9.17 A\_ownedPort\_encapsulatedClassifier [Association]**

**11.9.17.1 Diagrams**

Encapsulated Classifiers

**11.9.17.2 Owned Ends**

* encapsulatedClassifier : EncapsulatedClassifier [0..1]{subsets

A\_ownedAttribute\_structuredClassifier::structuredClassifier} (opposite EncapsulatedClassifier::ownedPort)

**11.9.18 A\_ownedReception\_class [Association]**

**11.9.18.1 Diagrams**

Classes

**11.9.18.2 Owned Ends**

* class : Class [0..1]{subsets Feature::featuringClassifier, subsets NamedElement::namespace} (opposite Class::ownedReception)

**11.9.19 A\_packagedElement\_component [Association]**

**11.9.19.1 Diagrams**

Components

**11.9.19.2 Owned Ends**

* component : Component [0..1]{subsets NamedElement::namespace} (opposite Component::packagedElement)

**11.9.20 A\_partWithPort\_connectorEnd [Association]**

**11.9.20.1 Diagrams**

Encapsulated Classifiers

**11.9.20.2 Owned Ends**

* connectorEnd : ConnectorEnd [0..\*] (opposite ConnectorEnd::partWithPort)

**11.9.21 A\_part\_structuredClassifier [Association]**

**11.9.21.1 Diagrams**

Structured Classifiers

**11.9.21.2 Owned Ends**

* structuredClassifier : StructuredClassifier [0..1] (opposite StructuredClassifier::part)

**11.9.22 A\_protocol\_port [Association]**

**11.9.22.1 Diagrams**

Encapsulated Classifiers

**11.9.22.2 Owned Ends**

* port : Port [0..\*] (opposite Port::protocol)

**11.9.23 A\_provided\_component [Association]**

**11.9.23.1 Diagrams**

Components

**11.9.23.2 Owned Ends**

* component : Component [0..\*] (opposite Component::provided)

**11.9.24 A\_provided\_port [Association]**

**11.9.24.1 Diagrams**

Encapsulated Classifiers

**11.9.24.2 Owned Ends**

* port : Port [0..\*] (opposite Port::provided)

**11.9.25 A\_realization\_abstraction\_component [Association]**

**11.9.25.1 Diagrams**

Components

**11.9.25.2 Member Ends**

* Component::realization
* ComponentRealization::abstraction

**11.9.26 A\_realizingClassifier\_componentRealization [Association]**

**11.9.26.1 Diagrams**

Components

**11.9.26.2 Owned Ends**

* componentRealization : ComponentRealization [0..\*]{subsets NamedElement::clientDependency} (opposite ComponentRealization::realizingClassifier)

**11.9.27 A\_redefinedConnector\_connector [Association]**

**11.9.27.1 Diagrams**

Structured Classifiers

**11.9.27.2 Owned Ends**

* connector : Connector [0..\*]{subsets A\_redefinedElement\_redefinableElement::redefinableElement} (opposite Connector::redefinedConnector)

**11.9.28 A\_redefinedPort\_port [Association]**

**11.9.28.1 Diagrams**

Encapsulated Classifiers

**11.9.28.2 Owned Ends**

* port : Port [0..\*]{subsets A\_redefinedProperty\_property::property} (opposite Port::redefinedPort)

**11.9.29 A\_required\_component [Association]**

**11.9.29.1 Diagrams**

Components

**11.9.29.2 Owned Ends**

* component : Component [0..\*] (opposite Component::required)

**11.9.30 A\_required\_port [Association]**

**11.9.30.1 Diagrams**

Encapsulated Classifiers

**11.9.30.2 Owned Ends**

* port : Port [0..\*] (opposite Port::required)

**11.9.31 A\_roleBinding\_collaborationUse [Association]**

**11.9.31.1 Diagrams**

Collaborations

**11.9.31.2 Owned Ends**

* collaborationUse : CollaborationUse [0..1]{subsets Element::owner} (opposite CollaborationUse::roleBinding)

**11.9.32 A\_role\_structuredClassifier [Association]**

**11.9.32.1 Diagrams**

Structured Classifiers

**11.9.32.2 Specializations**

A\_ownedAttribute\_structuredClassifier

**11.9.32.3 Owned Ends**

* /structuredClassifier : StructuredClassifier [0..\*]{union, subsets

A\_member\_memberNamespace::memberNamespace} (opposite StructuredClassifier::role)

**11.9.33 A\_superClass\_class [Association]**

**11.9.33.1 Diagrams**

Classes

**11.9.33.2 Owned Ends**

* class : Class [0..\*]{subsets A\_general\_classifier::classifier} (opposite Class::superClass)

**11.9.34 A\_type\_collaborationUse [Association]**

**11.9.34.1 Diagrams**

Collaborations

**11.9.34.2 Owned Ends**

* collaborationUse : CollaborationUse [0..\*] (opposite CollaborationUse::type)

**11.9.35 A\_type\_connector [Association]**

**11.9.35.1 Diagrams**

Structured Classifiers

**11.9.35.2 Owned Ends**

* connector : Connector [0..\*] (opposite Connector::type)

## 12 Packages

### 12.1 Summary

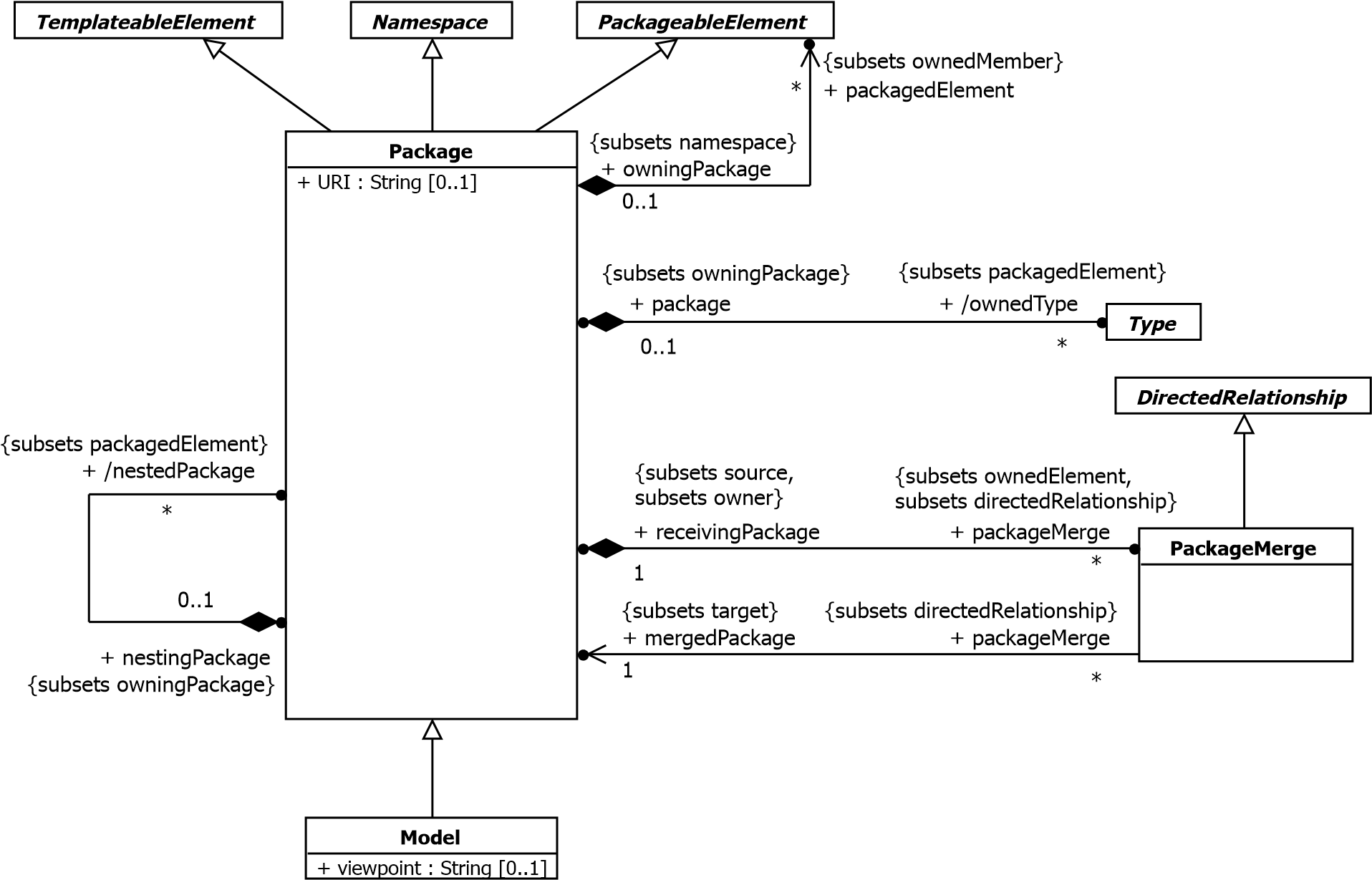
Packages provide the main generic structuring and organizing capability of UML. There are specializations for Models and for Profiles which organize extensions to UML.

### 12.2 Packages

**12.2.1 Summary**

This sub clause provides the specification for Packages and Models.

**12.2.2 Abstract Syntax**



**Figure 12.1 Packages**

**12.2.3 Semantics**

**12.2.3.1 Package**

A Package is a namespace for its members, which comprise those elements associated via packagedElement (which are said to be *owned* or *contained*), and those *imported*.

A Package definition can extend the contents of other Packages through the *merging* of the contained elements.

A Package may be defined as a template and bound to other templates: see sub clause 7.3, Templates, for further information.

The URI can be specified to provide a unique identifier for a Package. Within UML there is no predetermined usage for this, with the exception of profiles (see sub clause 12.3.3). It may, for example, be used by model management facilities for model identification. The URI should hence be unique and unchanged once assigned. There is no requirement that the URI be dereferenceable (though this is of course permitted).

**12.2.3.2 PackageMerge**

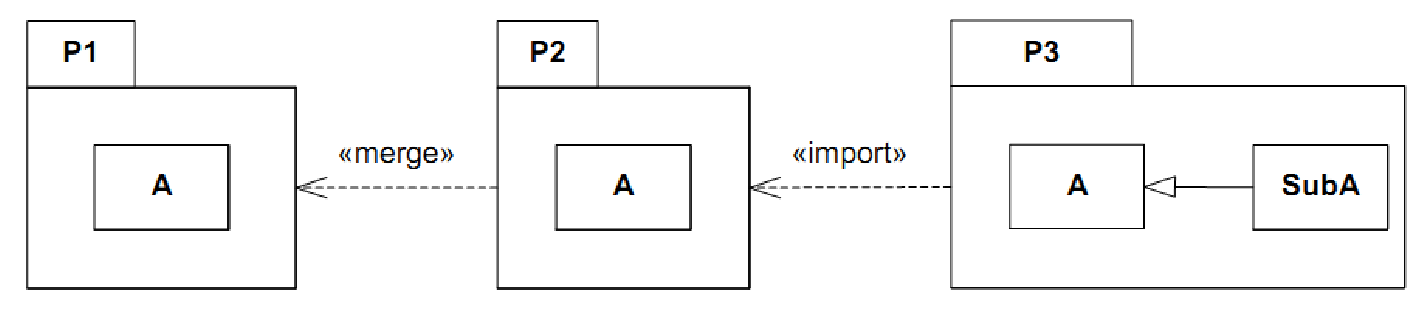
A PackageMerge is a directed relationship between two Packages that indicates that the contents of the target mergedPackage are combined into the source receivingPackage according to a set of rules defined below. It is very similar to Generalization in the sense that the source element conceptually adds the characteristics of the target element to its own characteristics resulting in an element that combines the characteristics of both. Just as a subclass is not normally depicted with its inherited features, a receiving Package is not normally depicted with the merged elements from its mergedPackages. In terms of model semantics, there is no difference between a model with explicit PackageMerges, and a model in which all the merges have been performed. Likewise XMI files containing PackageMerge are semantically equivalent to the same XMI files with the PackageMerges expanded.

Also, as with Generalization, a Package may not merge itself (directly or indirectly).

This capability is designed to be used when elements defined in different Packages have the same name and are intended to represent the same concept. A given base concept may be merged for different purposes, with each purpose defined in a separate receiving Package. By selecting different receiving packages, it is possible to obtain a custom definition of a concept for a specific end.

Thus, any reference to a model element contained in the receiving Package implies a reference to the results of the merge rather than to the increment that is contained in that Package. This is illustrated by the example in Figure 12.2 in which Package P2 defines an increment of Class A originally defined in P1. Package P2 merges the contents of Package P1, which implies the merging of P1::A into increment P2::A. Package P3 defines a subclass of P2::A called SubA. In this case, element A in Package P2 (P2::A) represents the *result* of the merge of P1::A into P2::A and not just the increment P2::A.

**NOTE.** If another package were to import P1, then a reference to A in the importing package would represent P1::A rather than the A resulting from merge.



**Figure 12.2 Illustration of the Meaning of Package Merge**

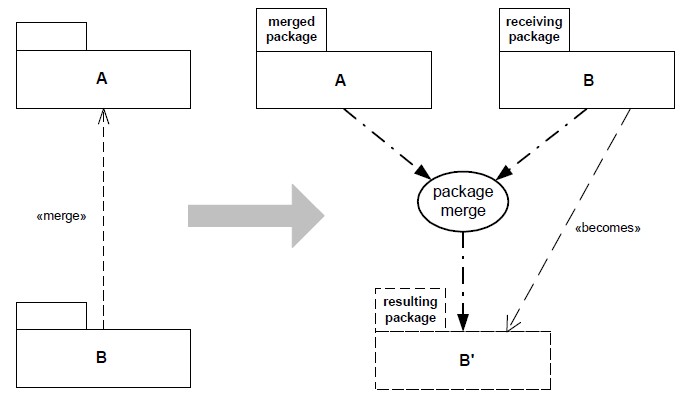
A PackageMerge can be viewed as an operation (that is itself a set of transformations) whereby the contents of the Package to be merged are combined with the contents of the receiving Package. In cases in which certain elements in the two Packages match (according to defined rules), their contents are (conceptually) merged into a single resulting element according to the formal rules of PackageMerge specified below. This operation is akin to “copying down” the features of superclasses into a subclass: the fully expanded subclass is the equivalent to the resulting package.

To understand the rules of PackageMerge, it is necessary to clearly distinguish between three distinct entities: the mergedPackage (e.g., P1 in Figure 12.2), the receivingPackage (e.g., P2), and the result of the merge transformations (also P2). The receivingPackage also plays the role of resultingPackage. This dual interpretation of the same model element can be confusing, so it is useful to introduce the following terminology that aids understanding:

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

This terminology is based on a conceptual view of PackageMerge that is represented by the schematic diagram in

Figure 12.3 (NB: this is not a UML diagram). The packagedElements (direct and indirect) of Packages A and B are all incorporated into the namespace of Package B'. However, it is important to emphasize that this view is merely a convenience for describing the semantics of PackageMerge and is not reflected in the *stored* model, that is, the *physical* model itself is not transformed in any way by the presence of PackageMerges.



**Figure 12.3 Conceptual View of the Package Merge Semantics**

The semantics of PackageMerge are defined by a set of constraints and transformations. The constraints specify the preconditions for a valid PackageMerge, while the transformations describe its semantic effects (i.e., postconditions). If any constraints are violated, the PackageMerge is ill-formed and the model that contains it is invalid. Different element metatypes have different semantics, but the general principle is always the same: a resulting element will not be any less capable than it was prior to the merge: meaning, for instance, that the resulting navigability, multiplicity, visibility, etc. of a receiving model element will not be reduced as a result of a PackageMerge. One of the key consequences of this is that model elements in the resulting Package are compatible extensions of the corresponding elements in the (unmerged) receiving package.

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

**12.2.3.3 General Package Merge Rules**

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

CONSTRAINTS:

1. There can be no cycles in the «merge» directed graph.
2. A Package cannot merge a Package in which it is contained (via owningPackage – direct or indirect). 3 A Package cannot merge a Package that it contains (via packagedElement – direct or indirect).
3. A merged element whose metatype is not a kind of Package, Class, DataType, Property, Association, Operation, Constraint, Enumeration, or EnumerationLiteral cannot have a receiving element with the same name and metatype unless that receiving element is an exact copy of the merged element (i.e., they are the same).
4. A PackageMerge is valid if and only if all the constraints (in this clause) required to perform the merge are satisfied.
5. Matching typed elements (e.g., Properties, Parameters) must have conforming types. For types that are Classes or Datatypes, a conforming type is either the same type or a common supertype. For all other cases, conformance means that the types must be the same.
6. A receiving element cannot have explicit references to any merged element.
7. Any redefinitions associated with matching RedefinableElements must not be conflicting.

TRANSFORMATIONS:

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

**12.2.3.4 Package Rules**

Elements that are kinds of Package match by name and metatype

CONSTRAINTS:

1. All Classifiers in the merged Package must have a non-empty qualifiedName and have isDistinguishableFrom() = **true** in the merged Package.
2. All Classifiers in the receiving Package must have a non-empty qualifiedName and have isDistinguishableFrom() = **true** in the receiving Package.

TRANSFORMATIONS:

1. A nestedPackage from the merged Package is transformed into a nestedPackage with the same name and contents in the resulting Package, unless the receiving Package already contains a nestedPackagethat matches. In the latter case, the merged nestedPackage is recursively merged with the matching receiving nestedPackage.
2. An ElementImport which is an elementImport of the receiving Package is transformed into a corresponding ElementImport in the resulting Package. Imported elements are not merged (unless there is also a PackageMerge to the Package owning the imported element).

**12.2.3.5 Class and DataType Rules**

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

TRANSFORMATIONS:

1. All Properties that are ownedAttributes of the merged Classifier are merged with the receiving Classifier to produce the resulting Classifier according to the Property transformation rules specified below.
2. nestedClassifiers are merged recursively according to the same rules.

**12.2.3.6 Property Rules**

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

CONSTRAINTS:

1. The value of isStatic of matching Properties must be the same.
2. The value of isUnique of matching Properties must be the same.
3. Any Constraints associated with matching Properties must not be conflicting.

TRANSFORMATIONS:

1. For merged Properties that do not have a matching receiving Property, the resulting Property is a Property in the resulting Classifier that is the same as the merged Property.
2. For merged Properties that have a matching receiving Property, the resulting Property is a Property with the same name and characteristics except where these characteristics are different. Where these characteristics are different, the resulting Property characteristics are determined by application of the appropriate transformation rules.
3. For matching Properties: if both Properties have isReadOnly = **true**, the resulting Property also has isReadOnly = **true**; otherwise, the resulting Property has isReadOnly = **false**.
4. For matching Properties: if both Properties have isOrdered = **false**, then the resulting Property also has isOrdered = **false**; otherwise, the resulting Property has isOrdered = **true**.
5. For matching Properties: if neither Property is designated as a subset of some derived union, then the resulting Property will not be designated as a subset; otherwise, the resulting Property will be designated as a subset of that derived union.
6. For matching Properties: different Constraints of matching Properties are all applied to the resulting Property.
7. For matching Properties: if either the merged and/or receiving elements have isUnique = **false**, the resulting element has isUnique = **false**; otherwise, the resulting element has isUnique = **true**.
8. The value of type for the resulting Property is transformed to refer to the corresponding type in the resulting Package.

**12.2.3.7 Association Rules**

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

CONSTRAINTS:

1. These rules only apply to binary Associations. (For merging n-ary associations the *default rule* is used)
2. The receiving association end must have aggregation = **composite** if the matching merged association end has aggregation = **composite**.
3. The receiving association end must be owned by the Association if the matching merged association end is owned by the Association.

TRANSFORMATIONS:

1. A merge of matching Associations is accomplished by merging the Association classifiers (using the merge rules for Classifiers) and merging their corresponding ownedEnd Properties according to the rules for Properties and the following rule for association ends.
2. For matching association ends: if neither association end is in ownedNavigableEnd, then the resulting association end is also not in ownedNavigableEnd. In all other cases, the resulting association end is in ownedNavigableEnd.

**12.2.3.8 Operation Rules**

Elements that are kinds of Operation match by name, Parameter order, and Parameter types, not including any return type.

CONSTRAINTS:

1. Operation Parameters and their types must conform to the same rules for type and multiplicity as were defined for Properties.
2. The receiving Operation must have isQuery = **true** if the matching merged Operation has isQuery = **true**.

TRANSFORMATIONS:

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

**12.2.3.9 Enumeration Rules**

Elements that are kinds of EnumerationLiteral match by owning Enumeration and Literal name.

CONSTRAINTS:

1 Matching EnumerationLiterals must be in the same order.

TRANSFORMATIONS:

1 Non-matching EnumerationLiterals from the merged Enumeration are included in the receiving Enumeration.

**12.2.3.10 Constraint Rules**

CONSTRAINTS:

1 Constraints must be mutually non-contradictory.

TRANSFORMATIONS:

1 The Constraints of the merged model elements are all added to the Constraints of the matching receiving model elements.

**12.2.3.11 Model**

A Model is a description of a system, where ‘system’ is meant in the broadest sense and may include not only software and hardware but organizations and processes. It describes the system from a certain *viewpoint* (or vantage point) for a certain category of *stakeholders* (e.g., designers, users, or customers of the system) and at a certain level of abstraction. A Model is complete in the sense that it covers the whole system, although only those aspects relevant to its purpose (i.e., within the given level of abstraction and viewpoint) are represented in the Model.

As a Package, a Model has a set of members that together describe the system being modeled. The organization of these elements varies by the modeling method being used. One approach is one or more composition hierarchies where a topmost Package/Component represents the boundary of the system. A Model may also contain elements describing relevant parts of the system’s environment. The environment is typically modeled by Actors and their Interfaces. As these are external to the system, they reside outside the Package/Component hierarchy. They may be collected in a separate Package, or owned directly by the Model as packagedElements.

Different Models can be defined for the same system, where typically the different Models are complementary and defined from the perspectives (viewpoints) of different system stakeholders. With composition of Models, a container model represents a comprehensive view of the system given by the different views defined by the contained Models.

Models can have Abstraction Dependencies between them: refinement (stereotyped by «Refine» from the Standard Profile) or mapping ( for example stereotyped by «Trace» from the Standard Profile). These are typically represented in more detail by Dependencies between the elements contained in the Models. Relationships between elements in different Models generally no direct impact on the contents of the Models because each Model is meant to be complete. However, they are useful for tracing refinements and for keeping track of cross-references between models.

**12.2.4 Notation**

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

Conformant tools may restrict the use of these notations to packagedElements*.* Optionally*,* elements that become available for use in an importing Package through a PackageImport or an ElementImport may have a distinct color or be dimmed to indicate that they are not packagedElements.

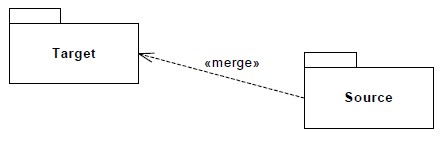
* If the members of the Package *are not* shown within the large rectangle, then the name of the Package should be placed within the large rectangle.
* If the members of the Package *are* shown within the large rectangle, then the name of the Package should be placed within the tab.

The visibility of a packagedElement may be indicated by preceding the name by a visibility symbol (‘+’ for **public** and ‘-’ for **private**). Packages may not have **protected** or **package** visibility.

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

The *URI* for a Package may be indicated with the text **{uri =** <uri>**}** following the Package name.

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



**Figure 12.4 Notation for Package Merge**

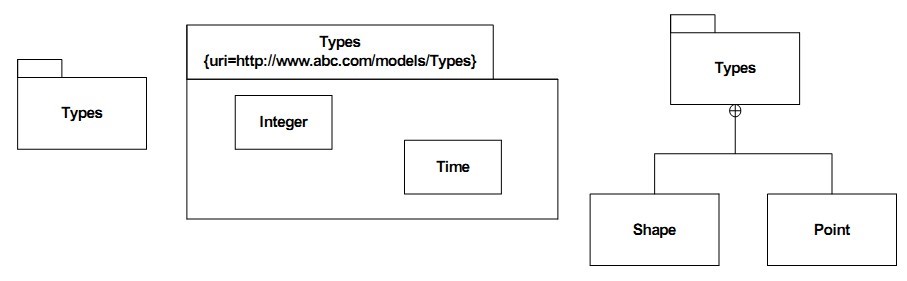
A Model is notated using the ordinary Package symbol (a folder icon) with a small triangle in the upper right corner of the large rectangle.

Optionally, especially if the members of the Model are shown within the large rectangle, the triangle may be drawn to the right of the Model name in the tab.

A Model may also be notated as a Package, using the ordinary Package symbol with the keyword «model» placed above the name of the Model.

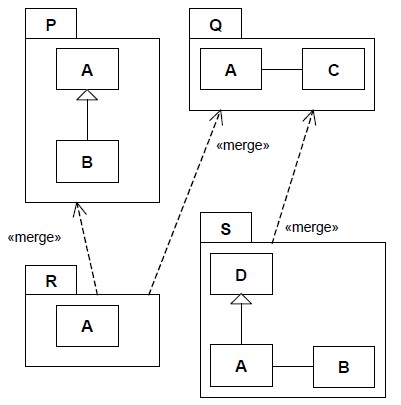
**12.2.5 Examples**

There are three alternative representations of the same Package named **Types** in Figure 12.5. The one on the left just shows the Package without revealing any of its members. The middle one shows some of the members within the borders of the Package rectangle (and also its URI), and the one to the right shows some of the members using the alternative ownership notation.



**Figure 12.5 Examples of a Package with Members**

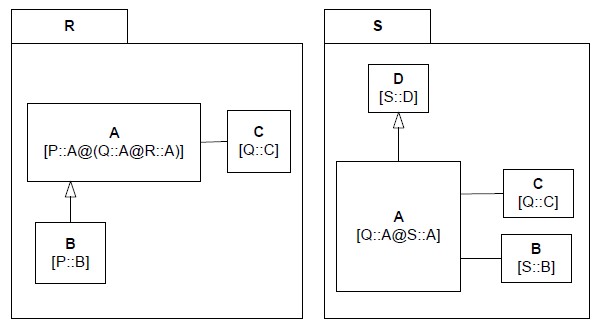
In Figure 12.6, packages P and Q are being merged by package R, while package S merges only package Q.



**Figure 12.6 Simple Example of Package Merge**

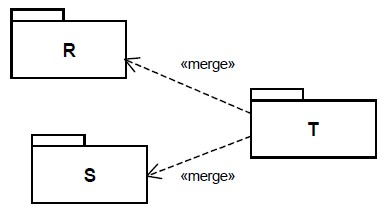
The conceptually resulting packages R and S are shown in Figure 12.7. The expressions in square brackets indicate which individual elements were merged to produce the final result, with the “@” character denoting the conceptual merge ‘transformation’ as an operator, where X@Y signifies the resulting element from the merge transformation applied to matching receiving element X and merged element Y.

**NOTE.** These expressions are not part of the standard notation, but are included here for explanatory purposes.



**Figure 12.7 Simple Example of Transformed Packages Following the Merges**

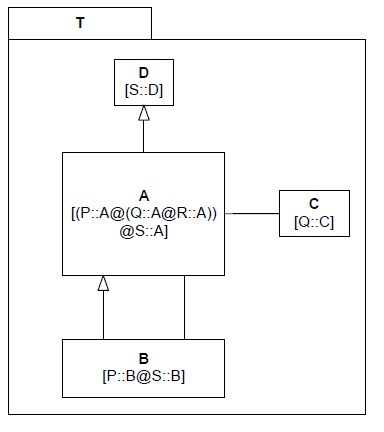
In Figure 12.8, additional PackageMerges are introduced by having Package T, which has no packagedElements of its own, merge Packages R and S defined previously.



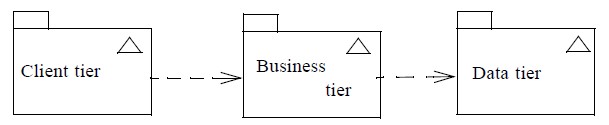
**Figure 12.8 Introducing Additional Package Merges**

In Figure 12.9, the conceptually resulting Package T is depicted. In this Package, the definitions of A, B, C, and D have all been brought together.

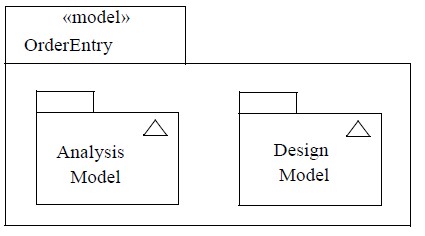
**NOTE.** The types of the ends of the Associations that were originally in the packages Q and S have all been updated to refer to the appropriate elements in Package T.



**Figure 12.9 Result of the Additional Package Merges**



**Figure 12.10 Three Models Representing Parts of a System**



**Figure 12.11 Two Views of One System Collected in a Container Model**

### 12.3 Profiles

**12.3.1 Summary**

The Profiles clause describes capabilities that allow metaclasses to be extended to adapt them for different purposes. This includes the ability to tailor the UML metamodel for different platforms (such as J2EE or .NET) or domains (such as real-time or Service Oriented Architecture). The Profiles clause is consistent with the OMG Meta Object Facility (MOF).

#### 12.3.1.1 Positioning Profiles versus Metamodels, MOF and UML

UML is reused at several meta-levels in various OMG specifications that deal with modeling. For example, MOF uses it to provide the ability to model metamodels. This clause deals with use cases comparable to the MOF at the meta-metalevel, which is one level higher than the rest of the superstructure specification. In order to allow this, the reference metamodel must be defined as an instance of UML that corresponds to its definition using MOF. Thus when defining a UML profile, the profile’s stereotypes are defined to extend the UML classes in the normative version of the UML metamodel whose XMI serialization is referenced in Annex E.

Profiles are not a first-class extension capability (i.e., it does not allow for creating new metamodels). Rather, the intention of Profiles is to give a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method. Each such adaptation is grouped in a Profile. It is not possible to remove any of the Constraints that apply to UML using a Profile, but it is possible to add new Constraints that are specific to the Profile. The only other restrictions are those inherent in this Profiles clause; there is nothing else that is intended to limit the way in which a metamodel is customized.

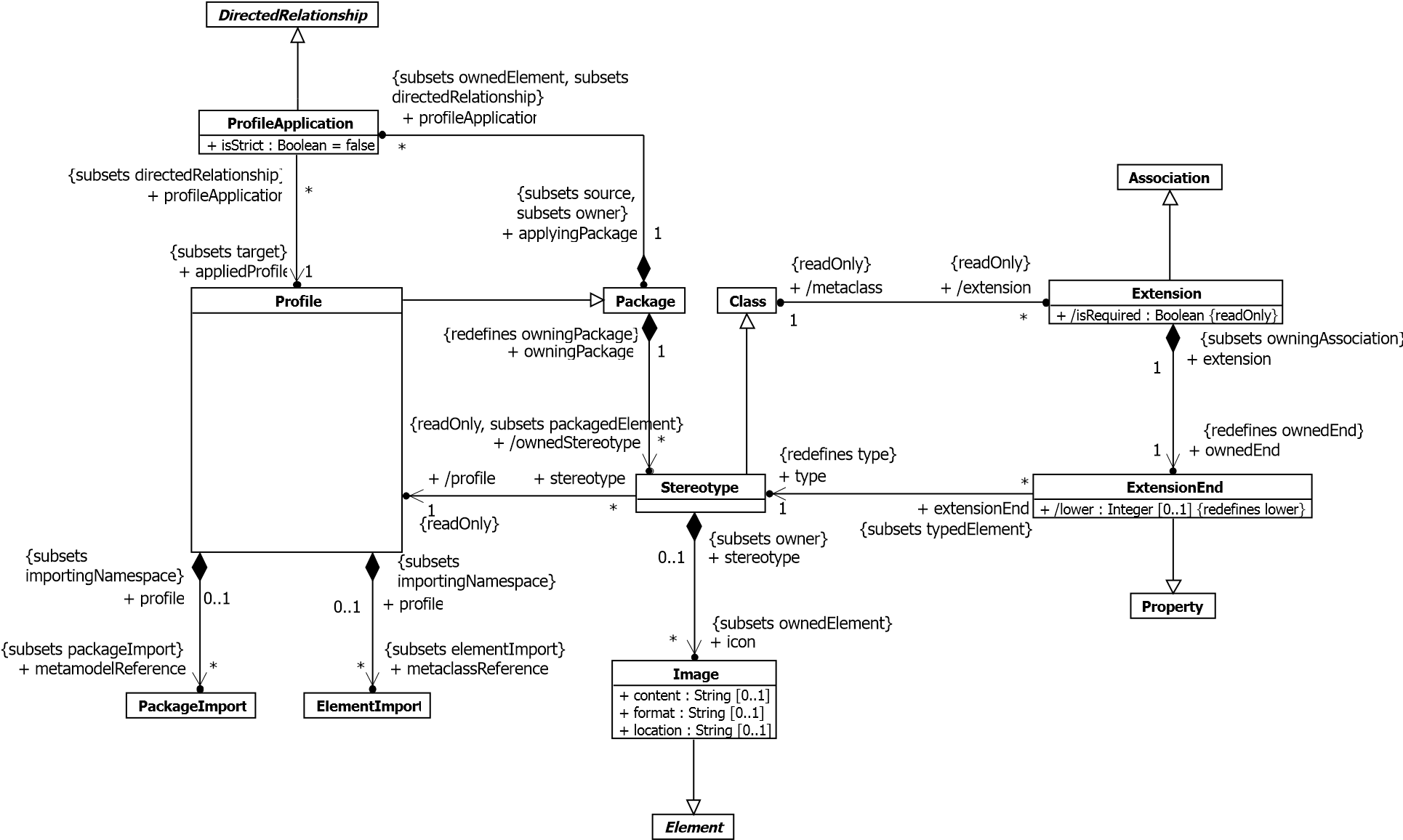
First-class extensibility is handled through MOF, where there are no restrictions at the metamodel level: it is possible to add subclasses and associations as necessary.

There are several reasons why you may want to extend UML:

* Give a terminology that is adapted to a particular platform or domain (for example EJB terminology like Home interfaces, Enterprise Java Beans, and Archives).
* Give a syntax for constructs that do not have a notation (such as in the case of Actions).
* Give a different notation for already existing symbols (such as being able to use a picture of a computer instead of the ordinary Node symbol to represent a computer in a network).
* Add additional semantics to UML or specific metaclasses.
* Add types that do not exist in UML (such as defining a timer, clock, or continuous time).
* Add Constraints that restrict the way UML’s constructs are used (such as disallowing multiple inheritance).
* Add information that can be used when transforming a model to another model or code (such as defining mapping rules between a model and Java code).

There is no simple answer for when to create a new metamodel, when to create a new profile, and when to create both (one for UML tooling, the other for MOF-based tooling).

**12.3.2 Abstract Syntax**



**Figure 12.12 Profiles**

**12.3.3 Semantics**

**12.3.3.1 Profiles**

A Profile is a restricted form of metamodel that can be used to extend UML, as described below. The primary extension construct is the Stereotype.

**12.3.3.1.1 Restricting Availability of UML Elements**

The metaclassReference ElementImports and metamodelReference PackageImports may be used to specify the Profile’s filtering rules. The filtering rules determine which UML elements are *available* when the Profile is applied and which ones are *hidden*.

**NOTE.** Applying a Profile to a model does not change that model in any way; it merely defines a view of the underlying model.

The effects of a metaclass being hidden (not available) are as follows:

* It is not possible to create new instances of that metaclass (or its subclasses).
* Existing instances of that metaclass (or its subclasses) can no longer be seen in diagrams or selected in lists, including browser panes.
* Relationships with existing instances of that metaclass (or its subclasses) can no longer be seen in diagrams or selected in lists, including browser panes.

Tools may vary in how they implement the above – for example they may hide the metaclass/instances completely in selection lists or make them grayed out/unselectable .

In order for the filtering rules (described further below) on a Profile to be activated, the Profile must be applied in *Strict* mode: specifically the isStrict attribute on the ProfileApplication must be set to true; otherwise the filtering rules are ignored for this profile application.

The most common case is when a Profile imports UML itself using a metamodelReference. A conformant tool may provide this as built-in behavior when the user creates a Profile. In that case, every UML metaclass is available.

Alternatively, specific metaclasses could be referenced through metaclassReferences and only those would then be available. A further option is to use one or more metamodelReferences to Package(s) that contain ElementImports for a subset of UML metaclasses. This allows the set to be reusable across many Profiles without having to specify individual metaclassReferences each time.

The visibility and alias properties of ElementImports are ignored when it is used as a metaclassReference*.*

Where both a metaclassReference and a metamodelReference are present on a profile, the latter is ignored and only the specific metaclasses are available.

In detail, the following rules are used to determine whether a model element is available after a Profile has been applied in Strict mode. Metaclasses and their instances are availableif they are:

1. referenced by an explicit metaclassReference, or
2. (in the absence of a metaclassReference) members (directly or transitively) of a Package that is referenced by an explicit metamodelReference, or
3. extended by a Stereotype which is a member of the applied profile (even if the extended metaclass itself is not available).

All other model elements are *hidden* (not available) when the Profile is applied in Strict mode.

This makes invalid the combination of applied profiles that specify non-overlapping (disjoint) sets of available metaclasses.

If a Profile P1 imports another Profile P2, then all metaclassReference and metamodelReference associations will be combined at the P1 level, and the filtering rules apply to this union. Stereotypes imported from another Profile using ElementImport or PackageImport are added to the namespace members of the importing profile.Profile Contents.

A Profile can define or import Classes, Associations, DataTypes, PrimitiveTypes and Enumerations as well as Stereotypes. More precisely all the constraints of a CMOF-compliant metamodel apply to a UML Profile. These are defined in detail in Section 14.4 of the MOF Core Specification. The effect of these constraints is that, except for Stereotypes and Extensions, all other Types defined or imported in a Profile must be exactly one of the Types explicitly mentioned in the above subset and that no specialization outside this subset is allowed. The term Profile-defined Type corresponds to a CMOF-compliant Class, Association, DataType, PrimitiveType or Enumeration defined or imported in a Profile.

Profile-defined Types can only be used as the type of Properties in that Profile or as a general classifier of another

Profile-defined Type. They cannot be used as Types in models the Profile is applied to, such as the type of a

TypedElement, the classifier of an InstanceSpecification or the general or specific classifier in a Generalization relationship. It is however possible to define these types in separate Packages and import them as needed in both Profiles and model Packages in order to use them for both purposes.

Stereotypes can participate only in binary Associations. The opposite class can be another Stereotype, a non-Stereotype Class that is a packagedElement of a Profile (directly or indirectly), or a UML metaclass. For these Associations there must be an ownedAttribute Property typed by the opposite class. Where the opposite class is not a stereotype, the opposite property must be an ownedMember of the Association itself rather than the other class/metaclass. The effect of these rules is that Associations in a Profile are not required to involve a Stereotype but may not own both of their Ends. According to CMOF-compliant metamodel constraints, Profile-defined binary Associations may have at most one end with aggregation = AggregationKind::composite; other ends shall have aggregation = AggregationKind::none. Furthermore, a Property of a Stereotype or Profile-defined Class can have composite aggregation if and only if its type is a Profiledefined Class whereas a Property of a Stereotype or Profile-defined Class or DataType shall have aggregation = AggregationKind::none if its type is a Profile-defined DataType, PrimitiveType, or Enumeration.

The most direct implementation of the Profile capability that a tool can provide is by having a metamodel based implementation, similar to the Profile metamodel. However, this is not a requirement of the current standard, which requires only the support of the specification, and the standard XMI based interchange capacities. The Profile capability has been designed to be implementable by tools that do not have a metamodel-based implementation. Practically any mechanism used to attach new values to model elements can serve as a valid profile implementation; however, creating such values requires a limited metamodel-like capability for creating and referring to instances of Profile-defined Classes and DataTypes as the values of Properties typed by such Classes or DataTypes and for referring to instances of Profile-defined Classes for creating link instances of Profile-defined Associations. As an example, the UML1.4 profile metamodel could be the basis for implementing a UML2-compliant profile tool.

**12.3.3.1.2 Integrating and Extending Profiles**

There is a number of ways to create, extend, and integrate Profiles. These are described briefly in this sub clause in order to foster better profile integration and reuse.

The simplest form of Profile integration is to simply apply multiple Profiles to the same Package. This requires no integration between the Profiles at all. Such Profiles might be designed to complement each other, addressing different concerns.

It is also possible for one Profile to reuse all of or parts of another, and to extend other Profiles. Like any other Class, Stereotypes can be defined in Packages or Profiles that can be factored for reuse. These Stereotypes can be directly reused by being referenced or specialized in other Profiles. Normal rules apply as to whether a referenced Stereotype is visible to users of the extending Profile.: a public import is needed to ensure that Stereotypes from other profiles are visible after applying the extending one.

For example, the *Unified Profile for DoDAF and MODAF* (UPDM) Profile could integrate with the SysML Profile to reuse Stereotypes such as Requirement and ViewPoint. UPDM could be designed to use ViewPoint in a manner that is semantically consistent with SysML. However UPDM could extend ViewPoint with additional properties and associations for its purposes. The UPDM specification could note to users that ViewPoint is a stereotype in UPDM that represents a "placeholder" to ViewPoint in SysML. Users could then apply UPDM to a model, and get UPDM's ViewPoint capabilities without any coupling with, or need for SysML. UPDM could then provide another compliance point that merges with the SysML profile resulting in stereotypes Requirement and ViewPoint having the capabilities of both profiles. The SysML::ViewPoint would be merged with the UPDM::ViewPoint allowing the shared semantics to be supported without making any changes to the existing model. Users who want UPDM with SysML would then apply this merged profile.

**12.3.3.1.3 MOF-Equivalent Semantics**

This sub clause specifies the semantics of Stereotypes and their instances using MOF. That does not mean that tools need implement Profiles using MOF, but that a non-MOF-based implementation must do whatever is necessary under the covers to ensure it behaves, in all observable ways, as if it were a MOF implementation.

The same mapping to MOF is used to determine how to serialize applied profiles using XMI. A Profile is an instance of the UML2 metamodel, not a CMOF metamodel. Therefore the MOF to XMI mapping rules do not directly apply for instances of a Profile. Figure 12.15 is an example of a mapping between a UML2 Profile and an equivalent CMOF model. This mapping is used as a means to explain and formally specify how Profiles are serialized and exchanged as XMI. Using the following Profile to CMOF mapping rules, the XMI specification can be used to determine how Profiles, and models with Profiles applied, are represented in XMI. In the mapping:

* A Profile maps to a CMOF Package.
* A Stereotype maps to a CMOF class with the same name and properties.
* A Metaclass is already a CMOF class so it maps to itself.
* An Extension maps to an Association as described in the Semantics sub clause of Extension.
* Any other elements in the Profile (i.e., non-Stereotype Classes, DataTypes, PrimitiveTypes, Enumerations and Associations) are treated as MOF elements.
* An instance of a Stereotype (created when the Stereotype is applied to an Element) maps to an instance of the

CMOF class representing the Stereotype. This stereotype instance is compositionally associated with the Element to which it applies using a Link that is an instance of the composite Association to which the Extension is mapped.

For a Profile the *URI* Property (inherited from Package) is used to determine the nsURI to be used to identify instances of the Profile in XMI.

**NOTE.** By default the name attribute of the Profile is used for the nsPrefix in XMI but this can be overridden by the CMOF tag org.omg.xmi.nsPrefix.

OMG normative Profiles, such as the UML Standard Profile, follow an OMG normative naming scheme for URIs. For non-standard profiles a recommended convention is:

nsUri = http://<profileParentQualifiedName>/<version>/<profileName>.xmi nsPrefix = *<profileName>*

where:

* <profileParentQualifiedName>is the qualified name of the Package containing the Profile (if any) with / (forward slash) substituted for ::, and all other illegal XML QName characters removed.
* *<*version*>* is a version identifier.

**NOTE.** For OMG normative profiles this is a date in the format YYYYMMnn where nn is a serial number within the month, and represents the version of the Profile XMI not that of the specification which might be reissued without affecting the XMI.

* <profileName> is the name of the Profile.

A Profile can be exchanged just like any model, as an XMI file, and models that have a Profile applied can also be interchanged.

Figure 12.19 shows a Stereotype named Home extending the Interface UML2 metaclass. Figure 12.15 illustrates the MOF correspondence for that example, basically by introducing an Association from the Home MOF class to the Interface MOF class. For illustration purposes, we add a Property “magic:String” to the Home Stereotype.

The first serialization below shows how the model in Figure 12.19 (definition of the Profile extending the UML2 metamodel) can be exchanged.

<?xml version="1.0" encoding="UTF-8"?>

<xmi:XMI xmlns:xmi=http://www.omg.org/spec/XMI/YYYYMMnn>

xmlns:mofext=http://www.omg.org/spec/MOF/YYYYMMnn xmlns:uml=http://www.omg.org/spec/UML/YYYYMMnn

<uml:Profile xmi:id="id0" xmi:type=”uml:Profile” name="HomeExample">

<metamodelReference xmi:type=”uml:PackageImport” xmi:id="id2">

<importedPackage href="http://www.omg.org/spec/UMLYYYYMMnn/UML.xmi#\_0"/>

</metamodelReference >

<packagedElement xmi:type="uml:Stereotype" xmi:id="id3" name="Home">

<ownedAttribute xmi:type="uml:Property" xmi:id="id5" name="base\_Interface" association="id6">

<type href="http://www.omg.org/spec/UML/YYYYMMnn/UML.xmi#Interface"/>

</ownedAttribute>

</packagedElement>

<packagedElement xmi:type="uml:Extension" xmi:id="id6" name="A\_Interface\_Home" memberEnd="id7 id5">

<ownedEnd xmi:type="uml:ExtensionEnd" xmi:id="id7" name="extension\_Home" type="id3" aggregation="composite">

</ownedEnd>

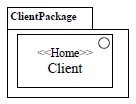
</packagedElement>

</uml:Profile>

<mofext:Tag xmi:type=”mofext:Tag” name="org.omg.xmi.nsPrefix" value="HomeExample"/>

<mofext:Tag xmi:type=”mofext:Tag” name="org.omg.xmi.nsURI" value="http://HomeExample/20120501/HomeExample.xmi"/> </xmi:XMI>

Figure 12.13 is an example model that includes an instance of Interface extended by the Home stereotype.



**Figure 12.13 Using the HomeExample Profile to Extend a Model**

The XMI serialization of a model to which zero or more Profiles are applied is an XMI file organized in two logical parts (which may be physically organized in any order within a file or in separate files):

1. the XMI serialization of the model,
2. the XMI serialization of the instances corresponding to each application of a Profile to the model or some part of it.

Since deleting the application of Profiles applied to the model or some parts of it must not modify the XMI serialization of the model itself, all XMI elements in Part (1) cannot have any XMI reference to any XMI element in Part (2). Typically, the values of the applied Stereotype’s “base” properties and of properties typed by metaclasses cause XMI elements corresponding to instances of Stereotypes in Part (2) to make reference to XMI elements in Part (1). In general, Part (2) contains the following kinds of instances:

* Instances of Stereotypes (see the example in Figure 12.13).
* Optionally, instances of Extensions according to their MOF-equivalent mapping as composite Associations.
* Instances of Profile-defined Classes and DataTypes. In particular, such instances should not be confused, replaced, or substituted with InstanceSpecifications, which are a UML-based model representation of instances but are not the same as, substitutable with or equivalent to the instances that they represent.
* Optionally, instances of Profile-defined composite and non-composite Associations.

The XMI below shows how the model of Figure 12.13 is serialized in XMI. A tool importing that XMI file can filter out the elements related to the HomeExample Profile, if the tool does not have this Profile definition.

<?xml version="1.0" encoding="UTF-8"?>

<xmi:XMI xmlns:xmi="http://www.omg.org/spec/XMI/YYYYMMnn" xmlns:uml="http://www.omg.org/spec/UML/YYYYMMnn"

xmlns:HomeExample="http://HomeExample/20120501/HomeExample.xmi">

<uml:Package xmi:type=”uml:Package” xmi:id="id1" name="ClientPackage">

<profileApplication xmi:type=”uml:ProfileApplication” xmi:id="id3">

<appliedProfile href="http://HomeExample/20120501/HomeExample.xmi#id0"/>

</profileApplication>

<packagedElement xmi:type="uml:Interface" xmi:id="id2" name="Client"/>

</uml:Package>

<!-- applied stereotypes -->

<HomeExample:Home xmi:id= "id4" base\_Interface="id2"/>

</xmi:XMI>

**12.3.3.2 Defining Profiles for Non-UML Metamodels**

In theory the Profiles capability can be used to define extensions for metamodels other than UML, though this capability has rarely, if at all, been used in practice. It would require any tooling implementing that metamodel to also support some kind of profile application mechanism – that is outside the scope of this specification. The following describes how the Profile definition mechanism may be used in this way.

In addition to UML, a Profile may be related to another MOF-compliant *reference metamodel*. In general a reference metamodel typically consists of metaclasses that are either imported or locally owned. All metaclasses that are extended by a profile have to be members (directly or indirectly) of the same reference metamodel. The metaclassReference ElementImports and metamodelReference PackageImports serve two purposes: (1) they identify the reference metamodel elements that are imported by the profile and (2) they specify the Profile’s filtering rules. The filtering rules determine which elements of the metamodel are *available* when the Profile is applied and which ones are *hidden*.

**NOTE.** Applying a Profile does not change the underlying model in any way; it merely defines a view of the underlying model.

In general, only model elements that are instances of imported reference metaclasses will be visible when the profile is applied. Instances of all other metaclasses will be hidden and further instances may not be created. By default, model elements whose metaclasses are owned by the reference metamodel are visible. This applies transitively to any subpackages of the reference metamodel according to the default rules of package import. If any metaclass is imported using a metaclassReferenceElementImport, then model elements whose metaclasses are the same as that metaclass are available. However, a metaclassReference blocks a metamodelReferencewhenever an element or Package of the referenced metamodel is also referenced by a metaclass reference. In such cases, only the elements that are explicitly referenced by the metaclassReference will be visible, while all other elements of the metamodel Package will be hidden.

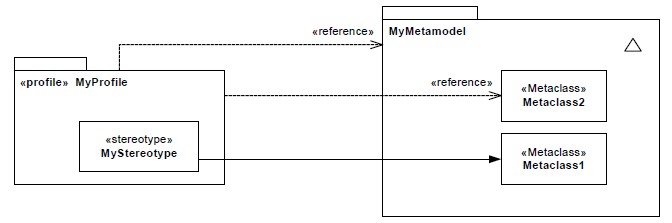
The following rules are used to determine whether a model element is available or hidden after a Profile has been applied. Model elements are *available* if they are instances of metaclasses that are:

1. referenced by an explicit metaclassReference, or
2. contained (directly or transitively) in a Package that is referenced by an explicit metamodelReference; unless there are other elements of subpackages of that Package that are explicitly referenced by a MetaclassReference, or
3. extended by a Stereotype owned by the applied profile (even if the extended metaclass itself is not visible).

All other model elements are hidden(not available) when the Profile is applied.

The most common case is when a Profile just imports an entire metamodel using a metamodelReference. In that case, every element instantiating a metaclass in the metamodel is visible.

In the example in Figure 12.14, MyMetamodel is a metamodel containing two metaclasses: Metaclass1 and Metaclass2. MyProfile is a profile that references MyMetamodel and Metaclass2. However, there is also an explicit metaclass reference to Metaclass2, which overrides the metamodel reference. An application of MyProfile to some model based on MyMetamodel will show instances of Metaclass2 (because it is referenced by an explicit metaclass reference). Also, those instances of Metaclass1 that are extended by an instance of MyStereotype will be visible. However, instances of Metaclass1 that are not extended by MyStereotype remain hidden.



**Figure 12.14 Specification of an Available Metaclass**

If a Profile P1 imports another Profile P2, then all metaclassReference and metamodelReference associations will be combined at the P2 level, and the filtering rules apply to this union.

The filtering rules defined at the Profile level are, in essence, merely a suggestion to modeling tools on what to do when a profile is applied to a model.

The isStrict attribute on a ProfileApplication specifies that the filtering rules have to be applied strictly. If isStrict is true on a ProfileApplication, then no other metaclasses than the accessible one defined by the profile shall be accessible when the Profile is applied on a model. This prohibits the combination of applied profiles that specify different accessible metaclasses.

**12.3.3.3 ProfileApplication**

A ProfileApplication is used to record which Profiles have been applied to a Package.

One or more Profiles that extend UML may be applied at will to a model Package. Applying a Profile means that it is possible to apply the Stereotypes that are defined as part of the Profile. It is possible to apply multiple Profiles to a Package, though this could make the Package invalid if they have conflicting Constraints. Applying a Profile means recursively applying all its nested and imported Profiles. Stereotypes that are public members of a Profile may be applied to applicable model elements in Packages to which the Profile has been applied.

When a Profile is applied, instances of the appropriate Stereotypes must be created for those elements that are instances of metaclasses with ExtensionEnds which have isRequired = **true**. The model is not well-formed without these instances.

Once a Profile has been applied to a Package, it is allowed to remove the applied Profile at will. Removing a Profile implies that all elements that are instances of Stereotypes defined in the Profile are deleted including the instances of Profile-defined Classes they compositionally aggregate and the instances of Profile-defined composite Associations linking them. Other instances that are not compositionally aggregated must also be deleted if their defining type is no longer accessible from any other Profile applied to the same model.The removal of an applied Profile leaves the instances of elements from the referenced metamodel intact. It is only the instances of the elements from the Profile that are deleted. This means that for example a profiled UML model can always be interchanged with another tool that does not support the profile and be interpreted as a pure UML model.

A Profile which is a packagedElement of another Profile can be applied individually. However, the nested Profile must specify any required metaclass and/or metamodel references if it contains any Stereotypes and may use PackageImport to indicate other Profiles to be co-applied. Metaclass and/or metamodel references are not inherited from a containing Profile.

**12.3.3.4 Stereotypes**

A Stereotype defines an extension for one or more metaclasses, and enables the use of specific terminology or notation in place of, or in addition to, the ones used for the extended metaclasses. If a Stereotype extends several metaclasses, it can only be applied to exactly one instance of one of those metaclasses at any point in time. It is, however, possible to detach the Stereotype instance from an instance of one metaclass and attach it to an instance of another metaclass.

A Stereotype is a limited kind of metaclass that cannot be used by itself, but must always be used in conjunction with one of the metaclasses it extends. Each Stereotype may extend one or more metaclasses through association (Extension) rather than generalization/specialization. Similarly, a metaclass may be extended by one or more Stereotypes. Relating an instance “S” of Stereotype to a metaclass “C” from UML using an “Extension” (which is a specific kind of Association) signifies that model elements of type “C” can be extended by an instance of “S” (see example in Figure 12.24 Defining a Stereotype). At the model level (such as in Figure 12.29) instances of “S” are related to “C” model elements (instances of “C”) by links (occurrences of the Association/Extension from “S” to “C”).

Any metaclass referenced by a metaclassReference or contained in a Package referenced by metamodelReference of the closest Profile directly or indirectly containing a Stereotype can be extended by the Stereotype. For example States, Transitions, Activities, Use Cases, Components, Properties, Dependencies, etc. can all be extended with Stereotypes if the metamodelReference is UML. A Stereotype may be contained in a Package in which case the metaclasses available for extension are those referenced by the closest parent Profile containing the Package.

Just like a Class, a Stereotype may have Properties, which have traditionally been referred to as Tag Definitions. When a Stereotype is applied to a model element, the values of the Properties have traditionally been referred to as *tagged values*. Stereotype specializes Class and its Properties have the same meaning in Stereotypes as they do in Class. A Stereotype Property can have composite aggregation; just like the value of a composite aggregation Property on a Class is owned by an instance of that Class, the value of a composite aggregation Property on a Stereotype is owned by an instance of that Stereotype. Since a profile can be unapplied without modifying the model it was originally applied to, instances of metaclasses in the model cannot refer to instances of stereotypes or to values of their properties. The type of a composite aggregation Stereotype Property cannot be a Stereotype (since Stereotypes are owned by their Extensions) or a metaclass (since instances of metaclasses are owned by other instances of metaclasses); however, the type of such Property can be a Class defined in the Profile or a DataType defined in the Profile or accessible via import or via the Profile’s metamodel reference.Tool vendors may choose to support extensibility that includes owned operations and behaviors, but are not required to do so. Tools must however support Stereotype ownedAttributes.

Its Profile or Package defines the namespace for the Stereotype. When Profiles are applied to a Package, the available Stereotypes for use are defined by the applied Profiles, and these Stereotypes can be displayed using the fully qualified name if needed in order to distinguish Stereotypes with the same name in different Profiles or Packages. PackageImport and ElementImport can be used to allow the use of unqualified names. Stereotypes directly owned by an applied Profile (ownedStereotype) may be used without qualified names.

**12.3.3.5 Images**

The Image class provides the necessary information to display an Image in a diagram. Icons are typically handled through the Image class.

Information such as physical placement or format is provided by the Image class. The Image class provides a generic way of representing images in different formats. Although some predefined values are specified for format for convenience and interoperability, the set of possible formats is open ended. However there is no requirement for a tool to be able to interpret and display any specific format, including those predefined values.

The format property indicates the format of the content, which is how the string content should be interpreted. The following values are reserved: **SVG**, **GIF**, **PNG**, **JPG**, **WMF**, **EMF**, **BMP**. In addition the prefix ‘**MIME:**’ is also reserved: this must be followed by a valid MIME type as defined by RFC3023. This option can be used as an alternative to express the reserved values above, for example “SVG” could instead be expressed “MIME: image/svg+xml.”

**12.3.3.6 Extensions**

An Extension is used to indicate that the properties of a metaclass are extended through a Stereotype, and gives the ability to flexibly add (and later remove) stereotypes to classes.

Extension is a kind of Association. One end of the Extension is an ordinary Property and the other end is an ExtensionEnd. The former ties the Extension to a (meta)Class, while the latter ties the Extension to a Stereotype that extends the Class.

A required Extension (isRequired = **true)** means that an instance of this Stereotype must be linked to each instance of the extended metaclass in the model to which the containing Profile has been applied (otherwise the model is not wellformed). If the extending Stereotype has subclasses, then at most one instance of the Stereotype or one of its subclasses is required.

A non-required Extension (isRequired = **false)** means that an instance of this Stereotype may be linked to an instance of an extended metaclass at will, and also later deleted at will; however, there is no requirement that each instance of a metaclass be stereotyped. However the same stereotype (or its subtypes) can never be applied twice to the same element. An instance of a Stereotype is deleted when either the instance of the extended metaclass is deleted, or when the Profile defining the stereotype is removed from the appliedProfiles of the Package.

The equivalence to a MOF construction for single metaclass extension is shown in Figure 12.15. This figure illustrates the case shown in Figure 12.19, where the Stereotype named Home extends the Interface metaclass. In this figure, Interface is an instance of the UML metaclass (a CMOF Class) and Home is an instance of a Stereotype (also considered a CMOF Class for this purpose). The MOF construct equivalent to an Extension is a composition from the extended metaclass to the extension Stereotype, owned by the extended metaclass. When the Extension is required, then the multiplicity of the property typed by the extension Stereotype is **1**.

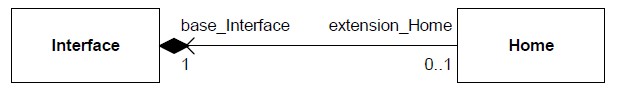
The name of the Property typed by the extended metaclass is:

*‘base\_’ extendedMetaclassName*

The name of the Property typed by the extension Stereotype (the ExtensionEnd) is: *‘extension\_’ stereotypeName*

Constraints are frequently added to Stereotypes. The above Properties may be used for expressing OCL navigations. For example, the following OCL expression states that a Home Interface shall not have attributes:

self.base\_Interface.ownedAttributes->isEmpty()



**Figure 12.15 MOF Model Equivalent to Extending "Interface" by the "Home" Stereotype**

An example for multiple metaclass extension is depicted in Figure 12.16. The Stereotype TestCase extends both metaclass Operation and Behavior.

<

<

Metaclass

>>

**O**

**p**

**e**

**r**

**a**

**ti**

**o**

**n**

<<

Metaclass

>>

**Behavior**

<<

stereotypes

>>

**TestCase**

**Figure 12.16 Example of Multiple Metaclass Extension**

The corresponding equivalence to a MOF construction for multiple metaclass extension is shown in Figure 12.17.

**T**

**e**

**s**

**t**

**C**

**a**

**s**

**e**

**Operation**

**Behavior**

b

a

s

e

\_

Operation

[

0

..

1

]

extension

\_

TestCase

[

0

..

1

]

extension

\_

TestCase

[

0

..

1

]

base

\_

Behavior

[

0

..

1

]

x

o

r

**Figure 12.17 MOF Model Equivalent to Multiple Metaclass Extension**

**12.3.3.7 ExtensionEnd**

An ExtensionEnd is used to tie an Extension to a Stereotype when extending a metaclass: it is a navigableOwnedEndofthe Extension, avoiding an extra ownedAttribute on the extended Class. It is always typed by a Stereotype and must always have isComposite = **true**.

The default multiplicity of an ExtensionEnd is 0..1. It may be 1..1 if the Stereotype is required but the upperBound may never be more than 1.

**12.3.4 Notation**

The notation for an Extension is an arrow pointing from a Stereotype to the extended Class, where the arrowhead is shown as a filled triangle. An Extension may have the same adornments as an ordinary Association, but they are typically elided and navigability arrows are never shown. If isRequired = **true**, the adornment **{required}** is shown near the ExtensionEnd.



**Figure 12.18 The Notation for an Extension**

It is possible to use the multiplicities 0..1 or 1 on the ExtensionEnd as an alternative to the adornment **{required}**. Due to how isRequired is derived, the multiplicity 0..1 corresponds to isRequired = **false**.

A Profile uses the same notation as a Package, with the addition that the keyword «profile» is shown before or above

the name of the Package. Profile::metaclassReferenceand Profile::metamodelReferenceuse the same notation as Package::elementImportand Package::packageImport, respectively but with the keyword «reference».

ProfileApplications are shown using a dashed arrow with an open arrowhead from the Package to each applied Profile. Either the keyword «apply» is shown near the arrow, or the keyword «strict» - the latter if isStrict = true.

If multiple appliedProfiles have Stereotypes with the same name, it may be necessary to qualify the name of the Stereotype (with the profile name).

A Stereotype uses the same notation as a Class, with the addition that the keyword «stereotype» is shown before or above the name of the Class.

When a Stereotype is applied to a model element (an instance of a Stereotype is linked to an instance of a metaclass), the name of the Stereotype is shown within a pair of guillemets above or before the name of the model element, or where the name would appear if the name is omitted or not displayed. For model elements that are not NamedElements but do have a graphical representation, unless specifically stated elsewhere, the stereotypes can be displayed within a pair of guillemets near the upper right corner of the graphical representation. If multiple stereotypes are applied, the names of the applied stereotypes are shown as a comma-separated list within a pair of guillemets. When the extended model element has a keyword, then the stereotype name(s) will be displayed close to the keyword, within the same or separate guillemets (example: «interface» «Clock» or «Clock, interface»).

Normally a Stereotype’s name starts with an upper-case letter, to follow the convention for naming Classes. However Profiles may use different conventions. Matching between the names of Stereotype definitions and applications is caseinsensitive, so naming stereotype applications with lower-case letters where the stereotypes are defined using uppercase letters is valid, although stylistically obsolete. For legacy reasons a tool may display stereotype names with the initial letter in lower case even when defined in upper case.

A tool can choose whether it will display Stereotypes or not. In particular, tools can choose not to display *required* stereotypes, but to display only the values of their ownedAttributes if any.

The values of the ownedAttributes of a Stereotype (or its generalizations) applied to a model element can be shown in one of the following three ways:

1. As part of a comment symbol connected to the graphic node representing the model element.
2. In separate compartments of the graphic node representing that model element.
3. Above the name string within the graphic node or, else, before the name string.

In the case where a compartment or comment symbol is used, the stereotype name may be shown in guillemets before the name string in addition to being included in the compartment or comment.

The values are displayed as name-value pairs:

<namestring> ‘=’ <valuestring>

If a Stereotype Property is multi-valued, then the <valuestring> is displayed as a comma-separated list: <valuestring> ::= <value> [‘,’ <value>]\* Certain values have special display rules:

* + - * As an alternative to a name-value pair, when displaying the values of Boolean Properties, tools may use the convention that if the <namestring> is displayed, then the value is **true**; otherwise, the value is **false**.
      * If the value is the name of a NamedElement, then, optionally, the qualifiedName of that element can be displayed.

If compartments are used to display Stereotype Property values, then an additional compartment is required for each applied Stereotype whose Property values are to be displayed. Each such compartment is headed by the name of the applied stereotype in guillemets. Such compartments are only applicable to elements for which compartments generally may be used: specifically Classifiers and States.

Within a comment symbol, or, if displayed before or above the model element’s name*,* the Property values from a specific Stereotype are optionally preceded with the name of the applied Stereotype within a pair of guillemets. This is useful if values of more than one applied stereotype should be shown.

When displayed in compartments or in a comment symbol, at most one namestring-valuestring pair can appear on a single line. When displayed above or before a model element’s name, the name-value pairs are separated by semicolons and all pairs for a given stereotype are enclosed in braces.

**12.3.4.1 Icon presentation**

It is possible to attach Images to a Stereotype that can be used in lieu of, or in addition to, the normal notation of a model element to which the Stereotype is applied.

When a Stereotype has a value for icon, the referenced Image can be graphically attached to the model elements to which the Stereotype has been applied. Every model element that has a graphical presentation can have an attached icon. When model elements are graphically expressed as:

* + - * Boxes (see Figure 12.25): the box may be replaced by the Image, and the name of the model element appears below the Image. This presentation option can be used only when a model element has one

single Stereotype applied and when Properties of the model element (e.g., ownedAttributes, ownedOperations of a Class) are not presented. As another option, the Image may be presented in a reduced size, inside and to the top of the box representing the model element. When several Stereotypes are applied, several Images may be presented within the box.

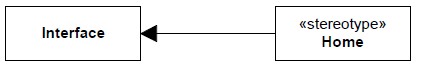
* + - * Lines: the Image may be placed close to the line.
      * Textual notation: the Image may be presented to the left of the textual notation.

Several Images may be referenced by a Stereotype’s icon Property. The interpretation of the different attached Images in that case is a semantic variation point. Some tools may use the different Images for different purposes: the icon replacing the box, for the reduced-size icon inside the box, for icons within tree browsers, etc. Alternatively, depending on the Image format, tools may choose to scale one single Image into different sizes for these different purposes.

Some model elements already use an icon for their default presentation. A typical example of this is the Actor model element, which uses the “stickman” icon. When a Stereotype with an icon is applied to such a model element, the Stereotype’s icon replaces the default presentation icon within diagrams.

**12.3.5 Examples**

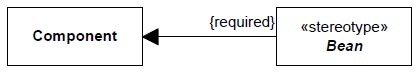
In Figure 12.19, a simple example of using an Extension is shown, where the stereotype Homeextends the metaclass Interface.



**Figure 12.19 Example of Using an Extension**

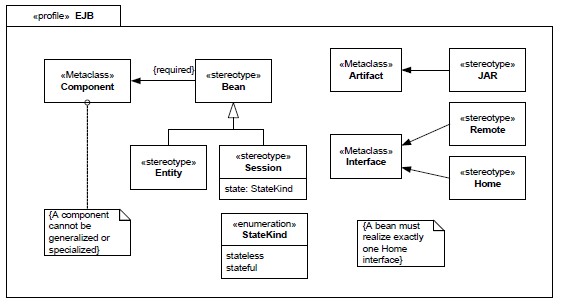
An instance of the stereotype Homecan be added to and removed from an instance of the class Interfaceat will, which provides for a flexible approach of dynamically adding (and removing) information specific to a Profile to a Package.

In Figure 12.20, each instance of metaclass Component in a model to which the Profile has been applied must have applied an instance of the stereotype Bean*,* as the Extension has isRequired = **true**. (As the stereotype Beanis abstract, this means that each instance of metaclass Component must be stereotyped by an instance of one of its concrete subclasses.) The model is not well-formed unless such a Stereotype is applied. This provides a way to express Extensions that should always be present for all instances of the base metaclass depending on which Profiles are applied.



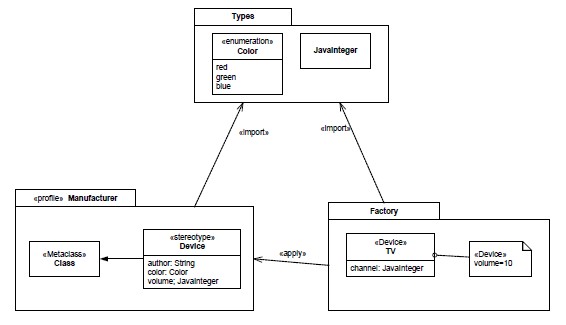
**Figure 12.20 Example of a Required Extension**

In Figure 12.21, a simple example of an EJB profile is shown.



**Figure 12.21 Defining a Simple EJB Profile**

The Profile defines that the abstract stereotype Beanis required to be applied to metaclass Component, which means that an instance of either of the concrete subclasses Entityand Sessionof Beanmust be linked to each instance of Component. The Constraints that are part of the Profile are evaluated when the Profile is applied to a Package, and these Constraints need to be satisfied in order for the model to be well-formed.



**Figure 12.22 Importing a Package from a Profile**

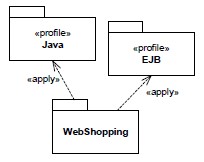
In Figure 12.22, the Package named **Types**is imported by the Profile named **Manufacturer**. The Enumeration named **Color** and the Class named **JavaInteger**are then used as the type of Properties of the Stereotype named Device as well as the standard PrimitiveType **String**.

If the Profile Manufactureris later applied to a Package, then the types from **Types**are not available for use in the

Package to which the Profile is applied unless package **Types**is explicitly imported. This means that the class **JavaInteger**can be used as the type of a Stereotype Property (e.g., in **Device**) but not as an ordinary Property (as part of the Class **TV**) unless Package **Factory**also imports Package **Types**(which it does).

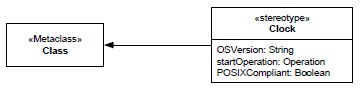
**NOTE.** The value of the volume Property is displayed once the Stereotype **Device**has been applied to the Class **TV**.

Given the profiles Javaand EJB, Figure 12.23shows how these may be applied to the Package **WebShopping**.

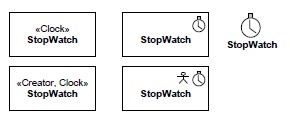


**Figure 12.23 Profiles Applied to a Package**

In Figure 12.24, a simple stereotype Clockis defined to be applicable at will (dynamically) to instances of the metaclass Class.



**Figure 12.24 Defining a Stereotype**



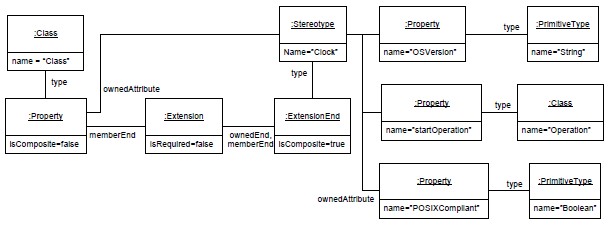
**Figure 12.25 Presentation Options for an Extended Class**

In Figure 12.26, an instance diagram of the example in Figure 12.24 is shown.

**NOTE.** The ExtensionEnd must be composite, and that the derived isRequired Property in this case is **false**.

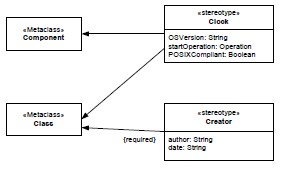
Figure 12.26 shows the instances representing the definition of the Stereotype named Clock defined in Figure 12.24. In this definition, the extended metaclass (:Class; “name = Class”) is defined in the UML2 metamodel (reference metamodel). In a UML modeling tool this representation of the UML2 standard metamodel would typically be in a “read only” form, or presented as proxies to the metaclass being extended.

(It is therefore still at the same meta-level as UML, and does not show the instance model of a model extended by the stereotype. An example of this is provided in Figure 12.28 and Figure 12.29.) The Semantics sub clause of the Extension concept explains the MOF equivalent, and how constraints can be attached to stereotypes.



**Figure 12.26 An Instance Diagram when Defining a Stereotype**

Figure 12.27 shows how the same Stereotype named Clockextends both the metaclass Component and the metaclass Class (though each instance of the Stereotype can extend only one model element). It also shows how different Stereotypes can extend the same metaclass.



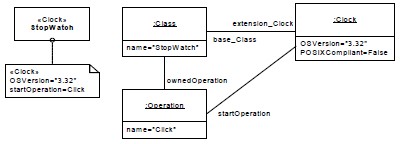
**Figure 12.27 Defining Multiple Stereotypes on Multiple Stereotypes**

Figure 12.28 shows how the Stereotype Clock, as defined in Figure 12.27, is applied to a Class named StopWatch.



**Figure 12.28 Using a Stereotype**

Figure 12.29 shows the underlying semantics for when the Stereotype named Clockis applied to a class called StopWatch. The right-hand side uses instance diagram notation to show the MOF-equivalent instances that should be used to understand the behavior and XMI serialization of the UML diagram on the left. The Extension between the Stereotype and the metaclass Class results in a link between the instance of Stereotype Clockand the (user-defined) Class named StopWatch.



**Figure 12.29 Showing Values of Stereotypes and a Simple Instance Specification**

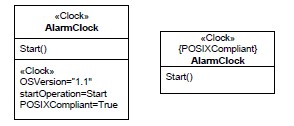
Next, two stereotypes, Clockand Creator, are applied to the same model element, as shown in Figure 12.30.

**NOTE.** The Property values of each of the applied Stereotypes are shown in a comment symbol attached to the model element.



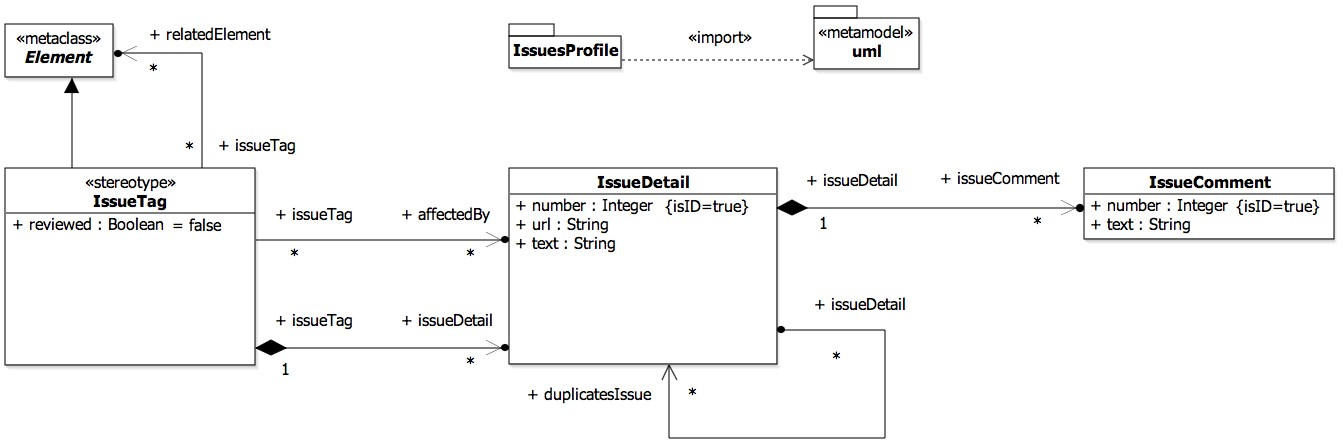
**Figure 12.30 Using Stereotypes and Showing Values**

Finally, two more alternative notational forms are shown in Figure 12.31.



**Figure 12.31 Other Notational Forms for Depicting Stereotype Values**

Figure 12.32 shows an example of a profile with profile-defined classes and binary composite and noncomposite associations.



**Figure 12.32 Example of a Profile defining Classes and binary composite and non-composite Associations**

The following shows the XMI serialization of the profile shown in Figure 12.32:

<?xml version="1.0" encoding="UTF-8"?>

<xmi:XMI xmlns:xmi="http://www.omg.org/spec/XMI/YYYYMMnn" xmlns:uml="http://www.omg.org/spec/UML/YYYYMMnn" xmlns:mofext="http://www.omg.org/spec/MOF/YYYYMMnn">

<uml:Profile xmi:type="uml:Profile"

URI="http://www.example.org/IssuesProfile"

xmi:id="id0" name="IssuesProfile" metamodelReference="id309">

<packageImport xmi:id="id309">

<importedPackage xmi:type="uml:Model"

href="http://www.omg.org/spec/UML/YYYYMMnn/UML.xmi#\_0"/>

</packageImport>

<packagedElement xmi:type="uml:Class" xmi:id="id312" name="IssueDetail">

<ownedAttribute xmi:id="id315" name="number">

<type xmi:type="uml:PrimitiveType"

href="http://www.omg.org/spec/UML/YYYYMMnn/PrimitiveTypes.xmi#Integer"/>

</ownedAttribute>

<ownedAttribute xmi:id="id318" name="url">

<type xmi:type="uml:PrimitiveType"

href="http://www.omg.org/spec/UML/YYYYMMnn/PrimitiveTypes.xmi#String"/>

</ownedAttribute>

<ownedAttribute xmi:id="id321" name="text">

<type xmi:type="uml:PrimitiveType"

href="http://www.omg.org/spec/UML/YYYYMMnn/PrimitiveTypes.xmi#String"/>

</ownedAttribute>

<ownedAttribute xmi:id="id324" name="issueComment" type="id343" aggregation="composite" association="id352">

<upperValue xmi:type="uml:LiteralUnlimitedNatural"

xmi:id="id328" value="\*"/> <lowerValue value="0"

xmi:type="uml:LiteralInteger"

xmi:id="id331"/>

</ownedAttribute>

<ownedAttribute xmi:id="id334" name="duplicatesIssue" type="id312" association="id364">

<upperValue xmi:type="uml:LiteralUnlimitedNatural"

xmi:id="id337" value="\*"/> <lowerValue value="0"

xmi:type="uml:LiteralInteger"

xmi:id="id340"/>

</ownedAttribute>

</packagedElement>

<packagedElement xmi:type="uml:Class" xmi:id="id343" name="IssueComment">

<ownedAttribute xmi:id="id346" name="number">

<type xmi:type="uml:PrimitiveType"

href="http://www.omg.org/spec/UML/YYYYMMnn/PrimitiveTypes.xmi#Integer"/>

</ownedAttribute>

<ownedAttribute xmi:id="id349" name="text">

<type xmi:type="uml:PrimitiveType"

href="http://www.omg.org/spec/UML/YYYYMMnn/PrimitiveTypes.xmi#String"/>

</ownedAttribute>

</packagedElement>

<packagedElement xmi:type="uml:Association"

xmi:id="id352"

name="A\_issueDetail\_issueComment" memberEnd="id324 id355">

<ownedEnd xmi:id="id355" name="issueDetail" type="id312" association="id352"/>

</packagedElement>

<packagedElement xmi:type="uml:Association"

xmi:id="id364"

name="A\_issueDetail\_duplicatesIssue"

memberEnd="id334 id367">

<ownedEnd xmi:id="id367" name="issueDetail" type="id312" association="id364">

<upperValue xmi:type="uml:LiteralUnlimitedNatural"

xmi:id="id371" value="\*"/> <lowerValue value="0"

xmi:type="uml:LiteralInteger"

xmi:id="id374"/>

</ownedEnd>

</packagedElement>

<packagedElement xmi:type="uml:Stereotype" xmi:id="id377" name="IssueTag"> <ownedAttribute xmi:id="id380" name="base\_Element" association="id418">

<type xmi:type="uml:Class"

href="http://www.omg.org/spec/UML/YYYYMMnn/UML.xmi#Element"/>

</ownedAttribute>

<ownedAttribute xmi:id="id383" name="reviewed">

<type xmi:type="uml:PrimitiveType"

href="http://www.omg.org/spec/UML/YYYYMMnn/PrimitiveTypes.xmi#Boolean"/>

<defaultValue xmi:type="uml:LiteralBoolean" xmi:id="id386"/>

</ownedAttribute>

<ownedAttribute xmi:id="id389" name="issueDetail" type="id312" aggregation="composite" association="id424">

<upperValue xmi:type="uml:LiteralUnlimitedNatural"

xmi:id="id393" value="\*"/> <lowerValue value="0"

xmi:type="uml:LiteralInteger"

xmi:id="id396"/>

</ownedAttribute>

<ownedAttribute xmi:id="id399" name="affectedBy" type="id312" association="id436">

<upperValue xmi:type="uml:LiteralUnlimitedNatural"

xmi:id="id403"

value="\*"/> <lowerValue value="0"

xmi:type="uml:LiteralInteger"

xmi:id="id406"/>

</ownedAttribute>

<ownedAttribute xmi:id="id409" name="relatedElement" association="id448">

<type xmi:type="uml:Class"

href="http://www.omg.org/spec/UML/YYYYMMnn/UML.xmi#Element"/>

<upperValue xmi:type="uml:LiteralUnlimitedNatural"

xmi:id="id412" value="\*"/> <lowerValue value="0"

xmi:type="uml:LiteralInteger"

xmi:id="id415"/>

</ownedAttribute>

</packagedElement>

<packagedElement xmi:type="uml:Extension" xmi:id="id418" name="Element\_IssueTag" memberEnd="id421 id380"> <ownedEnd xmi:type="uml:ExtensionEnd"

xmi:id="id421" name="extension\_IssueTag"

type="id377" aggregation="composite" association="id418"/>

</packagedElement>

<packagedElement xmi:type="uml:Association" xmi:id="id424" name="A\_issueTag\_issueDetail" memberEnd="id389 id427">

<ownedEnd xmi:id="id427" name="issueTag" type="id377" association="id424"/>

</packagedElement>

<packagedElement xmi:type="uml:Association" xmi:id="id436" name="A\_issueTag\_affectedBy" memberEnd="id399 id439">

<ownedEnd xmi:id="id439" name="issueTag" type="id377" association="id436">

<upperValue xmi:type="uml:LiteralUnlimitedNatural"

xmi:id="id442" value="\*"/> <lowerValue value="0"

xmi:type="uml:LiteralInteger"

xmi:id="id445"/>

</ownedEnd>

</packagedElement>

<packagedElement xmi:type="uml:Association" xmi:id="id448"

name="A\_issueTag\_relatedElement" memberEnd="id409 id451">

<ownedEnd xmi:id="id451" name="issueTag" type="id377" association="id448">

<upperValue xmi:type="uml:LiteralUnlimitedNatural"

xmi:id="id454" value="\*"/> <lowerValue value="0"

xmi:type="uml:LiteralInteger"

xmi:id="id457"/>

</ownedEnd>

</packagedElement>

</uml:Profile>

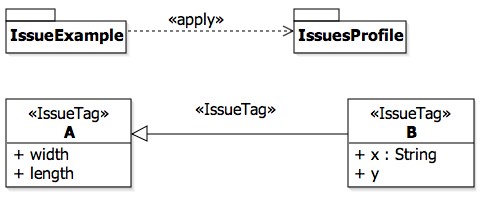
<mofext:Tag xmi:type="mofext:Tag"

org.omg.xmi.nsURI="http://www.example.org/IssuesProfile"/>

<mofext:Tag xmi:type="mofext:Tag"

org.omg.xmi.nsPrefix="IssuesProfile"/> </xmi:XMI>

Figure 12.33 shows an example of applying the profile shown in Figure 12.32.



**Figure 12.33 Diagram example of applying a profile defining Classes and Associations and of creating instances of such Classes. Tools can provide a notation similar to that of object diagrams for instances of Profile-defined Classes, DataTypes and Associations**

The following shows the XMI serialization of the example shown in Figure 12.33 without link instances of profiledefined associations:

<?xml version="1.0" encoding="UTF-8"?>

<xmi:XMI xmlns:xmi="http://www.omg.org/spec/XMI/YYYYMMnn" xmlns:uml="http://www.omg.org/spec/UML/YYYYMMnn"

xmlns:IssuesProfile="http://www.example.org/IssuesProfile">

<uml:Package xmi:id="ex0" name="IssueExample">

<packagedElement xmi:type="uml:Class" xmi:id="ex57" name="B">

<generalization xmi:id="ex60" general="ex65"/>

<ownedAttribute xmi:id="ex63" name="x">

<type xmi:type="uml:PrimitiveType"

href="http://www.omg.org/spec/UML/YYYYMMnn/PrimitiveTypes.xmi#String"/>

</ownedAttribute>

<ownedAttribute xmi:id="ex64" name="y"/>

</packagedElement>

<packagedElement xmi:type="uml:Class" xmi:id="ex65" name="A">

<ownedAttribute xmi:id="ex69" name="width"/>

<ownedAttribute xmi:id="ex70" name="length"/>

</packagedElement>

<profileApplication xmi:id="ex77">

<appliedProfile href="http://www.example.org/IssuesProfile#id0"/>

</profileApplication>

</uml:Package>

<IssuesProfile:IssueTag xmi:id="ex66" base\_Element="ex65" relatedElement="ex69 ex70">

<issueDetail xmi:id="ex67" number="1"

url="http://www.example.org/issues/1" text="Some attributes lack a type.">

<issueComment xmi:id="ex68" number="3"

text="Type should be string."/>

</issueDetail>

</IssuesProfile:IssueTag>

<IssuesProfile:IssueTag xmi:id="ex58" base\_Element="ex57" affectedBy="ex66 ex61" relatedElement="ex64">

<issueDetail xmi:id="ex59" number="2"

url="http://www.example.org/issues/2" text="Some attributes lack a type."

duplicatesIssue="ex67"/>

</IssuesProfile:IssueTag>

<IssuesProfile:IssueTag xmi:id="ex61" base\_Element="ex60">

<issueDetail xmi:id="ex62" number="4"

url="http://www.example.org/issues/4"

text="Why does B specialize A?"/>

</IssuesProfile:IssueTag>

</xmi:XMI>

The following shows the XMI serialization of the example shown in Figure 12.33 with link instances of profile-defined associations serialized:

<?xml version="1.0" encoding="UTF-8"?>

<xmi:XMI xmlns:xmi="http://www.omg.org/spec/XMI/YYYYMMnn" xmlns:uml="http://www.omg.org/spec/UML/YYYYMMnn"

xmlns:IssuesProfile="http://www.example.org/IssuesProfile">

<uml:Package xmi:id="ex0" name="IssueExample">

<packagedElement xmi:type="uml:Class" xmi:id="ex73" name="B">

<generalization xmi:id="ex76" general="ex81"/>

<ownedAttribute xmi:id="ex79" name="x">

<type xmi:type="uml:PrimitiveType"

href="http://www.omg.org/spec/UML/YYYYMMnn/PrimitiveTypes.xmi#String"/>

</ownedAttribute>

<ownedAttribute xmi:id="ex80" name="y"/>

</packagedElement>

<packagedElement xmi:type="uml:Class" xmi:id="ex81" name="A">

<ownedAttribute xmi:id="ex85" name="width"/>

<ownedAttribute xmi:id="ex86" name="length"/>

</packagedElement>

<packagedElement xmi:type="uml:Package" xmi:id="ex87"

name="PrimitiveTypes"/>

<profileApplication xmi:id="ex97">

<appliedProfile href="http://www.example.org/IssuesProfile#id0"/>

</profileApplication>

</uml:Package>

<IssuesProfile:IssueTag xmi:id="ex82" base\_Element="ex81" relatedElement="ex85 ex86">

<issueDetail xmi:id="ex83" number="1"

url="http://www.example.org/issues/1" text="Some attributes lack a type.">

<issueComment xmi:id="ex84" number="3"

text="Type should be string."/>

</issueDetail>

</IssuesProfile:IssueTag>

<IssuesProfile:A\_issueTag\_issueDetail xmi:id="d1e109" issueTag="ex82" issueDetail="ex83"/>

<IssuesProfile:A\_issueDetail\_issueComment xmi:id="d1e110" issueDetail="ex83" issueComment="ex84"/>

<IssuesProfile:A\_issueTag\_relatedElement xmi:id="ex82ex85" issueTag="ex82" relatedElement="ex85"/>

<IssuesProfile:A\_issueTag\_relatedElement xmi:id="ex82ex86" issueTag="ex82" relatedElement="ex86"/>

<IssuesProfile:IssueTag xmi:id="ex74" base\_Element="ex73" affectedBy="ex82 ex77" relatedElement="ex80">

<issueDetail xmi:id="ex75" number="2"

url="http://www.example.org/issues/2" text="Some attributes lack a type."

duplicatesIssue="ex83"/>

</IssuesProfile:IssueTag>

<IssuesProfile:A\_issueTag\_issueDetail xmi:id="d1e112" issueTag="ex74" issueDetail="ex75"/>

<IssuesProfile:A\_issueDetail\_duplicatesIssue xmi:id="ex75ex83" issueDetail="ex75" duplicatesIssue="ex83"/>

<IssuesProfile:A\_issueTag\_affectedBy xmi:id="ex74ex82" issueTag="ex74" affectedBy="ex82"/>

<IssuesProfile:A\_issueTag\_affectedBy xmi:id="ex74ex77" issueTag="ex74" affectedBy="ex77"/>

<IssuesProfile:A\_issueTag\_relatedElement xmi:id="ex74ex80" issueTag="ex74" relatedElement="ex80"/>

<IssuesProfile:IssueTag xmi:id="ex77" base\_Element="ex76">

<issueDetail xmi:id="ex78" number="4"

url="http://www.example.org/issues/4" text="Why does B specialize A?"/>

</IssuesProfile:IssueTag>

<IssuesProfile:A\_issueTag\_issueDetail xmi:id="d1e114" issueTag="ex77" issueDetail="ex78"/>

</xmi:XMI>

### 12.4 Classifier Descriptions

**12.4.1 Extension [Class]**

**12.4.1.1 Description**

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

**12.4.1.2 Diagrams**

Profiles, Classes

**12.4.1.3 Generalizations**

Association

**12.4.1.4 Attributes**

* /isRequired : Boolean [1..1]

Indicates whether an instance of the extending stereotype must be created when an instance of the extended class is created. The attribute value is derived from the value of the lower property of the ExtensionEnd referenced by Extension::ownedEnd; a lower value of 1 means that isRequired is true, but otherwise it is false. Since the default value of ExtensionEnd::lower is 0, the default value of isRequired is false.

**12.4.1.5 Association Ends**

* /metaclass : Class [1..1]{} (opposite Class::extension)

References the Class that is extended through an Extension. The property is derived from the type of the memberEnd that is not the ownedEnd.

* ♦ ownedEnd : ExtensionEnd [1..1]{redefines Association::ownedEnd} (opposite

A\_ownedEnd\_extension::extension)

References the end of the extension that is typed by a Stereotype.

**12.4.1.6 Operations**

* isRequired() : Boolean

The query isRequired() is true if the owned end has a multiplicity with the lower bound of 1. body: ownedEnd.lowerBound() = 1

* metaclass() : Class

The query metaclass() returns the metaclass that is being extended (as opposed to the extending stereotype). body: metaclassEnd().type.oclAsType(Class)

* metaclassEnd() : Property

The query metaclassEnd() returns the Property that is typed by a metaclass (as opposed to a stereotype). body: memberEnd->reject(p | ownedEnd->includes(p.oclAsType(ExtensionEnd)))->any(true)

**12.4.1.7 Constraints**

* non\_owned\_end

The non-owned end of an Extension is typed by a Class. inv: metaclassEnd()->notEmpty() and metaclassEnd().type.oclIsKindOf(Class)

* is\_binary

An Extension is binary, i.e., it has only two memberEnds. inv: memberEnd->size() = 2

**12.4.2 ExtensionEnd [Class]**

**12.4.2.1 Description**

An extension end is used to tie an extension to a stereotype when extending a metaclass. The default multiplicity of an extension end is 0..1.

**12.4.2.2 Diagrams**

Profiles

**12.4.2.3 Generalizations**

Property

**12.4.2.4 Attributes**

* /lower : Integer [0..1]

This redefinition changes the default multiplicity of association ends, since model elements are usually extended by 0 or 1 instance of the extension stereotype.

**12.4.2.5 Association Ends**

* type : Stereotype [1..1]{redefines TypedElement::type} (opposite A\_type\_extensionEnd::extensionEnd)

References the type of the ExtensionEnd. Note that this association restricts the possible types of an ExtensionEnd to only be Stereotypes.

**12.4.2.6 Operations**

* lowerBound() : Integer [0..1] {redefines MultiplicityElement::lowerBound()}

The query lowerBound() returns the lower bound of the multiplicity as an Integer. This is a redefinition of the default lower bound, which normally, for MultiplicityElements, evaluates to 1 if empty. body: if lowerValue=null then 0 else lowerValue.integerValue() endif

**12.4.2.7 Constraints**

* multiplicity

The multiplicity of ExtensionEnd is 0..1 or 1.

inv: (lowerBound() = 0 or lowerBound() = 1) and upperBound() = 1

* aggregation

The aggregation of an ExtensionEnd is composite. inv: self.aggregation = AggregationKind::composite

**12.4.3 Image [Class]**

**12.4.3.1 Description**

Physical definition of a graphical image.

**12.4.3.2 Diagrams**

Profiles

**12.4.3.3 Generalizations**

Element

**12.4.3.4 Attributes**

* content : String [0..1]

This contains the serialization of the image according to the format. The value could represent a bitmap, image such as a GIF file, or drawing 'instructions' using a standard such as Scalable Vector Graphic (SVG) (which is XML based).

* format : String [0..1]

This indicates the format of the content, which is how the string content should be interpreted. The following values are reserved: SVG, GIF, PNG, JPG, WMF, EMF, BMP. In addition the prefix 'MIME: ' is also reserved. This option can be used as an alternative to express the reserved values above, for example "SVG" could instead be expressed as "MIME: image/svg+xml".

* location : String [0..1]

This contains a location that can be used by a tool to locate the image as an alternative to embedding it in the stereotype.

**12.4.4 Model [Class]**

**12.4.4.1 Description**

A model captures a view of a physical system. It is an abstraction of the physical system, with a certain purpose. This purpose determines what is to be included in the model and what is irrelevant. Thus the model completely describes those aspects of the physical system that are relevant to the purpose of the model, at the appropriate level of detail.

**12.4.4.2 Diagrams**

Packages

**12.4.4.3 Generalizations**

Package

**12.4.4.4 Attributes**

 viewpoint : String [0..1]

The name of the viewpoint that is expressed by a model (this name may refer to a profile definition).

**12.4.5 Package [Class]**

**12.4.5.1 Description**

A package can have one or more profile applications to indicate which profiles have been applied. Because a profile is a package, it is possible to apply a profile not only to packages, but also to profiles. Package specializes

TemplateableElement and PackageableElement specializes ParameterableElement to specify that a package can be used as a template and a PackageableElement as a template parameter. A package is used to group elements, and provides a namespace for the grouped elements.

**12.4.5.2 Diagrams**

Packages, Profiles, Namespaces

**12.4.5.3 Generalizations**

PackageableElement, TemplateableElement, Namespace

**12.4.5.4 Specializations**

Model, Profile

**12.4.5.5 Attributes**

* URI : String [0..1]

Provides an identifier for the package that can be used for many purposes. A URI is the universally unique identification of the package following the IETF URI specification, RFC 2396 http://www.ietf.org/rfc/rfc2396.txt and it must comply with those syntax rules.

**12.4.5.6 Association Ends**

* ♦ /nestedPackage : Package [0..\*]{subsets Package::packagedElement} (opposite Package::nestingPackage) References the packaged elements that are Packages.
* nestingPackage : Package [0..1]{subsets A\_packagedElement\_owningPackage::owningPackage} (opposite

Package::nestedPackage)

References the Package that owns this Package.

* ♦ /ownedStereotype : Stereotype [0..\*]{subsets Package::packagedElement} (opposite

A\_ownedStereotype\_owningPackage::owningPackage)

References the Stereotypes that are owned by the Package.

* ♦ /ownedType : Type [0..\*]{subsets Package::packagedElement} (opposite Type::package) References the packaged elements that are Types.
* ♦ packageMerge : PackageMerge [0..\*]{subsets Element::ownedElement, subsets

A\_source\_directedRelationship::directedRelationship} (opposite PackageMerge::receivingPackage) References the PackageMerges that are owned by this Package.

* ♦ packagedElement : PackageableElement [0..\*]{subsets Namespace::ownedMember} (opposite

A\_packagedElement\_owningPackage::owningPackage)

Specifies the packageable elements that are owned by this Package.

* ♦ profileApplication : ProfileApplication [0..\*]{subsets Element::ownedElement, subsets

A\_source\_directedRelationship::directedRelationship} (opposite ProfileApplication::applyingPackage) References the ProfileApplications that indicate which profiles have been applied to the Package.

**12.4.5.7 Operations**

* allApplicableStereotypes() : Stereotype [0..\*]

The query allApplicableStereotypes() returns all the directly or indirectly owned stereotypes, including stereotypes contained in sub-profiles.

body: let ownedPackages : Bag(Package) = ownedMember->select(oclIsKindOf(Package))-

>collect(oclAsType(Package)) in

ownedStereotype->union(ownedPackages.allApplicableStereotypes())->flatten()->asSet()

* containingProfile() : Profile [0..1]

The query containingProfile() returns the closest profile directly or indirectly containing this package (or this package itself, if it is a profile).

body: if self.oclIsKindOf(Profile) then

self.oclAsType(Profile) else self.namespace.oclAsType(Package).containingProfile()

endif

* makesVisible(el : NamedElement) : Boolean

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

pre: member->includes(el) body: ownedMember->includes(el) or

(elementImport->select(ei|ei.importedElement = VisibilityKind::public)-

>collect(importedElement.oclAsType(NamedElement))->includes(el)) or

(packageImport->select(visibility = VisibilityKind::public)->collect(importedPackage.member-

>includes(el))->notEmpty())

* mustBeOwned() : Boolean {redefines Element::mustBeOwned()}

The query mustBeOwned() indicates whether elements of this type must have an owner. body: false

* nestedPackage() : Package [0..\*]

Derivation for Package::/nestedPackage

body: packagedElement->select(oclIsKindOf(Package))->collect(oclAsType(Package))->asSet()

* ownedStereotype() : Stereotype [0..\*]

Derivation for Package::/ownedStereotype

body: packagedElement->select(oclIsKindOf(Stereotype))->collect(oclAsType(Stereotype))>asSet()

* ownedType() : Type [0..\*]

Derivation for Package::/ownedType

body: packagedElement->select(oclIsKindOf(Type))->collect(oclAsType(Type))->asSet()

* visibleMembers() : PackageableElement [0..\*]

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

body: member->select( m | m.oclIsKindOf(PackageableElement) and self.makesVisible(m))>collect(oclAsType(PackageableElement))->asSet()

**12.4.5.8 Constraints**

* elements\_public\_or\_private

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

inv: packagedElement->forAll(e | e.visibility<> null implies e.visibility = VisibilityKind::public or e.visibility = VisibilityKind::private)

**12.4.6 PackageMerge [Class]**

**12.4.6.1 Description**

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

**12.4.6.2 Diagrams**

Packages

**12.4.6.3 Generalizations**

DirectedRelationship

**12.4.6.4 Association Ends**

* mergedPackage : Package [1..1]{subsets DirectedRelationship::target} (opposite

A\_mergedPackage\_packageMerge::packageMerge)

References the Package that is to be merged with the receiving package of the PackageMerge.

* receivingPackage : Package [1..1]{subsets DirectedRelationship::source, subsets Element::owner} (opposite

Package::packageMerge)

References the Package that is being extended with the contents of the merged package of the PackageMerge.

**12.4.7 Profile [Class]**

**12.4.7.1 Description**

A profile defines limited extensions to a reference metamodel with the purpose of adapting the metamodel to a specific platform or domain.

**12.4.7.2 Diagrams**

Profiles

**12.4.7.3 Generalizations**

Package

**12.4.7.4 Association Ends**

* ♦ metaclassReference : ElementImport [0..\*]{subsets Namespace::elementImport} (opposite

A\_metaclassReference\_profile::profile)

References a metaclass that may be extended.

* ♦ metamodelReference : PackageImport [0..\*]{subsets Namespace::packageImport} (opposite

A\_metamodelReference\_profile::profile)

References a package containing (directly or indirectly) metaclasses that may be extended.

**12.4.7.5 Constraints**

* metaclass\_reference\_not\_specialized

An element imported as a metaclassReference is not specialized or generalized in a Profile.

inv: metaclassReference.importedElement->

select(c | c.oclIsKindOf(Classifier) and

(c.oclAsType(Classifier).allParents()->collect(namespace)->includes(self)))-

>isEmpty() and

packagedElement->

select(oclIsKindOf(Classifier))->collect(oclAsType(Classifier).allParents())-> intersection(metaclassReference.importedElement->select(oclIsKindOf(Classifier))>collect(oclAsType(Classifier)))->isEmpty()

* references\_same\_metamodel

All elements imported either as metaclassReferences or through metamodelReferences are members of the same base reference metamodel.

inv: metamodelReference.importedPackage.elementImport.importedElement.allOwningPackages()-> union(metaclassReference.importedElement.allOwningPackages() )->notEmpty()

**12.4.8 ProfileApplication [Class]**

**12.4.8.1 Description**

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

**12.4.8.2 Diagrams**

Profiles

**12.4.8.3 Generalizations**

DirectedRelationship

**12.4.8.4 Attributes**

* isStrict : Boolean [1..1] = false

Specifies that the Profile filtering rules for the metaclasses of the referenced metamodel shall be strictly applied.

**12.4.8.5 Association Ends**

* appliedProfile : Profile [1..1]{subsets DirectedRelationship::target} (opposite

A\_appliedProfile\_profileApplication::profileApplication)

References the Profiles that are applied to a Package through this ProfileApplication.

* applyingPackage : Package [1..1]{subsets DirectedRelationship::source, subsets Element::owner} (opposite

Package::profileApplication)

The package that owns the profile application.

**12.4.9 Stereotype [Class]**

**12.4.9.1 Description**

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

**12.4.9.2 Diagrams**

Profiles

**12.4.9.3 Generalizations**

Class

**12.4.9.4 Association Ends**

* ♦ icon : Image [0..\*]{subsets Element::ownedElement} (opposite A\_icon\_stereotype::stereotype)

Stereotype can change the graphical appearance of the extended model element by using attached icons. When this association is not null, it references the location of the icon content to be displayed within diagrams presenting the extended model elements.

* /profile : Profile [1..1]{} (opposite A\_profile\_stereotype::stereotype) The profile that directly or indirectly contains this stereotype.

**12.4.9.5 Operations**

* containingProfile() : Profile

The query containingProfile returns the closest profile directly or indirectly containing this stereotype. body: self.namespace.oclAsType(Package).containingProfile()

* profile() : Profile
  1. stereotype must be contained, directly or indirectly, in a profile.body: self.containingProfile()

**12.4.9.6 Constraints**

* binaryAssociationsOnly

Stereotypes may only participate in binary associations. inv: ownedAttribute.association->forAll(memberEnd->size()=2)

* generalize
  1. Stereotype may only generalize or specialize another Stereotype.

inv: allParents()->forAll(oclIsKindOf(Stereotype))

and Classifier.allInstances()->forAll(c | c.allParents()->exists(oclIsKindOf(Stereotype)) implies c.oclIsKindOf(Stereotype))

* name\_not\_clash

Stereotype names should not clash with keyword names for the extended model element.

Cannot be expressed in OCL

* associationEndOwnership

Where a stereotype’s property is an association end for an association other than a kind of extension, and the other end is not a stereotype, the other end must be owned by the association itself.

inv: ownedAttribute

->select(association->notEmpty() and not association.oclIsKindOf(Extension) and not

type.oclIsKindOf(Stereotype)) ->forAll(opposite.owner = association)

* base\_property\_upper\_bound

The upper bound of base-properties is exactly 1.

Cannot be expressed in OCL

* base\_property\_multiplicity\_single\_extension

If a Stereotype extends only one metaclass, the multiplicity of the corresponding base-property shall be 1..1.

Cannot be expressed in OCL

* base\_property\_multiplicity\_multiple\_extension

If a Stereotype extends more than one metaclass, the multiplicity of the corresponding base-properties shall be [0..1]. At any point in time, only one of these base-properties can contain a metaclass instance during runtime.

Cannot be expressed in OCL

### 12.5 Association Descriptions

**12.5.1 A\_appliedProfile\_profileApplication [Association]**

**12.5.1.1 Diagrams**

Profiles

**12.5.1.2 Owned Ends**

* profileApplication : ProfileApplication [0..\*]{subsets A\_target\_directedRelationship::directedRelationship} (opposite ProfileApplication::appliedProfile)

**12.5.2 A\_icon\_stereotype [Association]**

**12.5.2.1 Diagrams**

Profiles

**12.5.2.2 Owned Ends**

* stereotype : Stereotype [0..1]{subsets Element::owner} (opposite Stereotype::icon)

**12.5.3 A\_mergedPackage\_packageMerge [Association]**

**12.5.3.1 Diagrams**

Packages

**12.5.3.2 Owned Ends**

* packageMerge : PackageMerge [0..\*]{subsets A\_target\_directedRelationship::directedRelationship} (opposite PackageMerge::mergedPackage)

**12.5.4 A\_metaclassReference\_profile [Association]**

**12.5.4.1 Diagrams**

Profiles

**12.5.4.2 Owned Ends**

* profile : Profile [0..1]{subsets ElementImport::importingNamespace} (opposite Profile::metaclassReference)

**12.5.5 A\_metamodelReference\_profile [Association]**

**12.5.5.1 Diagrams**

Profiles

**12.5.5.2 Owned Ends**

* profile : Profile [0..1]{subsets PackageImport::importingNamespace} (opposite Profile::metamodelReference)

**12.5.6 A\_nestedPackage\_nestingPackage [Association]**

**12.5.6.1 Diagrams**

Packages

**12.5.6.2 Member Ends**

* Package::nestedPackage
* Package::nestingPackage

**12.5.7 A\_ownedEnd\_extension [Association]**

**12.5.7.1 Diagrams**

Profiles

**12.5.7.2 Owned Ends**

* extension : Extension [1..1]{subsets Property::owningAssociation} (opposite Extension::ownedEnd)

**12.5.8 A\_ownedStereotype\_owningPackage [Association]**

**12.5.8.1 Diagrams**

Profiles

**12.5.8.2 Generalizations**

A\_packagedElement\_owningPackage

**12.5.8.3 Owned Ends**

* owningPackage : Package [1..1]{redefines A\_packagedElement\_owningPackage::owningPackage} (opposite Package::ownedStereotype)

**12.5.9 A\_ownedType\_package [Association]**

**12.5.9.1 Diagrams**

Packages

**12.5.9.2 Member Ends**

* Package::ownedType
* Type::package

**12.5.10 A\_packageMerge\_receivingPackage [Association]**

**12.5.10.1 Diagrams**

Packages

**12.5.10.2 Member Ends**

* Package::packageMerge
* PackageMerge::receivingPackage

**12.5.11 A\_packagedElement\_owningPackage [Association]**

**12.5.11.1 Diagrams**

Packages

**12.5.11.2 Specializations**

A\_ownedStereotype\_owningPackage

**12.5.11.3 Owned Ends**

* owningPackage : Package [0..1]{subsets NamedElement::namespace} (opposite Package::packagedElement)

**12.5.12 A\_profileApplication\_applyingPackage [Association]**

**12.5.12.1 Diagrams**

Profiles

**12.5.12.2 Member Ends**

* Package::profileApplication
* ProfileApplication::applyingPackage

**12.5.13 A\_profile\_stereotype [Association]**

**12.5.13.1 Diagrams**

Profiles

**12.5.13.2 Owned Ends**

* stereotype : Stereotype [0..\*] (opposite Stereotype::profile)

**12.5.14 A\_type\_extensionEnd [Association]**

**12.5.14.1 Diagrams**

Profiles

**12.5.14.2 Owned Ends**

* extensionEnd : ExtensionEnd [0..\*]{subsets A\_type\_typedElement::typedElement} (opposite ExtensionEnd::type)

## 13 Common Behavior

### 13.1 Summary

This clause specifies the core concepts underlying all behavioral modeling in UML. Structural models of Classifiers in UML define the allowable instances that may exist at any point in time, what values their StructuralFeatures may have and how those instances may be related to each other. Behavioral modeling, on the other hand, models how these instances may change over time.

UML provides Behavior, Event, and Trigger constructs to model the corresponding fundamental concepts of behavioral modeling.

*Behavior* is the basic concept for modeling dynamic change. Behavior may be *executed,* either by direct invocation or through the creation of an *active object* that hosts the behavior. Behavior may also be *emergent,* resulting from the interaction of one or more participant objects that are themselves carrying out their own individual behaviors.

Dynamic behavior results in *events* of interest that occur at specific points in time. Such events may be implicit, occurring on the change of some value or the passage of some interval of time. They may also be explicit, occurring when an operation is called or an asynchronous *signal* is received.

The occurrence of an event may then *trigger* new behavior, or change the course of already executing behavior. Explicit events thus provide the basic mechanism for communication between behaviors, in which an action carried out in one behavior, such as calling an operation or sending a signal, can trigger a response in another behavior.

The remainder of this clause further details the fundamental UML modeling mechanisms of Behaviors, Events and Triggers. These mechanisms then provide the framework for the specification in the following clauses of various complete UML behavioral modeling constructs.

### 13.2 Behaviors

**13.2.1 Summary**

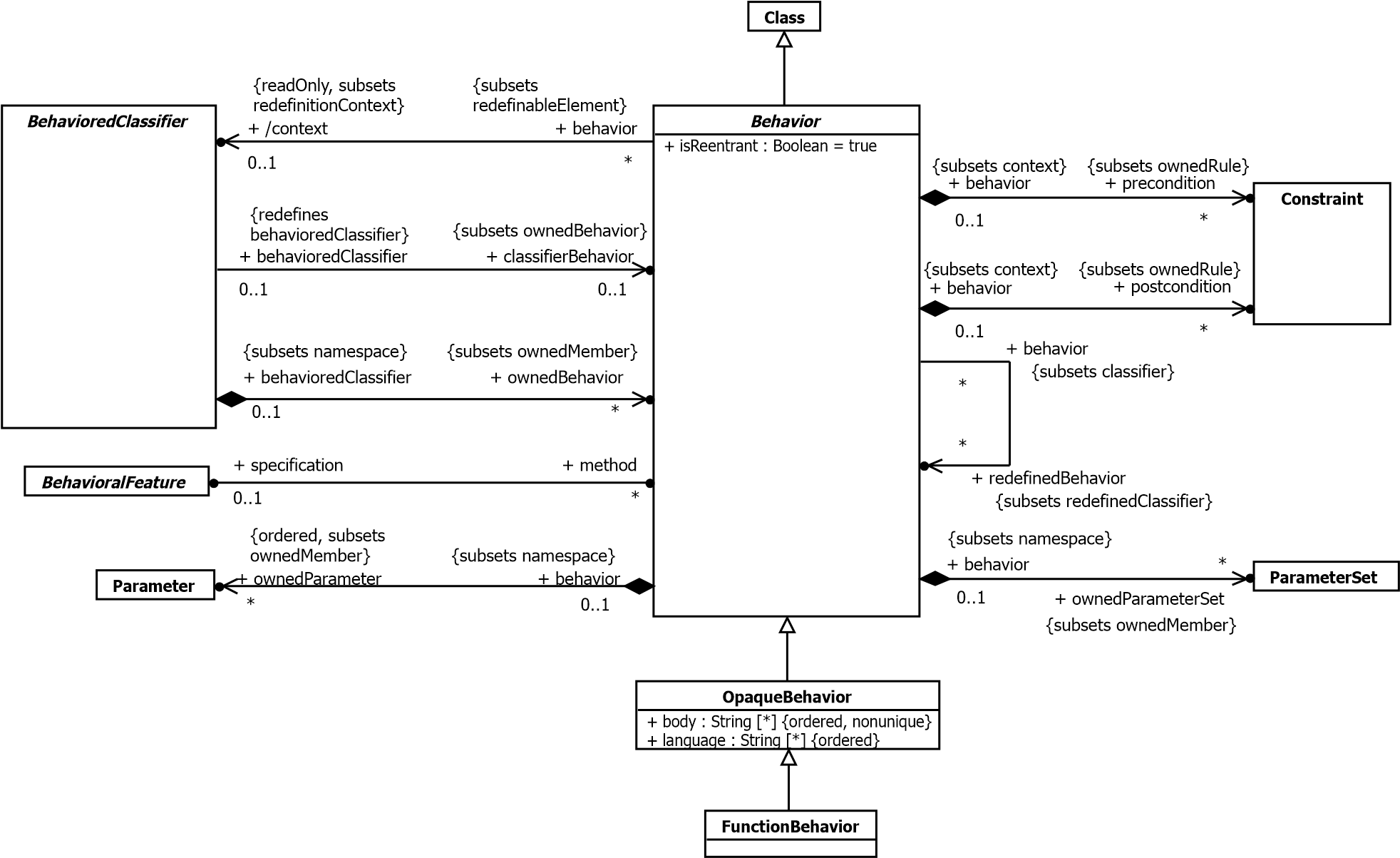
This sub clause introduces the framework for modeling behavior in UML. The concrete subtypes of Behavior, described in subsequent clauses, then provide different mechanisms to specify behaviors.

A variety of behavioral specification mechanisms are supported by UML, including:

* StateMachines that model finite automata (see Clause 14)
* Activities defined using Petri-net-like graphs (see Clause 15)
* Interactions that model partially-ordered sequences of event occurrences (see Clause 17).

These behavioral specification mechanisms differ in their expressive power and domain of applicability. This means that not all behaviors can be described by each of the mechanisms. Nevertheless, many behaviors can be described by one or more of the mechanisms, in which case the choice of mechanism is one of convenience, or, alternatively, multiple mechanisms can be used to provide different models of the same behavior.

**13.2.2 Abstract Syntax**



**Figure 13.1 Behaviors**

**13.2.3 Semantics**

**13.2.3.1 Behaviors**

A Behavior is a specification of events that may occur dynamically over time (see also sub clause 13.3 on the explicit modeling of Events in UML). This specification may be prescriptive of specifically what events may occur in what situations, descriptive of emergent behavior or illustrative of possible sequences of event occurrences. Every Behavior defines at least one event, the event of its invocation. A Behavior may be invoked directly, via a BehavioralFeature that it implements as a method or as the classifierBehavior of a BehavioredClassifier.

On each invocation, the subsequent actual sequence of event occurrences due to the invocation, consistent with the specification of the Behavior, is called an *execution trace* for the Behavior. An execution trace always begins with the invocation of the Behavior and may continue indefinitely (if the Behavior does not terminate), or it may end in the occurrence of a *termination* event for the Behavior, in which case the execution of the Behavior is said to have *completed*. A Behavior may either complete *normally*, or it may complete as a result of the raising of an exception, in which case, if the Behavior was invoked synchronously (see below), the exception is propagated to the caller (see also the discussion of exceptions in sub clause 15.5.3). Event occurrences during an execution trace include both occurrences caused by the Behavior, such as attribute value changes, creation and destruction of objects and invocation of other Behaviors, and occurrences that trigger responses within the Behavior, such as the changing of a monitored value or the receipt of a Signal instance.

Behaviors in UML are kinds of Classes, which means that they may be instantiated as objects. An object that is an instance of a Behavior is known as a behavior *execution.* Invoking the Behavior corresponds to instantiating the Behavior, and there is a specific execution trace corresponding to each Behavior execution.

Since a Behavior is a Class, it may be specialized and may also itself own StructuralFeatures and BehavioralFeatures. These features may be referenced in the specification of the Behavior. An execution of the Behavior may then access these features, such as reading and modifying attributes of the Behavior. Public features of a Behavior may also be referenced from outside of the Behavior, as usual for the features of any Class.

A Behavior may be invoked many times. A *reentrant* Behavior (i.e., one with its isReentrant property equal to true*)* may be invoked again before a previous invocation has completed (this is the default). On the other hand, a *non-reentrant* Behavior (i.e, one with its isReentrant property equal to false) shall not be invoked again if a previous invocation has not completed. A reentrant Behavior may have many ongoing executions at any one time, but a non-reentrant Behavior shall have at most one uncompleted execution at any time. If an invoking Behavior attempts to invoke a non-reentrant Behavior that already has an uncompleted execution, then the invoker shall block until the existing execution completes (or indefinitely, if the execution never completes).

A Behavior may be invoked *synchronously* or *asynchronously.* Synchronous invocation means that an invoking Behavior retains a reference to the invoked Behavior execution and waits for the execution to complete. Asynchronous invocation, on the other hand, means that the invoked Behavior execution proceeds concurrently with the invoking Behavior.

The preconditions for a Behavior define conditions that shall be true when the Behavior is invoked. These preconditions may be assumed in the detailed specification of the Behavior. The semantics of an invocation of a Behavior when a precondition is not satisfied are intentionally undefined.

The postconditions for a Behavior define conditions that will be true when the invocation of the Behavior completes successfully, assuming the preconditions were satisfied. These postconditions shall be satisfied in the detailed specification of the Behavior.

**13.2.3.2 Behavior Parameters**

A Behavior may have Parameters (see sub clause 9.4) that provide the ability to pass values into and out of Behavior executions.

When a Behavior is invoked, *argument* values may be provided corresponding to Parameters with direction “in” or “inout”, as constrained by the multiplicity of those Parameters. If such an input Parameter has a defaultValue, and no explicit argument value is given for it, then the defaultValue is evaluated to provide argument values for the Parameter (even if the Parameter has a multiplicity lower bound of 0, so having no value would be valid for it). Argument values are available to affect the course of the invoked Behavior execution.

When a Behavior execution completes, it may produce *result* values corresponding to Parameters with direction “inout,”

“out,” and “return,” as constrained by the multiplicity of those Parameters. If such an output Parameter has a defaultValue, and no explicit result value is given for it, then the defaultValue is evaluated to provide result values for the Parameter (even if the Parameter has a multiplicity lower bound of 0, so having no value would be valid for it). If the Behavior was invoked synchronously, then result values are returned to the invoker. However, if the Behavior was invoked asynchronously, then any result values are lost when the Behavior execution completes.

Parameters may also be marked as *streaming* (i.e., have the isStreaming property be *true*). Such Parameters allow values to be passed into and out of a Behavior execution any time during its course, rather than just on invocation and completion.

If an input Parameter is streaming*,* then argument values may be provided for the Parameter during the course of a

Behavior execution rather than just at invocation. One or more argument values may be *posted* to a streaming input Parameter at or any time after the invocation of a Behavior and before its completion. These argument values are then available to affect the further course of the Behavior execution from that time forward.

If an output Parameter is streaming, then a Behavior execution may provide result values for the Parameter during its course rather than just at completion. One or more result values may be *posted* to a streaming output Parameter any time after the invocation of a Behavior up to or at its completion. These result values are then available to affect the further course of the execution of the *invoking* Behavior from that time forward.

**NOTE.** In order for an invoker to be able to obtain posted values from streaming output Parameters, the invoked Behavior has to be invoked synchronously, even though streamed outputs could potentially trigger asynchronous responses in the invoker. (See sub clause 16.3.3 on the semantics of CallActions in the case of streaming Parameters, including the effect of the multiplicity of such Parameters.)

A reentrant Behavior shall not have streaming Parameters, because there are potentially multiple executions of the Behavior going at the same time, and it would be ambiguous which execution would be receiving or producing streamed values.

A Behavior may have one or more output Parameters marked as isException=true. In this case, when an execution of the Behavior completes, either none of these Parameters shall have values or exactly one shall have a value and no other parameters (exception or otherwise) shall have any values.

**NOTE.** Returning a value in an exception Parameter is *not* considered to be “raising an exception” in the sense described in sub clause 15.5.3.

A Behavior with input ParameterSets can only accept inputs from Parameters in one of the sets per execution. A Behavior with output ParameterSets can only post outputs to the Parameters in one of the sets per execution. The semantics of conditions of input and output ParameterSets are the same as Behavior preconditions and postconditions, respectively, but apply only to the set of Parameters specified.

**13.2.3.3 Opaque and Function Behaviors**

An OpaqueBehavior is a Behavior whose specification is given in a textual language other than UML.

An OpaqueBehavior has a body that consists of a sequence of text Strings representing alternative means of specifying the required behavior. A corresponding sequence of language Strings may be used to specify the languages in which each of the body Strings is to be interpreted. Languages are matched to body Strings by order. The UML specification does not define how body Strings are interpreted relative to any language, though other specifications may define specific language Strings to be used to indicate interpretation with respect to those specifications (e.g., “OCL” for expressions to be interpreted according to the OCL specification).

**NOTE.** It is not required to specify the languages. If they are unspecified, then the interpretation of any body Strings shall be determined implicitly from the form of the bodies or the context of use of the OpaqueBehavior.

If an OpaqueBehavior has more than one body String, then any one of the bodies can be used to determine the behavior of the OpaqueBehavior. The UML specification does not determine how this choice is made.

A FunctionBehavior is an OpaqueBehavior that does not access or modify any objects or other external data. During the execution of a FunctionBehavior, no communication or interaction with anything external to the FunctionBehavior is allowed. The amount of time to compute its results is undefined. A FunctionBehavior may raise exceptions for certain input values, in which case the computation is abandoned.

FunctionBehaviors thus represent functions that transform a set of input argument values (given by the input Parameters of the FunctionBehavior) to a set of output result values (given by the output Parameters of the FunctionBehavior). The execution of a FunctionBehavior depends only on the argument values and has no other effect than to compute result values. Examples of functions that might be modeled as FunctionBehaviors include primitive arithmetic, Boolean, and String functions.

**13.2.3.4 Behaviored Classifiers**

A BehavioredClassifier is a Classifier that may have ownedBehaviors, at most one of which may be considered to specify the behavior of the BehavioredClassifier itself. Conversely, a Behavior that is the ownedBehavior of a

BehavioredClassifier has that BehavioredClassifier as its context*.* The specification of such a Behavior may reference features of the context BehavioredClassifier as well as any other elements visible to the context BehavioredClassifier.

A Behavior that is not directly an ownedBehavior of a BehavioredClassifier may nevertheless still have a context. To determine the context of a Behavior that is not directly an ownedBehavior, find the first BehavioredClassifier reached by following the chain of ownership relationships from the Behavior, if any. If there is such a BehavioredClassifier, then it is the context, unless it is itself a Behavior with a non-empty context, in which case this is also the context for the original Behavior. For example, the context of an entry Behavior (see sub clause 14.2) in a StateMachine owned by a BehavioredClassifier is the classifier that owns the StateMachine, not the StateMachine.

A Behavior that is owned directly by a Class as a nestedClassifier (see sub clause 11.4), rather than as an ownedBehavior, does *not* have the Class as its context. The nestedClassifiers of a Class are simply nested in the Class considered as a Namespace. As a nestedClassifier, a Behavior has visibility of elements defined within the owning Class and other elements visible to that Class, and it may itself be visible outside the Class, depending on its declared visibility. But its semantics as a “stand-alone” Behavior are not otherwise affected by being nested in the Class.

If a Behavior has a context, then an execution of the Behavior always has an associated *context object* that is an instance of the context BehavioredClassifier (as long as that BehavioredClassifier is instantiable). A Behavior without a context BehavioredClassifier may still be invoked as a “stand-alone” Behavior. In this case, the Behavior execution serves as its own context object. The Behavior execution also serves as its own context object in the case that the context

BehavioredClassifier is not instantiable, that is, if it is a Component with isIndirectlyInstantiated=true (see sub clause 11.6) or a Collaboration (see sub clause 11.7). Thus, a Behavior execution always has a context object, whether or not the Behavior has an explicit, instantiable context BehavioredClassifier.

A BehavioredClassifier may have a distinguished ownedBehavior called its classifierBehavior. A classifierBehavior describes the behavior an instance of the owning Classifier may undergo in the course of its lifetime. The classifierBehavior of a BehavioredClassifier is considered to be invoked when an instance of the owning BehavioredClassifier is created and the resulting execution has the new instance as its context object. The execution is terminated if the instance is destroyed.

The precise semantics of a classifierBehavior depend on the kind of BehavioredClassifier that owns it. For example, the classifierBehavior of a Collaboration (see sub clause 11.7) represents emergent behavior of all the parts, whereas the classifierBehavior of a Class (see sub clause 11.4) is just the behavior of instances of the Class separated from the behaviors of any of its parts. However, a passive Class (with isActive=false) shall not have a classifierBehavior.

**13.2.3.5 Behavioral Features and Methods**

There are two kinds of BehavioralFeatures: Operations (see sub clause 9.6) and Receptions (see sub clause 10.3). Of the different kinds of BehavioredClassifiers in UML, only Classes may have BehavioralFeatures and only active Classes may have Receptions (see sub clause 11.4). Calling an Operation on or sending a Signal instance to an object of a Class is a *request* for the object to carry out an identified BehavioralFeature. An Operation call identifies a specific operation to be invoked. The receipt of an instance of a Signal, on the other hand, is considered to be a request for any Reception of the receiving object that references that Signal or any direct or indirect generalization of it.

A BehavioralFeature of a Class may be implemented by one or more method Behaviors. Such a BehavioralFeature specifies that instances of the owning Class will respond to a request for the BehavioralFeature by invoking one of the feature’s implementing methods. A Behavior shall be the method for no more than one BehavioralFeature, called its specification*.* The specification of a Behavior shall be an owned or inherited member of the Class of which the Behavior is an ownedBehavior. It is possible to have more than one method associated with a single BehavioralFeature, but there shall be at most one Behavior for a particular pairing of a Class (as owner of the Behavior) and a BehavioralFeature (as the specification for the Behavior). This means that a single BehavioralFeature may have methods both in its owning Class and any direct or indirect subclass of that Class, but with no more than one method per Class.

The receiving object becomes the context object for the execution of any invoked methods.

**NOTE.** Methods of a Reception are always invoked asynchronously, while the methods of an Operation may be invoked either synchronously or asynchronously, depending on how the Operation is called.

The method resolution process shall be based on the BehavioralFeature being requested, the object receiving the request and any data values associated with the request (i.e., Operation input parameter values or Signal attribute values). However, the UML specification does not mandate that a conforming UML tool support any particular resolution process. In general, the resolution process may be complicated, to include such mechanisms as before-after methods, delegation, etc. In some of these variations, multiple Behaviors may be executed as a result of a single call. If no methods are identified by the resolution process, then it is undefined what happens.

The following is a simple object-oriented resolution process for a CallEvent that always results in at most one method being identified:

If the Class of the receiving object owns a method for the Operation identified in the CallEvent, then that method is the result of the resolution. Otherwise, the superclass of the Class of the receiving object is examined for a method for the Operation, and so on up the generalization hierarchy until a method is found or the root of the hierarchy is reached. If a Class has multiple superclasses, then all of them are examined for a method. If no method is found, or a method is found in more than one ancestor Class along different paths, then the model is ill-formed for this resolution process and it results in no method.

A method of an Operation shall have Parameters corresponding to the Parameters of the Operation. Similarly, a method of a Reception shall have Parameters corresponding to the attributes of the Signal referenced by the Reception, which are considered as effective “in” Parameters of the Reception. The data values associated with a request – input Operation parameter values or Signal attribute values – are then passed to a method invoked due to the request via the method parameters. For a synchronous Operation call, output Parameter values from the method execution are also passed back to the Operation caller via the corresponding Operation output Parameters.

However, no specific approach is defined for matching the Parameters of the method to the Parameters of the BehavioralFeature. Possible approaches include exact match (i.e., the type of the corresponding Parameters, in order, must be the same), co-variant match (the type of a Parameter of the method may be a subtype of the type of the Parameter of the BehavioralFeature), contra-variant match (the type of a Parameter of the method may be a supertype of the type of the Parameter of the BehavioralFeature), or a combination thereof.

**13.2.4 Notation**

The notation for various subclasses of Behavior are defined in subsequent clauses.

The notation for Signals and Receptions is covered under Simple Classifiers in sub clause 10.3.4.

The notation for active Classes is covered under Structured Classifiers in sub clause 11.4.4.

**13.2.5 Examples**

None.

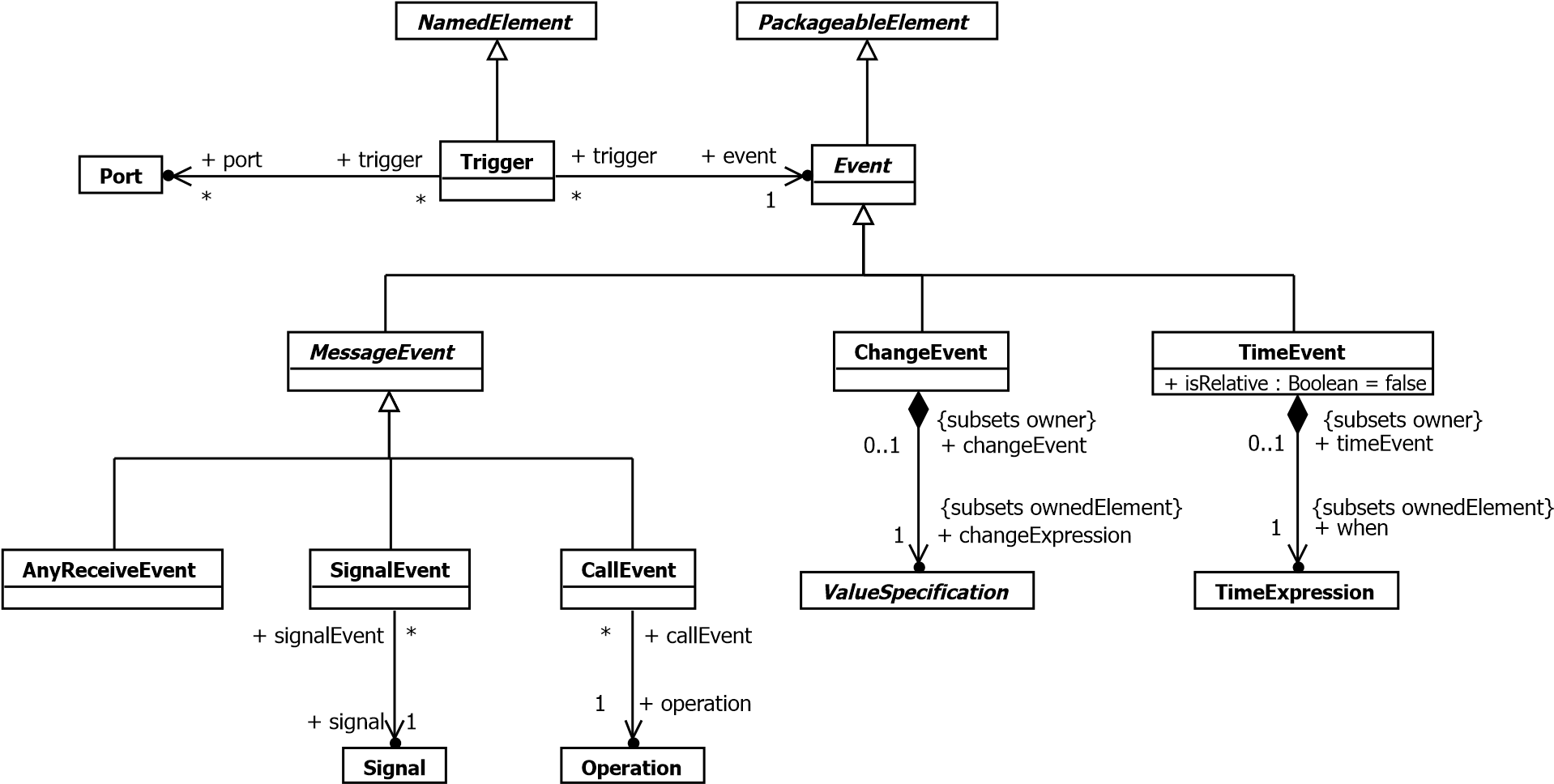
### 13.3 Events

**13.3.1 Summary**

An Event is a something that may occur at a specific instant in time. One Event may have many occurrences, which may happen at different times. In this sense, an Event can be considered a classification of its occurrences, though Events are not actually Classifiers in UML.

Of particular importance are Events that trigger a response within a Behavior. Such Events that may be explicitly modeled within UML include TimeEvents that occur at a specified time or after a duration, ChangeEvents that occur when a specified Boolean value becomes true and MessageEvents that occur on the receipt of a *message,* which is a communication from one Behavior to another requesting an Operation call or Signal reception.

**13.3.2 Abstract Syntax**



**Figure 13.2 Events**

**13.3.3 Semantics**

**13.3.3.1 Event Dispatching**

An Event is the specification of some occurrence that may potentially trigger behavioral effects. A Trigger specifies a specific point in the definition of a Behavior at which an Event occurrence may have such an effect. Event is a PackageableElement, allowing Events to be modeled independently of their use. A Trigger, however, always appears as a part of some larger behavioral specification (e.g., on a StateMachine Transition or in an AcceptEventAction). A single Event may be used in several different Triggers.

As discussed in sub clause 13.2.3, a Behavior execution always has an associated context object (which may be the execution itself). A context object mediates the handling of Event occurrences for all of its associated Behavior executions. When an Event occurrence is recognized by a context object, it may have an immediate effect or it may be saved for later triggered effect. An immediate effect is manifested by direct invocation of a Behavior as determined by the Event, such as the invocation of the method of a BehavioralFeature (see sub clause 13.2.3). A triggered effect is manifested by the storage of the occurrence in the *event pool* of the object and the later consumption of the occurrence by an ongoing Behavior execution that reaches a Trigger that matches the Event corresponding to the occurrence in the pool.

In general, when a Behavior execution comes to a *wait point* where it needs a Trigger to continue, the event pool of its context object is examined for an event that satisfies the outstanding Trigger (or Triggers). If the pool contains an event occurrence that satisfies one of the Triggers, the occurrence is removed from the pool and *dispatched* to the Behavior, which continues its execution as specified. Any data associated with the Event occurrence are made available to the triggered Behavior during its further execution.

**NOTE.** All Behaviors with the same context object share the event pool of that object, but any Event occurrence in the pool can be consumed by only one Behavior.

There is no requirement for a specific order in which Event occurrences in an event pool are examined or dispatched. If an event pool contains an occurrence that satisfies no Triggers at a wait point, then the general semantics of BehavioredClassifiers do not specify what happens to it. (However, see the specific semantics for the dispatching and deferring of event occurrences for StateMachines in sub clause 14.2.)

**13.3.3.2 Message Events**

A message is a communication in which a sender makes a request for either an Operation call or Signal reception by a receiver*.* This communication involves two events: the event of sending the message and the event of receiving the message. Sending events, however, are not modeled as explicit model elements in UML, though they are implicit in the execution of InvocationActions (see sub clause 16.3) and occurrences of such events can be modeled in Interactions (see sub clause 17.5). A MessageEvent, on the other hand, is an explicit model of the receipt of a message, in order to be able to specify a Trigger that responds to occurrences of that event.

A message contains:

* Data associated with the request being made (arguments for Operation parameters or values for Signal attributes).
* Information about the nature of the request (i.e., the BehavioralFeature invoked).
* For a synchronous invocation, sufficient information to enable the return of a reply from the invoked Behavior.

While each message is targeted at exactly one receiver object and caused by exactly one sending object, an occurrence of a sending event may result in a number of messages being generated (as in SignalBroadcastAction, see sub clause 16.3). The receiver of a message may be the same as the sender, it may be local (i.e., an object held in a slot of the currently executing Behavior or its context object) or it may be remote. The manner of transmitting the message, the amount of time required to transmit it, the order in which the transmissions reach their receiver object and the path for reaching the receiver object are undefined.

The receipt of a message is manifested as a MessageEvent occurrence. A CallEvent is a MessageEvent for messages requesting that a specific Operation be called. A SignalEvent is a MessageEvent for messages requesting the reception of an instance of a specific Signal. An AnyReceiveEvent is a MessageEvent for any message that is not explicitly handled by any other related Trigger.

In the case of a CallEvent for an Operation or a SignalEvent for a Signal that matches a Reception on the receiver, if the Operation or Reception has one or more methods, then the method resolution process described for Behavioral Features and Methods in sub clause 13.2.3 shall be carried out to determine a method to be used to handle a MessageEvent occurrence. If a method is so identified, it is invoked to respond to the message request. Otherwise, the MessageEvent occurrence is saved in the event pool of the receiving object. When a MessageEvent occurrence is dispatched from the event pool and matches a Trigger defined in the Behavior specification for the receiver, it causes the execution of a response within the Behavior.

A Trigger for an AnyReceiveEvent may be triggered by the receipt of any message (Signal send or Operation call). However, if there is a relevant SignalEvent or CallEvent Trigger that specifically matches the message, then the

AnyReceiveEvent Trigger is *not* triggered by the message. Which other Triggers are related to an AnyReceiveEvent

Trigger depends on the context of the Trigger (in particular, see sub clause 14.2 on Transitions and sub clause 16.10 on AcceptEventActions). An AnyReceiveEvent may also be triggered by the receipt of a message containing an object other than a SignalInstance, as may be sent by a SendObjectAction (see sub clause 16.3.3).

A Trigger may also specify one or more ports, in which case the event of the Trigger shall be a MessageEvent. In this case the Trigger only matches event occurrences for messages received through one of the specified Ports (see also sub clause 11.3 on EncapsulatedClassifiers and Ports).

**13.3.3.3 Change Events**

A ChangeEvent occurs when a Boolean changeExpression becomes true. For example, this could be as a result of a change in the value of some Attribute or a change in the value referenced by a link corresponding to an Association. A ChangeEvent occurs implicitly and is not the result of any explicit action.

An occurrence is considered to be generated any time the value of the changeExpression changes from false to true. However, it is not defined specifically when a changeExpression is evaluated or whether a ChangeEvent occurrence remains available for detection even if the associated changeExpression value changes back to false before the occurrence is consumed.

**13.3.3.4 Time Events**

A TimeEvent specifies an instant in time at which it occurs. The instant is specified using a TimeExpression(see sub clause 8.4). If the TimeEvent is absolute, then the time resulting from the evaluation of theTimeExpression is the absolute time at which the TimeEvent occurs. If the TimeEvent is relative, then theTimeEvent shall be used in the context of a Trigger, and the time of occurrence is relative to a starting timedetermined for the Trigger.

As discussed above under “Event Dispatching”, a Behavior may come to a wait point at which it has one ormore Triggers available to which event occurrences may be dispatched. If such an outstanding Trigger has arelative TimeEvent, then the starting time for that TimeEvent is the time at which the Behavior came to thewait point.

**13.3.4 Notation**

There is no notation for Events outside of the context of their use in Triggers. A Trigger is denoted textually based on the kind of Event it is for:

*<trigger> ::= <call-event> | <signal-event> | <any-receive-event> | <time-event> | <change-event>* where:

* A CallEvent is denoted by the name of the triggering Operation, optionally followed by an assignment specification:

*<call-event> ::= <name> [‘(‘ [<assignment-specification>] ‘)’]*

*<assignment-specification> ::= <assigned-name> [‘,’ <assigned-name>]\** where:

*<assigned-name>* is an implicit assignment of the argument value for the corresponding Parameter of the Operation to a Property or Variable of the context object for the triggered Behavior.

*<assignment-specification>* is optional and may be omitted even if the Operation has Parameters. No standard mapping is defined from an assignment specification to the UML abstract syntax. A conforming tool is not required to support this notation. If it does, it may provide a mapping to standard UML abstract syntax, e.g., by implicitly inserting Actions to carry out the behavior implied by the notation.

* A SignalEvent is denoted by name of the triggering Signal, optionally followed by an assignment specification:

*<signal-event> ::= <name> [‘(‘ [<assignment-specification>] ‘)’] <assignment-specification> ::= <attr-name> [‘,’<attr-name>]\** where *<assignment-specification>* is defined as for CallEvent above.

* Any AnyReceiveEvent is denoted by “all”:

*<any-receive-event> ::=* ‘all’

* A ChangeEvent is denoted by “when” followed by a Boolean ValueSpecification:

*<change-event> ::=* ‘when’ *<value-specification>*

See Clause 8 for the notation for various kinds of ValueSpecifications.

* A relative TimeEvent is denoted with “after” followed by a TimeExpression, such as “after 5 seconds.” An absolute TimeEvent is specified with “at” followed by a TimeExpression, such as “at Jan. 1, 2000, Noon”. *<time-event> ::= <relative-time-event> | <absolute-time-event>*

*<relative-time-event> ::=* ‘after’ *<time-expression> <absolute-time-event> ::=* ‘at’ *<time-expression>*

See also sub clause 8.4.4 on the notation for TimeExpressions.

**13.3.5 Examples**

None.

### 13.4 Classifier Descriptions

**13.4.1 AnyReceiveEvent [Class]**

**13.4.1.1 Description**

A trigger for an AnyReceiveEvent is triggered by the receipt of any message that is not explicitly handled by any related trigger.

**13.4.1.2 Diagrams**

Events

**13.4.1.3 Generalizations**

MessageEvent

**13.4.2 Behavior [Abstract Class]**

**13.4.2.1 Description**

Behavior is a specification of how its context BehavioredClassifier changes state over time. This specification may be either a definition of possible behavior execution or emergent behavior, or a selective illustration of an interesting subset of possible executions. The latter form is typically used for capturing examples, such as a trace of a particular execution.

**13.4.2.2 Diagrams**

Behaviors, Object Nodes, Activities, Control Nodes, Expressions, Structured Classifiers, Behavior State Machines, Interfaces, Interactions, Occurrences, Features, Invocation Actions, Other Actions

**13.4.2.3 Generalizations**

Class

**13.4.2.4 Specializations**

OpaqueBehavior, Activity, StateMachine, Interaction

**13.4.2.5 Attributes**

* isReentrant : Boolean [1..1] = true

Tells whether the Behavior can be invoked while it is still executing from a previous invocation.

**13.4.2.6 Association Ends**

* /context : BehavioredClassifier [0..1]{subsets RedefinableElement::redefinitionContext} (opposite

A\_context\_behavior::behavior)

The BehavioredClassifier that is the context for the execution of the Behavior. A Behavior that is directly owned as a nestedClassifier does not have a context. Otherwise, to determine the context of a Behavior, find the first BehavioredClassifier reached by following the chain of owner relationships from the Behavior, if any. If there is such a BehavioredClassifier, then it is the context, unless it is itself a Behavior with a non-empty context, in which case that is also the context for the original Behavior. For example, following this algorithm, the context of an entry Behavior in a StateMachine is the BehavioredClassifier that owns the StateMachine. The features of the context BehavioredClassifier as well as the Elements visible to the context Classifier are visible to the Behavior.

* ♦ ownedParameter : Parameter [0..\*]{ordered, subsets Namespace::ownedMember} (opposite

A\_ownedParameter\_behavior::behavior)

References a list of Parameters to the Behavior which describes the order and type of arguments that can be given when the Behavior is invoked and of the values which will be returned when the Behavior completes its execution.

* ♦ ownedParameterSet : ParameterSet [0..\*]{subsets Namespace::ownedMember} (opposite

A\_ownedParameterSet\_behavior::behavior) The ParameterSets owned by this Behavior.

* ♦ postcondition : Constraint [0..\*]{subsets Namespace::ownedRule} (opposite

A\_postcondition\_behavior::behavior)

An optional set of Constraints specifying what is fulfilled after the execution of the Behavior is completed, if its precondition was fulfilled before its invocation.

* ♦ precondition : Constraint [0..\*]{subsets Namespace::ownedRule} (opposite

A\_precondition\_behavior::behavior)

An optional set of Constraints specifying what must be fulfilled before the Behavior is invoked.

* specification : BehavioralFeature [0..1] (opposite BehavioralFeature::method)

Designates a BehavioralFeature that the Behavior implements. The BehavioralFeature must be owned by the BehavioredClassifier that owns the Behavior or be inherited by it. The Parameters of the BehavioralFeature and the implementing Behavior must match. A Behavior does not need to have a specification, in which case it either is the classifierBehavior of a BehavioredClassifier or it can only be invoked by another Behavior of the Classifier.

* redefinedBehavior : Behavior [0..\*]{subsets Classifier::redefinedClassifier} (opposite

A\_redefinedBehavior\_behavior::behavior)

References the Behavior that this Behavior redefines. A subtype of Behavior may redefine any other subtype of Behavior. If the Behavior implements a BehavioralFeature, it replaces the redefined Behavior. If the Behavior is a classifierBehavior, it extends the redefined Behavior.

**13.4.2.7 Operations**

* context() : BehavioredClassifier [0..1]

A Behavior that is directly owned as a nestedClassifier does not have a context. Otherwise, to determine the context of a Behavior, find the first BehavioredClassifier reached by following the chain of owner relationships from the Behavior, if any. If there is such a BehavioredClassifier, then it is the context, unless it is itself a Behavior with a non-empty context, in which case that is also the context for the original Behavior.

body: if nestingClass <> null then

null else

let b:BehavioredClassifier = self.behavioredClassifier(self.owner) in if b.oclIsKindOf(Behavior) and b.oclAsType(Behavior).\_'context' <> null then

b.oclAsType(Behavior).\_'context'

else b endif endif

* behavioredClassifier(from : Element) : BehavioredClassifier [0..1]

The first BehavioredClassifier reached by following the chain of owner relationships from the Behavior, if any.

body: if from.oclIsKindOf(BehavioredClassifier) then

from.oclAsType(BehavioredClassifier)

else if from.owner = null then

null else

self.behavioredClassifier(from.owner)

endif endif

* inputParameters() : Parameter [0..\*]{ordered}

The in and inout ownedParameters of the Behavior.

body: ownedParameter->select(direction=ParameterDirectionKind::\_'in' or direction=ParameterDirectionKind::inout)

* outputParameters() : Parameter [0..\*]{ordered} The out, inout and return ownedParameters.

body: ownedParameter->select(direction=ParameterDirectionKind::out or

direction=ParameterDirectionKind::inout or direction=ParameterDirectionKind::return)

**13.4.2.8 Constraints**

* most\_one\_behavior

There may be at most one Behavior for a given pairing of BehavioredClassifier (as owner of the Behavior) and BehavioralFeature (as specification of the Behavior).

inv: specification <> null implies \_'context'.ownedBehavior>select(specification=self.specification)->size() = 1

* parameters\_match

If a Behavior has a specification BehavioralFeature, then it must have the same number of ownedParameters as its specification. The Behavior Parameters must also "match" the BehavioralParameter Parameters, but the exact requirements for this matching are not formalized.

inv: specification <> null implies ownedParameter->size() = specification.ownedParameter>size()

* feature\_of\_context\_classifier

The specification BehavioralFeature must be a feature (possibly inherited) of the context BehavioredClassifier of the Behavior.

inv: \_'context'.feature->includes(specification)

**13.4.3 CallEvent [Class]**

**13.4.3.1 Description**

A CallEvent models the receipt by an object of a message invoking a call of an Operation.

**13.4.3.2 Diagrams**

Events

**13.4.3.3 Generalizations**

MessageEvent

**13.4.3.4 Association Ends**

 operation : Operation [1..1] (opposite A\_operation\_callEvent::callEvent) Designates the Operation whose invocation raised the CalEvent.

**13.4.4 ChangeEvent [Class]**

**13.4.4.1 Description**

A ChangeEvent models a change in the system configuration that makes a condition true.

**13.4.4.2 Diagrams**

Events

**13.4.4.3 Generalizations**

Event

**13.4.4.4 Association Ends**

 ♦ changeExpression : ValueSpecification [1..1]{subsets Element::ownedElement} (opposite

A\_changeExpression\_changeEvent::changeEvent)

A Boolean-valued ValueSpecification that will result in a ChangeEvent whenever its value changes from false to true.

**13.4.5 Event [Abstract Class]**

**13.4.5.1 Description**

An Event is the specification of some occurrence that may potentially trigger effects by an object.

**13.4.5.2 Diagrams**

Events

**13.4.5.3 Generalizations**

PackageableElement

**13.4.5.4 Specializations**

ChangeEvent, MessageEvent, TimeEvent

**13.4.6 FunctionBehavior [Class]**

**13.4.6.1 Description**

A FunctionBehavior is an OpaqueBehavior that does not access or modify any objects or other external data.

**13.4.6.2 Diagrams**

Behaviors

**13.4.6.3 Generalizations**

OpaqueBehavior

**13.4.6.4 Operations**

* hasAllDataTypeAttributes(d : DataType) : Boolean

The hasAllDataTypeAttributes query tests whether the types of the attributes of the given DataType are all DataTypes, and similarly for all those DataTypes.

body: d.ownedAttribute->forAll(a |

a.type.oclIsKindOf(DataType) and

hasAllDataTypeAttributes(a.type.oclAsType(DataType)))

**13.4.6.5 Constraints**

* one\_output\_parameter

A FunctionBehavior has at least one output Parameter.

inv: self.ownedParameter->

select(p | p.direction = ParameterDirectionKind::out or p.direction= ParameterDirectionKind::inout or p.direction= ParameterDirectionKind::return)->size() >= 1

* types\_of\_parameters

The types of the ownedParameters are all DataTypes, which may not nest anything but other DataTypes.

inv: ownedParameter->forAll(p | p.type <> null and

p.type.oclIsTypeOf(DataType) and hasAllDataTypeAttributes(p.type.oclAsType(DataType)))

**13.4.7 MessageEvent [Abstract Class]**

**13.4.7.1 Description**

A MessageEvent specifies the receipt by an object of either an Operation call or a Signal instance.

**13.4.7.2 Diagrams**

Events

**13.4.7.3 Generalizations**

Event

**13.4.7.4 Specializations**

AnyReceiveEvent, CallEvent, SignalEvent

**13.4.8 OpaqueBehavior [Class]**

**13.4.8.1 Description**

An OpaqueBehavior is a Behavior whose specification is given in a textual language other than UML.

**13.4.8.2 Diagrams**

Behaviors

**13.4.8.3 Generalizations**

Behavior

**13.4.8.4 Specializations**

FunctionBehavior

**13.4.8.5 Attributes**

* body : String [0..\*]

Specifies the behavior in one or more languages.

* language : String [0..\*]

Languages the body strings use in the same order as the body strings.

**13.4.9 SignalEvent [Class]**

**13.4.9.1 Description**

A SignalEvent represents the receipt of an asynchronous Signal instance.

**13.4.9.2 Diagrams**

Events

**13.4.9.3 Generalizations**

MessageEvent

**13.4.9.4 Association Ends**

 signal : Signal [1..1] (opposite A\_signal\_signalEvent::signalEvent) The specific Signal that is associated with this SignalEvent.

**13.4.10 TimeEvent [Class]**

**13.4.10.1 Description**

A TimeEvent is an Event that occurs at a specific point in time.

**13.4.10.2 Diagrams**

Events

**13.4.10.3 Generalizations**

Event

**13.4.10.4 Attributes**

* isRelative : Boolean [1..1] = false

Specifies whether the TimeEvent is specified as an absolute or relative time.

**13.4.10.5 Association Ends**

* ♦ when : TimeExpression [1..1]{subsets Element::ownedElement} (opposite A\_when\_timeEvent::timeEvent) Specifies the time of the TimeEvent.

**13.4.10.6 Constraints**

* when\_non\_negative

The ValueSpecification when must return a non-negative Integer. inv: when.integerValue() >= 0

**13.4.11 Trigger [Class]**

**13.4.11.1 Description**

A Trigger specifies a specific point at which an Event occurrence may trigger an effect in a Behavior. A Trigger may be qualified by the Port on which the Event occurred.

**13.4.11.2 Diagrams**

Events, Behavior State Machines, Accept Event Actions

**13.4.11.3 Generalizations**

NamedElement

**13.4.11.4 Association Ends**

* event : Event [1..1] (opposite A\_event\_trigger::trigger) The Event that detected by the Trigger.
* port : Port [0..\*] (opposite A\_port\_trigger::trigger)

A optional Port of through which the given effect is detected.

**13.4.11.5 Constraints**

* trigger\_with\_ports

If a Trigger specifies one or more ports, the event of the Trigger must be a MessageEvent.

inv: port->notEmpty() implies event.oclIsKindOf(MessageEvent)

### 13.5 Association Descriptions

**13.5.1 A\_changeExpression\_changeEvent [Association]**

**13.5.1.1 Diagrams**

Events

**13.5.1.2 Owned Ends**

* changeEvent : ChangeEvent [0..1]{subsets Element::owner} (opposite ChangeEvent::changeExpression)

**13.5.2 A\_context\_behavior [Association]**

**13.5.2.1 Diagrams**

Behaviors

**13.5.2.2 Owned Ends**

* behavior : Behavior [0..\*]{subsets A\_redefinitionContext\_redefinableElement::redefinableElement} (opposite Behavior::context)

**13.5.3 A\_event\_trigger [Association]**

**13.5.3.1 Diagrams**

Events

**13.5.3.2 Owned Ends**

* trigger : Trigger [0..\*] (opposite Trigger::event)

**13.5.4 A\_operation\_callEvent [Association]**

**13.5.4.1 Diagrams**

Events

**13.5.4.2 Owned Ends**

* callEvent : CallEvent [0..\*] (opposite CallEvent::operation)

**13.5.5 A\_ownedParameterSet\_behavior [Association]**

**13.5.5.1 Diagrams**

Behaviors

**13.5.5.2 Owned Ends**

* behavior : Behavior [0..1]{subsets NamedElement::namespace} (opposite Behavior::ownedParameterSet)

**13.5.6 A\_ownedParameter\_behavior [Association]**

**13.5.6.1 Diagrams**

Behaviors

**13.5.6.2 Owned Ends**

* behavior : Behavior [0..1]{subsets NamedElement::namespace} (opposite Behavior::ownedParameter)

**13.5.7 A\_port\_trigger [Association]**

**13.5.7.1 Diagrams**

Events

**13.5.7.2 Owned Ends**

* trigger : Trigger [0..\*] (opposite Trigger::port)

**13.5.8 A\_postcondition\_behavior [Association]**

**13.5.8.1 Diagrams**

Behaviors

**13.5.8.2 Owned Ends**

* behavior : Behavior [0..1]{subsets Constraint::context} (opposite Behavior::postcondition)

**13.5.9 A\_precondition\_behavior [Association]**

**13.5.9.1 Diagrams**

Behaviors

**13.5.9.2 Owned Ends**

* behavior : Behavior [0..1]{subsets Constraint::context} (opposite Behavior::precondition)

**13.5.10 A\_redefinedBehavior\_behavior [Association]**

**13.5.10.1 Diagrams**

Behaviors

**13.5.10.2 Owned Ends**

* behavior : Behavior [0..\*]{subsets A\_redefinedClassifier\_classifier::classifier} (opposite Behavior::redefinedBehavior)

**13.5.11 A\_signal\_signalEvent [Association]**

**13.5.11.1 Diagrams**

Events

**13.5.11.2 Owned Ends**

* signalEvent : SignalEvent [0..\*] (opposite SignalEvent::signal)

**13.5.12 A\_when\_timeEvent [Association]**

**13.5.12.1 Diagrams**

Events

**13.5.12.2 Owned Ends**

* timeEvent : TimeEvent [0..1]{subsets Element::owner} (opposite TimeEvent::when)