

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

where `<parameter-kind>` is the name of the metaclass for the exposed element. The syntax of `<template-param-name>` depends on the kind of parameteredElement for this template parameter substitution and of `<default>` depends upon the kind of element. See “ParameterableElement (from Templates)” on page 602 (and its subclasses).

Examples

See TemplateableElement.

17.5.5 TemplateParameterSubstitution (from Templates)

A template parameter substitution relates the actual parameter(s) to a formal template parameter as part of a template binding.

Generalizations

- “Element (from Kernel)” on page 60

Description

TemplateParameterSubstitution associates one or more actual parameters with a formal template parameter within the context of a TemplateBinding.

Attributes

No additional attributes

Associations

- `actual` : ParameterableElement[1..*] The elements that are the actual parameters for this substitution.
- `binding` : TemplateBinding[1] The template binding that owns this substitution. Subsets Element::owner.
- `formal` : TemplateParameter[1] The formal template parameter that is associated with this substitution.
- `ownedActual` : ParameterableElement[0..*] The actual parameters that are owned by this substitution. Subsets Element::ownedElement and actual.

Constraints

- [1] The actual parameter must be compatible with the formal template parameter (e.g., the actual parameter for a class template parameter must be a class).

`actual->forAll(a | a.isCompatibleWith(formal.parameteredElement))`

Semantics

A TemplateParameterSubstitution specifies the set of actual parameters to be substituted for a formal template parameter within the context of a template binding.

Notation

See TemplateBinding.

Examples

See `TemplateBinding`.

17.5.6 `TemplateSignature` (from `Templates`)

A template signature bundles the set of formal template parameters for a templated element.

Generalizations

- “Element (from Kernel)” on page 60

Description

A `TemplateSignature` is owned by a `TemplateableElement` and has one or more `TemplateParameters` that define the signature for binding this template.

Attributes

No additional attributes

Associations

- `ownedParameter` : `TemplateParameter`[*] The formal template parameters that are owned by this template signature. Subsets `parameter` and `Element::ownedElement`.
- `parameter` : `TemplateParameter`[1..*] The complete set of formal template parameters for this template signature.
- `template` : `TemplateableElement`[1] The element that owns this template signature. Subsets `Element::owner`.

Constraints

- [1] Parameters must own the elements they parameter or those elements must be owned by the element being templated.
`templatedElement.ownedElement->includesAll(parameter.parameteredElement - parameter.ownedParameteredElement)`
- [2] `parameter` is the owned parameter.
`parameter = ownedParameter`

Semantics

A `TemplateSignature` specifies the set of formal template parameters for the associated templated element. The formal template parameters specify the elements that may be substituted in a binding of the template.

There are constraints on what may be parametered by a template parameter. Either the parameter owns the parametered element, or the element is owned, directly or indirectly, by the template subclasses of `TemplateSignature` can add additional rules constraining what a parameter can reference in the context of a particular kind of template.

Notation

See `TemplateableElement` for a description of how the template parameters are shown as part of the notation for the template.

Examples

See `TemplateableElement`.

ClassifierTemplates

The Classifier templates diagram specifies the abstract mechanisms that support defining classifier templates, bound classifiers, and classifier template parameters. Specific subclasses of Classifier must also specialize one or more of the abstract metaclasses defined in this diagram in order to expose these capabilities in a concrete manner.

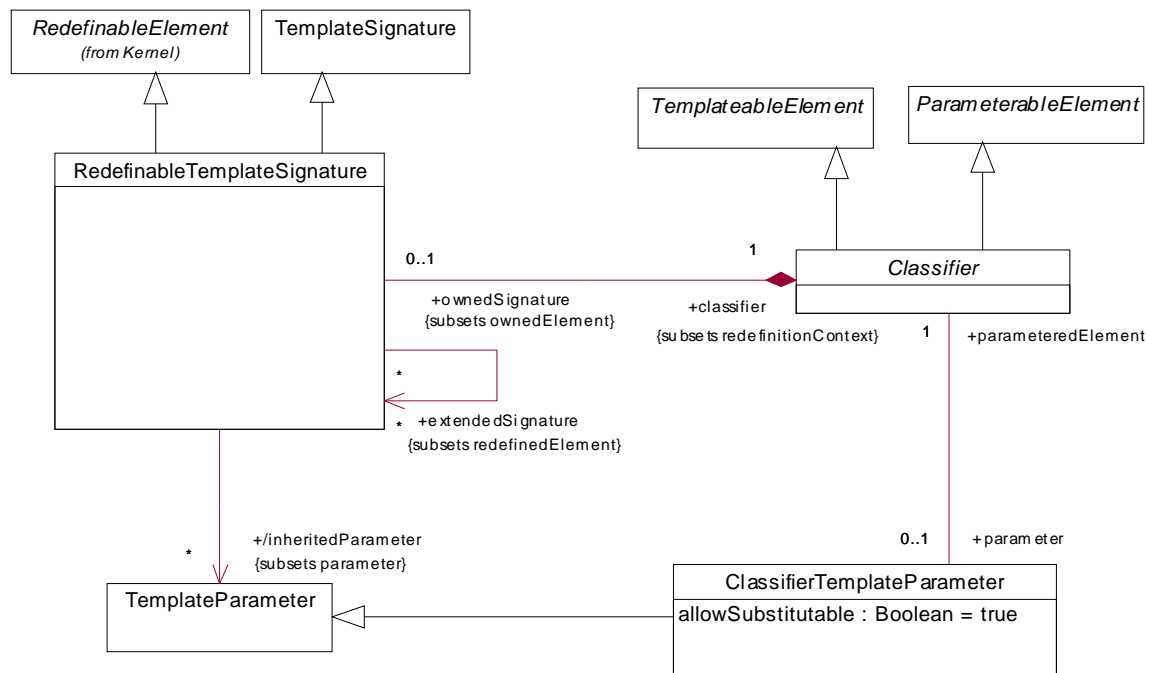


Figure 17.18 - Classifier templates

17.5.7 Classifier (from Templates)

Classifier is defined to be a kind of templateable element so that a classifier can be parameterized, and as a kind of parameterable element so that a classifier that can be exposed as a formal template parameter, and provided as an actual parameter in a binding of a template.

Generalizations

- “ParameterableElement (from Templates)” on page 602
- “TemplateableElement (from Templates)” on page 604
- “Classifier (from Kernel, Dependencies, PowerTypes)” on page 48 (*merge increment*)

Description

Classifier specializes `Kernel::Classifier` `TemplateableElement` and `ParameterableElement` to specify that a classifier can be parameterized, be exposed as a formal template parameter, and can be specified as an actual parameter in a binding of a template.

A classifier with template parameters is often called a template classifier, while a classifier with a binding is often called a bound classifier.

By virtue of `Classifier` being defined here, all subclasses of `Classifier` (such as `Class`, `Collaboration`, `Component`, `Datatype`, `Interface`, `Signal`, and `UseCases`) can be parameterized, bound, and used as template parameters. The same holds for `Behavior` as a subclass of `Class`, and thereby all subclasses of `Behavior` (such as `Activity`, `Interaction`, `StateMachine`).

Attributes

No additional attributes

Associations

- `ownedSignature` : `RedefinableTemplateSignature`[0..1]
The optional template signature specifying the formal template parameters. Subsets `Element::ownedElement`.
- `parameter` : `ParameterableElement` [0..1]
The template parameter that exposes this element as a formal parameter. Redefines `ParameterableElement::parameter`.

Constraints

No additional constraints

Additional Operations

[1] The query `isTemplate()` returns whether this templateable element is actually a template.

`Classifier::isTemplate() : Boolean;`

`isTemplate = oclAsType(TemplatableElement).isTemplate() or general->exists(g | g.isTemplate())`

Semantics

Classifier in general

`Classifier` provides the abstract mechanism that can be specialized to support subclass of `Classifiers` to be templates, exposing subclasses of `Classifier` as formal template parameters, and as actual parameters in a binding of a template.

`Classifier` as a kind of templateable element provides the abstract mechanism that can be specialized by subclasses of `Classifier` to support being defined as a template, or being bound to a template.

A bound classifier may have contents in addition to those of the template classifier. In this case the semantics are equivalent to inserting an anonymous general classifier that contains the contents, and the bound classifier is defined to be a specialization this anonymous general classifier. This supports the use of elements within the bound classifier as actual parameters in a binding.

A bound classifier may have multiple bindings. In this case the semantics are equivalent to inserting an anonymous general bound classifier for each binding, and specializing all these bound classifiers by this (formerly) bound classifier.

The formal template parameters for a classifier include all the formal template parameters of all the templates it specializes. For this reason the classifier may reference elements that are exposed as template parameters of the specializes templates.

Collaboration

A Collaboration supports the ability to be defined as a template. A collaboration may be defined to be bound from template collaboration(s).

A collaboration template will typically have the types of its parts as class template parameters. Consider the Collaboration in Figure 9.11 on page 165. This Collaboration can be bound from a Collaboration template of the form found in Figure 17.23, by means of the binding described in Figure 17.24. We have here used that the default kind of template parameter is a class (i.e., `SubjectType` and `ObserverType` are class template parameters).

A bound Collaboration is not the same as a `CollaborationUse`; in fact, parameterized Collaborations (and binding) cannot express what `CollaborationUses` can. Consider the Sale Collaboration in Figure 9.14 on page 168. It is defined by means of two parts (Buyer and Seller) representing roles in this collaboration. The two `CollaborationUses` “wholesale” and “retail” in Figure 9.15 on page 169 cannot be defined as bound Collaborations.

A bound Collaboration is a Collaboration, while a `CollaborationUse` is not. A `CollaborationUse` is defined by means of `RoleBindings`, binding parts in a Collaboration (here Buyer and Seller) to parts in another classifier (here broker, produce, and consumer in `BrokeredSale`) with the semantics that the interaction described in Sale will occur between broker, producer, and consumer. Binding eventual Buyer and Seller part template parameters (of a Sale Collaboration template) to broker and producer would just provide that the parts broker and producer are visible within the bound Sale Collaboration. And anyway, even if Sale had two part template parameters Buyer and Seller, it could not use these for defining an internal structure as is the case in Figure 9.15 on page 169. Parameters, by the very nature, represent elements that are defined ‘outside’ a Collaboration template and can therefore not be parts of an internal structure of the Collaboration template.

Semantic Variation Points

If template parameter constraints apply, then the actual classifier is constrained as follows. If the classifier template parameter:

- has a generalization, then an actual classifier must have generalization with the same general classifier.
- has a substitution, then an actual classifier must have a substitution with the same contract.
- has neither a generalization nor a substitution, then an actual classifier can be any classifier.

If template parameter constraints do not apply, then an actual classifier can be any classifier.

Notation

See `ClassifierTemplateParameter` for a description of how a parameterable classifier is displayed as a formal template parameter.

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

When a bound classifier is used directly as the type of an attribute, then *<classifier expression>* acts as the *type* of the attribute in the notation for an attribute:

$$[<visibility>] ['/' <name> [':' <attr-type>] [['<multiplicity>'] ['=' <default>] [['<attr-property>' [',' <attr-property>] *]]]]$$

When a bound classifier is used directly as the type of a part, then *<classifier-expression>* acts as the *classname* of the part in the notation for a part:

(([<name>] ':' <classname>) / <name>) ['<multiplicity>']

Presentation Options

Collaboration extends the presentation option for bound elements described under *TemplateableElement* so that the binding information can be displayed in the internal structure compartment.

Examples

Class templates

As *Classifier* is an abstract class, the following example is an example of concrete subclass (*Class*) of *Classifier* being a template.

The example shows a class template (named *FArray*) with two formal template parameters. The first formal template parameter (named *T*) is an unconstrained class template parameter. The second formal template parameter (named *k*) is an integer expression template parameter that has a default of 10. There is also a bound class (named *AddressList*) that substitutes the *Address* for *T* and 3 for *k*.

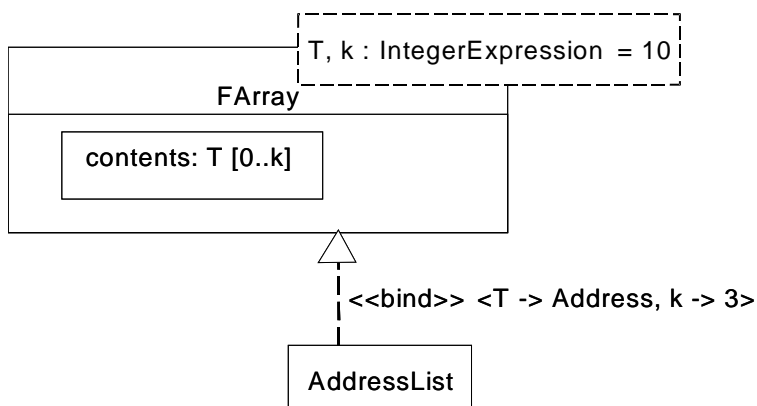


Figure 17.19 - Template Class and Bound Class

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

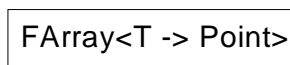


Figure 17.20 - Anonymous Bound Class

The following figure shows a template class (named Car) with two formal template parameters. The first formal template parameter (named CarEngine) is a class template parameter that is constrained to conform to the Engine class. The second formal template parameter (named n) is an integer expression template parameter.

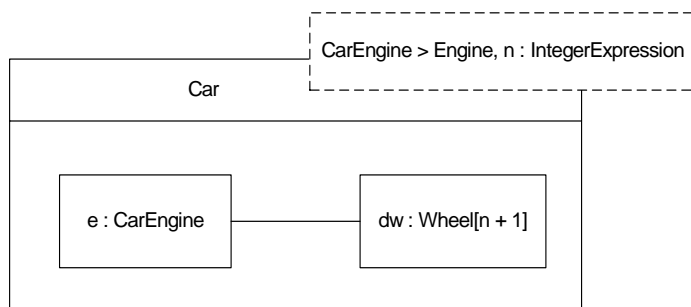


Figure 17.21 - Template Class with a constrained class parameter

The following figure shows a bound class (named DieselCar) that binds CarEngine to DieselEngine and n to 2.

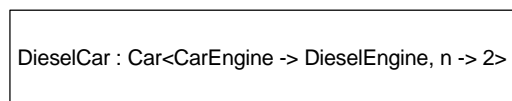


Figure 17.22 - Bound Class

Collaboration templates

The example below shows a collaboration template (named ObserverPattern) with two formal template parameters (named SubjectType and ObserverType). Both formal template parameters are unconstrained class template parameters.

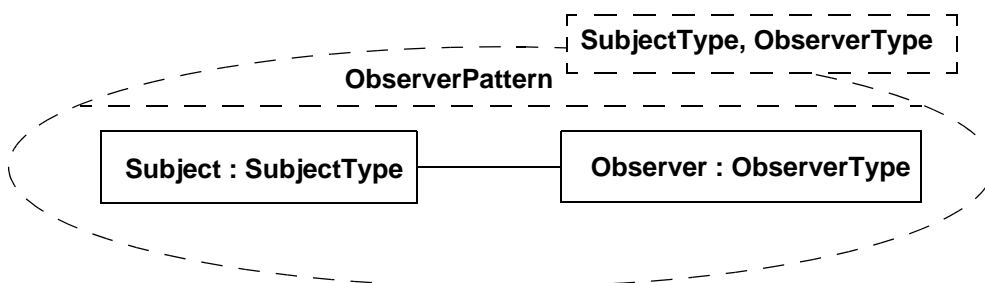


Figure 17.23 - Template Collaboration

The following figure shows a bound collaboration (named Observer) that substitutes CallQueue for SubjectType, and SlidingBarIcon for ObserverType.

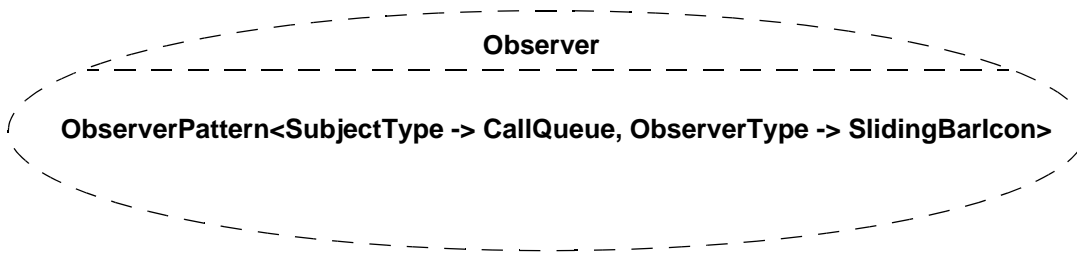


Figure 17.24 - Bound Collaboration

17.5.8 ClassifierTemplateParameter (from Templates)

A classifier template parameter exposes a classifier as a formal template parameter.

Generalizations

- “TemplateParameter (from Templates)” on page 607

Description

ClassifierTemplateParameter is a template parameter where the parametered element is a Classifier in its capacity of being a kind of ParameterableElement.

Attributes

- allowSubstitutable : Boolean[1]
Constrains the required relationship between an actual parameter and the parameteredElement for this formal parameter. Default is true.

Associations

- parameteredElement : Classifier[1]
The parameterable classifier for this template parameter. Redefines TemplateParameter::parameteredElement.

Constraints

No additional constraints

Semantics

See Classifier for additional semantics related to the compatibility of actual and formal classifier parameters.

Notation

A classifier template parameter extends the notation for a template parameter to include an optional type constraint:

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

<constraint> ::= [‘{’ *<contract>* ‘}’] *<classifier-name>*

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

The *parameter-kind* indicates the metaclass of the parametered element. It may be suppressed if it is ‘class’.

The *classifier-name* of *constraint* designates the type constraint of the parameter, which reflects the general classifier for the parametered element for this template parameter. The ‘contract’ option indicates that `allowsSubstitutable` is true, meaning the actual parameter must be a classifier that may substitute for the classifier designated by the *classifier-name*. A classifier template parameter with a constraint but without ‘contract’ indicates that the actual classifier must be a specialization of the classifier designated by the *classifier-name*.

Examples

See Classifier.

17.5.9 RedefinableTemplateSignature (from Templates)

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

Generalizations

- “TemplateSignature (from Templates)” on page 610
- “RedefinableElement (from Kernel)” on page 125

Description

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

Attributes

No additional attributes

Associations

- classifier : Classifier[1]
The classifier that owns this template signature. Subsets RedefinableElement::redefinitionContext and Template::templatedElement.
- / inheritedParameter : TemplateParameter[*]
The formal template parameters of the extendedSignature. Subsets Template::parameter.
- extendedSignature : RedefinableTemplateSignature[*]
The template signature that is extended by this template signature. Subsets RedefinableElement::redefinedElement.

Constraints

- [1] The inherited parameters are the parameters of the extended template signature.
inheritedParameter = if extendedSignature->isEmpty() then Set{} else extendedSignature.parameter endif

Additional Operations

- [1] The query `isConsistentWith()` specifies, for any two RedefinableTemplateSignatures in a context in which redefinition is possible, whether redefinition would be logically consistent. A redefining template signature is always consistent with a redefined template signature, since redefinition only adds new formal parameters.
- RedefinableTemplateSignature::isConsistentWith(redefinee: RedefinableElement): Boolean;
pre: redefinee.isRedefinitionContextValid(self)

isConsistentWith = redefinee.ocllsKindOf(RedefineableTemplateSignature)

Semantics

A RedefinableTemplateSignature may extend an inherited template signature in order to specify additional formal template parameters that apply within the templateable classifier that owns this RedefinableTemplateSignature. All the formal template parameters of the extended signatures are included as formal template parameters of the extending signature, along with any parameters locally specified for the extending signature.

Notation

Notation as for redefinition in general.

Package Templates

The Package templates diagram supports the specification of template packages and package template parameters.

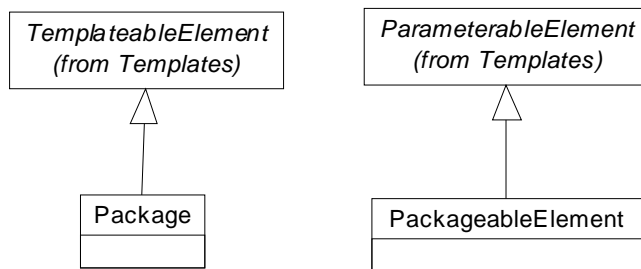


Figure 17.25 - Package templates

17.5.10 Package (from Templates)

Generalizations

- “TemplateableElement (from Templates)” on page 604
- “Package (from Kernel)” on page 103

Description

Package specializes TemplateableElement and PackageableElement specializes ParameterableElement to specify that a package can be used as a template and a PackageableElement as a template parameter.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

A Package supports the ability to be defined as a template and PackageableElements may, therefore, be parameterized in a package template. A package template parameter may refer to any element owned or used by the package template, or templates nested within it, and so on recursively. That is, there are no special rules concerning how the parameters of a package template are defined.

A package may be defined to be bound to one or more template packages. The semantics for these bindings is as described in the general case, with the exception that we need to spell out the rules for merging the results of multiple bindings. In that case, the effect is the same as taking the intermediate results and merging them into the eventual result using package merge. This is illustrated by the example below.

Notation

See TemplateableElement for a description of the general notation that is defined to support these added capabilities.

Examples

The example below shows a package template (named ResourceAllocation) with three formal template parameters. All three formal template parameters (named Resource, ResourceKind, and System) are unconstrained class template parameters. There is also a bound package (named CarRental) that substitutes Car for Resource, CarSpec for ResourceKind, and CarRentalSystem for System.

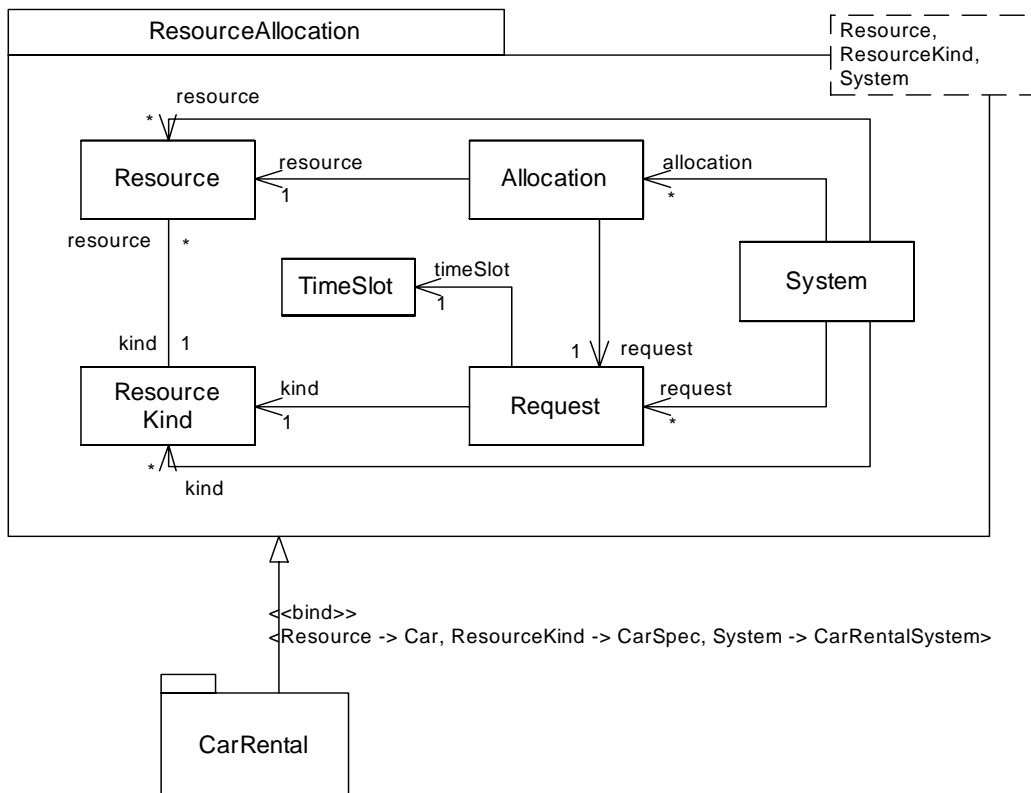


Figure 17.26 - Template Package and Bound Package

17.5.11 PackageableElement (from Templates)

Generalizations

- “ParameterableElement (from Templates)” on page 602
- “PackageableElement (from Kernel)” on page 105

Description

PackageableElements are extended to enable any such element to serve as a template parameter.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

NameExpressions

The NameExpressions diagram supports the use of string expressions to specify the name of a named element.

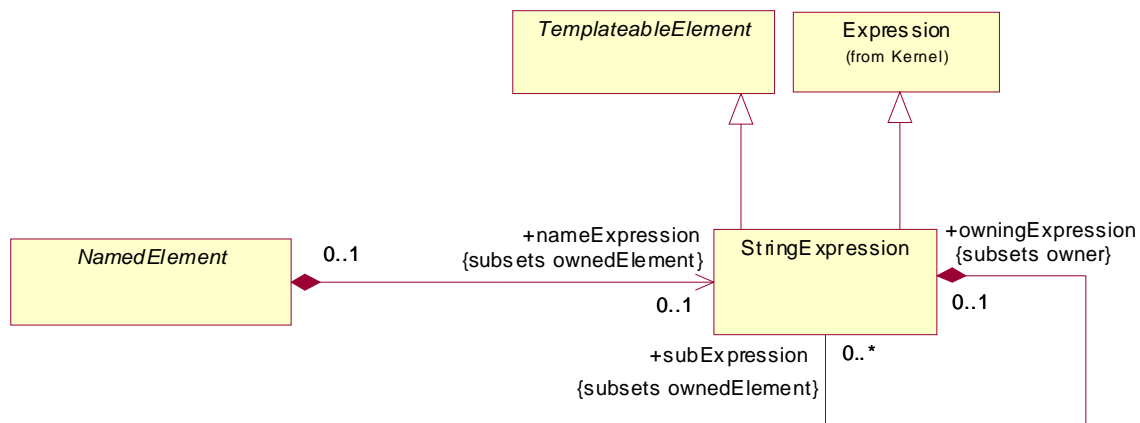


Figure 17.27 - Name expression

17.5.12 NamedElement (from Templates)

A named element is extended to support using a string expression to specify its name. This allows names of model elements to involve template parameters. The actual name is evaluated from the string expression only when it is sensible to do so (e.g., when a template is bound).

Generalizations

- “NamedElement (from Kernel, Dependencies)” on page 93 (*merge increment*)

Description

NamedElement specializes Kernel::NamedElement and adds a composition association to Expression.

Attributes

No additional attributes

Associations

- nameExpression : Expression [0..1] The expression used to define the name of this named element.

Constraints

No additional constraints

Semantics

A NamedElement may, in addition to a name, be associated with a string expression. This expression is used to calculate the name in the special case when it is a string literal. This allows string expressions, whose sub expressions may be parametered elements, to be associated with named elements in a template. When a template is bound, the sub expressions are substituted with the actuals substituted for the template parameters. In many cases, the resultant expression will be a string literal (if we assume that a concatenation of string literals is itself a string literal), in which case this will provide the name of the named element.

A NamedElement may have both a name and a name expression associated with it. In which case, the name can be used as an alias for the named element, which will surface, for example, in an OCL string. This avoids the need to use string expressions in surface notation, which is often cumbersome, although it doesn't preclude it.

Notation

The expression associated with a named element can be shown in two ways, depending on whether an alias is required or not. Both notations are illustrated in Figure 17.28.

- *No alias*: The string expression appears as the name of the model element.
- *With alias*: Both the string expression and the alias is shown wherever the name usually appears. The alias is given first and the string expression underneath.

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

Examples

The figure shows a modified version of the ResourceAllocation package template where the first two formal template parameters have been changed to be string expression parameters. These formal template parameters are used within the package template to name some of the classes and association ends. The figure also shows a bound package (named TrainingAdmin) that has two bindings to this ResourceAllocation template. The first binding substitutes the string “Instructor”

for Resource, the string “Qualification” for ResourceKind, and the class TrainingAdminSystem for System. The second binding substitutes the string “Facility” for Resource, the string “FacilitySpecification” for ResourceKind, and the class TrainingAdminSystem is again substituted for System.

The result of the binding includes both classes Instructor, Qualification, and InstructorAllocation as well as classes Facility, FacilitySpecification, and FacilityAllocation. The associations are similarly replicated. Note that Request will have two attributes derived from the single <resourceKind> attribute (shown here by an arrow), namely qualification and facilitySpecification.

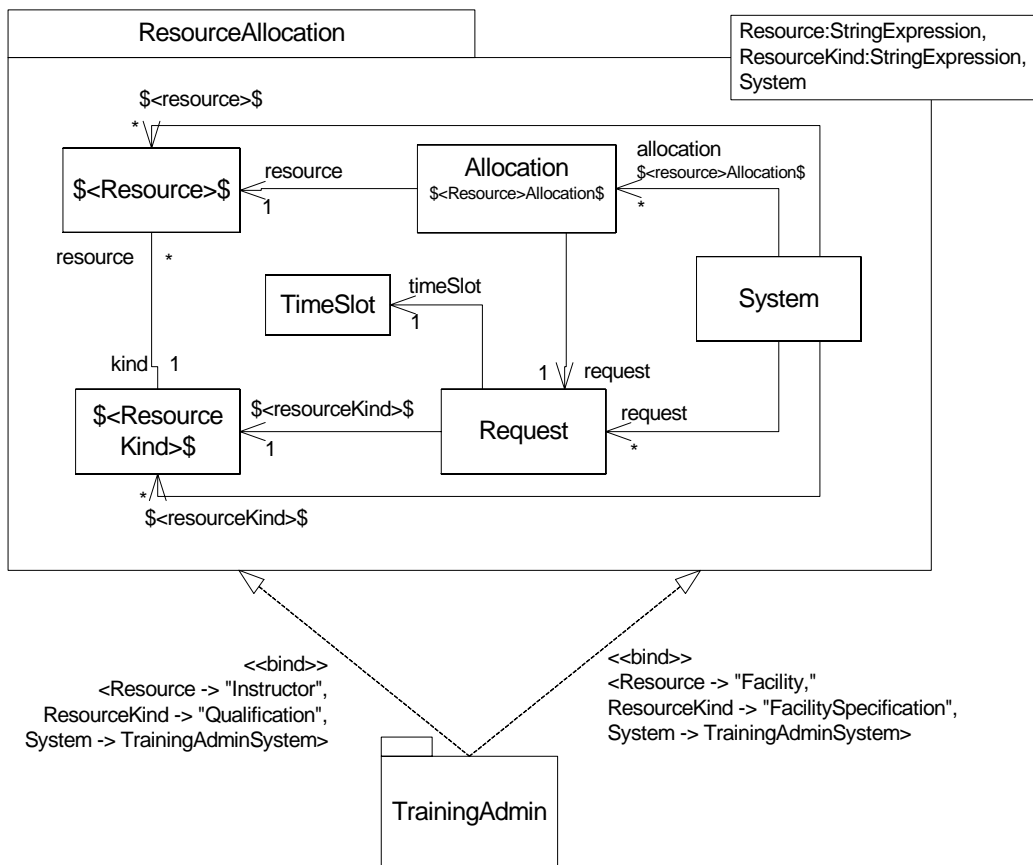


Figure 17.28 - Template Package with string parameters

17.5.13 StringExpression (from Templates)

An expression that specifies a string value that is derived by concatenating a set of sub string expressions, some of which might be template parameters.

Generalizations

- “TemplateableElement (from Templates)” on page 604
- “Expression (from Kernel)” on page 65

Description

StringExpression is a specialization of the general Expression metaclass that adds the ability to contain sub expressions and whose operands are exclusively LiteralStrings.

Attributes

No additional attributes

Associations

- subExpression : StringExpression[0..*] The StringExpressions that constitute this StringExpression. Subsets *Element::ownedElement*

Constraints

- [1] All the operands of a StringExpression must be LiteralStrings.
operand->forAll (op | op.oclIsKindOf (LiteralString))
- [2] If a StringExpression has sub expressions, it cannot have operands and vice versa (this avoids the problem of having to define a collating sequence between operands and subexpressions).
if subExpression->notEmpty() **then** operand->isEmpty() **else** operand->notEmpty()

Additional Operations

- [1] The query stringValue() returns the string that concatenates, in order, all the component string literals of all the subexpressions that are part of the StringExpression.
StringExpression::stringValue() : String;
if subExpression->notEmpty()
 then subExpression->iterate(se; stringValue = "" | stringValue.concat(se.stringValue()))
 else operand->iterate()(op; stringValue = "" | stringValue.concat(op.value))

Semantics

A StringExpression is a composite expression whose elements are either nested StringExpressions or LiteralStrings. The string value of the expression is obtained by concatenating the respective string values of all the subexpressions in order. If the expression has no subexpressions, the string value is obtained by concatenating the LiteralStrings that are the operands of the StringExpression in order.

Notation

See the notation section of “NamedElement (from Templates)” on page 620.

Examples

See Figure 17.28 on page 622.

Operation templates

The Operation templates diagram supports the specification of operation templates.

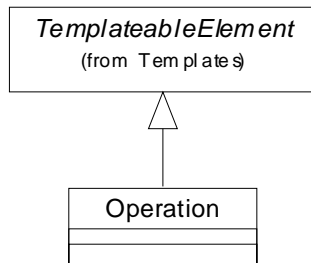


Figure 17.29 - Operation templates

17.5.14 Operation (from Templates)

Generalizations

- “TemplateableElement (from Templates)” on page 604
- “Operation (from Kernel, Interfaces)” on page 99 (*merge increment*)

Description

Operation specializes TemplateableElement in order to support specification of template operations and bound operations.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

An Operation supports the ability to be defined as a template. An operation may be defined to be bound to template operation(s).

Notation

The template parameters and template parameter binding of a template operation are two lists in between the name of the operation and the parameters of the operation.

`<visibility> <name> ‘< <template-parameter-list> ‘>’ ‘<< <binding-expression-list> ‘>>’ (‘ <parameter> [‘,’<parameter>]* ‘)’ [‘:’ <property-string>]`

OperationTemplateParameters

The Operation template parameters diagram supports the specification of operation template parameters.

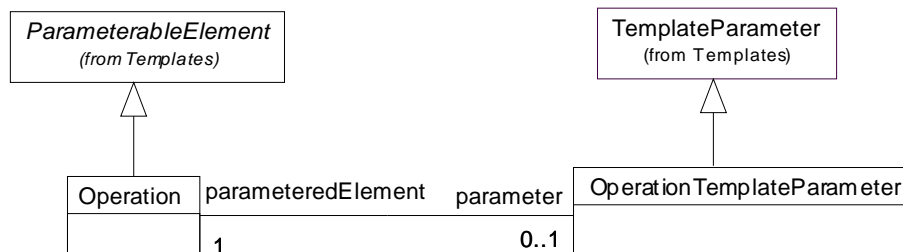


Figure 17.30 - Operation template parameters

17.5.15 Operation (from Templates)

Generalizations

- “ParameterableElement (from Templates)” on page 602
- “Operation (from Kernel, Interfaces)” on page 99 (*merge increment*)

Description

Operation specializes ParameterableElement to specify that an operation can be exposed as a formal template parameter, and provided as an actual parameter in a binding of a template.

Attributes

No additional attributes

Associations

- **parameter** : ParameterableElement [0..1]
The template parameter that exposes this element as a formal parameter. Redefines ParameterableElement::parameter.

Constraints

No additional constraints

Semantics

An Operation may be exposed by a template as a formal template parameter. Within a template classifier an operation template parameter may be used as any other operation defined in an enclosing namespace. Any references to the operation template parameter within the template will end up being a reference to the actual operation in the bound classifier. For example, a call to the operation template parameter will be a call to the actual operation.

Notation

See OperationTemplateParameter for a description of the general notation that is defined to support these added capabilities.

Within the notation for formal template parameters and template parameter bindings, an operation is shown as *<operation-name> ‘(‘ [<operation-parameter> [‘,’ <operation-parameter>]*] ‘)’*.

17.5.16 OperationTemplateParameter (from Templates)

An operation template parameter exposes an operation as a formal parameter for a template.

Generalizations

- “TemplateParameter (from Templates)” on page 607

Description

OperationTemplateParameter is a template parameter where the parametered element is an Operation.

Attributes

No additional attributes

Associations

- parameteredElement : Operation[1] The operation for this template parameter. Redefines TemplateParameter::parameteredElement.

Constraints

No additional constraints

Semantics

See Operation for additional semantics related to the compatibility of actual and formal operation parameters.

Notation

An operation template parameter extends the notation for a template parameter to include the parameters for the operation:

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

<parameter> ::= <operation-name> ‘(‘ [<op-parameter> [‘,’ <op-parameter>]] ‘)’*

<default> ::= <operation-name> ‘(‘ [<op-parameter> [‘,’ <op-parameter>]] ‘)’*

ConnectableElement template parameters

The Connectable element template parameters package supports the specification of ConnectableElement template parameters.

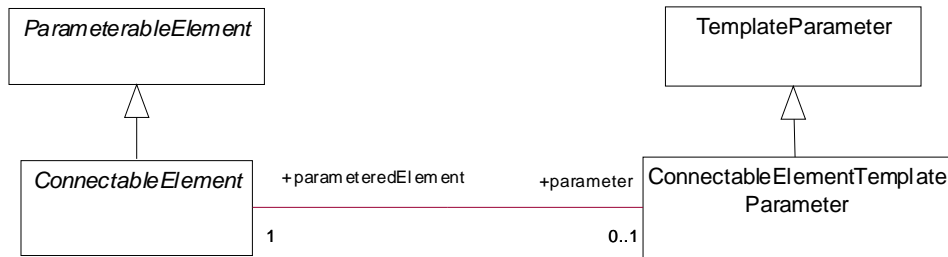


Figure 17.31 - Connectable element template parameters

17.5.17 ConnectableElement (from Templates)

A connectable element may be exposed as a connectable element template parameter.

Generalizations

- “ParameterableElement (from Templates)” on page 602.
- “ConnectableElement (from InternalStructures)” on page 170 (*merge increment*).

Description

ConnectableElement is the connectable element of a ConnectableElementTemplateParameter.

Attributes

No additional attributes

Associations

- **parameter** : ConnectableElementTemplateParameter [0..1]
The ConnectableElementTemplateParameter for this ConnectableElement parameter. Redefines TemplateParameter::parameter.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

17.5.18 ConnectableElementTemplateParameter (from Templates)

A connectable element template parameter exposes a connectable element as a formal parameter for a template.

Generalizations

- “TemplateParameter (from Templates)” on page 607.

Description

ConnectableElementTemplateParameter is a template parameter where the parametered element is a ConnectableElement.

Attributes

No additional attributes

Associations

- parameteredElement : ConnectableElement[1]
The ConnectableElement for this template parameter. Redefines TemplateParameter::parameteredElement.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

PropertyTemplateParameters

The Property template parameters diagram supports the specification of property template parameters.

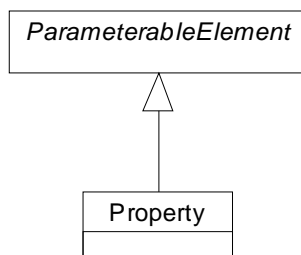


Figure 17.32 - The elements defined in the PropertyTemplates package

17.5.19 Property (from Templates)

Generalizations

- “ParameterableElement (from Templates)” on page 602
- “Property (from Kernel, AssociationClasses)” on page 118

Description

Property specializes ParameterableElement to specify that a property can be exposed as a formal template parameter, and provided as an actual parameter in a binding of a template.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] A binding of a property template parameter representing an attribute must be to an attribute.

Additional Operations

[1] The query `isCompatibleWith()` determines if this parameterable element is compatible with the specified parameterable element. By default parameterable element P is compatible with parameterable element Q if the kind of P is the same or a subtype as the kind of Q. In addition, for properties, the type must be conformant with the type of the specified parameterable element.

`Property::isCompatibleWith(p : ParameterableElement) : Boolean;`

`isCompatibleWith = p->oclIsKindOf(self.oclType) and self.type.conformsTo(p.oclAsType(TypedElement).type)`

Semantics

A Property may be exposed by a template as a formal template parameter. Within a template a property template parameter may be used as any other property defined in an enclosing namespace. Any references to the property template parameter within the template will end up being a reference to the actual property in the bound element.

Notation

See ParameterableElement for a description of the general notation that is defined to support these added capabilities.

ValueSpecificationTemplateParameters

The ValueSpecification template parameters diagram supports the specification of value specification template parameters.

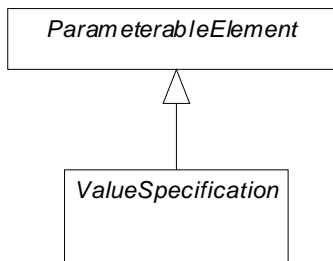


Figure 17.33 - ValueSpecification template parameters

17.5.20 ValueSpecification (from Templates)

Generalizations

- “ParameterableElement (from Templates)” on page 602
- “ValueSpecification (from Kernel)” on page 132 (*merge increment*)

Description

`ValueSpecification` specializes `ParameterableElement` to specify that a value specification can be exposed as a formal template parameter, and provided as an actual parameter in a binding of a template.

Attributes

No additional attributes

Associations

- `typeConstraint : Classifier[0..1]` Optional specification of the type of the value.

Constraints

No additional attributes

Additional Operations

- [1] The query `isCompatibleWith()` determines if this parameterable element is compatible with the specified parameterable element. By default parameterable element `P` is compatible with parameterable element `Q` if the kind of `P` is the same or a subtype as the kind of `Q`. In addition, for `ValueSpecification`, the type must be conformant with the type of the specified parameterable element.

Property: `isCompatibleWith(p : ParameterableElement) : Boolean;`

`isCompatibleWith = p->oclIsKindOf(self.oclType) and self.type.conformsTo(p.oclAsType(TypedElement).type)`

Semantics

The semantics is as in the general case. However, two aspects are worth commenting on. The first is to note that a value specification may be an expression with substructure (i.e., an instance of the `Expression` class), in which case a template parameter may expose a subexpression, not necessarily the whole expression itself. An example of this is given in

Figure 17.21 where the parametered element with label 'n' appears within the expression 'n+1.' Secondly, to note that by extending NamedElement to optionally own a name expression, strings that are part of these named expressions may be parametered.

Notation

Where a parametered ValueSpecification is used within an expression, the name of the parameter is used instead of any symbol (in case of an Expression) or value (in case of a Literal) would otherwise appear.

18 Profiles

18.1 Overview

The Profiles package contains mechanisms that allow metaclasses from existing metamodels to be extended to adapt them for different purposes. This includes the ability to tailor the UML metamodel for different platforms (such as J2EE or .NET) or domains (such as real-time or business process modeling). The profiles mechanism is consistent with the OMG Meta Object Facility (MOF).

18.1.1 Positioning profiles versus metamodels, MOF and UML

The infrastructure specification is reused at several meta-levels in various OMG specifications that deal with modeling. For example, MOF uses it to provide the ability to model metamodels, whereas the UML superstructure uses it to model the UML model. This chapter deals with use cases comparable to the MOF at the meta-meta-level, which is one level higher than the rest of the superstructure specification. Thus, in this chapter, when we mention “Class,” in most cases we are dealing with the meta-metaclass “Class” (used to define every meta class in the UML superstructure specification (Activity, Class, State, Use Case, etc.)).

18.1.2 Profiles History and design requirements

The Profile mechanism has been specifically defined for providing a lightweight extension mechanism to the UML standard. In UML 1.1, stereotypes and tagged values were used as string-based extensions that could be attached to UML model elements in a flexible way. In subsequent revisions of UML, the notion of a Profile was defined in order to provide more structure and precision to the definition of Stereotypes and Tagged values. The UML2.0 infrastructure and superstructure specifications have carried this further, by defining it as a specific meta-modeling technique. Stereotypes are specific metaclasses, tagged values are standard metaattributes, and profiles are specific kinds of packages.

The following requirements have driven the definition of profile semantics from inception:

1. A profile must provide mechanisms for specializing a reference metamodel (such as a set of UML packages) in such a way that the specialized semantics do not contradict the semantics of the reference metamodel. That is, profile constraints may typically define well-formedness rules that are more constraining (but consistent with) those specified by the reference metamodel.
2. It must be possible to interchange profiles between tools, together with models to which they have been applied, by using the UML XMI interchange mechanisms. A profile must therefore be defined as an interchangeable UML model. In addition to exchanging profiles together with models between tools, profile application should also be definable “by reference” (e.g., “import by name”); that is, a profile does not need to be interchanged if it is already present in the importing tool.
3. A profile must be able to reference domain-specific UML libraries where certain model elements are pre-defined.
4. It must be possible to specify which profiles are being applied to a given Package (or any specializations of that concept). This is particularly useful during model interchange so that an importing environment can interpret a model correctly.
5. It should be possible to define a UML extension that combines profiles and model libraries (including template libraries) into a single logical unit. However, within such a unit, for definitional clarity and for ease of interchange (e.g., ‘reference by name’), it should still be possible to keep the libraries and the profiles distinct from each other.

6. A profile should be able to specialize the semantics of standard UML metamodel elements. For example, in a model with the profile “Java model,” generalization of classes should be able to be restricted to single inheritance without having to explicitly assign a stereotype «Java class» to each and every class instance.
7. A notational convention for graphical stereotype definitions as part of a profile should be provided.
8. In order to satisfy requirement [1] above, UML Profiles should form a metamodel extension mechanism that imposes certain restrictions on how the UML metamodel can be modified. The reference metamodel is considered as a “read only” model, that is extended without changes by profiles. It is therefore forbidden to insert new metaclasses in the UML metaclass hierarchy (i.e., new super-classes for standard UML metaclasses) or to modify the standard UML metaclass definitions (e.g., by adding meta-associations). Such restrictions do not apply in a MOF context where in principle any metamodel can be reworked in any direction.
9. The vast majority of UML case tools should be able to implement Profiles. The design of UML profiles should therefore not constrain these tools to have an internal implementation based on a meta-metamodel/metamodel architecture.
10. Profiles can be dynamically applied to or retracted from a model. It is possible on an existing model to apply new profiles, or to change the set of applied profiles.
11. Profiles can be dynamically combined. Frequently, several profiles will be applied at the same time on the same model. This profile combination may not be foreseen at profile definition time.
12. Models can be exchanged regardless of the profiles known by the destination target. The destination of the exchange of a model extended by a profile may not know the profile, and is not required to interpret a specific profile description. The destination environment interprets extensions only if it possesses the required profiles.

Extensibility

The profiles mechanism is not a first-class extension mechanism (i.e., it does not allow for modifying existing metamodels). Rather, the intention of profiles is to give a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method. Each such adaptation is grouped in a profile. It is not possible to take away any of the constraints that apply to a metamodel such as UML using a profile, but it is possible to add new constraints that are specific to the profile. The only other restrictions are those inherent in the profiles mechanism; there is nothing else that is intended to limit the way in which a metamodel is customized.

First-class extensibility is handled through MOF, where there are no restrictions on what you are allowed to do with a metamodel: you can add and remove metaclasses and relationships as you find necessary. Of course, it is then possible to impose methodology restrictions that you are not allowed to modify existing metamodels, but only extend them. In this case, the mechanisms for first-class extensibility and profiles start coalescing.

There are several reasons why you may want to customize a metamodel:

- Give a terminology that is adapted to a particular platform or domain (such as capturing EJB terminology like home interfaces, enterprise java beans, and archives).
- Give a syntax for constructs that do not have a notation (such as in the case of actions).
- Give a different notation for already existing symbols (such as being able to use a picture of a computer instead of the ordinary node symbol to represent a computer in a network).
- Add semantics that is left unspecified in the metamodel (such as how to deal with priority when receiving signals in a statemachine).

- Add semantics that does not exist in the metamodel (such as defining a timer, clock, or continuous time).
- Add constraints that restrict the way you may use the metamodel and its constructs (such as disallowing actions from being able to execute in parallel within a single transition).
- Add information that can be used when transforming a model to another model or code (such as defining mapping rules between a model and Java code).

Profiles and Metamodels

There is no simple answer for when you should create a new metamodel and when you instead should create a new profile.

18.2 Abstract syntax

Package structure

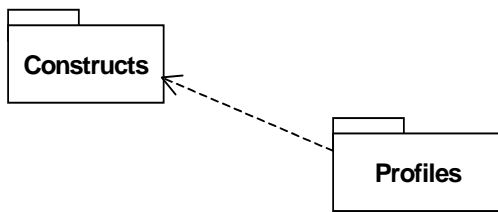


Figure 18.1 - Dependencies between packages described in this chapter

The classes of the Profiles package are depicted in Figure 18.2, and subsequently specified textually.

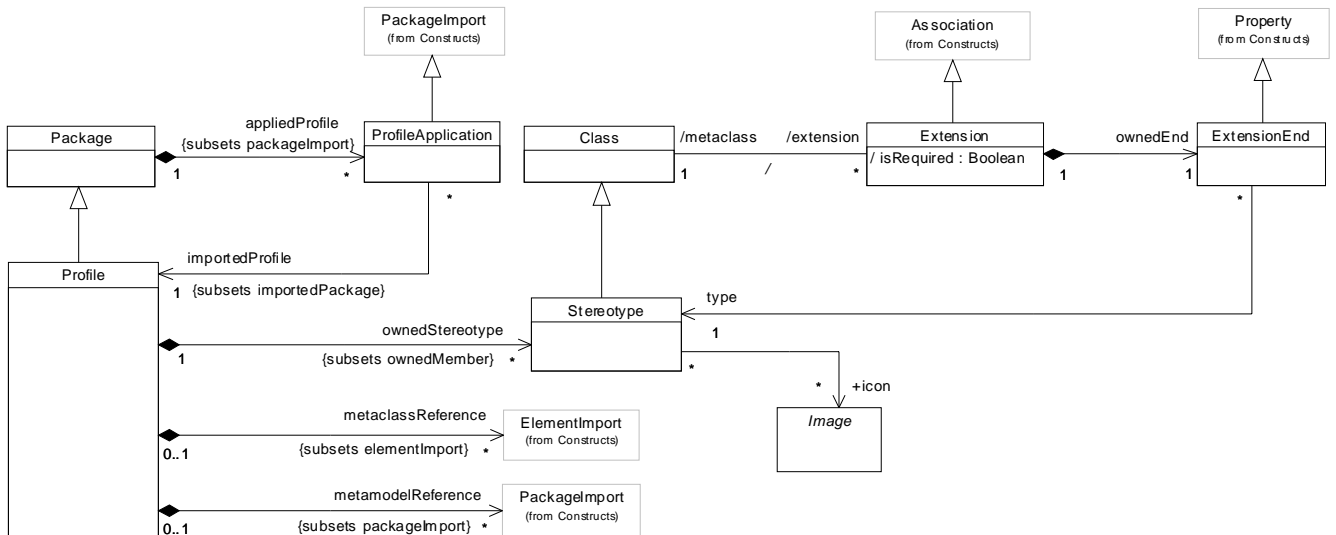


Figure 18.2 - The elements defined in the Profiles package

18.3 Class descriptions

18.3.1 Class (from Profiles)

Generalizations

- InfrastructureLibrary::Constructs::Class (*merge increment*)

Description

Class has derived association that indicates how it may be extended through one or more stereotypes.

Stereotype is the only kind of metaclass that cannot be extended by stereotypes.

Attributes

No additional attributes

Associations

- / extension: Extension [*] References the Extensions that specify additional properties of the metaclass. The property is derived from the extensions whose memberEnds are typed by the Class.

Constraints

No additional constraints

Semantics

No additional semantics

Notation

No additional notation

Presentation Options

A Class that is extended by a Stereotype may be extended by the optional stereotype «metaclass» (see Annex C., “Standard Stereotypes”) shown above or before its name.

Examples

In Figure 18.3, an example is given where it is made explicit that the extended class *Interface* is in fact a metaclass (from a reference metamodel).



Figure 18.3 - Showing that the extended class is a metaclass

Changes from previous UML

A link typed by UML 1.4 `ModelElement::stereotype` is mapped to a link that is typed by `Class::extension`.

18.3.2 Extension (from Profiles)

An extension is used to indicate that the properties of a metaclass are extended through a stereotype, and gives the ability to flexibly add (and later remove) stereotypes to classes.

Generalizations

- `InfrastructureLibrary::Constructs::Association`

Description

Extension is a kind of Association. One end of the Extension is an ordinary Property and the other end is an `ExtensionEnd`. The former ties the Extension to a Class, while the latter ties the Extension to a Stereotype that extends the Class.

Attributes

- `/isRequired: Boolean` Indicates whether an instance of the extending stereotype must be created when an instance of the extended class is created. The attribute value is derived from the multiplicity of *Extension::ownedEnd*; a multiplicity of 1 means that `isRequired` is *true*, but otherwise it is *false*. Since the default multiplicity of an `ExtensionEnd` is 0..1, the default value of `isRequired` is *false*.

Associations

- `ownedEnd: ExtensionEnd [1]` References the end of the extension that is typed by a Stereotype. Redefines *Association::ownedEnd*.
- `/metaclass: Class [1]` References the Class that is extended through an Extension. The property is derived from the type of the `memberEnd` that is not the `ownedEnd`.

Constraints

- [1] The non-owned end of an Extension is typed by a Class.
`metaclassEnd()->notEmpty() and metaclass()->oclIsKindOf(Class)`
- [2] An Extension is binary (i.e., it has only two `memberEnds`).
`memberEnd->size() = 2`

Additional Operations

- [1] The query `metaclassEnd()` returns the Property that is typed by a metaclass (as opposed to a stereotype).
`Extension::metaclassEnd(): Property;`
`metaclassEnd = memberEnd->reject(ownedEnd)`
- [2] The query `metaclass()` returns the metaclass that is being extended (as opposed to the extending stereotype).
`Extension::metaclass(): Class;`
`metaclass = metaclassEnd().type`
- [3] The query `isRequired()` is true if the owned end has a multiplicity with the lower bound of 1.
`Extension::isRequired(): Boolean;`

isRequired = (ownedEnd->lowerBound() = 1)

Semantics

A required extension means that an instance of a stereotype must always be linked to an instance of the extended metaclass. The instance of the stereotype is typically deleted only when either the instance of the extended metaclass is deleted, or when the profile defining the stereotype is removed from the applied profiles of the package. The model is not well formed if an instance of the stereotype is not present when isRequired is true.

A non-required extension means that an instance of a stereotype can be linked to an instance of an extended metaclass at will, and also later deleted at will; however, there is no requirement that each instance of a metaclass be extended. An instance of a stereotype is further deleted when either the instance of the extended metaclass is deleted, or when the profile defining the stereotype is removed from the applied profiles of the package.

The equivalence to a MOF construction is shown in Figure 18.4. This figure illustrates the case shown in Figure 18.6, where the “Home” stereotype extends the “Interface” metaclass. The MOF construct equivalent to an extension is an aggregation from the extended metaclass to the extension stereotype, navigable from the extension stereotype to the extended metaclass. When the extension is required, then the cardinality on the extension stereotype is “1.” The role names are provided using the following rule: The name of the role of the extended metaclass is:

‘base\$’ extendedMetaclassName

The name of the role of the extension stereotype is:

‘extension\$’ stereotypeName

Constraints are frequently added to stereotypes. The role names will be used for expressing OCL navigations. For example, the following OCL expression states that a Home interface shall not have attributes:

self.baseInterface.ownedAttributes->size() = 0

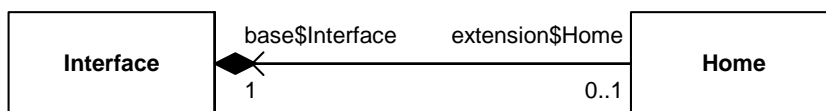


Figure 18.4 - MOF model equivalent to extending “interface” by the “Home” stereotype

Notation

The notation for an Extension is an arrow pointing from a Stereotype to the extended Class, where the arrowhead is shown as a filled triangle. An Extension may have the same adornments as an ordinary association, but navigability arrows are never shown. If isRequired is true, the property {required} is shown near the ExtensionEnd.



Figure 18.5 - The notation for an Extension

Presentation Options

It is possible to use the multiplicities 0..1 or 1 on the ExtensionEnd as an alternative to the property {required}. Due to how isRequired is derived, the multiplicity 0..1 corresponds to isRequired being *false*.

Style Guidelines

Adornments of an Extension are typically elided.

Examples

In Figure 18.6, a simple example of using an extension is shown, where the stereotype *Home* extends the metaclass *Interface*.



Figure 18.6 - An example of using an Extension

An instance of the stereotype *Home* can be added to and deleted from an instance of the class *Interface* at will, which provides for a flexible approach of dynamically adding (and removing) information specific to a profile to a model.

In Figure 18.7, an instance of the stereotype *Bean* always needs to be linked to an instance of class *Component* since the Extension is defined to be required. (Since the stereotype *Bean* is abstract, this means that an instance of one of its concrete subclasses always has to be linked to an instance of class *Component*.) The model is not well formed unless such a stereotype is applied. This provides a way to express extensions that should always be present for all instances of the base metaclass depending on which profiles are applied.

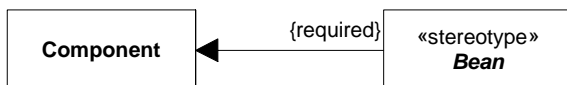


Figure 18.7 - An example of a required Extension

Changes from previous UML

Extension did not exist as a metaclass in UML 1.x.

Occurrences of Stereotype::baseClass of UML 1.4 is mapped to an instance of Extension, where the ownedEnd is typed by Stereotype and the other end is typed by the metaclass that is indicated by the baseClass.

18.3.3 ExtensionEnd (from Profiles)

An extension end is used to tie an extension to a stereotype when extending a metaclass.

Generalizations

- InfrastructureLibrary::Constructs::Property

Description

ExtensionEnd is a kind of Property that is always typed by a Stereotype.

An ExtensionEnd is never navigable. If it was navigable, it would be a property of the extended classifier. Since a profile is not allowed to change the referenced metamodel, it is not possible to add properties to the extended classifier. As a consequence, an ExtensionEnd can only be owned by an Extension.

The aggregation of an ExtensionEnd is always composite.

The default multiplicity of an ExtensionEnd is 0..1.

Attributes

No additional attributes

Associations

- type: Stereotype [1] References the type of the ExtensionEnd. Note that this association restricts the possible types of an ExtensionEnd to only be Stereotypes. Redefines *Property::type*.

Constraints

- [1] The multiplicity of ExtensionEnd is 0..1 or 1.
(self->lowerBound() = 0 **or** self->lowerBound() = 1) **and** self->upperBound() = 1
- [2] The aggregation of an ExtensionEnd is composite.
self.aggregation = #composite

Additional Operations

- [1] The query lowerBound() returns the lower bound of the multiplicity as an Integer. This is a redefinition of the default lower bound, which was 1.
ExtensionEnd::lowerBound() : [Integer];
lowerBound = **if** lowerValue->isEmpty() **then** 0 **else** lowerValue->IntegerValue() **endif**

Semantics

No additional semantics

Notation

No additional notation

Examples

See “Class (from Profiles)” on page 636.

Changes from previous UML

ExtensionEnd did not exist as a metaclass in UML 1.4. See “Class (from Profiles)” on page 636 for further details.

18.3.4 Image (from Profiles)

Physical definition of a graphical image.

Generalizations

None

Description

The Image class provides the necessary information to display an Image in a diagram. Icons are typically handled through the Image class.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Information such as physical localization or format is provided by the Image class. The Image class is abstract. It must be concretely defined within specifications dedicated to graphic handling (see, for example, the UML 2.0 Diagram Interchange OMG adopted specification).

18.3.5 Package (from Profiles)

Generalizations

- InfrastructureLibrary::Constructs::Package (*merge increment*)

Description

A package can have one or more ProfileApplications to indicate which profiles have been applied.

Because a profile is a package, it is possible to apply a profile not only to packages, but also to profiles.

Attributes

No additional attributes

Associations

- appliedProfile: ProfileApplication [*] References the ProfileApplications that indicate which profiles have been applied to the Package. Subsets *Package::packageImport*.

Constraints

No additional constraints

Semantics

The association “appliedProfile” between a package and a profile crosses metalevels: It links one element from a model (a kind of package) to an element of its metamodel and represents the set of profiles that define the extensions applicable to the package. Although this kind of situation is rare in the UML metamodel, this only shows that model and metamodel can coexist on the same space, and can have links between them.

Notation

No additional notation

Changes from previous UML

In UML 1.4, it was not possible to indicate which profiles were applied to a package.

18.3.6 Profile (from Profiles)

A profile defines limited extensions to a reference metamodel with the purpose of adapting the metamodel to a specific platform or domain.

Generalizations

- InfrastructureLibrary::Constructs::Package

Description

A Profile is a kind of Package that extends a reference metamodel. The primary extension construct is the Stereotype, which is defined as part of Profiles.

A profile introduces several constraints, or restrictions, on ordinary metamodeling through the use of the metaclasses defined in this package.

A profile is a restricted form of a metamodel that must always be related to a *reference metamodel*, such as UML, as described below. A profile cannot be used without its reference metamodel, and defines a limited capability to extend metaclasses of the reference metamodel. The extensions are defined as stereotypes that apply to existing metaclasses.

Attributes

No additional attributes

Associations

- metaclassReference: ElementImport [*] References a metaclass that may be extended. Subsets *Package::elementImport*
- metamodelReference: PackageImport [*] References a package containing (directly or indirectly) metaclasses that may be extended. Subsets *Package::packageImport*.
- ownedStereotype: Stereotype [*] References the Stereotypes that are owned by the Profile. Subsets *Package::ownedMember*

Constraints

- [1] An element imported as a metaclassReference is not specialized or generalized in a Profile.
self.metaclassReference.importedElement->

```
select(c | c.oclIsKindOf(Classifier) and
(c.generalization.namespace = self or
(c.specialization.namespace = self) )->isEmpty()
```

- [2] All elements imported either as metaclassReferences or through metamodelReferences are members of the same base reference metamodel.

```
self.metamodelReference.importedPackage.elementImport.importedElement.allOwningPackages()->
union(self.metaclassReference.importedElement.allOwningPackages() )->notEmpty()
```

Additional Operations

- [1] The query allOwningPackages() returns all the directly or indirectly owning packages.

```
NamedElement::allOwningPackages(): Set(Package)
allOwningPackages = self.namespace->select(p | p.oclIsKindOf(Package))->
union(p.allOwningPackages())
```

Semantics

A profile by definition extends a reference metamodel. It is not possible to define a standalone profile that does not directly or indirectly extend an existing metamodel. The profile mechanism may be used with any metamodel that is created from MOF, including UML and CWM.

A reference metamodel typically consists of metaclasses that are either imported or locally owned. All metaclasses that are extended by a profile have to be members of the same reference metamodel. A tool can make use of the information about which metaclasses are extended in different ways. For example to filter or hide elements when a profile is applied, or to provide specific tool bars that apply to certain profiles. However, elements of a package or model cannot be deleted simply through the application of a profile. Each profile thus provides a simple viewing mechanism.

As part of a profile, it is not possible to have an association between two stereotypes or between a stereotype and a metaclass. The effect of new (meta)associations within profiles can be achieved in limited ways either by:

1. Adding new constraints within a profile that specialize the usage of some associations of the reference metamodel, or
2. extending the Dependency metaclass by a stereotype and defining specific constraint on this stereotype.

As an illustration of the first approach, the examples in Figure 18.6 and Figure 18.7 could be extended by adding a “HomeRealization” stereotype that extends the “InterfaceRealization” UML metaclass. The “Bean” stereotype will introduce the constraint that the “interfaceRealization” association can only target “InterfaceRealization” elements extended by a “HomeRealization” stereotype and the “HomeRealization” stereotype will add the constraint that the “contract” association can only target interfaces extended by a “Home” stereotype. As an illustration of the second approach, one can define a stereotype “Sclass” extending Class and a stereotype “Sstate” extending State. In order to specify the default state of an “Sclass,” a “DefaultState” stereotype extending “Dependency” can be defined, with the constraints that a DefaultState Dependency can only exist between an Sclass and an Sstate.

The most direct implementation of the Profile mechanism that a tool can provide is by having a metamodel based implementation, similar to the Profile metamodel. However, this is not a requirement of the current standard, which requires only the support of the specified notions, and the standard XMI based interchange capacities. The profile mechanism has been designed to be implementable by tools that do not have a metamodel-based implementation. Practically any mechanism used to attach new values to model elements can serve as a valid profile implementation. As an example, the UML1.4 profile metamodel could be the basis for implementing a UML2.0-compliant profile tool.

Using XMI to exchange Profiles

As shown in Figure 18.4 on page 638 (Extension : Semantics), there is a direct correspondence between a profile definition and a MOF metamodel. XMI can therefore be directly applied to exchange Profiles. We will take the example Figure 18.6 on page 639 (Extension : Notation) of a profile that we will call “HomeExample” to illustrate how a profile can be exchanged. We will see that a profile can be exchanged as a model, as an XMI schema definition, and that models extended by the profile can also interchange their definition and extension.

Figure 18.6 on page 639 shows a “Home” stereotype extending the “Interface” UML2 metaclass. Figure 18.4 on page 638 illustrates the MOF correspondence for that example, basically by introducing an association from the “Home” MOF class to the “Interface” MOF class. For illustration purposes, we add a property (tagged value definition in UML1.4) called “magic:String” to the “Home” stereotype.

The first serialization below shows how the model Figure 18.5 on page 638 (instance of the profile and UML2 metamodel) can be exchanged.

```
<?xml version="1.0" encoding="UTF-8"?>
<XMI xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:mof="http://schema.omg.org/spec/MOF/2.0"
xmlns:uml="http://schema.omg.org/spec/UML/2.0">
  <uml:Profile xmi:id="id1" name="HomeExample">

    <ownedStereotype xsi:type="uml:Stereotype" xmi:id="id2" name="Home">
      <ownedAttribute xmi:id="id3" name="base_Interface" association="id5" type="id9"/>
      <ownedAttribute xmi:id="id4" name="magic" type="id10"/>
    </ownedStereotype>

    <ownedMember xsi:type="uml:Extension" xmi:id="id5" memberEnd="id3 id6">
      <ownedEnd xsi:type="uml:ExtensionEnd" name="extensionHome" xmi:id="id6"
        type="id2" association="id5" isComposite="true" lower="0"/>
    </ownedMember>

    <!--There should be metaclass reference between the profile and the extended metaclass -->
    <metaclassReference xmi:id="id8">
      <uml:elementImport xsi:type="uml:ElementImport" importedElement="id9"/>
    </metaclassReference>
  </uml:Profile>

  <mof:Class xmi:id="id9" href="http://schema.omg.org/spec/UML/2.0/uml.xml#Interface"/>
  <mof:PrimitiveType xmi:id="id10" href="http://schema.omg.org/spec/UML/2.0/uml.xml#String"/>

  <mof:Tag name="org.omg.xmi.nsURI"
    value="http://www.mycompany.com/schemas/HomeExample.xmi" element="id1"/>
  <mof:Tag name="org.omg.xmi.nsPrefix" value="HomeExample" element="id1"/>
  <mof:Tag name="org.omg.xmi.xmiName" value="base_Interface" element="id3"/>
</XMI>
```

We will now obtain an XMI definition from the «HomeExample» profile. That XMI description will itself define in XML how the models extended by the HomeExample will be exchanged in XMI. We can see here that an XMI schema separated from the standard UML2 schema can be obtained. This XMI definition is stored in a file called “HomeExample.xmi.”

```

<?xml version="1.0" encoding="UTF-8"?>

<xsd:schema targetNamespace =
"http://www.mycompany.com/schemas/HomeExample.xmi"
xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:uml="http://www.omg.org/spec/UML/2.0"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:import namespace="http://schema.omg.org/spec/XMI/2.1" schemaLocation="XMI.xsd"/>
  <xsd:import namespace="http://schema.omg.org/spec/UML/2.0" schemaLocation="UML20.xsd"/>

  <xsd:complexType name="Home">
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element name="base_Interface" type="xmi:Any"/>
      <xsd:element name="magic" type="xsd:string"/>
      <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="base_Interface" type="uml:Interface"
    <xsd:attribute name="magic" type="xsd:string" use="optional"/>
    use="optional"/>
  </xsd:complexType>

  <xsd:element name="Home" type="Home"/>
</xsd:schema>

```

Let us provide an example of an Interface extended by the Home stereotype.

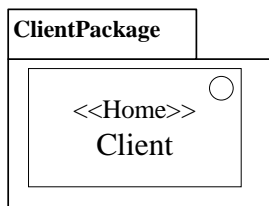


Figure 18.8 - Using the “HomeExample” profile to extend a model

Now the XMI code below shows how this model extended by the profile is serialized. A tool importing that XMI file can filter out the elements related to the “HomeExample” schema, if the tool does not have this schema (profile) definition.

```

<?xml version="1.0" encoding="UTF-8"?>
<XMI xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:uml="http://schema.omg.org/spec/UML/2.0"
xmlns:HomeExample="http://www.mycompany.com/schemas/HomeExample.xmi">

  <uml:Package xmi:id="id1" name="ClientPackage">
    <ownedMember xsi:type="uml:Interface" xmi:id="id2" name="Client"/>
    <packageImport xsi:type="uml:ProfileApplication" xmi:id="id3">

```

```

    <importedProfile href="HomeExample.xmi#idl" />
  </packageImport>
</uml:Package>

<!-- Stereotypes -->
<HomeExample:Home base_Interface="id2" magic="1234" />

</XMI>

```

Notation

A Profile uses the same notation as a Package, with the addition that the keyword «profile» is shown before or above the name of the Package. *Profile::metaclassReference* and *Profile::metamodelReference* uses the same notation as *Package::elementImport* and *Package::packageImport*, respectively.

Examples

In Figure 18.9, a simple example of an EJB profile is shown.

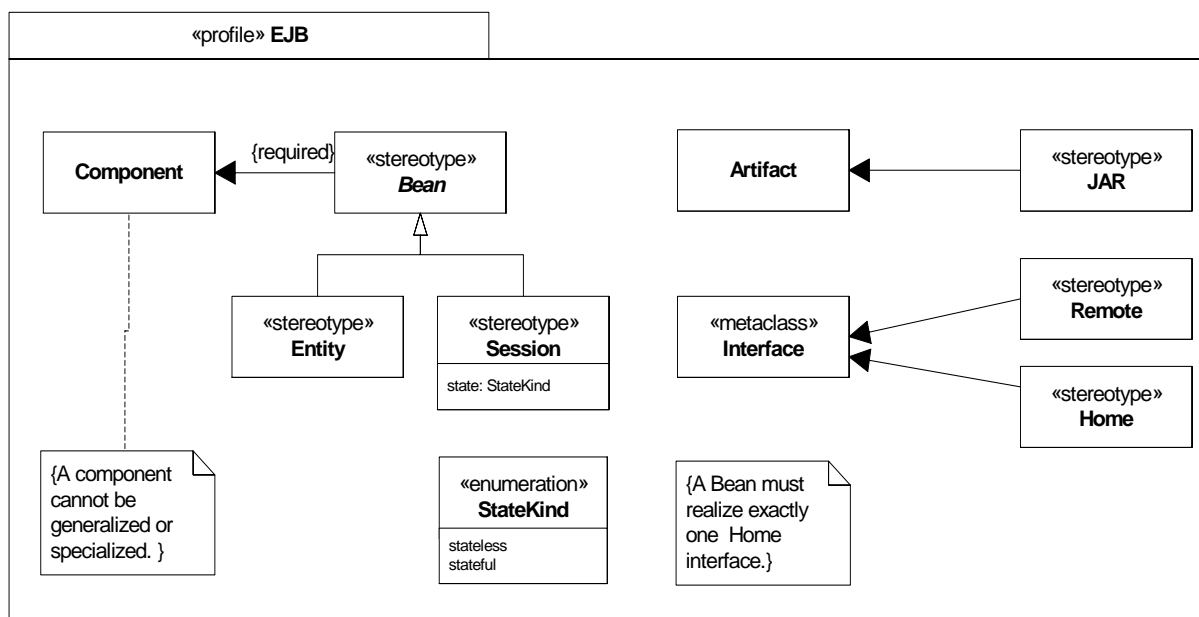


Figure 18.9 - Defining a simple EJB profile

The profile states that the abstract stereotype *Bean* is required to be applied to metaclass *Component*, which means that an instance of either of the concrete subclasses *Entity* and *Session* of *Bean* must be linked to each instance of *Component*. The constraints that are part of the profile are evaluated when the profile has been applied to a package, and need to be satisfied in order for the model to be well formed.

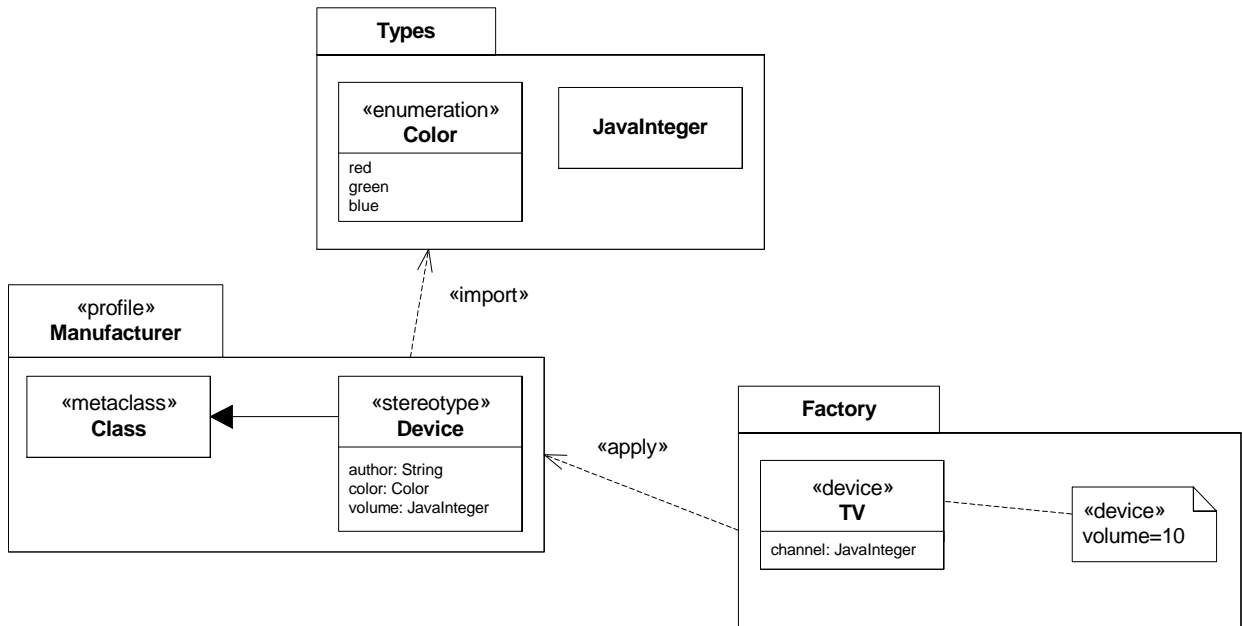


Figure 18.10 - Importing a package from a profile

In Figure 18.10, the package *Types* is imported from the profile *Manufacturer*. The data type *Color* is then used as the type of one of the properties of the stereotype *Device*, just like the predefined type *String* is also used. Note that the class *JavaInteger* may also be used as the type of a property.

If the profile *Manufacturer* is later applied to a package, then the types from *Types* are also available for use in the package to which the profile is applied (since profile application is a kind of import). This means that for example the class *JavaInteger* can be used as both a metaproperty (as part of the stereotype *Device*) and an ordinary property (as part of the class *TV*). Note how the metaproperty is given a value when the stereotype *Device* is applied to the class *TV*.

18.3.7 ProfileApplication (from Profiles)

A profile application is used to show which profiles have been applied to a package.

Generalizations

- InfrastructureLibrary::Constructs::PackageImport

Description

ProfileApplication is a kind of PackageImport that adds the capability to state that a Profile is applied to a Package.

Attributes

No additional attributes

Associations

- importedProfile: Profile [1] References the Profiles that is applied to a Package through this ProfileApplication. Subsets *PackageImport::importedPackage*

Constraints

No additional constraints

Semantics

One or more profiles may be applied at will to a package that is created from the same metamodel that is extended by the profile. Applying a profile means that it is allowed, but not necessarily required, to apply the stereotypes that are defined as part of the profile. It is possible to apply multiple profiles to a package as long as they do not have conflicting constraints. If a profile that is being applied depends on other profiles, then those profiles must be applied first.

When a profile is applied, instances of the appropriate stereotypes should be created for those elements that are instances of metaclasses with required extensions. The model is not well formed without these instances.

Once a profile has been applied to a package, it is allowed to remove the applied profile at will. Removing a profile implies that all elements that are instances of elements defined in a profile are deleted. A profile that has been applied cannot be removed unless other applied profiles that depend on it are first removed.

Note – The removal of an applied profile leaves the instances of elements from the referenced metamodel intact. It is only the instances of the elements from the profile that are deleted. This means that for example a profiled UML model can always be interchanged with another tool that does not support the profile and be interpreted as a pure UML model.

Notation

The names of Profiles are shown using a dashed arrow with an open arrowhead from the package to the applied profile. The keyword «apply» is shown near the arrow.

If multiple applied profiles have stereotypes with the same name, it may be necessary to qualify the name of the stereotype (with the profile name).

Examples

Given the profiles *Java* and *EJB*, Figure 18.11 shows how these have been applied to the package *WebShopping*.

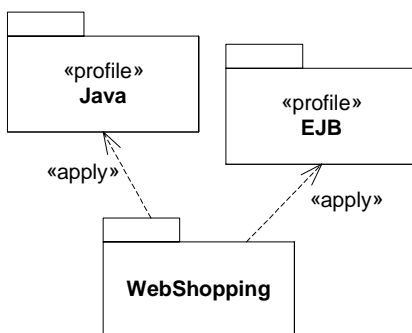


Figure 18.11 - Profiles applied to a package

18.3.8 Stereotype (from Profiles)

A stereotype defines how an existing metaclass may be extended, and enables the use of platform or domain specific terminology or notation in place of, or in addition to, the ones used for the extended metaclass.

Generalizations

- InfrastructureLibrary::Constructs::Class

Description

Stereotype is a kind of Class that extends Classes through Extensions.

Just like a class, a stereotype may have properties, which may be referred to as tag definitions. When a stereotype is applied to a model element, the values of the properties may be referred to as tagged values.

Attributes

No additional attributes

Associations

- icon : Image [*] Stereotype can change the graphical appearance of the extended model element by using attached icons. When this association is not null, it references the location of the icon content to be displayed within diagrams presenting the extended model elements.

Constraints

- [1] A Stereotype may only generalize or specialize another Stereotype.
 generalization.general->forAll(e | e.ocllsKindOf(Stereotype)) **and**
 generalization.specific->forAll(e | e.ocllsKindOf(Stereotype))
- [2] Stereotype names should not clash with keyword names for the extended model element.

Semantics

A stereotype is a limited kind of metaclass that cannot be used by itself, but must always be used in conjunction with one of the metaclasses it extends. Each stereotype may extend one or more classes through extensions as part of a profile. Similarly, a class may be extended by one or more stereotypes.

An instance “S” of Stereotype is a kind of (meta) class. Relating it to a metaclass “C” from the reference metamodel (typically UML) using an “Extension” (which is a specific kind of association), signifies that model elements of type C can be extended by an instance of “S” (see example Figure 18.10). At the model level (such as in Figure 18.14) instances of “S” are related to “C” model elements (instances of “C”) by links (occurrences of the association/extension from “S” to “C”).

Any model element from the reference metamodel (any UML model element) can be extended by a stereotype. For example in UML, States, Transitions, Activities, Use cases, Components, Attributes, Dependencies, etc. can all be extended with stereotypes.

Notation

A Stereotype uses the same notation as a Class, with the addition that the keyword «stereotype» is shown before or above the name of the Class.

When a stereotype is applied to a model element (an instance of a stereotype is linked to an instance of a metaclass), the name of the stereotype is shown within a pair of guillemets above or before the name of the model element. If multiple stereotypes are applied, the names of the applied stereotypes are shown as a comma-separated list with a pair of guillemets. When the extended model element has a keyword, then the stereotype name will be displayed close to the keyword, within separate guillemets (example: «interface» «Clock»).

Presentation Options

If multiple stereotypes are applied to an element, it is possible to show this by enclosing each stereotype name within a pair of guillemets and listing them after each other. A tool can choose whether it will display stereotypes or not. In particular, some tools can choose not to display “required stereotypes,” but to display only their attributes (tagged values) if any.

The values of a stereotype that has been applied to a model element can be shown as part of a comment symbol tied to the model element. The values from a specific stereotype are optionally preceded with the name of the applied stereotype within a pair of guillemets, which is useful if values of more than one applied stereotype should be shown.

If the extension end is given a name, this name can be used in lieu of the stereotype name within the pair of guillemets when the stereotype is applied to a model element.

It is possible to attach a specific notation to a stereotype that can be used in lieu of the notation of a model element to which the stereotype is applied.

Icon presentation

When a stereotype includes the definition of an icon, this icon can be graphically attached to the model elements extended by the stereotype. Every model element that has a graphical presentation can have an attached icon. When model elements are graphically expressed as:

- Boxes (see Figure 18.13 on page 651): the box can be replaced by the icon, and the name of the model element appears below the icon. This presentation option can be used only when a model element is extended by one single stereotype and when properties of the model element (i.e., attributes, operations of a class) are not presented. As another option, the icon can be presented in a reduced shape, inside and on top of the box representing the model element. When several stereotypes are applied, several icons can be presented within the box.
- Links: the icon can be placed close to the link.
- Textual notation: the icon can be presented to the left of the textual notation.

Several icons can be attached to a stereotype. The interpretation of the different attached icons in that case is a semantic variation point. Some tools may use different images for the icon replacing the box, for the reduced icon inside the box, for icons within explorers, etc. Depending on the image format, other tools may choose to display one single icon into different sizes.

Some model elements are already using an icon for their default presentation. A typical example of this is the Actor model element, which uses the “stickman” icon. In that case, when a model element is extended by a stereotype with an icon, the stereotype’s icon replaces the default presentation icon within diagrams.

Style Guidelines

The first letter of an applied stereotype should not be capitalized. The values of an applied stereotype are normally not shown.

Examples

In Figure 18.12, a simple stereotype *Clock* is defined to be applicable at will (dynamically) to instances of the metaclass *Class*.



Figure 18.12 - Defining a stereotype

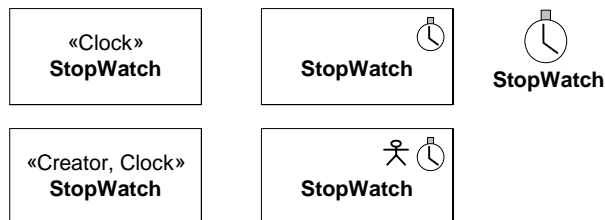


Figure 18.13 - Presentation options for an extended class

In Figure 18.14, an instance specification of the example in Figure 18.12 is shown. Note that the extension must be composite, and that the derived *isRequired* attribute in this case is false. Figure 18.14 shows the repository schema of the stereotype “clock” defined in Figure 18.12. In this schema, the extended instance (*:Class*; “name = Class”) is defined in the UML2.0 (reference metamodel) repository. In a UML modeling tool these extended instances referring to the UML2.0 standard would typically be in a “read only” form, or presented as proxies to the metaclass being extended.

(It is therefore still at the same meta-level as UML, and does not show the instance model of a model extended by the stereotype. An example of this is provided in Figure 18.16 and Figure 18.17.) The Semantics sub section of the Extension concept explains the MOF equivalent, and how constraints can be attached to stereotypes.

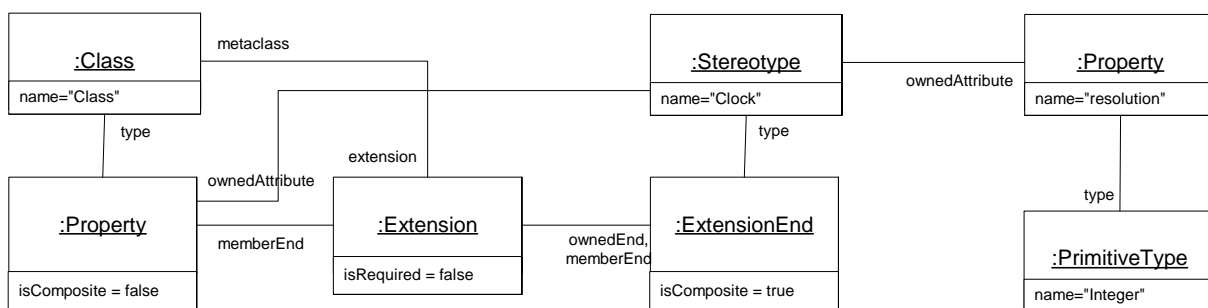


Figure 18.14 - An instance specification when defining a stereotype

Figure 18.15 shows how the same stereotype *Clock* can extend either the metaclass *Component* or the metaclass *Class*. It also shows how different stereotypes can extend the same metaclass.

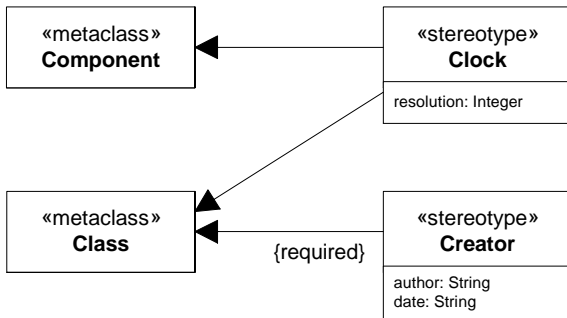


Figure 18.15 - Defining multiple stereotypes on multiple stereotypes

Figure 18.16 shows how the stereotype *Clock*, as defined in Figure 18.15, is applied to a class called *StopWatch*.



Figure 18.16 - Using a stereotype

Figure 18.17 shows an example instance model for when the stereotype *Clock* is applied to a class called *StopWatch*. The extension between the stereotype and the metaclass results in a link between the instance of stereotype *Clock* and the (user-defined) class *StopWatch*.

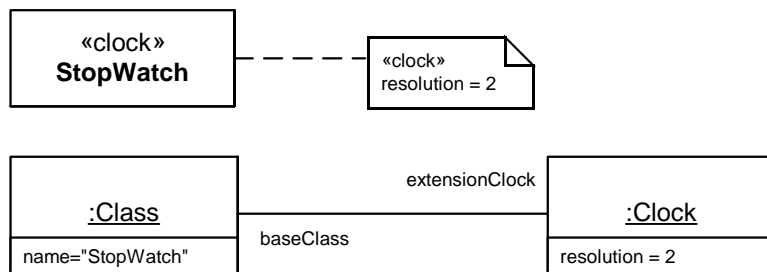


Figure 18.17 - Showing values of stereotypes and a simple instance specification

Next, two stereotypes, *Clock* and *Creator*, are applied to the same model element, as shown in Figure 18.18. Note that the attribute values of each of the applied stereotypes can be shown in a comment symbol attached to the model element.

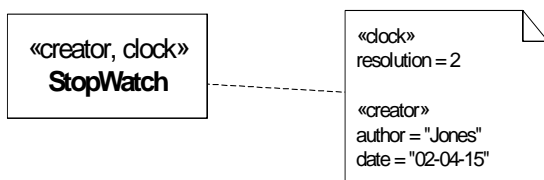


Figure 18.18 - Using stereotypes and showing values

Changes from previous UML

In UML 1.3, tagged values could extend a model element without requiring the presence of a stereotype. In UML 1.4, this capability, although still supported, was deprecated, to be used only for backward compatibility reasons. In UML 2.0, a tagged value can only be represented as an attribute defined on a stereotype. Therefore, a model element must be extended by a stereotype in order to be extended by tagged values. However, the “required” extension mechanism can, in effect, provide the 1.3 capability, since a tool can in those circumstances automatically define a stereotype to which “unattached” attributes (tagged values) would be attached.

18.4 Diagrams

Structure diagrams

This section outlines the graphic elements that may be shown in structure diagrams, and provides cross references where detailed information about the semantics and concrete notation for each element can be found. It also furnishes examples that illustrate how the graphic elements can be assembled into diagrams.

Graphical nodes

The graphic nodes that can be included in structure diagrams are shown in Table 18.1.

Table 18.1 - Graphic nodes included in structure diagrams

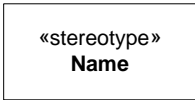
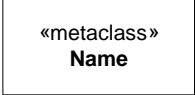
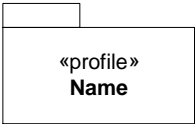
Node Type	Notation	Reference
Stereotype		See “Stereotype (from Profiles)” on page 649.


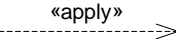
Table 18.1 - Graphic nodes included in structure diagrams

Node Type	Notation	Reference
Metaclass		See “Class (from Profiles)” on page 636.
Profile		See “Profile (from Profiles)” on page 642.

Graphical paths

The graphic paths that can be included in structure diagrams are shown in Table 18.2.

Table 18.2 - Graphic paths included in structure diagrams

Node Type	Notation	Reference
Extension		See “Class (from Profiles)” on page 636.
ProfileApplication		See “ProfileApplication (from Profiles)” on page 647.

Part IV - Annexes

This section contains the annexes for this specification.

- Annex A - Diagrams
- Annex B - UML Keywords
- Annex C - Standard Stereotypes
- Annex D - Component Profile Examples
- Annex E - Tabular Notations
- Annex F - Classifiers Taxonomy
- Annex G - XMI Serialization and Schema

Annex A (normative)

Diagrams

This annex describes the general properties of UML diagrams and how they relate to a UML (repository) model and to elements of this. It also introduces the different diagram types of UML.

A UML model consists of elements such as packages, classes, and associations. The corresponding UML diagrams are graphical representations of parts of the UML model. UML diagrams contain graphical elements (nodes connected by paths) that represent elements in the UML model. As an example, two associated classes defined in a package will, in a diagram for the package, be represented by two class symbols and an association path connecting these two class symbols.

Each diagram has a *contents area*. As an option, it may have a *frame* and a *heading* as shown in Figure A.1.

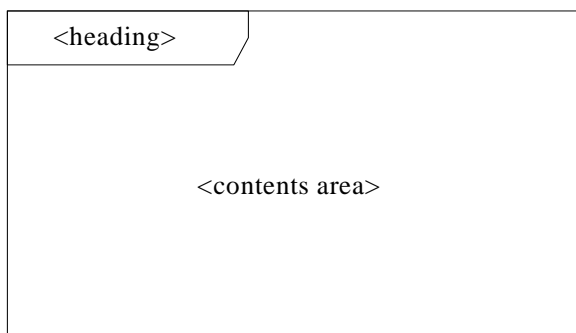


Figure A.1

The frame is a rectangle. The frame is primarily used in cases where the diagrammed element has graphical border elements, like ports for classes and components (in connection with composite structures), and entry/exit points on statemachines. In cases where not needed, the frame may be omitted and implied by the border of the diagram area provided by a tool. In case the frame is omitted, the heading is also omitted.

The diagram contents area contains the graphical symbols; the primary graphical symbols define the type of the diagram (e.g., a class diagram is a diagram where the primary symbols in the contents area are class symbols).

The heading is a string contained in a name tag (rectangle with cutoff corner) in the upper leftmost corner of the rectangle, with the following syntax:

`[<kind>]<name>[<parameters>]`

The heading of a diagram represents the kind, name, and parameters of the namespace enclosing or the model element owning elements that are represented by symbols in the contents area. Most elements of a diagram contents area represent model elements that are defined in the namespace or are owned by another model element.

As an example, Figure A.2 is a class diagram of a package P: classes C1 and C2 are defined in the namespace of the package P.

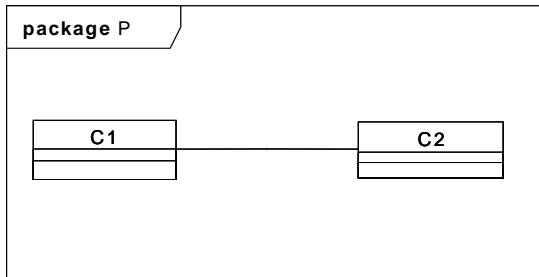
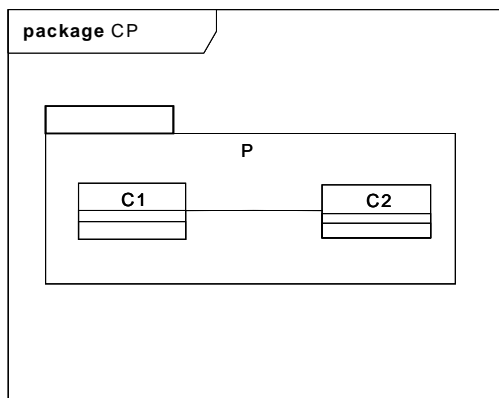


Figure A.2 - Class diagram of package P

Figure A.3 illustrates that a package symbol for package P (in some containing package CP) may show the same contents as the class diagram for the package. i) is a diagram for package CP with graphical symbols representing the fact that the CP package contains a package P. ii) is a class diagram for this package P. Note that the package symbol in i) does not have to contain the class symbols and the association symbol; for more realistic models, the package symbols will typically just have the package names, while the class diagrams for the packages will have class symbols for the classes defined in the packages.

i) Package symbol (as part of a larger diagram)



ii) Class diagram for the same package

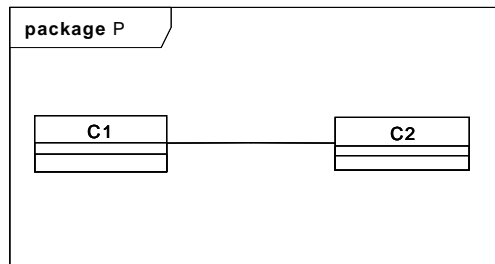
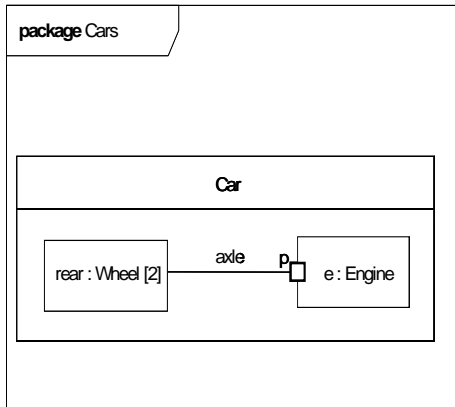


Figure A.3 - Two diagrams of packages

In Figure A.4 i) is a class diagram for the package Cars, with a class symbol representing the fact that the Cars package contains a class Car. ii) is a composite structure diagram for this class Car. The class symbol in i) does not have to contain the structure of the class in a compartment; for more realistic models, the class symbols will typically just have the class names, while the composite structure diagrams for the classes will have symbols for the composite structures.

i) Class symbol for class Car(as part of a larger diagram)



ii) Composite structure diagram for the same class Car

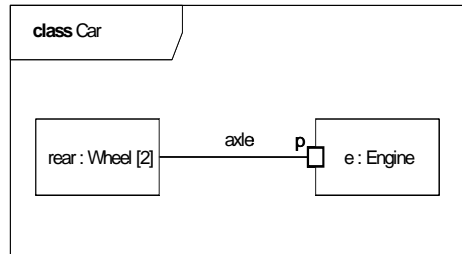


Figure A.4 - A class diagram and a composite structure diagram

UML diagrams may have the following kinds of frame names as part of the heading:

- activity
- class
- component
- interaction
- package
- state machine
- use case

In addition to the long form names for diagram heading types, the following abbreviated forms can also be used:

- **act** (for activity frames)
- **cmp** (for component frames)
- **sd** (for interaction frames)
- **pkg** (for package frames)
- **stm** (for state machine frames)
- **uc** (for use case frames)

As is shown in Figure A.5, there are two major kinds of diagram types: structure diagrams and behavior diagrams.

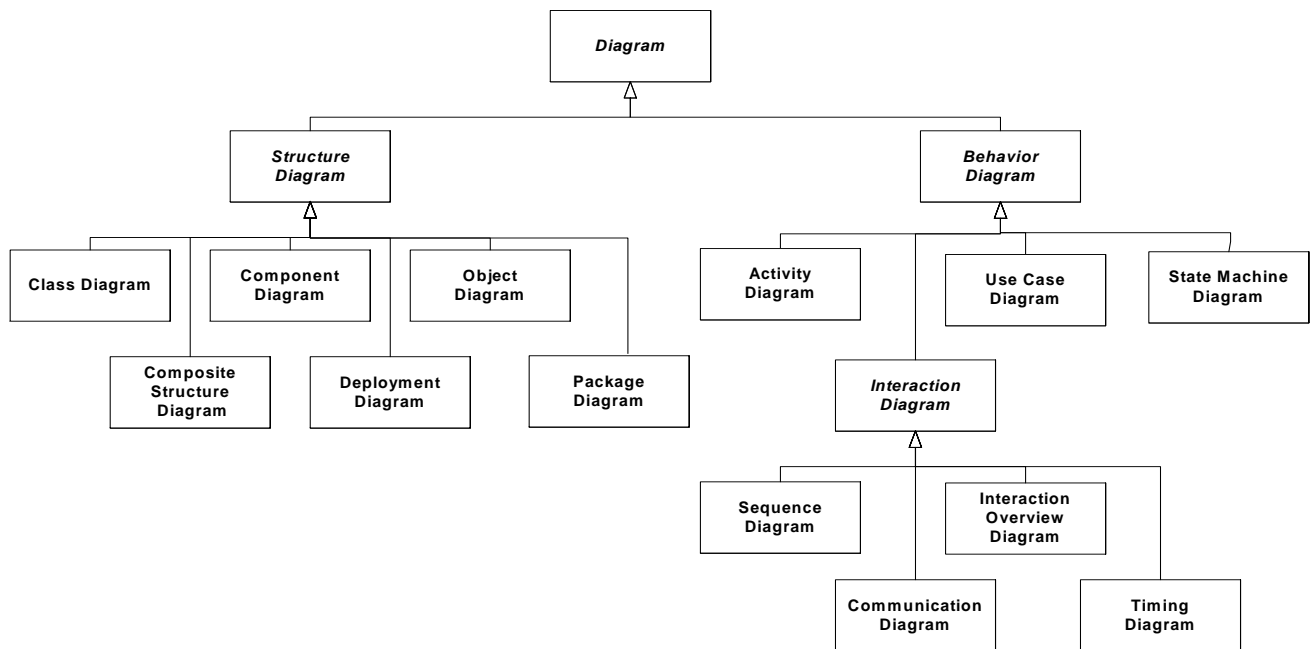


Figure A.5 - The taxonomy of structure and behavior diagram

Structure diagrams show the static structure of the objects in a system. That is, they depict those elements in a specification that are irrespective of time. The elements in a structure diagram represent the meaningful concepts of an application, and may include abstract, real-world and implementation concepts. For example, a structure diagram for an airline reservation system might include classifiers that represent seat assignment algorithms, tickets, and a credit authorization service. Structure diagrams do not show the details of dynamic behavior, which are illustrated by behavioral diagrams. However, they may show relationships to the behaviors of the classifiers exhibited in the structure diagrams.

Behavior diagrams show the dynamic behavior of the objects in a system, including their methods, collaborations, activities, and state histories. The dynamic behavior of a system can be described as a series of changes to the system over time. Behavior diagrams can be further classified into several other kinds as illustrated in Figure A.5.

Please note that this taxonomy provides a logical organization for the various major kinds of diagrams. However, it does not preclude mixing different kinds of diagram types, as one might do when one combines structural and behavioral elements (e.g., showing a state machine nested inside an internal structure). Consequently, the boundaries between the various kinds of diagram types are not strictly enforced.

The constructs contained in each of the thirteen UML diagrams is described in the Superstructure chapters as indicated below.

- Activity Diagram - “Activities” on page 285
- Class Diagram - “Classes” on page 21
- Communication Diagram - “Interactions” on page 443
- Component Diagram - “Components” on page 139
- Composite Structure Diagram - “Composite Structures” on page 157
- Deployment diagram - “Deployments” on page 189

- Interaction Overview Diagram - “Interactions” on page 443
- Object Diagram - “Classes” on page 21
- Package Diagram - “Classes” on page 21
- State Machine Diagram - “State Machines” on page 507
- Sequence Diagram - “Interactions” on page 443
- Timing Diagram - “Interactions” on page 443
- Use Case Diagram - “Use Cases” on page 569

Annex B (normative)

UML Keywords

UML keywords are reserved words that are an integral part of the UML notation and normally appear as text annotations attached to a UML graphic element or as part of a text line in a UML diagram. These words have special significance in the context in which they are defined and, therefore, cannot be used to name user-defined model elements where such naming would result in ambiguous interpretation of the model. For example, the keyword “trace” is a system-defined stereotype of Abstraction (see Annex C, “Standard Stereotypes”) and, therefore, cannot be used to define any user-defined stereotype.

In UML, keywords are used for four different purposes:

- To distinguish a particular UML concept (metaclass) from others sharing the same general graphical form. For instance, the «interface» keyword in the header box of a classifier rectangle is used to distinguish an Interface from other kinds of Classifiers.
- To distinguish a particular kind of relationship between UML concepts (meta-association) from other relationships sharing the same general graphical form. For example, dashed lines between elements are used for a number of different relationships, including Dependencies, relationships between UseCases and an extending UseCases, and so on.
- To specify the value of some modifier attached to a UML concept (meta-attribute value). Thus, the keyword «singleExecution» appearing within an Activity signifies that the “isSingleExecution” attribute of that Activity is true.
- To indicate a Standard Stereotype (see Annex C, “Standard Stereotypes”). For example, the «modelLibrary» keyword attached to a package identifies that the package contains a set of model elements intended to be shared by multiple models.

Keywords are always enclosed in guillemets («keyword»), which serve as visual cues to more readily distinguish when a keyword is being used. (Note that guillemets are a special kind of quotation marks and should not be confused with or replaced by duplicated “greater than” (>>) or “less than” (<<) symbols, except in situations where the available character set may not include guillemets.) In addition to identifying keywords, guillemets are also used to distinguish the usage of stereotypes defined in user profiles. This means that:

1. Not all words appearing between guillemets are necessarily keywords (i.e., reserved words), and
2. words appearing in guillemets do not necessarily represent stereotypes.

If multiple keywords and/or stereotype names apply to the same model element, they all appear between the same pair of guillemets, separated by commas:

“«” <label> [“,” <label>]* “»”

where:

<label> ::= <keyword> | <stereotype-label>

Keywords are context sensitive and, in a few cases, the same keyword is used for different purposes in different contexts. For instance, the «create» keyword can appear next to an operation name to indicate that it is a constructor operation, and it can also be used to label a Usage dependency between two Classes to indicate that one Class creates instances of the other. This means that it is possible in principle to use a keyword for a user-defined stereotype (provided that it is used in a context that does not conflict with the keyword context). However, such practices are discouraged since they are likely to lead to confusion.

The keywords currently defined as part of standard UML are specified in Table B.1, sorted in alphabetical order. The following is the interpretation of the individual columns in this table:

- *Keyword* provides the exact spelling of the keyword (without the guillemets).
- *Language Unit* identifies the language unit in which the keyword is defined (and, implicitly, the chapter in which the keyword is described).
- *Metamodel Element* specifies the element of the UML metamodel (either a metaclass or a metaclass feature) that the keyword denotes.
- *Semantics* gives a brief description of the semantics of the keyword (see further explanations below); more detailed explanations are provided in the Notation sections of the corresponding metaclass description. The following formats are used:
 - 1) If the entry contains the name of a UML metaclass, this indicates that the keyword is simply used to identify the corresponding metaclass.
 - 2) If the entry is a constraint (usually but not necessarily an OCL expression), it specifies a constraint that applies to metamodel elements that are tagged with that keyword.
 - 3) If the entry is in the form “standard stereotype:L<x>”, where <x> = 1, 2, or 3, it means that the keyword represents a stereotype that is defined at compliance level. In those cases, the more detailed description of the semantics can be found in Appendix C, “Standard Stereotypes.”
- *Notation Placement* indicates where the keyword appears (see further explanations below). The following conventions are used to specify the notation placement:
 - 1) “box header” means that the keyword appears in the name compartment of a classifier rectangle.
 - 2) “list-box header” means that the keyword is used as a header on a list box appearing as part of a classifier specification.
 - 3) “dashed-line label” means that the keyword is used as a label on some dashed line, such as a Dependency.
 - 4) “inline label” means that the keyword appears as part of a text line (usually at the front), such as an attribute definition.
 - 5) “between braces” means that the keyword appears between “curly” brackets (similar to the constraint notation) and is used to select the value of some property of a metaclass.
 - 6) “swimlane header” means that the keyword appears as the header of a swimlane in an activity diagram.

Table B.1 - UML Keywords

Keyword	Language Unit	Metamodel Element	Semantics	Notation Placement
abstraction	Classes	Abstraction	Abstraction	box header
access	Classes	PackageImport	(visibility <> #public)	dashed-line label
activity	Activities	Activity	Activity	box header
actor	Use Cases	Actor	Actor	box header
after	CommonBehaviors	TimeEvent	isRelative = true	Inline label
all	CommonBehaviors	AnyReceiveEvent	Any event	Inline label
apply	Profiles	ProfileApplication	Package::appliedProfile->collect(ap ap.importedProfile)	dashed-line label
artifact	Deployments	Artifact	Artifact	box header
artifacts	Deployments	Component		list-box header
at	CommonBehaviors	TimeEvent	isRelative = false	Inline label
attribute	Activities	ActivityPartition::represents	ActivityPartition::represents ->forAll(r r.oclIsKindOf(Property))	swimlane header
auxiliary	Classes	Classifier	standard stereotype:L1	box header
buildcomponent	Components	Component	standard stereotype:L3	box header
call	Classes	Usage	standard stereotype:L1	dashed-line label
centralBuffer	Activities	CentralBufferNode	CentralBufferNode	box header
class	Activities	ActivityPartition::represents	ActivityPartition::represents ->forAll(r r.oclIsKindOf(Class))	swimlane header
component	Components	Component	Component	box header
create	Classes	Usage	standard stereotype:L1	dashed-line label
create	Classes	BehavioralFeature	standard stereotype:L1 (constructor)	inline label on operation
create	CompositeStructures	Dependency	Applies only to dependencies between an InstanceSpec and Parameter of an operation, indicating a return parameter of an Operation that is a constructor.	dashed-line label
datastore	Activities	DataStoreNode	DataStoreNode	box header
datatype	Classes	DataType	DataType	box header
delegate	Components	Connector	Connector::kind = #delegation	connector label
deploy	Deployments	Connector	delegation connector	dashed-line label
deployment spec	Deployments	Deployment Specification	DeploymentSpecification	box header

Table B.1 - UML Keywords

Keyword	Language Unit	Metamodel Element	Semantics	Notation Placement
derive	Classes	Abstraction	standard stereotype:L1	dashed-line label
destroy	Classes	BehavioralFeature	standard stereotype:L1	inline label on operation
device	Deployments	Device	Device	box header
document	Deployments	Artifact	standard stereotype:L2	box header
element access	Classes	ElementImport	not (visibility = #public)	dashed-line label
element import	Classes	ElementImport	visibility = #public	dashed-line label
entity	Components	Component	standard stereotype:L2 (business concept)	box header
enumeration	Classes	Enumeration	Enumeration	box header
executable	Deployments	Artifact	standard stereotype:L2	box header
executionEnvironment	Deployments	ExecutionEnvironment	ExecutionEnvironment	box header
extend	Use Cases	Extend	Extend	dashed-line label
extended	State Machines	Region	Region::extendedRegion ->notEmpty()	between braces
extended	State Machines	StateMachine	StateMachine::redefinedBehavior->notEmpty()	between braces
external	Activities	ActivityPartition	ActivityPartition::isExternal = true	swimlane header
file	Deployments	Artifact	standard stereotype:L2	box header
focus	Classes	Class	standard stereotype:L1	box header
framework	Classes	Package	standard stereotype:L1	box header
from	CommonBehaviors	Trigger	to show port name	inline label
implement	Components	Component	standard stereotype:L1	box header
implementationClass	Classes	Class	standard stereotype:L1	box header
import	Classes	PackageImport	visibility = #public	dashed-line label
include	Use Cases	Include	Include	dashed-line label
information	Auxiliary::Information-Flows	InformationItem	InformationItem	box header
instantiate	Classes	Dependency	Dependency	dashed-line label
instantiate	Classes	Usage	standard stereotype:L1	dashed-line label
interface	Classes	Interface	Interface	box header
library	Deployments	Artifact	standard stereotype:L2	box header
localPostcondition	Actions	Constraint	Action::localPostcondition	box header

Table B.1 - UML Keywords

Keyword	Language Unit	Metamodel Element	Semantics	Notation Placement
localPrecondition	Actions	Constraint	Action::localPrecondition	box header
manifest	Deployments	Manifestation	Manifestation	dashed-line label
merge	Classes	PackageMerge	PackageMerge	dashed-line label
metaclass	Profiles	Classifier	metaclass being stereotyped	box header
metamodel	Auxiliary::Models	Model	standard stereotype:L3	box header
model	Auxiliary::Models	Model	Model	box header
modelLibrary	Classes	Package	standard stereotype:L1	box header
multicast	Activities	ObjectFlow	ObjectFlow::isMulticast	flow label
multireceive	Activities	ObjectFlow	ObjectFlow::isMultireceive	flow label
occurrence	CompositeStructures	Collaboration	(Behavior::context) from a Collaboration to the owning BehavioredClassifier, indicating that the collaboration represents the use of a classifier.	dashed-line label
postcondition	Activities	Constraint	Behavior::postcondition	box header
precondition	Activities	Constraint	Behavior::precondition	box header
primitive	Classes	PrimitiveType	PrimitiveType	box header
process	Components	Component	standard stereotype:L2	box header
profile	Profiles	Profile	Profile	box header
provided interfaces	Components	Component	Component::provided	list-box header
realization	Classes	Classifier	standard stereotype:L2	box header
realizations	Components	Component	Component::realizations->collect(r r.realizingClassifier)	list-box header
refine	Classes	Abstraction	standard stereotype:L1	dashed-line label
representation	Auxiliary::Information-Flows	Classifier	InformationFlow::conveyed	dashed-line label
represents	CompositeStructures	Collaboration	(Behavior::context) from a Collaboration to the owning BehavioredClassifier; collaboration is USED in the classifier	dashed-line label
required interfaces	Components	Component	Component::required	list-box header
responsibility	Classes	Usage	standard stereotype:L1	dashed-line label
script	Deployments	Artifact	standard stereotype:L1	box header
selection	Activities	Behavior	ObjectFlow::selection	box header
selection	Activities	Behavior	ObjectNode::selection	box header
send	Classes	Usage	standard stereotype:L1	dashed-line label

Table B.1 - UML Keywords

Keyword	Language Unit	Metamodel Element	Semantics	Notation Placement
service	Components	Component	standard stereotype:L2	box header
signal	CommonBehaviors	Signal	Signal	box header
singleExecution	Activities	Activity	Activity::isSingleExecution = true	inside box
source	Deployments	Artifact	standard stereotype:L2	box header
specification	Components	Classifier	standard stereotype:L2	box header
statemachine	State Machines	BehavioredClassifier::ownedBehavior	BehavioredClassifier::ownedBehavior.oclIsKindOf(StateMachine)	box header
stereotype	Profiles	Stereotype	Stereotype	box header
structured	Activities	StructuredActivityNode	StructuredActivityNode	box header
substitute	Classes	Substitution	Substitution	dashed-line label
subsystem	Components	Component	standard stereotype:L2	box header
systemModel	Auxiliary::Models	Model	standard stereotype:L3	box header
trace	Classes	Abstraction	standard stereotype:L1	dashed-line label
transformation	Activities	Behavior	ObjectFlow::transformation	box header
type	Classes	Class	standard stereotype:L1	box header
use	Classes	Usage	Usage	dashed-line label
utility	Classes	Class	standard stereotype:L1	box header
when	CommonBehaviors	ChangeEvent	ChangeEvent::changeExpression	inline label

Annex C (normative)

Standard Stereotypes

This annex describes the predefined standard stereotypes for UML. The standard stereotypes are specified in three separate system-defined profiles, corresponding to the top three compliance levels of UML. These profiles can be applied to a user model just like any other profile. However, it is not necessary to include an explicit profile definition in such cases as it is assumed that such definitions are included (implicitly or explicitly) within any tool that is compliant with the standard. Of course, a tool need only support the profile that is consistent with its level of standard compliance.

The stereotypes belonging to the profile are described using a compact tabular form rather than graphically. The first column gives the name of the stereotype label corresponding to the stereotype. The actual name of the stereotype is the same as the stereotype label except that the first letter of each is capitalized. The second column identifies the language unit of the stereotype. The third column identifies the metaclass to which the stereotype applies and the last column provides a description of the meaning of the stereotype.

C.1 StandardProfileL1

Name	Language Unit	Applies to	Description
«auxiliary»	Classes::Kernel	Class	A class that supports another more central or fundamental class, typically by implementing secondary logic or control flow. The class that the auxiliary supports may be defined explicitly using a Focus class or implicitly by a dependency relationship. Auxiliary classes are typically used together with Focus classes, and are particularly useful for specifying the secondary business logic or control flow of components during design. See also: «focus».
«call»	Classes::Dependencies	Usage	A usage dependency whose source is an operation and whose target is an operation. The relationship may also be subsumed to the class containing an operation, with the meaning that there exists an operation in the class to which the dependency applies. A call dependency specifies that the source operation or an operation in the source class invokes the target operation or an operation in the target class. A call dependency may connect a source operation to any target operation that is within scope including, but not limited to, operations of the enclosing classifier and operations of other visible classifiers.
«create»	Dependencies	Usage	A usage dependency denoting that the client classifier creates instances of the supplier classifier.

Name	Language Unit	Applies to	Description
«create»	Classes::Kernel	BehavioralFeature	Specifies that the designated feature creates an instance of the classifier to which the feature is attached. May be promoted to the Classifier containing the feature.
«derive»	Classes::Dependencies	Abstraction	Specifies a derivation relationship among model elements that are usually, but not necessarily, of the same type. A derived dependency specifies that the client may be computed from the supplier. The mapping specifies the computation. The client may be implemented for design reasons, such as efficiency, even though it is logically redundant.
«destroy»	Classes::Kernel	BehavioralFeature	Specifies that the designated feature destroys an instance of the classifier to which the feature is attached. May be promoted to the classifier containing the feature.
«focus»	Classes::Kernel	Class	A class that defines the core logic or control flow for one or more auxiliary classes that support it. Support classes may be defined explicitly using Auxiliary classes or implicitly by dependency relationships. Focus classes are typically used together with one or more Auxiliary classes, and are particularly useful for specifying the core business logic or control flow of components during design. See also: «auxiliary».
«framework»	Classes::Kernel	Package	A package that contains model elements that specify a reusable architecture for all or part of a system. Frameworks typically include classes, patterns, or templates. When frameworks are specialized for an application domain they are sometimes referred to as application frameworks.
«implementation Class»	Classes::Kernel	Class	The implementation of a class in some programming language (e.g., C++, Smalltalk, Java) in which an instance may not have more than one class. This is in contrast to Class, for which an instance may have multiple classes at one time and may gain or lose classes over time, and an object (a child of instance) may dynamically have multiple classes. An Implementation class is said to realize a Classifier if it provides all of the operations defined for the Classifier with the same behavior as specified for the Classifier's operations. An Implementation Class may realize a number of different Types. Note that the physical attributes and associations of the Implementation class do not have to be the same as those of any Classifier it realizes and that the Implementation Class may provide methods for its operations in terms of its physical attributes and associations. See also: «type».
«instantiate»	Classes::Dependencies	Usage	A usage dependency among classifiers indicating that operations on the client create instances of the supplier.
«metaclass»	Classes::Kernel	Class	A class whose instances are also classes.

Name	Language Unit	Applies to	Description
«modelLibrary»	Classes::Kernel	Package	A package that contains model elements that are intended to be reused by other packages. Model libraries are frequently used in conjunction with applied profiles. This is expressed by defining a dependency between a profile and a model library package, or by defining a model library as contained in a profile package. The classes in a model library are not stereotypes and tagged definitions extending the metamodel. A model library is analogous to a class library in some programming languages. When a model library is defined as a part of a profile, it is imported or deleted with the application or removal of the profile. The profile is implicitly applied to its model library. In the other case, when the model library is defined as an external package imported by a profile, the profile requires that the model library be there in the model at the stage of the profile application. The application or the removal of the profile does not affect the presence of the model library elements.
«refine»	Classes::Dependencies	Abstraction	Specifies a refinement relationship between model elements at different semantic levels, such as analysis and design. The mapping specifies the relationship between the two elements or sets of elements. The mapping may or may not be computable, and it may be unidirectional or bidirectional. Refinement can be used to model transformations from analysis to design and other such changes.
«responsibility»	Classes::Kernel	Usage	A contract or an obligation of an element in its relationship to other elements.
«script»	Deployments::Artifacts	Artifact	A script file that can be interpreted by a computer system. Subclass of «file».
«send»	Classes::Dependencies	Usage	A usage dependency whose source is an operation and whose target is a signal, specifying that the source sends the target signal.
«trace»	Classes::Dependencies	Abstraction	Specifies a trace relationship between model elements or sets of model elements that represent the same concept in different models. Traces are mainly used for tracking requirements and changes across models. Since model changes can occur in both directions, the directionality of the dependency can often be ignored. The mapping specifies the relationship between the two, but it is rarely computable and is usually informal.
«type»	Classes::Kernel	Class	A class that specifies a domain of objects together with the operations applicable to the objects, without defining the physical implementation of those objects. However, it may have attributes and associations. Behavioral specifications for type operations may be expressed using, for example, activity diagrams. An object may have at most one implementation class, however it may conform to multiple different types. See also: «implementationClass».
«utility»	Classes::Kernel	Class	A class that has no instances, but rather denotes a named collection of non-member attributes and operations, all of which are class-scoped.

C.2 StandardProfileL2

Name	Language Unit	Applies to	Description
«document»	Deployments:: Artifacts	Artifact	A generic file that is not a «source» file or «executable». Subclass of «file».
«entity»	Components:: BasicComponents	Component	A persistent information component representing a business concept.
«executable»	Deployments:: Artifacts	Artifact	A program file that can be executed on a computer system. Subclass of «file».
«file»	Deployments:: Artifacts	Artifact	A physical file in the context of the system developed.
«implement»	Components:: BasicComponents	Component	A component definition that is not intended to have a specification itself. Rather, it is an implementation for a separate «specification» to which it has a Dependency.
«library»	Deployments:: Artifacts	Artifact	A static or dynamic library file. Subclass of «file».
«process»	Components:: BasicComponents	Component	A transaction based component.
«realization»	Classes::Kernel	Classifier	A classifier that specifies a domain of objects and that also defines the physical implementation of those objects. For example, a Component stereotyped by «realization» will only have realizing Classifiers that implement behavior specified by a separate «specification» Component. See «specification». This differs from «implementation class» because an «implementation class» is a realization of a Class that can have features such as attributes and methods that are useful to system designers.
«service»	Components:: BasicComponents	Component	A stateless, functional component (computes a value).
«source»	Deployments:: Artifacts	Artifact	A source file that can be compiled into an executable file. Subclass of «file».
«specification»	Classes::Kernel	Classifier	A classifier that specifies a domain of objects without defining the physical implementation of those objects. For example, a Component stereotyped by «specification» will only have provided and required interfaces, and is not intended to have any realizing Classifiers as part of its definition. This differs from «type» because a «type» can have features such as attributes and methods that are useful to analysts modeling systems. Also see: «realization»
«subsystem»	Components:: BasicComponents	Component	A unit of hierarchical decomposition for large systems. A subsystem is commonly instantiated indirectly. Definitions of subsystems vary widely among domains and methods, and it is expected that domain and method profiles will specialize this construct. A subsystem may be defined to have specification and realization elements. See also: «specification» and «realization».

C.3 StandardProfileL3

Name	Subpackage	Applies to	Description
«buildComponent»	Components:: BasicComponents	Component	A collection of elements defined for the purpose of system level development activities, such as compilation and versioning.
«metamodel»	AuxilliaryConstructs:: Models	Model	A model of a model, that typically contains containing metaclasses. See <<metaclass>>.
«systemModel»	AuxilliaryConstructs:: Models	Model	A systemModel is a stereotyped model that contains a collection of models of the same physical system. A systemModel also contains all relationships and constraints between model elements contained in different models.

Changes from previous UML

The following table lists predefined standard elements for UML 1.x that are now obsolete.

Standard Element Name	Applies to Base Element
«access»	Permission
«appliedProfile»	Package
«association»	AssociationEnd
«copy»	Flow
«create»	CallEvent
«create»	Usage
«destroy»	CallEvent
«facade»	Package
«friend»	Permission
«invariant»	Constraint
«local»	AssociationEnd
«parameter»	AssociationEnd
«postcondition»	Constraint
«powertype»	Class
«precondition»	Constraint
«profile»	Package
«realize»	Abstraction
«requirement»	Comment
«self»	AssociationEnd
«signalflow»	ObjectFlowState

«stateInvariant»	Constraint
«stub»	Package
«table»	Artifact
«thread»	Classifier
«topLevel»	Package

Annex D

Component Profile Examples

This annex describes example profiles for J2EE/Enterprise Java Beans (EJB), NET, COM and CORBA Component Model (CCM) components. These profiles are not meant to be either normative or complete, but are provided as an illustration of how UML 2.0 can be customized to model component architectures.

D.1 J2EE/EJB Component Profile Example

Table D.1 shows an example profile for Enterprise Java Beans components.

Table D.1 - Example Profile for Enterprise Java Beans

Stereotype	Base Class	Parent	Tags	Constraints	Description
EJBEntityBean «EJBEntityBean»	Component	N/A	N/A	N/A	Indicates that a Component represents an EJB Entity Bean, a component that manages the business logic of an application.
EJBSessionBean «EJBSessionBean»	Component	N/A	N/A	N/A	Indicates that a Component represents an EJB Session Bean, a component that processes transactions.
EJBMessageDrivenBean «EJBMessageDrivenBean»	Component	N/A	N/A	N/A	Indicates that a Component represents an EJB Message-Driven Bean, a component that handles messages, and is invoked on the arrival of a message.
EJBHome «EJBHome»	Interface	N/A	N/A	N/A	Indicates that the Interface is an EJB Home interface that supports lifecycle and class-level operations.
EJBRemote «EJBRemote»	Interface	N/A	N/A	N/A	Indicates that the Interface is an EJB Remote interface that supports business-specific operations.
EJBService «EJBService»	Interface	N/A	N/A	N/A	Indicates that the Interface is an EJB Service interface that is exposed as a webservice definition.
EJBCreate «EJBCreate»	Method	N/A	N/A	N/A	Indicates that the Method is an EJB Create Method that facilitates a create operation.
EJBBusiness «EJBBusiness»	Method	N/A	N/A	N/A	Indicates that the Method is an EJB instance-level method that supports the business logic of the EJB associated with the Remote interface.
EJBSecurityRoleRef «EJBSecurityRoleRef»	Association	N/A	N/A	N/A	Indicates an Association between an EJB client and an EJB Role Name Reference supplier.
EJBRoleName «EJBRoleName»	Actor	N/A	N/A	N/A	Indicates the name of a security role used in the definitions of method permissions, etc.

Table D.1 - Example Profile for Enterprise Java Beans

Stereotype	Base Class	Parent	Tags	Constraints	Description
EJBRoleNameRef «EJBRoleNameRef»	Actor	N/A	N/A	N/A	Indicates the name of a security role reference used programmatically in an EJB's source code and mapped to an EJB Role Name.
JavaSourceFile «JavaSourceFile»	Artifact	«file»	N/A	N/A	Indicates that the Artifact represents a Java source file.
JAR «JAR»	Artifact	«file»	N/A	N/A	Indicates that the Artifact represents a JAR (Java ARchive) file.
EJBQL «EJBQL»	Expression	N/A	N/A	N/A	Indicates that the expression conforms to the EJB Query Language syntax.

D.2 COM Component Profile Example

Table D.2 shows an example profile for COM components.

Table D.2 - Example Profile for COM Components

Stereotype	Base Class	Parent	Tags	Constraints	Description
COMCoClass «COMCoClass»	Component	N/A	N/A	N/A	Indicates a Component specification in COM (as defined in a type library). Specifies all externally visible aspects of a component.
COMAtIClass «COMAtIClass»	Component	N/A	N/A	N/A	Indicates a component implementation class using the ATL framework. An ATL class realizes a CoClass and provides its implementation.
COMInterface «COMInterface»	Interface	N/A	N/A	N/A	Indicates a component interface. COMInterfaces are offered or required by CoClasses and realized by COMAtIClasses.
COMConnectionPoint «COMConnectionPoint»	Dependency	N/A	N/A	N/A	Indicates that a component publishes an interface for subscribers. It is a special kind of a provided interface.
COMTypeLibrary «COMTypeLibrary»	Package	N/A	N/A	N/A	Indicates a packaging of COM types (COMCoClasses, COMInterface) to be deployed as a library.
COMTLB «COMTLB»	Artifact				Indicates a (runtime) component specification file (similar to a deployment descriptor).
COMExe «COMExe»	Artifact	«file»	N/A	N/A	An artifact that deploys an executable COM server application.
COMDLL «COMDLL»	Artifact	«file»	N/A	N/A	An artifact that deploys a component library.

D.3 .NET Component Profile Example

Table D.3 shows an example profile for .NET components.

Table D.3 - Example Profile for .NET Components

Stereotype	Base Class	Parent	Tags	Constraints	Description
NetComponent «NetComponent»	Component	N/A	N/A	N/A	Indicates that a Component represents a component in the .NET framework.
NETProperty «NETProperty»	Property	N/A	N/A	N/A	Indicates a Property offered by a component for public get/set operations.
NETAssembly «NETAssembly»	Package	N/A	N/A	N/A	Indicates a .NET assembly, which is a runtime packaging entity for components and other type definitions.
MSI «MSI»	Artifact	N/A	N/A	N/A	Indicates a component self installer file.
DLL «DLL»	Artifact	«file»	N/A	N/A	Indicates a portable executable (PE) of type DLL.
EXE «EXE»	Artifact	«file»	N/A	N/A	Indicates a portable executable (PE) of type EXE.

D.4 CCM Component Profile Example

Table D.4 shows an example profile for CCM components.

Table D.4 - Example Profile for CCM Components

Stereotype	Base Class	Parent	Tags	Constraints	Description
CCMEntity «CCMEntity»	Component	N/A	N/A	N/A	Indicates that a Component represents an Entity CC, a component that manages the business logic of an application.
CCMSession «CCMSession»	Component	N/A	N/A	N/A	Indicates that a Component represents a Session CC, a component that processes transactions.
CCMService «CCMService»	Component	N/A	N/A	N/A	Indicates that a Component represents a Service CC, a component that models single independent execution of an "operation."
CCMProcess «CCMService»	Component	N/A	N/A	N/A	Indicates that a Component represents a service CC, a component realizing a long-lived business process.
CCMHome «CCMHome»	Interface	N/A		Must have a manages dependency.	Indicates that the Interface is a CCMHome that provides factory and finder methods for a particular CC.
CCMManages «CCMManages»	Dependency	N/A		Always between a CCMHome and some CCM component.	Associates a CCM home with the CC component it manages.
CCMFactory «CCMFactory»	Operation	N/A	N/A	Operation is owned by CCMHome.	Indicates that the Operation is a CCMHome Create Method that facilitates a create operation.

Table D.4 - Example Profile for CCM Components

Stereotype	Base Class	Parent	Tags	Constraints	Description
CCMFinder «CCMFinder»	Operation	N/A	N/A	Operation is owned by CCMHome.	Indicates that the operation is a finder method of a CCMHome.
CCMProvided «CCMProvided»	Port	N/A	N/A	Port owns a single provided interface.	Indicates a port that models a CC facet.
CCMRequired «CCMRequired»	Port	N/A	N/A	Port owns a single provided interface.	Indicates a port that models a CC Receptacle.
CCMPackage «CCMPackage»	Artifact	N/A	N/A	N/A	Indicates an artifact that deploys a set of CCs.
CCMComponentDescriptor «CCMComponentDescriptor»	Artifact	«file»	N/A	N/A	Indicates that the Artifact represents a CCMComponentDescriptor.
CCMSoftPkgDescriptor «CCMComponentDescriptor»	Artifact	«file»	N/A	N/A	Indicates that the Artifact represents a CCM softPkg descriptor.

Annex E

(normative)

Tabular Notations

This annex describes optional tabular notations for UML behavioral diagrams, that some vendors or users may want to use as alternatives to UML's graphic notation. Although this appendix mostly describes tabular notations for sequence diagrams, the approach may also be applied to other kinds of behavioral diagrams.

E.1 Tabular Notation for Sequence Diagrams

This section describes an optional tabular notation for sequence diagrams. The table row descriptions for this notation follow:

1. **Lifeline Class:** Designates Class name of Lifeline. If there is no Class name on the Lifeline symbol, this class name is omitted.
2. **Lifeline Instance:** Designates Instance name of Lifeline. If there is no Instance name on the Lifeline symbol, this instance name is omitted.
3. **Constraint:** Designates some kind of constraint. For example, indication of oblique line is denoted as "{delay}." To represent CombinedFragments, those operators are denoted with an index adorned by square bracket. In a case of InteractionUse, it is shown as parenthesized "Diagram ID," which designates referred Interaction Diagram, with "ref" tag, like "ref(M.sq)."
4. **Message Sending Class:** Designates the message sending class name for each incoming arrow.
5. **Message Sending instance:** Designates the message sending instance name for each incoming arrow. In a case of Gate message that is outgoing message from InteractionUse, it is shown as parenthesized "Diagram ID," which designates referred Interaction Diagram, with underscore, like "_ (M.sq)."
6. **Diagram ID:** Identifies the document that describes the corresponding sequence/communication diagram and can be the name of the file that contains the corresponding sequence or communication diagram.
7. **Generated instance name:** An identifier name that is given to each instance symbol in the sequence /communication diagram. The identifier name is unique in each document.
8. **Sequence Number:** The corresponding message number on the sequence /communication diagram.
9. **Weak Order:** Designates partial (relative) orders of events, as ordered on individual lifelines and across lifelines, given a message receive event has to occur after its message send event. See definition of weak order (section 34.1 in the U2 partners submission.) Events are shown as "e" + event order + event direction (incoming or outgoing).
10. **Message name:** The corresponding message name on the sequence /communication diagram.
11. **Parameter:** A set of parameter variable names and parameter types of the corresponding message on the sequence/communication diagrams.
12. **Return value:** The return value type of the corresponding message on the sequence/communication diagram.

13. Message Receiving Class: Designates the message receiving class name for each outgoing arrow.
14. Message Receiving Instance: Designates the message receiving instance name for each outgoing arrow. In a case of Gate message that is outgoing message from ordinary instance symbol, it is shown as parenthesized message name with “out_” tag, like “(out_s).”
15. Other End: Designates event order of another end on the each message.

Examples

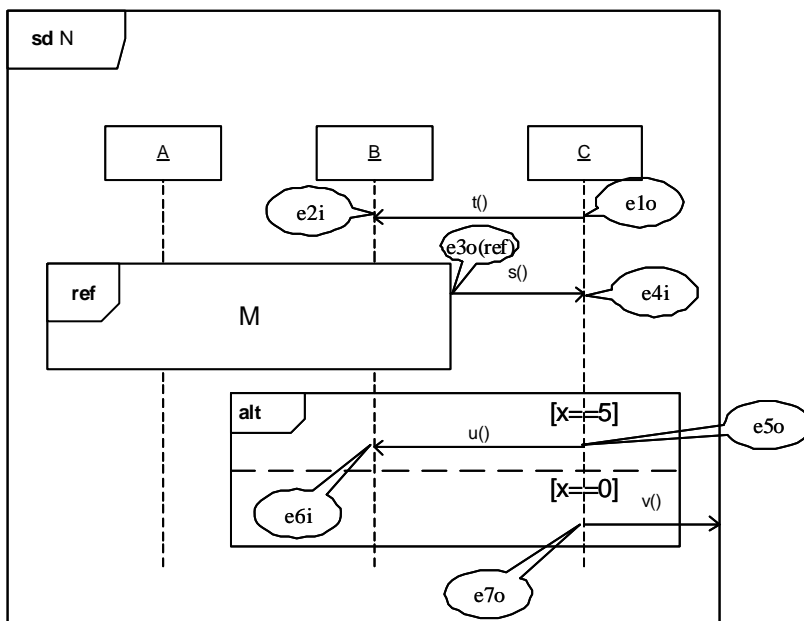


Figure E.1 - Sequence diagram enhanced with identification of the Event occurrences

Table E.1 - Interaction Table describing Figure E-1

Lifeline Class	Lifeline Inst	Constraint	Message Sending Class	Message Sending Instance	Diag ID	Generated Instance Name	Sequence No	Weak Order	Mesg Name	Parameter	Return Value	Message Receiving Class	Message Receive Instance	Other End
	C				N.sq			e1o	t				B	e2i
	B			C	N.sq			e2i	t					e1o
	A	Ref (M.sq)			N.sq			e3o(ref)	s				C	e4i
	B	Ref (M.sq)			N.sq			e3o(ref)	s				C	e4i
	C			_(M.sq)	N.sq			e4i	s					e3o(ref)
	C	alt [1]x==5			N.sq			e5o	u				B	e6i
	B	alt[1] x==5		B	N.sq			e6i	u					e5o
	C	alt[2] x==o			N.sq			e7o	v				out_v	

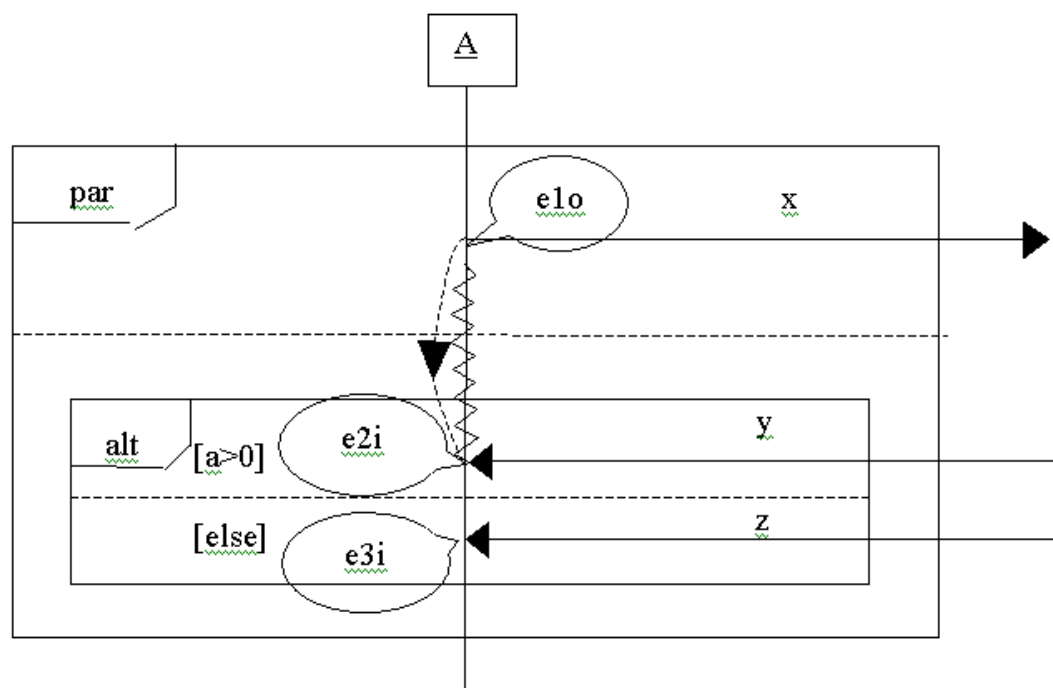


Figure E.2 - Sequence diagram with guards, parallel composition and alternatives

Table E.2 - Interaction Table for Figure E-2

Lifeline Class	Lifeline Inst	Constraint	Message Sending Class	Message Sending Inst	Diag ID	Generated Instance Name	Seq No	Weak Order	Message Name	Parameter	Return Value	Message Receiving Class	Message Receive Inst	Other End
	A	par[1]			para_sq			e1o	x				out_x	
	A	par[2].alt[1] {after e1o} a > 0		in_v	para_sq			e2i	y					
	A	par[2].alt[2]else		in_z	para_sq			e3i	z					
								-						

E.2 Tabular Notation for Other Behavioral Diagrams

The approach for defining tabular notation for sequence diagrams should also be applicable to other major behavioral diagram types, such as state machine diagrams and activity diagrams.

Annex F (normative)

Classifiers Taxonomy

The class inheritance hierarchy of the Classifiers in the UML 2.0 Superstructure is shown in Figure F.1. The root of the Classifier hierarchy is the Classifier defined in the Classes::Kernel package, and includes numerous merge increments and subclasses and their increments. The Classifier hierarchy includes package references, so that readers can refer to their definitions in the appropriate packages.

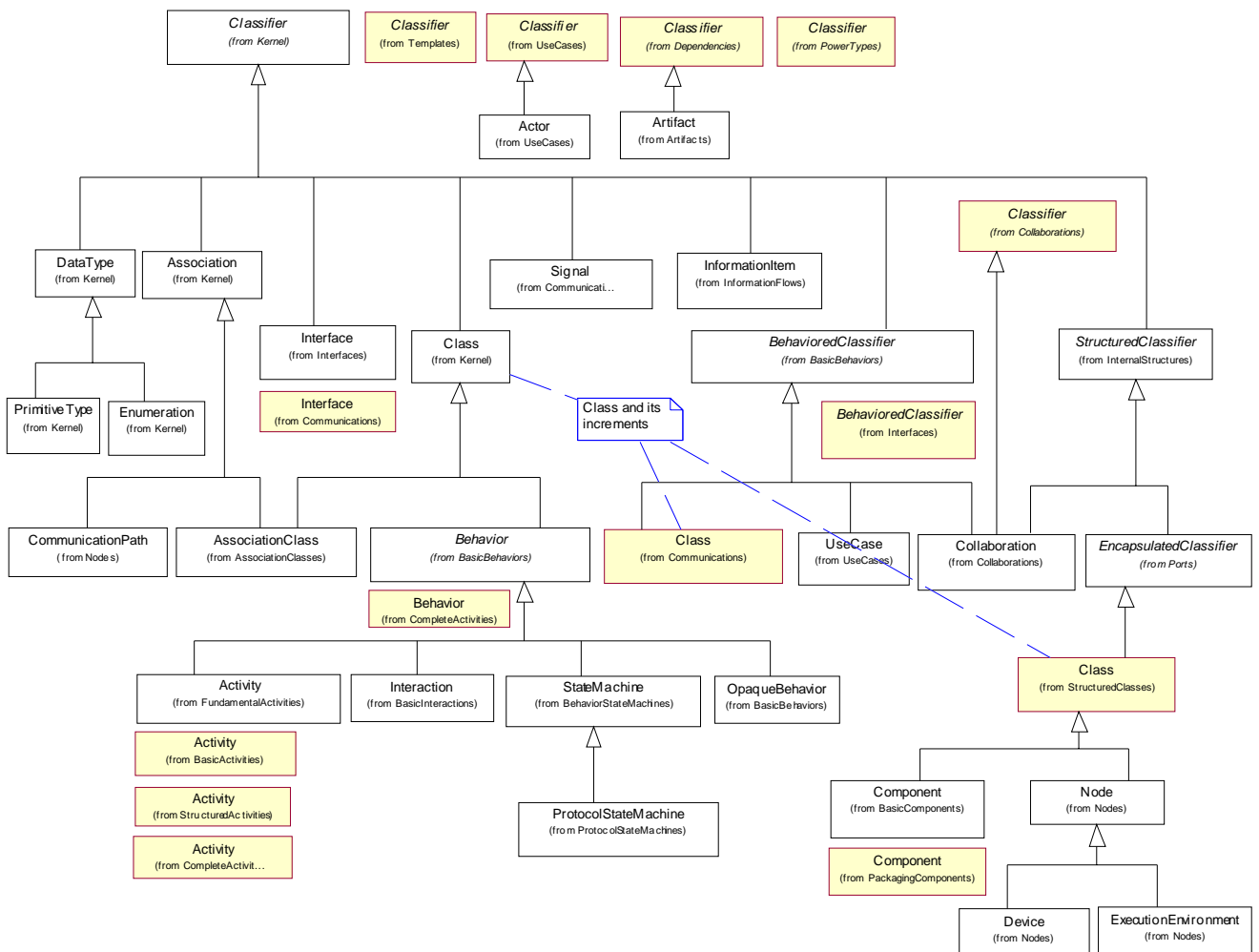


Figure F.1 - Classifier hierarchy for UML 2.0 Superstructure

Annex G

(normative)

XMI Serialization and Schema

UML 2.0 models are serialized in XMI 2.0 according to the rules specified by the proposed MOF 2.0 XMI specification (OMG document ad/2002-12-07). The XML schema for MOF 2.0 models that support the MOF 2.0 XMI specification is available in OMG document ad/2002-12-09.

The XMI for serializing the UML 2.0 Superstructure as an instance of MOF 2.0 according to the rules specified by the proposed MOF2 XMI specification (available in OMG document ad/2003-04-04). It is expected that the normative XMI for this specification will be generated by a Finalization Task Force, which will architecturally align and finalize the relevant specifications.

Index

A

Abstraction (from Dependencies) 35
AcceptCallAction (from CompleteActions) 227
AcceptEventAction (as specialized) 299
AcceptEventAction (from CompleteActions) 228
Action 230, 301, 452, 467
ActionExecutionSpecification (from BasicInteractions) 452
ActionInputPin 231, 305
Activity 305, 315, 322, 323, 330
ActivityEdge 315
ActivityFinalNode 320
ActivityGroup 322
ActivityNode 323
ActivityParameterNode 326
ActivityPartition 329
activityScope 401
Actor (from UseCases) 570
actual 609
actualGate 473
addition 577
AddStructuralFeatureValueAction 233
AddVariableValueAction 234, 335
after 467
aggregation 118
AggregationKind (from Kernel) 35
alias 61
allFeatures 49
allowSubstitutable 616
alt 472
ancestor 546
annotatedElement 53
AnyReceiveEvent (from Communications) 415
appliedProfile 641
argument 250, 473, 477
Artifact (from Artifacts, Nodes) 192
assert 473
association 119, 240
Association (from Kernel) 36
AssociationClass (from AssociationClasses) 42
associationEnd 119
attribute 48

B

Behavior 238, 336, 432, 452
Behavior (from BasicBehaviors) 416
BehavioralFeature (from BasicBehaviors, Communications) 418
BehavioralFeature (from CompleteActivities) 336
BehavioralFeature (from Kernel) 44
BehavioredClassifier (from BasicBehaviors, Communications) 419
BehavioredClassifier (from Interfaces) 45
BehaviorExecutionSpecification (from BasicInteractions) 452
binding 609
body 53, 98, 255, 342, 432
bodyCondition 99
bodyOutput 342, 372
bodyPart 372
Boolean (from PrimitiveTypes) 596
booleanValue 86, 133

boundElement 606

BroadcastSignalAction 235

C

CallAction 237
CallBehaviorAction 237, 337
CallConcurrencyKind (from Communications) 421
CallEvent (from Communications) 421
CallOperationAction 239, 339
CentralBufferNode 340
cfragmentGate 453
ChangeEvent (from Communications) 422
changeExpression 423
ChangeTrigger (from Communications) 422
class 99
Class (from Communications) 423
Class (from Kernel) 45
Class (from Profiles) 636
Class (from StructuredClasses) 162
classifier 79, 246, 259, 260, 617
Classifier (from Collaborations) 163
Classifier (from Kernel, Dependencies, PowerTypes) 48
Classifier (from Templates) 611
Classifier (from UseCases) 572
classifierBehavior 420
ClassifierTemplateParameter (from Templates) 616
Clause 342
ClearAssociationAction 240
ClearAttributeAction 241
ClearStructuralFeatureAction 241
ClearVariableAction 242
client 58
clientDependency 94
Collaboration (from Collaborations) 164
collaborationRole 164
collaborationUse 163
CollaborationUse (from Collaborations) 166
CombinedFragment (from Fragments) 453
Comment (from Kernel) 53
CommunicationPath (from Nodes) 195
Component (from BasicComponents, PackagingComponents) 142
composite 36
concurrency 419
concurrent 421
condition 386, 574
ConditionalNode 342
configuration 197
conformance (of StateMachine) 516, 545
ConnectableElement (from InternalStructures) 170
ConnectableElement (from Templates) 627
ConnectableElementTemplateParameter (from Templates) 628
connection 533
connectionPoint 533
connectionPoint (of StateMachine) 545
ConnectionPointReference (from BehaviorStateMachines) 511
connector 477
Connector (from BasicComponents) 150
Connector (from InternalStructures) 170
ConnectorEnd (from InternalStructures, Ports) 172
ConnectorKind (from BasicComponents) 153
consider 473

- ConsiderIgnoreFragment (from Fragments) 458
- constrainedElement 54
- Constraint (from Kernel) 54
- containedNode 397
- container 553
- container (StateVertex) 563
- context 54, 230, 417
- Continuation (from Fragments) 459
- contract 85, 129
- ControlFlow 344
- ControlNode 346
- conveyed 591
- covered 471, 482
- CreateLinkAction 243
- CreateLinkObjectAction 244
- CreateObjectAction 245
- CreationEvent (from BasicInteractions) 462
- critical 472

D

- DataStoreNode 347
- datatype 119
- DataType (from Kernel) 56
- decider 342, 372
- decisionInput 349
- DecisionNode 349
- declarative protocol state machines 517
- decomposedAs 476
- default 116, 118, 608
- defaultValue 116, 119
- deferrableEvent (State) 533
- deferrableTrigger 533
- definingEnd 173
- definingFeature 127
- Dependency (from Dependencies) 58
- deployedArtifact 196
- DeployedArtifact (from Nodes) 195
- deployedElement 201
- deployment 199, 201, 206
- Deployment (from ComponentDeployments, Nodes) 196
- deploymentLocation 199
- DeploymentSpecification (from ComponentDeployments) 198
- DeploymentTarget (from Nodes) 200
- destroyAt 254
- DestroyLinkAction 246
- DestroyObjectAction 248
- DestructionEvent (from BasicInteractions) 462
- Device (from Nodes) 201
- DirectedRelationship (from Kernel) 59
- direction 116
- doActivity 533
- doActivity (State) 533
- duration 427
- Duration (from SimpleTime) 424
- DurationConstraint (from SimpleTime) 425
- DurationInterval (from SimpleTime) 426
- DurationObservationAction (from SimpleTime) 427

E

- edge 307
- edgeContents 323

- effect 384
- effect (Transition) 553
- Element (from Kernel) 60
- elementImport 96
- ElementImport (from Kernel) 61
- EncapsulatedClassifier (from Ports) 173
- enclosingOperand 471
- end 170, 171, 253, 263
- endData 243, 247, 250
- endType 36
- entry (ConnectionPoint) 511
- entry (State) 533
- entry point (kind of Pseudostate) 524
- enumeration 65
- Enumeration (from Kernel) 63
- EnumerationLiteral (from Kernel) 64
- event 424, 439, 441, 464, 467, 481, 482
- Event (from Communications) 428
- exception 259
- ExceptionHandler 351
- exceptionInput 351
- exceptionType 351
- executable protocol state machines 517
- ExecutableNode 354, 395
- execution 464
- ExecutionEnvironment (from Nodes) 202
- ExecutionEvent (from BasicInteractions) 463
- executionLocation 199
- ExecutionOccurrenceSpecification (from BasicInteractions) 464
- ExecutionSpecification (from BasicInteractions) 464
- exit (ConnectionPoint) 511
- exit (State) 533
- ExpansionKind 354
- ExpansionNode 354
- ExpansionRegion 355
- expression 132
- Expression (from Kernel) 65
- extend 579
- Extend (from UseCases) 573
- extendedCase 573
- extendedRegion 529
- extendedSignature 617
- extendedState 533
- extendedStateMachine 545
- extension 574, 636
- Extension (from Profiles) 637
- ExtensionEnd (from Profiles) 639
- extensionLocation 574
- extensionPoint 579
- ExtensionPoint (from UseCases) 575
- external 562

F

- feature 48
- Feature (from Kernel) 66
- featuringClassifier 66
- FIFO 383
- filename 193
- FinalNode 360
- FinalState (from BehaviorStateMachines) 513
- finish 465

- first 277
- firstTime 424, 439
- FlowFinalNode 362
- ForkNode 363
- formal 609
- formalGate 467
- fragment 467, 472
- fromAction 231
- FunctionBehavior (from BasicBehaviors) 428

G

- Gate (from Fragments) 466
- general 48, 67
- generalization 49, 72
- Generalization (from Kernel, PowerTypes) 67
- generalizationSet 68
- GeneralizationSet (from PowerTypes) 71
- generalMachine 516
- generalOrdering 471
- GeneralOrdering (from BasicInteractions) 466
- group 307
- guard 315, 472
- Guard (Trigger) 553
- guarded 421

H

- handler 354
- handlerBody 351

I

- icon 649
- ignore 473
- Image (from Profiles) 640
- implementingClassifier 85
- importedElement 61
- importedMember 96
- importedPackage 106
- importedProfile 648
- importingNamespace 61, 106
- include 579
- Include (from UseCases) 576
- includedCase 577
- incoming 324
- incoming (StateVertex) 563
- InformationFlow (from InformationFlows) 590
- InformationItem (from InformationFlows) 592
- inGroup 315, 324
- inheritedMember 49
- inheritedParameter 617
- InitialNode 365
- inPartition 315, 324
- input 230, 250
- inputElement 356
- InputPin (as specialized) 366
- InputPin (from BasicActions) 249
- insertAt 233, 234, 252
- instance 81
- InstanceSpecification (from Kernel) 78
- InstanceSpecification (from Nodes) 204
- InstanceValue (from Kernel) 81
- inState 380

- inStructuredGroup 315, 324
- Integer (from PrimitiveTypes) 597
- integerValue 87, 133
- interaction 476, 477
- Interaction (from BasicInteraction, Fragments) 467
- InteractionConstraint (from Fragments) 470
- InteractionFragment (from BasicInteractions, Fragments) 471
- InteractionOperand (from Fragments) 471
- interactionOperator 453
- InteractionOperator (from Fragments) 472
- InteractionUse (fromFragments) 473
- interface 100
- Interface (from Communications) 429
- Interface (from Interfaces) 82
- Interface (from ProtocolStateMachines) 514
- interfaceRealization 45
- InterfaceRealization (from Interfaces) 85
- internal 562
- InterruptibleActivityRegion 366
- interruptibleRegion 315, 324
- interruptingEdge 367
- Interval (from SimpleTime) 430
- IntervalConstraint (from SimpleTime) 430
- invariant 487
- InvocationAction (from Actions) 174
- InvocationAction (from BasicActions) 249
- isAbstract 48, 419
- isActive 423
- isAssured 343
- isBehavior 175
- isCombineDuplicate 369
- isComposite 118
- isComposite (derived attribute of State) 532
- isComputable 86, 87, 89, 90, 133
- isConcurrent (CompositeState) 532
- isConsistentWith 120
- isControl 388
- isControlType 380
- isCovering 71
- isDerived 36, 118
- isDestroyDuplicates 254
- isDestroyLinks 248
- isDestroyOwnedObjects 248
- isDeterminate 343
- isDimension 329
- isDirect 260
- isDisjoint 71
- isException 384
- isExternal 329
- isIndirectlyInstantiated 143
- isInternal (Transition) 553
- isLeaf 125
- isMultiCast 375
- isMultireceive 375
- isNull 87, 133
- isOrdered 90, 99
- isOrthogonal (derived attribute of State) 532
- isQuery 99
- isReadOnly 118, 128, 307
- isRedefinitionContextValid 546
- isReentrant 417

- isRelative 438
- isRemoveDuplicates 269, 271
- isReplaceAll 233, 234, 252, 268
- isRequired 637
- isService 175
- isSimple (derived attribute of State) 532
- isSingleCopy 307
- isSingleExecution 307
- isStatic 66
- isStream 384
- isSubmachineState (derived attribute of State) 532
- isSubstitutable 67
- isSynchronous 237
- isTestedFirst 372
- isUnique 90, 99
- isUnmarshall 228
- iterative 354

J

- JoinNode 368
- joinSpec 369

K

- kind 150
- kind (PseudoState) 522

L

- language 98, 255, 432
- LCA 546
- lifeline 467
- Lifeline (from BasicInteractions, Fragments) 475
- LIFO 383
- LinkAction 250
- LinkEndCreationData 251
- LinkEndData 253
- LinkEndDestructionData 254
- LiteralBoolean (from Kernel) 85
- LiteralInteger (from Kernel) 86
- LiteralNull (from Kernel) 87
- LiteralSpecification (from Kernel) 88
- LiteralString (from Kernel) 88
- LiteralUnlimitedNatural (from Kernel) 89
- local 562
- localPostcondition 301
- localPrecondition 301
- location 197
- loop 472
- LoopNode 371
- loopVariable 372
- loopVariableInput 372
- lower 90, 99
- lowerValue 91

M

- manifestation 193
- Manifestation (from Artifacts) 204
- mapping 35
- max 426, 430, 439
- maxint 470
- member 96
- memberEnd 36
- mergedPackage 107

- MergeNode 373
- mergingPackage 108
- message 458, 467, 480
- Message (from BasicInteractions) 477
- MessageEnd (from BasicInteractions) 480
- MessageEvent (from Communications) 431
- messageKind 477
- MessageKind (from BasicInteractions) 480
- MessageOccurrenceSpecification (from BasicInteractions) 480
- messageSort 477
- MessageSort (from BasicInteractions) 481
- metaclass 637
- metaclassReference 642
- metamodelReference 642
- method 419
- min 426, 430, 439
- minint 470
- mode 355
- Model (from Models) 594
- MultiplicityElement (from BasicActions) 255
- MultiplicityElement (from Kernel) 90
- mustIsolate 396

N

- name 94
- NamedElement (from Kernel, Dependencies) 93
- NamedElement (from Templates) 620
- nameExpression 621
- namespace 94
- Namespace (from Kernel) 95
- navigableOwnedEnd 36
- neg 472
- nestedArtifact 193
- nestedClassifier 46
- nestedInterface 82
- nestedPackage 103
- nestingPackage 103
- newClassifier 268
- node 307
- Node (from Nodes) 205
- nodeContents 322
- none 36
- now 440

O

- object 240, 260, 263, 264, 268, 275, 276, 278
- ObjectFlow 375
- ObjectNode 380
- ObjectNodeOrderingKind 383
- OccurrenceSpecification (from BasicInteractions) 481
- oldClassifier 268
- onPort 174
- OpaqueAction 255
- OpaqueBehavior (from BasicBehaviors) 432
- OpaqueExpression (from BasicBehaviors) 432
- OpaqueExpression (from Kernel) 97
- operand 65, 453
- operation 116, 239, 422, 486
- Operation (from Communications) 433
- Operation (from Kernel, Interfaces) 99
- Operation (from Templates) 624, 625

- OperationTemplateParameter (from Templates) 626
- opposite 119
- opt 472
- ordered 383
- ordering 380
- OutputPin 256
- out 117
- outgoing 324
- outgoing (StateVertex) 563
- output 230
- outputElement 356
- OutputPin 383
- ownedActual 609
- ownedAttribute 46, 57, 82, 182
- ownedBehavior 420
- ownedComment 60
- ownedConnector 182
- ownedDefault 608
- ownedElement 60
- ownedEnd 36, 637
- ownedLiteral 64
- ownedMember 96, 103, 143
- ownedOperation 46, 57, 82, 193
- ownedParameter 44, 99, 610
- ownedParameteredElement 608
- ownedParameterSets 336, 337
- ownedPort 174
- ownedProperty 193
- ownedReception 424, 430
- ownedRule 96
- ownedSignature 612
- ownedStereotype 642
- ownedTemplateSignature 604
- ownedTrigger 420
- ownedType 103
- ownedUseCase 572
- owner 60
- owningAssociation 119
- owningInstance 127
- owningParameter 603

P

- package 49, 103, 134
- Package (from Kernel) 103
- Package (from Profiles) 641
- Package (from Templates) 618
- PackageableElement (from Kernel) 105
- PackageableElement (from Templates) 620
- packageImport 96
- PackageImport (from Kernel) 106
- packageMerge 103
- PackageMerge (from Kernel) 107
- par 472
- parallel 354
- parameter 326, 417, 610, 612, 625, 627
- Parameter (from Collaborations) 175
- Parameter (from CompleteActivities) 383
- Parameter (from Kernel, AssociationClasses) 115
- ParameterableElement (from Templates) 602
- ParameterDirectionKind (from Kernel) 117
- parameteredElement 608, 616, 626, 628

- ParameterEffectKind (from CompleteActivities) 385
- parameterInSet 386
- ParameterSet 384, 386
- parameterSubstitution 606
- part 182
- PartDecomposition (from Fragments) 482
- partition 307
- partWithPort 173
- Pin 256, 387
- port (Event in Statemachine) 186, 415, 431, 441
- Port (from Ports) 175
- Port (from ProtocolStateMachines) 515
- postCondition 519
- postcondition 99, 417
- powertype 72
- powertypeExtent 49
- preCondition 519
- precondition 99, 417
- predecessorClause 342
- PrimitiveType (from Kernel) 117
- private 134
- Profile (from Profiles) 642
- ProfileApplication (from Profiles) 647
- Property (from InternalStructures) 179
- Property (from Kernel, AssociationClasses) 118
- Property (from Nodes) 207
- Property (from Templates) 629
- protected 134
- protectedNode 351
- protocol 515
- ProtocolConformance (from ProtocolStateMachines) 515
- ProtocolStateMachine (from ProtocolStateMachines) 516
- ProtocolTransition (from ProtocolStateMachines) 518
- provided 143, 176
- Pseudostate (from BehaviorStateMachines) 522
- PseudoStateKind (from BehaviorStateMachines) 528
- public 134

Q

- qualifiedName 94
- qualifier 119, 253, 258, 264
- QualifierValue 257

R

- raisedException 44, 100, 419
- RaiseExceptionAction 258
- ReadExtentAction 259
- ReadIsClassifiedObjectAction 260
- ReadLinkAction 261
- ReadLinkObjectEndAction 263
- ReadLinkObjectEndQualifierAction 264
- ReadSelfAction 265
- ReadStructuralFeatureAction 266
- ReadVariableAction 267
- realization 143, 590
- Realization (from BasicComponents) 153
- Realization (from Dependencies) 124
- realizingActivityEdge 591
- realizingConnector 591
- realizingMessage 591
- receiveEvent 477

- Reception (from Communications) 434
- ReclassifyObjectAction 268
- RedefinableElement (from Kernel) 125
- RedefinableTemplateSignature (from Templates) 617
- redefinedBehavior 417
- redefinedClassifier 49
- redefinedConnector 171
- redefinedElement 125, 315, 324
- redefinedInterface 82
- redefinedOperation 100
- redefinedPort 176
- redefinedProperty 119
- redefinitionContext 125, 529, 533, 553
- referred 519
- refersTo 473
- region 533
- Region (from BehaviorStateMachines) 529
- region (of StateMachine) 545
- regionAsInput 355
- regionAsOutput 355
- relatedElement 127
- Relationship (from Kernel) 126
- removeAt 270, 271
- RemoveStructuralFeatureValueAction 269
- RemoveVariableValueAction 270
- replacedTransition (Transition) 553
- ReplyAction 271
- replyToCall 272
- replyValue 272
- representation 163
- represented 592
- represents 330, 476
- request 273
- required 143, 176
- result 229, 237, 245, 246, 259, 260, 261, 263, 264, 265, 266, 267, 277, 278, 343, 372, 433
- return 117
- returnInformation 228, 272
- role 172, 182
- roleBinding 167

S

- scope 401
- second 277
- selection 376, 380
- selector 476
- sendEvent 477
- SendObjectAction 272
- SendObjectAction (as specialized) 393
- SendOperationEvent (from BasicInteractions) 485
- SendSignalAction 273
- SendSignalAction (as specialized) 394
- SendSignalEvent (from BasicInteractions) 486
- seq 473
- SequenceNode 395
- sequential 421
- setting 460
- setupPart 372
- shared 36
- signal 236, 274, 434, 436, 486
- Signal (from Communications) 435

- SignalEvent (from Communications) 435
- signature 477, 608
- slot 79
- Slot (from Kernel) 127
- source 59, 315, 591
- source (Transition) 553
- specific 67
- specification 54, 79, 417
- specificMachine 516
- start 465
- StartClassifierBehaviorAction 275
- state 511, 529
- State (from BehaviorStateMachines) 531
- stateGroup 522
- stateInvariant 533
- StateInvariant (from BasicInteractions) 487
- stateMachine 522
- statemachine 529
- StateMachine (from BehaviorStateMachines) 545
- Stereotype (from Profiles) 649
- stream 354
- strict 473
- String (from PrimitiveTypes) 598
- StringExpression (from Templates) 622
- stringValue 89, 133
- structuralFeature 276
- StructuralFeature (from Kernel) 128
- StructuralFeatureAction (from IntermediateActions) 275
- StructuredActivityNode 396
- StructuredClassifier (from InternalStructures) 182
- structuredNode 307
- subExpression 623
- subgroup 322, 330
- subject 578
- submachine 533
- submachineState 522
- subsettingProperty 119
- subsettingContext 120
- substitutingClassifier 129
- substitution 49
- Substitution (from Dependencies) 129
- subvertex (Region) 529
- successorClause 342
- superClass 46
- superGroup 322
- superPartition 330
- supplier 58
- supplierDependency 94
- symbol 65

T

- target 59, 239, 248, 273, 274, 315, 591
- target (Transition) 553
- template 607, 610
- TemplateableElement (from Templates) 604
- templateBinding 604
- TemplateBinding (from Templates) 606
- templateParameter 603
- TemplateParameter (from Templates) 607
- TemplateParameterSubstitution (from Templates) 609
- TemplateSignature (from Templates) 610

test 342, 372
 TestIdentityAction 277
 TimeConstraint (from SimpleTime) 437
 TimeEvent (from BehaviorStateMachines) 552
 TimeEvent (from Communications, SimpleTime) 438
 TimeExpression (from SimpleTime) 439
 TimeInterval (from SimpleTime) 439
 TimeObservationAction (from SimpleTime) 440
 toAfter 482
 toBefore 482
 top 516, 545
 transformation 376
 transition 516, 529, 545
 Transition (from BehaviorStateMachines) 553
 TransitionKind (from BehaviorStateMachines) 561
 trigger 228
 Trigger (from Communications) 441
 Trigger (from InvocationActions) 186
 trigger (Transition) 553
 type 100, 130, 131, 167, 171, 640
 Type (from Kernel) 130
 typeConstraint 630
 TypedElement (from Kernel) 131

U
 UnlimitedNatural (from PrimitiveTypes) 599
 unlimitedValue 90, 133
 UnmarshallAction 278
 UnmarshallAction (as specialized) 398
 unmarshallType 278
 unordered 383
 upper 90, 99
 upperBound 380
 upperValue 91
 Usage (from Dependencies) 131
 useCase 572
 UseCase (from UseCases) 578
 utilizedElement 205

V
 value 86, 89, 127, 253, 258, 279, 280, 282, 283
 ValuePin 279
 ValuePin (as specialized) 399
 ValueSpecification (from Kernel) 132
 ValueSpecification (from Templates) 630
 ValueSpecificationAction 280
 ValueSpecificationAction (as specialized) 399
 Variable 401
 variable 281, 307, 397
 Variable (from StructuredActivities) 186
 VariableAction 281
 Vertex (from BehaviorStateMachines) 562
 viewpoint 594
 visibility 61, 94, 105, 106
 VisibilityKind (from Kernel) 133

W
 weight 315
 when 438
 WriteLinkAction 281
 WriteStructuralFeatureAction 282

 WriteVariableAction 283

