

## Constraints

No additional constraints

## Semantics

A slot relates an instance specification, a structural feature, and a value or values. It represents that an entity modeled by the instance specification has a structural feature with the specified value or values. The values in a slot must conform to the defining feature of the slot (in type, multiplicity, etc.).

## Notation

See “InstanceSpecification (from Kernel).”

### 7.3.49 StructuralFeature (from Kernel)

A structural feature is a typed feature of a classifier that specifies the structure of instances of the classifier.

## Generalizations

- “Feature (from Kernel)” on page 66
- “MultiplicityElement (from Kernel)” on page 90
- “TypedElement (from Kernel)” on page 131

## Description

A structural feature is a typed feature of a classifier that specifies the structure of instances of the classifier. Structural feature is an abstract metaclass.

By specializing multiplicity element, it supports a multiplicity that specifies valid cardinalities for the collection of values associated with an instantiation of the structural feature.

## Attributes

- `isReadOnly`: Boolean                      States whether the feature’s value may be modified by a client. Default is false.

## Associations

No additional associations

## Constraints

No additional constraints

## Semantics

A structural feature specifies that instances of the featuring classifier have a slot whose value or values are of a specified type.

## Notation

A read only structural feature is shown using {readOnly} as part of the notation for the structural feature. A modifiable structural feature is shown using {unrestricted} as part of the notation for the structural feature. This annotation may be suppressed, in which case it is not possible to determine its value from the diagram.

## Presentation Options

It is possible to only allow suppression of this annotation when isReadOnly=false. In this case it is possible to assume this value in all cases where {readOnly} is not shown.

## Changes from previous UML

The meta-attribute *targetScope*, which characterized StructuralFeature and AssociationEnd in prior UML is no longer supported.

### 7.3.50 Substitution (from Dependencies)

#### Generalizations

- “Realization (from Dependencies)” on page 124

#### Description

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

#### Associations

- contract: Classifier [1] (Specializes *Dependency.target*.)
- substitutingClassifier: Classifier [1] (Specializes *Dependency.client*.)

#### Attributes

None

#### Constraints

No additional constraints

#### Semantics

The substitution relationship denotes runtime substitutability that is not based on specialization. Substitution, unlike specialization, does not imply inheritance of structure, but only compliance of publicly available contracts. A substitution like relationship is instrumental to specify runtime substitutability for domains that do not support specialization such as certain component technologies. It requires that (1) interfaces implemented by the contract classifier are also implemented by the substituting classifier, or else the substituting classifier implements a more specialized interface type. And, (2) the any port owned by the contract classifier has a matching port (see ports) owned by the substituting classifier.

## Notation

A Substitution dependency is shown as a dependency with the keyword «substitute» attached to it.

## Examples

In the example below, a generic Window class is substituted in a particular environment by the Resizable Window class.

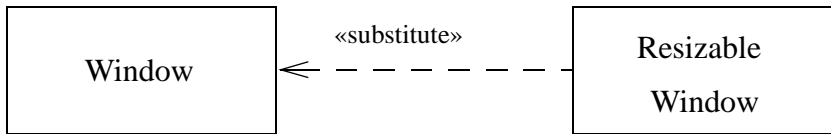


Figure 7.72 - An example of a substitute dependency

### 7.3.51 Type (from Kernel)

A type constrains the values represented by a typed element.

#### Generalizations

- “PackageableElement (from Kernel)” on page 105

#### Description

A type serves as a constraint on the range of values represented by a typed element. Type is an abstract metaclass.

#### Attributes

No additional attributes

#### Associations

No additional associations

#### Constraints

No additional constraints

#### Additional Operations

- [1] The query `conformsTo()` gives true for a type that conforms to another. By default, two types do not conform to each other. This query is intended to be redefined for specific conformance situations.  
`conformsTo(other: Type): Boolean;`  
`conformsTo = false`

#### Semantics

A type represents a set of values. A typed element that has this type is constrained to represent values within this set.

#### Notation

No general notation

### 7.3.52 TypedElement (from Kernel)

A typed element has a type.

#### Generalizations

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

#### Description

A typed element is an element that has a type that serves as a constraint on the range of values the element can represent. Typed element is an abstract metaclass.

#### Attributes

No additional attributes

#### Associations

- type: Type [0..1]      The type of the TypedElement.

#### Constraints

No additional constraints

#### Semantics

Values represented by the element are constrained to be instances of the type. A typed element with no associated type may represent values of any type.

#### Notation

No general notation

### 7.3.53 Usage (from Dependencies)

#### Generalizations

- “Dependency (from Dependencies)” on page 58

#### Description

A usage is a relationship in which one element requires another element (or set of elements) for its full implementation or operation. In the metamodel, a Usage is a Dependency in which the client requires the presence of the supplier.

#### Attributes

No additional attributes

#### Associations

No additional associations

## Constraints

No additional constraints

## Semantics

The usage dependency does not specify how the client uses the supplier other than the fact that the supplier is used by the definition or implementation of the client.

## Notation

A usage dependency is shown as a dependency with a «use» keyword attached to it.

## Examples

In the example below, an Order class requires the Line Item class for its full implementation.

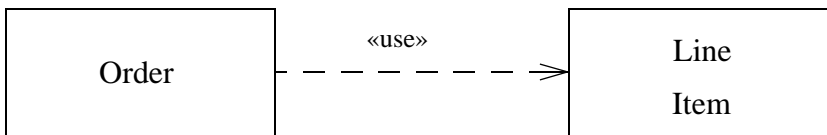


Figure 7.73 - An example of a use dependency

### 7.3.54 ValueSpecification (from Kernel)

A value specification is the specification of a (possibly empty) set of instances, including both objects and data values.

## Generalizations

- “PackageableElement (from Kernel)” on page 105
- “TypedElement (from Kernel)” on page 131

## Description

ValueSpecification is an abstract metaclass used to identify a value or values in a model. It may reference an instance or it may be an expression denoting an instance or instances when evaluated.

## Attributes

- expression: Expression[0..1] If this value specification is an operand, the owning expression. Subsets *Element::owner*.

## Associations

No additional associations

## Constraints

No additional constraints

## Additional Operations

These operations are introduced here. They are expected to be redefined in subclasses. Conforming implementations may be able to compute values for more expressions that are specified by the constraints that involve these operations.

- [1] The query `isComputable()` determines whether a value specification can be computed in a model. This operation cannot be fully defined in OCL. A conforming implementation is expected to deliver true for this operation for all value specifications that it can compute, and to compute all of those for which the operation is true. A conforming implementation is expected to be able to compute the value of all literals.

```
ValueSpecification::isComputable(): Boolean;  
isComputable = false
```

- [2] The query `integerValue()` gives a single Integer value when one can be computed.

```
ValueSpecification::integerValue() : [Integer];  
integerValue = Set{}
```

- [3] The query `booleanValue()` gives a single Boolean value when one can be computed.

```
ValueSpecification::booleanValue() : [Boolean];  
booleanValue = Set{}
```

- [4] The query `stringValue()` gives a single String value when one can be computed.

```
ValueSpecification::stringValue() : [String];  
stringValue = Set{}
```

- [5] The query `unlimitedValue()` gives a single UnlimitedNatural value when one can be computed.

```
ValueSpecification::unlimitedValue() : [UnlimitedNatural];  
unlimitedValue = Set{}
```

- [6] The query `isNull()` returns true when it can be computed that the value is null.

```
ValueSpecification::isNull() : Boolean;  
isNull = false
```

## Semantics

A value specification yields zero or more values. It is required that the type and number of values is suitable for the context where the value specification is used.

## Notation

No general notation

### 7.3.55 VisibilityKind (from Kernel)

VisibilityKind is an enumeration type that defines literals to determine the visibility of elements in a model.

## Generalizations

None

## Description

VisibilityKind is an enumeration of the following literal values:

- public
- private
- protected
- package

### Additional Operations

[1] The query `bestVisibility()` examines a set of `VisibilityKinds` that includes only public and private, and returns public as the preferred visibility.

```
VisibilityKind::bestVisibility(vis: Set(VisibilityKind)) : VisibilityKind;
```

```
pre: not vis->includes(#protected) and not vis->includes(#package)
```

```
bestVisibility = if vis->includes(#public) then #public else #private endif
```

### Semantics

`VisibilityKind` is intended for use in the specification of visibility in conjunction with, for example, the Imports, Generalizations, and Packages packages. Detailed semantics are specified with those mechanisms. If the `Visibility` package is used without those packages, these literals will have different meanings, or no meanings.

- A public element is visible to all elements that can access the contents of the namespace that owns it.
- A private element is only visible inside the namespace that owns it.
- A protected element is visible to elements that have a generalization relationship to the namespace that owns it.
- A package element is owned by a namespace that is not a package, and is visible to elements that are in the same package as its owning namespace. Only named elements that are not owned by packages can be marked as having package visibility. Any element marked as having package visibility is visible to all elements within the nearest enclosing package (given that other owning elements have proper visibility). Outside the nearest enclosing package, an element marked as having package visibility is not visible.

In circumstances where a named element ends up with multiple visibilities (for example, by being imported multiple times) public visibility overrides private visibility. If an element is imported twice into the same namespace, once using a public import and once using a private import, it will be public.

### Notation

The following visual presentation options are available for representing `VisibilityKind` enumeration literal values:

- '+' public
- '-' private
- '#' protected
- '~' package

## 7.4 Diagrams


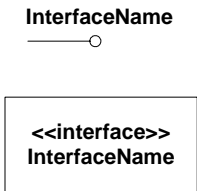
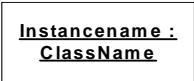

### Structure diagram

This 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 7.2.


**Table 7.2 - Graphic nodes included in structure diagrams**

<i>NODE TYPE</i>	<i>NOTATION</i>	<i>REFERENCE</i>
Class		See “Class (from Kernel)” on page 45.
Interface		See “Interface (from Interfaces)” on page 82.
InstanceSpecification		See “InstanceSpecification (from Kernel)” on page 78. (Note that instances of any classifier can be shown by prefixing the classifier name by the instance name followed by a colon and underlining the complete name string.)
Package		See “Package (from Kernel)” on page 103.

## Graphical paths

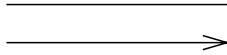




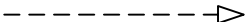
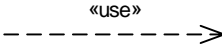
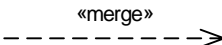
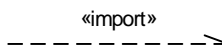
The graphic paths that can be included in structure diagrams are shown in Table 7.3.

**Table 7.3 - Graphic paths included in structure diagrams**

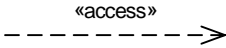
<i>PATH TYPE</i>	<i>NOTATION</i>	<i>REFERENCE</i>
Aggregation		See “AggregationKind (from Kernel)” on page 35.



**Table 7.3 - Graphic paths included in structure diagrams**

<i>PATH TYPE</i>	<i>NOTATION</i>	<i>REFERENCE</i>
Association		See “Association (from Kernel)” on page 36.
Composition		See “AggregationKind (from Kernel)” on page 35.
Dependency		See “Dependency (from Dependencies)” on page 58.
Generalization		See “Generalization (from Kernel, PowerTypes)” on page 67.
InterfaceRealization		See “InterfaceRealization (from Interfaces)” on page 85.
Realization		See “Realization (from Dependencies)” on page 124.
Usage		See “Usage (from Dependencies)” on page 131.
Package Merge		See “PackageMerge (from Kernel)” on page 107.
PackageImport (public)		See “PackageImport (from Kernel)” on page 106.

**Table 7.3 - Graphic paths included in structure diagrams**

<i>PATH TYPE</i>	<i>NOTATION</i>	<i>REFERENCE</i>
PackageImport (private)		See “PackageImport (from Kernel)” on page 106.

### Variations

Variations of structure diagrams often focus on particular structural aspects, such as relationships between packages, showing instance specifications, or relationships between classes. There are no strict boundaries between different variations; it is possible to display any element you normally display in a structure diagram in any variation.

#### *Class diagram*

The following nodes and edges are typically drawn in a class diagram:

- Association
- Aggregation
- Class
- Composition
- Dependency
- Generalization
- Interface
- InterfaceRealization
- Realization

#### *Package diagram*

The following nodes and edges are typically drawn in a package diagram:

- Dependency
- Package
- PackageExtension
- PackageImport

#### *Object diagram*

The following nodes and edges are typically drawn in an object diagram:

- InstanceSpecification
- Link (i.e., Association)



## 8 Components

### 8.1 Overview

The Components package specifies a set of constructs that can be used to define software systems of arbitrary size and complexity. In particular, the package specifies a component as a modular unit with well-defined interfaces that is replaceable within its environment. The component concept addresses the area of component-based development and component-based system structuring, where a component is modeled throughout the development life cycle and successively refined into deployment and run-time.

An important aspect of component-based development is the reuse of previously constructed components. A component can always be considered an autonomous unit within a system or subsystem. It has one or more provided and/or required interfaces (potentially exposed via ports), and its internals are hidden and inaccessible other than as provided by its interfaces. Although it may be dependent on other elements in terms of interfaces that are required, a component is encapsulated and its dependencies are designed such that it can be treated as independently as possible. As a result, components and subsystems can be flexibly reused and replaced by connecting (“wiring”) them together via their provided and required interfaces. The aspects of autonomy and reuse also extend to components at deployment time. The artifacts that implement component are intended to be capable of being deployed and re-deployed independently, for instance to update an existing system.

The Components package supports the specification of both logical components (e.g., business components, process components) and physical components (e.g., EJB components, CORBA components, COM+ and .NET components, WSDL components, etc.), along with the artifacts that implement them and the nodes on which they are deployed and executed. It is anticipated that profiles based around components will be developed for specific component technologies and associated hardware and software environments.

#### Basic Components

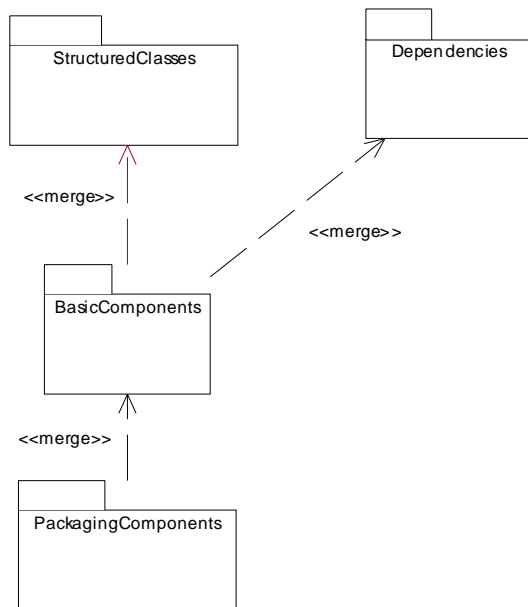
The BasicComponents package focuses on defining a component as an executable element in a system. It defines the concept of a component as a specialized class that has an external specification in the form of one or more provided and required interfaces, and an internal implementation consisting of one or more classifiers that realize its behavior. In addition, the BasicComponents package defines specialized connectors for ‘wiring’ components together based on interface compatibility.

#### Packaging Components

The PackagingComponents package focuses on defining a component as a coherent group of elements as part of the development process. It extends the concept of a basic component to formalize the aspects of a component as a ‘building block’ that may own and import a (potentially large) set of model elements.

## 8.2 Abstract syntax

Figure 8.1 shows the dependencies of the Component packages.



**Figure 8.1 - Dependencies between packages described in this chapter (transitive dependencies to Kernel and Interfaces packages are not shown).**

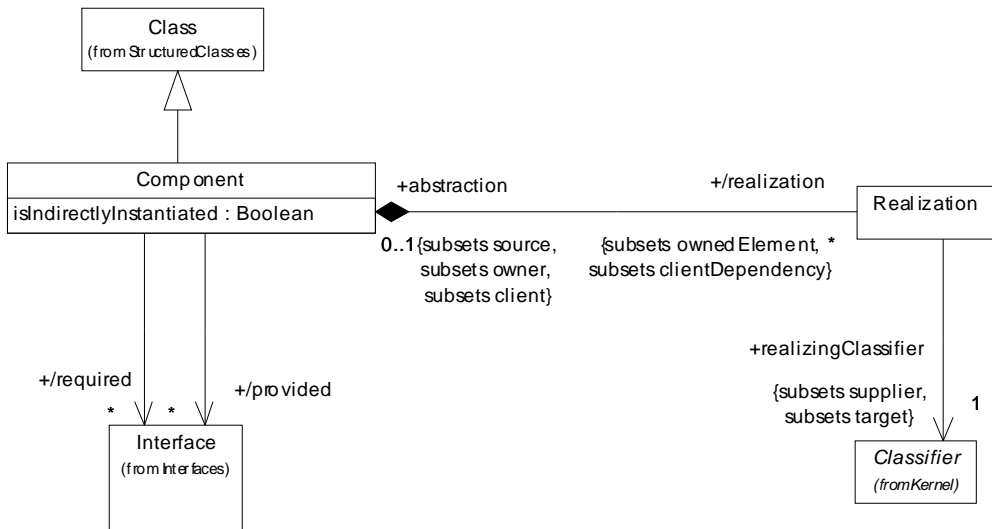


Figure 8.2 - The metaclasses that define the basic Component construct

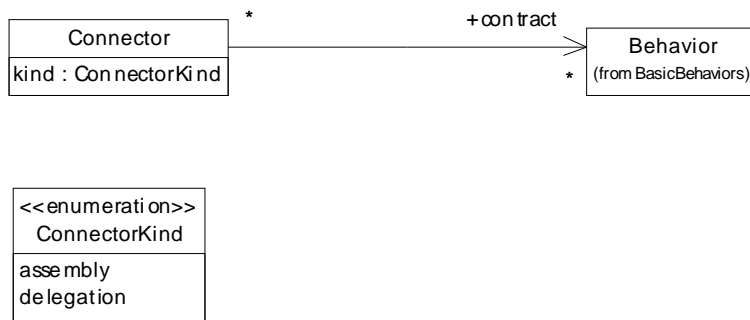


Figure 8.3 - The metaclasses that define the component wiring constructs

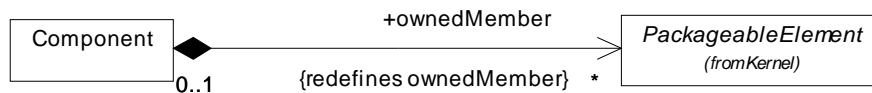


Figure 8.4 - The packaging capabilities of Components

## 8.3 Class Descriptions

### 8.3.1 Component (from BasicComponents, PackagingComponents)

A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.

A component defines its behavior in terms of provided and required interfaces. As such, a component serves as a type whose conformance is defined by these provided and required interfaces (encompassing both their static as well as dynamic semantics). One component may therefore be substituted by another only if the two are type conformant. Larger pieces of a system’s functionality may be assembled by reusing components as parts in an encompassing component or assembly of components, and wiring together their required and provided interfaces.

A component is modeled throughout the development life cycle and successively refined into deployment and run-time. A component may be manifest by one or more artifacts, and in turn, that artifact may be deployed to its execution environment. A deployment specification may define values that parameterize the component’s execution. (See Deployment chapter).

#### Generalizations

- “Class (from StructuredClasses)” on page 160

#### Description

##### BasicComponents

A component is a subtype of Class that provides for a Component having attributes and operations, and being able to participate in Associations and Generalizations. A Component may form the abstraction for a set of realizingClassifiers that realize its behavior. In addition, because a Class itself is a subtype of an EncapsulatedClassifier, a Component may optionally have an internal structure and own a set of Ports that formalize its interaction points.

A component has a number of provided and required Interfaces, that form the basis for wiring components together, either using Dependencies, or by using Connectors. A provided Interface is one that is either implemented directly by the component or one of its realizingClassifiers, or it is the type of a provided Port of the Component. A required interface is designated by a Usage Dependency from the Component or one of its realizingClassifiers, or it is the type of a required Port.

## PackagingComponents

A component is extended to define the grouping aspects of packaging components. This defines the Namespace aspects of a Component through its inherited ownedMember and elementImport associations. In the namespace of a component, all model elements that are involved in or related to its definition are either owned or imported explicitly. This may include, for example, UseCases and Dependencies (e.g., mappings), Packages, Components, and Artifacts.

## Attributes

### BasicComponents

- **isIndirectlyInstantiated** : Boolean {default = true}  
The kind of instantiation that applies to a Component. If *false*, the component is instantiated as an addressable object. If *true*, the Component is defined at design-time, but at run-time (or execution-time) an object specified by the Component does not exist, that is, the component is instantiated indirectly, through the instantiation of its realizing classifiers or parts. Several standard stereotypes use this meta attribute (e.g., «specification», «focus», «subsystem»).

## Associations

### BasicComponents

- **/provided: Interface [\*]**  
The interfaces that the component exposes to its environment. These interfaces may be Realized by the Component or any of its realizingClassifiers, or they may be the Interfaces that are provided by its public Ports. The provided interfaces association is a derived association:  
**context** Component::provided **derive**:  
**let** implementedInterfaces = self.implementation->collect(impl|impl.contract) **and**  
**let** realizedInterfaces = RealizeInterfaces(self) **and**  
**let** realizingClassifierInterfaces = RealizedInterfaces(self.realizingClassifier) **and**  
**let** typesOfRequiredPorts = self.ownedPort.provided **in**  
(((implementedInterfaces->union(realizedInterfaces)->union(realizingClassifierInterfaces))->  
union(typesOfRequiredPorts))->asSet())
- **/required: Interface [\*]**  
The interfaces that the component requires from other components in its environment in order to be able to offer its full set of provided functionality. These interfaces may be Used by the Component or any of its realizingClassifiers, or they may be the Interfaces that are required by its public Ports. The required interfaces association is a derived association:  
**context** Component::required **derive**:  
**let** usedInterfaces = UsedInterfaces(self) **and**  
**let** realizingClassifierUsedInterfaces = UsedInterfaces(self.realizingClassifier) **and**  
**let** typesOfUsedPorts = self.ownedPort.required **in**  
((usedInterfaces->union(realizingClassifierUsedInterfaces))->  
union(typesOfUsedPorts))->asSet())
- **/realization: Realization [\*]**  
The set of Realizations owned by the Component. Realizations reference the Classifiers of which the Component is an abstraction (i.e., that realize its behavior).

## PackagingComponents

- **ownedMember: PackageableElement [\*]**  
The set of PackageableElements that a Component owns. In the namespace of a component, all model elements that are involved in or related to its definition may be owned or imported explicitly. These may include, for example, Classes, Interfaces, Components, Packages, Use cases, Dependencies (e.g., mappings), and Artifacts.



## Constraints

No further constraints

## Additional Operations

[1] Utility returning the set of realized interfaces of a component:

```
def: RealizedInterfaces : (classifier : Classifier) : Interface = (classifier.clientDependency->
    select(dependency|dependency.ocllsKindOf(Realization) and dependency.supplier.ocllsKindOf(Interface)))->
    collect(dependency|dependency.client)
```

[2] Utility returning the set of required interfaces of a component:

```
def: UsedInterfaces : (classifier : Classifier) : Interface = (classifier.supplierDependency->
    select(dependency|dependency.ocllsKindOf(Usage) and dependency.supplier.ocllsKindOf(interface)))->
    collect(dependency|dependency.supplier)
```

## Semantics

A component is a self contained unit that encapsulates the state and behavior of a number of classifiers. A component specifies a formal contract of the services that it provides to its clients and those that it requires from other components or services in the system in terms of its provided and required interfaces.

A component is a substitutable unit that can be replaced at design time or run-time by a component that offers equivalent functionality based on compatibility of its interfaces. As long as the environment obeys the constraints expressed by the provided and required interfaces of a component, it will be able to interact with this environment. Similarly, a system can be extended by adding new component types that add new functionality.

The required and provided interfaces of a component allow for the specification of structural features such as attributes and association ends, as well as behavioral features such as operations and events. A component may implement a provided interface directly, or, its realizing classifiers may do so. The required and provided interfaces may optionally be organized through ports, these enable the definition of named sets of provided and required interfaces that are typically (but not always) addressed at run-time.

A component has an *external view* (or “black-box” view) by means of its publicly visible properties and operations. Optionally, a behavior such as a protocol state machine may be attached to an interface, port, and to the component itself, to define the external view more precisely by making dynamic constraints in the sequence of operation calls explicit. Other behaviors may also be associated with interfaces or connectors to define the ‘contract’ between participants in a collaboration (e.g., in terms of use case, activity, or interaction specifications).

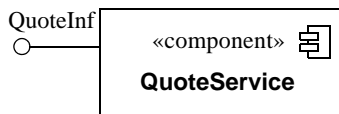
The wiring between components in a system or other context can be structurally defined by using dependencies between component interfaces (typically on structure diagrams). Optionally, a more detailed specification of the structural collaboration can be made using parts and connectors in composite structures, to specify the role or instance level collaboration between components (See Chapter Composite Structures).

A component also has an *internal view* (or “white-box” view) by means of its private properties and realizing classifiers. This view shows how the external behavior is realized internally. The mapping between external and internal view is by means of dependencies (on structure diagrams), or delegation connectors to internal parts (on composite structure diagrams). Again, more detailed behavior specifications such as interactions and activities may be used to detail the mapping from external to internal behavior.

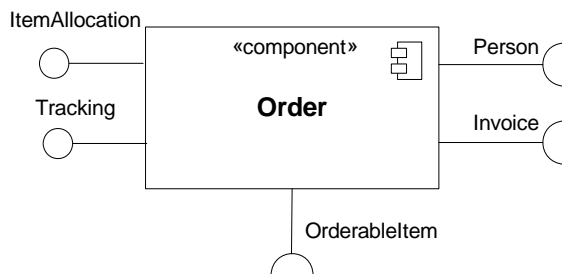
A number of UML standard stereotypes exist that apply to component. For example, «subsystem» to model large-scale components, and «specification» and «realization» to model components with distinct specification and realization definitions, where one specification may have multiple realizations (see the UML Standard Elements Appendix).

## Notation

A component is shown as a Classifier rectangle with the keyword «component». Optionally, in the right hand corner a component icon can be displayed. This is a classifier rectangle with two smaller rectangles protruding from its left hand side.

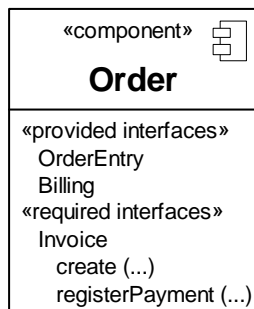


**Figure 8.5 - A Component with one provided interface**



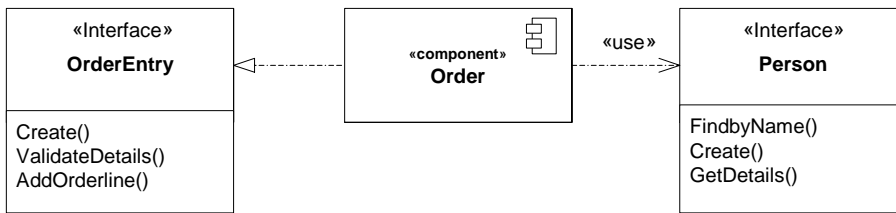
**Figure 8.6 - A Component with two provided and three required interfaces**

An external view of a Component is by means of Interface symbols sticking out of the Component box (external, or black-box view). Alternatively, the interfaces and/or individual operations and attributes can be listed in the compartments of a component box (for scalability, tools may offer way of listing and abbreviating component properties and behavior).



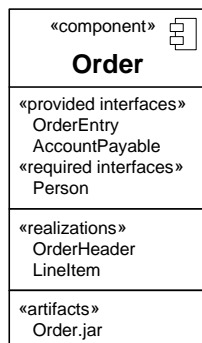
**Figure 8.7 - Black box notation showing a listing of the properties of a component**

For displaying the full signature of an interface of a component, the interfaces can also be displayed as typical classifier rectangles that can be expanded to show details of operations and events.



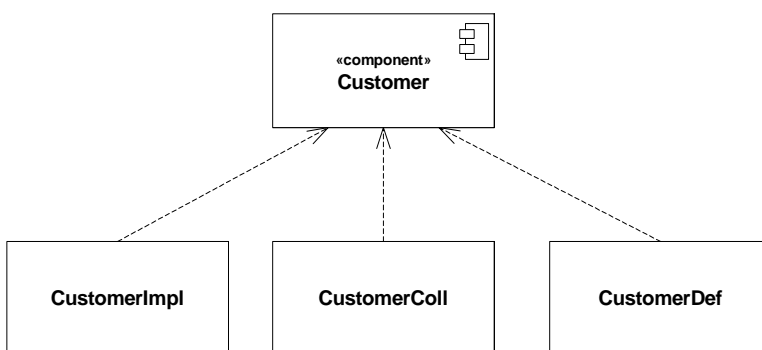
**Figure 8.8 - Explicit representation of the provided and required interfaces, allowing interface details such as operation to be displayed (when desired).**

An internal, or white box view of a Component is where the realizing classifiers are listed in an additional compartment. Compartments may also be used to display a listing of any parts and connectors, or any implementing artifacts.



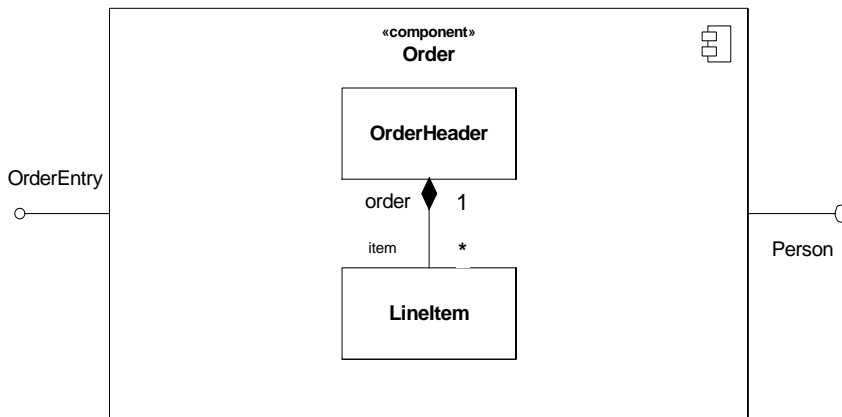
**Figure 8.9 - A white-box representation of a component**

The internal classifiers that realize the behavior of a component may be displayed by means of general dependencies. Alternatively, they may be nested within the component shape.



**Figure 8.10 - A representation of the realization of a complex component**

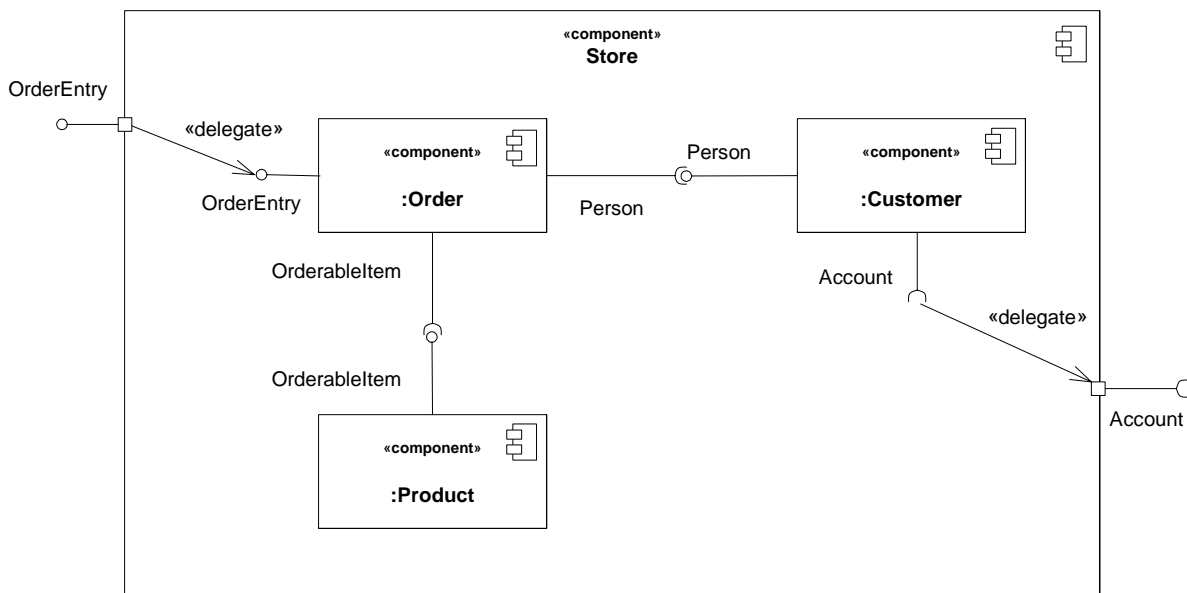
Alternatively, the internal classifiers that realize the behavior of a component may be displayed nested within the component shape.



**Figure 8.11 - An alternative nested representation of a complex component**

If more detail is required of the role or instance level containment of a component, then an internal structure consisting of parts and connectors can be defined for that component. This allows, for example, explicit part names or connector names to be shown in situations where the same Classifier (Association) is the type of more than one Part (Connector). That is, the Classifier is instantiated more than once inside the component, playing different roles in its realization. Optionally, specific instances (InstanceSpecifications) can also be referred to as in this notation.

Interfaces that are exposed by a Component and notated on a diagram, either directly or through a port definition, may be inherited from a supertype component. These interfaces are indicated on the diagram by preceding the name of the interface by a forward slash. An example of this can be found in Figure 8.14, where “/orderedItem” is an interface that is implemented by a supertype of the Product component.



**Figure 8.12 - An internal or white-box view of the internal structure of a component that contains other components as parts of its internal assembly.**

Artifacts that implement components can be connected to them by physical containment or by an «implement» relationship, which is an instance of the meta association between Component and Artifact.

## Examples

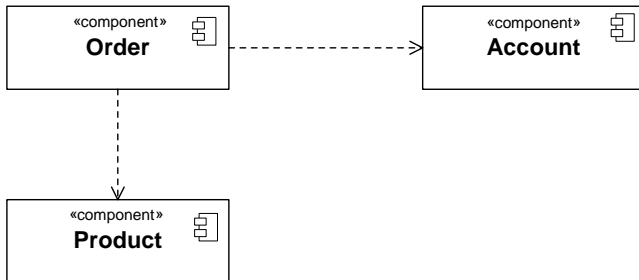


Figure 8.13 - Example of an overview diagram showing components and their general dependencies

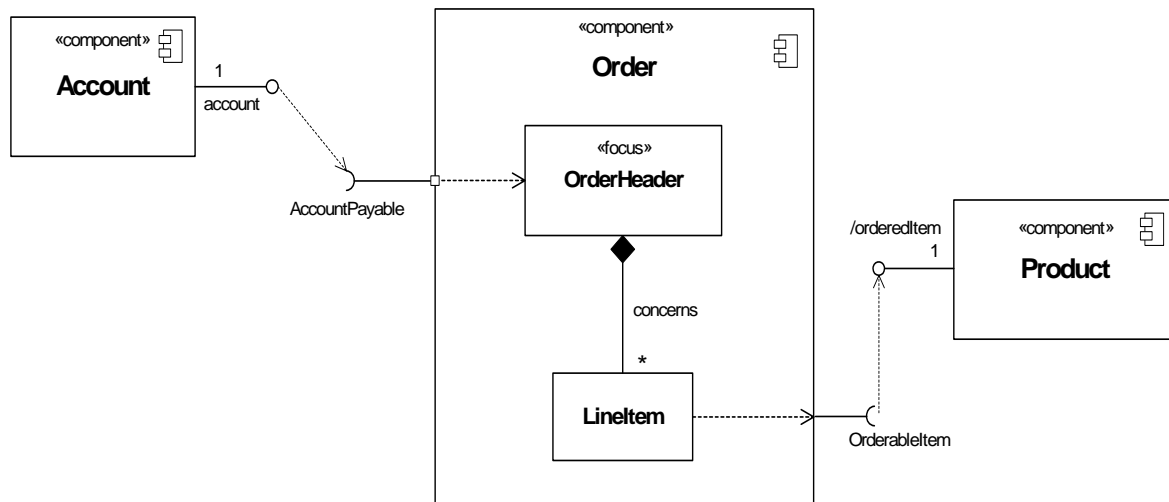
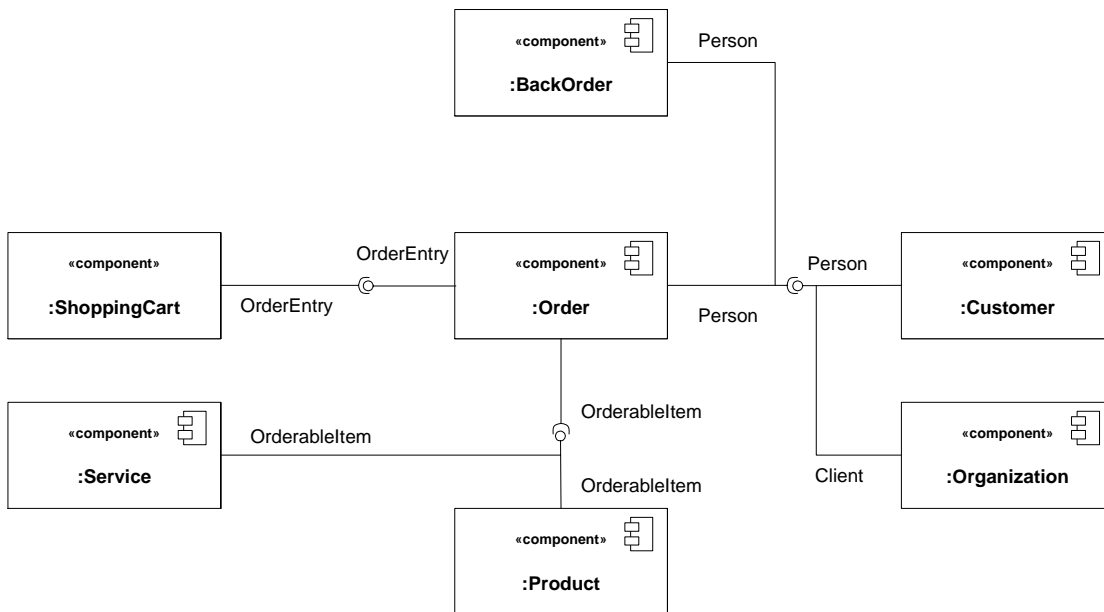


Figure 8.14 - Example of a platform independent model of a component, its provided and required interfaces, and wiring through dependencies on a structure diagram.



**Figure 8.15 - Example of a composite structure of components, with connector wiring between provided and required interfaces of parts (Note: “Client” interface is a subtype of “Person”).**

The wiring of components can be represented on structure diagrams by means of classifiers and dependencies between them (Note: the ball-and-socket notation from Figure 8.15 may be used as a notation option for dependency based wiring). On composite structure diagrams, detailed wiring can be performed at the role or instance level by defining parts and connectors.

## Changes from previous UML

The following changes from UML 1.x have been made.

The component model has made a number of implicit concepts from the UML 1.x model explicit, and made the concept more applicable throughout the modeling life cycle (rather than the implementation focus of UML 1.x). In particular, the “resides” relationship from 1.x relied on namespace aspects to define both namespace aspects as well as ‘residence’ aspects. These two aspects have been separately modeled in the UML metamodel in 2.0. The basic residence relationship in 1.x maps to the realizingClassifiers relationship in 2.0. The namespace aspects are defined through the basic namespace aspects of Classifiers in UML 2.0, and extended in the PackagingComponents metamodel for optional namespace relationships to elements other than classifiers.

In addition, the Component construct gains the capabilities from the general improvements in CompositeStructures (around Parts, Ports, and Connectors).

In UML 2.0, a Component is notated by a classifier symbol that no longer has two protruding rectangles. These were cumbersome to draw and did not scale well in all circumstances. Also, they interfered with any interface symbols on the edge of the Component. Instead, a «component» keyword notation is used in UML 2.0. Optionally, a component icon that is similar to the UML 1.4 icon can still be used in the upper right-hand corner of the component symbol. For backward compatibility reasons, the UML 1.4 notation with protruding rectangles can still be used.

### 8.3.2 Connector (from BasicComponents)

The connector concept is extended in the Components package to include interface based constraints and notation.

A *delegation* connector is a connector that links the external contract of a component (as specified by its ports) to the internal realization of that behavior by the component's parts. It represents the forwarding of signals (operation requests and events): a signal that arrives at a port that has a delegation connector to a part or to another port will be passed on to that target for handling.

An *assembly* connector is a connector between two components that defines that one component provides the services that another component requires. An assembly connector is a connector that is defined from a required interface or port to a provided interface or port.

#### Generalizations

- “Connector (from InternalStructures)” on page 170 (*merge increment*)

#### Description

In the metamodel, a connector kind attribute is added to the Connector metaclass. Its value is an enumeration type with valid values “assembly” or “delegation.”

#### Attributes

BasicComponents

- kind : ConnectorKind      Indicates the kind of connector.

#### Associations

No additional associations

#### Constraints

- [1] A delegation connector must only be defined between used Interfaces or Ports of the same kind (e.g., between two provided Ports or between two required Ports).
- [2] If a delegation connector is defined between a used Interface or Port and an internal Part Classifier, then that Classifier must have an “implements” relationship to the Interface type of that Port.
- [3] If a delegation connector is defined between a source Interface or Port and a target Interface or Port, then the target Interface must support a signature compatible subset of Operations of the source Interface or Port.
- [4] In a complete model, if a source Port has delegation connectors to a set of delegated target Ports, then the union of the Interfaces of these target Ports must be signature compatible with the Interface that types the source Port.
- [5] An assembly connector must only be defined from a required Interface or Ports to a provided Interface or Port.

#### Semantics

A delegation connector is a declaration that behavior that is available on a component instance is not actually realized by that component itself, but by another instance that has “compatible” capabilities. This may be another Component or a (simple) Class. The latter situation is modeled through a delegation connector from a Component Interface or Port to a contained Class that functions as a Part. In that case, the Class must have an implements relationship to the Interface of the Port.

Delegation connectors are used to model the hierarchical decomposition of behavior, where services provided by a component may ultimately be realized by one that is nested multiple levels deep within it. The word delegation suggests that concrete message and signal flow will occur between the connected ports, possibly over multiple levels. It should be noted that such signal flow is not always realized in all system environments or implementations (i.e., it may be design time only).

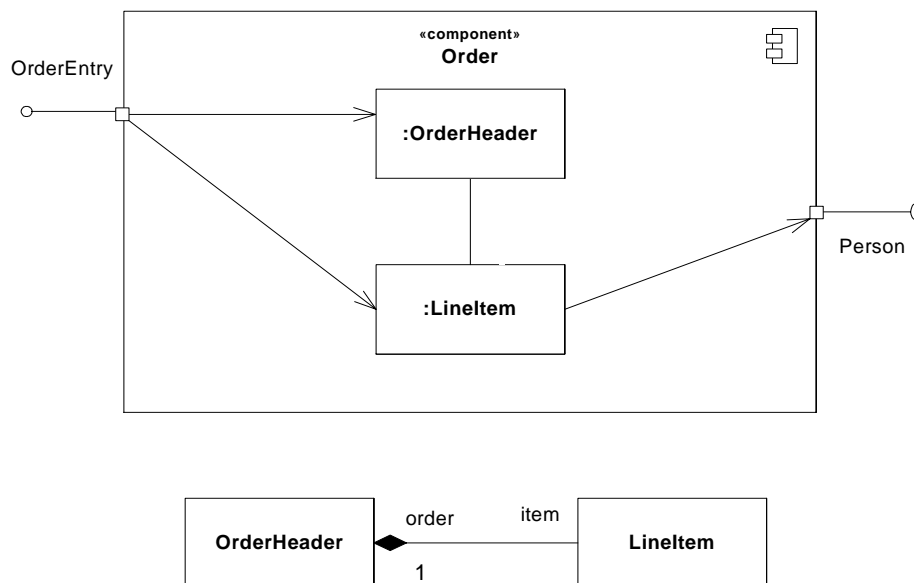
A port may delegate to a set of ports on subordinate components. In that case, these subordinate ports must collectively offer the delegated functionality of the delegating port. At execution time, signals will be delivered to the appropriate port. In the cases where multiple target ports support the handling of the same signal, the signal will be delivered to all these subordinate ports.

The execution time semantics for an assembly connector are that signals travel along an instance of a connector, originating in a required port and delivered to a provided port. Multiple connectors directed from a single required interface or port to provided interfaces on different components indicates that the instance that will handle the signal will be determined at execution time. Similarly, multiple required ports that are connected to a single provided port indicates that the request may originate from instances of different component types.

The interface compatibility between provided and required ports that are connected enables an existing component in a system to be replaced by one that (minimally) offers the same set of services. Also, in contexts where components are used to extend a system by offering existing services, but also adding new functionality, assembly connectors can be used to link in the new component definition. That is, by adding the new component type that offers the same set of services as existing types, and defining new assembly connectors to link up its provided and required ports to existing ports in an assembly.

## Notation

A delegation connector is notated as a Connector from the delegating source Port to the handling target Part, and vice versa for required Interfaces or Ports.

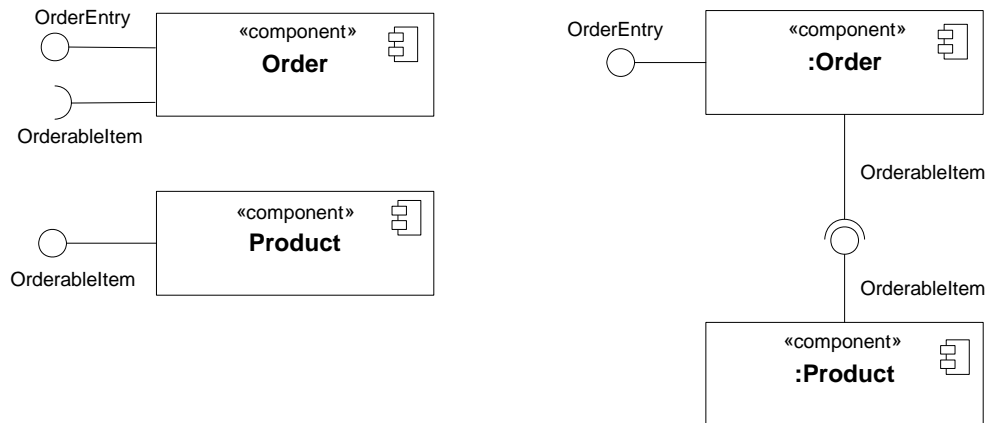


**Figure 8.16 - Delegation connectors connect the externally provided interfaces of a component to the parts that realize or require them.**



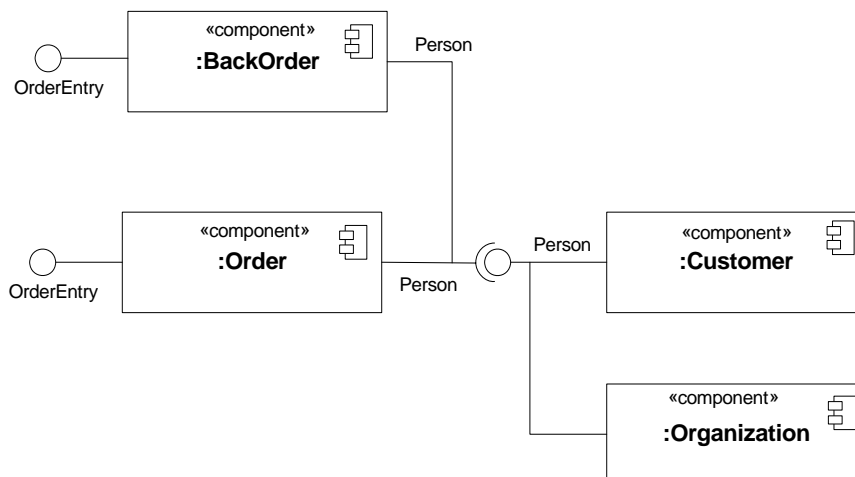
An assembly connector is notated by a “ball-and-socket” connection between a provided interface and a required interface. This notation allows for succinct graphical wiring of components, a requirement for scaling in complex systems.

When this notation is used to connect “complex” ports that are typed by multiple provided and/or required interfaces, the various interfaces are listed as an ordered set, designated with {provided} or {required} if needed.



**Figure 8.17 - An assembly connector maps a required interface of a component to a provided interface of another component in a certain context (definition of components, e.g., in a library on the left, an assembly of those components on the right).**

In a system context where there are multiple components that provide or require a particular interface, a notation abstraction can be used that combines by joining the multiple connectors. This abstraction is similar to the one defined for aggregation and subtyping relationships.



Note: Client interface is a subtype of Person interface

**Figure 8.18 - As a notation abstraction, multiple wiring relationships can be visually grouped together in a component assembly.**

## Changes from previous UML

The following changes from UML 1.x have been made — Connector is not defined in UML 1.4.

### 8.3.3 ConnectorKind (from BasicComponents)

#### Generalizations

None

#### Description

ConnectorKind is an enumeration of the following literal values:

- assembly            Indicates that the connector is an assembly connector.
- delegation        Indicates that the connector is a delegation connector.

### 8.3.4 Realization (from BasicComponents)

The Realization concept is specialized in the Components package to (optionally) define the Classifiers that realize the contract offered by a component in terms of its provided and required interfaces. The component forms an abstraction from these various Classifiers.

#### Generalizations

- “Realization (from Dependencies)” on page 124 (*merge increment*)

#### Description

In the metamodel, a Realization is a subtype of Dependencies::Realization.

#### Attributes

No additional attributes

#### Associations

No additional associations

#### Constraints

No additional constraints

#### Semantics

A component’s behavior may typically be realized (or implemented) by a number of Classifiers. In effect, it forms an abstraction for a collection of model elements. In that case, a component owns a set of Realization Dependencies to these Classifiers.

It should be noted that for the purpose of applications that require multiple different sets of realizations for a single component specification, a set of standard stereotypes are defined in the UML Standard Profile. In particular, «specification» and «realization» are defined there for this purpose.

## Notation

A component realization is notated in the same way as the realization dependency (i.e., as a general dashed line with an open arrow-head).

## Changes from previous UML

The following changes from UML 1.x have been made: Realization is defined in UML 1.4 as a ‘free standing’ general dependency - it is not extended to cover component realization specifically. These semantics have been made explicit in UML 2.0.





## 8.4 Diagrams

### Structure diagram

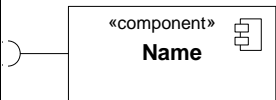
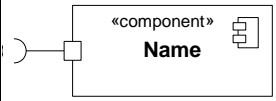
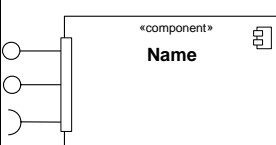
#### Graphical nodes

The graphic nodes that can be included in structure diagrams are shown in Table 8.1.

**Table 8.1 - Graphic nodes included in structure diagrams**

<i>NODE TYPE</i>	<i>NOTATION</i>	<i>REFERENCE</i>
Component	 	See “Component”
Component implements Interface		See “Interface”
Component has provided Port (typed by Interface)		See “Port”

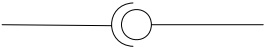
**Table 8.1 - Graphic nodes included in structure diagrams**

<i>NODE TYPE</i>	<i>NOTATION</i>	<i>REFERENCE</i>
Component uses Interface		See “Interface”
Component has required Port (typed by Interface)		See “Port”
Component has complex Port (typed by provided and required Interfaces)		See “Port”

### Graphical paths

The graphic paths that can be included in structure diagrams are shown in Table 8.2.

**Table 8.2 - Graphic nodes included in structure diagrams**

<i>PATH TYPE</i>	<i>NOTATION</i>	<i>REFERENCE</i>
Assembly connector		See “assembly connector.” Also used as notation option for wiring between interfaces using Dependencies.

### Variations

Variations of structure diagrams often focus on particular structural aspects, such as relationships between packages, showing instance specifications, or relationships between classes. There are no strict boundaries between different variations; it is possible to display any element you normally display in a structure diagram in any variation.

#### Component diagram

The following nodes and edges are typically drawn in a component diagram:

- Component
- Interface

- Realization, Interface Realization, Usage Dependencies
- Class
- Artifact
- Port

## 9 Composite Structures

### 9.1 Overview

The term “structure” in this chapter refers to a composition of interconnected elements, representing run-time instances collaborating over communications links to achieve some common objectives.

#### Internal Structures

The `InternalStructure` subpackage provides mechanisms for specifying structures of interconnected elements that are created within an instance of a containing classifier. A structure of this type represents a decomposition of that classifier and is referred to as its “internal structure.”

#### Ports

The `Ports` subpackage provides mechanisms for isolating a classifier from its environment. This is achieved by providing a point for conducting interactions between the internals of the classifier and its environment. This interaction point is referred to as a “port.” Multiple ports can be defined for a classifier, enabling different interactions to be distinguished based on the port through which they occur. By decoupling the internals of the classifier from its environment, ports allow a classifier to be defined independently of its environment, making that classifier reusable in any environment that conforms to the interaction constraints imposed by its ports.

#### Collaborations

Objects in a system typically cooperate with each other to produce the behavior of a system. The behavior is the functionality that the system is required to implement.

A behavior of a collaboration will eventually be exhibited by a set of cooperating instances (specified by classifiers) that communicate with each other by sending signals or invoking operations. However, to understand the mechanisms used in a design, it may be important to describe only those aspects of these classifiers and their interactions that are involved in accomplishing a task or a related set of tasks, projected from these classifiers. *Collaborations* allow us to describe only the relevant aspects of the cooperation of a set of instances by identifying the specific roles that the instances will play. *Interfaces* allow the externally observable properties of an instance to be specified without determining the classifier that will eventually be used to specify this instance. Consequentially, the roles in a collaboration will often be typed by interfaces and will then prescribe properties that the participating instances must exhibit, but will not determine what class will specify the participating instances.

#### StructuredClasses

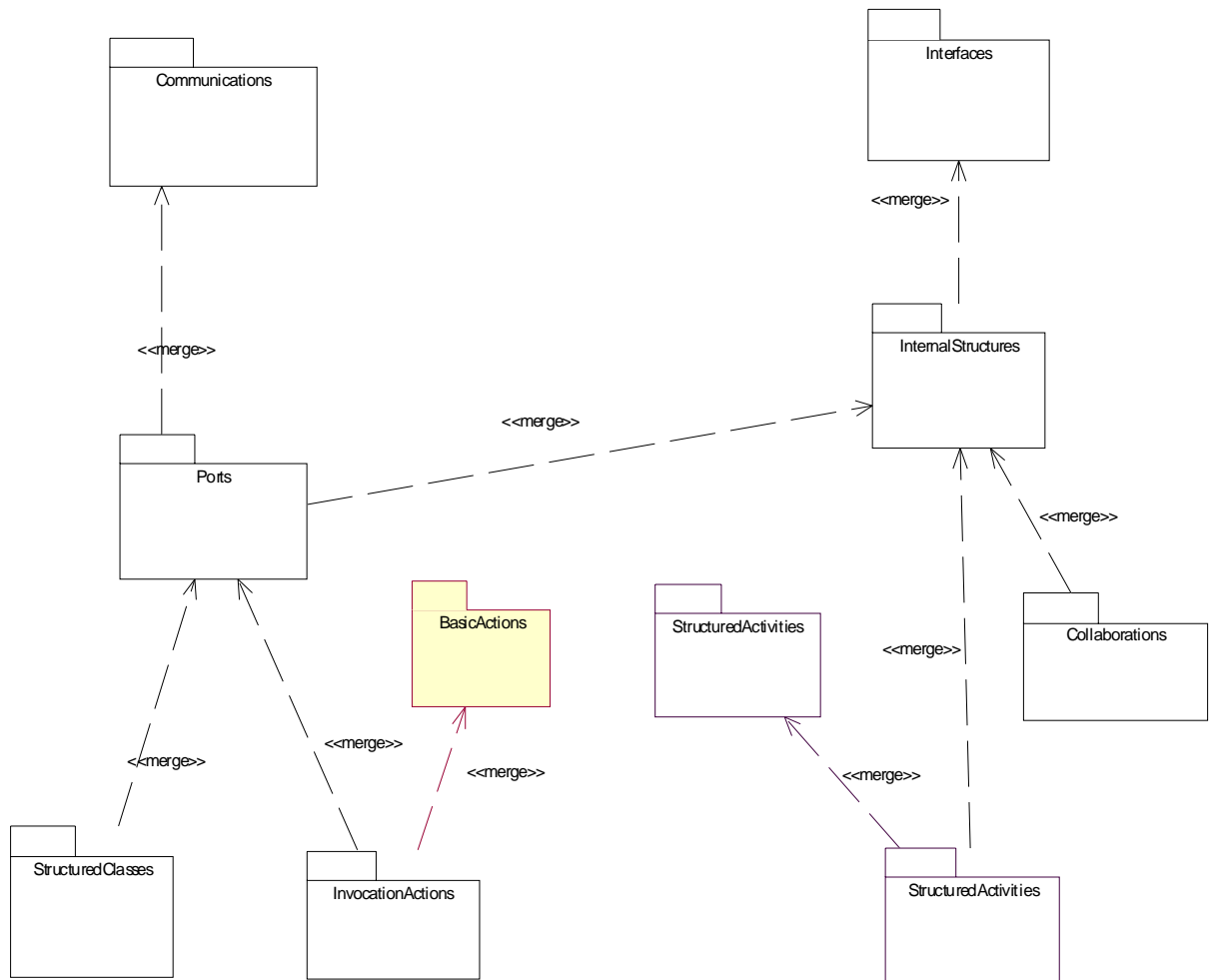
The `StructuredClasses` subpackage supports the representation of classes that may have ports as well as internal structure.

#### Actions

The `Actions` subpackage adds actions that are specific to the features introduced by composite structures (e.g., the sending of messages via ports).

### 9.2 Abstract syntax

Figure 9.1 shows the dependencies of the `CompositeStructures` packages.



**Figure 9.1 - Dependencies between packages described in this chapter**

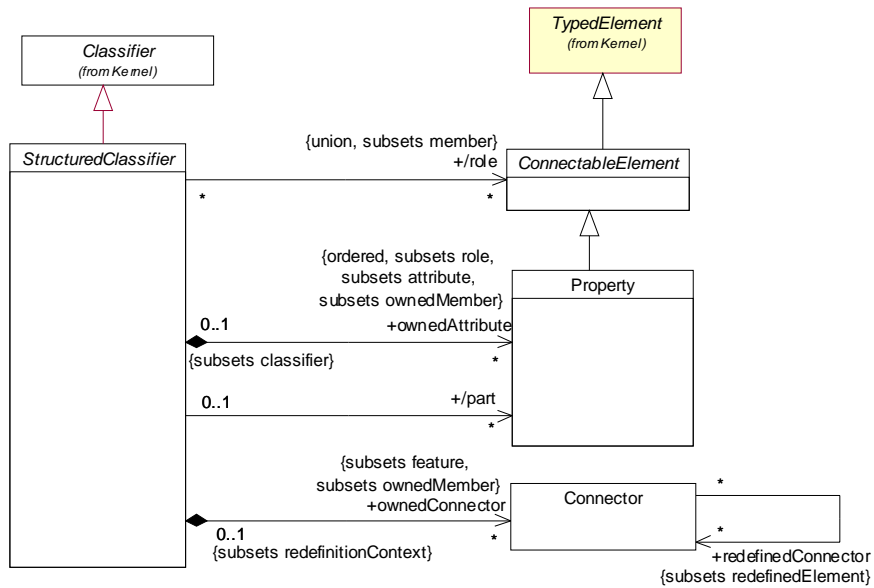


Figure 9.2 - Structured classifier

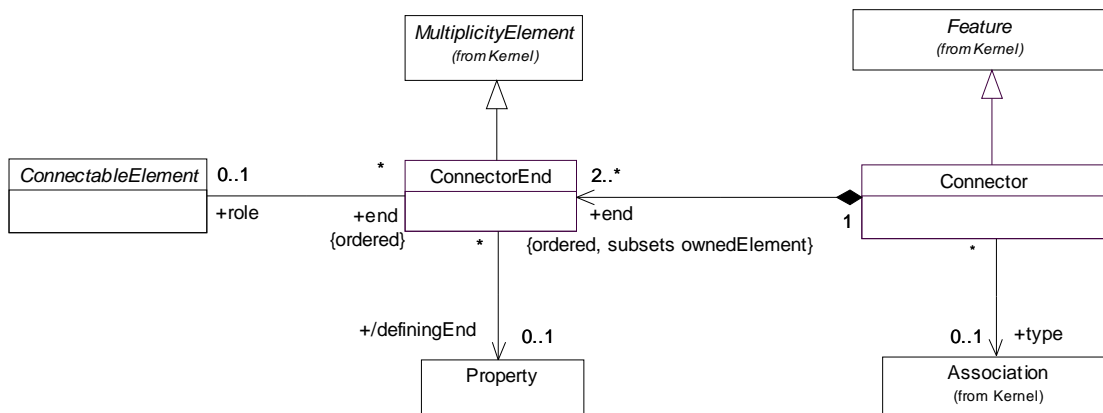
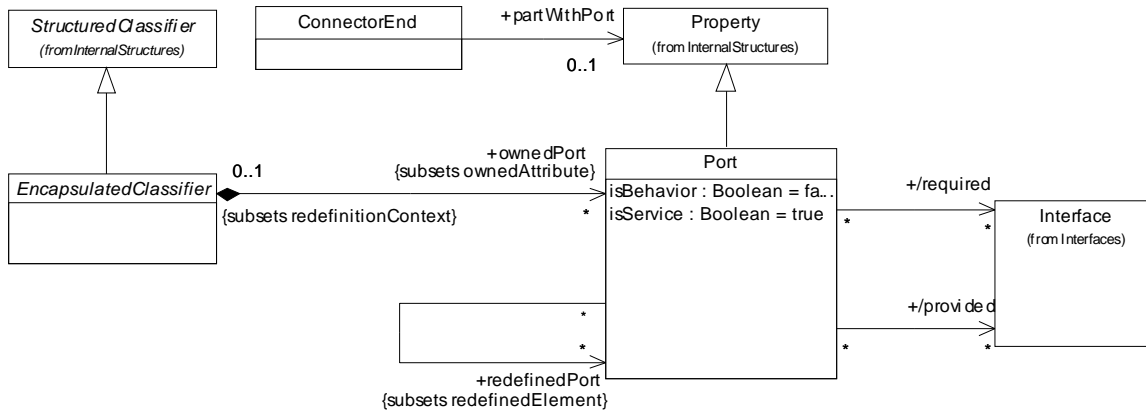


Figure 9.3 - Connectors

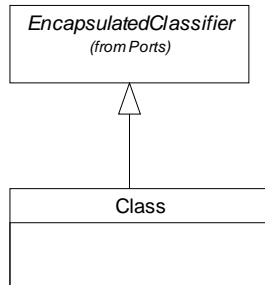


## Package Ports



**Figure 9.4 - The Port metaclass**

## Package StructuredClasses



**Figure 9.5 - Classes with internal structure**

## Package Collaborations

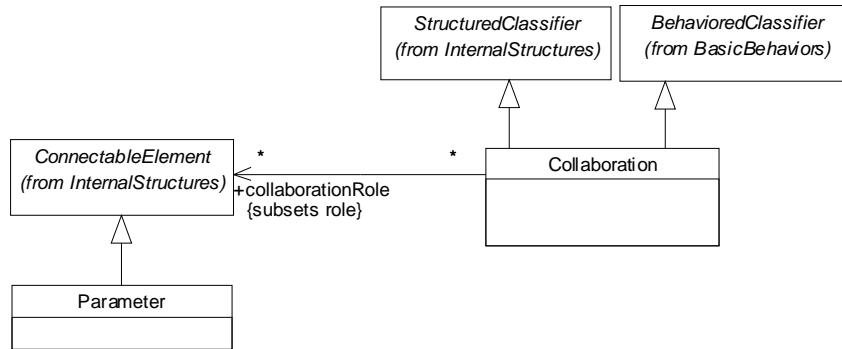


Figure 9.6 - Collaboration

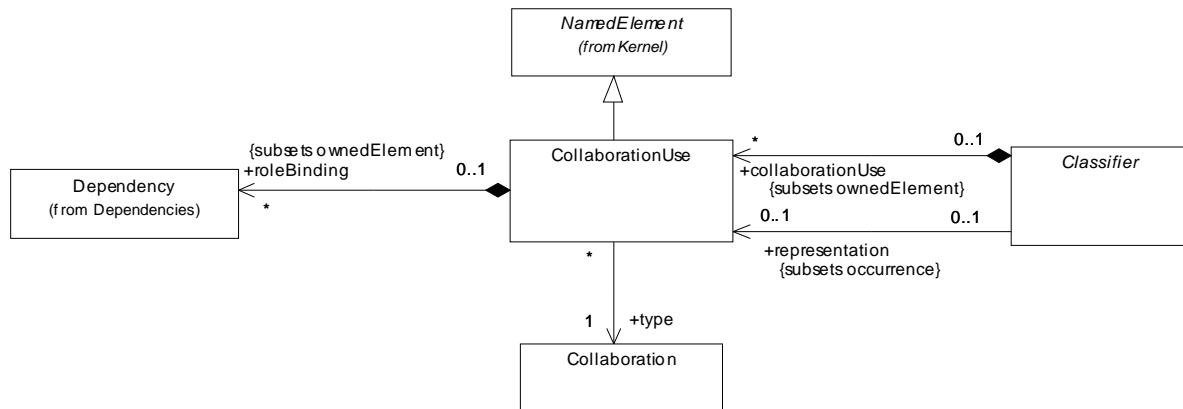
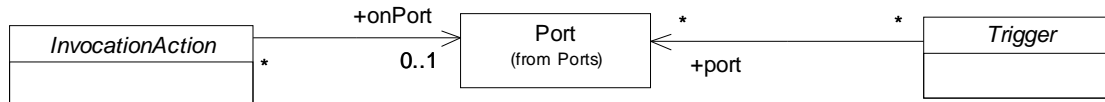


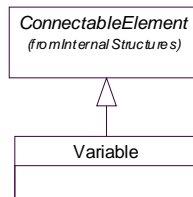
Figure 9.7 - Collaboration.use and role binding

*Package InvocationActions*



**Figure 9.8 - Actions specific to composite structures**

*Package StructuredActivities*



**Figure 9.9 - Extension to Variable**

## 9.3 Class Descriptions

### 9.3.1 Class (from StructuredClasses)

#### Generalizations

- “EncapsulatedClassifier (from Ports)” on page 173.

#### Description

Extends the metaclass *Class* with the capability to have an internal structure and ports.

#### Semantics

See “Property (from InternalStructures)” on page 179, “Connector (from InternalStructures)” on page 170, and “Port (from Ports)” on page 175 for the semantics of the features of *Class*. Initialization of the internal structure of a class is discussed in section “StructuredClassifier” on page 179.

A class acts as the namespace for various kinds of classifiers defined within its scope, including classes. Nesting of classifiers limits the visibility of the classifier to within the scope of the namespace of the containing class and is used for reasons of information hiding. Nested classifiers are used like any other classifier in the containing class.

#### Notation

See “Class (from Kernel)” on page 45, “StructuredClassifier” on page 179, and “Port” on page 175.

## Presentation Options

A dashed arrow with an open arrowhead, optionally labeled with the keyword «create», may be used to relate an instance value to a constructor for a class, describing the single value returned by the constructor, which must have the class as its classifier. The arrowhead points at the operation defining the constructor. The constructor may reference parameters declared by the operation. A constructor is any operation having a single return result parameter of the type of the owning class. The instance value at the base of the arrow represents the default value of the single return result parameter of a constructor.

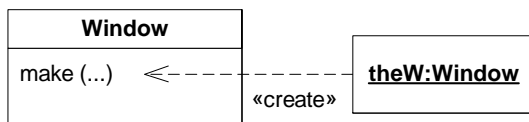


Figure 9.10 - Instance specification describes the return value of an operation

## Changes from previous UML

Class has been extended with internal structure and ports.

### 9.3.2 Classifier (from Collaborations)

#### Generalizations

- 7.3.8, “Classifier (from Kernel, Dependencies, PowerTypes),” on page 48

#### Description

Classifier is extended with the capability to own collaboration uses. These collaboration uses link a collaboration with the classifier to give a description of the workings of the classifier.

#### Associations

- collaborationUse: CollaborationUse References the collaboration uses owned by the classifier. (Subsets *Element.ownedElement*)
- representation: CollaborationUse [0..1] References a collaboration use which indicates the collaboration that represents this classifier. (Subsets *Classifier.occurrence*)

#### Semantics

A classifier can own collaboration uses that relate (aspects of) this classifier to a collaboration. The collaboration describes those aspects of this classifier.

One of the collaboration uses owned by a classifier may be singled out as representing the behavior of the classifier as a whole. The collaboration that is related to the classifier by this collaboration use shows how the instances corresponding to the structural features of this classifier (e.g., its attributes and parts) interact to generate the overall behavior of the classifier. The representing collaboration may be used to provide a description of the behavior of the classifier at a different level of abstraction than is offered by the internal structure of the classifier. The properties of the classifier are mapped to roles in the collaboration by the role bindings of the collaboration use.

## Notation

See “CollaborationUse (from Collaborations)” on page 166

## Changes from previous UML

Replaces and widens the applicability of *Collaboration.usedCollaboration*. Together with the newly introduced concept of internal structure replaces *Collaboration.representedClassifier*.

### 9.3.3 Collaboration (from Collaborations)

A collaboration describes a structure of collaborating elements (roles), each performing a specialized function, which collectively accomplish some desired functionality. Its primary purpose is to explain how a system works and, therefore, it typically only incorporates those aspects of reality that are deemed relevant to the explanation. Thus, details, such as the identity or precise class of the actual participating instances are suppressed.

#### Generalizations

- “BehavioredClassifier (from BasicBehaviors, Communications)” on page 419
- “StructuredClassifier (from InternalStructures)” on page 182

#### Description

A collaboration is represented as a kind of classifier and defines a set of cooperating entities to be played by instances (its roles), as well as a set of connectors that define communication paths between the participating instances. The cooperating entities are the properties of the collaboration (see “Property (from InternalStructures)” on page 179).

A collaboration specifies a view (or projection) of a set of cooperating classifiers. It describes the required links between instances that play the roles of the collaboration, as well as the features required of the classifiers that specify the participating instances. Several collaborations may describe different projections of the same set of classifiers.

#### Attributes

No additional attributes

#### Associations

- collaborationRole: ConnectableElement [\*]      References connectable elements (possibly owned by other classifiers), which represent roles that instances may play in this collaboration. (Subsets *StructuredClassifier.role*)

#### Constraints

No additional constraints

#### Semantics

Collaborations are generally used to explain how a collection of cooperating instances achieve a joint task or set of tasks. Therefore, a collaboration typically incorporates only those aspects that are necessary for its explanation and suppresses everything else. Thus, a given object may be simultaneously playing roles in multiple different collaborations, but each collaboration would only represent those aspects of that object that are relevant to its purpose.

A collaboration defines a set of cooperating participants that are needed for a given task. The roles of a collaboration will be played by instances when interacting with each other. Their relationships relevant for the given task are shown as connectors between the roles. Roles of collaborations define a usage of instances, while the classifiers typing these roles specify all required properties of these instances. Thus, a collaboration specifies what properties instances must have to be able to participate in the collaboration. A role specifies (through its type) the required set of features a participating instance must have. The connectors between the roles specify what communication paths must exist between the participating instances.

Neither all features nor all contents of the participating instances nor all links between these instances are always required in a particular collaboration. Therefore, a collaboration is often defined in terms of roles typed by interfaces (see “Interface (from Interfaces)” on page 82). An interface is a description of a set of properties (externally observable features) required or provided by an instance. An interface can be viewed as a projection of the externally observable features of a classifier realizing the interface. Instances of different classifiers can play a role defined by a given interface, as long as these classifiers realize the interface (i.e., have all the required properties). Several interfaces may be realized by the same classifier, even in the same context, but their features may be different subsets of the features of the realizing classifier.

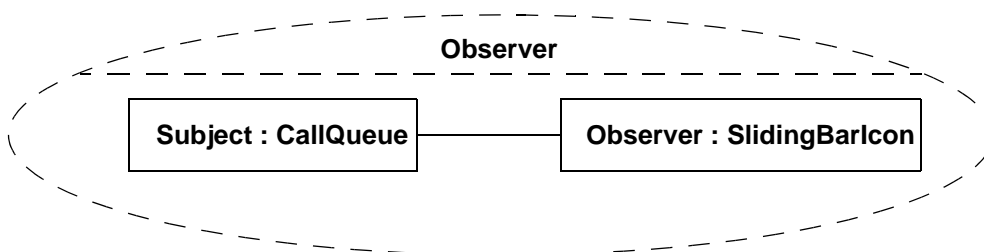
Collaborations may be specialized from other collaborations. If a role is extended in the specialization, the type of a role in the specialized collaboration must conform to the type of the role in the general collaboration. The specialization of the types of the roles does not imply corresponding specialization of the classifiers that realize those roles. It is sufficient that they conform to the constraints defined by those roles.

A collaboration may be attached to an operation or a classifier through a CollaborationUse. A collaboration used in this way describes how this operation or this classifier is realized by a set of cooperating instances. The connectors defined within the collaboration specify links between the instances when they perform the behavior specified in the classifier. The collaboration specifies the context in which behavior is performed. Such a collaboration may constrain the set of valid interactions that may occur between the instances that are connected by a link.

A collaboration is not directly instantiable. Instead, the cooperation defined by the collaboration comes about as a consequence of the actual cooperation between the instances that play the roles defined in the collaboration (the collaboration is a selective view of that situation).

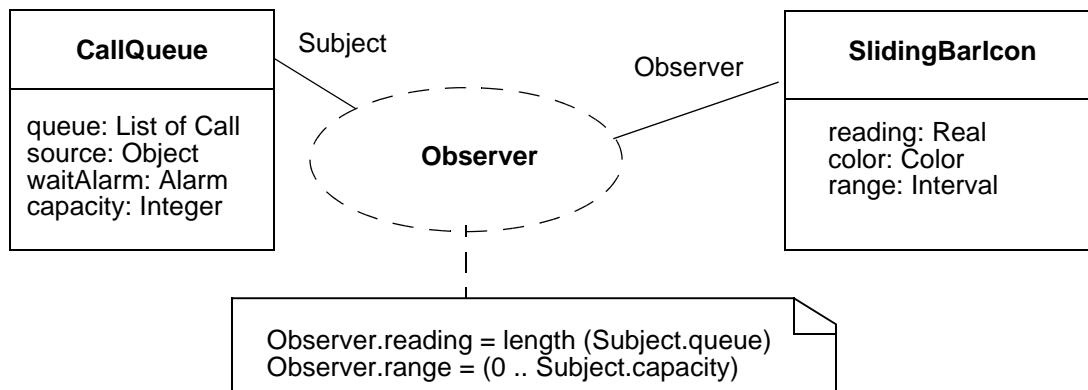
## Notation

A collaboration is shown as a dashed ellipse icon containing the name of the collaboration. The internal structure of a collaboration as comprised by roles and connectors may be shown in a compartment within the dashed ellipse icon. Alternatively, a composite structure diagram can be used.



**Figure 9.11 - The internal structure of the Observer collaboration shown inside the collaboration icon (a connection is shown between the Subject and the Observer role).**

Using an alternative notation for properties, a line may be drawn from the collaboration icon to each of the symbols denoting classifiers that are the types of properties of the collaboration. Each line is labeled by the name of the property. In this manner, a collaboration icon can show the use of a collaboration together with the actual classifiers that occur in that particular use of the collaboration (see Figure 9.12).



**Figure 9.12 - In the Observer collaboration two roles, a Subject and an Observer, collaborate to produce the desired behavior. Any instance playing the Subject role must possess the properties specified by CallQueue, and similarly for the Observer role.**

## Rationale

The primary purpose of collaborations is to explain how a system of communicating entities collectively accomplish a specific task or set of tasks without necessarily having to incorporate detail that is irrelevant to the explanation. It is particularly useful as a means for capturing standard design patterns.

## Changes from previous UML

The contents of a collaboration is specified as its internal structure relying on roles and connectors; the concepts of ClassifierRole, AssociationRole, and AssociationEndRole have been superseded. A collaboration in UML 2.0 is a kind of classifier, and can have any kind of behavioral descriptions associated. There is no loss in modeling capabilities.

### 9.3.4 CollaborationUse (from Collaborations)

A collaboration use represents the application of the pattern described by a collaboration to a specific situation involving specific classes or instances playing the roles of the collaboration.

## Generalizations

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

## Description

A collaboration use represents one particular use of a collaboration to explain the relationships between the properties of a classifier. A collaboration use shows how the pattern described by a collaboration is applied in a given context, by binding specific entities from that context to the roles of the collaboration. Depending on the context, these entities could

be structural features of a classifier, instance specifications, or even roles in some containing collaboration. There may be multiple occurrences of a given collaboration within a classifier, each involving a different set of roles and connectors. A given role or connector may be involved in multiple occurrences of the same or different collaborations.

Associated dependencies map features of the collaboration type to features in the classifier. These dependencies indicate which role in the classifier plays which role in the collaboration.

### Attributes

No additional attributes

### Associations

- **type:** Collaboration [1]      The collaboration that is used in this occurrence. The collaboration defines the cooperation between its roles that are mapped to properties of the classifier owning the collaboration use.
- **roleBinding:** Dependency [\*]      A mapping between features of the collaboration type and features of the classifier or operation. This mapping indicates which connectable element of the classifier or operation plays which role(s) in the collaboration. A connectable element may be bound to multiple roles in the same collaboration use (that is, it may play multiple roles).

### Constraints

- [1] All the client elements of a *roleBinding* are in one classifier and all supplier elements of a *roleBinding* are in one collaboration and they are compatible.
- [2] Every role in the collaboration is bound within the collaboration use to a connectable element within the classifier or operation.
- [3] The connectors in the classifier connect according to the connectors in the collaboration.

### Semantics

A collaboration use relates a feature in its collaboration type to a connectable element in the classifier or operation that owns the collaboration use.

Any behavior attached to the collaboration type applies to the set of roles and connectors bound within a given collaboration use. For example, an interaction among parts of a collaboration applies to the classifier parts bound to a single collaboration use. If the same connectable element is used in both the collaboration and the represented element, no role binding is required.

### Semantic Variation Points

It is a semantic variation when client and supplier elements in role bindings are compatible.

### Notation

A collaboration use is shown by a dashed ellipse containing the name of the occurrence, a colon, and the name of the collaboration type. For every role binding, there is a dashed line from the ellipse to the client element; the dashed line is labeled on the client end with the name of the supplier element.



## Presentation Options

A dashed arrow with an open arrowhead may be used to show that a collaboration is used in a classifier, optionally labeled with the keyword «represents». A dashed arrow with an open arrowhead may also be used to show that a collaboration represents a classifier, optionally labeled with the keyword «occurrence». The arrowhead points at the owning classifier. When using this presentation option, the role bindings are shown explicitly as dependencies.

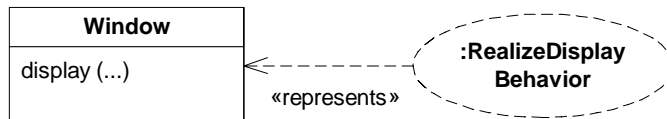


Figure 9.13 - Collaboration occurrence relates a classifier to a collaboration

## Examples

This example shows the definition of two collaborations, *Sale* (Figure 9.14) and *BrokeredSale* (Figure 9.15). *Sale* is used twice as part of the definition of *BrokeredSale*. *Sale* is a collaboration among two roles, a *seller* and a *buyer*. An interaction, or other behavior specification, could be attached to *Sale* to specify the steps involved in making a *Sale*.

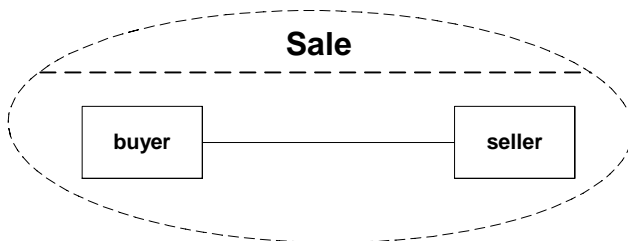
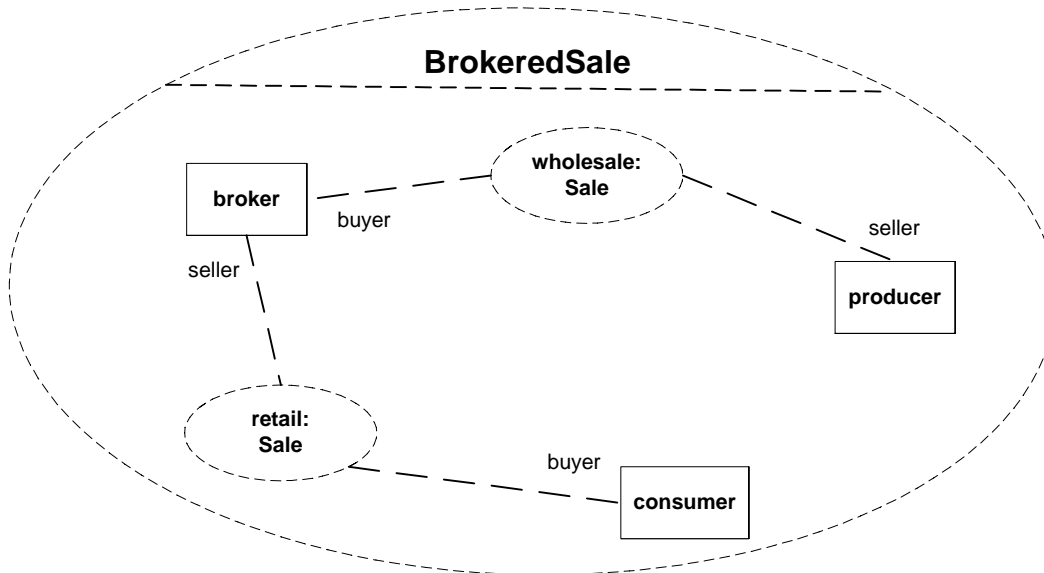


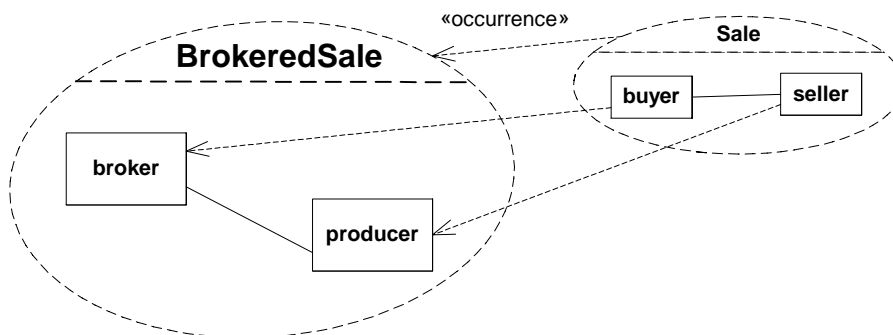
Figure 9.14 - The *Sale* collaboration

*BrokeredSale* is a collaboration among three roles, a *producer*, a *broker*, and a *consumer*. The specification of *BrokeredSale* shows that it consists of two occurrences of the *Sale* collaboration, indicated by the dashed ellipses. The occurrence *wholesale* indicates a *Sale* in which the *producer* is the *seller* and the *broker* is the *buyer*. The occurrence *retail* indicates a *Sale* in which the *broker* is the *seller* and the *consumer* is the *buyer*. The connectors between *sellers* and *buyers* are not shown in the two occurrences; these connectors are implicit in the *BrokeredSale* collaboration in virtue of them being comprised of *Sale*. The *BrokeredSale* collaboration could itself be used as part of a larger collaboration.



**Figure 9.15 - The BrokeredSale collaboration**

Figure 9.16 shows part of the *BrokeredSale* collaboration in a presentation option.



**Figure 9.16 - A subset of the BrokeredSale collaboration**

### Rationale

A collaboration use is used to specify the application of a pattern specified by a collaboration to a specific situation. In that regard, it acts as the invocation of a macro with specific values used for the parameters (roles).

### Changes from previous UML

This metaclass has been added.

### 9.3.5 ConnectableElement (from InternalStructures)

#### Generalizations

- “TypedElement (from Kernel)” on page 131

#### Description

A ConnectableElement is an abstract metaclass representing a set of instances that play roles of a classifier. Connectable elements may be joined by attached connectors and specify configurations of linked instances to be created within an instance of the containing classifier.

#### Attributes

No additional attributes

#### Associations

- end: ConnectorEnd Denotes a connector that attaches to this connectable element.

#### Constraints

No additional constraints

#### Semantics

The semantics of ConnectableElement is given by its concrete subtypes.

#### Notation

None

#### Rationale

This metaclass supports factoring out the ability of a model element to be linked by a connector.

#### Changes from previous UML

This metaclass generalizes the concept of classifier role from 1.x.

### 9.3.6 Connector (from InternalStructures)

Specifies a link that enables communication between two or more instances. This link may be an instance of an association, or it may represent the possibility of the instances being able to communicate because their identities are known by virtue of being passed in as parameters, held in variables or slots, or because the communicating instances are the same instance. The link may be realized by something as simple as a pointer or by something as complex as a network connection. In contrast to associations, which specify links between any instance of the associated classifiers, connectors specify links between instances playing the connected parts only.

#### Generalizations

- “Feature (from Kernel)” on page 66

## Description

Each connector may be attached to two or more connectable elements, each representing a set of instances. Each connector end is distinct in the sense that it plays a distinct role in the communication realized over a connector. The communications realized over a connector may be constrained by various constraints (including type constraints) that apply to the attached connectable elements.

## Attributes

No additional attributes

## Associations

- **end: ConnectorEnd [2..\*]**  
A connector consists of at least two connector ends, each representing the participation of instances of the classifiers typing the connectable elements attached to this end. The set of connector ends is ordered.  
(Subsets *Element.ownedElement*)
- **type: Association [0..1]**  
An optional association that specifies the link corresponding to this connector.
- **redefinedConnector: Connector [0..\*]**  
A connector may be redefined when its containing classifier is specialized. The redefining connector may have a type that specializes the type of the redefined connector. The types of the connector ends of the redefining connector may specialize the types of the connector ends of the redefined connector. The properties of the connector ends of the redefining connector may be replaced. (Subsets *Element.redefinedElement*)

## Constraints

- [1] The types of the connectable elements that the ends of a connector are attached to must conform to the types of the association ends of the association that types the connector, if any.
- [2] The connectable elements attached to the ends of a connector must be compatible.
- [3] The ConnectableElements attached as roles to each ConnectorEnd owned by a Connector must be roles of the Classifier that owned the Connector, or they must be ports of such roles.

## Semantics

If a connector between two roles of a classifier is a feature of an instantiable classifier, it declares that a link may exist within an instance of that classifier. If a connector between two roles of a classifier is a feature of an uninstantiable classifier, it declares that links may exist within an instance of the classifier that realizes the original classifier. These links will connect instances corresponding to the parts joined by the connector.

Links corresponding to connectors may be created upon the creation of the instance of the containing classifier (see “StructuredClassifier” on page 179). The set of links is a subset of the total set of links specified by the association typing the connector. All links are destroyed when the containing classifier instance is destroyed.

If the type of the connector is omitted, the type is inferred based on the connector, as follows: If the type of a role (i.e, the connectable element attached to a connector end) realizes an interface that has a unique association to another interface which is realized by the type of another role (or an interface compatible to that interface is realized by the type of another role), then that association is the type of the connector between these parts. If the connector realizes a collaboration (that is, a collaboration use maps the connector to a connector in an associated collaboration through role bindings), then the type of the connector is an anonymous association with association ends corresponding to each connector end. The type of each association end is the classifier that realizes the parts connected to the matching connector in the collaboration.

Any adornments on the connector ends (either the original connector or the connector in the collaboration) specify adornments of the ends of the inferred association. Otherwise, the type of the connector is an anonymously named association with association ends corresponding to each connector end. The type of each association end is the type of the part that each corresponding connector end is attached to. Any adornments on the connector ends specify adornments of the ends of the inferred association. Any inferred associations are always bidirectionally navigable and are owned by the containing classifier.

## Semantic Variation Points

What makes connectable elements compatible is a semantic variation point.

## Notation

A connector is drawn using the notation for association (see “Association (from Kernel)” on page 36). The optional name string of the connector obeys the following syntax:

`( [ name ] ‘.’ <classname> ) / <name>`

where `<name>` is the name of the connector, and `<classname>` is the name of the association that is its type. A stereotype keyword within guillemets may be placed above or in front of the connector name. A property string may be placed after or below the connector name.

## Examples

Examples are shown in “StructuredClassifier” on page 179.

## Changes from previous UML

Connector has been added in UML 2.0. The UML 1.4 concept of association roles is subsumed by connectors.

### 9.3.7 ConnectorEnd (from InternalStructures, Ports)

#### Generalizations

- “MultiplicityElement (from Kernel)” on page 90

#### Description

A connector end is an endpoint of a connector, which attaches the connector to a connectable element. Each connector end is part of one connector.

#### Attributes

No additional attributes

#### Associations

##### *InternalStructures*

- role: ConnectableElement [1]      The connectable element attached at this connector end. When an instance of the containing classifier is created, a link may (depending on the multiplicities) be created to an instance of the classifier that types this connectable element.

- **definingEnd:** Property [0..1]      A derived association referencing the corresponding association end on the association that types the connector owning this connector end. This association is derived by selecting the association end at the same place in the ordering of association ends as this connector end.

#### *Ports*

- **partWithPort:** Property [0..1]      Indicates the role of the internal structure of a classifier with the port to which the connector end is attached.

### **Constraints**

- [1] If a connector end is attached to a port of the containing classifier, *partWithPort* will be empty.
- [2] If a connector end references both a *role* and a *partWithPort*, then the *role* must be a port that is defined by the type of the *partWithPort*.
- [3] The property held in *self.partWithPort* must not be a Port.

### **Semantics**

#### *InternalStructures*

A connector end describes which connectable element is attached to the connector owning that end. Its multiplicity indicates the number of instances that may be linked to each instance of the property connected on the other end.

### **Notation**

#### *InternalStructures*

Adornments may be shown on the connector end corresponding to adornments on association ends (see “Association (from Kernel)” on page 36). These adornments specify properties of the association typing the connector. The multiplicity indicates the number of instances that may be connected to each instance of the role on the other end. If no multiplicity is specified, the multiplicity matches the multiplicity of the role the end is attached to.

#### *Ports*

If the end is attached to a port on a part of the internal structure and no multiplicity is specified, the multiplicity matches the multiplicity of the port multiplied by the multiplicity of the part (if any).

### **Changes from previous UML**

Connector end has been added in UML 2.0. The UML 1.4 concept of association end roles is subsumed by connector ends.

## **9.3.8 EncapsulatedClassifier (from Ports)**

### **Generalizations**

- “StructuredClassifier (from InternalStructures)” on page 182

### **Description**

Extends a classifier with the ability to own ports as specific and type checked interaction points.

## Attributes

No additional attributes

## Associations

- ownedPort: Port      References a set of ports that an encapsulated classifier owns. (Subsets *Classifier.feature* and *Namespace.ownedMember*)

## Constraints

No additional constraints

## Semantics

See “Port” on page 175.

## Notation

See “Port” on page 175.

## Changes from previous UML

This metaclass has been added to UML.

### 9.3.9 InvocationAction (from Actions)

#### Generalizations

- “InvocationAction (from BasicActions)” on page 249 (*merge increment*)

#### Description

In addition to targeting an object, invocation actions can also invoke behavioral features on ports from where the invocation requests are routed onwards on links deriving from attached connectors. Invocation actions may also be sent to a target via a given port, either on the sending object or on another object.

#### Associations

- onPort: Port [0..1]      An optional port of the receiver object on which the behavioral feature is invoked.

#### Constraints

[1] The *onPort* must be a port on the receiver object.

#### Semantics

The target value of an invocation action may also be a port. In this case, the invocation request is sent to the object owning this port as identified by the port identity, and is, upon arrival, handled as described in “Port” on page 175.

#### Notation

The optional port is identified by the phrase “via <port>” in the name string of the icon denoting the particular invocation action (for example, see “CallOperationAction (from BasicActions)” on page 239).

### 9.3.10 Parameter (from Collaborations)

#### Generalizations

- “ConnectableElement (from InternalStructures)” on page 170
- “Parameter (from Kernel, AssociationClasses)” on page 115 (*merge increment*)

#### Description

Parameters are allowed to be treated as connectable elements.

#### Constraints

[1] A parameter may only be associated with a connector end within the context of a collaboration.

### 9.3.11 Port (from Ports)

A port is a property of a classifier that specifies a distinct interaction point between that classifier and its environment or between the (behavior of the) classifier and its internal parts. Ports are connected to properties of the classifier by connectors through which requests can be made to invoke the behavioral features of a classifier. A Port may specify the services a classifier provides (offers) to its environment as well as the services that a classifier expects (requires) of its environment.

#### Generalizations

- “Property (from InternalStructures)” on page 179

#### Description

Ports represent interaction points between a classifier and its environment. The interfaces associated with a port specify the nature of the interactions that may occur over a port. The required interfaces of a port characterize the requests that may be made from the classifier to its environment through this port. The provided interfaces of a port characterize requests to the classifier that its environment may make through this port.

A port has the ability to specify that any requests arriving at this port are handled by the behavior of the instance of the owning classifier, rather than being forwarded to any contained instances, if any.

#### Attributes

- **isService:** Boolean  
If *true*, indicates that this port is used to provide the published functionality of a classifier. If *false*, this port is used to implement the classifier but is not part of the essential externally-visible functionality of the classifier and can, therefore, be altered or deleted along with the internal implementation of the classifier and other properties that are considered part of its implementation. The default value for this attribute is *true*.
- **isBehavior:** Boolean  
Specifies whether requests arriving at this port are sent to the classifier behavior of this classifier (see “BehavioredClassifier (from BasicBehaviors, Communications)” on page 419). Such ports are referred to as *behavior port*. Any invocation of a behavioral feature targeted at a behavior port will be handled by the instance of the owning classifier itself, rather than by any instances that this classifier may contain. The default value is *false*.



## Associations

- **required: Interface**  
References the interfaces specifying the set of operations and receptions that the classifier expects its environment to handle. This association is derived as the set of interfaces required by the type of the port or its supertypes.
- **provided: Interface**  
References the interfaces specifying the set of operations and receptions that the classifier offers to its environment, and which it will handle either directly or by forwarding it to a part of its internal structure. This association is derived from the interfaces realized by the type of the port or by the type of the port, if the port was typed by an interface.
- **redefinedPort: Port**  
A port may be redefined when its containing classifier is specialized. The redefining port may have additional interfaces to those that are associated with the redefined port or it may replace an interface by one of its subtypes. (Subsets *Element.redefinedElement*)

## Constraints

- [1] The required interfaces of a port must be provided by elements to which the port is connected.
- [2] Port.aggregation must be *composite*.
- [3] When a port is destroyed, all connectors attached to this port will be destroyed also.
- [4] A defaultValue for port cannot be specified when the type of the Port is an Interface.

## Semantics

A port represents an interaction point between a classifier instance and its environment or between a classifier instance and instances it may contain. A port by default has public visibility. However, a behavior port may be hidden but does not have to be.

The required interfaces characterize services that the owning classifier expects from its environment and that it may access through this interaction point: Instances of this classifier expect that the features owned by its required interfaces will be offered by one or more instances in its environment. The provided interfaces characterize the behavioral features that the owning classifier offers to its environment at this interaction point. The owning classifier must offer the features owned by the provided interfaces.

The provided and required interfaces completely characterize any interaction that may occur between a classifier and its environment at a port with respect to the data communicated at this port and the behaviors that may be invoked through this port. The interfaces do not necessarily establish the exact sequences of interactions across the port. When an instance of a classifier is created, instances corresponding to each of its ports are created and held in the slots specified by the ports, in accordance with its multiplicity. These instances are referred to as “interaction points” and provide unique references. A link from that instance to the instance of the owning classifier is created through which communication is forwarded to the instance of the owning classifier or through which the owning classifier communicates with its environment. It is, therefore, possible for an instance to differentiate between requests for the invocation of a behavioral feature targeted at its different ports. Similarly, it is possible to direct such requests at a port, and the requests will be routed as specified by the links corresponding to connectors attached to this port. (In the following, “requests arriving at a port” shall mean “request occurrences arriving at the interaction point of this instance corresponding to this port.”)

The interaction point object must be an instance of a classifier that realizes the provided interfaces of the port. If the port was typed by an interface, the classifier typing the interaction point object realizes that interface. If the port was typed by a class, the interaction point object will be an instance of that class. The latter case allows elaborate specification of the communication over a port. For example, it may describe that communication is filtered, modified in some way, or routed to other parts depending on its contents as specified by the classifier that types the port.

If connectors are attached to both the port when used on a property within the internal structure of a classifier and the port on the container of an internal structure, the instance of the owning classifier will forward any requests arriving at this port along the link specified by those connectors. If there is a connector attached to only one side of a port, any requests arriving at this port will terminate at this port.

For a behavior port, the instance of the owning classifier will handle requests arriving at this port (as specified in the behavior of the classifier, see Chapter 13, “Common Behaviors”), if this classifier has any behavior. If there is no behavior defined for this classifier, any communication arriving at a behavior port is lost.

## Semantic Variation Points

If several connectors are attached on one side of a port, then any request arriving at this port on a link derived from a connector on the other side of the port will be forwarded on links corresponding to these connectors. It is a semantic variation point whether these requests will be forwarded on all links, or on only one of those links. In the latter case, one possibility is that the link at which this request will be forwarded will be arbitrarily selected among those links leading to an instance that had been specified as being able to handle this request (i.e., this request is specified in a provided interface of the part corresponding to this instance).

## Notation

A port of a classifier is shown as a small square symbol. The name of the port is placed near the square symbol. If the port symbol is placed overlapping the boundary of the rectangle symbol denoting that classifier this port is exposed (i.e., its visibility is *public*). If the port is shown inside the rectangle symbol, then the port is hidden and its visibility is as specified (it is *protected* by default).

A port of a classifier may also be shown as a small square symbol overlapping the boundary of the rectangle symbol denoting a part typed by that classifier (see Figure 9.17). The name of the port is shown near the port; the multiplicity follows the name surrounded by brackets. Name and multiplicity may be elided.

The type of a port may be shown following the port name, separated by colon (“:”). A provided interface may be shown using the “lollipop” notation (see “Interface (from Interfaces)” on page 82) attached to the port. A required interface may be shown by the “socket” notation attached to the port. The presentation options shown there are also applicable to interfaces of ports. Figure 9.17 shows the notation for ports: *p* is a port on the *Engine* class. The provided interface (also its type) of port *p* is *powertrain*. The multiplicity of *p* is “1.” In addition, a required interface, *power*, is shown also. The figure on the left shows the provided interface using the “lollipop” notation, while the figure on the right shows the interface as the type of the port.

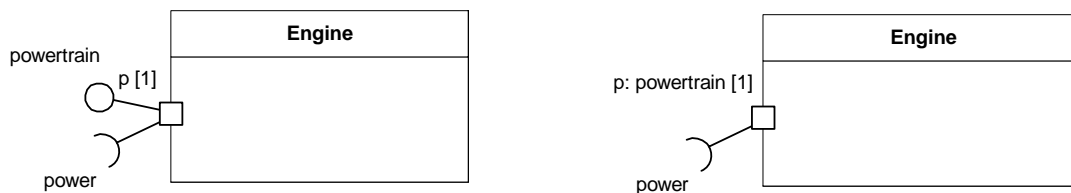
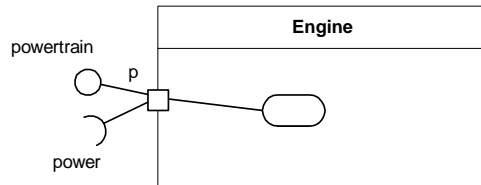


Figure 9.17 - Port notation

A behavior port is indicated by a port being connected through a line to a small state symbol drawn inside the symbol representing the containing classifier. (The small state symbol indicates the behavior of the containing classifier.) Figure 9.18 shows the behavior port *p*, as indicated by its connection to the state symbol representing the behavior of the *Engine* class. Its provided interface is *powertrain*. In addition, a required interface, *power*, is shown also.

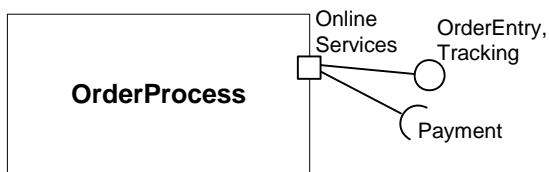


**Figure 9.18 - Behavior port notation**

### Presentation Options

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

If there are multiple interfaces associated with a port, these interfaces may be listed with the interface icon, separated by commas. Figure 9.19 below shows a port *OnlineServices* on the *OrderProcess* class with two provided interfaces, *OrderEntry* and *Tracking*, as well as a required interface *Payment*.



**Figure 9.19 - Port notation showing multiple provided interfaces**

## Examples

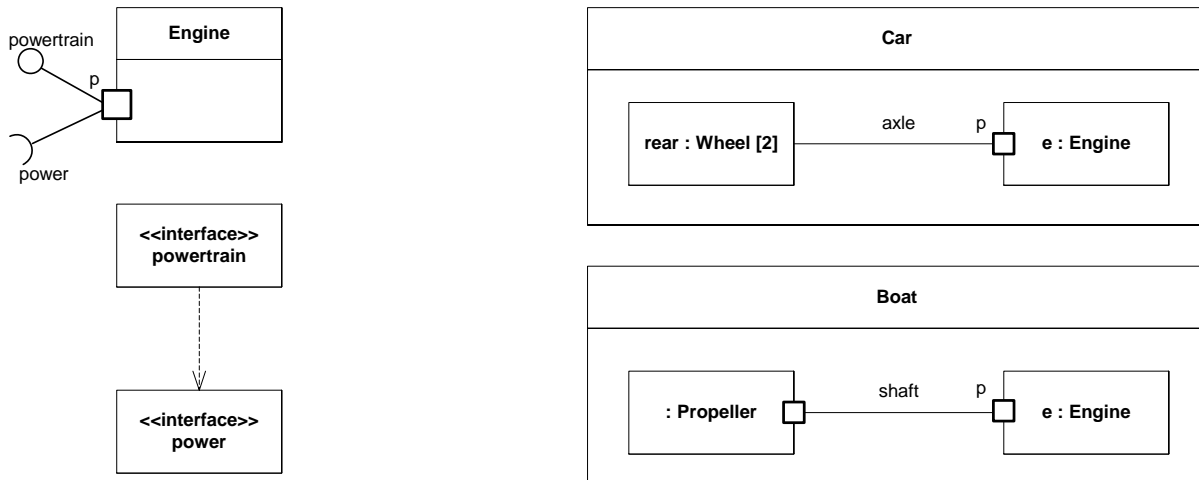


Figure 9.20 - Port examples

Figure 9.20 shows a class *Engine* with a port *p* with a provided interface *powertrain*. This interface specifies the services that the engine offers at this port (i.e., the operations and receptions that are accessible by communication arriving at this port). The interface *power* is the required interface of the engine. The required interface specifies the services that the engine expects its environment to provide. At port *p*, the *Engine* class is completely encapsulated; it can be specified without any knowledge of the environment the engine will be embedded in. As long as the environment obeys the constraints expressed by the provided and required interfaces of the engine, the engine will function properly.

Two uses of the *Engine* class are depicted: Both a boat and a car contain a part that is an engine. The *Car* class connects port *p* of the engine to a set of wheels via the *axle*. The *Boat* class connects port *p* of the engine to a propeller via the *shaft*. As long as the interaction between the *Engine* and the part linked to its port *p* obeys the constraints specified by the provided and required interfaces, the engine will function as specified, whether it is an engine of a car or an engine of a boat. (This example also shows that connectors need not necessarily attach to parts via ports (as shown in the *Car* class.)

## Rationale

The required and provided interfaces of a port specify everything that is necessary for interactions through that interaction point. If all interactions of a classifier with its environment are achieved through ports, then the internals of the classifier are fully isolated from the environment. This allows such a classifier to be used in any context that satisfies the constraints specified by its ports.

## Changes from previous UML

This metaclass has been added to UML.

### 9.3.12 Property (from InternalStructures)

#### Generalizations

- “Property (from Kernel, AssociationClasses)” on page 118 (*merge increment*)

## Description

A property represents a set of instances that are owned by a containing classifier instance.

## Attributes

No additional attributes

## Associations

No additional associations

## Constraints

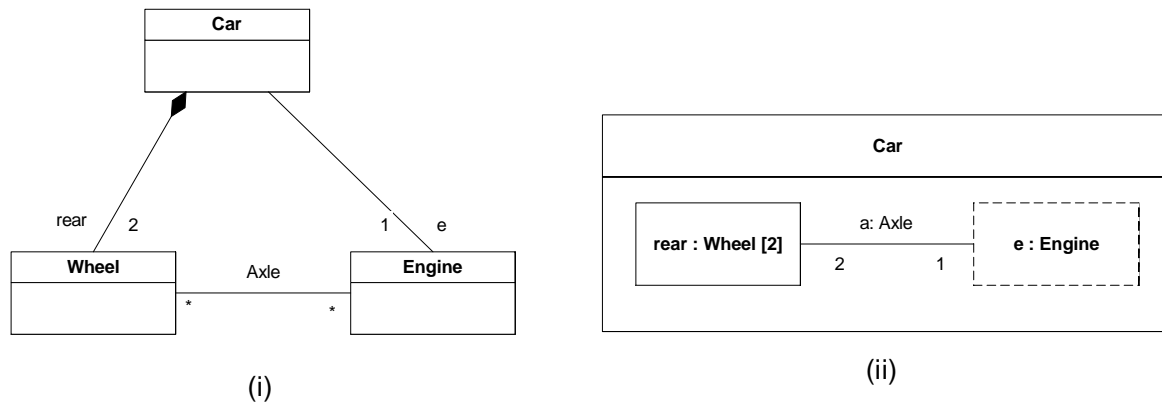
No additional constraints

## Semantics

When an instance of the containing classifier is created, a set of instances corresponding to its properties may be created either immediately or at some later time. These instances are instances of the classifier typing the property. A property specifies that a set of instances may exist; this set of instances is a subset of the total set of instances specified by the classifier typing the property.

A part declares that an instance of this classifier may contain a set of instances by composition. All such instances are destroyed when the containing classifier instance is destroyed. Figure 9.21 shows two possible views of the *Car* class. In subfigure (i), *Car* is shown as having a composition association with role name *rear* to a class *Wheel* and an association with role name *e* to a class *Engine*. In subfigure (ii), the same is specified. However, in addition, in subfigure (ii) it is specified that *rear* and *e* belong to the internal structure of the class *Car*. This allows specification of detail that holds only for instances of the *Wheel* and *Engine* classes within the context of the class *Car*, but which will not hold for wheels and engines in general. For example, subfigure (i) specifies that any instance of class *Engine* can be linked to an arbitrary number of instances of class *Wheel*. Subfigure (ii), however, specifies that within the context of class *Car*, the instance playing the role of *e* may only be connected to two instances playing the role of *rear*. In addition, the instances playing the *e* and *rear* roles may only be linked if they are roles of the same instance of class *Car*.

In other words, subfigure (ii) asserts additional constraints on the instances of the classes *Wheel* and *Engine*, when they are playing the respective roles within an instance of class *Car*. These constraints are not true for instances of *Wheel* and *Engine* in general. Other wheels and engines may be arbitrarily linked as specified in subfigure (i).



**Figure 9.21 - Properties**

### Notation

A part is shown by graphical nesting of a box symbol with a solid outline representing the part within the symbol representing the containing classifier in a separate compartment. A property specifying an instance that is not owned by composition by the instance of the containing classifier is shown by graphical nesting of a box symbol with a dashed outline.

The contained box symbol has only a name compartment, which contains a string according to the syntax defined in the Notation subsection of “Property (from Kernel, AssociationClasses)” on page 118. Detail may be shown within the box symbol indicating specific values for properties of the type classifier when instances corresponding to the property symbol are created.

### Presentation Options

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

A property symbol may be shown containing just a single name (without the colon) in its name string. This implies the definition of an anonymously named class nested within the namespace of the containing class. The part has this anonymous class as its type. Every occurrence of an anonymous class is different from any other occurrence. The anonymously defined class has the properties specified with the part symbol. It is allowed to show compartments defining attributes and operations of the anonymously named class.

## Examples



Figure 9.22 - Property examples

Figure 9.22 shows examples of properties. On the left, the property denotes that the containing instance will own four instances of the *Wheel* class by composition. The multiplicity is shown using the presentation option discussed above. The property on the right denotes that the containing instance will reference one or two instances of the *Engine* class. For additional examples, see 9.3.13, “StructuredClassifier (from InternalStructures),” on page 182.

## Changes from previous UML

A connectable element used in a collaboration subsumes the concept of ClassifierRole.

### 9.3.13 StructuredClassifier (from InternalStructures)

#### Generalizations

- “Classifier (from Kernel, Dependencies, PowerTypes)” on page 48

#### Description

A structured classifier is an abstract metaclass that represents any classifier whose behavior can be fully or partly described by the collaboration of owned or referenced instances.

#### Attributes

No additional attributes

#### Associations

- **role:** ConnectableElement References the roles that instances may play in this classifier. (Abstract union; subsets *Classifier.feature*)
- **ownedAttribute:** Property References the properties owned by the classifier. (Subsets *StructuredClassifier.role*, *Classifier.attribute*, and *Namespace.ownedMember*)
- **part:** Property References the properties specifying instances that the classifier owns by composition. This association is derived, selecting those owned properties where *isComposite* is *true*.
- **ownedConnector:** Connector References the connectors owned by the classifier. (Subsets *Classifier.feature* and *Namespace.ownedMember*)

#### Constraints

[1] The multiplicities on connected elements must be consistent.

## Semantics

The multiplicities on the structural features and connector ends indicate the number of instances (objects and links) that may be created within an instance of the containing classifier, either when the instance of the containing classifier is created, or at a later time. The lower bound of the multiplicity range indicates the number of instances that are created (unless indicated differently by an associated instance specification or an invoked constructor function); the upper bound of the multiplicity range indicates the maximum number of instances that may be created. The slots corresponding to the structural features are initialized with these instances.

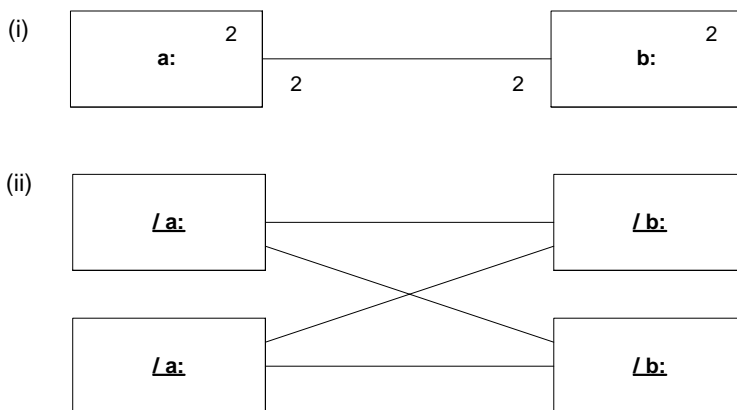
The manner of creation of the containing classifier may override the default instantiation. When an instance specification is used to specify the initial instance to be created for a classifier (see “Class” on page 162), the multiplicities of its parts determine the number of initial instances that will be created within that classifier. Initially, there will be as many instances held in slots as indicated by the corresponding multiplicity. Multiplicity ranges on such instance specifications may not contain upper bounds.

All instances corresponding to parts of a structured classifier are destroyed recursively, when an instance of that structured classifier is deleted. The instance is removed from the extent of its classifier, and is itself destroyed.

## Semantic Variation Points

The rules for matching the multiplicities of connector ends and those of parts and ports they interconnect are a semantic variation point. Also, the specific topology that results from such multi-connectors will differ from system to system. One possible approach to this is illustrated in Figure 9.23 and Figure 9.24.

For each instance playing a role in an internal structure, there will initially be as many links as indicated by the multiplicity of the opposite ends of connectors attached to that role (see “ConnectorEnd” on page 172 for the semantics where no multiplicities are given for an end). If the multiplicities of the ends match the multiplicities of the roles they are attached to (see Figure 9.23 i), the initial configuration that will be created when an instance of the containing classifier is created consists of the set of instances corresponding to the roles (as specified by the multiplicities on the roles) fully connected by links (see the resultant instance, Figure 9.23 ii).



**Figure 9.23 - “Star” connector pattern**

Multiplicities on connector ends serve to restrict the number of initial links created. Links will be created for each instance playing the connected roles according to their ordering until the minimum connector end multiplicity is reached for both ends of the connector (see the resultant instance, Figure 9.24 ii). In this example, only two links are created, resulting in an array pattern.



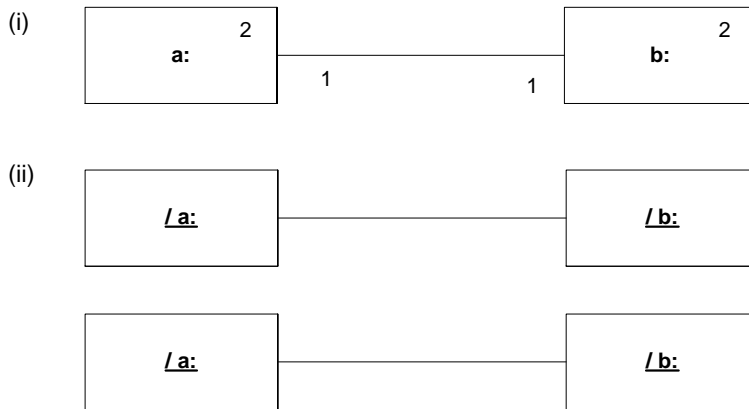


Figure 9.24 - “Array” connector pattern

### Notation

The namestring of a role in an instance specification obeys the following syntax:

`{<name> [ '/' <rolename> ] | '/' <rolename> } [ ':' <classifiername> [ ',' <classifiername> ]* ]`

The name of the instance specification may be followed by the name of the part that the instance plays. The name of the part may only be present if the instance plays a role.

### Examples

The following example shows two classes, *Car* and *Wheel*. The *Car* class has four parts, all of type *Wheel*, representing the four wheels of the car. The front wheels and the rear wheels are linked via a connector representing the front and rear axle, respectively. An implicit association is defined as the type of each axle with each end typed by the *Wheel* class. Figure 9.25 specifies that whenever an instance of the *Car* class is created, four instances of the *Wheel* class are created and held by composition within the car instance. In addition, one link each is created between the front wheel instances and the rear wheel instances.

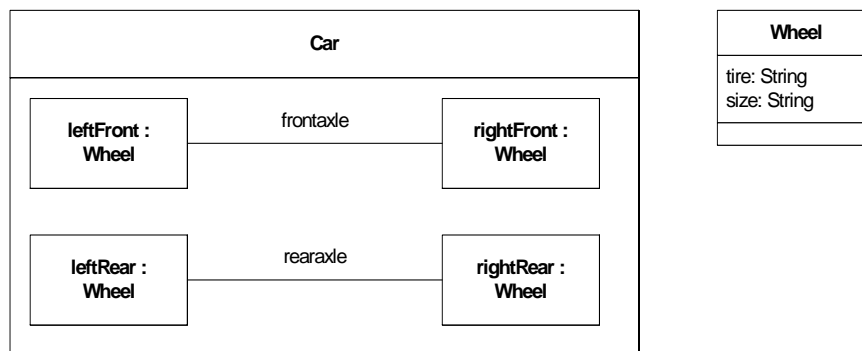
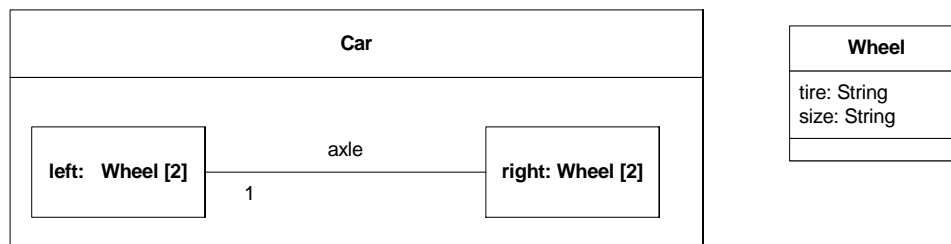


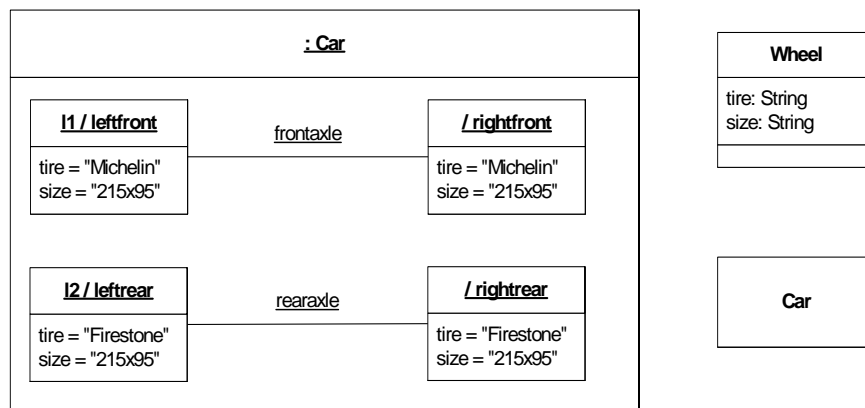
Figure 9.25 - Connectors and parts in a structure diagram

Figure 9.26 specifies an equivalent system, but relies on multiplicities to show the replication of the wheel and axle arrangement. This diagram specifies that there will be two instances of the left wheel and two instances of the right wheel, with each matching instance connected by a link deriving from the connector representing the axle. As specified by the multiplicities, no additional instances of the *Wheel* class can be added as left or right parts for a *Car* instance.



**Figure 9.26 - Connectors and parts in a structure diagram using multiplicities**

Figure 9.27 shows an instance of the *Car* class (as specified in Figure 9.25). It describes the internal structure of the *Car* that it creates and how the four contained instances of *Wheel* will be initialized. In this case, every instance of *Wheel* will have the predefined size and use the brand of tire as specified. The left wheel instances are given names, and all wheel instances are shown as playing the respective roles. The types of the wheel instances have been suppressed.



**Figure 9.27 - A instance of the Car class**

Finally, Figure 9.28 shows a constructor for the *Car* class (see “Class” on page 162). This constructor takes a parameter *brand* of type *String*. It describes the internal structure of the *Car* that it creates and how the four contained instances of *Wheel* will be initialized. In this case, every instance of *Wheel* will have the predefined size and use the brand of tire passed as parameter. The left wheel instances are given names, and all wheel instances are shown as playing the parts. The types of the wheel instances have been suppressed.

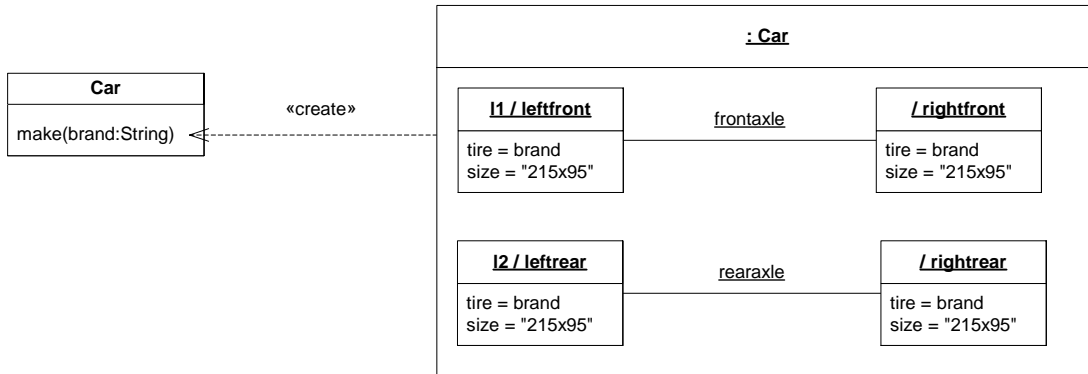


Figure 9.28 - A constructor for the Car class

### 9.3.14 Trigger (from InvocationActions)

#### Generalizations

- “Trigger (from Communications)” on page 441 (*merge increment*)

#### Description

A trigger specification may be qualified by the port on which the event occurred.

#### Associations

- port: Port [\*] Specifies the ports at which a communication that caused an event may have arrived.

#### Semantics

Specifying one or more ports for an event implies that the event triggers the execution of an associated behavior only if the event was received via one of the specified ports.

#### Notation

The ports of a trigger are specified following a trigger signature by a list of port names separated by comma, preceded by the keyword «from»:

‘«from»’ <port-name> [‘,’ <port-name>]\*

### 9.3.15 Variable (from StructuredActivities)

#### Generalizations

- “Variable (from StructuredActivities)” on page 401 (*merge increment*)

#### Description

A variable is considered a connectable element.

## Semantics

Extends variable to specialize connectable element.

## 9.4 Diagrams

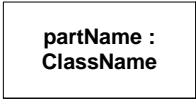



### Composite structure diagram

A composite structure diagram depicts the internal structure of a classifier, as well as the use of a collaboration in a collaboration use.

#### Graphical nodes

Additional graphical nodes that can be included in composite structure diagrams are shown in Table 9.1.



**Table 9.1 - Graphic nodes included in composite structure diagrams**

Node Type	Notation	Reference
Part		See “Property (from InternalStructures)” on page 179.
Port		See “Ports” on page 175. A port may appear either on a contained part representing a port on that part, or on the boundary of the class diagram, representing a port on the represented classifier itself. The optional <i>ClassifierName</i> is only used if it is desired to specify a class of an object that implements the port.
Collaboration		See “Collaboration” on page 164.
CollaborationUse		See “CollaborationUse (from Collaborations)” on page 166.

### *Graphical paths*

Additional graphical paths that can be included in composite structure diagrams are shown in Table 9.2.

**Table 9.2 - Graphic nodes included in composite structure diagrams**

Path Type	Notation	Reference
Connector		See “Connector” on page 170.
Role binding		See “CollaborationUse (from Collaborations)” on page 166.

### **Structure diagram**

All graphical nodes and paths shown on composite structure diagrams can also be shown on other structure diagrams.

# 10 Deployments

## 10.1 Overview

The Deployments package specifies a set of constructs that can be used to define the execution architecture of systems that represent the assignment of software artifacts to nodes. Nodes are connected through communication paths to create network systems of arbitrary complexity. Nodes are typically defined in a nested manner, and represent either hardware devices or software execution environments. Artifacts represent concrete elements in the physical world that are the result of a development process.

The Deployment package supports a streamlined model of deployment that is deemed sufficient for the majority of modern applications. Where more elaborate deployment models are required, it can be extended through profiles or meta models to model specific hardware and software environments.

### Artifacts

The Artifacts package defines the basic Artifact construct as a special kind of Classifier.

### Nodes

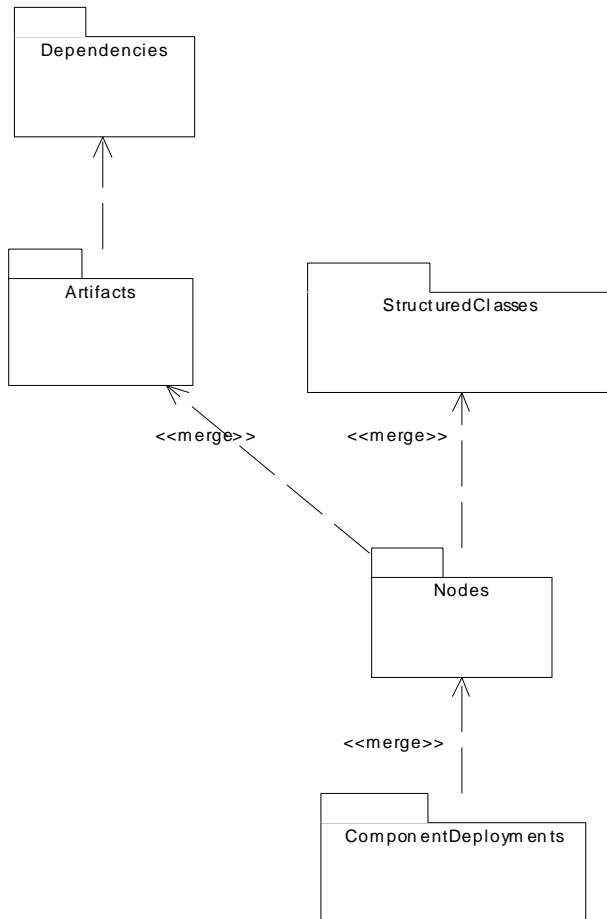
The Nodes package defines the concept of Node, as well as the basic deployment relationship between Artifacts and Nodes.

### Component Deployments

The ComponentDeployments package extends the basic deployment model with capabilities to support deployment mechanisms found in several common component technologies.

## 10.2 Abstract syntax

Figure 10.1 shows the dependencies of the Deployments packages.



**Figure 10.1 - Dependencies between packages described in this chapter**

## Package Artifacts

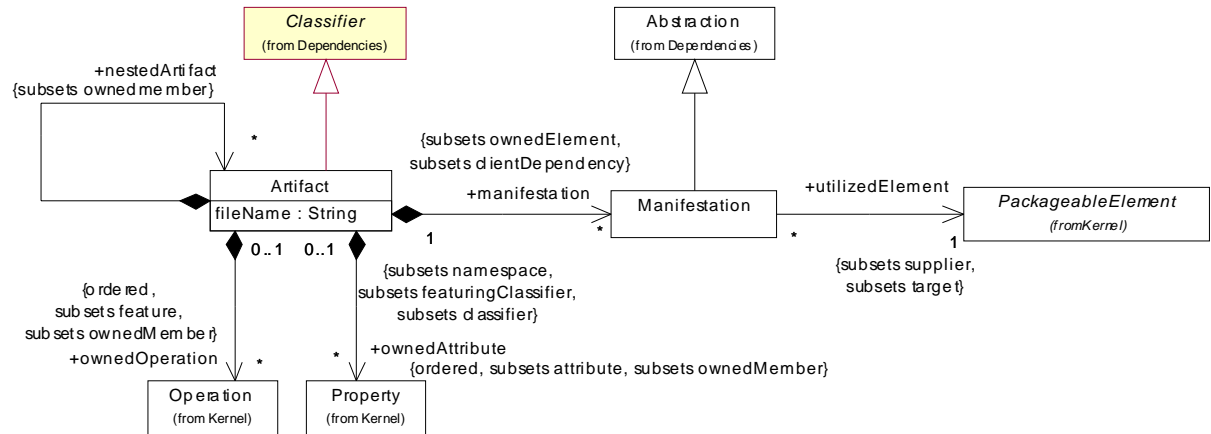


Figure 10.2 - The elements defined in the Artifacts package

## Package Nodes

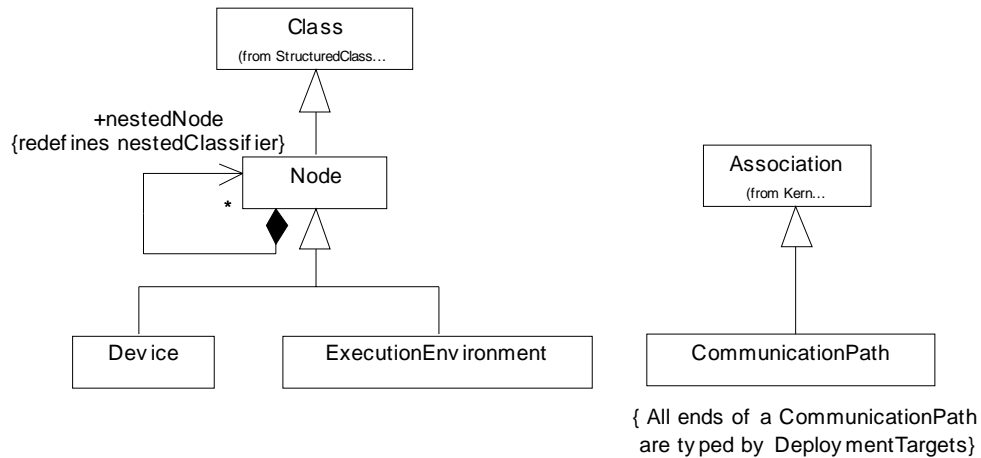
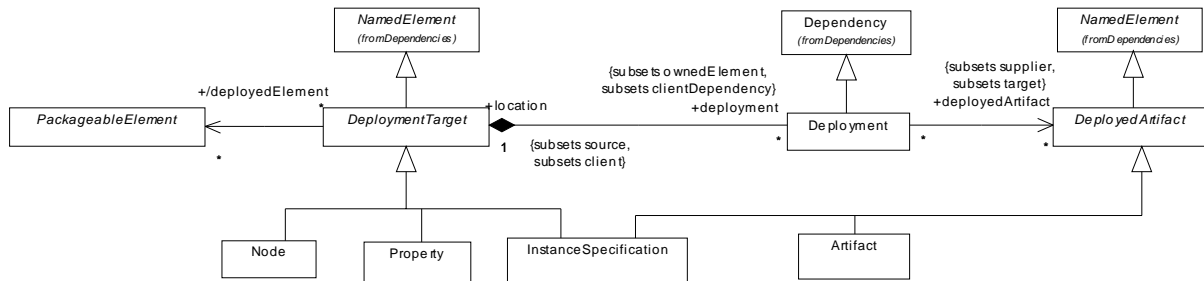


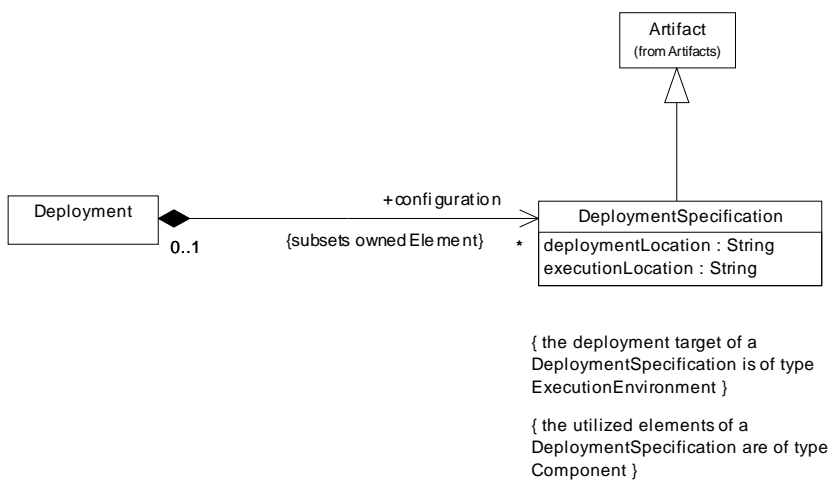
Figure 10.3 - The definition of the Node concept





**Figure 10.4 - Definition of the Deployment relationship between DeploymentTargets and DeployedArtifacts**

### *Package ComponentDeployments*



**Figure 10.5 - Metaclasses that define component Deployment**

## **10.3 Class Descriptions**

### **10.3.1 Artifact (from Artifacts, Nodes)**

An artifact is the specification of a physical piece of information that is used or produced by a software development process, or by deployment and operation of a system. Examples of artifacts include model files, source files, scripts, and binary executable files, a table in a database system, a development deliverable, or a word-processing document, a mail message.

## Generalizations

- “Classifier (from Kernel, Dependencies, PowerTypes)” on page 48
- “DeployedArtifact (from Nodes)” on page 195

## Description

### *Package Artifacts*

In the metamodel, an Artifact is a Classifier that represents a physical entity. Artifacts may have Properties that represent features of the Artifact, and Operations that can be performed on its instances. Artifacts can be involved in Associations to other Artifacts (e.g., composition associations). Artifacts can be instantiated to represent detailed copy semantics, where different instances of the same Artifact may be deployed to various Node instances (and each may have separate property values, e.g., for a ‘time-stamp’ property).

### *Package Node*

As part of the Nodes package, an Artifact is extended to become the source of a deployment to a Node. This is achieved by specializing the abstract superclass DeployedArtifact defined in the Nodes package.

## Attributes

### *Package Artifacts*

- filename : String [0..1]      A concrete name that is used to refer to the Artifact in a physical context. Example: file system name, universal resource locator.

## Associations

### *Package Artifacts*

- nestedArtifact: Artifact [\*]      The Artifacts that are defined (nested) within the Artifact. The association is a specialization of the *ownedMember* association from Namespace to NamedElement.
- ownedProperty : Property [\*]      The attributes or association ends defined for the Artifact. The association is a specialization of the *ownedMember* association.
- ownedOperation : Operation [\*]      The Operations defined for the Artifact. The association is a specialization of the *ownedMember* association.
- manifestation : Manifestation [\*]      The set of model elements that are manifested in the Artifact. That is, these model elements are utilized in the construction (or generation) of the artifact. This association is a specialization of the clientDependency association.

## Constraints

No additional constraints

## Semantics

An Artifact defined by the user represents a concrete element in the physical world. A particular instance (or ‘copy’) of an artifact is deployed to a node instance. Artifacts may have composition associations to other artifacts that are nested within it. For instance, a deployment descriptor artifact for a component may be contained within the artifact that implements that component. In that way, the component and its descriptor are deployed to a node instance as one artifact instance.

Specific profiles are expected to stereotype artifact to model sets of files (e.g., as characterized by a ‘file extension’ on a file system). The UML Standard Profile defines several standard stereotypes that apply to Artifacts, e.g., «source» or «executable» (See Annex C - Standard Stereotypes). These stereotypes can be further specialized into implementation and platform specific stereotypes in profiles. For example, an EJB profile might define «jar» as a subclass of «executable» for executable Java archives.

## Notation

An artifact is presented using an ordinary class rectangle with the key-word «artifact». Alternatively, it may be depicted by an icon.

Optionally, the underlining of the name of an artifact instance may be omitted, as the context is assumed to be known to users.



Figure 10.6 - An Artifact instance

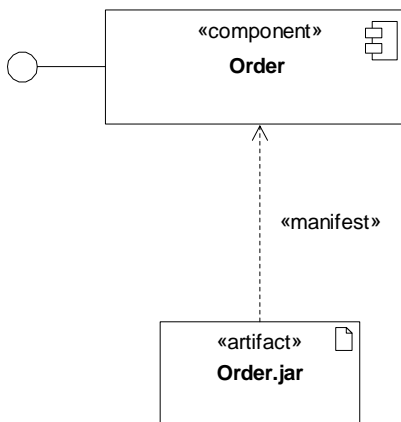


Figure 10.7 - A visual representation of the manifestation relationship between artifacts and components

## Changes from previous UML

The following changes from UML 1.x have been made: Artifacts can now manifest any PackageableElement (not just Components, as in UML 1.x).

### 10.3.2 CommunicationPath (from Nodes)

A communication path is an association between two DeploymentTargets, through which they are able to exchange signals and messages.

#### Generalizations

- “Association (from Kernel)” on page 36

#### Description

In the metamodel, CommunicationPath is a subclass of Association.

#### Attributes

No additional attributes

#### Associations

No additional associations

#### Constraints

[1] The association ends of a CommunicationPath are typed by DeploymentTargets.

#### Semantics

A communication path is an association that can only be defined between deployment targets, to model the exchange of signals and messages between them.

#### Notation

No additional notation

#### Changes from previous UML

The following changes from UML 1.x have been made: CommunicationPath was implicit in UML 1.x. It has been made explicit to formalize the modeling of networks of complex Nodes.

### 10.3.3 DeployedArtifact (from Nodes)

A deployed artifact is an artifact or artifact instance that has been deployed to a deployment target.

#### Generalizations

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

#### Description

In the metamodel, DeployedArtifact is an abstract metaclass that is a specialization of NamedElement. A DeployedArtifact is involved in one or more Deployments to a DeploymentTarget.

### Attributes

No additional attributes

### Associations

No additional associations

### Constraints

No additional constraints

### Semantics

Deployed artifacts are deployed to a deployment target.

### Notation

No additional notation

### Changes from previous UML

The following changes from UML 1.x have been made: The capability to deploy artifacts and artifact instances to nodes has been made explicit based on UML 2.0 instance modeling through the addition of this abstract metaclass.

## 10.3.4 Deployment (from ComponentDeployments, Nodes)

### *Package Nodes*

A deployment is the allocation of an artifact or artifact instance to a deployment target.

### *Package ComponentDeployments*

A component deployment is the deployment of one or more artifacts or artifact instances to a deployment target, optionally parameterized by a deployment specification. Examples are executables and configuration files.

### Generalizations

- “Dependency (from Dependencies)” on page 58

### Description

In the metamodel, Deployment is a subtype of Dependency.

### Attribute

No additional attributes

### Associations

### *Package Nodes*

- `deployedArtifact : Artifact [*]`      The Artifacts that are deployed onto a Node. This association specializes the supplier association.

- location : Node [1] The Node which is the target of a Deployment. This association specializes the client association.

#### *Package ComponentDeployments*

- configuration : deploymentSpecification [\*] The specification of properties that parameterize the deployment and execution of one or more Artifacts. This association is specialized from the ownedMember association.

### Constraints

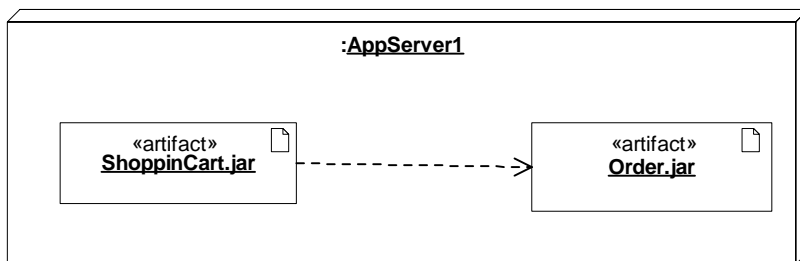
No additional constraints

### Semantics

The deployment relationship between a DeployedArtifact and a DeploymentTarget can be defined at the “type” level and at the “instance level.” For example, a ‘type level’ deployment relationship can be defined between an “application server” Node and an “order entry request handler” executable Artifact. At the ‘instance level’ 3 specific instances “app-server1” ... “app-server3” may be the deployment target for six “request handler\*” instances. Finally, for modeling complex deployment target models consisting of nodes with a composite structure defined through ‘parts,’ a Property (that functions as a part) may also be the target of a deployment.

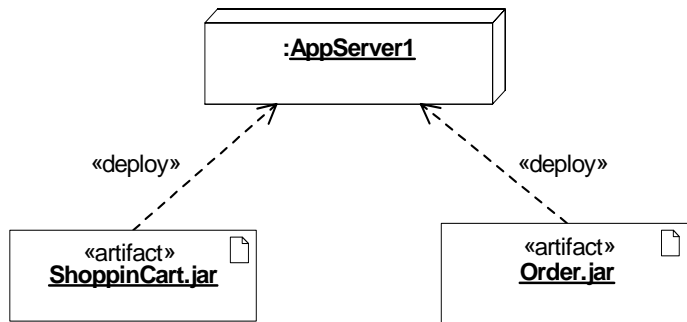
### Notation

Deployment diagrams show the allocation of Artifacts to Nodes according to the Deployments defined between them.

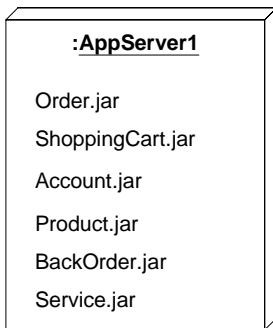


**Figure 10.8 - A visual representation of the deployment location of artifacts (including a dependency between the artifacts).**

An alternative notation to containing the deployed artifacts within a deployment target symbol is to use a dependency labeled «deploy» that is drawn from the artifact to the deployment target.



**Figure 10.9 - Alternative deployment representation of using a dependency called «deploy»**



**Figure 10.10 - Textual list based representation of the deployment location of artifacts**

## Changes from previous UML

The following changes from UML 1.x have been made — an association to DeploymentSpecification has been added.

### 10.3.5 DeploymentSpecification (from ComponentDeployments)

A deployment specification specifies a set of properties that determine execution parameters of a component artifact that is deployed on a node. A deployment specification can be aimed at a specific type of container. An artifact that reifies or implements deployment specification properties is a deployment descriptor.

#### Generalizations

- “Artifact (from Artifacts, Nodes)” on page 192

#### Description

In the metamodel, a DeploymentSpecification is a subtype of Artifact. It defines a set of deployment properties that are specific to a certain Container type. An instance of a DeploymentSpecification with specific values for these properties may be contained in a complex Artifact.

## Attributes

### *ComponentDeployments Package*

- deploymentLocation : String      The location where an Artifact is deployed onto a Node. This is typically a 'directory' or 'memory address.'
- executionLocation : String      The location where a component Artifact executes. This may be a local or remote location.

## Associations

### *ComponentDeployments Package*

- deployment : Deployment [0..1]      The deployment with which the DeploymentSpecification is associated.

## Constraints

- [1] The DeploymentTarget of a DeploymentSpecification is a kind of ExecutionEnvironment.
- [2] The deployedElements of a DeploymentTarget that are involved in a Deployment that has an associated DeploymentSpecification is a kind of Component (i.e., the configured components).

## Semantics

A Deployment specification is a general mechanism to parameterize a Deployment relationship, as is common in various hardware and software technologies. The deployment specification element is expected to be extended in specific component profiles. Non-normative examples of the standard stereotypes that a profile might add to deployment specification are, for example, «concurrencyMode» with tagged values {thread, process, none}, or «transactionMode» with tagged values {transaction, nestedTransaction, none}.

## Notation

A DeploymentSpecification is graphically displayed as a classifier rectangle that is attached to a component artifact that is deployed on a container using a regular dependency notation. If the deployment relationship is made explicit (as in Figure 10.13), the Dependency may be attached to that relationship.



**Figure 10.11 - DeploymentSpecification for an artifact (specification and instance levels)**



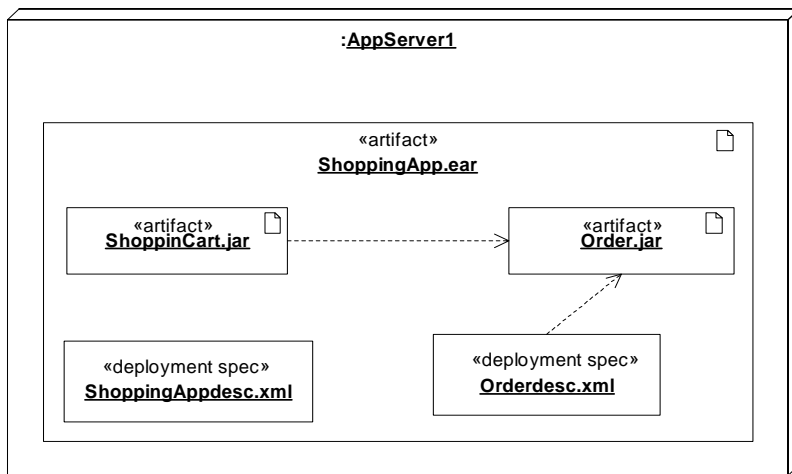


Figure 10.12 - DeploymentSpecifications related to the artifacts that they parameterize

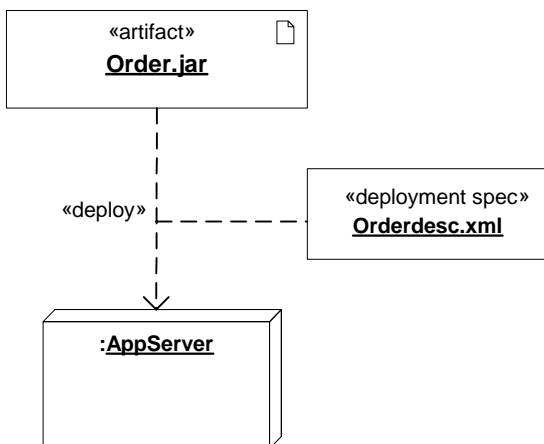


Figure 10.13 - A DeploymentSpecification for an artifact

### Changes from previous UML

The following changes from UML 1.x have been made — DeploymentSpecification does not exist in UML 1.x.

### 10.3.6 DeploymentTarget (from Nodes)

A deployment target is the location for a deployed artifact.

#### Generalizations

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

## Description

In the metamodel, DeploymentTarget is an abstract metaclass that is a specialization of NamedElement. A DeploymentTarget owns a set of Deployments.

## Attributes

No additional attributes

## Associations

### Nodes Package

- deployment : Deployment [\*]  
The set of Deployments for a DeploymentTarget. This association specializes the clientDependency association.
- / deployedElement : PackageableElement [\*]  
The set of elements that are manifested in an Artifact that is involved in Deployment to a DeploymentTarget. The association is a derived association.  
**context** DeploymentTarget::deployedElement **derive:**  
`((self.deployment->collect(deployedArtifact))->collect(manifestation))->collect(utilizedElement)`

## Constraints

No additional constraints

## Semantics

Artifacts are deployed to a deployment target. The deployment target owns the set of deployments that target it.

## Notation

No additional notation

## Changes from previous UML

The following changes from UML 1.x have been made: The capability to deploy artifacts and artifact instances to nodes has been made explicit based on UML 2.0 instance modeling.

### 10.3.7 Device (from Nodes)

A Device is a physical computational resource with processing capability upon which artifacts may be deployed for execution. Devices may be complex (i.e., they may consist of other devices).

## Generalizations

- “Node (from Nodes)” on page 205

## Description

In the metamodel, a Device is a subclass of Node.

## Attributes

No additional attributes

## Associations

No additional associations

## Constraints

No additional constraints

## Semantics

A device may be a nested element, where a physical machine is decomposed into its elements, either through namespace ownership or through attributes that are typed by Devices.

## Notation

A Device is notated by a Node annotated with the stereotype «device».

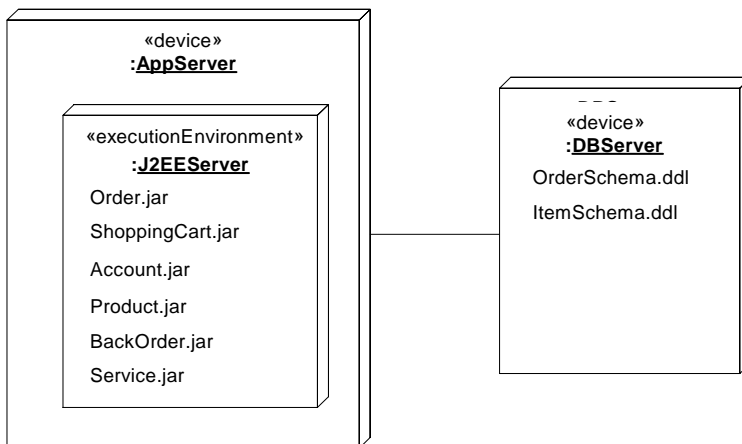


Figure 10.14 - Notation for a Device

## Changes from previous UML

The following changes from UML 1.x have been made — Device is not defined in UML 1.x.

### 10.3.8 ExecutionEnvironment (from Nodes)

An ExecutionEnvironment is a node that offers an execution environment for specific types of components that are deployed on it in the form of executable artifacts.

## Generalizations

- “Node (from Nodes)” on page 205

## Description

In the metamodel, an ExecutionEnvironment is a subclass of Node. It is usually part of a general Node, representing the physical hardware environment on which the ExecutionEnvironment resides. In that environment, the ExecutionEnvironment implements a standard set of services that Components require at execution time (at the modeling level these services are usually implicit). For each component Deployment, aspects of these services may be determined by properties in a DeploymentSpecification for a particular kind of ExecutionEnvironment.

## Attributes

No additional attributes

## Associations

No additional associations

## Constraints

No additional constraints

## Semantics

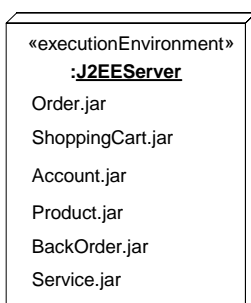
ExecutionEnvironment instances are assigned to node instances by using composite associations between nodes and ExecutionEnvironments, where the ExecutionEnvironment plays the role of the part. ExecutionEnvironments can be nested (e.g., a database ExecutionEnvironment may be nested in an operating system ExecutionEnvironment). Components of the appropriate type are then deployed to specific ExecutionEnvironment nodes.

Typical examples of standard ExecutionEnvironments that specific profiles might define stereotypes for are «OS», «workflow engine», «database system», and «J2EE container».

An ExecutionEnvironment can optionally have an explicit interface of system level services that can be called by the deployed elements, in those cases where the modeler wants to make the ExecutionEnvironment software execution environment services explicit.

## Notation

A ExecutionEnvironment is notated by a Node annotated with the stereotype «executionEnvironment».



**Figure 10.15 - Notation for a ExecutionEnvironment (example of an instance of a J2EEServer ExecutionEnvironment)**

## Changes from previous UML

The following changes from UML 1.x have been made — ExecutionEnvironment is not defined in UML 1.x.

### 10.3.9 InstanceSpecification (from Nodes)

An instance specification is extended with the capability of being a deployment target in a deployment relationship, in the case that it is an instance of a node. It is also extended with the capability of being a deployed artifact, if it is an instance of an artifact.

#### Generalizations

- “DeployedArtifact (from Nodes)” on page 195
- “DeploymentTarget (from Nodes)” on page 200
- “InstanceSpecification (from Kernel)” on page 78 (*merge increment*)

#### Description

In the metamodel, InstanceSpecification is a specialization of DeploymentTarget and DeployedArtifact.

#### Attributes

No additional attributes

#### Associations

No additional associations

#### Constraints

- [1] An InstanceSpecification can be a DeploymentTarget if it is the instance specification of a Node and functions as a part in the internal structure of an encompassing Node.
- [2] An InstanceSpecification can be a DeployedArtifact if it is the instance specification of an Artifact.

#### Semantics

No additional semantics

#### Notation

An instance can be attached to a node using a deployment dependency, or it may be visually nested inside the node.

## Changes from previous UML

The following changes from UML 1.x have been made — the capability to deploy artifact instances to node instances existed in UML 1.x, and has been made explicit based on UML 2.0 instance modeling.

### 10.3.10 Manifestation (from Artifacts)

A manifestation is the concrete physical rendering of one or more model elements by an artifact.

## Generalizations

- “Abstraction (from Dependencies)” on page 35

## Description

In the metamodel, a Manifestation is a subtype of Abstraction. A Manifestation is owned by an Artifact.

## Attributes

No additional attributes

## Associations

Artifacts

- utilizedElement : PackageableElement [1] The model element that is utilized in the manifestation in an Artifact. This association specializes the supplier association.

## Constraints

No additional associations

## Semantics

An artifact embodies or manifests a number of model elements. The artifact owns the manifestations, each representing the utilization of a packageable element.

Specific profiles are expected to stereotype the manifestation relationship to indicate particular forms of manifestation. For example, <<tool generated>> and <<custom code>> might be two manifestations for different classes embodied in an artifact.

## Notation

A manifestation is notated in the same way as an abstraction dependency, i.e., as a general dashed line with an open arrow-head labeled with the keyword <<manifest>>.

## Changes from previous UML

The following changes from UML 1.x have been made: Manifestation is defined as a meta association in UML 1.x, prohibiting stereotyping of manifestations. In UML 1.x, the concept of Manifestation was referred to as ‘implementation’ and annotated in the notation as <<implement>>. Since this was one of the many uses of the word ‘implementation’ this has been replaced by <<manifest>>.

### 10.3.11 Node (from Nodes)

A node is computational resource upon which artifacts may be deployed for execution.

Nodes can be interconnected through communication paths to define network structures.

## Generalizations

- “Class (from StructuredClasses)” on page 162

- “DeploymentTarget (from Nodes)” on page 200

## Description

In the metamodel, a Node is a subclass of Class. It is associated with a Deployment of an Artifact. It is also associated with a set of Elements that are deployed on it. This is a derived association in that these PackageableElements are involved in a Manifestation of an Artifact that is deployed on the Node. Nodes may have an internal structure defined in terms of parts and connectors associated with them for advanced modeling applications.

## Attributes

No additional attributes

## Associations

### *Nodes Package*

- nestedNode : Node [\*]     The Nodes that are defined (nested) within the Node. The association is a specialization of the *ownedMember* association from Namespace to NamedElement.

## Constraints

[1] The internal structure of a Node (if defined) consists solely of parts of type Node.

## Semantics

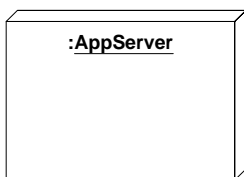
Nodes can be connected to represent a network topology by using communication paths. Communication paths can be defined between nodes such as “application server” and “client workstation” to define the possible communication paths between nodes. Specific network topologies can then be defined through links between node instances.

Hierarchical nodes (i.e., nodes within nodes) can be modeled using composition associations, or by defining an internal structure for advanced modeling applications.

Non-normative examples of nodes are «application server», «client workstation», «mobile device», «embedded device».

## Notation

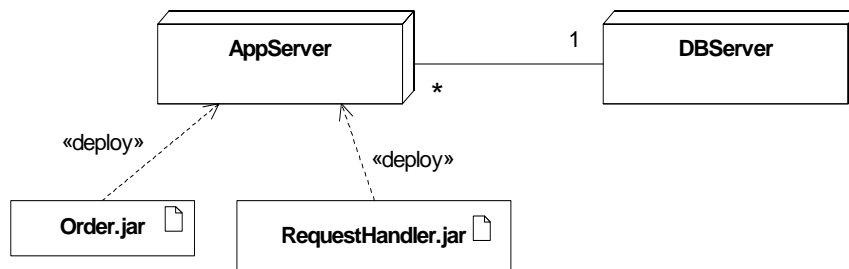
A node is shown as a figure that looks like a 3-dimensional view of a cube.



**Figure 10.16 - An instance of a Node**

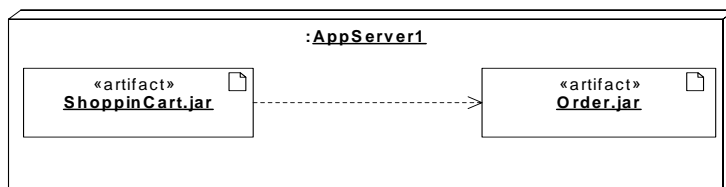
Dashed arrows with the keyword «deploy» show the capability of a node type to support a component type. Alternatively, this may be shown by nesting component symbols inside the node symbol.

Nodes may be connected by associations to other nodes. A link between node instances indicates a communication path between the nodes.



**Figure 10.17 - Communication path between two Node types with deployed Artifacts**

Artifacts may be contained within node instance symbols. This indicates that the items are deployed on the node instances.



**Figure 10.18 - A set of deployed component artifacts on a Node**

### Changes from previous UML

The following changes from UML 1.x have been made: to be written.

### 10.3.12 Property (from Nodes)

A Property is extended with the capability of being a DeploymentTarget in a Deployment relationship. This enables modeling the deployment to hierarchical Nodes that have Properties functioning as internal parts.

#### Generalizations

- “Property (from InternalStructures)” on page 179 (*merge increment*)

#### Description

In the metamodel, Property is a specialization of DeploymentTarget.

#### Attributes

No additional attributes

#### Associations

No additional associations



## Constraints

- [1] A Property can be a DeploymentTarget if it is a kind of Node and functions as a part in the internal structure of an encompassing Node.

## Semantics

No additional semantics

## Notation

No additional notation

## Changes from previous UML

The following changes from UML 1.x have been made — the capability to deploy to Nodes with an internal structure has been added to UML 2.0.

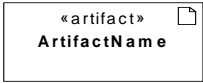


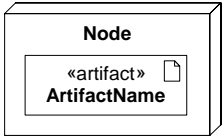
# 10.4 Diagrams

## Deployment diagram


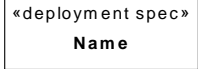
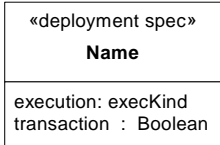
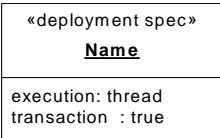
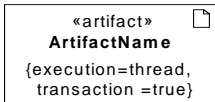
### Graphical nodes

The graphic nodes that can be included in deployment diagrams are shown in Table 10.1.

**Table 10.1 - Graphic nodes included in deployment diagrams**

Node Type	Notation	Reference
Artifact		See “Artifact.”
Node		See “Node.” Has keyword options «device» and «execution environment».
Artifact deployed on Node		See “Deployment.”
Node with deployed Artifacts		See “Deployment.”

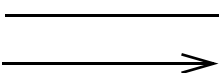
**Table 10.1 - Graphic nodes included in deployment diagrams**

Node Type	Notation	Reference
Node with deployed Artifacts		See “Deployment” (alternative, textual notation).
Deployment specification		See “Deployment Specification.”
Deployment specification - with properties		See “Deployment Specification.”
Deployment specification - with property values		See “Deployment Specification.”
Artifact with annotated deployment properties		See “Artifact.”

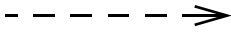
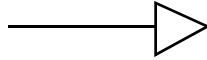
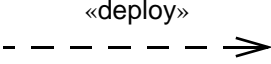
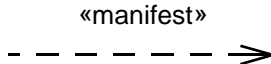
### Graphical paths

The graphic paths that can be included in deployment diagrams are shown in Table 10.2 .

**Table 10.2 - Graphic nodes included in deployment diagrams**

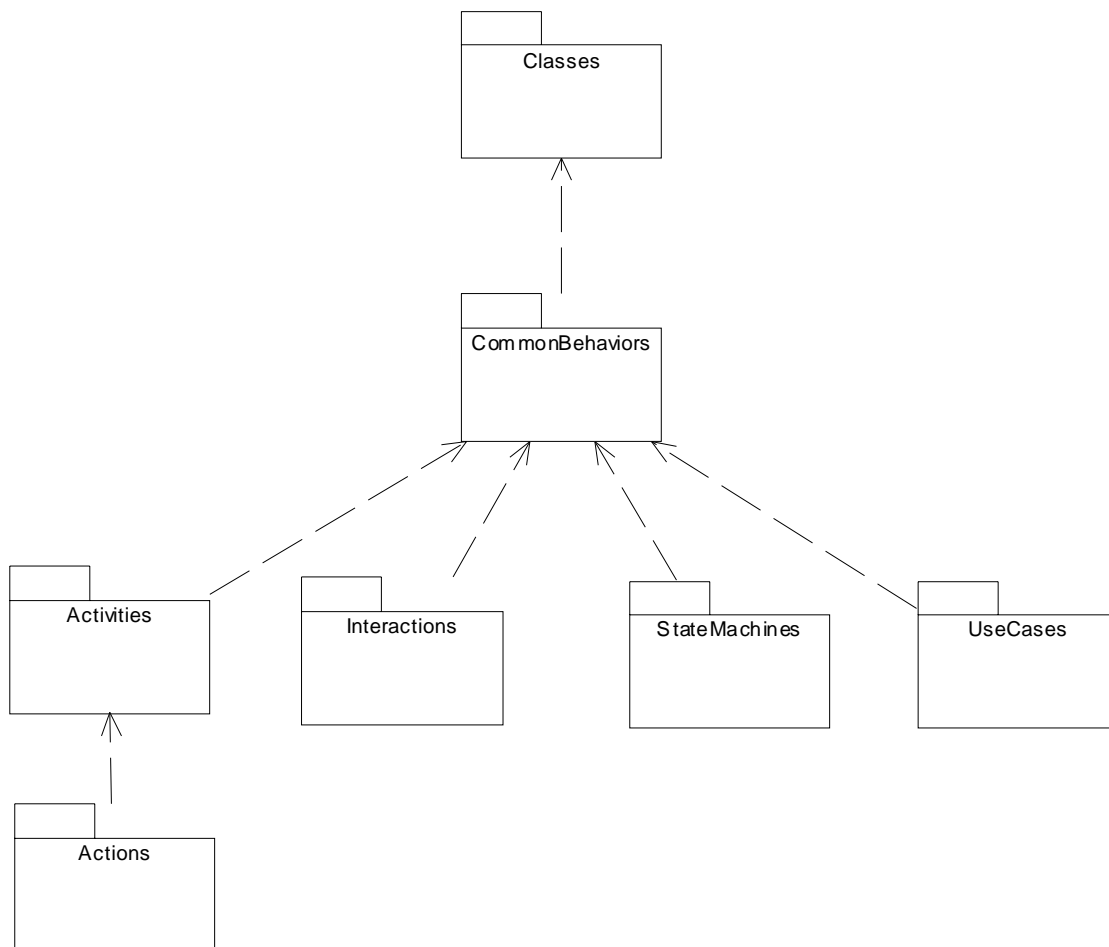
Path Type	Notation	Reference
Association		See “Association (from Kernel)” on page 36. Used to model communication paths between DeploymentTargets.

**Table 10.2 - Graphic nodes included in deployment diagrams**

Path Type	Notation	Reference
Dependency		See “Dependency (from Dependencies)” on page 58. Used to model general dependencies. In Deployment diagrams, this notation is used to depict the following metamodel associations: (i) the relationship between an Artifact and the model element(s) that it implements, and (ii) the deployment of an Artifact (instance) on a Node (instance).
Generalization		See “Generalization (from Kernel, PowerTypes)” on page 67.
Deployment		The Deployment relationship
Manifestation		The Manifestation relationship

## Part II - Behavior

This part specifies the dynamic, behavioral constructs (e.g., activities, interactions, state machines) used in various behavioral diagrams, such as activity diagrams, sequence diagrams, and state machine diagrams. The UML packages that support behavioral modeling, along with the structure packages they depend upon (CompositeStructures and Classes) are shown in the figure below.



**Part II, Figure 1 - UML packages that support behavioral modeling**

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



# 11 Actions

## 11.1 Overview

### Basic Concepts

An action is the fundamental unit of behavior specification. An action takes a set of inputs and converts them into a set of outputs, though either or both sets may be empty. The most basic action provides for implementation-dependent semantics, while other packages provide more specific actions. Some of the actions modify the state of the system in which the action executes. The values that are the inputs to an action may be described by value specifications, obtained from the output of actions that have one output (in StructuredActions), or in ways specific to the behaviors that use them. For example, the activity flow model supports providing inputs to actions from the outputs of other actions.

Actions are contained in behaviors, which provide their context. Behaviors provide constraints among actions to determine when they execute and what inputs they have. The Actions chapter is concerned with the semantics of individual, primitive actions.

Basic actions include those that perform operation calls, signal sends, and direct behavior invocation. Operation calls are specified in the model and can be dynamically selected only through polymorphism. Signals are specified by a signal object, whose type represents the kind of message transmitted between objects, and can be dynamically created. Note that operations may be bound to activities, state machine transitions, or other behaviors. The receipt of signals may be bound to activities, state machine transitions, or other behaviors.

### Intermediate Concepts

The intermediate level describes the various action primitive actions. These primitive actions are defined in such a way as to enable the maximum range of mappings. Specifically, primitive actions are defined so that they either carry out a computation or access object memory, but never both. This approach enables clean mappings to a physical model, even those with data organizations different from that suggested by the specification. In addition, any re-organization of the data structure will leave the specification of the computation unaffected.

A surface action language would encompass both primitive actions and the control mechanisms provided by behaviors. In addition, a surface language may map higher-level constructs to the actions. For example, creating an object may involve initializing attribute values or creating objects for mandatory associations. The specification defines the create action to only create the object, and requires further actions to initialize attribute values and create objects for mandatory associations. A surface language could choose to define a creation operation with initialization as a single unit as a shorthand for several actions.

A particular surface language could implement each semantic construct one-to-one, or it could define higher-level, composite constructs to offer the modeler both power and convenience. This specification, then, expresses the fundamental semantics in terms of primitive behavioral concepts that are conceptually simple to implement. Modelers can work in terms of higher-level constructs as provided by their chosen surface language or notation.

The semantic primitives are defined to enable the construction of different execution engines, each of which may have different performance characteristics. A model compiler builder can optimize the structure of the software to meet specific performance requirements, so long as the semantic behavior of the specification and the implementation remain the same. For example, one engine might be fully sequential within a single task, while another may separate the classes into different processors based on potential overlapping of processing, and yet others may separate the classes in a client-server, or even a three-tier model.

The modeler can provide “hints” to the execution engine when the modeler has special knowledge of the domain solution that could be of value in optimizing the execution engine. For example, instances could—by design—be partitioned to match the distribution selected, so tests based on this partitioning can be optimized on each processor. The execution engines are not required to check or enforce such hints. An execution engine can either assume that the modeler is correct, or just ignore it. An execution engine is not required to verify that the modeler’s assertion is true.

When an action violates aspects of static UML modeling that constrain runtime behavior, the semantics is left undefined. For example, attempting to create an instance of an abstract class is undefined - some languages may make this action illegal, others may create a partial instance for testing purposes. The semantics are also left undefined in situations that require classes as values at runtime. However, in the execution of actions the lower multiplicity bound is ignored and no error or undefined semantics is implied. (Otherwise it is impossible to use actions to pass through the intermediate configurations necessary to construct object configurations that satisfy multiplicity constraints.) The modeler must determine when minimum multiplicity should be enforced, and these points cannot be everywhere or the configuration cannot change.

### *Invocation Actions*

More invocation actions are defined for broadcasting signals to the available “universe” and transmitting objects that are not signals.

### *Read Write Actions*

Objects, structural features, links, and variables have values that are available to actions. Objects can be created and destroyed; structural features and variables have values; links can be created and destroyed, and can reference values through their ends; all of which are available to actions. Read actions get values, while write actions modify values and create and destroy objects and links. Read and write actions share the structures for identifying the structural features, links, and variables they access.

Object actions create and destroy objects. Structural feature actions support the reading and writing of structural features. The abstract metaclass `StructuralFeatureAction` statically specifies the structural feature being accessed. The object to access is specified dynamically, by referring to an input pin on which the object will be placed at runtime. The semantics for static features is undefined. Association actions operate on associations and links. In the description of these actions, the term “associations” does not include those modeled with association classes, unless specifically indicated. Similarly, a “link” is not a link object unless specifically indicated. The semantics of actions that read and write associations that have a static end is undefined.

Value specifications cover various expressions ranging from implementation-dependent constants and complex expressions, even with side-effects. An action is defined for evaluating these. Also see “ValuePin (from BasicActions)” on page 279.

## **Complete Concepts**

The major constructs associated with complete actions are outlined below.

### *Read Write Actions*

Additional actions deal with the relation between object and class and link objects. These read the instances of a given classifier, check which classifier an instance is classified by, and change the classifier of an instance. Link object actions operate on instances of association classes. Also the reading and writing actions of associations are extended to support qualifiers.

## Other Actions

Actions are defined for accepting events, including operation calls, and retrieving the property values of an object all at once. The `StartClassifierBehaviorAction` provides a way to indicate when the classifier behavior of a newly created object should begin to execute.

## Structured Concepts

These actions operate in the context of activities and structured nodes. Variable actions support the reading and writing of variables. The abstract metaclass `VariableAction` statically specifies the variable being accessed. Variable actions can only access variables within the activity of which the action is a part. An action is defined for raising exceptions and a kind of input pin is defined for accepting the output of an action without using flows.

## 11.2 Abstract Syntax

The package dependencies of the Actions chapter are shown in Figure 11.1.

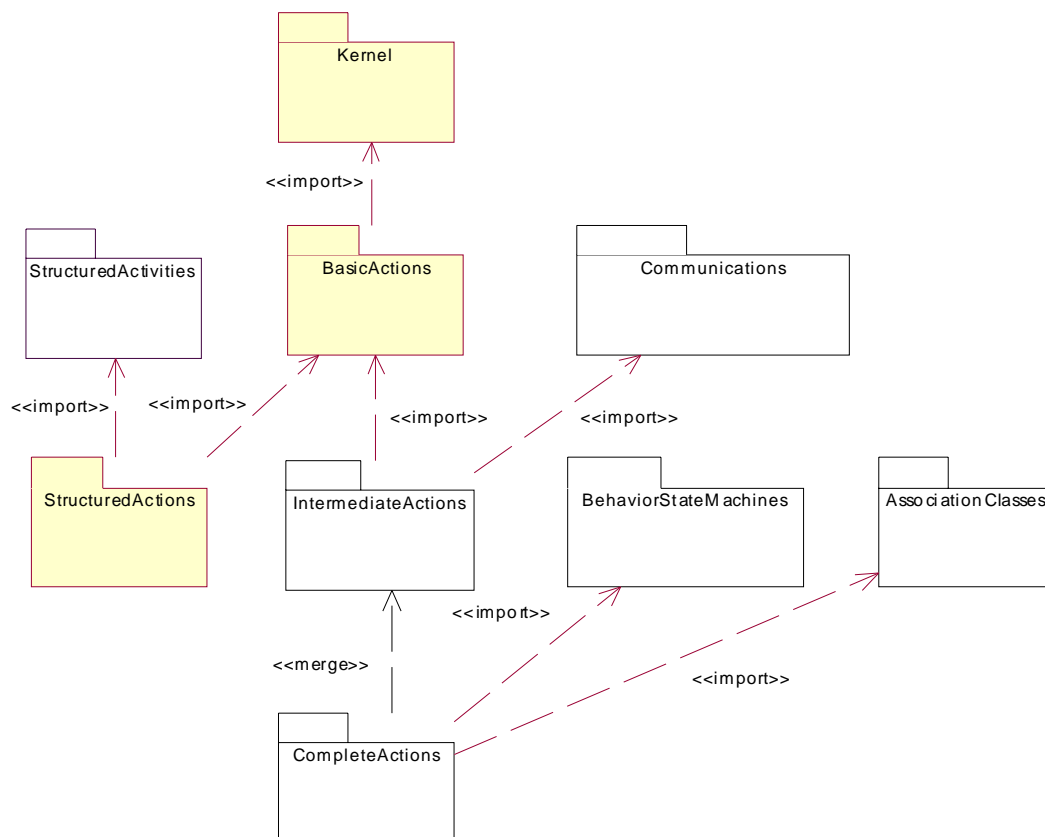


Figure 11.1 - Dependencies of the Action packages



## Package BasicActions

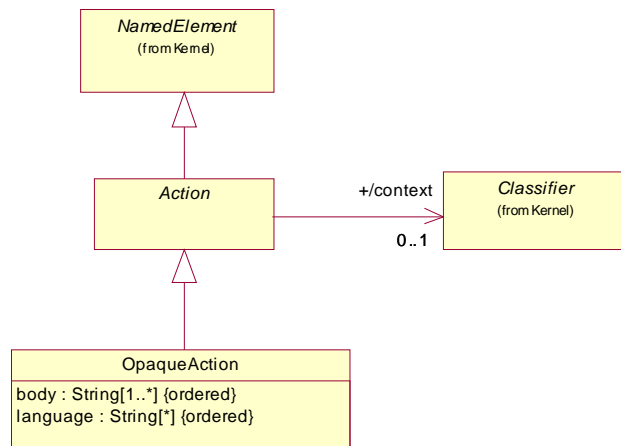


Figure 11.2 - Basic actions

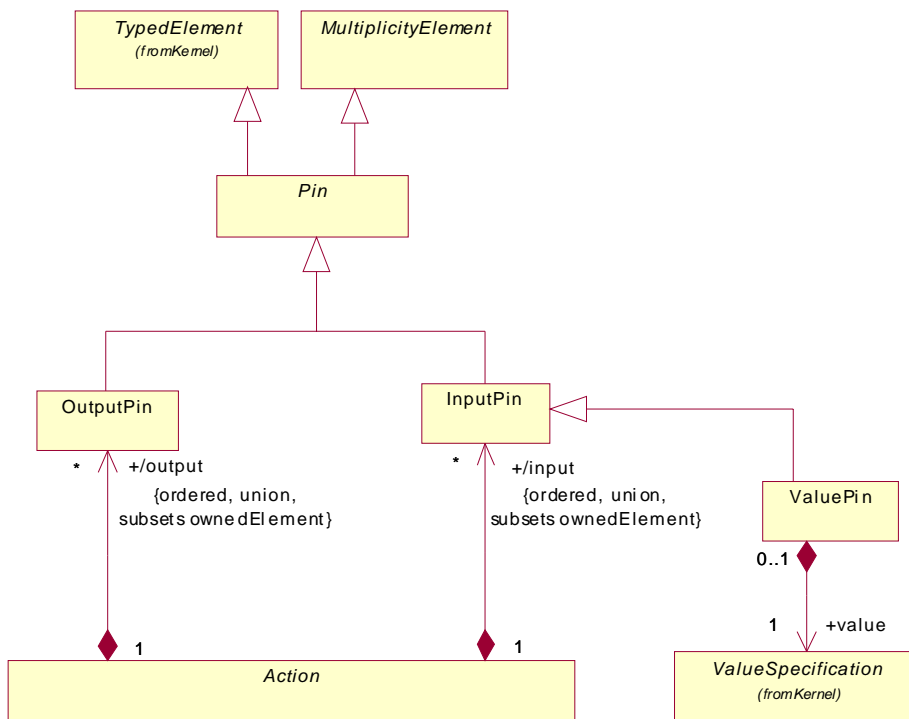


Figure 11.3 - Basic pins

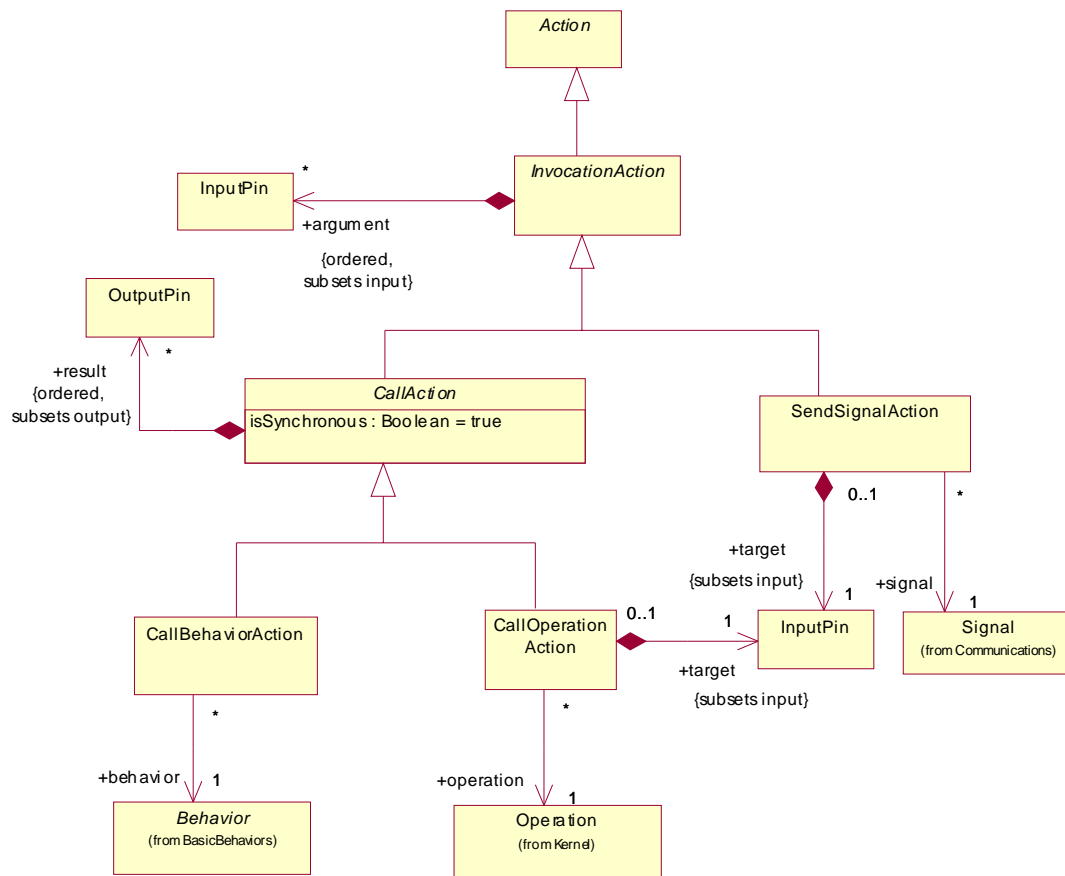


Figure 11.4 - Basic invocation actions

## Package IntermediateActions

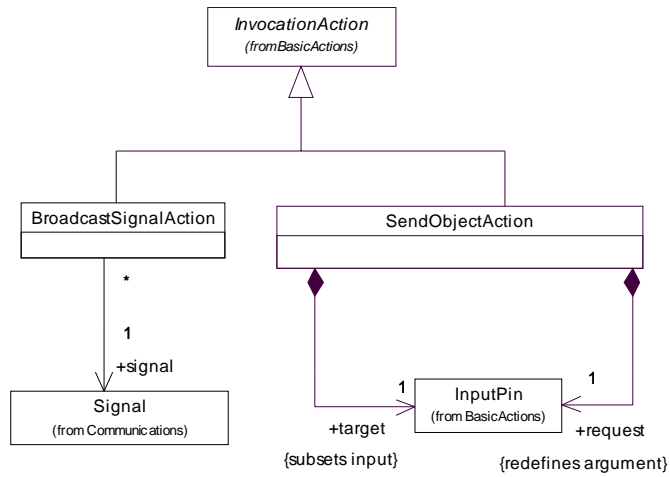


Figure 11.5 - Intermediate invocation actions

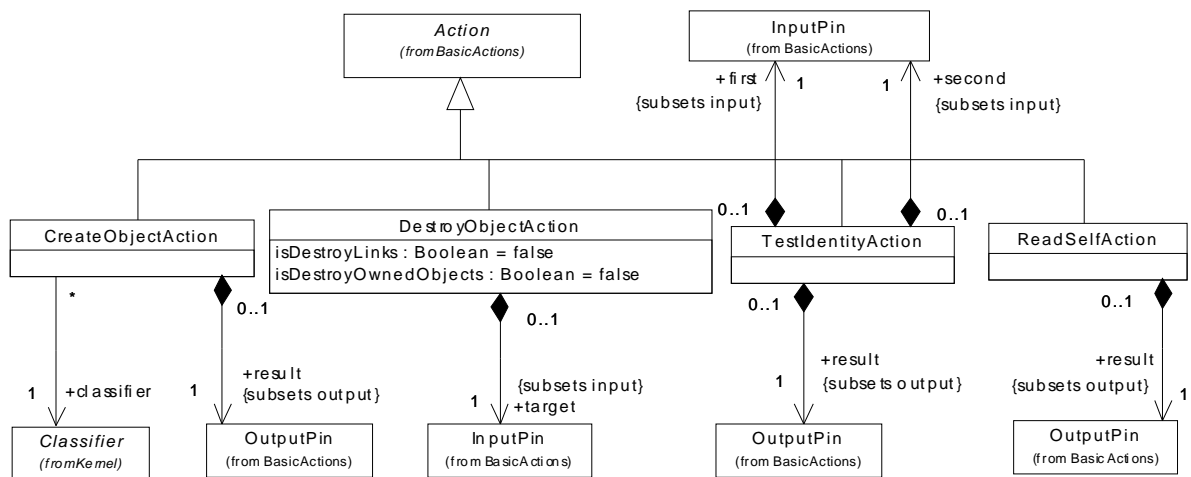
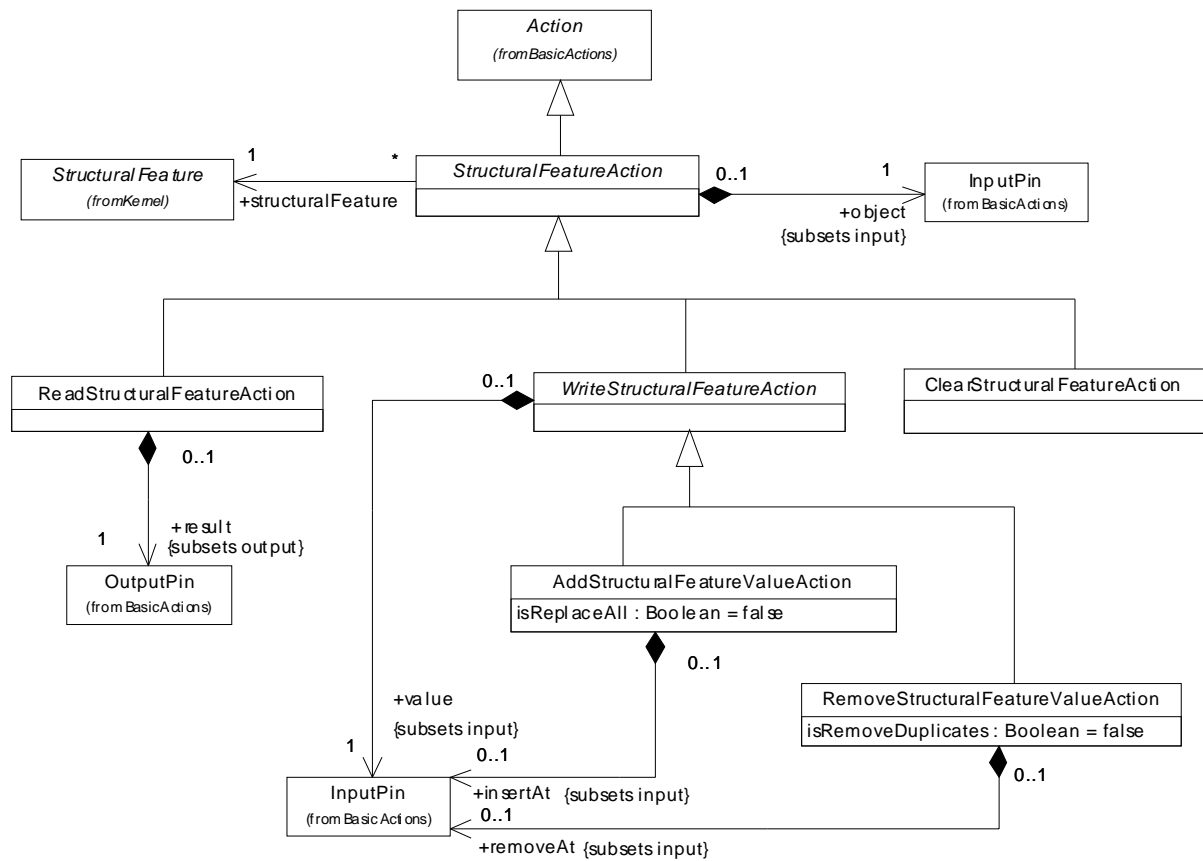
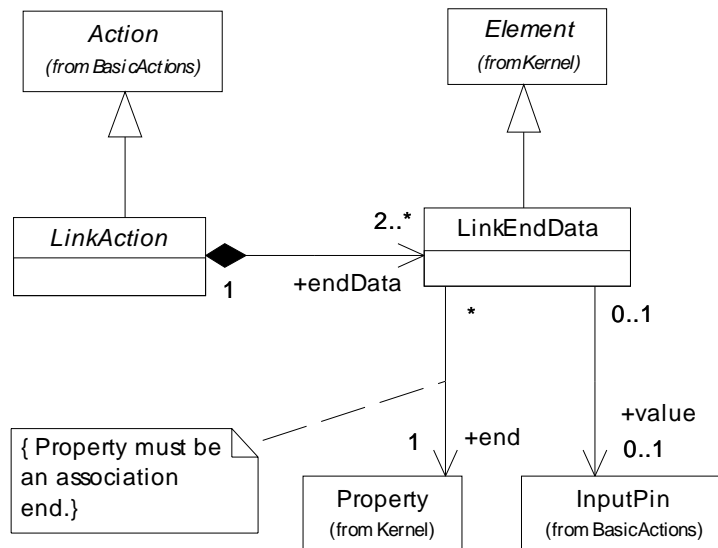


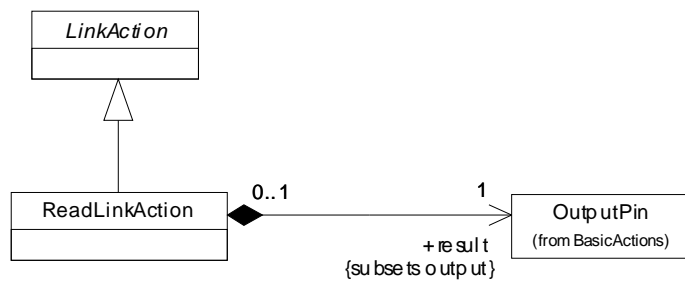
Figure 11.6 - Object actions



**Figure 11.7 - Structural Feature Actions**



**Figure 11.8 - Link identification**



**Figure 11.9 - Read link actions**

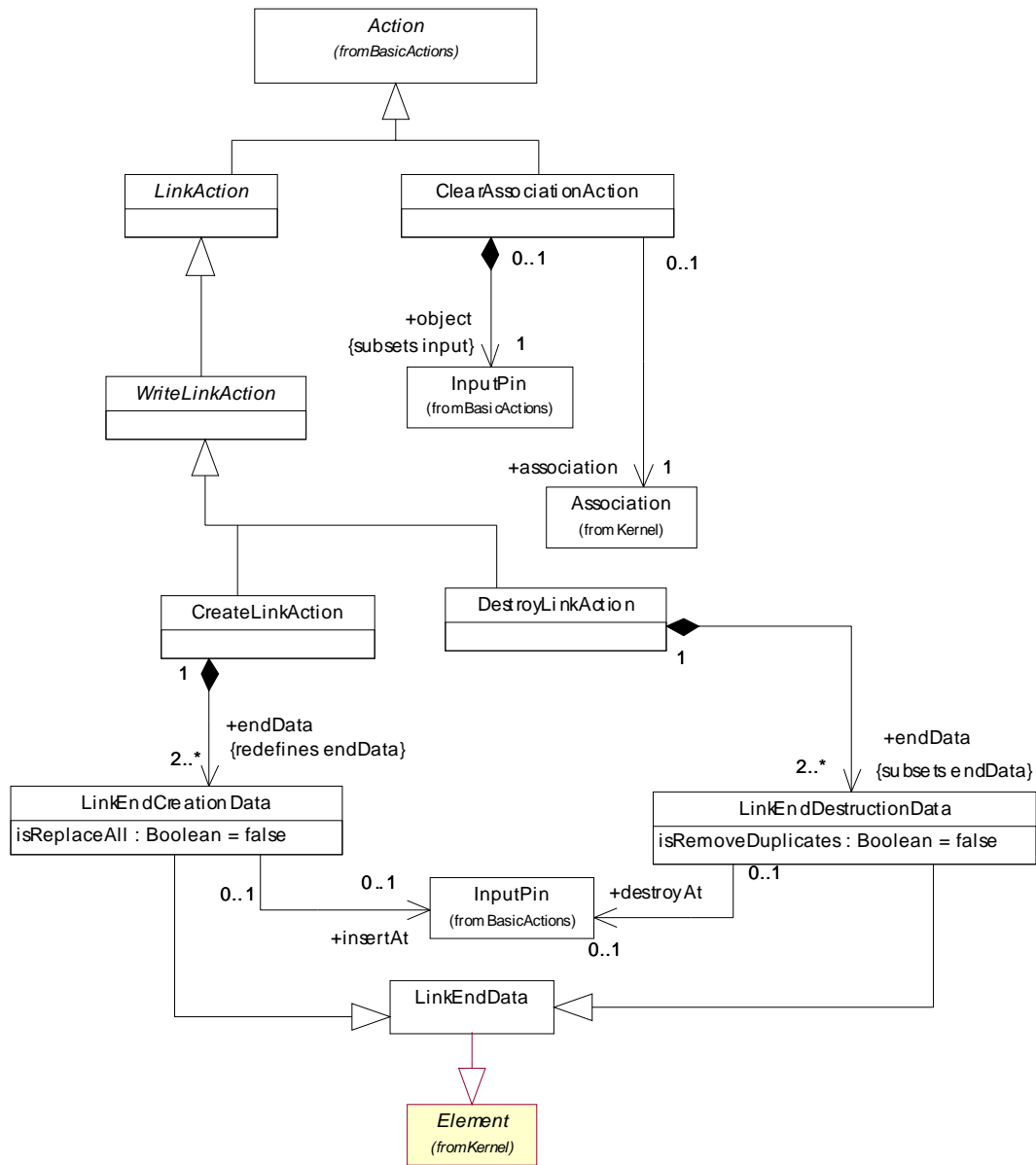
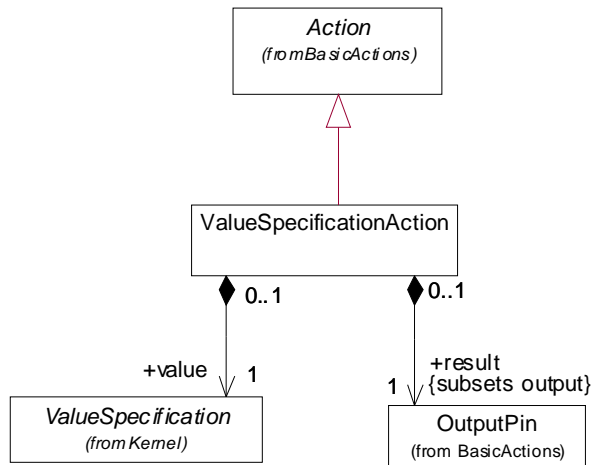


Figure 11.10 - Write link actions



**Figure 11.11 - Miscellaneous actions**

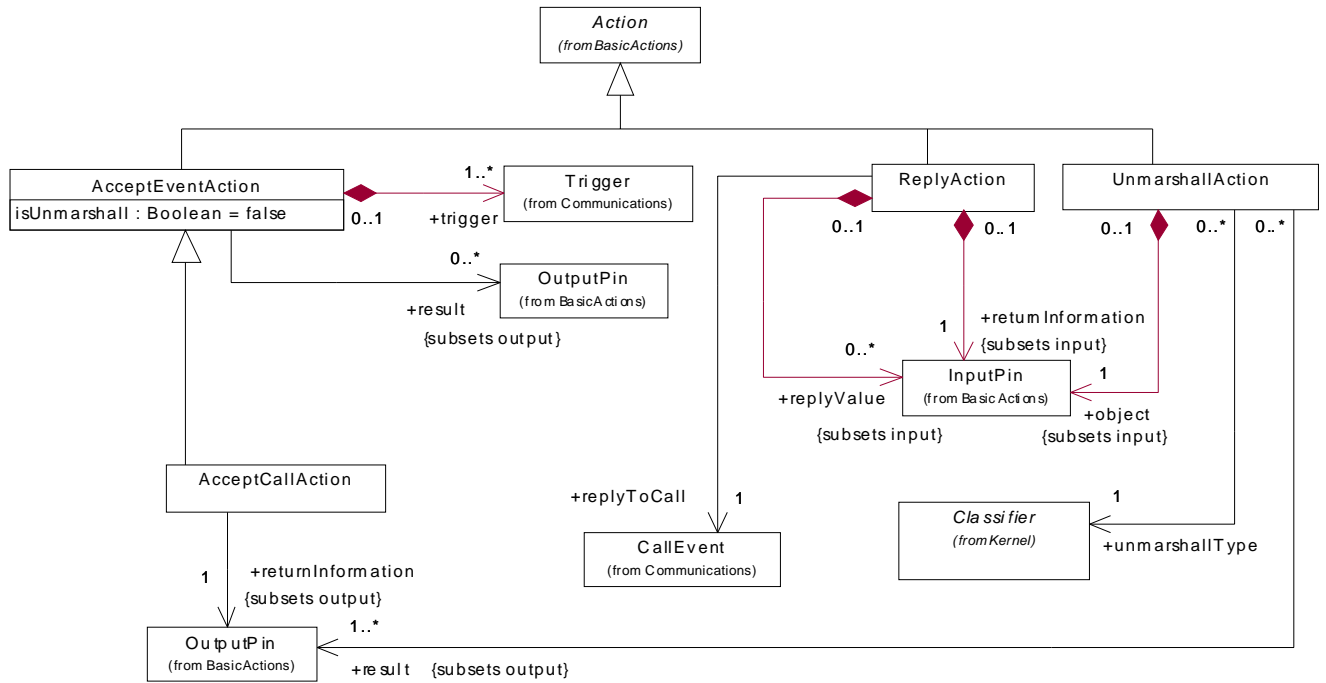


Figure 11.12 - Accept event actions



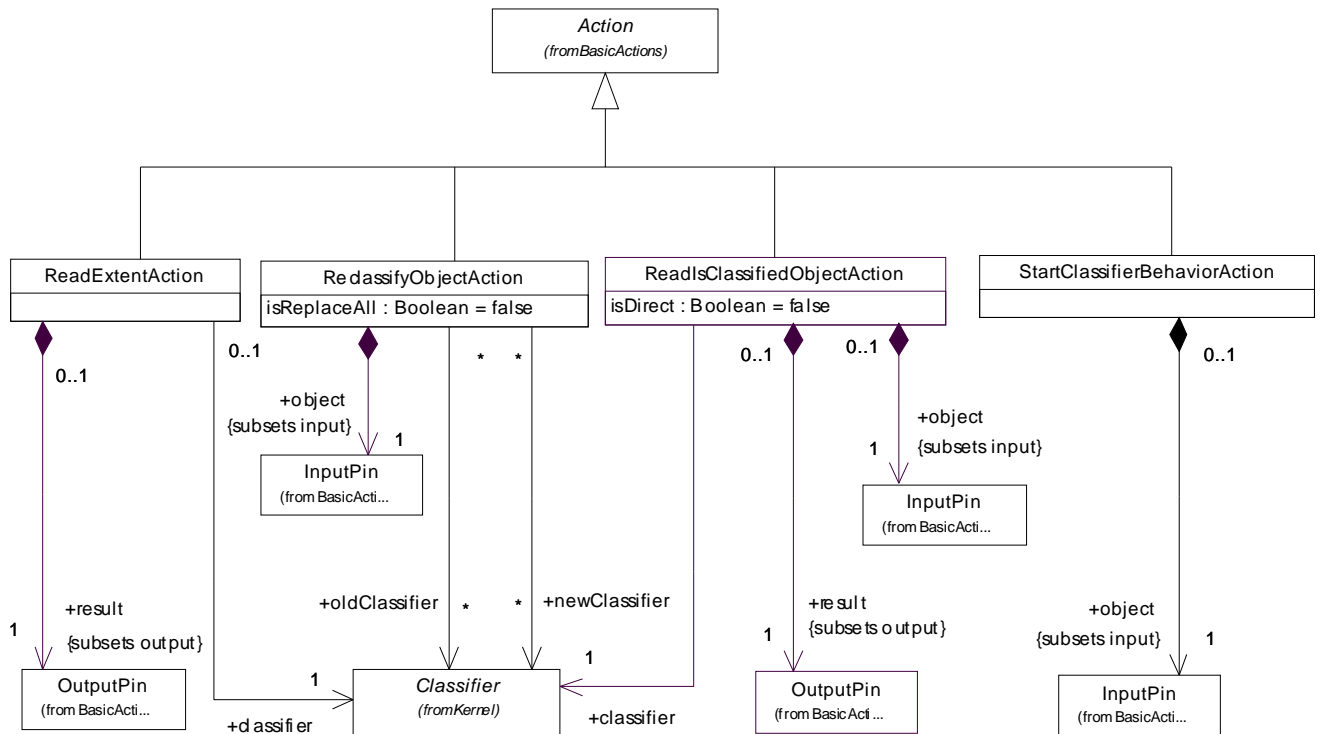


Figure 11.13 - Object actions (CompleteActions)

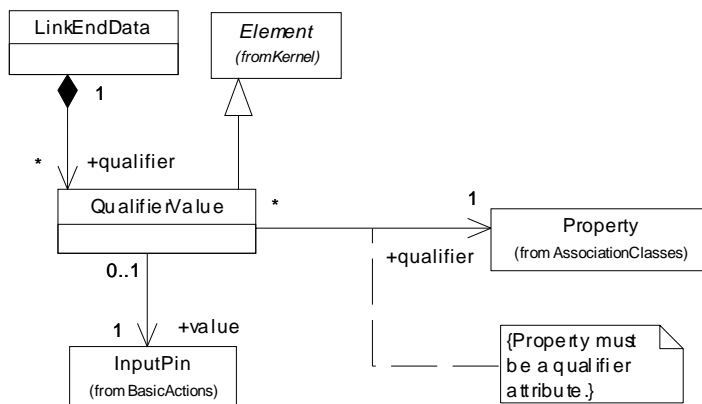


Figure 11.14 - Link identification (CompleteActions)

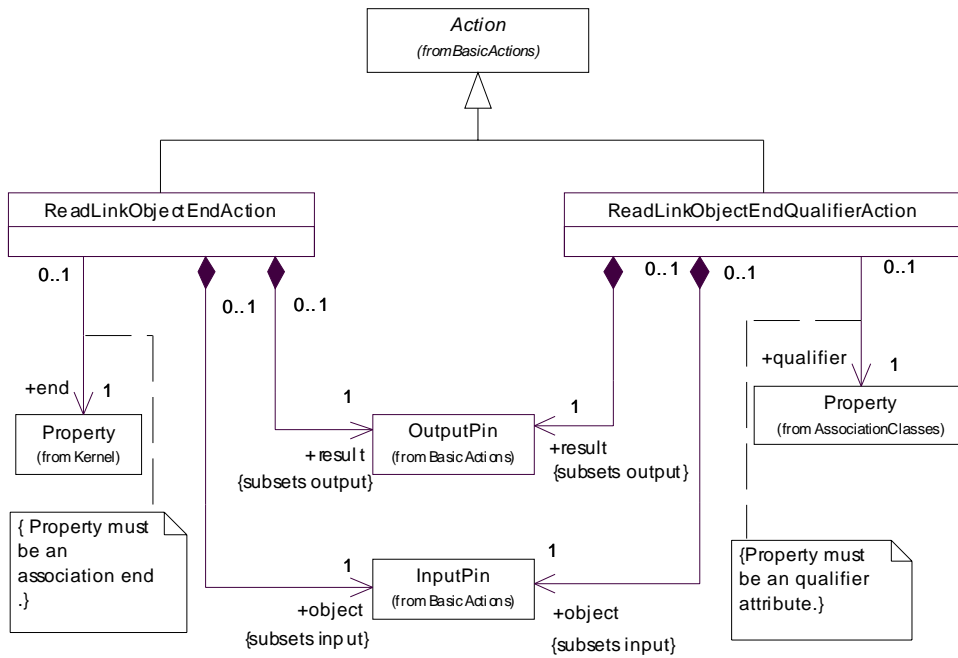


Figure 11.15 - Read link actions (CompleteActions)

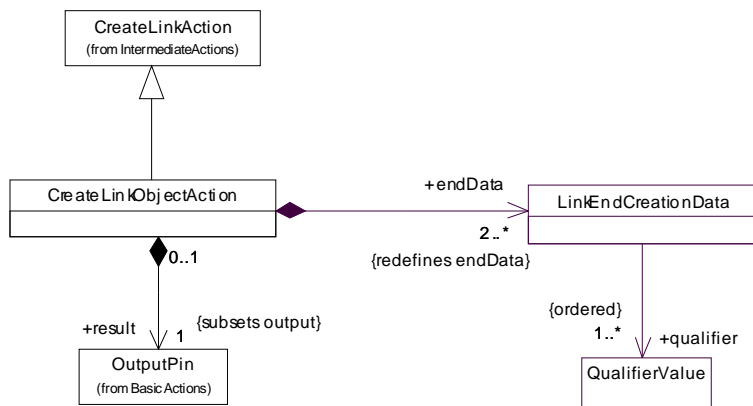


Figure 11.16 - Write link actions (CompleteActions)

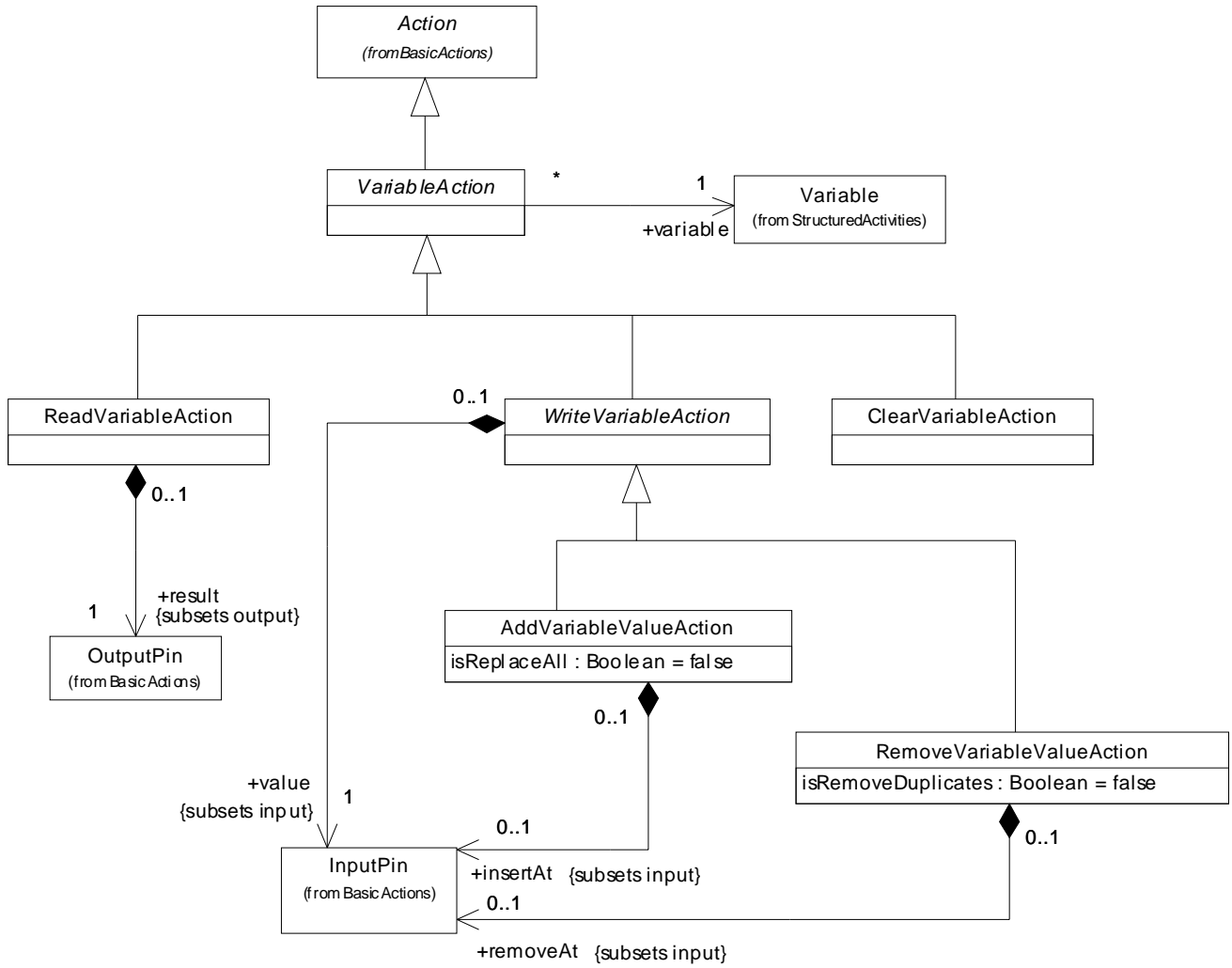


Figure 11.17 - Variable actions

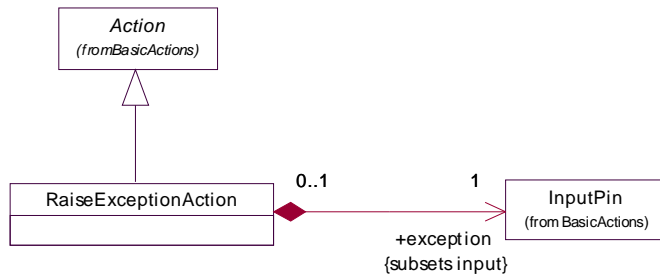


Figure 11.18 - Raise exception action

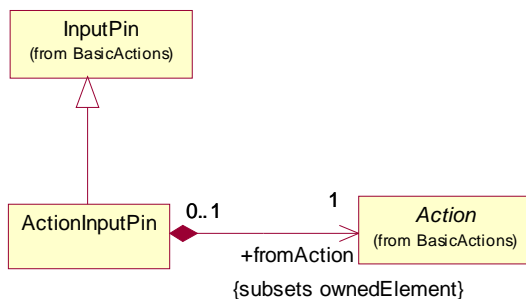


Figure 11.19 - Action input pin

## 11.3 Class Descriptions

### 11.3.1 AcceptCallAction (from CompleteActions)

#### Generalizations

- “AcceptEventAction (from CompleteActions)” on page 228

#### Description

AcceptCallAction is an accept event action representing the receipt of a synchronous call request. In addition to the normal operation parameters, the action produces an output that is needed later to supply the information to the ReplyAction necessary to return control to the caller.

This action is for synchronous calls. If it is used to handle an asynchronous call, execution of the subsequent reply action will complete immediately with no effects.

#### Attributes

No additional attributes

## Associations

- **returnInformation:** OutputPin [1..1] — Pin where a value is placed containing sufficient information to perform a subsequent reply and return control to the caller. The contents of this value are opaque. It can be passed and copied but it cannot be manipulated by the model.

## Constraints

- [1] The result pins must match the in and inout parameters of the operation specified by the trigger event in number, type, and order.
- [2] The trigger event must be a CallEvent.  
`trigger.event.ocllsKindOf(CallEvent)`
- [3] `isUnmarshall` must be true for an AcceptCallAction.  
`isUnmarshall = true`

## Semantics

This action accepts (event) occurrences representing the receipt of calls on the operation specified by the trigger call event. If an ongoing activity behavior has executed an accept call action that has not completed and the owning object has an event occurrence representing a call of the specified operation, the accept call action claims the event occurrence and removes it from the owning object. If several accept call actions concurrently request a call on the same operation, it is unspecified which one claims the event occurrence, but one and only one action will claim the event. The argument values of the call are placed on the result output pins of the action. Information sufficient to perform a subsequent reply action is placed in the returnInformation output pin. The execution of the accept call action is then complete. This return information value is opaque and may only be used by ReplyAction.

Note that the target class must not define a method for the operation being received. Otherwise, the operation call will be dispatched to that method instead of being put in the event buffer to be handled by AcceptCallAction. In general, classes determine how operation calls are handled, namely by a method, by a behavior owned directly by the class, by a state machine transition, and so on. The class must ensure that the operation call is handled in a way that AcceptCallAction has access to the call event.

### 11.3.2 AcceptEventAction (from CompleteActions)

#### Generalizations

- “Action (from BasicActions)” on page 230

#### Description

AcceptEventAction is an action that waits for the occurrence of an event meeting specified condition.

#### Attributes

- **isUnmarshall :** Boolean = false      Indicates whether there is a single output pin for the event, or multiple output pins for attributes of the event.

#### Associations

- **trigger :** Trigger [1..\*]      The type of events accepted by the action, as specified by triggers. For triggers with signal events, a signal of the specified type or any subtype of the specified signal type is accepted.

- result: OutputPin [0..\*] Pins holding the received event objects or their attributes. Event objects may be copied in transmission, so identity might not be preserved.

## Constraints

- [1] AcceptEventActions may have no input pins.
- [2] There are no output pins if the trigger events are only ChangeEvents, or if they are only CallEvents when this class is AcceptEventAction and not one of its children.
- [3] If the trigger events are all TimeEvents, there is exactly one output pin.
- [4] If isUnmarshalled is true, there must be exactly one trigger for events of type SignalEvent. The number of result output pins must be the same as the number of attributes of the signal. The type and ordering of each result output pin must be the same as the corresponding attribute of the signal. The multiplicity of each result output pin must be compatible with the multiplicity of the corresponding attribute.

## Semantics

Accept event actions handle event occurrences detected by the object owning the behavior (also see “InterruptibleActivityRegion (from CompleteActivities)” on page 366). Event occurrences are detected by objects independently of actions and the occurrences are stored by the object. The arrangement of detected event occurrences is not defined, but it is expected that extensions or profiles will specify such arrangements. If the accept event action is executed and the object detected an event occurrence matching one of the triggers on the action and the occurrence has not been accepted by another action or otherwise consumed by another behavior, then the accept event action completes and outputs a value describing the occurrence. If such a matching occurrence is not available, the action waits until such an occurrence becomes available, at which point the action may accept it. In a system with concurrency, several actions or other behaviors might contend for an available event occurrence. Unless otherwise specified by an extension or profile, only one action accepts a given occurrence, even if the occurrence would satisfy multiple concurrently executing actions.

If the occurrence is a signal event occurrence and unmarshall is false, the result value contains a signal object whose reception by the owning object caused the occurrence. If the occurrence is a signal event occurrence and isUnmarshall is true, the attribute values of the signal are placed on the result output pins of the action. Signal objects may be copied in transmission and storage by the owning object, so identity might not be preserved. An action whose trigger is a SignalTrigger is informally called an accept signal action. If the occurrence is a time event occurrence, the result value contains the time at which the occurrence transpired. Such an action is informally called a wait time action. If the occurrences are all occurrences of ChangeEvent, or all CallEvent, or a combination of these, there are no output pins (however, see “AcceptCallAction (from CompleteActions)” on page 227). See CommonBehavior for a description of Event specifications. If the triggers are a combination of SignalEvents and ChangeEvents, the result is a null value if a change event occurrence or a call event occurrence is accepted.

This action handles asynchronous messages, including asynchronous calls. It cannot be used with synchronous calls (except see “AcceptCallAction (from CompleteActions)” on page 227).

## Notation

An accept event action is notated with a concave pentagon. A wait time action is notated with an hour glass.



**Figure 11.20 - Accept event notations.**

## Examples

“AcceptEventAction (as specialized)” on page 299

## Rationale

Accept event actions are introduced to handle processing of events during the execution of a behavior.

## Changes from previous UML

AcceptEventAction is new in UML 2.0.

### 11.3.3 Action (from BasicActions)

#### Generalizations

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

#### Description

An action is a named element that is the fundamental unit of executable functionality. The execution of an action represents some transformation or processing in the modeled system, be it a computer system or otherwise.

#### Attributes

No additional attributes

#### Associations

- `/input : InputPin [*]` The ordered set of input pins connected to the Action. These are among the total set of inputs.
- `/output : OutputPin [*]` The ordered set of output pins connected to the Action. The action places its results onto pins in this set.
- `/context : Classifier [1]` The classifier that owns the behavior of which this action is a part.

#### Constraints

No additional constraints

## Semantics

An action execution represents the run-time behavior of executing an action within a specific behavior execution. As Action is an abstract class, all action executions will be executions of specific kinds of actions. When the action executes, and what its actual inputs are, is determined by the concrete action and the behaviors in which it is used.

## Notation

No specific notation. See extensions in Activities chapter.

## Changes from previous UML

Action is the same concept as in UML 1.5, but modeled independently of the behaviors that use it.

### 11.3.4 ActionInputPin (from StructuredActions)

#### Generalizations

- “InputPin (from BasicActions)” on page 249

#### Description

An action input pin is a kind of pin that executes an action to determine the values to input to another.

#### Attributes

No additional attributes

#### Associations

- fromAction : Action [1]      The action used to provide values.

#### Constraints

- [1] The fromAction of an action input pin must have exactly one output pin.
- [2] The fromAction of an action input pin must only have action input pins as input pins.
- [3] The fromAction of an action input pin cannot have control or data flows coming into or out of it or its pins.

## Semantics

If an action is otherwise enabled, the fromActions on action input pins are enabled. The outputs of these are used as the values of the corresponding input pins. The process recurs on the input pins of the fromActions, if they also have action input pins. The recursion bottoms out at actions that have no inputs, such as for read variables or the self object. This forms a tree that is an action model for nested expressions.

## Notation

No specific notation

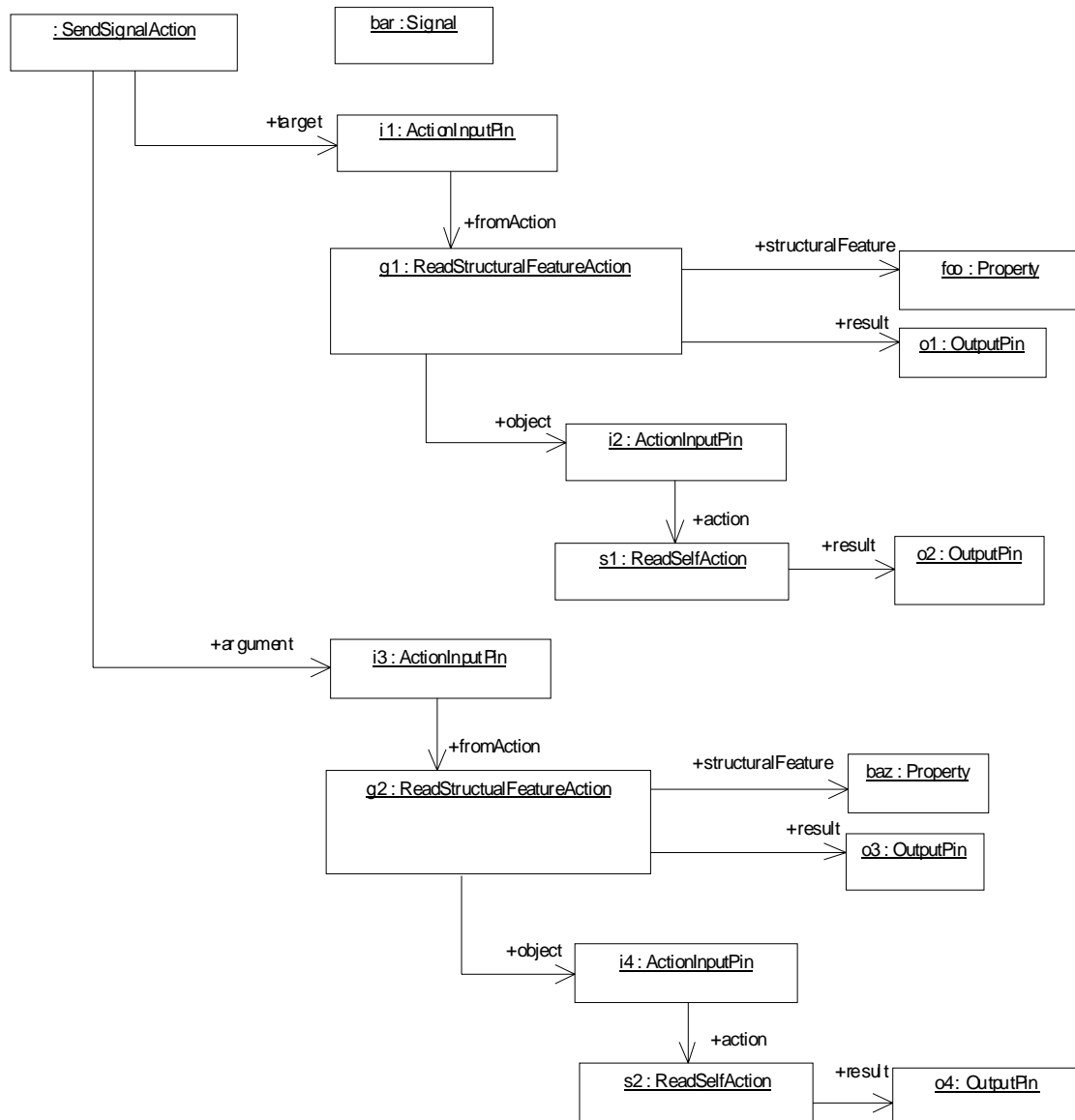
## Example

Example (in action language provided just for example, not normative):

```
self.foo->bar(self.baz);
```



meaning get the foo attribute of self, then send a bar signal to it with argument from the baz attribute of self. The repository model is shown below.



**Figure 11.21 - Example repository model**

### Rationale

ActionInputPin is introduced to pass values between actions in expressions without using flows.

### 11.3.5 AddStructuralFeatureValueAction (from IntermediateActions)

AddStructuralFeatureValueAction is a write structural feature action for adding values to a structural feature.

#### Generalizations

- “WriteStructuralFeatureAction (from IntermediateActions)” on page 282.

#### Description

Structural Features are potentially multi-valued and ordered, so the action supports specification of insertion points for new values. It also supports the removal of existing values of the structural feature before the new value is added.

The object to access is specified dynamically, by referring to an input pin on which the object will be placed at runtime. The type of the value of this pin is the classifier that owns the specified structural feature, and the value’s multiplicity is 1..1.

#### Attributes

- isReplaceAll : Boolean [1..1] = false      Specifies whether existing values of the structural feature of the object should be removed before adding the new value.

#### Associations

- insertAt : InputPin [0..1] (Specialized from Action:input)  
Gives the position at which to insert a new value or move an existing value in ordered structural features. The type of the pin is UnlimitedNatural, but the value cannot be zero. This pin is omitted for unordered structural features.

#### Constraints

- [1] Actions adding a value to ordered structural features must have a single input pin for the insertion point with type UnlimitedNatural and multiplicity of 1..1, otherwise the action has no input pin for the insertion point.

```
let insertAtPins : Collection = self.insertAt in
  if self.structuralFeature.isOrdered = #false
  then insertAtPins->size() = 0
  else let insertAtPin : InputPin = insertAt->asSequence()->first() in
    insertAtPins->size() = 1
    and insertAtPin.type = UnlimitedNatural
    and insertAtPin.multiplicity.is(1,1))
  endif
```

#### Semantics

If isReplaceAll is true, then the existing values of the structural feature are removed before the new one added, except if the new value already exists, then it is not removed under this option. If isReplaceAll is false and the structural feature is unordered and non-unique, then adding an existing value has no effect. If the feature is an association end, the semantics are the same as creating a link, the participants of which are the object owning the structural feature and the new value.

Values of a structural feature may be ordered or unordered, even if the multiplicity maximum is 1. Adding values to ordered structural features requires an insertion point for a new value using the insertAt input pin. The insertion point is a positive integer giving the position to insert the value, or unlimited, to insert at the end. A positive integer less than or equal to the current number of values means to insert the new value at that position in the sequence of existing values,

with the integer one meaning the new value will be first in the sequence. A value of unlimited for insertAt means to insert the new value at the end of the sequence. The semantics is undefined for a value of zero or an integer greater than the number of existing values. The insertion point is required for ordered structural features and omitted for unordered structural features. Reinserting an existing value at a new position in an ordered unique structural feature moves the value to that position (this works because structural feature values are sets). The insertion point is ignored when replacing all values.

The semantics is undefined for adding a value that violates the upper multiplicity of the structural feature. Removing a value succeeds even when that violates the minimum multiplicity—the same as if the minimum were zero. The modeler must determine when minimum multiplicity of structural features should be enforced.

The semantics is undefined for adding a new value for a structural feature with isReadOnly=true after initialization of the owning object.

## Notation

No specific notation

## Rationale

AddStructuralFeatureValueAction is introduced to add structural feature values. isReplaceAll is introduced to replace and add in a single action, with no intermediate states of the object where only some of the existing values are present.

## Changes from previous UML

AddStructuralFeatureValueAction is new in UML 2.0. It generalizes AddAttributeAction in UML 1.5.

### 11.3.6 AddVariableValueAction (from StructuredActions)

AddVariableValueAction is a write variable action for adding values to a variable.

## Generalizations

- “WriteVariableAction (from StructuredActions)” on page 283

## Description

Variables are potentially multi-valued and ordered, so the action supports specification of insertion points for new values. It also supports the removal of existing values of the variable before the new value is added.

## Attributes

- isReplaceAll : Boolean [1..1] = false      Specifies whether existing values of the variable should be removed before adding the new value.

## Associations

- insertAt : InputPin [0..1] (Specialized from Action:input)  
Gives the position at which to insert a new value or move an existing value in ordered variables. The type is UnlimitedINatural, but the value cannot be zero. This pin is omitted for unordered variables.

## Constraints

- [1] Actions adding values to ordered variables must have a single input pin for the insertion point with type `UnlimitedNatural` and multiplicity of 1..1, otherwise the action has no input pin for the insertion point.

```
let insertAtPins : Collection = self.insertAt in
    if self.variable.ordering = #unordered
    then insertAtPins->size() = 0
    else let insertAtPin : InputPin = insertAt->asSequence()->first() in
        insertAtPins->size() = 1
        and insertAtPin.type = UnlimitedNatural
        and insertAtPin.multiplicity.is(1,1))
    endif
```

## Semantics

If `isReplaceAll` is true, then the existing values of the variable are removed before the new one added, except if the new value already exists, then it is not removed under this option. If `isReplaceAll` is false and the variable is unordered and non-unique, then adding an existing value has no effect.

Values of a variable may be ordered or unordered, even if the multiplicity maximum is 1. Adding values to ordered variables requires an insertion point for a new value using the `insertAt` input pin. The insertion point is a positive integer giving the position to insert the value, or unlimited, to insert at the end. A positive integer less than or equal to the current number of values means to insert the new value at that position in the sequence of existing values, with the integer one meaning the new value will be first in the sequence. A value of unlimited for `insertAt` means to insert the new value at the end of the sequence. The semantics is undefined for a value of zero or an integer greater than the number of existing values. The insertion point is required for ordered variables and omitted for unordered variables. Reinserting an existing value at a new position in an ordered unique variable moves the value to that position (this works because variable values are sets).

The semantics is undefined for adding a value that violates the upper multiplicity of the variable. Removing a value succeeds even when that violates the minimum multiplicity—the same as if the minimum were zero. The modeler must determine when minimum multiplicity of variables should be enforced.

## Notation

No specific notation

## Rationale

`AddVariableValueAction` is introduced to add variable values. `isReplaceAll` is introduced to replace and add in a single action, with no intermediate states of the variable where only some of the existing values are present.

## Changes from previous UML

`AddVariableValueAction` is unchanged from UML 1.5.

### 11.3.7 BroadcastSignalAction (from IntermediateActions)

#### Generalizations

- “`InvocationAction` (from `BasicActions`)” on page 249

## Description

BroadcastSignalAction is an action that transmits a signal instance to all the potential target objects in the system, which may cause the firing of a state machine transitions or the execution of associated activities of a target object. The argument values are available to the execution of associated behaviors. The requestor continues execution immediately after the signals are sent out. It does not wait for receipt. Any reply messages are ignored and are not transmitted to the requestor.

## Attributes

No additional attributes

## Associations

- signal: Signal [1]      The specification of signal object transmitted to the target objects.

## Constraints

- [1] The number and order of argument pins must be the same as the number and order of attributes in the signal.
- [2] The type, ordering, and multiplicity of an argument pin must be the same as the corresponding attribute of the signal.

## Semantics

When all the prerequisites of the action execution are satisfied, a signal object is generated from the argument values according to signal and this signal object is transmitted concurrently to each of the implicit broadcast target objects in the system. The manner of identifying the set of objects that are broadcast targets is a semantic variation point and may be limited to some subset of all the objects that exist. There is no restriction on the location of target objects. The manner of transmitting the signal object, the amount of time required to transmit it, the order in which the transmissions reach the various target objects, and the path for reaching the target objects are undefined.

- [1] When a transmission arrives at a target object, it may invoke a behavior in the target object. The effect of receiving such transmission is specified in Chapter 13, “Common Behaviors.” Such effects include executing activities and firing state machine transitions.
- [2] A broadcast signal action receives no reply from the invoked behavior; any attempted reply is simply ignored, and no transmission is performed to the requestor.

## Semantic Variation Points

The determination of the set of broadcast target objects is a semantic variation point.

## Notation

No specific notation

## Rationale

Sends a signal to a set of system defined target objects.

## Changes from previous UML

Same as UML 1.5.

### 11.3.8 CallAction (from BasicActions)

#### Generalizations

- “InvocationAction (from BasicActions)” on page 249.

#### Description

CallAction is an abstract class for actions that invoke behavior and receive return values.

#### Attributes

- isSynchronous: Boolean      If *true*, the call is synchronous and the caller waits for completion of the invoked behavior. If *false*, the call is asynchronous and the caller proceeds immediately and does not expect a return value.

#### Associations

- result: OutputPin [0..\*]      A list of output pins where the results of performing the invocation are placed.

#### Constraints

- [1] Only synchronous call actions can have result pins.
- [2] The number and order of argument pins must be the same as the number and order of parameters of the invoked behavior or behavioral feature. Pins are matched to parameters by order.
- [3] The type, ordering, and multiplicity of an argument pin must be the same as the corresponding parameter of the behavior or behavioral feature.

#### Semantics

Parameters on behaviors and operations are totally ordered lists. To match parameters to pins on call actions, select the sublist of that list that corresponds to in and inout owned parameters (i.e., Behavior.ownedParameter). The input pins on Action::input are matched in order against these parameters in the sublist order. Then take the sublist of the parameter list that corresponds to out, inout, and return parameters. The output pins on Action::output are matched in order against these parameters in sublist order.

See children of CallAction.

### 11.3.9 CallBehaviorAction (from BasicActions)

#### Generalizations

- “CallAction (from BasicActions)” on page 237

#### Description

CallBehaviorAction is a call action that invokes a behavior directly rather than invoking a behavioral feature that, in turn, results in the invocation of that behavior. The argument values of the action are available to the execution of the invoked behavior. The execution of the call behavior action waits until the execution of the invoked behavior completes and a result is returned on its output pin. In particular, the invoked behavior may be an activity.

## Attributes

No additional attributes

## Associations

- behavior : Behavior [1..1]      The invoked behavior. It must be capable of accepting and returning control.

## Constraints

- [1] The number of argument pins and the number of parameters of the behavior of type *in* and *in-out* must be equal.
- [2] The number of result pins and the number of parameters of the behavior of type *return*, *out*, and *in-out* must be equal.
- [3] The type, ordering, and multiplicity of an argument or result pin is derived from the corresponding parameter of the behavior.

## Semantics

- [1] When all the prerequisites of the action execution are satisfied, *CallBehaviorAction* invokes its specified behavior with the values on the input pins as arguments. When the behavior is finished, the output values are put on the output pins. Each parameter of the behavior of the action provides output to a pin or takes input from one. No other implementation specifics are implied, such as call stacks, and so on. See “Pin (from BasicActions)” on page 256.
- [2] If the call is asynchronous, the action completes immediately. Execution of the invoked behavior proceeds without any further dependency on the execution of the behavior containing the invoking action. Once the invocation of the behavior has been initiated, execution of the asynchronous action is complete.
- [3] An asynchronous invocation completes when its behavior is started, or is at least ensured to be started at some point. Any return or out values from the invoked behavior are not passed back to the containing behavior. When an asynchronous invocation is done, the containing behavior continues regardless of the status of the invoked behavior. For example, the containing behavior may complete even though the invoked behavior is not finished.
- [4] If the call is synchronous, execution of the calling action is blocked until it receives a reply from the invoked behavior. The reply includes values for any return, out, or inout parameters.
- [5] If the call is synchronous, when the execution of the invoked behavior completes, the result values are placed on the result pins of the call behavior action, and the execution of the action is complete (*StructuredActions*, *ExtraStructuredActivities*). If the execution of the invoked behavior yields an exception, the exception is transmitted to the call behavior action to begin search for a handler. See *RaiseExceptionAction*.

## Notation

See specialization of “*CallBehaviorAction* (as specialized)” on page 337.

## Presentation Options

See specialization of “*CallBehaviorAction* (as specialized)” on page 337.

## Rationale

Invokes a behavior directly without the need for a behavioral feature.

## Changes from previous UML

Same as UML 1.5

### 11.3.10 CallOperationAction (from BasicActions)

#### Generalizations

- “CallAction (from BasicActions)” on page 237

#### Description

CallOperationAction is an action that transmits an operation call request to the target object, where it may cause the invocation of associated behavior. The argument values of the action are available to the execution of the invoked behavior. If the action is marked synchronous, the execution of the call operation action waits until the execution of the invoked behavior completes and a reply transmission is returned to the caller; otherwise execution of the action is complete when the invocation of the operation is established and the execution of the invoked operation proceeds concurrently with the execution of the calling behavior. Any values returned as part of the reply transmission are put on the result output pins of the call operation action. Upon receipt of the reply transmission, execution of the call operation action is complete.

#### Attributes

No additional attributes

#### Associations

- operation: Operation [1] The operation to be invoked by the action execution.
- target: InputPin [1] The target object to which the request is sent. The classifier of the target object is used to dynamically determine a behavior to invoke. This object constitutes the context of the execution of the operation.

#### Constraints

- [1] The number of argument pins and the number of owned parameters of the operation of type *in* and *in-out* must be equal.
- [2] The number of result pins and the number of owned parameters of the operation of type *return*, *out*, and *in-out* must be equal.
- [3] The type, ordering, and multiplicity of an argument or result pin is derived from the corresponding owned parameter of the operation.
- [4] The type of the target pin must be the same as the type that owns the operation.

#### Semantics

The inputs to the action determine the target object and additional actual arguments of the call.

- [1] When all the prerequisites of the action execution are satisfied, information comprising the operation and the argument pin values of the action execution is created and transmitted to the target object. The target objects may be local or remote. The manner of transmitting the call, the amount of time required to transmit it, the order in which the transmissions reach the various target objects, and the path for reaching the target objects are undefined.
- [2] When a call arrives at a target object, it may invoke a behavior in the target object. The effect of receiving such call is specified in Chapter 13, “Common Behaviors.” Such effects include executing activities and firing state machine transitions.
- [3] If the call is synchronous, when the execution of the invoked behavior completes, its return results are transmitted back as a reply to the calling action execution. The manner of transmitting the reply, the time required for transmission, the



representation of the reply transmission, and the transmission path are unspecified. If the execution of the invoked behavior yields an exception, the exception is transmitted to the caller where it is reraised as an exception in the execution of the calling action. Possible exception types may be specified by attaching them to the called Operation using the *raisedException* association.

- [4] If the call is asynchronous, the caller proceeds immediately and the execution of the call operation action is complete. Any return or out values from the invoked operation are not passed back to the containing behavior. If the call is synchronous, the caller is blocked from further execution until it receives a reply from the invoked behavior.
- [5] When the reply transmission arrives at the invoking action execution, the return result values are placed on the result pins of the call operation action, and the execution of the action is complete.

### **Semantic Variation Points**

The mechanism for determining the method to be invoked as a result of a call operation is unspecified.

### **Notation**

See “CallOperationAction (as specialized)” on page 339

### **Presentation Options**

See “CallOperationAction (as specialized)” on page 339

### **Rationale**

Calls an operation on a specified target object.

### **Changes from previous UML**

Same as UML 1.5.

## **11.3.11 ClearAssociationAction (from IntermediateActions)**

ClearAssociationAction is an action that destroys all links of an association in which a particular object participates.

### **Generalizations**

- “Action (from BasicActions)” on page 230

### **Description**

This action destroys all links of an association that have a particular object at one end.

### **Attributes**

No additional attributes

### **Associations**

- association : Association [1..1] Association to be cleared.
- object : InputPin [1..1] (Specialized from Action:input) Gives the input pin from which is obtained the object whose participation in the association is to be cleared.

## Constraints

- [1] The type of the input pin must be the same as the type of at least one of the association ends of the association.  
`self.association->exists(end.type = self.object.type)`
- [2] The multiplicity of the input pin is 1..1.  
`self.object.multiplicity.is(1,1)`

## Semantics

This action has a statically-specified association. It has an input pin for a runtime object that must be of the same type as at least one of the association ends of the association. All links of the association in which the object participates are destroyed even when that violates the minimum multiplicity of any of the association ends. If the association is a class, then link object identities are destroyed.

## Notation

No specific notation

## Rationale

ClearAssociationAction is introduced to remove all links from an association in which an object participates in a single action, with no intermediate states where only some of the existing links are present.

## Changes from previous UML

ClearAssociationAction is unchanged from UML 1.5.

### 11.3.12 ClearStructuralFeatureAction (from IntermediateActions)

ClearStructuralFeatureAction is a structural feature action that removes all values of a structural feature.

## Generalizations

- “StructuralFeatureAction (from IntermediateActions)” on page 275

## Description

This action removes all values of a structural feature.

## Attributes

No additional attributes

## Associations

No additional associations

## Constraints

No additional constraints

## Semantics

All values are removed even when that violates the minimum multiplicity of the structural feature—the same as if the minimum were zero. The semantics is undefined for a structural feature with `isReadOnly = true` after initialization of the object owning the structural feature, unless the structural feature has no values. The action has no effect if the structural feature has no values. If the feature is an association end, the semantics are the same as for `ClearAssociationAction` on the object owning the structural feature.

## Notation

No specific notation

## Rationale

`ClearStructuralFeatureAction` is introduced to remove all values from a structural feature in a single action, with no intermediate states where only some of the existing values are present.

## Changes from previous UML

`ClearStructuralFeatureAction` is new in UML 2.0. It generalizes `ClearAttributeAction` from UML 1.5.

### 11.3.13 ClearVariableAction (from StructuredActions)

`ClearVariableAction` is a variable action that removes all values of a variable.

## Generalizations

- “VariableAction (from StructuredActions)” on page 281

## Description

This action removes all values of a variable.

## Attributes

No additional attributes

## Associations

No additional associations

## Constraints

No additional constraints

## Semantics

All values are removed even when that violates the minimum multiplicity of the variable—the same as if the minimum were zero.

## Notation

No specific notation

## Rationale

ClearVariableAction is introduced to remove all values from a variable in a single action, with no intermediate states where only some of the existing values are present.

## Changes from previous UML

ClearVariableAction is unchanged from UML 1.5.

### 11.3.14 CreateLinkAction (from IntermediateActions)

(IntermediateActions) CreateLinkAction is a write link action for creating links.

## Generalizations

- “WriteLinkAction (from IntermediateActions)” on page 281

## Description

This action can be used to create links and link objects. There is no return value in either case. This is so that no change of the action is required if the association is changed to an association class or vice versa. CreateLinkAction uses a specialization of LinkEndData called LinkEndCreationData, to support ordered associations. The insertion point is specified at runtime by an additional input pin, which is required for ordered association ends and omitted for unordered ends. The insertion point is a positive integer giving the position to insert the link, or unlimited, to insert at the end. Reinserting an existing end at a new position in an ordered unique structural feature moves the end to that position.

CreateLinkAction also uses LinkEndCreationData to support the destruction of existing links of the association that connect any of the objects of the new link. When the link is created, this option is available on an end-by-end basis, and causes all links of the association emanating from the specified ends to be destroyed before the new link is created.

## Attributes

No additional attributes

## Associations

- endData : LinkEndCreationData [2..\*] (Redefined from LinkAction:endData)  
Specifies ends of association and inputs.

## Constraints

- [1] The association cannot be an abstract classifier.  
`self.association().isAbstract = #false`

## Semantics

CreateLinkAction creates a link or link object for an association or association class. It has no output pin, because links are not necessarily values that can be passed to and from actions. When the action creates a link object, the object could be returned on output pin, but it is not for consistency with links. This allows actions to remain unchanged when an association is changed to an association class or vice versa. The semantics of CreateLinkObjectAction applies to creating link objects with CreateLinkAction.

This action also supports the destruction of existing links of the association that connect any of the objects of the new link. This option is available on an end-by-end basis, and causes all links of the association emanating from the specified ends to be destroyed before the new link is created. If the link already exists, then it is not destroyed under this option. Otherwise, recreating an existing link has no effect if the structural feature is unordered and non-unique.

The semantics is undefined for creating a link for an association class that is abstract. The semantics is undefined for creating a link that violates the upper multiplicity of one of its association ends. A new link violates the upper multiplicity of an end if the cardinality of that end after the link is created would be greater than the upper multiplicity of that end. The cardinality of an end is equal to the number of links with objects participating in the other ends that are the same as those participating in those other ends in the new link, and with qualifier values on all ends the same as the new link, if any.

The semantics is undefined for creating a link that has an association end with `isReadOnly=true` after initialization of the other end objects, unless the link being created already exists. Objects participating in the association across from a writeable end can have links created as long as the objects across from all read only ends are still being initialized. This means that objects participating in links with two or more read only ends cannot have links created unless all the linked objects are being initialized.

Creating ordered association ends requires an insertion point for a new link using the `insertAt` input pin of `LinkEndCreationData`. The pin is of type `UnlimitedNatural` with multiplicity of 1..1. A pin value that is a positive integer less than or equal to the current number of links means to insert the new link at that position in the sequence of existing links, with the integer one meaning the new link will be first in the sequence. A value of unlimited for `insertAt` means to insert the new link at the end of the sequence. The semantics is undefined for value of zero or an integer greater than the number of existing links. The `insertAt` input pin does not exist for unordered association ends. Reinserting an existing end at a new position in an ordered unique structural feature moves the end so that it is in the position specified after the action is complete.

## Notation

No specific notation

## Rationale

`CreateLinkAction` is introduced to create links.

## Changes from previous UML

`CreateLinkAction` is unchanged from UML 1.5.

### 11.3.15 CreateLinkObjectAction (from CompleteActions)

`CreateLinkObjectAction` creates a link object.

## Generalizations

- “`CreateLinkAction` (from `IntermediateActions`)” on page 243

## Description

This action is exclusively for creating links of association classes. It returns the created link object.

## Attributes

No additional attributes

## Associations

- result [1..1] : OutputPin [1..1] (Specialized from Action:output)  
Gives the output pin on which the result is put.

## Constraints

- [1] The association must be an association class.  
`self.association().oclIsKindOf(Class)`
- [2] The type of the result pin must be the same as the association of the action.  
`self.result.type = self.association()`
- [3] The multiplicity of the output pin is 1..1.  
`self.result.multiplicity.is(1,1)`

## Semantics

CreateLinkObjectAction inherits the semantics of CreateLinkAction, except that it operates on association classes to create a link object. The additional semantics over CreateLinkAction is that the new or found link object is put on the output pin. If the link already exists, then the found link object is put on the output pin. The semantics of CreateObjectAction applies to creating link objects with CreateLinkObjectAction.

## Notation

No specific notation

## Rationale

CreateLinkObjectAction is introduced to create link objects in a way that returns the link object. Compare CreateLinkAction.

## Changes from previous UML

CreateLinkObjectAction is unchanged from UML 1.5.

### 11.3.16 CreateObjectAction (from IntermediateActions)

CreateObjectAction is an action that creates an object that conforms to a statically specified classifier and puts it on an output pin at runtime.

## Generalizations

- “Action (from BasicActions)” on page 230

## Description

This action instantiates a classifier.

## Attributes

No additional attributes

## Associations

- classifier : Classifier [1..1] Classifier to be instantiated.
- result : OutputPin [1..1] (Specialized from Action:output) Gives the output pin on which the result is put.

## Constraints

- [1] The classifier cannot be abstract.  
`not (self.classifier.isAbstract = #true)`
- [2] The classifier cannot be an association class.  
`not self.classifier.ocllsKindOf(AssociationClass)`
- [3] The type of the result pin must be the same as the classifier of the action.  
`self.result.type = self.classifier`
- [4] The multiplicity of the output pin is 1..1.  
`self.result.multiplicity.is(1,1)`

## Semantics

The new object is created, and the classifier of the object is set to the given classifier. The new object is returned as the value of the action. The action has no other effect. In particular, no behaviors are executed, no initial expressions are evaluated, and no state machine transitions are triggered. The new object has no structural feature values and participates in no links.

## Notation

No specific notation

## Rationale

CreateObjectAction is introduced for creating new objects.

## Changes from previous UML

Same as UML 1.5

### 11.3.17 DestroyLinkAction (from IntermediateActions)

DestroyLinkAction is a write link action that destroys links and link objects.

## Generalizations

- “WriteLinkAction (from IntermediateActions)” on page 281.

## Description

This action destroys a link or a link object. Link objects can also be destroyed with `DestroyObjectAction`. The link is specified in the same way as link creation, even for link objects. This allows actions to remain unchanged when their associations are transformed from ordinary ones to association classes and vice versa.

`DestroyLinkAction` uses a specialization of `LinkEndData`, called `LinkEndDestructionData`, to support ordered non-unique associations. The position of the link to be destroyed is specified at runtime by an additional input pin, which is required for ordered non-unique association ends and omitted for other kinds of ends. This is a positive integer giving the position of the link to destroy.

`DestroyLinkAction` also uses `LinkEndDestructionData` to support the destruction of duplicate links of the association on ends that are non-unique. This option is available on an end-by-end basis, and causes all duplicate links of the association emanating from the specified ends to be destroyed.

## Attributes

No additional attributes

## Associations

- `endData : LinkEndDestructionData [2..*]` {Redefined from *LinkAction::endData*}  
Specifies ends of association and inputs.

## Constraints

No additional constraints

## Semantics

Destroying a link that does not exist has no effect. The semantics of `DestroyObjectAction` applies to destroying a link that has a link object with `DestroyLinkAction`.

The semantics is undefined for destroying a link that has an association end with `isReadOnly = true` after initialization of the other end objects, unless the link being destroyed does not exist. Objects participating in the association across from a writeable end can have links destroyed as long as the objects across from all read only ends are still being initialized. This means objects participating in two or more `readOnly` ends cannot have links destroyed, unless all the linked objects are being initialized.

Destroying links for non-unique ordered association ends requires identifying the position of the link using the input pin of `LinkEndDestructionData`. The pin is of type `UnlimitedNatural` with multiplicity 1..1. A pin value that is a positive integer less than or equal to the current number of links means to destroy the link at that position in the sequence of existing links, with the integer one meaning the first link in the sequence. The semantics is undefined for value of zero, for an integer greater than the number of existing links, and for unlimited. The `destroyAt` input pin only exists for ordered non-unique association ends.

## Notation

No specific notation

## Rationale

`DestroyLinkAction` is introduced for destroying links.



## Changes from previous UML

DestroyLinkAction is unchanged from UML 1.5.

### 11.3.18 DestroyObjectAction (from IntermediateActions)

DestroyObjectAction is an action that destroys objects.

#### Generalizations

- “Action (from BasicActions)” on page 230

#### Description

This action destroys the object on its input pin at runtime. The object may be a link object, in which case the semantics of DestroyLinkAction also applies.

#### Attributes

- isDestroyLinks : Boolean = false      Specifies whether links in which the object participates are destroyed along with the object. Default value is *false*.
- isDestroyOwnedObjects : Boolean = false      Specifies whether objects owned by the object are destroyed along with the object. Default value is *false*.

#### Associations

- target : InputPin [1..1] (Specialized from Action:input)      The input pin providing the object to be destroyed.

#### Constraints

- [1] The multiplicity of the input pin is 1..1.  
self.target.multiplicity.is(1,1)
- [2] The input pin has no type.  
self.target.type->size() = 0

#### Semantics

The classifiers of the object are removed as its classifiers, and the object is destroyed. The default action has no other effect. In particular, no behaviors are executed, no state machine transitions are triggered, and references to the destroyed objects are unchanged. If isDestroyLinks is *true*, links in which the object participates are destroyed along with the object according to the semantics of DestroyLinkAction, except for link objects, which are destroyed according to the semantics of DestroyObjectAction with the same attribute values as the original DestroyObjectAction. If isDestroyOwnedObjects is *true*, objects owned by the object are destroyed according to the semantics of DestroyObjectAction with the same attribute values as the original DestroyObjectAction.

Destroying an object that is already destroyed has no effect.

#### Notation

No specific notation

## **Rationale**

DestroyObjectAction is introduced for destroying objects.

## **Changes from previous UML**

Same as UML 1.5

### **11.3.19 InputPin (from BasicActions)**

#### **Generalizations**

- “Pin (from BasicActions)” on page 256

#### **Description**

An input pin is a pin that holds input values to be consumed by an action.

#### **Attributes**

No additional attributes

#### **Associations**

No additional associations

#### **Constraints**

No additional constraints

#### **Semantics**

An action cannot start execution if an input pin has fewer values than the lower multiplicity. The upper multiplicity determines how many values are consumed by a single execution of the action.

#### **Notation**

No specific notation. See extensions in Activities.

## **Rationale**

## **Changes from previous UML**

InputPin is the same concept as in UML 1.5, but modeled independently of the behaviors that use it.

### **11.3.20 InvocationAction (from BasicActions)**

#### **Generalizations**

- “Action (from BasicActions)” on page 230

#### **Description**

Invocation is an abstract class for the various actions that invoke behavior.

## Attributes

No additional attributes

## Associations

- argument : InputPin [0..\*]                      Specification of an argument value that appears during execution.

## Constraints

No additional constraints

## Semantics

See children of InvocationAction.

### 11.3.21 LinkAction (from IntermediateActions)

LinkAction is an abstract class for all link actions that identify their links by the objects at the ends of the links and by the qualifiers at ends of the links.

## Generalizations

- “Action (from BasicActions)” on page 230

## Description

A link action creates, destroys, or reads links, identifying a link by its end objects and qualifier values, if any.

## Attributes

No additional attributes

## Associations

- endData : LinkEndData [2..\*]                      Data identifying one end of a link by the objects on its ends and qualifiers.
- input : InputPin [1..\*] (Specialized from Action:input)Pins taking end objects and qualifier values as input.

## Constraints

- [1] The association ends of the link end data must all be from the same association and include all and only the association ends of that association.  
self.endData->collect(end) = self.association()->collect(connection))
- [2] The association ends of the link end data must not be static.  
self.endData->forall(end.oclisKindOf(NavigableEnd) implies end.isStatic = #false)
- [3] The input pins of the action are the same as the pins of the link end data and insertion pins.  
self.input->asSet() =  
    let ledpins : Set = self.endData->collect(value) in  
        if self.oclisKindOf(LinkEndCreationData)  
            then ledpins->union(self.endData.oclAsType(LinkEndCreationData).insertAt)  
        else ledpins

Additional operations:

[1] association operates on LinkAction. It returns the association of the action.

```
association();  
association = self.endData->asSequence().first().end.association
```

### Constraints

[1] The input pins of the action are the same as the pins of the link end data, qualifier values, and insertion pins.

```
self.input->asSet() =  
  let ledpins : Set =  
    if self.endData.ocllsKindOf(CompleteActions::LinkEndData)  
    then self.endData->collect(value)->union(self.endData.qualifier.value)  
    else self.endData->collect(value) in  
    if self.ocllsKindOf(LinkEndCreationData)  
    then ledpins->union(self.endData.oclAsType(LinkEndCreationData).insertAt)  
    else ledpins
```

### Semantics

For actions that write links, all association ends must have a corresponding input pin so that all end objects are specified when creating or deleting a link. An input pin identifies the end object by being given a value at runtime. It has the type of the association end and multiplicity of 1..1, since a link always has exactly one object at its ends.

The behavior is undefined for links of associations that are static on any end.

For the semantics of link actions see the children of LinkAction.

### Notation

No specific notation

### Rationale

LinkAction is introduced to abstract aspects of link actions that identify links by the objects on their ends.

In CompleteActions, LinkAction is extended for qualifiers.

### Changes from previous UML

LinkAction is unchanged from UML 1.5.

### 11.3.22 LinkEndCreationData (from IntermediateActions, CompleteActions)

LinkEndCreationData is not an action. It is an element that identifies links. It identifies one end of a link to be created by CreateLinkAction.

### Generalizations

- “LinkEndData (from IntermediateActions, CompleteActions)” on page 253.

## Description

This class is required when using `CreateLinkAction` to specify insertion points for ordered ends and for replacing all links at end. A link cannot be passed as a runtime value to or from an action. Instead, a link is identified by its end objects and qualifier values, as required. This requires more than one piece of data, namely, the statically-specified end in the user model, the object on the end, and the qualifier values for that end. These pieces are brought together around `LinkEndData`. Each association end is identified separately with an instance of the `LinkEndData` class.

Qualifier values are used in `CompleteActions`.

## Attributes

- `isReplaceAll : Boolean [1..1] = false` Specifies whether the existing links emanating from the object on this end should be destroyed before creating a new link.

## Associations

- `insertAt : InputPin [0..1]`  
Specifies where the new link should be inserted for ordered association ends, or where an existing link should be moved to. The type of the input is `UnlimitedNatural`, but the input cannot be zero. This pin is omitted for association ends that are not ordered.

## Constraints

- [1] `LinkEndCreationData` can only be end data for `CreateLinkAction` or one of its specializations.  
`self.LinkAction.ocllsKindOf(CreateLinkAction)`
- [2] Link end creation data for ordered association ends must have a single input pin for the insertion point with type `UnlimitedNatural` and multiplicity of `1..1`, otherwise the action has no input pin for the insertion point.

```
let insertAtPins : Collection = self.insertAt in
  if self.end.ordering = #unordered
    then insertAtPins->size() = 0
  else let insertAtPin : InputPin = insertAts->asSequence()->first() in
    insertAtPins->size() = 1
    and insertAtPin.type = UnlimitedNatural
    and insertAtPin.multiplicity.is(1,1)
  endif
```

## Semantics

See `CreateLinkAction`, also see `LinkAction` and all its children.

## Notation

No specific notation

## Rationale

`LinkEndCreationData` is introduced to indicate which inputs are for which link end objects and qualifiers.

## Changes from previous UML

LinkEndCreationData is unchanged from UML 1.5.

### 11.3.23 LinkEndData (from IntermediateActions, CompleteActions)

#### Generalizations

- “Element (from Kernel)” on page 60

#### Description

*Package IntermediateActions*

LinkEndData is not an action. It is an element that identifies links. It identifies one end of a link to be read or written by the children of LinkAction. A link cannot be passed as a runtime value to or from an action. Instead, a link is identified by its end objects and qualifier values, if any. This requires more than one piece of data, namely, the statically-specified end in the user model, the object on the end, and the qualifier values for that end, if any. These pieces are brought together around LinkEndData. Each association end is identified separately with an instance of the LinkEndData class.

#### Attributes

No additional attributes

#### Associations

- end : Property [1..1]      Association end for which this link-end data specifies values.
- value : InputPin [0..1]      Input pin that provides the specified object for the given end. This pin is omitted if the link-end data specifies an “open” end for reading.

#### Associations

*Package CompleteActions*

- qualifier : QualifierValue [\*]      List of qualifier values.

#### Constraints

- [1] The property must be an association end.  
self.end.association->size() = 1
- [2] The type of the end object input pin is the same as the type of the association end.  
self.value.type = self.end.type
- [3] The multiplicity of the end object input pin must be “1..1.”  
self.value.multiplicity.is(1,1)

#### Constraints

*Package CompleteActions*

- [1] The qualifiers include all and only the qualifiers of the association end.  
self.qualifier->collect(qualifier) = self.end.qualifier

[2] The end object input pin is not also a qualifier value input pin.  
self.value->excludesAll(self.qualifier.value)

## Semantics

See LinkAction and its children.

## Notation

No specific notation

## Rationale

LinkEndData is introduced to indicate which inputs are for which link end objects and qualifiers.

## Changes from previous UML

LinkEndData is unchanged from UML 1.5.

### 11.3.24 LinkEndDestructionData (from IntermediateActions)

LinkEndDestructionData is not an action. It is an element that identifies links. It identifies one end of a link to be destroyed by DestroyLinkAction.

## Generalizations

- “LinkEndData (from IntermediateActions, CompleteActions)” on page 253.

## Description

This class is required when using DestroyLinkAction, to specify links to destroy for non-unique ordered ends. A link cannot be passed as a runtime value to or from an action. See description of “LinkEndData (from IntermediateActions, CompleteActions)” on page 253.

Qualifier values are used in CompleteActions.

## Attributes

- isDestroyDuplicates : Boolean = false      Specifies whether to destroy duplicates of the value in non-unique association ends.

## Associations

- destroyAt : InputPin [0..1]      Specifies the position of an existing link to be destroyed in ordered non-unique association ends. The type of the pin is UnlimitedNatural, but the value cannot be zero or unlimited.

## Constraints

- [1] LinkEndDestructionData can only be end data for DestroyLinkAction or one of its specializations.
- [2] LinkEndDestructionData for ordered non-unique association ends must have a single destroyAt input pin if isDestroyDuplicates is false. It must be of type UnlimitedNatural and have a multiplicity of 1..1. Otherwise, the action has no input pin for the removal position.

## Semantics

See “DestroyLinkAction (from IntermediateActions)” on page 246, also see “LinkAction (from IntermediateActions)” on page 250 and all of its subclasses.

## Notation

No specific notation

## Rationale

LinkeEndDestructionData is introduced to indicate which links to destroy for ordered non-unique ends.

### 11.3.25 MultiplicityElement (from BasicActions)

#### Generalizations

- “MultiplicityElement (from Kernel)” on page 90 (*merge increment*)

#### Operations

- [1] The operation `compatibleWith` takes another multiplicity as input. It checks if one multiplicity is compatible with another.
- ```
compatibleWith(other : Multiplicity) : Boolean;  
compatibleWith(other) = Integer.allInstances()->  
    forAll(i : Integer | self.includesCardinality(i) implies other.includesCardinality(i))
```
- [2] The operation `is` determines if the upper and lower bound of the ranges are the ones given.
- ```
is(lowerbound : integer, upperbound : integer) : Boolean  
is(lowerbound, upperbound) = (lowerbound = self.lowerbound and upperbound = self.upperbound)
```

### 11.3.26 OpaqueAction (from BasicActions)

#### Generalizations

- “Pin (from BasicActions)” on page 256

#### Description

An action with implementation-specific semantics.

#### Attributes

- `body : String [1..*]` Specifies the action in one or more languages.
- `language : String [*]` Languages the body strings use, in the same order as the body strings.

#### Associations

No additional associations

#### Constraints

No additional constraints



## **Semantics**

The semantics of the action are determined by the implementation.

## **Notation**

No specific notation

## **Rationale**

OpaqueAction is introduced for implementation-specific actions or for use as a temporary placeholder before some other action is chosen.

### **11.3.27 OutputPin (from BasicActions)**

#### **Generalizations**

- “Pin (from BasicActions)” on page 256

#### **Description**

An output pin is a pin that holds output values produced by an action.

#### **Attributes**

No additional attributes

#### **Associations**

No additional associations

#### **Constraints**

No additional constraints

## **Semantics**

An action cannot terminate itself if an output pin has fewer values than the lower multiplicity. An action may not put more values in an output pin in a single execution than the upper multiplicity of the pin.

## **Notation**

No specific notation. See extensions in Activities.

## **Changes from previous UML**

OutputPin is the same concept as in UML 1.5, but modeled independently of the behaviors that use it.

### **11.3.28 Pin (from BasicActions)**

#### **Generalizations**

- “MultiplicityElement (from BasicActions)” on page 255

- “TypedElement (from Kernel)” on page 131

### **Description**

A pin is a typed element and multiplicity element that provides provide values to actions and accept result values from them.

### **Attributes**

No additional attributes

### **Associations**

No additional associations

### **Constraints**

- [1] If the action is an invocation action, the number and types of pins must be the same as the number of parameters and types of the invoked behavior or behavioral feature. Pins are matched to parameters by order.

### **Semantics**

A pin represents an input to an action or an output from an action. The definition on an action assumes that pins are ordered.

Pin multiplicity controls action execution, not the number of tokens in the pin (see upperBound on “ObjectNode (from BasicActivities, CompleteActivities)” on page 380). See “InputPin (from BasicActions)” and “OutputPin (from BasicActions)” for semantics of multiplicity. Pin multiplicity is not unique, because multiple tokens with the same value can reside in an object node.

### **Notation**

No specific notation. See extensions in Activities.

### **Rationale**

Pins are introduced to model inputs and outputs of actions.

### **Changes from previous UML**

Pin is the same concept as in UML 1.5, but modeled independently of the behaviors that use it.

## **11.3.29 QualifierValue (from CompleteActions)**

QualifierValue is not an action. It is an element that identifies links. It gives a single qualifier within a link end data specification. See LinkEndData.

### **Generalizations**

- “Element (from Kernel)” on page 60

## Description

A link cannot be passed as a runtime value to or from an action. Instead, a link is identified by its end objects and qualifier values, as required. This requires more than one piece of data, namely, the end in the user model, the object on the end, and the qualifier values for that end. These pieces are brought together around `LinkEndData`. Each association end is identified separately with an instance of the `LinkEndData` class.

## Attributes

No additional attributes

## Associations

- `qualifier : Property [1..1]`      Attribute representing the qualifier for which the value is to be specified.
- `value : InputPin [1..1]`      Input pin from which the specified value for the qualifier is taken.

## Constraints

- [1] The qualifier attribute must be a qualifier of the association end of the link-end data.  
`self.LinkEndData.end->collect(qualifier)->includes(self.qualifier)`
- [2] The type of the qualifier value input pin is the same as the type of the qualifier attribute.  
`self.value.type = self.qualifier.type`
- [3] The multiplicity of the qualifier value input pin is “1..1.”  
`self.value.multiplicity.is(1,1)`

## Semantics

See `LinkAction` and its children.

## Notation

No specific notation

## Rationale

`QualifierValue` is introduced to indicate which inputs are for which link end qualifiers.

## Changes from previous UML

`QualifierValue` is unchanged from UML 1.5.

## 11.3.30 RaiseExceptionAction (from StructuredActions)

### Generalizations

- “Action (from BasicActions)” on page 230

## Description

`RaiseExceptionAction` is an action that causes an exception to occur. The input value becomes the exception object.

## Attributes

No additional attributes

## Associations

- exception : InputPin [1..1]      An input pin whose value becomes an exception object.

## Semantics

When a raise exception action is executed, the value on the input pin is raised as an exception. The value may be copied in this process, so identity may not be preserved. Raising the exception terminates the immediately containing structured node or activity and begins a search of enclosing nested scopes for an exception handler that matches the type of the exception object. See “ExceptionHandler (from ExtraStructuredActivities)” on page 351 for details of handling exceptions.

## Notation

No specific notation

## Rationale

Raise exception action allows models to generate exceptions. Otherwise the only exception types would be predefined built-in exception types, which would be too restrictive.

## Changes from previous UML

RaiseExceptionAction replaces JumpAction from UML 1.5. Their behavior is essentially the same, except that it is no longer needed for performing simple control constructs such as break and continue.

### 11.3.31 ReadExtentAction (from CompleteActions)

#### Generalizations

- “Action (from BasicActions)” on page 230

#### Description

ReadExtentAction is an action that retrieves the current instances of a classifier.

#### Attributes

No additional attributes

#### Associations

- classifier : Classifier [1..1]      The classifier whose instances are to be retrieved.
- result : OutputPin [1..1]      The runtime instances of the classifier.

#### Constraints

- [1] The type of the result output pin is the classifier.
- [2] The multiplicity of the result output pin is “0..\*.”

```
self.result.multiplicity.is(0,#null)
```

## Semantics

The extent of a classifier is the set of all instances of a classifier that exist at any one time.

## Semantic Variation Points

It is not generally practical to require that reading the extent produce all the instances of the classifier that exist in the entire universe. Rather, an execution engine typically manages only a limited subset of the total set of instances of any classifier and may manage multiple distributed extents for any one classifier. It is not formally specified which managed extent is actually read by a ReadExtentAction.

## Notation

No specific notation

## Rationale

ReadExtentAction is introduced to provide access to the runtime instances of a classifier.

## Changes from previous UML

ReadExtentAction is unchanged from UML 1.5.

### 11.3.32 ReadIsClassifiedObjectAction (from CompleteActions)

ReadIsClassifiedObjectAction is an action that determines whether a runtime object is classified by a given classifier.

## Generalizations

- “Action (from BasicActions)” on page 230

## Description

This action tests the classification of an object against a given class. It can be restricted to testing direct instances.

## Attributes

- `isDirect` : Boolean [1..1]      Indicates whether the classifier must directly classify the input object. The default value is *false*.

## Associations

- `classifier` : Classifier [1..1]      The classifier against which the classification of the input object is tested.
- `object` : InputPin [1..1]      Holds the object whose classification is to be tested. (Specializes *Action.input*.)
- `result` : OutputPin [1..1]      After termination of the action, will hold the result of the test. (Specializes *Action.output*.)

## Constraints

[1] The multiplicity of the input pin is 1..1.

```
self.object.multiplicity.is(1,1)
```

[2] The input pin has no type.

```
self.object.type->isEmpty()
```

[3] The multiplicity of the output pin is 1..1.

```
self.result.multiplicity.is(1,1)
```

[4] The type of the output pin is Boolean.

```
self.result.type = Boolean
```

## Semantics

The action returns *true* if the input object is classified by the specified classifier. It returns *true* if the *isDirect* attribute is *false* and the input object is classified by the specified classifier, or by one of its (direct or indirect) descendents. Otherwise, the action returns *false*.

## Notation

No specific notation

## Rationale

ReadisClassifiedObjectAction is introduced for run-time type identification.

## Changes from previous UML

ReadisClassifiedObjectAction is unchanged from UML 1.5.

### 11.3.33 ReadLinkAction (from IntermediateActions)

ReadLinkAction is a link action that navigates across associations to retrieve objects on one end.

## Generalizations

- “LinkAction (from IntermediateActions)” on page 250

## Description

This action navigates an association towards one end, which is the end that does not have an input pin to take its object (the “open” end). The objects put on the result output pin are the ones participating in the association at the open end, conforming to the specified qualifiers, in order if the end is ordered. The semantics is undefined for reading a link that violates the navigability or visibility of the open end.

## Attributes

No additional attributes

## Associations

- result : OutputPin [1] (Specialized from Action:output) The pin on which are put the objects participating in the association at the end not specified by the inputs.

## Constraints

- [1] Exactly one link-end data specification (the “open” end) must not have an end object input pin.
- ```
self.endData->select(ed | ed.value->size() = 0)->size() = 1
```

[2] The type and ordering of the result output pin are the same as the type and ordering of the open association end.

```
let openend : AssociationEnd = self.endData->select(ed | ed.value->size() = 0)->asSequence()->first().end in
  self.result.type = openend.type
  and self.result.ordering = openend.ordering
```

[3] The multiplicity of the open association end must be compatible with the multiplicity of the result output pin.

```
let openend : AssociationEnd = self.endData->select(ed | ed.value->size() = 0)->asSequence()->first().end in
  openend.multiplicity.compatibleWith(self.result.multiplicity)
```

[4] The open end must be navigable.

```
let openend : AssociationEnd = self.endData->select(ed | ed.value->size() = 0)->asSequence()->first().end in
  openend.isNavigable()
```

[5] Visibility of the open end must allow access to the object performing the action.

```
let host : Classifier = self.context in
let openend : AssociationEnd = self.endData->select(ed | ed.value->size() = 0)->asSequence()->first().end in
  openend.visibility = #public
  or self.endData->exists(oed | not oed.end = openend
    and (host = oed.end.participant
      or (openend.visibility = #protected
        and host.allSupertypes->includes(oed.end.participant))))
```

## Semantics

Navigation of a binary association requires the specification of the source end of the link. The target end of the link is not specified. When qualifiers are present, one navigates to a specific end by giving objects for the source end of the association and qualifier values for all the ends. These inputs identify a subset of all the existing links of the association that match the end objects and qualifier values. The result is the collection of objects for the end being navigated towards, one object from each identified link.

In a ReadLinkAction, generalized for n-ary associations, one of the link-end data must have an unspecified object (the “open” end). The result of the action is a collection of objects on the open end of links of the association, such that the links have the given objects and qualifier values for the other ends and the given qualifier values for the open end. This result is placed on the output pin of the action, which has a type and ordering given by the open end. The multiplicity of the open end must be compatible with the multiplicity of the output pin. For example, the modeler can set the multiplicity of this pin to support multiple values even when the open end only allows a single value. This way the action model will be unaffected by changes in the multiplicity of the open end. The semantics are defined only when the open end is navigable, and visible to the host object of the action.

## Notation

No specific notation

## Rationale

ReadLinkAction is introduced to navigate across links.

## Changes from previous UML

ReadLinkAction is unchanged from UML 1.5.

### 11.3.34 ReadLinkObjectEndAction (from CompleteActions)

ReadLinkObjectEndAction is an action that retrieves an end object from a link object.

#### Generalizations

- “Action (from BasicActions)” on page 230

#### Description

This action reads the object on an end of a link object. The association end to retrieve the object from is specified statically, and the link object to read is provided on the input pin at run time.

#### Attributes

No additional attributes

#### Associations

- |                                                            |                                                             |
|------------------------------------------------------------|-------------------------------------------------------------|
| • end : Property [1..1]                                    | Link end to be read.                                        |
| • object : InputPin [1..1] (Specialized from Action:input) | Gives the input pin from which the link object is obtained. |
| • result : OutputPin [1..1]                                | Pin where the result value is placed.                       |

#### Constraints

- [1] The property must be an association end.  
self.end.association.notEmpty()
- [2] The association of the association end must be an association class.  
self.end.Association.ocllsKindOf(AssociationClass)
- [3] The ends of the association must not be static.  
self.end.association.memberEnd->forall(e | **not** e.isStatic)
- [4] The type of the object input pin is the association class that owns the association end.  
self.object.type = self.end.association
- [5] The multiplicity of the object input pin is “1..1.”  
self.object.multiplicity.is(1,1)
- [6] The type of the result output pin is the same as the type of the association end.  
self.result.type = self.end.type
- [7] The multiplicity of the result output pin is 1..1.  
self.result.multiplicity.is(1,1)

#### Semantics

ReadLinkObjectEndAction retrieves an end object from a link object.

#### Notation

No specific notation



## Rationale

ReadLinkObjectEndAction is introduced to navigate from a link object to its end objects.

## Changes from previous UML

ReadLinkObjectEndAction is unchanged from UML 1.5.

### 11.3.35 ReadLinkObjectEndQualifierAction (from CompleteActions)

ReadLinkObjectEndAction is an action that retrieves a qualifier end value from a link object.

## Generalizations

- “Action (from BasicActions)” on page 230

## Description

This action reads a qualifier value or values on an end of a link object. The association end to retrieve the qualifier from is specified statically, and the link object to read is provided on the input pin at run time.

## Attributes

No additional attributes

## Associations

- qualifier : Property [1..1] The attribute representing the qualifier to be read.
- object : InputPin [1..1] (Specialized from Action:input) Gives the input pin from which the link object is obtained.
- result : OutputPin [1..1] Pin where the result value is placed.

## Constraints

- [1] The qualifier attribute must be a qualifier attribute of an association end.  
`self.qualifier.associationEnd->size() = 1`
- [2] The association of the association end of the qualifier attribute must be an association class.  
`self.qualifier.associationEnd.association.ocllsKindOf(AssociationClass)`
- [3] The ends of the association must not be static.  
`self.qualifier.associationEnd.association.memberEnd->forall(e | not e.isStatic)`
- [4] The type of the object input pin is the association class that owns the association end that has the given qualifier attribute.  
`self.object.type = self.qualifier.associationEnd.association`
- [5] The multiplicity of the qualifier attribute is 1..1.  
`self.qualifier.multiplicity.is(1,1)`
- [6] The multiplicity of the object input pin is “1..1.”  
`self.object.multiplicity.is(1,1)`
- [7] The type of the result output pin is the same as the type of the qualifier attribute.  
`self.result.type = self.qualifier.type`
- [8] The multiplicity of the result output pin is “1..1.”

```
self.result.multiplicity.is(1,1)
```

## Semantics

ReadLinkObjectEndAction retrieves a qualifier end value from a link object.

## Notation

No specific notation

## Rationale

ReadLinkObjectEndQualifierAction is introduced to navigate from a link object to its end objects.

## Changes from previous UML

ReadLinkObjectEndQualifierAction is unchanged from UML 1.5, except the name was corrected from ReadLinkObjectQualifierAction.

### 11.3.36 ReadSelfAction (from IntermediateActions)

ReadSelfAction is an action that retrieves the host object of an action.

## Generalizations

- “Action (from BasicActions)” on page 230

## Description

Every action is ultimately a part of some behavior, which is in turn optionally attached in some way to the specification of a classifier (for example, as the body of a method or as part of a state machine). When the behavior executes, it does so in the context of some specific host instance of that classifier. This action produces this host instance, if any, on its output pin. The type of the output pin is the classifier to which the behavior is associated in the user model.

## Attributes

No additional attributes

## Associations

- result : OutputPin [1..1] (Specialized from Action:output) Gives the output pin on which the hosting object is placed.

## Constraints

- [1] The action must be contained in a behavior that has a host classifier.  
self.context->size() = 1
- [2] If the action is contained in a behavior that is acting as the body of a method, then the operation of the method must not be static.
- [3] The type of the result output pin is the host classifier.  
self.result.type = self.context
- [4] The multiplicity of the result output pin is “1..1.”

```
self.result.multiplicity.is(1,1)
```

## Semantics

Every action is part of some behavior, as are behaviors invoked by actions or other elements of behaviors. Behaviors are optionally attached in some way to the specification of a classifier.

For behaviors that have no other context object, the behavior itself is the context object. See behaviors as classes in Common Behaviors and discussion of reflective objects in Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities).

## Notation

No specific notation

## Rationale

ReadSelfAction is introduced to provide access to the context object when it is not available as a parameter.

## Changes from previous UML

ReadSelfAction is unchanged from UML 1.5.

### 11.3.37 ReadStructuralFeatureAction (from IntermediateActions)

ReadStructuralFeatureAction is a structural feature action that retrieves the values of a structural feature.

## Generalizations

- “StructuralFeatureAction (from IntermediateActions)” on page 275.

## Description

This action reads the values of a structural feature in order if the structural feature is ordered.

## Attributes

No additional attributes

## Associations

- result : OutputPin [1..1] (Specialized from Action:output) Gives the output pin on which the result is put.

## Constraints

- [1] The type and ordering of the result output pin are the same as the type and ordering of the structural feature.  
self.result.type = self.structuralFeature.type  
and self.result.ordering = self.structuralFeature.ordering
- [2] The multiplicity of the structural feature must be compatible with the multiplicity of the output pin.  
self.structuralFeature.multiplicity.compatibleWith(self.result.multiplicity)

## Semantics

The values of the structural feature of the input object are placed on the output pin of the action. If the feature is an association end, the semantics are the same as reading links of the association with the feature as the open end. The type and ordering of the output pin are the same as the specified structural feature. The multiplicity of the structural feature must be compatible with the multiplicity of the output pin. For example, the modeler can set the multiplicity of this pin to support multiple values even when the structural feature only allows a single value. This way the action model will be unaffected by changes in the multiplicity of the structural feature.

## Notation

No specific notation

## Rationale

ReadStructuralFeatureAction is introduced to retrieve the values of a structural feature.

## Changes from previous UML

ReadStructuralFeatureAction is new in UML 2.0. It generalizes ReadAttributeAction from UML 1.5.

### 11.3.38 ReadVariableAction (from StructuredActions)

ReadVariableAction is a variable action that retrieves the values of a variable.

## Generalizations

- “VariableAction (from StructuredActions)” on page 281.

## Description

This action reads the values of a variable in order if the variable is ordered.

## Attributes

No additional attributes

## Associations

- result : OutputPin [1..1] (Specialized from Action:output) Gives the output pin on which the result is put.

## Constraints

- [1] The type and ordering of the result output pin of a read-variable action are the same as the type and ordering of the variable.  
self.result.type = self.variable.type  
and self.result.ordering = self.variable.ordering
- [2] The multiplicity of the variable must be compatible with the multiplicity of the output pin.  
self.variable.multiplicity.compatibleWith(self.result.multiplicity)

## Semantics

The values of the variable are placed on the output pin of the action. The type and ordering of the output pin are the same as the specified variable. The multiplicity of the variable must be compatible with the multiplicity of the output pin. For example, the modeler can set the multiplicity of this pin to support multiple values even when the variable only allows a single value. This way the action model will be unaffected by changes in the multiplicity of the variable.

## Notation

No specific notation

## Rationale

ReadVariableAction is introduced to retrieve the values of a variable.

## Changes from previous UML

ReadVariableAction is unchanged from UML 1.5.

### 11.3.39 ReclassifyObjectAction (from CompleteActions)

ReclassifyObjectAction is an action that changes which classifiers classify an object.

## Generalizations

- “Action (from BasicActions)” on page 230

## Description

ReclassifyObjectAction adds given classifier to an object and removes given classifiers from that object. Multiple classifiers may be added and removed at a time.

## Attributes

- `isReplaceAll` : Boolean [1..1]      Specifies whether existing classifiers should be removed before adding the new classifiers. The default value is *false*.

## Associations

- `object` : InputPin [1..1]      Holds the object to be reclassified. (Specializes *Action.input*.)
- `newClassifier` : Classifier [0..\*]      A set of classifiers to be added to the classifiers of the object.
- `oldClassifier` : Classifier [0..\*]      A set of classifiers to be removed from the classifiers of the object.

## Constraints

- [1] None of the new classifiers may be abstract.  
`not self.newClassifier->exists(isAbstract = true)`
- [2] The multiplicity of the input pin is 1..1.  
`self.argument.multiplicity.is(1,1)`
- [3] The input pin has no type.  
`self.argument.type->size() = 0`

## Semantics

After the action completes, the input object is classified by its existing classifiers and the “new” classifiers given to the action; however, the “old” classifiers given to the actions no longer classify the input object. The identity of the object is preserved, no behaviors are executed, and no initial expressions are evaluated. “New” classifiers replace existing classifiers in an atomic step, so that structural feature values and links are not lost during the reclassification, when the “old” and “new” classifiers have structural features and associations in common.

Neither adding a classifier that duplicates an already existing classifier, nor removing a classifier that is not classifying the input object, has any effect. Adding and removing the same classifiers has no effect.

If *isReplaceAll* is *true*, then the existing classifiers are removed before the “new” classifiers are added, except if the “new” classifier already classifies the input object, in which case this classifier is not removed. If *isReplaceAll* is *false*, then adding an existing value has no effect.

It is an error, if any of the “new” classifiers is abstract or if all classifiers are removed from the input object.

## Notation

No specific notation

## Rationale

ReclassifyObjectAction is introduced to change the classifiers of an object.

## Changes from previous UML

ReclassifyObjectAction is unchanged from UML 1.5.

### 11.3.40 RemoveStructuralFeatureValueAction (from IntermediateActions)

RemoveStructuralFeatureValueAction is a write structural feature action that removes values from structural features.

## Generalizations

- “WriteStructuralFeatureAction (from IntermediateActions)” on page 282.

## Description

The object to access is specified dynamically, by referring to an input pin on which the object will be placed at runtime. The type of the value of this pin is the classifier that owns the specified structural feature, and the value’s multiplicity is 1..1.

Structural features are potentially multi-valued and ordered, and may support duplicates, so the action supports specification of removal points for new values. It also supports the removal of all duplicate values.

## Attributes

- `isRemoveDuplicates` : Boolean = false [1..1]      Specifies whether to remove duplicates of the value in non-unique structural features.

## Associations

- `removeAt` : InputPin [0..1] Specifies the position of an existing value to remove in ordered non-unique structural features. The type of the pin is Unlimitednatural, but the value cannot be zero or unlimited.

## Constraints

- [1] Actions removing a value from ordered non-unique structural features must have a single `removeAt` input pin if `isRemoveDuplicates` is false. It must be of type Unlimited Natural with multiplicity 1..1. Otherwise, the action has no `removeAt` input pin.

## Semantics

Structural features are potentially multi-valued. Removing a value succeeds even when it violates the minimum multiplicity. Removing a value that does not exist has no effect. If the feature is an association end, the semantics are the same as for destroying links, the participants of which are the object owning the structural feature and the value being removed.

Values of a structural feature may be duplicate in non-unique structural features. The `isRemoveDuplicates` attribute indicates whether to remove all duplicates of the specified value. The `removeAt` input pin is required if `isRemoveDuplicates` is false in ordered non-unique structural features. It indicates the position of an existing value to remove. It must be a positive integer less than or equal to the current number of values. The semantics is undefined for zero or an integer greater than the number of existing values, and for unlimited.

The semantics is undefined for removing an existing value for a structural feature with `isReadOnly=true`. The semantics is undefined for removing an existing value of a structural feature with `settability readOnly` after initialization of the owning object.

## Notation

No specific notation

## Rationale

`RemoveStructuralFeatureValueAction` is introduced to remove structural feature values.

## Changes from previous UML

`RemoveStructuralFeatureValueAction` is new in UML 2.0. It generalizes `RemoveAttributeValueAction` in UML 2.0.

### 11.3.41 RemoveVariableValueAction (from StructuredActions)

`RemoveVariableValueAction` is a write variable action that removes values from variables.

## Generalizations

- “`WriteVariableAction` (from StructuredActions)” on page 283

## Description

One value is removed from the set of possible variable values.

## Attributes

- `isRemoveDuplicates` : Boolean = false [1..1]      Specifies whether to remove duplicates of the value in non-unique variables.

## Associations

- `removeAt` : InputPin [0..1]      Specifies the position of an existing value to remove in ordered non-unique variables. The type of the pin is `UnlimitedNatural`, but the value cannot be zero or unlimited.

## Constraints

- [1] Actions removing a value from ordered non-unique variables must have a single `removeAt` input pin if `isRemoveDuplicates` is false. It must be of type `UnlimitedNatural` with multiplicity of 1..1, otherwise the action has no `removeAt` input pin.

## Semantics

Variables are potentially multi-valued. Removing a value succeeds even when it violates the minimum multiplicity. Removing a value that does not exist has no effect. Variables are potentially multi-valued and ordered, and may support duplicates, so the action supports specification of removal points for new values. It also supports the removal of all duplicate values.

Values of a variable may be duplicate in non-unique variables. The `isRemoveDuplicates` attribute indicates whether to remove all duplicates of the specified value. The `removeAt` input pin is required if `isRemoveDuplicates` is false in ordered non-unique variables. It indicates the position of an existing value to remove. It must be a positive integer less than or equal to the current number of values. The semantics is undefined for zero, for an integer greater than the number of existing values and for unlimited.

## Notation

No specific notation

## Rationale

`RemoveVariableValueAction` is introduced to remove variable values.

## Changes from previous UML

`RemoveVariableValueAction` is unchanged from UML 1.5.

### 11.3.42 ReplyAction (from CompleteActions)

#### Generalizations

- “`AcceptEventAction` (from `CompleteActions`)” on page 228

#### Description

`ReplyAction` is an action that accepts a set of return values and a value containing return information produced by a previous `accept` call action. The reply action returns the values to the caller of the previous call, completing execution of the call.



## Attributes

No additional attributes

## Associations

- `replyToCall` : Trigger [1..1]      The trigger specifying the operation whose call is being replied to.
- `replyValue` : OutputPin [0..\*]      A list of pins containing the reply values of the operation. These values are returned to the caller.
- `returnInformation` : InputPin [1..1]      A pin containing the return information value produced by an earlier `AcceptCallAction`.

## Constraints

- [1] The reply value pins must match the return, out, and inout parameters of the operation on the event on the trigger in number, type, and order.
- [2] The event on `replyToCall` trigger must be a `CallEvent`.  
`replyToCallEvent.oclIsKindOf(CallEvent)`

## Semantics

The execution of a reply action completes the execution of a call that was initiated by a previous `AcceptCallAction`. The two are connected by the `returnInformation` value, which is produced by the `AcceptCallAction` and consumed by the `ReplyAction`. The information in this value is used by the execution engine to return the reply values to the caller and to complete execution of the original call. The details of transmitting call requests, encoding return information, and transmitting replies are opaque and unavailable to models, therefore they need not be and are not specified in this document.

Return information may be copied, stored in objects, and passed around, but it may only be used in a reply action once. If the same return information value is supplied to a second `ReplyAction`, the execution is in error and the behavior of the system is unspecified. It is not intended that any profile give any other meaning the return information. The operation specified by the call event on the trigger must be consistent with the information returned at runtime.

If the return information is lost to the execution or if a reply is never made, the caller will never receive a reply and therefore will never complete execution. This is not inherently illegal but it represents an unusual situation at the very least.

### 11.3.43 `SendObjectAction` (from `IntermediateActions`)

#### Generalizations

- “`InvocationAction` (from `BasicActions`)” on page 249

#### Description

`SendObjectAction` is an action that transmits an object to the target object, where it may invoke behavior such as the firing of state machine transitions or the execution of an activity. The value of the object is available to the execution of invoked behaviors. The requestor continues execution immediately. Any reply message is ignored and is not transmitted to the requestor.

## Attributes

No additional attributes

## Associations

- request: InputPin [1]      The request object, which is transmitted to the target object. The object may be copied in transmission, so identity might not be preserved. (Specialized from *InvocationAction.argument*)
- target: InputPin [1]      The target object to which the object is sent.

## Constraints

No additional constraints

## Semantics

- [1] When all the control and data flow prerequisites of the action execution are satisfied, the object on the input pin is transmitted to the target object. The target object may be local or remote. The object on the input pin may be copied during transmission, so identity might not be preserved. The manner of transmitting the object, the amount of time required to transmit it, the order in which the transmissions reach the various target objects, and the path for reaching the target objects are undefined.
- [2] When a transmission arrives at a target object, it may invoke behavior in the target object. The effect of receiving an object is specified in Chapter 13, “Common Behaviors.” Such effects include executing activities and firing state machine transitions.
- [3] A send object action receives no reply from the invoked behavior; any attempted reply is simply ignored, and no transmission is performed to the requestor.

## Notation

No specific notation

## Presentation Options

If the activity in which a send object action is used will always send a signal, then the `SendSignalAction` notation can be used.

## Rationale

Sends any object to a specified target object.

## Changes from previous UML

`SendObjectAction` is new in UML 2.0.

### 11.3.44 `SendSignalAction` (from `BasicActions`)

## Generalizations

- “`InvocationAction` (from `BasicActions`)” on page 249

## Description

SendSignalAction is an action that creates a signal instance from its inputs, and transmits it to the target object, where it may cause the firing of a state machine transition or the execution of an activity. The argument values are available to the execution of associated behaviors. The requestor continues execution immediately. Any reply message is ignored and is not transmitted to the requestor. If the input is already a signal instance, use SendObjectAction.

## Attributes

No additional attributes

## Associations

- signal: Signal [1]      The type of signal transmitted to the target object.
- target: InputPin [1]      The target object to which the signal is sent.

## Constraints

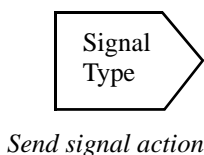
- [1] The number and order of argument pins must be the same as the number and order of attributes in the signal.
- [2] The type, ordering, and multiplicity of an argument pin must be the same as the corresponding attribute of the signal.

## Semantics

- [1] When all the prerequisites of the action execution are satisfied, a signal instance of the type specified by *signal* is generated from the argument values and his signal instance is transmitted to the identified target object. The target object may be local or remote. The signal instance may be copied during transmission, so identity might not be preserved. The manner of transmitting the signal object, the amount of time required to transmit it, the order in which the transmissions reach the various target objects, and the path for reaching the target objects are undefined.
- [2] When a transmission arrives at a target object, it may invoke behavior in the target object. The effect of receiving a signal object is specified in Chapter 13, “Common Behaviors.” Such effects include executing activities and firing state machine transitions.
- [3] A send signal action receives no reply from the invoked behavior; any attempted reply is simply ignored, and no transmission is performed to the requestor.

## Notation

A send signal action is notated with a convex pentagon.



**Figure 11.22 - Send signal notation**

## Examples

See extension in “SendSignalAction (as specialized)” on page 394.

### **Rationale**

Sends a signal to a specified target object.

### **Changes from previous UML**

Same as UML 1.5.

## **11.3.45 StartClassifierBehaviorAction (from CompleteActions)**

### **Generalizations**

- “Action (from BasicActions)” on page 230

### **Description**

StartClassifierBehaviorAction is an action that starts the classifier behavior of the input.

### **Attributes**

No additional attributes

### **Associations**

- object : InputPin [1..1]     Holds the object on which to start the owned behavior. (Specializes *Action.input*.)

### **Constraints**

[1] The multiplicity of the input pin is 1..1.

[2] If the input pin has a type, then the type must have a classifier behavior.

### **Semantics**

When a StartClassifierBehaviorAction is invoked, it initiates the classifier behavior of the classifier of the input object. If the behavior has already been initiated, or the object has no classifier behavior, this action has no effect.

### **Notation**

No specific notation

### **Rationale**

This action is provided to permit the explicit initiation of classifier behaviors, such as state machines and code, in a detailed, low-level “raw” specification of behavior.

### **Changes from previous UML**

StartClassifierBehaviorAction is a generalization of the UML 1.5 StartStateMachineAction.

## **11.3.46 StructuralFeatureAction (from IntermediateActions)**

(IntermediateActions) StructuralFeatureAction is an abstract class for all structural feature actions.

## Generalizations

- “Action (from BasicActions)” on page 230

## Description

This abstract action class statically specifies the structural feature being accessed.

The object to access is specified dynamically, by referring to an input pin on which the object will be placed at runtime. The type of the value of this pin is the classifier that owns the specified structural feature, and the value’s multiplicity is 1..1.

## Attributes

No additional attributes

## Associations

- structuralFeature : StructuralFeature [1..1]                      Structural feature to be read.
- object : InputPin [1..1] (Specialized from Action:input)      Gives the input pin from which the object whose structural feature is to be read or written is obtained.

## Constraints

- [1] The structural feature must not be static.  
    self.structuralFeature.isStatic = #false
- [2] The type of the object input pin is the same as the classifier of the object passed on this pin.
- [3] The multiplicity of the input pin must be 1..1.  
    self.object.multiplicity.is(1,1)
- [4] Visibility of structural feature must allow access to the object performing the action.  
    let host : Classifier = self.context in  
        self.structuralFeature.visibility = #public  
        or host = self.structuralFeature.featuringClassifier.type  
        or (self.structuralFeature.visibility = #protected and host.allSupertypes  
            ->includes(self.structuralFeature.featuringClassifier.type)))
- [5] A structural feature has exactly one featuringClassifier.  
    self.structuralFeature.featuringClassifier->size() = 1

## Semantics

A structural feature action operates on a statically specified structural feature of some classifier. The action requires an object on which to act, provided at runtime through an input pin. If the structural feature is an association end, then actions on the feature have the same semantics as actions on the links that have the feature as an end. See specializations of StructuralFeatureAction. The semantics is undefined for accessing a structural feature that violates its visibility. The semantics for static features are undefined.

The structural features of an object may change over time due to dynamic classification. However, the structural feature specified in a structural feature action is inherited from a single classifier, and it is assumed that the object passed to a structural feature action is classified by that classifier directly or indirectly. The structural feature is referred to as a user model element, so it is uniquely identified, even if there are other structural features of the same name on other classifiers.

## Notation

No specific notation

## Rationale

StructuralFeatureAction is introduced for the abstract aspects of structural feature actions.

## Changes from previous UML

StructuralFeatureAction is new in UML 2.0. It generalizes AttributeAction in UML 1.5.

### 11.3.47 TestIdentityAction (from IntermediateActions)

TestIdentifyAction is an action that tests if two values are identical objects.

## Generalizations

- “Action (from BasicActions)” on page 230

## Description

This action returns true if the two input values are the same identity, false if they are not.

## Attributes

No additional attributes

## Associations

- first: InputPin [1..1] (Specialized from Action:input) Gives the pin on which an object is placed.
- result: OutputPin [1..1] — (Specialized from Action:output) Tells whether the two input objects are identical.
- second: InputPin [1..1] — (Specialized from Action:input) Gives the pin on which an object is placed.

## Constraints

- [1] The input pins have no type.  
self.first.type->size() = 0  
and self.second.type->size() = 0
- [2] The multiplicity of the input pins is 1..1.  
self.first.multiplicity.is(1,1)  
and self.second.multiplicity.is(1,1)
- [3] The type of the result is Boolean.  
self.result.type.ocIsTypeOf(Boolean)

## Semantics

When all the prerequisites of the action have been satisfied, the input values are obtained from the input pins and made available to the computation. If the two input values represent the same object (regardless of any implementation-level encoding), the value true is placed on the output pin of the action execution, otherwise the value false is placed on the output pin. The execution of the action is complete.

## Notation

No specific notation

## Rationale

TestIdentityAction is introduced to tell when two values refer to the same object.

## Changes from previous UML

TestIdentityAction is unchanged from UML 1.5.

### 11.3.48 UnmarshallAction (from CompleteActions)

UnmarshallAction is an action that breaks an object of a known type into outputs each of which is equal to a value from a structural feature of the object.

## Generalizations

- “AcceptEventAction (from CompleteActions)” on page 228.

## Description

The outputs of this action correspond to the structural features of the specified type. The input must be of this type.

## Attributes

No additional attributes

## Associations

- object : InputPin [1..1]                      The object to be unmarshalled.
- unmarshallType : Classifier [1..1]      The type of the object to be unmarshalled.
- result : OutputPin [1..\*]                      The values of the structural features of the input object.

## Constraints

- [1] The type of the object input pin must be the same as the unmarshall classifier.
- [2] The multiplicity of the object input pin is 1..1.
- [3] The number of result output pins must be the same as the number of structural features of the unmarshall classifier.
- [4] The type and ordering of each result output pin must be the same as the corresponding structural features of the unmarshall classifier.
- [5] The multiplicity of each result output pin must be compatible with the multiplicity of the corresponding structural features of the unmarshall classifier.

[6] The unmarshall classifier must have at least one structural feature.

### **Semantics**

When an object is available on the input pin, the values of the structural features of the specified classifier are retrieved from the object and placed on the output pins, in the order of the structural features of the specified classifier.

### **Notation**

No specific notation

### **Examples**

See “UnmarshallAction (as specialized)” on page 398.

### **Rationale**

UnmarshallAction is introduced to read all the structural features of an object at once.

### **Changes from previous UML**

UnmarshallAction is the same as UML 1.5, except that the name of the metaassociation to the input pin is changed.

## **11.3.49 ValuePin (from BasicActions)**

### **Generalizations**

- “InputPin (from BasicActions)” on page 249

### **Description**

A value pin is an input pin that provides a value by evaluating a value specification.

### **Attributes**

No additional attributes

### **Associations**

- value : ValueSpecification [1..1]      Value that the pin will provide.

### **Constraints**

[1] The type of value specification must be compatible with the type of the value pin.

### **Semantics**

The value of the pin is the result of evaluating the value specification.

### **Notation**

No specific notation. See extensions in Activities.



## Rationale

ValuePin is introduced to provide the most basic way of providing inputs to actions.

## Changes from previous UML

ValuePin is new to UML 2.

### 11.3.50 ValueSpecificationAction (from IntermediateActions)

ValueSpecificationAction is an action that evaluates a value specification.

## Generalizations

- “Action (from BasicActions)” on page 230

## Description

The action returns the result of evaluating a value specification.

## Attributes

No additional attributes

## Associations

- value : ValueSpecification [1]      Value specification to be evaluated. {Specializes *Action::output*}

## Constraints

[1] The type of value specification must be compatible with the type of the result pin.

[2] The multiplicity of the result pin is 1..1.

## Semantics

The value specification is evaluated when the action is enabled.

## Notation

See “ValueSpecificationAction (as specialized)” on page 399.

## Examples

See “ValueSpecificationAction (as specialized)” on page 399.

## Rationale

ValueSpecificationAction is introduced for injecting constants and other value specifications into behavior.

## Changes from previous UML

ValueSpecificationAction replaces LiteralValueAction from UML 1.5.

### 11.3.51 VariableAction (from StructuredActions)

#### Generalizations

- “Action (from BasicActions)” on page 230

#### Description

VariableAction is an abstract class for actions that operate on a statically specified variable.

#### Attributes

No additional attributes

#### Associations

- variable : Variable [1..1]      Variable to be read.

#### Constraints

- [1] The action must be in the scope of the variable.  
self.variable.isAccessibleBy(self)

#### Semantics

Variable action is an abstract metaclass. For semantics see its concrete subtypes.

#### Notation

No specific notation

#### Rationale

VariableAction is introduced for the abstract aspects of variable actions.

#### Changes from previous UML

VariableAction is unchanged from UML 1.5.

### 11.3.52 WriteLinkAction (from IntermediateActions)

WriteLinkAction is an abstract class for link actions that create and destroy links.

#### Generalizations

- “LinkAction (from IntermediateActions)” on page 250

#### Description

A write link action takes a complete identification of a link and creates or destroys it.

#### Attributes

No additional attributes

## Associations

No additional associations

## Constraints

- [1] All end data must have exactly one input object pin.  
`self.endData.forall(value->size() = 1)`

## Semantics

See children of WriteLinkAction.

## Notation

No specific notation

## Rationale

WriteLinkAction is introduced to navigate across links.

## Changes from previous UML

WriteLinkAction is unchanged from UML 1.5.

### 11.3.53 WriteStructuralFeatureAction (from IntermediateActions)

WriteStructuralFeatureAction is an abstract class for structural feature actions that change structural feature values.

## Generalizations

- “StructuralFeatureAction (from IntermediateActions)” on page 275

## Description

A write structural feature action operates on a structural feature of an object to modify its values. It has an input pin on which the value that will be added or removed is put. Other aspects of write structural feature actions are inherited from StructuralFeatureAction.

## Attributes

No additional attributes

## Associations

- `value : InputPin [1..1]` (Specialized from Action:input)    Value to be added or removed from the structural feature.

## Constraints

- [1] The type input pin is the same as the classifier of the structural feature.  
`self.value.type = self.structuralFeature.featuringClassifier`
- [2] The multiplicity of the input pin is 1..1.  
`self.value.multiplicity.is(1,1)`

## Semantics

None.

## Notation

No specific notation

## Rationale

WriteStructuralFeatureAction is introduced to abstract aspects of structural feature actions that change structural feature values.

## Changes from previous UML

WriteStructuralFeatureAction is new in UML 2.0. It generalizes WriteAttributeAction in UML 1.5.

### 11.3.54 WriteVariableAction (from StructuredActions)

WriteVariableAction is an abstract class for variable actions that change variable values.

## Generalizations

- “VariableAction (from StructuredActions)” on page 281

## Description

A write variable action operates on a variable to modify its values. It has an input pin on which the value that will be added or removed is put. Other aspects of write variable actions are inherited from VariableAction.

## Attributes

No additional attributes

## Associations

- value : InputPin [1..1] (Specialized from Action:input)    Value to be added or removed from the variable.

## Constraints

- [1] The type input pin is the same as the type of the variable.  
self.value.type = self.variable.type
- [2] The multiplicity of the input pin is 1..1.  
self.value.multiplicity.is(1,1)

## Semantics

See children of WriteVariableAction.

## Notation

No specific notation

### **Rationale**

WriteVariableAction is introduced to abstract aspects of structural feature actions that change variable values.

### **Changes from previous UML**

WriteVariableAction is unchanged from UML 1.5.

## **11.4 Diagrams**

See “Diagrams” on page 402.

## 12 Activities

### 12.1 Overview

Activity modeling emphasizes the sequence and conditions for coordinating lower-level behaviors, rather than which classifiers own those behaviors. These are commonly called control flow and object flow models. The actions coordinated by activity models can be initiated because other actions finish executing, because objects and data become available, or because events occur external to the flow.

#### Actions and activities

An *action execution* corresponds to the execution of a particular action. Similarly, an *activity execution* is the execution of an activity, ultimately including the executions of actions within it. Each action in an activity may execute zero, one, or more times for each activity execution. At the minimum, actions need access to data, they need to transform and test data, and actions may require sequencing. The activities specification (at the higher compliance levels) allows for several (logical) threads of control executing at once and synchronization mechanisms to ensure that activities execute in a specified order. Semantics based on concurrent execution can then be mapped easily into a distributed implementation. However, the fact that the UML allows for concurrently executing objects does not necessarily imply a distributed software structure. Some implementations may group together objects into a single task and execute sequentially—so long as the behavior of the implementation conforms to the sequencing constraints of the specification.

There are potentially many ways of implementing the same specification, and any implementation that preserves the information content and behavior of the specification is acceptable. Because the implementation can have a different structure from that of the specification, there is a mapping between the specification and its implementation. This mapping need not be one-to-one: an implementation need not even use object-orientation, or it might choose a different set of classes from the original specification.

The mapping may be carried out by hand by overlaying physical models of computers and tasks for implementation purposes, or the mapping could be carried out automatically. This specification neither provides the overlays, nor does it provide for code generation explicitly, but the specification makes both approaches possible.

See the “Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)” and “Action (from CompleteActivities, FundamentalActivities, StructuredActivities)” metaclasses for more introduction and semantic framework.

#### FundamentalActivities

The fundamental level defines activities as containing nodes, which includes actions. This level is shared between the flow and structured forms of activities.

#### BasicActivities

This level includes control sequencing and data flow between actions, but explicit forks and joins of control, as well as decisions and merges, are not supported. The basic and structured levels are orthogonal. Either can be used without the other or both can be used to support modeling that includes both flows and structured control constructs.

#### IntermediateActivities

The intermediate level supports modeling of activity diagrams that include concurrent control and data flow, and decisions. It supports modeling similar to traditional Petri nets with queuing. It requires the basic level.

The intermediate and structured levels are orthogonal. Either can be used without the other or both can be used to support modeling that includes both concurrency and structured control constructs.

### **CompleteActivities**

The complete level adds constructs that enhance the lower level models, such as edge weights and streaming.

### **StructuredActivities**

The structured level supports modeling of traditional structured programming constructs, such as sequences, loops, and conditionals, as an addition to fundamental activity nodes. It requires the basic level. It is compatible with the intermediate and complete levels.

### **CompleteStructuredActivities**

This level adds support for data flow output pins of sequences, conditionals, and loops. It depends on the basic layer for flows.

### **ExtraStructuredActivities**

The extra structure level supports exception handling as found in traditional programming languages and invocation of behaviors on sets of values. It requires the structured level.

## 12.2 Abstract Syntax

Figure 12.1 shows the dependencies of the activity packages.

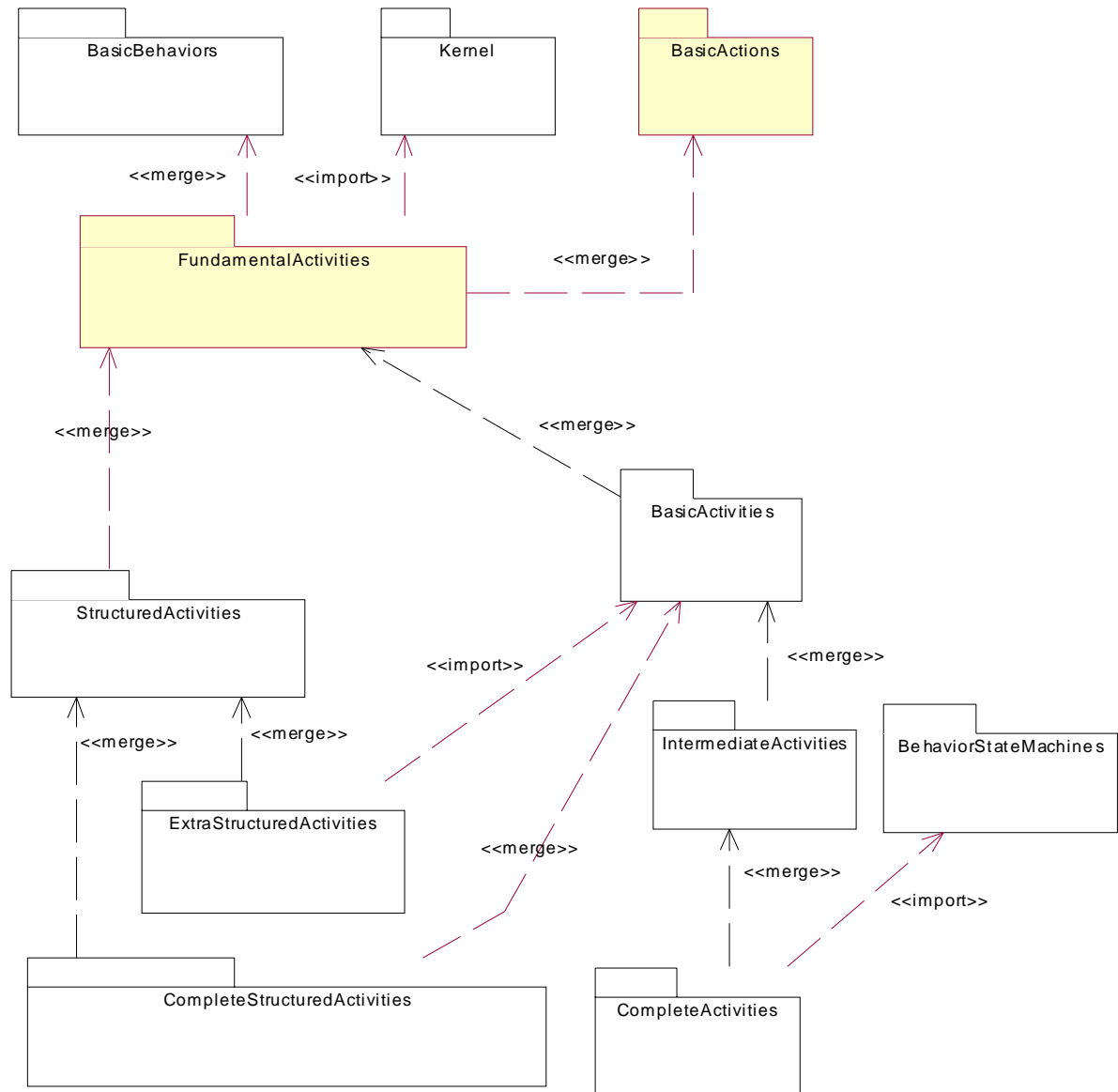


Figure 12.1 - Dependencies of the Activity packages



## Package FundamentalActivities

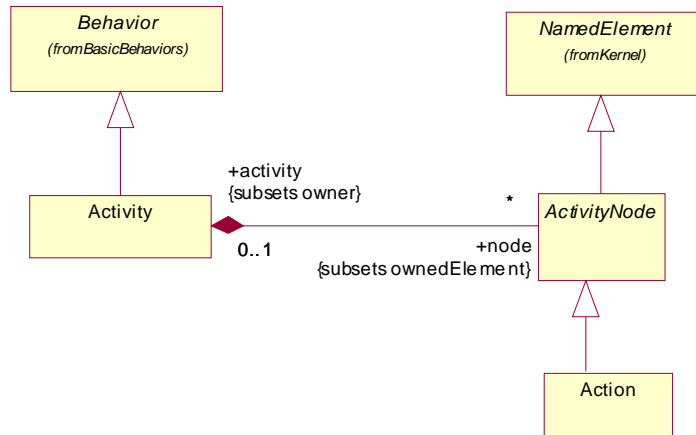


Figure 12.2 - Fundamental nodes

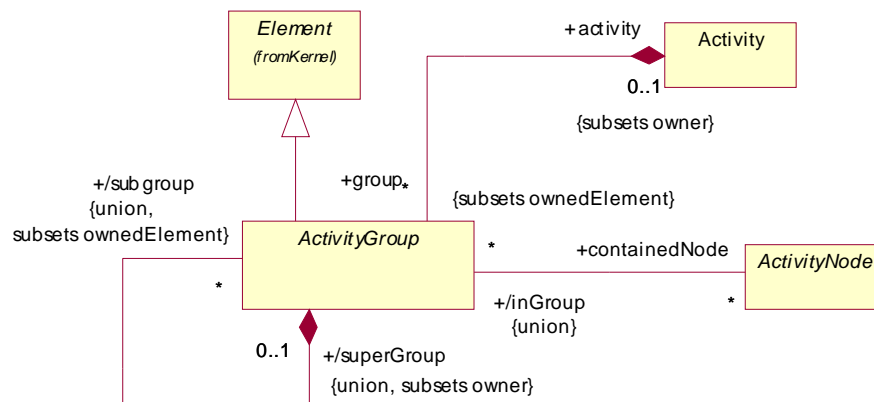


Figure 12.3 - Fundamental groups

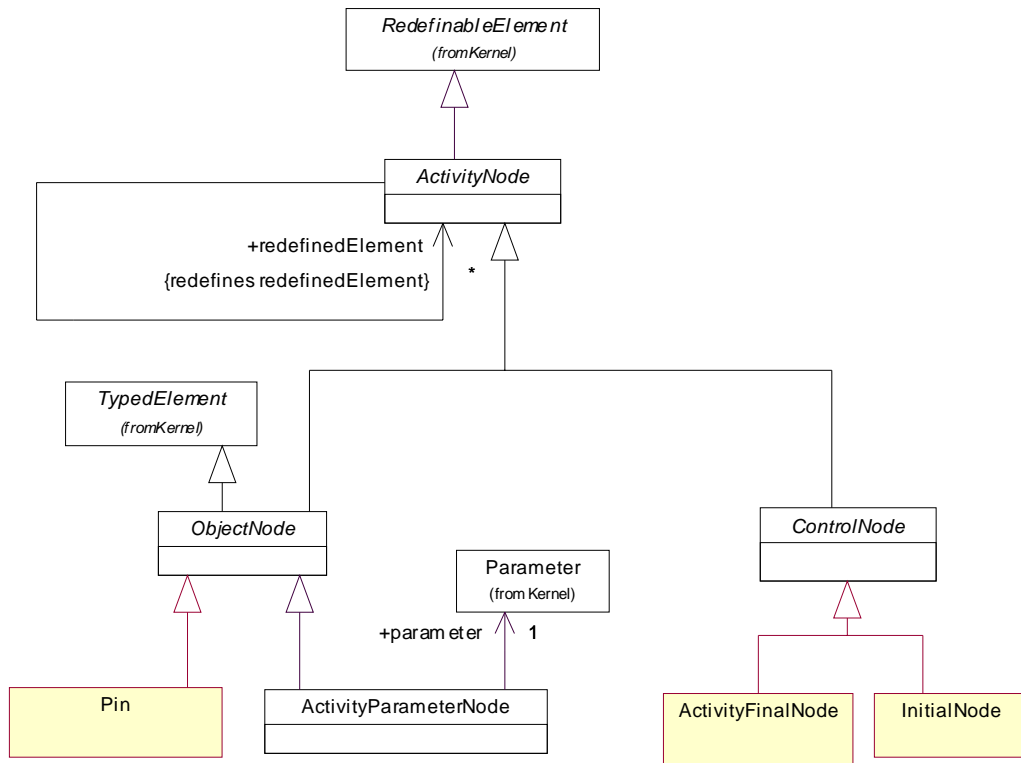


Figure 12.4 - Nodes (BasicActivities)

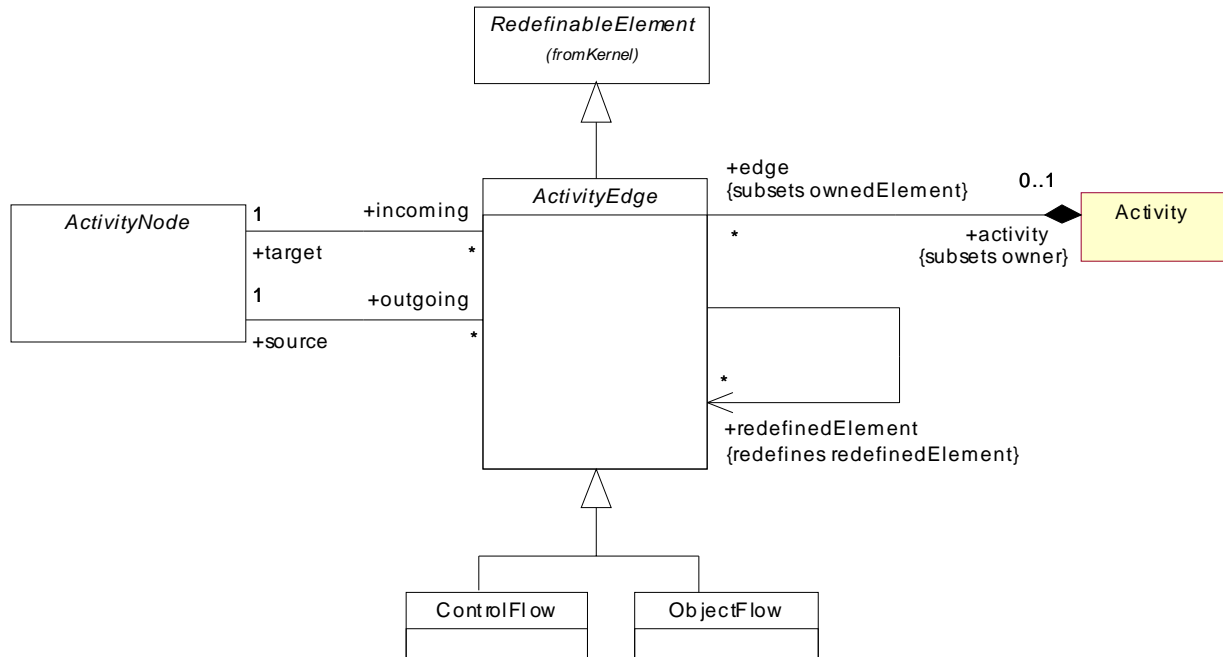


Figure 12.5 - Flows

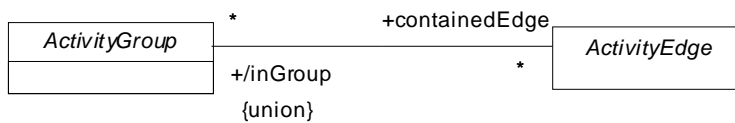


Figure 12.6 - Groups

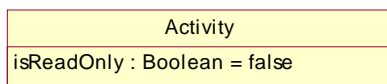
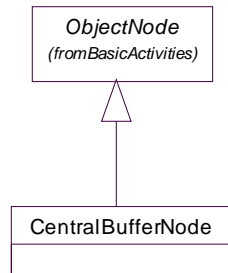
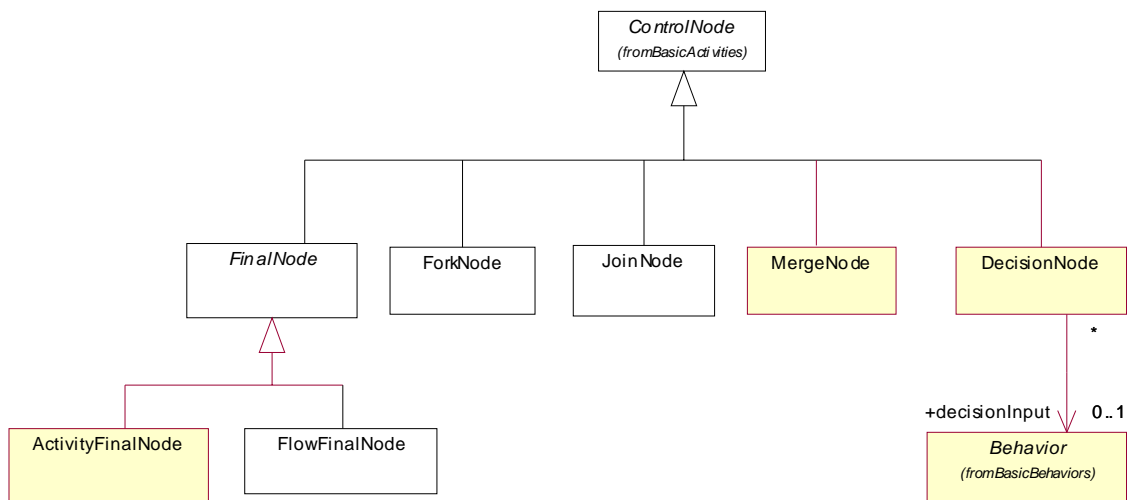


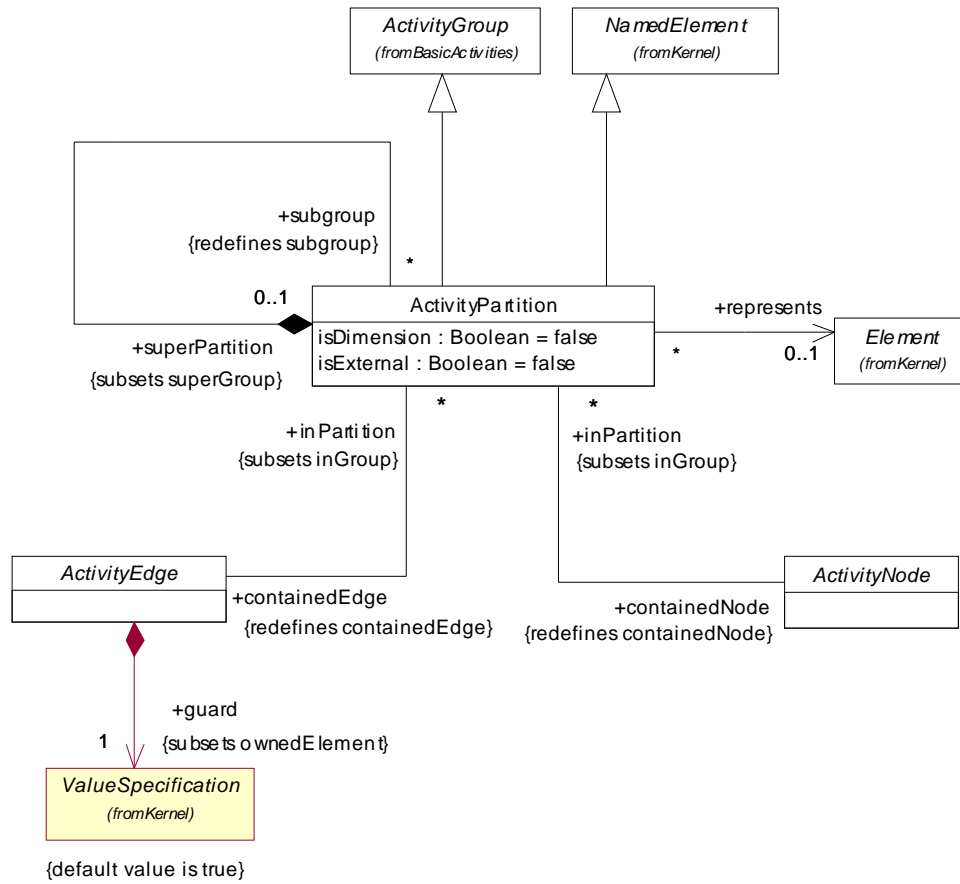
Figure 12.7 - Elements



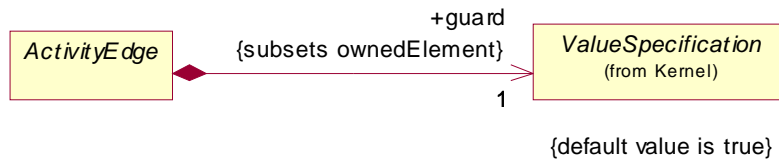
**Figure 12.8 - Object nodes (*IntermediateActivities*)**



**Figure 12.9 - Control nodes (*IntermediateActivities*)**



**Figure 12.10 - Partitions**



**Figure 12.11 - Flows (IntermediateActivities)**