# Part 5 - Static Structure Diagrams

Class diagrams show the static structure of the model, in particular, the things that exist (such as classes and types), their internal structure, and their relationships to other things. Class diagrams do not show temporal information, although they may contain reified occurrences of things that have or things that describe temporal behavior. An object diagram shows instances compatible with a particular class diagram.

This section discusses classes and their variations, including templates and instantiated classes, and the relationships between classes (association and generalization) and the contents of classes (attributes and operations).

## 3.19 Class Diagram

A class diagram is a graph of Classifier elements connected by their various static relationships. Note that a “class” diagram may also contain interfaces, packages, relationships, and even instances, such as objects and links. Perhaps a better name would be “static structural diagram” but “class diagram” is shorter and well established.

### 3.19.1 Semantics

A class diagram is a graphic view of the static structural model. The individual class diagrams do not represent divisions in the underlying model.

### 3.19.2 Notation

A class diagram is a collection of static declarative model elements, such as classes, interfaces, and their relationships, connected as a graph to each other and to their contents. Class diagrams may be organized into packages either with their underlying models or as separate packages that build upon the underlying model packages.

### 3.19.3 Mapping

A class diagram does not necessarily match a single semantic entity. A package within the static structural model may be represented by one or more class diagrams. The division of the presentation into separate diagrams is for graphical convenience and does not imply a partitioning of the model itself. The contents of a diagram map into elements in the static semantic model. If a diagram is part of a package, then its contents map into elements in the same package (including possible references to elements accessed or imported from other packages).

## 3.20 Object Diagram

An object diagram is a graph of instances, including objects and data values. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time. The use of object diagrams is fairly limited, mainly to show examples of data structures.

Tools need not support a separate format for object diagrams. Class diagrams can contain objects, so a class diagram with objects and no classes is an “object diagram.” The phrase is useful, however, to characterize a particular usage achievable in various ways.

## 3.21 Classifier

*Classifier* is the metamodel superclass of *Class, DataType,* and *Interface.* All of these have similar syntax and are therefore all notated using the rectangle symbol with keywords used as necessary. Because classes are most common in diagrams, a rectangle without a keyword represents a class, and the other subclasses of *Classifier* are indicated with keywords. In the sections that follow, the discussion will focus on *Class,* but most of the notation applies to the other element kinds as semantically appropriate and as described later under their own sections.

## 3.22 Class

A *class* is the descriptor for a set of objects with similar structure, behavior, and relationships. The model is concerned with describing the intension of the class, that is, the rules that define it. The run-time execution provides its extension, that is, its instances. UML provides notation for declaring classes and specifying their properties, as well as using classes in various ways. Some modeling elements that are similar in form to classes (such as interfaces, signals, or utilities) are notated using keywords on class symbols; some of these are separate metamodel classes and some are stereotypes of Class. Classes are declared in class diagrams and used in most other diagrams. UML provides a graphical notation for declaring and using classes, as well as a textual notation for referencing classes within the descriptions of other model elements.

### 3.22.1 Semantics

A class represents a concept within the system being modeled. Classes have data structure and behavior and relationships to other elements.

The name of a class has scope within the package in which it is declared and the name must be unique (among class names) within its package.

### 3.22.2 Basic Notation

A class is drawn as a solid-outline rectangle with three compartments separated by horizontal lines. The top name compartment holds the class name and other general properties of the class (including stereotype); the middle list compartment holds a list of attributes; the bottom list compartment holds a list of operations.

See Section 3.23, “Name Compartment,” on page 3-38 and Section 3.24, “List Compartment,” on page 3-38 for more details.

#### 3.22.2.1 References

By default a class shown within a package is assumed to be defined within that package. To show a reference to a class defined in another package, use the syntax

*Package-name*::*Class-name*

as the name string in the name compartment. A full pathname can be specified by chaining together package names separated by double colons (::).

### 3.22.3 Presentation Options

Either or both of the attribute and operation compartments may be suppressed. A separator line is not drawn for a missing compartment. If a compartment is suppressed, no inference can be drawn about the presence or absence of elements in it. Compartment names can be used to remove ambiguity, if necessary (Section3.24, “ List Compartment,” on page 3-38).

Additional compartments may be supplied as a tool extension to show other predefined or user-defined model properties (for example, to show business rules, responsibilities, variations, events handled, exceptions raised, and so on). Most compartments are simply lists of strings. More complicated formats are possible, but UML does not specify such formats; they are a tool responsibility. Appearance of each compartment should preferably be implicit based on its contents. Compartment names may be used, if needed.

Tools may provide other ways to show class references and to distinguish them from class declarations.

A class symbol with a stereotype icon may be “collapsed” to show just the stereotype icon, with the name of the class either inside the class or below the icon. Other contents of the class are suppressed.

### 3.22.4 Style Guidelines

- Center class name in boldface.
- Center keyword (including stereotype names) in plain face within guillemets above class name.

- For those languages that distinguish between uppercase and lowercase characters, capitalize class names; that is, begin them with an uppercase character.
- Left justify attributes and operations in plain face.
- Begin attribute and operation names with a lowercase letter.
- Show the names of abstract classes or the signatures of abstract operations in italics.

As a tool extension, boldface may be used for marking special list elements; for example, to designate candidate keys in a database design. This might encode some design property modeled as a tagged value, for example.

Show full attributes and operations when needed and suppress them in other contexts or references.

### 3.22.5 Example

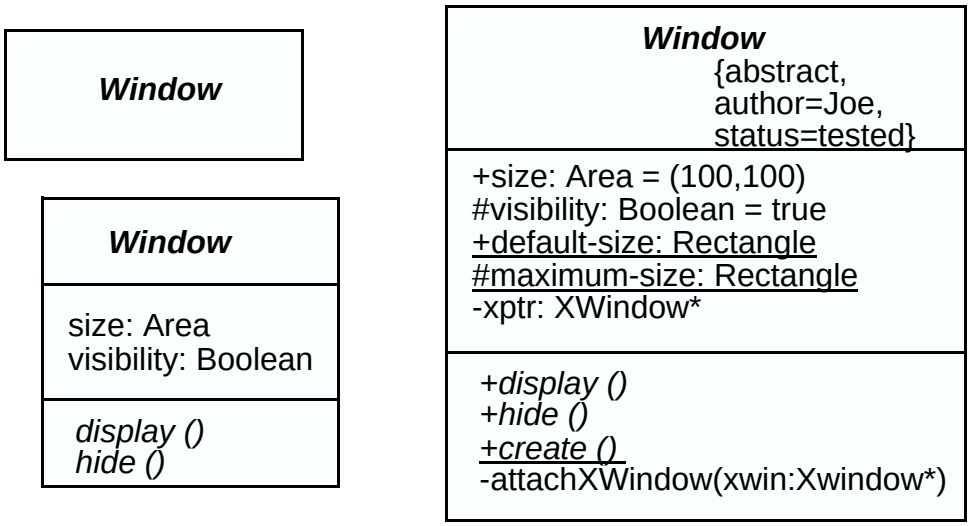


*Figure 3-20* Class Notation: Details Suppressed, Analysis-level Details, Implementation-level Details

### 3.22.6 Mapping

A class symbol maps into a Class element within the package that owns the diagram. The name compartment contents map into the class name and into properties of the class (built-in attributes or tagged values). The attribute compartment maps into a list of Attributes of the Class. The operation compartment maps into a list of Operations of the Class.

The property string {location=*name*} maps into an implementationLocation association to a Component. The *name* is the name of the containing Component.

## 3.23 Name Compartment

### 3.23.1 Notation

The name compartment displays the name of the class and other properties in up to three sections:

An optional stereotype keyword may be placed above the class name within guillemets, and/or a stereotype icon may be placed in the upper right corner of the compartment. The stereotype name must not match a predefined keyword.

The name of the class appears next. If the class is abstract, this can be indicated by italicizing its name (for those languages that support italicization) or by placing the keyword *abstract* in a property list below or after the name; for example, Invoice {abstract}. Note that any explicit specification of generalization status takes precedence over the name font.

A list of strings denoting properties (metamodel attributes or tagged values) may be placed in braces below the class name. The list may show class-level attributes for which there is no UML notation and it may also show tagged values. The presence of a keyword for a Boolean type without a value implies the value *true*. For example, a leaf class shows the property "{leaf}".

The stereotype and property list are optional.

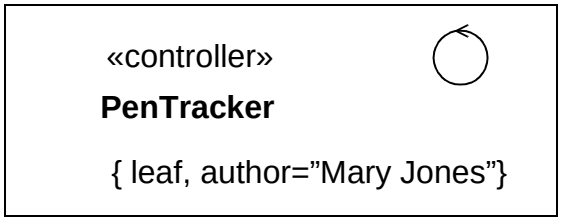


*Figure 3-21* Name Compartment

### 3.23.2 Mapping

The contents of the name compartment map into the name, stereotype, and various properties of the Class represented by the class symbol.

## 3.24 List Compartment

### 3.24.1 Notation

A list compartment holds a list of strings, each of which is the encoded representation of a feature, such as an attribute or operation. The strings are presented one to a line with overflow to be handled in a tool-dependent manner. In addition to lists of

attributes or operations, optional lists can show other kinds of predefined or user-defined values, such as responsibilities, rules, or modification histories. UML does not define these optional lists. The manipulation of user-defined lists is tool-dependent.

The items in the list are ordered and the order may be modified by the user. The order of the elements is meaningful information and must be accessible within tools (for example, it may be used by a code generator in generating a list of declarations). The list elements may be presented in a different order to achieve some other purpose (for example, they may be sorted in some way). Even if the list is sorted, the items maintain their original order in the underlying model. The ordering information is merely suppressed in the view.

An ellipsis ( . . . ) as the final element of a list or the final element of a delimited section of a list indicates that additional elements in the model exist that meet the selection condition, but that are not shown in that list. Such elements may appear in a different view of the list.

#### 3.24.1.1 Group properties

A property string may be shown as an element of the list, in which case it applies to all of the succeeding list elements until another property string appears as a list element. This is equivalent to attaching the property string to each of the list elements individually. The property string does not designate a model element. Examples of this usage include indicating a stereotype and specifying visibility. Keyword strings may also be used in a similar way to qualify subsequent list elements.

#### 3.24.1.2 Compartment name

A compartment may display a name to indicate which kind of compartment it is. The name is displayed in a distinctive font centered at the top of the compartment. This capability is useful if some compartments are omitted or if additional user-defined compartments are added. For a Class, the predefined compartments are named **attributes** and **operations**. An example of a user-defined compartment might be **requirements**. The name compartment in a class must always be present; therefore, it does not require or permit a compartment name.

### 3.24.2 Presentation Options

A tool may present the list elements in a sorted order, in which case the inherent ordering of the elements is not visible. A sort is based on some internal property and does not indicate additional model information. Example sort rules include:

- alphabetical order,
- ordering by stereotype (such as constructors, destructors, then ordinary methods),
- ordering by visibility (public, then package, then protected, then private).

The elements in the list may be filtered according to some selection rule. The specification of selection rules is a tool responsibility. The absence of items from a filtered list indicates that no elements meet the filter criterion, but no inference can be

drawn about the presence or absence of elements that do not meet the criterion. However, the ellipsis notation is available to show that invisible elements exist. It is a tool responsibility whether and how to indicate the presence of either local or global filtering, although a stand-alone diagram should have some indication of such filtering if it is to be understandable.

If a compartment is suppressed, no inference can be drawn about the presence or absence of its elements. An empty compartment indicates that no elements meet the selection filter (if any).

Note that attributes may also be shown by composition (see Figure 3-45 on page 3-83).

### 3.24.3 Example

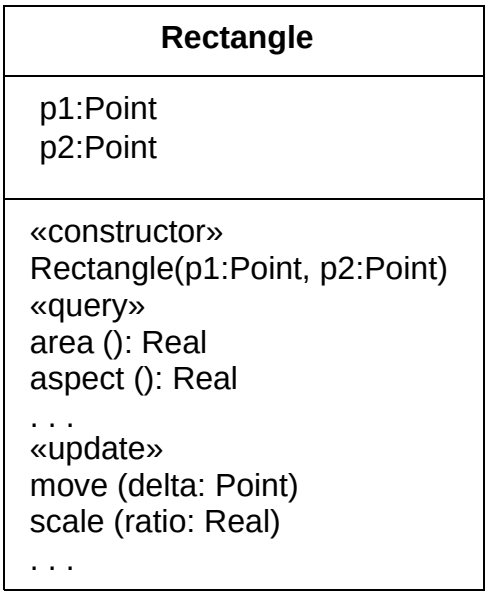


*Figure 3-22* Stereotype Keyword Applied to Groups of List Elements

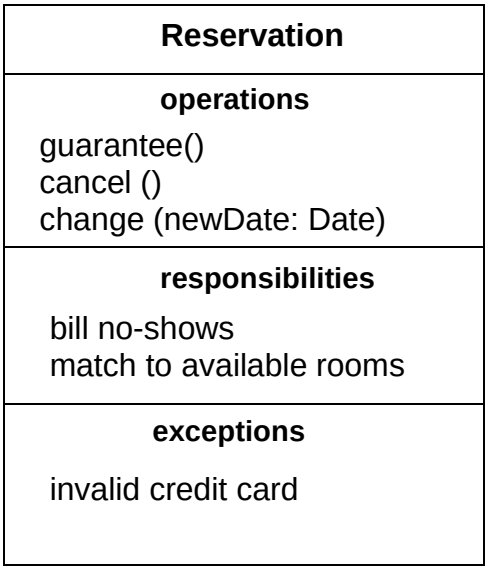


*Figure 3-23* Compartments with Names

### 3.24.4 Mapping

The entries in a list compartment map into a list of ModelElements, one for each list entry. The ordering of the ModelElements matches the list compartment entries (unless the list compartment is sorted in some way). In this case, no implication about the ordering of the Elements can be made (the ordering can be seen by turning off sorting). However, a list entry string that is a stereotype indication (within guillemets) or a property indication (within braces) does not map into a separate ModelElement. Instead, the corresponding property applies to each subsequent ModelElement until the appearance of a different stand-alone stereotype or property indicator. The property specifications are conceptually duplicated for each list Element, although a tool might maintain an internal mechanism to store or modify them together. The presence of an ellipsis ("...") as a list entry implies that the semantic model contains at least one Element with corresponding properties that is not visible in the list compartment.

## 3.25 Attribute

Strings in the attribute compartment are used to show attributes in classes. A similar syntax is used to specify qualifiers, template parameters, operation parameters, and so on (some of these omit certain terms).

### 3.25.1 Semantics

Note that an attribute is semantically equivalent to a composition association; however, the intent and usage is normally different.

The type of an attribute is a Classifier.

### 3.25.2 Notation

An attribute is shown as a text string that can be parsed into the various properties of an attribute model element. The default syntax is:

*visibility name* : *type-expression* [ *multiplicity ordering* ] = *initial-value* { *property-string* }

- Where *visibility* is one of:

  \+ public visibility

  # protected visibility

  - private visibility

  ~ .package visibility

  The visibility marker may be suppressed. The absence of a visibility marker indicates that the visibility is not shown (not that it is undefined or public). A tool should assign visibilities to new attributes even if the visibility is not shown. The visibility marker is a shorthand for a full *visibility* property specification string.

  Visibility may also be specified by keywords (*public, protected, private, package*). This form is used particularly when it is used as an inline list element that applies to an entire block of attributes.

  Additional kinds of visibility might be defined for certain programming languages, such as C++ *implementation* visibility (actually all forms of nonpublic visibility are language-dependent). Such visibility must be specified by property string or by a tool-specific convention.

- Where *name* is an identifier string that represents the name of the attribute.
- Where [ *multiplicity ordering*] shows the multiplicity and the ordering of the attribute (Section 3.44, “Multiplicity,” on page 3-75). The term may be omitted, in which case the multiplicity is 1..1 (exactly one).
- The *ordering* property is meaningful if the multiplicity upper bound is greater than one. It may be one of:
  - *(absent)* — the values are unordered
  - unordered — the values are unordered
  - ordered — the values are ordered
- Where *type-expression* is either
  - if it is a simple word, the name of a classifier, or
  - a language-dependent string that maps into a ProgrammingLanguageDataType.
- Where *initial-value* is a language-dependent expression for the initial value of a newly created object. The initial value is optional (the equal sign is also omitted). An explicit constructor for a new object may augment or modify the default initial value.

- Where *property-string* indicates property values that apply to the element. The property string is optional (the braces are omitted if no properties are specified).

A class-scope attribute is shown by underlining the name and type expression string; otherwise, the attribute is instance-scope.

class-scope-attribute

The notation justification is that a class-scope attribute is an instance value in the executing system, just as an object is an instance value, so both may be designated by underlining. An instance-scope attribute is not underlined; that is the default.

There is no symbol for whether an attribute is changeable (the default is changeable). A nonchangeable attribute is specified with the property "{frozen}".

In the absence of a multiplicity indicator, an attribute holds exactly 1 value. Multiplicity may be indicated by placing a multiplicity indicator in brackets after the classifier name, for example:

colors : Color [3]
points : Point [2..* ordered]

Note that a multiplicity of 0..1 provides for the possibility of null values: the absence of a value, as opposed to a particular value from the range. For example, the following declaration permits a distinction between the *null* value and the empty string:

name : String [0..1]

A stereotype keyword in guillemets precedes the entire attribute string, including any visibility indicators. A property list in braces follows the rest of the attribute string.

### 3.25.3 Presentation Options

The type expression may be suppressed (but it has a value in the model).

The initial value may be suppressed, and it may be absent from the model. It is a tool responsibility whether and how to show this distinction.

A tool may show the visibility indication in a different way, such as by using a special icon or by sorting the elements by group.

A tool may show the individual fields of an attribute as columns rather than a continuous string.

If the type-expression string is not a word, then it is assumed to be expressed in the syntax of a particular programming language, such as C++ or Smalltalk. This form is assumed if the string is not a word. Specific tagged properties may be included in the string. The programming language must be known from the general context of the diagram or a tool supporting it. In this case, the type-expression maps into a ProgrammingLanguageDataType whose expression attribute specifies the language name and the string representation of the data type in that language.

Particular attributes within a list may be suppressed (see Section 3.24, "List Compartment," on page 3-38).

### 3.25.4 Style Guidelines

Attribute names typically begin with a lowercase letter. Attribute names are in plain face.

### 3.25.5 Example

+size: Area = (100,100)
#visibility: Boolean = invisible
+default-size: Rectangle
#maximum-size: Rectangle
-xptr: XWindowPtr

### 3.25.6 Mapping

A string entry within the attribute compartment maps into an Attribute within the Class corresponding to the class symbol. The properties of the attribute map in accord with the preceding descriptions. If the visibility is absent, then no conclusion can be drawn about the Attribute visibilities unless a filter is in effect; for example, only public attributes shown. Likewise, if the type or initial value are omitted. The omission of an underline always indicates an instance-scope attribute. The omission of multiplicity denotes a multiplicity of 1.

Any properties specified in braces following the attribute string map into properties on the Attribute. In addition, any properties specified on a previous stand-alone property specification entry apply to the current Attribute (and to others).

## 3.26 Operation

Entries in the operation compartment are strings that show operations defined on classes and methods supplied by classes.

### 3.26.1 Semantics

An operation is a service that an instance of the class may be requested to perform. It has a name and a list of arguments.

### 3.26.2 Notation

An operation is shown as a text string that can be parsed into the various properties of an operation model element. The default syntax is:

*visibility name* ( *parameter-list* ) : *return-type-expression* { *property-string* }

- Where *visibility* is one of:

  \+ public visibility

  \# protected visibility

- private visibility

~ package visibility

The visibility marker may be suppressed. The absence of a visibility marker indicates that the visibility is not shown (not that it is undefined or public). The visibility marker is a shorthand for a full *visibility* property specification string.

Visibility may also be specified by keywords (*public, protected, private, package*). This form is used particularly when it is used as an inline list element that applies to an entire block of operations.

Additional kinds of visibility might be defined for certain programming languages, such as C++ *implementation* visibility (actually all forms of nonpublic visibility are language-dependent). Such visibility must be specified by property string or by a tool-specific convention.

- Where *name* is an identifier string.
- Where *return-type-expression* is a language-dependent specification of the implementation type or types of the value returned by the operation. The colon and the return-type are omitted if the operation does not return a value (as for C++ void). A list of expressions may be supplied to indicate multiple return values.
- Where *parameter-list* is a comma-separated list of formal parameters, each specified using the syntax:

  *kind name* : *type-expression* = *default-value*

  - where *kind* is **in, out,** or **inout**, with the default **in** if absent.
  - where *name* is the name of a formal parameter.
  - where *type-expression* is the (language-dependent) specification of an implementation type.
  - where *default-value* is an optional value expression for the parameter, expressed in and subject to the limitations of the eventual target language.
- Where *property-string* indicates property values that apply to the element. The property string is optional (the braces are omitted if no properties are specified).

A class-scope operation is shown by underlining the name and type expression string. An instance-scope operation is the default and is not marked.

An operation that does not modify the system state (one that has no side effects) is specified by the property "{query}"; otherwise, the operation may alter the system state, although there is no guarantee that it will do so.

The concurrency semantics of an operation are specified by a property string of the form "{concurrency = *name*}, where *name* is one of the names: *sequential, guarded, concurrent*. As a shorthand, one of the names may be used by itself in a property string to indicate the corresponding concurrency value. In the absence of a specification, the concurrency semantics are unspecified and must therefore be assumed to be sequential in the worst case.

The top-most appearance of an operation signature declares the operation on the class (and inherited by all of its descendents). If this class does not implement the operation; that is, does not supply a method, then the operation may be marked as "{abstract}" or the operation signature may be italicized to indicate that it is abstract. A subordinate appearance of the operation signature without the {abstract} property indicates that the subordinate class implements a method on the operation.

The actual text or procedure of a method may be indicated in a note attached to the operation.

If the objects of a class accept and respond to a given signal, an operation entry with the keyword «signal» indicates that the class accepts the given signal. The syntax is identical to that of an operation. The response of the object to the reception of the signal is shown with a state machine. Among other uses, this notation can show the response of objects of a class to error conditions and exceptions, which should be modeled as signals.

The specification of operation behavior is given as a note attached to the operation. The text of the specification should be enclosed in braces if it is a formal specification in some language (a semantic Constraint); otherwise, it should be plain text if it is just a natural-language description of the behavior (a Comment).

A stereotype keyword in guillemets precedes the entire operation string, including any visibility indicators. A property list in braces follows the entire operation string.

### 3.26.3 Presentation Options

The argument list and return type may be suppressed (together, not separately).

A tool may show the visibility indication in a different way, such as by using a special icon or by sorting the elements by group.

The syntax of the operation signature string can be that of a particular programming language, such as C++ or Smalltalk. Specific tagged properties may be included in the string.

A procedure body for a method may be shown in a note attached to the operation entry within the compartment (Figure 3-24 on page 3-47). The line is drawn to the string within the compartment. This approach is useful mainly for showing small method bodies.

.

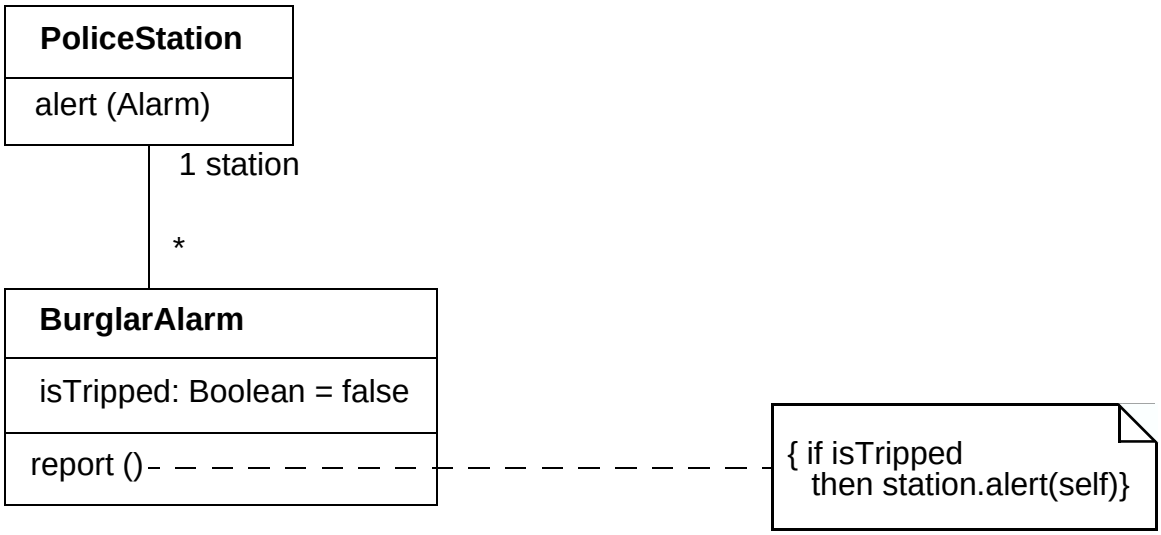


*Figure 3-24* Note showing method body

### 3.26.4 Style Guidelines

Operation names typically begin with a lowercase letter. Operation names are in plain face. An abstract operation may be shown in italics.

### 3.26.5 Example

*+display (): Location*
*+hide ()*
*+create ()*
-attachXWindow(xwin:Xwindow*)

*Figure 3-25* Operation List with a Variety of Operations

### 3.26.6 Mapping

A string entry within the operation compartment maps into an Operation or a Method within the Class corresponding to the class symbol. The properties of the operation map in accordance with the preceding descriptions. See the description of Section3.25, “Attribute,” on page 3-41 for additional details. Parameters without keywords map into Parameters with kind=in, otherwise according to the keyword. Return value names map into Parameters with kind=return.

If the entry has the keyword «signal», then it maps into a Reception on the Class instead.

The topmost appearance of an operation specification in a class hierarchy maps into an Operation definition in the corresponding Class or Interface. Interfaces do not have methods. In a Class, each appearance of an operation entry maps into the presence of a Method in the corresponding Class, unless the operation entry contains the {abstract} property (including use of conventions such as italics for abstract operations). If an abstract operation entry appears within a hierarchy in which the same operation has already been defined in an ancestor, it has no effect but is not an error unless the declarations are inconsistent.

Note that the operation string entry does not specify the body of a method.

## 3.27 Nested Class Declarations

### 3.27.1 Semantics

A class declared within another class belongs to the namespace of the other class and may only be used within it. This construct is primarily used for implementation reasons and for information hiding.

### 3.27.2 Notation

A declaring class and a class in its namespace may be connected by a line, with an “anchor” icon on the end connected to a declaring class (Figure 3-26 on page 3-48). An anchor icon is a cross inside a circle. The contents of the package are declared within the class and belong to its namespace.

### 3.27.3 Mapping

If Class B is attached to Class A by an “anchor” line with the “anchor” symbol on Class A, then Class B is declared within the Namespace of Class A. That is, the relationship between Class A and Class B is the namespace-ownedElement association.

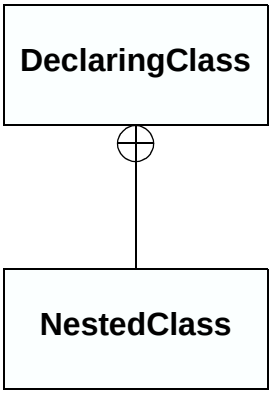


*Figure 3-26* Nested class declaration

## 3.28 Type and Implementation Class

### 3.28.1 Semantics

Classes can be stereotyped as Types or Implementation Classes (although they can be left undifferentiated as well). A Type is used to specify a domain of objects together with operations applicable to the objects without defining the physical implementation of those objects. A Type may not include any methods, but it may provide behavioral specifications for its operations. It may also have attributes and associations that are defined solely for the purpose of specifying the behavior of the type's operations.

An Implementation Class defines the physical data structure (for attributes and associations) and methods of an object as implemented in traditional languages (C++, Smalltalk, etc.). An Implementation Class is said to *realize* a Type if it provides all of the operations defined for the Type with the same behavior as specified for the Type's operations. An Implementation Class may realize a number of different Types.

### 3.28.2 Notation

An undifferentiated class is shown with no stereotype. A type is shown with the stereotype "«type»." An implementation class is shown with the stereotype "«implementationClass»." A tool is also free to allow a default setting for an entire diagram, in which case all of the class symbols without explicit stereotype indications map into Classes with the default stereotype. This might be useful for a model that is close to the programming level.

The implementation of a type by a class is modeled as the Realization relationship, shown as a dashed line with a solid triangular arrowhead (a dashed "generalization arrow"). This symbol implies the realizing class provides at least all the operations of the Type, with conforming behavior, but it does not imply inheritance of structure (attributes or associations). The generalization hierarchy of a set of classes frequently parallels the generalization hierarchy of a set of types that they realize, but this is not mandatory, as long as each class provides the operations of the types that it realizes.

### 3.28.3 Example

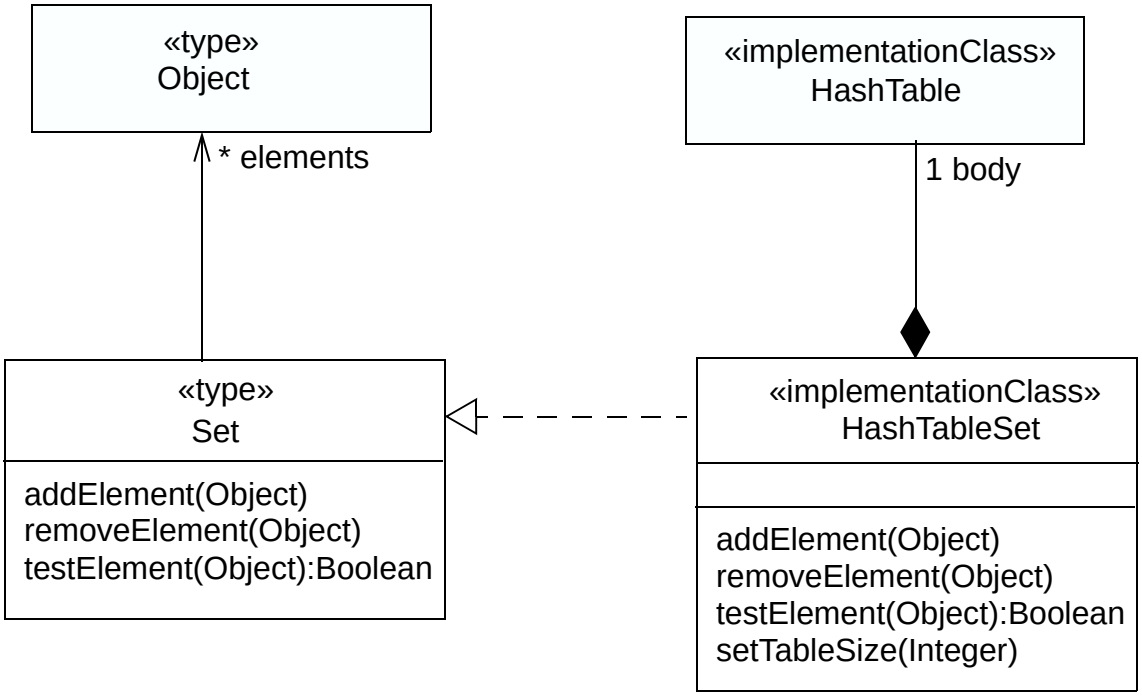


*Figure 3-27* Notation for Types and Implementation Classes

### 3.28.4 Mapping

A class symbol with a stereotype (including “type” and “implementationClass”) maps into a Class with the corresponding stereotype. A class symbol without a stereotype maps into a Class with the default stereotype for the diagram (if a default has been defined by the modeler or tool); otherwise, it maps into a Class with no stereotype. The realization arrow between two symbols maps into an Abstraction relationship, with the «realize» stereotype, between the Classifiers corresponding to the two symbols. Realization is usually used between a class and an interface, but may also be used between any two classifiers to show conformance of behavior.

## 3.29 Interfaces

### 3.29.1 Semantics

An interface is a specifier for the externally-visible operations of a class, component, or other classifier (including subsystems) without specification of internal structure. Each interface often specifies only a limited part of the behavior of an actual class. Interfaces do not have implementation. They lack attributes, states, or associations; they only have operations. (An interface may be the target of a one-way association,

however, but it may not have an association that it can navigate.) Interfaces may have generalization relationships. An interface is formally equivalent to an abstract class with no attributes and no methods and only abstract operations, but Interface is a peer of Class within the UML metamodel (both are Classifiers).

## 3.29.2 Notation

An interface is a Classifier and may be shown using the full rectangle symbol with compartments and the keyword «interface». A list of operations supported by the interface is placed in the operation compartment. The attribute compartment may be omitted because it is always empty.

An interface may also be displayed as a small circle with the name of the interface placed below the symbol. The circle may be attached by a solid line to classifiers that support it. This indicates that the class provides all of the operations in the interface type (and possibly more). The operations provided are not shown on the circle notation; use the full rectangle symbol to show the list of operations. A class that uses or requires the operations supplied by the interface may be attached to the circle by a dashed arrow pointing to the circle. The dashed arrow implies that the class requires no more than the operations specified in the interface; the client class is not required to actually use *all* of the interface operations.

The Realization relationship from a classifier to an interface that it supports is shown by a dashed line with a solid triangular arrowhead (a “dashed generalization symbol”). This is the same notation used to indicate realization of a type by an implementation class. In fact, this symbol can be used between any two classifier symbols, with the meaning that the client (the one at the tail of the arrow) supports at least all of the operations defined in the supplier (the one at the head of the arrow), but with no necessity to support any of the data structure of the supplier (attributes and associations).

## 3.29.3 Example

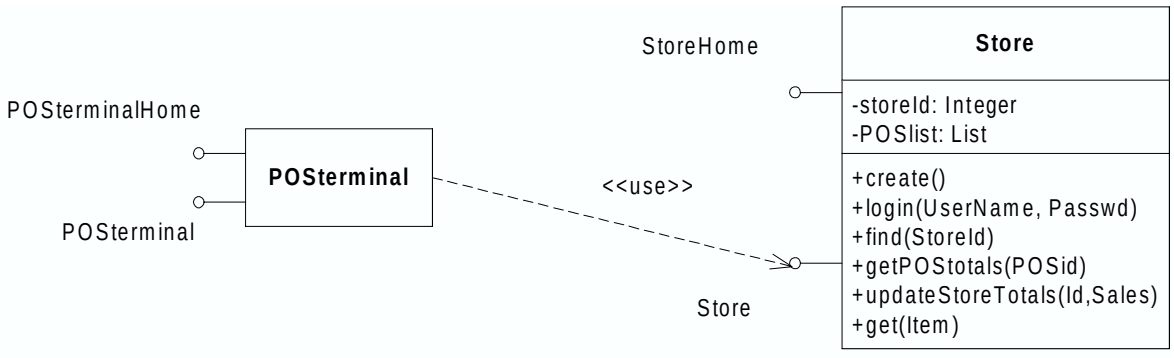


*Figure 3-28* Shorthand Version of Interface Notation

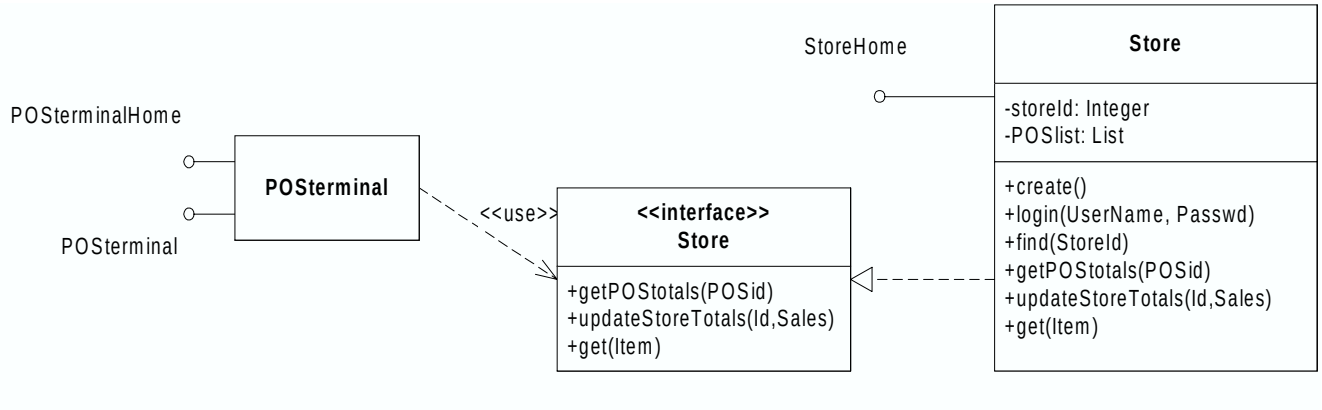


*Figure 3-29* Longhand Version of Interface Notation

### 3.29.4 Mapping

A class rectangle symbol with stereotype «interface», or a circle on a class diagram, maps into an Interface element with the name given by the symbol. The operation list of a rectangle symbol maps into the list of Operation elements of the Interface.

A dashed generalization arrow from a class symbol to an interface symbol, or a solid line connecting a class symbol and an interface circle, maps into an Abstraction dependency with the «realize» stereotype between the corresponding Classifier and Interface elements. A dependency arrow from a class symbol to an interface symbol maps into a Usage dependency between the corresponding Classifier and Interface.

## 3.30 Parameterized Class (Template)

### 3.30.1 Semantics

A template is the descriptor for a class with one or more unbound formal parameters. It defines a family of classes, each class specified by binding the parameters to actual values. Typically, the parameters represent attribute types; however, they can also represent integers, other types, or even operations. Attributes and operations within the template are defined in terms of the formal parameters so they too become bound when the template itself is bound to actual values.

A template is not a directly usable class because it has unbound parameters. Its parameters must be bound to actual values to create a bound form that is a class. Only a class can be a superclass or the target of an association (a one-way association *from* the template *to* another class is permissible, however). A template may be a subclass of an ordinary class. This implies that all classes formed by binding it are subclasses of the given superclass.

Parameterization can be applied to other ModelElements, such as Collaborations or even entire Packages. The description given here for classes applies to other kinds of modeling elements in the obvious way.

### 3.30.2 Notation

A small dashed rectangle is superimposed on the upper right-hand corner of the rectangle for the class (or to the symbol for another modeling element). The dashed rectangle contains a parameter list of formal parameters for the class and their implementation types. The list must not be empty, although it might be suppressed in the presentation. The name, attributes, and operations of the parameterized class appear as normal in the class rectangle; however, they may also include occurrences of the formal parameters. Occurrences of the formal parameters can also occur inside of a context for the class, for example, to show a related class identified by one of the parameters.

### 3.30.3 Presentation Options

The parameter list may be comma-separated or it may be one per line.

Parameters are restricted attributes, shown as strings with the syntax:

*name* : *type* = *default-value*

- Where *name* is an identifier for the parameter with scope inside the template.
- Where *type* is a string designating a *Classifier* for the parameter. If it is a simple word, it must be the name of a Classifier. Otherwise it is a programming-language dependent string that maps into a ProgrammingLanguageDataType according to the programming language (if any) for the diagram context or specified in a support tool.
- Where *default-value* is a string designating an Expression for a default value that is used when the corresponding argument is omitted in a Binding. The equal sign and expression may be omitted, in which case there is no default value and the argument must be supplied in a Binding.

If the type name is omitted, the parameter type is assumed to be Classifier. The value supplied for an argument in a Binding must be the name of a Classifier (including a class or a data type). Other parameter types (such as Integer) must be explicitly shown. The value supplied for an argument in a Binding must be an actual instance value of the given kind.

### 3.30.4 Example

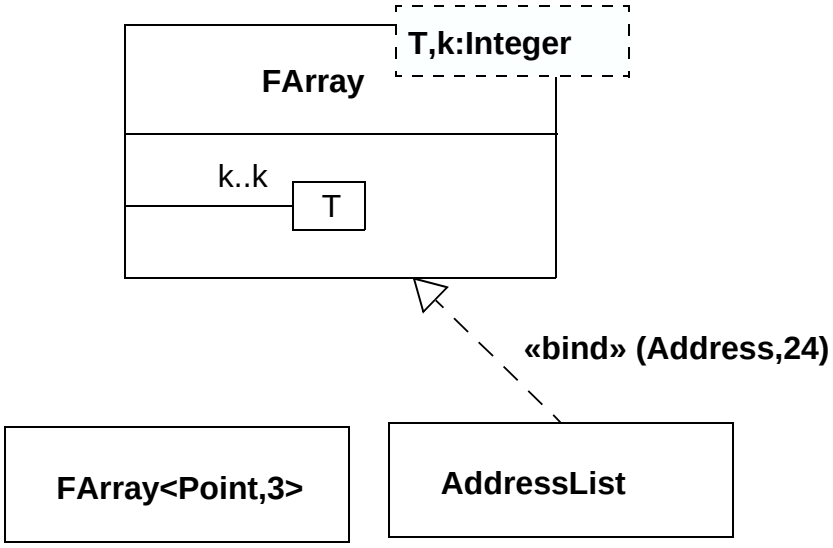


*Figure 3-30* Template Notation with Use of Parameter as a Reference

### 3.30.5 Mapping

The addition of the template dashed box to a symbol causes the addition of the parameter names in the list as ModelElements within the Namespace of the ModelElement corresponding to the base symbol (or to the Namespace containing a ModelElement that is not itself a Namespace). Each of the parameter ModelElements has the templateParameter association to the base ModelElement.

## 3.31 Bound Element

### 3.31.1 Semantics

A template cannot be used directly in an ordinary relationship such as generalization or association, because it has a free parameter that is not meaningful outside of a scope that declares the parameter. To be used, a template's parameters must be *bound* to actual values. The actual value for each parameter is an expression defined within the scope of use. If the referencing scope is itself a template, then the parameters of the referencing template can be used as actual values in binding the referenced template. The parameter names in the two templates cannot be assumed to correspond because they have no scope outside of their respective templates.

### 3.31.2 Notation

A bound element is indicated by a text syntax in the name string of an element, as follows:

*Template-name* '<' *value-list* '>'

- Where *value-list* is a comma-delimited non-empty list of value expressions.
- Where *Template-name* is identical to the name of a template.

For example, VArray<Point,3> designates a class described by the template Varray.

The number and type of values must match the number and type of the template parameters for the template of the given name.

The bound element name may be used anywhere that an element name of the parameterized kind could be used. For example, a bound class name could be used within a class symbol on a class diagram, as an attribute type, or as part of an operation signature.

Note that a bound element is fully specified by its template; therefore, its content may not be extended. Declaration of new attributes or operations for classes is not permitted, for example, but a bound class could be subclassed and the subclass extended in the usual way.

The relationship between the bound element and its template alternatively may be shown by a Dependency relationship with the keyword «bind». The arguments are shown in parentheses after the keyword. In this case, the bound form may be given a name distinct from the template.

### 3.31.3 Style Guidelines

The attribute and operation compartments are normally suppressed within a bound class, because they must not be modified in a bound template.

### 3.31.4 Example

See Figure 3-30 on page 3-54.

### 3.31.5 Mapping

The use of the bound element syntax for the name of a symbol maps into a Binding dependency between the dependent ModelElement (such as Class) corresponding to the bound element symbol and the provider ModelElement (again, such as Class) whose name matches the name part of the bound element without the arguments. If the name does not match a template element or if the number of arguments in the bound element does not match the number of parameters in the template, then the model is ill formed. Each argument position in the bound element maps into a TemplateArgument bearing a binding link to the Binding dependency and a modelElement link to the

ModelElement that is implicitly substituted for the template parameter in the corresponding position in the template definition. An explicitly drawn «bind» dependency symbol maps to a Binding dependency with arguments as described above.

## 3.32 Utility

A utility is a grouping of global variables and procedures in the form of a class declaration. This is not a fundamental construct, but a programming convenience. The attributes and operations of the utility become global variables and procedures. A utility is modeled as a stereotype of a classifier.

### 3.32.1 Semantics

The instance-scope attributes and operations of a utility are interpreted as global attributes and operations. It is inappropriate for a utility to declare class-scope attributes and operations because the instance-scope members are already interpreted as being at class scope.

### 3.32.2 Notation

A utility is shown as the stereotype «utility» of Class. It may have both attributes and operations, all of which are treated as global attributes and operations.

### 3.32.3 Example

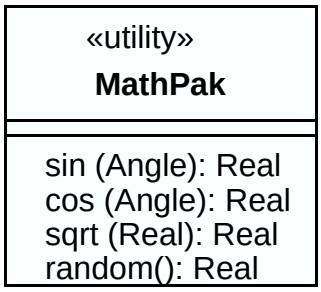


*Figure 3-31* Notation for Utility

### 3.32.4 Mapping

This is not a special symbol. It simply maps into a Class element with the «utility» stereotype.

## 3.33 Metaclass

### 3.33.1 Semantics

A metaclass is a class whose instances are classes.

### 3.33.2 Notation

A metaclass is shown as the stereotype «metaclass» of Class.

### 3.33.3 Mapping

This is not a special symbol. It simply maps into a Class element with the «metaclass» stereotype.

## 3.34 Enumeration

### 3.34.1 Semantics

An Enumeration is a user-defined data type whose instances are a set of user-specified named enumeration literals. The literals have a relative order but no algebra is defined on them.

### 3.34.2 Notation

An Enumeration is shown using the Classifier notation (a rectangle) with the keyword «enumeration». The name of the Enumeration is placed in the upper compartment. An ordered list of enumeration literals may be placed, one to a line, in the middle compartment. Operations defined on the literals may be placed in the lower compartment. The lower and middle compartments may be suppressed.

### 3.34.3 Mapping

Maps into an Enumeration with the given list of enumeration literals.

## 3.35 Stereotype Declaration

### 3.35.1 Semantics

A Stereotype is a user-defined metaelement whose structure matches an existing UML metaelement (its “base class”). Because it is user defined, a stereotype declaration is an element that appears at the “model” layer of the UML four-layer metamodeling hierarchy although it conceptually belongs in the layer above, the metamodel layer.

### 3.35.2 Notation

Because stereotypes span two different metamodeling layers, a special notation is required to clearly indicate the crossover between the two layers. Specifically, it is necessary to show how a model-level element (the stereotype) relates to its metaelement (its UML base class). This is denoted using a special stereotype of Dependency called «stereotype» as shown in Figure 3-32 on page 3-59.

The Stereotype itself is shown using the Classifier notation (a rectangle) with the keyword «stereotype» (Figure 3-32). The name of the Stereotype is placed in the upper compartment. Constraints on elements described by the stereotype may be placed in a named compartment called **Constraints**. Required tags may be placed in a named compartment called **Tags**. Individual items (tags) in the list are defined according to the following format:

```
tagDefinitionName : String [multiplicity]
```

where `string` can be either a string matching the name of a data type representing the type of the values of the tag, or it is a reference to a metaclass or a stereotype. In the latter case, the string has the form:

«metaclass» metaclassName

or

«stereotype» stereotypeName

where `metaclassName` is the name of the referenced metaclass and is the name of the references stereotype. The multiplicity element is optional and conforms to standard rules for specifying multiplicities. In case of a range specification, a lower bound of zero indicates an optional tag.

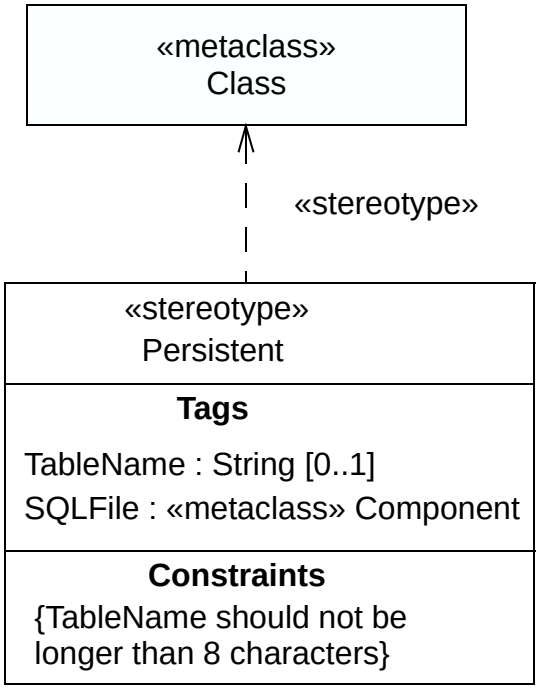


*Figure 3-32* Notational form for declaring a stereotype

In the example diagram in Figure 3-32, the stereotype Persistent is a stereotype of the UML metaelement Class. TableName is an optional tag whose type is a model type called String while SQLFile is a reference to an instance of Component in the model.

An icon can be defined for the stereotype, but its graphical definition is outside the scope of UML and must be handled by an editing tool.

An alternative and usually more compact way of specifying stereotypes and tags using tables is shown in Figure 3-33 and Figure 3-34, respectively.

| Stereotype | Base Class | Parent | Tags | Constraints | Description |
|---|---|---|---|---|---|
| Architectural Element | Generalizable Element | N/A | N/A | N/A | A generic stereotype that is the parent of all other stereotypes used for architectural modeling . |
| Capsule | Class | Architectural Element | isDynamic | self.isActive = true | Indicates a class that is used to model the structural components of an architecture specification. |

*Figure 3-33* Tabular form for specifying stereotypes

| Tag | Stereotype | Type | Multiplicity | Description |
|---|---|---|---|---|
| isDynamic | Capsule | UML::Datatypes::Boolean | 1 | Used to identify if the associated capsule class may be created and destroyed dynamically. |

*Figure 3-34* Tabular form for specifying tags

Each row of the stereotype specification table in Figure 3-33 defines one stereotype and each row in the tag specification table in Figure 3-34 contains one tag definition.

The columns of the stereotype specification table are defined as follows:

- *Stereotype* - the name of the stereotype.
- *Base Class* - the UML metamodel element that serves as the base for the stereotype.
- *Parent* - the direct parent of the stereotype being defined (NB: if one exists, otherwise the symbol “N/A” is used).
- *Tags* - a list of all tags of the tagged values that may be associated with this stereotype (or N/A if none are defined).
- *Constraints* - a list of constraints associated with the stereotype.
- *Description* - an informal description with possible explanatory comments.

The columns of the tag specification table are defined as follows:

- *Tag* - the name of the tag.
- *Stereotype* - the name of the stereotype that owns this tag, or “N/A” if it is a stand alone tag.
- *Type* - the name of the type of the values that can be associated with the tag.
- *Multiplicity* - the maximum number of values that may be associated with one tag instance.
- *Description* - an informal description with possible explanatory comments.

In the case of both the stereotype specification table and the tag specification table, columns that are not applicable may be omitted.

In the example stereotype specification table of Figure 3-34, two related stereotypes are defined. The first row declares the stereotype ArchitecturalElement, which is a stereotype of GeneralizableElement, while the second row declares the stereotype Capsule, which is a specialization of the ArchitecturalElement stereotype, but which applies only to instances of Class, which is a subclass of GeneralizableElement in the metamodel.

The equivalent declaration as the one table in Figure 3-34, less the constraints and the informal descriptions, is shown graphically in Figure 3-35.

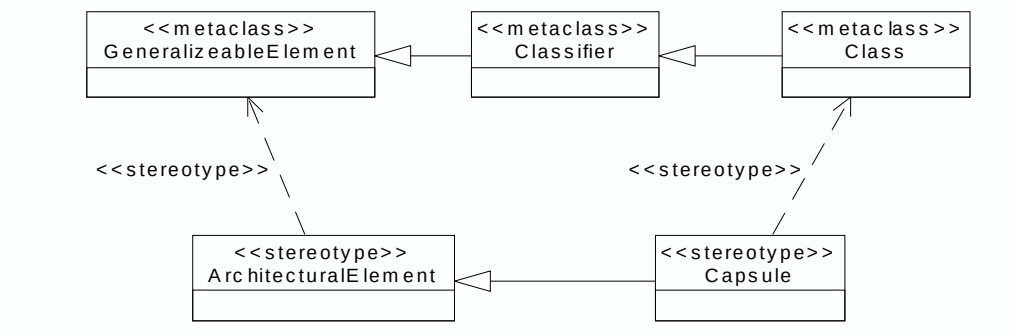


*Figure 3-35* Graphical equivalent of the stereotype declarations shown in Figure 3-34

### 3.35.3 Mapping

A classifier with a stereotype «metaclass» maps into a UML metaelement and a classifier with a stereotype «stereotype» maps into a Stereotype. The «stereotype» dependency maps to the baseClass attribute definition of the stereotype. The constraints listed in the **Constraints** compartment map to stereotype constraints and the items in the **Tags** compartment map to the defined tags of the stereotype. Each item in the **Tags** list maps to a TagDefinition. The string before the colon separator maps to the name of the tag definition while the string following the colon maps to an instance of Name. If a multiplicity specification is included in the item, it maps to the multiplicity attribute of the tag definition.

## 3.36 Powertype

### 3.36.1 Semantics

A Powertype is a user-defined metaelement whose instances are classes in the model.

### 3.36.2 Notation

A Powertype is shown using the Classifier notation (a rectangle) with the stereotype keyword «powertype». The name of the Powertype is placed in the upper compartment. Because the elements are ordinary classes, attributes and operations on the powertype are usually not defined by the user.

The instances of the powertype may be indicated by placing a dashed line across the parent lines of the classes with the syntax

```
discriminatorName: powertypeName,
```

where the powertype name on the line implicitly defines a powertype if one is not explicitly defined.

### 3.36.3 Mapping

Maps into a Class with the «powertype» stereotype with the given classes as instances.

## 3.37 Class Pathnames

### 3.37.1 Notation

Class symbols (rectangles) serve to define a class and its properties, such as relationships to other classes. A reference to a class in a different package is notated by using a pathname for the class, in the form:

*package-name* :: *class-name*

References to classes also appear in text expressions, most notably in type specifications for attributes and variables. In these places a reference to a class is indicated by simply including the name of the class itself, including a possible package name, subject to the syntax rules of the expression.

### 3.37.2 Example

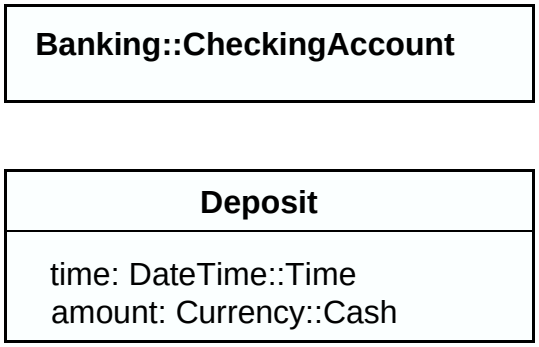


*Figure 3-36* Pathnames for Classes in Other Packages

### 3.37.3 Mapping

A class symbol whose name string is a pathname represents a reference to the Class with the given name inside the package with the given name. The name is assumed to be defined in the target package; otherwise, the model is ill formed. A Relationship from a symbol in the current package; that is, the package containing the diagram and its mapped elements to a symbol in another package is part of the current package.

## 3.38 Accessing or Importing a Package

### 3.38.1 Semantics

An element may reference an element contained in a different package. On the package level, the «access» dependency indicates that the contents of the target package may be referenced by the client package or packages recursively embedded within it. The target references must have visibility sufficient for the referents: public visibility for an unrelated package, public or protected visibility for a descendant of the target package, or any visibility for a package nested inside the target package (an access dependency is not required for the latter case). A package nested inside the package making the access gets the same access.

Note that an access dependency does not modify the namespace of the client or in any other way automatically create references; it merely grants permission to establish references. Note also that a tool could automatically create access dependencies for users if desired when references are created.

An import dependency grants access and also loads the names with appropriate visibility in the target namespace into the accessing package; that is, a pathname is not necessary to reference them. Such names are not automatically re-exported; however; a name must be explicitly re-exported (and may be given a new name and visibility at the same time).

### 3.38.2 Notation

The access dependency is displayed as a dependency arrow from the referencing (client) package to the target (supplier) package containing the target of the references. The arrow has the stereotype keyword «access». This dependency indicates that elements within the client package may legally reference elements within the supplier. The references must also satisfy visibility constraints specified by the supplier. Note that the dependency does not automatically create any references. It merely grants permission for them to be established.

The import dependency has the same notation as the access dependency except it has the stereotype keyword «import».

### 3.38.3 Example

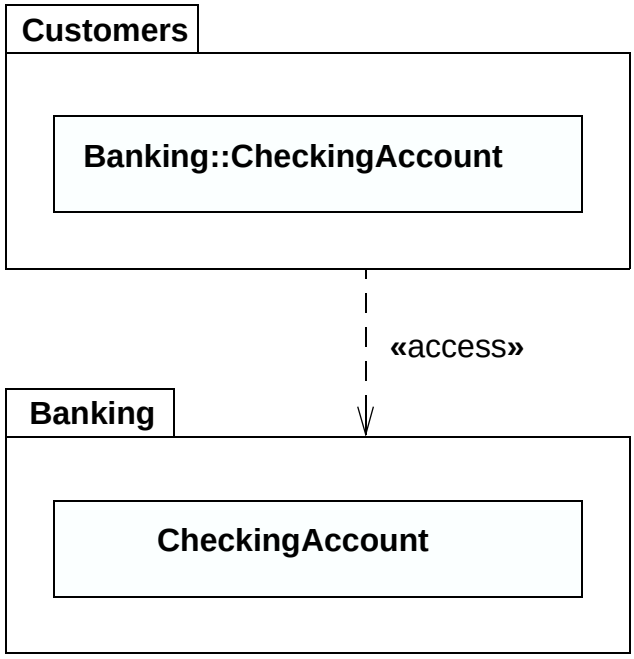


*Figure 3-37* Access Dependency Among Packages

### 3.38.4 Mapping

This is not a special symbol. It maps into a Permission dependency with the stereotype «access» or «import» between the two packages.

## 3.39 Object

### 3.39.1 Semantics

An object represents a particular instance of a class. It has identity and attribute values. A similar notation also represents a role within a collaboration because roles have instance-like characteristics.

### 3.39.2 Notation

The object notation is derived from the class notation by underlining instance-level elements, as explained in the general comments in Section 3.12, “Type-Instance Correspondence,” on page 3-14.

An object shown as a rectangle with two compartments.

The top compartment shows the name of the object and its class, all underlined, using the syntax:

*objectname* : *classname*

The classname can include a full pathname of enclosing package, if necessary. The package names precede the classname and are separated by double colons. For example:

```
display_window: WindowingSystem::GraphicWindows::Window
```

A stereotype for the class may be shown textually (in guillemets above the name string) or as an icon in the upper right corner. The stereotype for an object must match the stereotype for its class.

To show multiple classes that the object is an instance of, use a comma-separated list of classnames. These classnames must be legal for multiple classification; that is, only one implementation class permitted, but multiple types permitted.

To show the presence of an object in a particular state of a class, use the syntax:

*objectname* : *classname* '[' *statename-list* ']'

The list must be a comma-separated list of names of states that can legally occur concurrently.

The second compartment shows the attributes for the object and their values as a list. Each value line has the syntax:

*attributename* : *type* = *value*

The type is redundant with the attribute declaration in the class and may be omitted.

The value is specified as a literal value. UML does not specify the syntax for literal value expressions; however, it is expected that a tool will specify such a syntax using some programming language.

The flow relationship between two values of the same object over time can be shown by connecting two object symbols by a dashed arrow with the keyword «become». If the flow arrow is on a collaboration diagram, the label may also include a sequence number to show when the value changes. Similarly, the keyword «copy» can be used to show the creation of one object from another object value.

### 3.39.3 Presentation Options

The name of the object may be omitted. In this case, the colon should be kept with the class name. This represents an anonymous object of the given class given identity by its relationships.

The class of the object may be suppressed (together with the colon).

The attribute value compartment as a whole may be suppressed.

Attributes whose values are not of interest may be suppressed.

Attributes whose values change during a computation may show their values as a list of values held over time. In an interactive tool, they might even change dynamically. An alternate notation is to show the same object more than once with a «becomes» relationship between them.

### 3.39.4 Style Guidelines

Objects may be shown on class diagrams. The elements on collaboration diagrams are not objects, because they describe many possible objects. They are instead roles that may be held by object. Objects in class diagrams serve mainly to show examples of data structures.

### 3.39.5 Variations

For a language such as *Self* in which operations can be attached to individual objects at run time, a third compartment containing operations would be appropriate as a language-specific extension.

### 3.39.6 Example

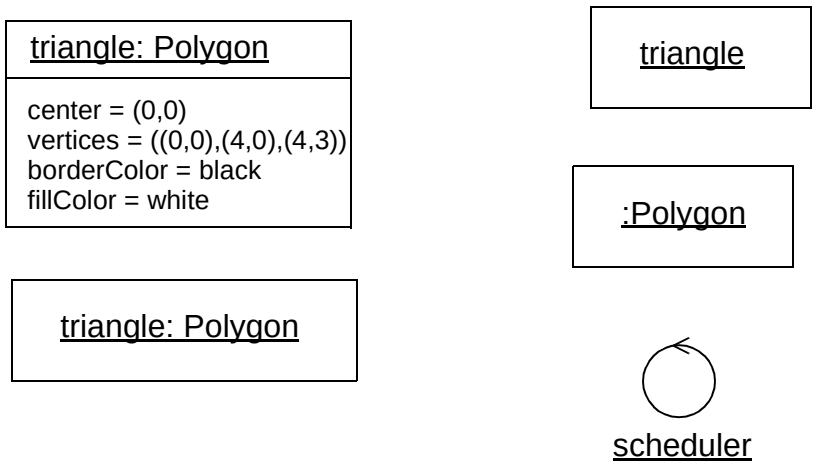


*Figure 3-38* Objects

### 3.39.7 Mapping

In an object diagram, or within an ordinary class diagram, an object symbol maps into an Object of the Class (or Classes) given by the *classname* part of the name string. The attribute list in the symbol maps into a set of AttributeLinks attached to the Object, with values given by the value expressions in the attribute list in the symbol. If a list of states in brackets follows the class name, then this maps into a ClassifierInState with the named Class as its type and the named States as the states. The ClassfierInState classifies the Object.

## 3.40 Composite Object

### 3.40.1 Semantics

A composite object represents a high-level object made of tightly-bound parts. This is an instance of a composite class, which implies the composition aggregation between the class and its parts. A composite object is similar to (but simpler and more restricted than) a collaboration; however, it is defined completely by composition in a static model. See Section 3.48, “Composition,” on page 3-81.

### 3.40.2 Notation

A composite object is shown as an object symbol. The name string of the composite object is placed in a compartment near the top of the rectangle (as with any object). The lower compartment holds the parts of the composite object instead of a list of attribute values. (However, even a list of attribute values may be regarded as the parts of a composite object, so there is not a great difference.) It is possible for some of the parts to be composite objects with further nesting.

### 3.40.3 Example

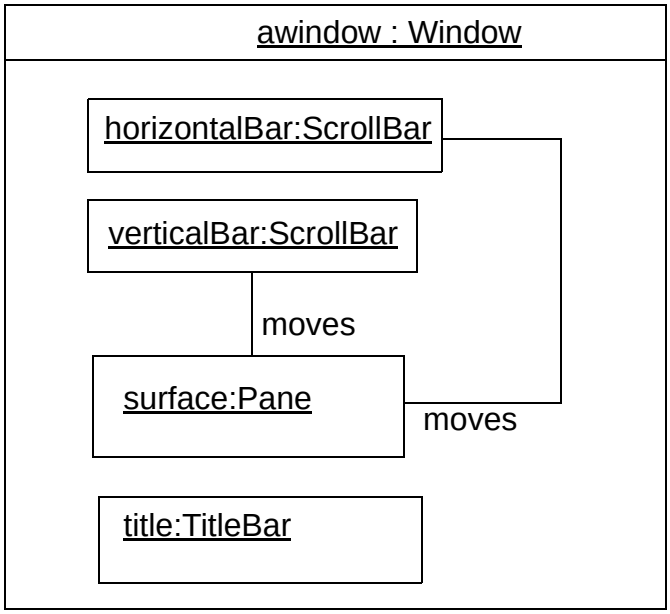


*Figure 3-39* Composite Objects

### 3.40.4 Mapping

A composite object symbol maps into an Object of the given Class with composition links to each of the Objects and Links corresponding to the class box symbols and to association path symbols directly contained within the boundary of the composite object symbol (and not contained within another deeper boundary).

## 3.41 Association

Binary associations are shown as lines connecting two classifier symbols. The lines may have a variety of adornments to show their properties. Ternary and higher-order associations are shown as diamonds connected to class symbols by lines.

## 3.42 Binary Association

### 3.42.1 Semantics

A binary association is an association among exactly two classifiers (including the possibility of an association from a classifier to itself).

### 3.42.2 Notation

A binary association is drawn as a solid path connecting two classifier symbols (both ends may be connected to the same classifier, but the two ends are distinct). The path may consist of one or more connected segments. The individual segments have no semantic significance, but may be graphically meaningful to a tool in dragging or resizing an association symbol. A connected sequence of segments is called a *path*.

In a binary association, both ends may attach to the same classifier. The links of such an association may connect two different instances from the same classifier or one instance to itself. The latter case may be forbidden by a constraint if necessary.

The end of an association where it connects to a classifier is called an *association end*. Most of the interesting information about an association is attached to its ends.

The path may also have graphical adornments attached to the main part of the path itself. These adornments indicate properties of the entire association. They may be dragged along a segment or across segments, but must remain attached to the path. It is a tool responsibility to determine how close association adornments may approach an end so that confusion does not occur. The following kinds of adornments may be attached to a path.

#### 3.42.2.1 association name

Designates the (optional) name of the association.

It is shown as a name string near the path (but not near enough to an end to be confused with a rolename). The name string may have an optional small black solid triangle in it. The point of the triangle indicates the direction in which to read the name. The name-direction arrow has no semantics significance, it is purely descriptive. The classifiers in the association are ordered as indicated by the name-direction arrow.

---

**Note –** There is no need for a *name direction* property on the association model; the ordering of the classifiers within the association *is* the name direction. This convention works even with n-ary associations.

---

A stereotype keyword within guillemets may be placed above or in front of the association name. A property string may be placed after or below the association name.

#### 3.42.2.2 association class symbol

Designates an association that has class-like properties, such as attributes, operations, and other associations. This is present if, and only if, the association is an association class. It is shown as a class symbol attached to the association path by a dashed line.

The association path and the association class symbol represent the same underlying model element, which has a single name. The name may be placed on the path, in the class symbol, or on both (but they must be the same name).

Logically, the association class and the association are the same semantic entity; however, they are graphically distinct. The association class symbol can be dragged away from the line, but the dashed line must remain attached to both the path and the class symbol.

### 3.42.3 Presentation Options

When two paths cross, the crossing may optionally be shown with a small semicircular jog to indicate that the paths do not intersect (as in electrical circuit diagrams).

### 3.42.4 Style Guidelines

Lines may be drawn using various styles, including orthogonal segments, oblique segments, and curved segments. The choice of a particular set of line styles is a user choice.

### 3.42.5 Options

#### 3.42.5.1 Xor-association

An xor-constraint indicates a situation in which only one of several potential associations may be instantiated at one time for any single instance. This is shown as a dashed line connecting two or more associations, all of which must have a classifier in

common, with the constraint string “{xor}” labeling the dashed line. Any instance of the classifier may only participate in one of the associations at one time. Each rolename must be different. (This is simply a predefined use of the constraint notation.)

### 3.42.6 Example

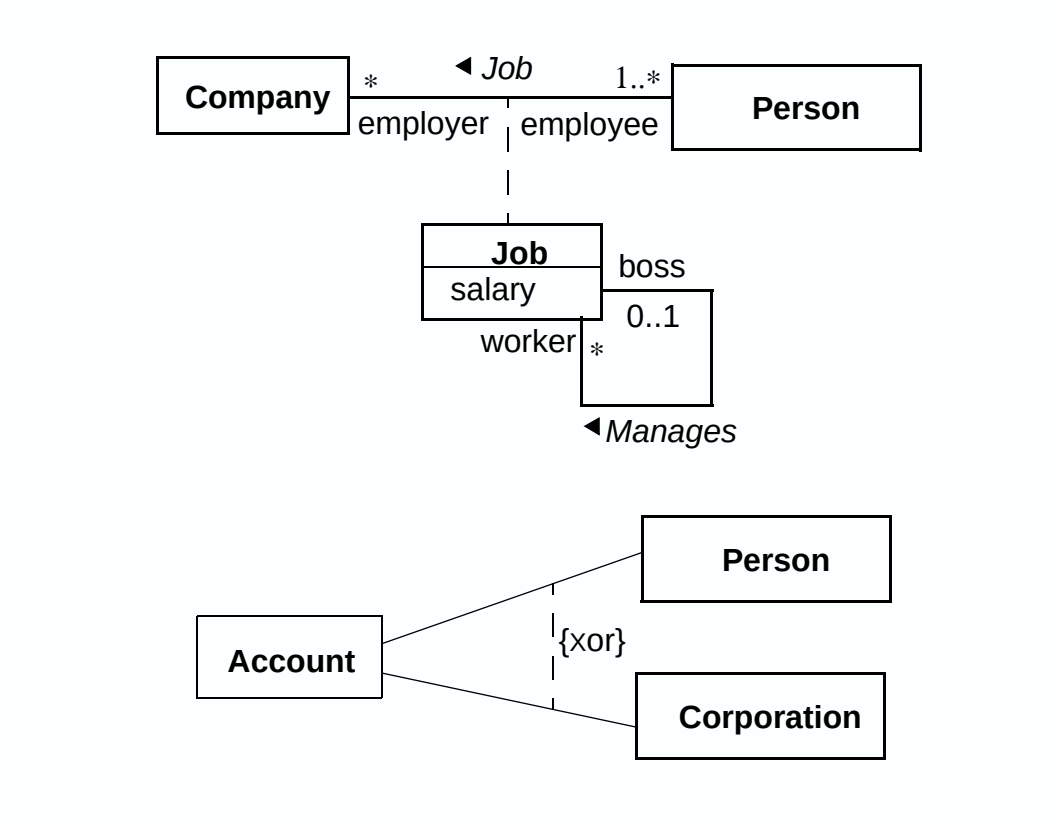


*Figure 3-40* Association Notation

### 3.42.7 Mapping

An association path connecting two class symbols maps to an Association between the corresponding Classifiers. If there is an arrow on the association name, then the Class corresponding to the tail of the arrow is the first class and the Classifier corresponding to the head of the arrow is the second Classifier in the ordering of ends of the Association; otherwise, the ordering of ends in the association is undetermined. The adornments on the path map into properties of the Association as described above. The Association is owned by the package containing the diagram.

## 3.43 Association End

### 3.43.1 Semantics

An association end is simply an end of an association where it connects to a classifier. It is part of the association, not part of the classifier. Each association has two or more ends. Most of the interesting details about an association are attached to its ends. An association end is not a separable element, it is just a mechanical part of an association.

### 3.43.2 Notation

The path may have graphical adornments at each end where the path connects to the classifier symbol. These adornments indicate properties of the association related to the classifier. The adornments are part of the association symbol, not part of the classifier symbol. The end adornments are either attached to the end of the line, or near the end of the line, and must drag with it. The following kinds of adornments may be attached to an association end.

#### 3.43.2.1 multiplicity

Specified by a text syntax. Multiplicity may be suppressed on a particular association or for an entire diagram. In an incomplete model the multiplicity may be unspecified in the model itself. In this case, it must be suppressed in the notation. See Section 3.44, “Multiplicity,” on page 3-75.

#### 3.43.2.2 ordering

If the multiplicity is greater than one, then the set of related elements can be ordered or unordered. If no indication is given, then it is unordered (the elements form a set). Various kinds of ordering can be specified as a constraint on the association end. The declaration does not specify how the ordering is established or maintained. Operations that insert new elements must make provision for specifying their position either implicitly (such as at the end) or explicitly. Possible values include:

- unordered - the elements form an unordered set. This is the default and need not be shown explicitly.
- ordered - the elements of the set have an ordering, but duplicates are still prohibited. This generic specification includes all kinds of ordering. This may be specified by the keyword syntax “{ordered}.”

An ordered relationship may be implemented in various ways; however, this is normally specified as a language-specified code generation property to select a particular implementation. An implementation extension might substitute the data structure to hold the elements for the generic specification “ordered.”

At implementation level, sorting may also be specified. It does not add new semantic information, but it expresses a design decision:

- sorted - the elements are sorted based on their internal values. The actual sorting rule is best specified as a separate constraint.

#### 3.43.2.3 *qualifier*

A qualifier is optional, but not suppressible. See Section 3.45, “Qualifier,” on page 3-76.

#### 3.43.2.4 *navigability*

An arrow may be attached to the end of the path to indicate that navigation is supported toward the classifier attached to the arrow. Arrows may be attached to zero, one, or two ends of the path. To be totally explicit, arrows may be shown whenever navigation is supported in a given direction. In practice, it is often convenient to suppress some of the arrows and just show exceptional situations. See Section 3.22.3, “Presentation Options,” on page 3-36 for details.

#### 3.43.2.5 *aggregation indicator*

A hollow diamond is attached to the end of the path to indicate aggregation. The diamond may not be attached to both ends of a line, but it need not be present at all. The diamond is attached to the class that is the aggregate. The aggregation is optional, but not suppressible.

If the diamond is filled, then it signifies the strong form of aggregation known as *composition*. See Section 3.48, “Composition,” on page 3-81.

#### 3.43.2.6 *rolename*

A name string near the end of the path. It indicates the role played by the class attached to the end of the path near the rolename. The rolename is optional, but not suppressible.

#### 3.43.2.7 *interface specifier*

The name of a Classifier with the syntax:

‘:’ *classifiername, . . .*

It indicates the behavior expected of an associated object by the related instance. In other words, the interface specifier specifies the behavior required to enable the association. In this case, the actual classifier usually provides more functionality than required for the particular association (since it may have other responsibilities).

The use of a rolename and interface specifier are equivalent to creating a small collaboration that includes just an association and two roles, whose structure is defined by the rolename and attached classifier on the original association. Therefore, the

original association and classifiers are a use of the collaboration. The original classifier must be compatible with the interface specifier (which can be an interface or a type, among other kinds of classifiers).

If an interface specifier is omitted, then the association may be used to obtain full access to the associated class.

#### 3.43.2.8 changeability

If the links are changeable (can be added, deleted, and moved), then no indicator is needed. The property {frozen} indicates that no links may be added, deleted, or moved from an object (toward the end with the adornment) after the object is created and initialized. The property {addOnly} indicates that additional links may be added (presumably, the multiplicity is variable); however, links may not be modified or deleted.

#### 3.43.2.9 visibility

Specified by a visibility indicator ('+', '#', '-' or explicit property name such as {public}) in front of the rolename. Specifies the visibility of the association traversing in the direction toward the given rolename. See Section 3.25, "Attribute," on page3-41 for details of visibility specification.

Other properties can be specified for association ends, but there is no graphical syntax for them. To specify such properties, use the constraint syntax near the end of the association path (a text string in braces). Examples of other properties include mutability.

### 3.43.3 Presentation Options

If there are two or more aggregations to the same aggregate, they may be drawn as a tree by merging the aggregation end into a single segment. This requires that all of the adornments on the aggregation ends be consistent. This is purely a presentation option, there are no additional semantics to it.

Various options are possible for showing the navigation arrows on a diagram. These can vary from time to time by user request or from diagram to diagram.

- Presentation option 1: Show all arrows. The absence of an arrow indicates navigation is not supported.
- Presentation option 2: Suppress all arrows. No inference can be drawn about navigation. This is similar to any situation in which information is suppressed from a view.
- Presentation option 3: Suppress arrows for associations with navigability in both directions, show arrows only for associations with one-way navigability. In this case, the two-way navigability cannot be distinguished from no-way navigation; however, the latter case is normally rare or nonexistent in practice. This is yet another example of a situation in which some information is suppressed from a view.

### 3.43.4 Style Guidelines

If there are multiple adornments on a single association end, they are presented in the following order, reading from the end of the path attached to the classifier toward the bulk of the path:

- qualifier
- aggregation symbol
- navigation arrow

Rolenames and multiplicity should be placed near the end of the path so that they are not confused with a different association. They may be placed on either side of the line. It is tempting to specify that they will always be placed on a given side of the line (clockwise or counterclockwise), but this is sometimes overridden by the need for clarity in a crowded layout. A rolename and a multiplicity may be placed on opposite sides of the same association end, or they may be placed together (for example, "* employee").

### 3.43.5 Example

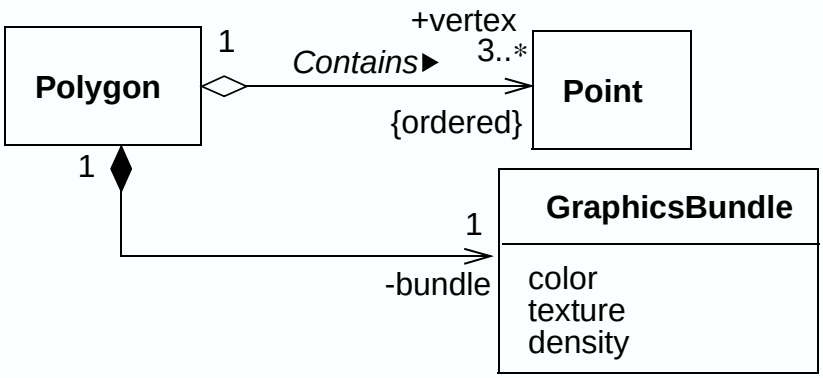


*Figure 3-41* Various Adornments on Association Roles

### 3.43.6 Mapping

The adornments on the end of an association path map into properties of the corresponding role of the Association. In general, implications cannot be drawn from the absence of an adornment (it may simply be suppressed) but see the preceding descriptions for details. The interface specifier maps into the "specification" rolename in the AssociationEnd-Classifier association.

## 3.44 Multiplicity

### 3.44.1 Semantics

A multiplicity item specifies the range of allowable cardinalities that a set may assume. Multiplicity specifications may be given for roles within associations, parts within composites, repetitions, and other purposes. Essentially a multiplicity specification is a subset of the open set of nonnegative integers.

### 3.44.2 Notation

A multiplicity specification is shown as a text string comprising a comma-separated sequence of integer intervals, where an interval represents a (possibly infinite) range of integers, in the format:

*lower-bound .. upper-bound*

where *lower-bound* and *upper-bound* are literal integer values, specifying the closed (inclusive) range of integers from the lower bound to the upper bound. In addition, the star character (*) may be used for the upper bound, denoting an unlimited upper bound. In a parameterized context (such as a template), the bounds could be expressions but they must evaluate to literal integer values for any actual use. Unbound expressions that do not evaluate to literal integer values are not permitted.

If a single integer value is specified, then the integer range contains the single integer value.

If the multiplicity specification comprises a single star (*), then it denotes the unlimited nonnegative integer range, that is, it is equivalent to 0..* (zero or more).

A multiplicity of 0..0 is meaningless as it would indicate that no instances can occur.

Expressions in some specification language can be used for multiplicities, but they must resolve to fixed integer ranges within the model; that is, no dynamic evaluation of expressions, essentially the same rule on literal values as most programming languages.

### 3.44.3 Style Guidelines

Preferably, intervals should be monotonically increasing. For example, “1..3,7,10” is preferable to “7,10,1..3”.

Two contiguous intervals should be combined into a single interval. For example, “0..1” is preferable to “0,1”.

### 3.44.4 Example

0..1

1

0..*

*

1..*

1..6

1..3,7..10,15,19..*

### 3.44.5 Mapping

A multiplicity string maps into a Multiplicity value with one or more MultiplicityRanges. Duplications or other nonstandard presentation of the string itself have no effect on the mapping. Note that Multiplicity is a value and not an object. It cannot stand on its own, but is the value of some element property.

## 3.45 Qualifier

### 3.45.1 Semantics

A qualifier is an attribute or list of attributes whose values serve to partition the set of instances associated with an instance across an association. The qualifiers are attributes of the association.

### 3.45.2 Notation

A qualifier is shown as a small rectangle attached to the end of an association path between the final path segment and the symbol of the classifier that it connects to. The qualifier rectangle is part of the association path, not part of the classifier. The qualifier rectangle drags with the path segments. The qualifier is attached to the source end of the association. An instance of the source classifier, together with a value of the qualifier, uniquely select a partition in the set of target classifier instances on the other end of the association; that is, every target falls into exactly one partition.

The multiplicity attached to the target end denotes the possible cardinalities of the set of target instances selected by the pairing of a source instance and a qualifier value. Common values include:

- “0..1” (a unique value may be selected, but every possible qualifier value does not necessarily select a value).
- “1” (every possible qualifier value selects a unique target instance; therefore, the domain of qualifier values must be finite).
- “*” (the qualifier value is an index that partitions the target instances into subsets).

The qualifier attributes are drawn within the qualifier box. There may be one or more attributes shown one to a line. Qualifier attributes have the same notation as classifier attributes, except that initial value expressions are not meaningful.

It is permissible (although somewhat rare), to have a qualifier on each end of a single association.

### 3.45.3 Presentation Options

A qualifier may not be suppressed (it provides essential detail whose omission would modify the inherent character of the relationship).

A tool may use a lighter line for qualifier rectangles than for class rectangles to distinguish them clearly.

### 3.45.4 Style Guidelines

The qualifier rectangle should be smaller than the attached class rectangle, although this is not always practical.

### 3.45.5 Example

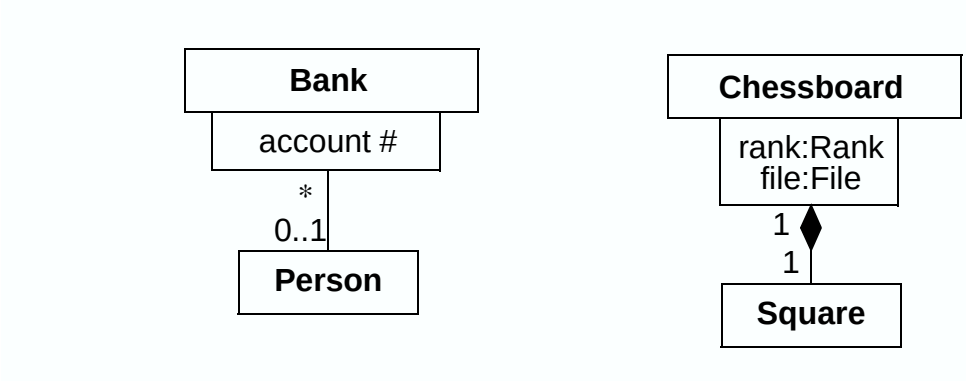


*Figure 3-42* Qualified Associations

### 3.45.6 Mapping

The presence of a qualifier box on an end of an association path maps into a list of qualifier attributes on the corresponding Association Role. Each attribute entry string inside the qualifier box maps into an Attribute.

## 3.46 Association Class

### 3.46.1 Semantics

An association class is an association that also has class properties (or a class that has association properties). Even though it is drawn as an association and a class, it is really just a single model element.

### 3.46.2 Notation

An association class is shown as a class symbol (rectangle) attached by a dashed line to an association path. The name in the class symbol and the name string attached to the association path are redundant and should be the same. The association path may have the usual adornments on either end. The class symbol may have the usual contents. There are no adornments on the dashed line.

### 3.46.3 Presentation Options

The class symbol may be suppressed. It provides subordinate detail whose omission does not change the overall relationship. The association path may not be suppressed.

### 3.46.4 Style Guidelines

The attachment point should not be near enough to either end of the path that it appears to be attached to, the end of the path, or to any of the association end adornments.

Note that the association path and the association class are a single model element and have a single name. The name can be shown on the path, the class symbol, or both. If an association class has only attributes, but no operations or other associations, then the name may be displayed on the association path and omitted from the association class symbol to emphasize its “association nature.” If it has operations and other associations, then the name may be omitted from the path and placed in the class rectangle to emphasize its “class nature.” In neither case are the actual semantics different.

### 3.46.5 Example

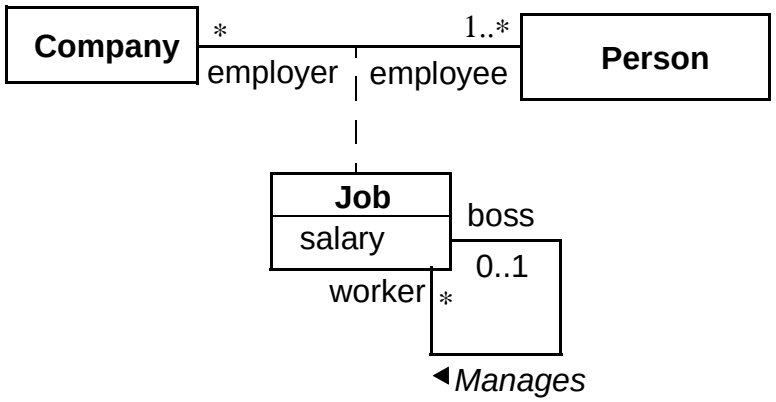


*Figure 3-43* Association Class

### 3.46.6 Mapping

An association path connecting two class boxes connected by a dashed line to another class box maps into a single AssociationClass element. The name of the AssociationClass element is taken from the association path, the attached class box, or both (they must be consistent if both are present). The Association properties map from the association path, as specified previously. The Class properties map from the class box, as specified previously. Any constraints or properties placed on either the association path or attached class box apply to the AssociationClass itself; they must not conflict.

## 3.47 N-ary Association

### 3.47.1 Semantics

An n-ary association is an association among three or more classifiers (a single classifier may appear more than once). Each instance of the association is an n-tuple of values from the respective classifier. A binary association is a special case with its own notation.

Multiplicity for n-ary associations may be specified, but is less obvious than binary multiplicity. The multiplicity on a role represents the potential number of instance tuples in the association when the other N-1 values are fixed.

An n-ary association may not contain the aggregation marker on any role.

### 3.47.2 Notation

An n-ary association is shown as a large diamond (that is, large compared to a terminator on a path) with a path from the diamond to each participant class. The name of the association (if any) is shown near the diamond. Role adornments may appear on each path as with a binary association. Multiplicity may be indicated; however, qualifiers and aggregation are not permitted.

An association class symbol may be attached to the diamond by a dashed line. This indicates an n-ary association that has attributes, operations, and/or associations.

### 3.47.3 Style Guidelines

Usually the lines are drawn from the points on the diamond or the midpoint of a side.

### 3.47.4 Example

This example shows the record of a team in each season with a particular goalkeeper. It is assumed that the goalkeeper might be traded during the season and can appear with different teams.

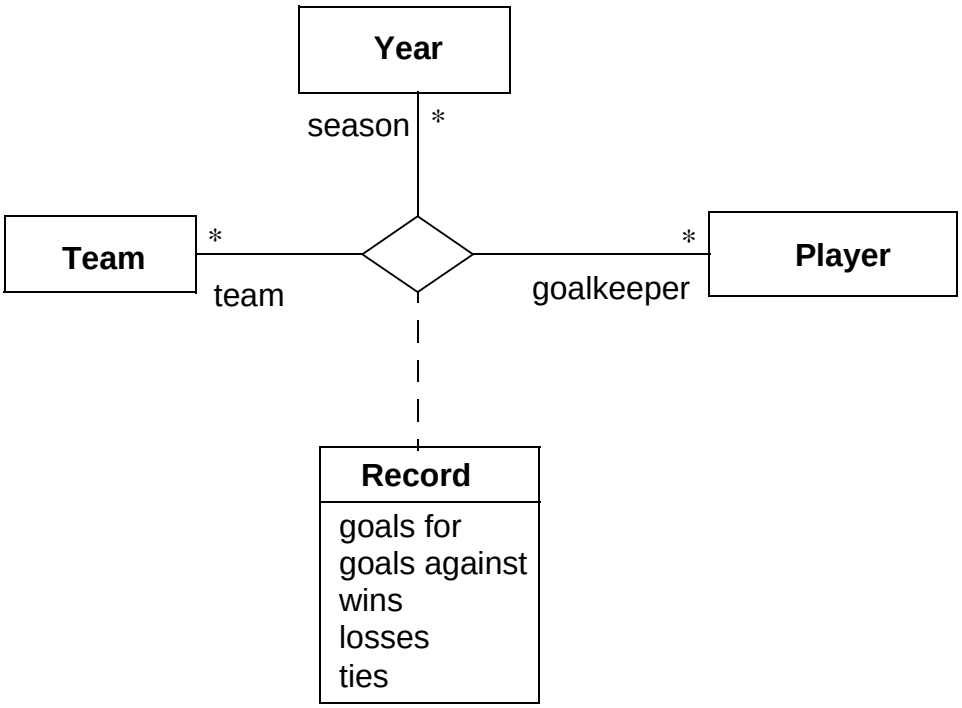


*Figure 3-44* Ternary association that is also an association class

### 3.47.5 Mapping

A diamond attached to some number of class symbols by solid lines maps into an N-ary Association whose AssociationEnds are attached to the corresponding Classes. The ordering of the Classifiers in the Association is indeterminate from the diagram. If a class box is attached to the diamond by a dashed line, then the corresponding Classifier supplies the classifier properties for an N-ary AssociationClass.

## 3.48 Composition

### 3.48.1 Semantics

Composite aggregation is a strong form of aggregation, which requires that a part instance be included in at most one composite at a time and that the composite object has sole responsibility for the disposition of its parts. The multiplicity of the aggregate end may not exceed one (it is unshared). See Section 3.43, “Association End,” on page 3-71 for further details.

The composite in a composition “projects” its identity onto the parts in the relationship. In other words, each part object in an object model can be identified with a unique composite object. It keeps its own identity as its primary identity. The point is that it can also be identified as being part of a unique composite. Composition is transitive. If object A is part of object B that is part of object C, then object A is also part of object C. A part may be identified with several composite objects in this way, each at a different level of detail.

The parts of a composition may include classes and associations (they may be formed into AssociationClasses if necessary). The meaning of an association in a composite object is that any tuple of objects connected by a single link must all belong to the *same* container object. In other words, the composite object projects its identity onto each link corresponding to the part end of a composition aggregation. If an association and two classes it relates are all related as parts to the same class as composite, a link that is an instance of the association is identified with an object that is an instance of the composite class; the objects connected by the link are also identified with the composite object; and they must all be associated with the same composite object.

### 3.48.2 Notation

Composition may be shown by a solid filled diamond as an association end adornment. Alternately, UML provides a graphically-nested form that is more convenient for showing composition in many cases.

Instead of using binary association paths using the composition aggregation adornment, composition may be shown by graphical nesting of the symbols of the elements for the parts within the symbol of the element for the whole. A nested class-like element may have a multiplicity within its composite element. The multiplicity is shown in the upper right corner of the symbol for the part. If the multiplicity mark is omitted, then the default multiplicity is many. This represents its multiplicity as a part within the composite classifier. A nested element may have a rolename within the composition; the name is shown in front of its type in the syntax:

*rolename* ‘:’ *classname*

This represents its rolename within its composition association to the composite.

Alternately, composition is shown by a solid-filled diamond adornment on the end of an association path attached to the element for the whole. The multiplicity may be shown in the normal way.

Note that attributes are, in effect, composition relationships between a classifier and the classifiers of its attributes.

An association drawn entirely within a border of the composite is considered to be part of the composition. Any instances on a single link of it must be from the same composite. An association drawn such that its path breaks the border of the composite is not considered to be part of the composition. Any instances on a single link of it may be from the same or different composites.

Note that the notation for composition resembles the notation for collaboration. A composition may be thought of as a collaboration in which all of the participants are parts of a single composite object.

Note that nested notation is not the correct way to show a class declared within another class. Such a declared class is not a structural part of the enclosing class but merely has scope within the namespace of the enclosing class, which acts like a package toward the inner class. Such a namescope containment may be shown by placing a package symbol in the upper right corner of the class symbol. A tool can allow a user to click on the package symbol to open the set of elements declared within it. The “anchor notation” (a cross in a circle on the end of a line) may also be used on a line between two class boxes to show that the class with the anchor icon declares the class on the other end of the line.

### 3.48.3 Design Guidelines

Note that a class symbol is a composition of its attributes and operations. The class symbol may be thought of as an example of the composition nesting notation (with some special layout properties). However, attribute notation subordinates the attributes strongly within the class; therefore, it should be used when the structure and identity of the attribute objects themselves is unimportant outside the class.

### 3.48.4 Example

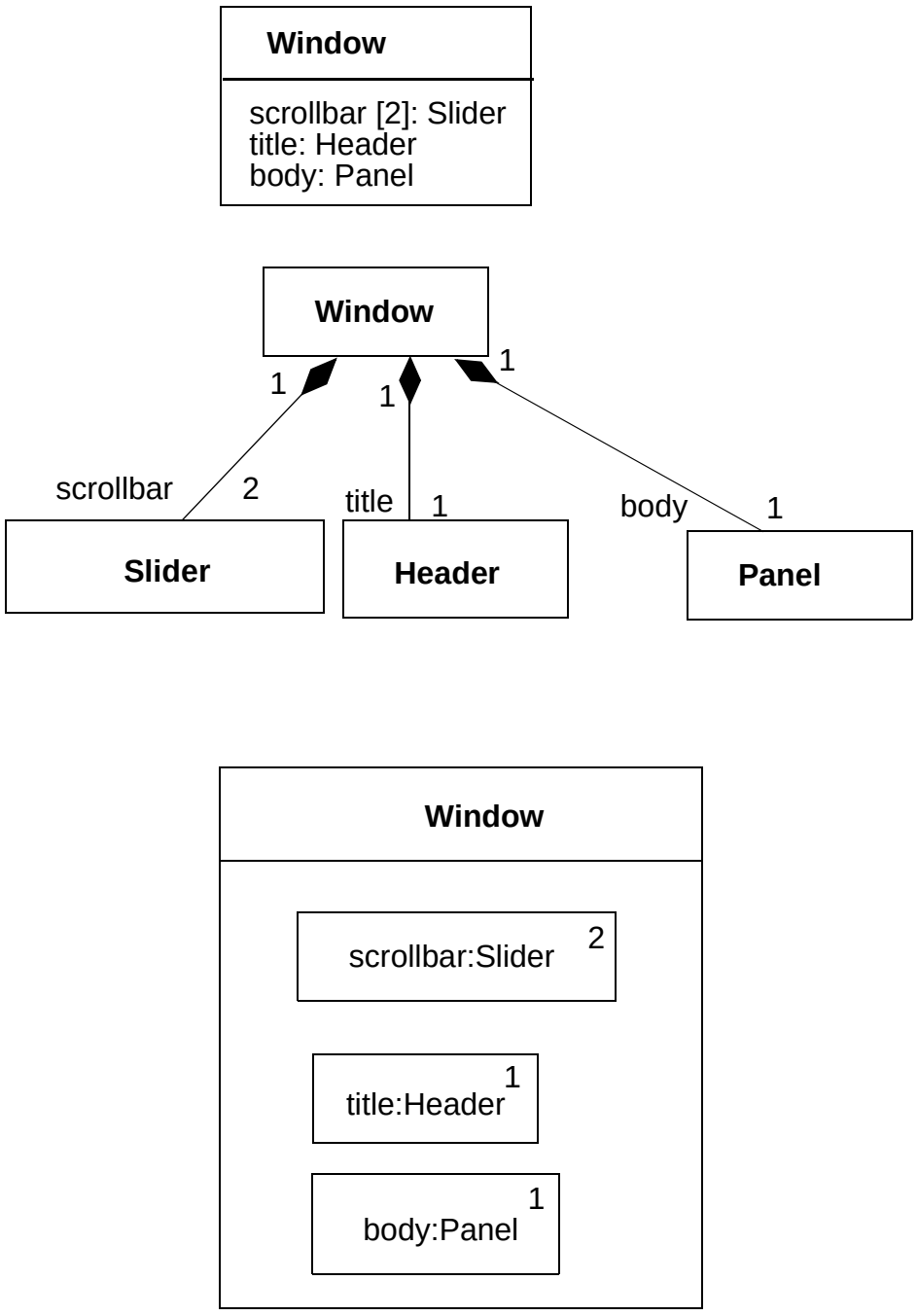


*Figure 3-45* Different Ways to Show Composition

### 3.48.5 Mapping

A class box with an attribute compartment maps into a Class with Attributes. Although attributes may be semantically equivalent to composition on a deep level, the mapped model distinguishes the two forms.

A solid diamond on an association path maps into the aggregation-composition property on the corresponding Association Role.

A class box with contained class boxes maps into a set of composition associations; that is, one composition association between the Class corresponding to the outer class box and each of the Classes corresponding to the enclosed class boxes. The multiplicity of the composite end of each association is 1. The multiplicity of each constituent end is 1 if not specified explicitly; otherwise, it is the value specified in the corner of the class box *or* specified on an association path from the outer class box boundary to an inner class box.

## 3.49 Link

### 3.49.1 Semantics

A link is a tuple (list) of object references. Most commonly, it is a pair of object references. It is an instance of an association.

### 3.49.2 Notation

A binary link is shown as a path between two instances. In the case of a link from an instance to itself, it may involve a loop with a single instance. See “Association” on page 3-68 for details of paths.

A rolename may be shown at each end of the link. An association name may be shown near the path. If present, it is underlined to indicate an instance. Links do not have instance names, they take their identity from the instances that they relate. Multiplicity is *not* shown for links because they are instances. Other association adornments (aggregation, composition, navigation) may be shown on the link ends.

A qualifier may be shown on a link. The value of the qualifier may be shown in its box.

#### 3.49.2.1 Implementation stereotypes

A stereotype may be attached to the link end to indicate various kinds of implementation. The following stereotypes may be used:

| | |
|---|---|
| «association» | association (default, unnecessary to specify except for emphasis) |
| «parameter» | method parameter |

| | |
|---|---|
| «local» | local variable of a method |
| «global» | global variable |
| «self» | self link (the ability of an instance to send a message to itself) |

#### 3.49.2.2 N-ary link

An n-ary link is shown as a diamond with a path to each participating instance. The other adornments on the association, and the adornments on the association ends, have the same possibilities as the binary link.

### 3.49.3 Example

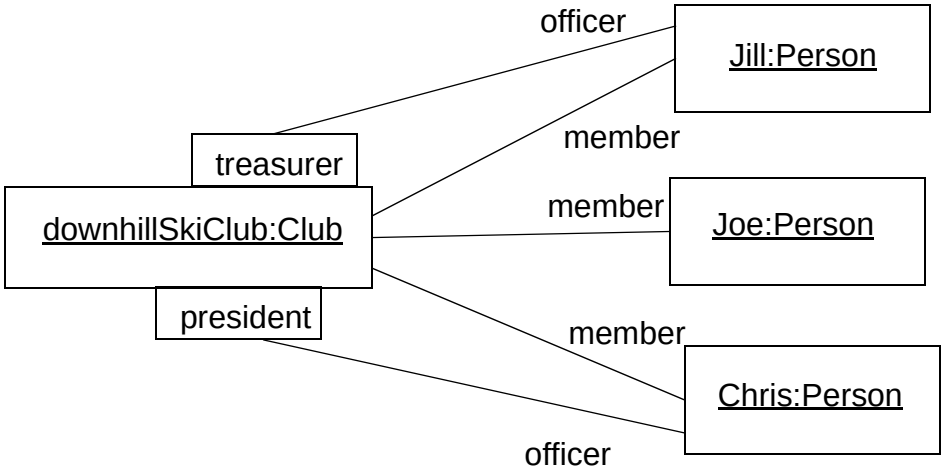


*Figure 3-46* Links

### 3.49.4 Mapping

Within an object diagram, each link path maps to a Link between the Instances corresponding to the connected class boxes. If a name is placed on the link path, then it is an instance of the given Association (and the rolenames must match or the diagram is ill formed).

## 3.50 Generalization

### 3.50.1 Semantics

Generalization is the taxonomic relationship between a more general element (the parent) and a more specific element (the child) that is fully consistent with the first element and that adds additional information. It is used for classes, packages, use cases, and other elements.

### 3.50.2 Notation

Generalization is shown as a solid-line path from the child (the more specific element, such as a subclass) to the parent (the more general element, such as a superclass), with a large hollow triangle at the end of the path where it meets the more general element.

A generalization path may have a text label called a discriminator that is the name of a partition of the children of the parent. The child is declared to be in the given partition. The absence of a discriminator label indicates the “empty string” discriminator which is a valid value (the “default” discriminator).

Generalization may be applied to associations as well as to classes. To notate generalization between associations, a generalization arrow may be drawn from a child association path to a parent association path. This notation may be confusing because lines connect other lines. An alternative notation is to represent each association as an association class and to draw the generalization arrow between the rectangles for the association classes, as with other classifiers. This approach can be used even if an association does not have any additional attributes, because a degenerate association class is a legal association.

The existence of additional children in the model that are not shown on a particular diagram may be shown using an ellipsis (. . .) in place of a child.

---

**Note –** This does not indicate that additional children may be added in the future. It indicates that additional children exist right now, but are not being seen. This is a notational convention that information has been suppressed, not a semantic statement.

---

Predefined constraints may be used to indicate semantic constraints among the children. A comma-separated list of keywords is placed in braces either near the shared triangle (if several paths share a single triangle) or near a dotted line that crosses all of the generalization lines involved. The following keywords (among others) may be used (the following constraints are predefined):

| | |
|---|---|
| overlapping | An element may have two or more children from the set as ancestors. An instance may be a direct or indirect instance of two or more of the children. |
| disjoint | No element may have two children in the set as ancestors. No instance may be a direct or indirect instance of two of the children. |
| complete | All children have been specified (whether or not shown). No additional children are expected. |
| incomplete | Some children have been specified, but the list is known to be incomplete. There are additional children that are not yet in the model. This is a statement about the model itself. Note that this is not the same as the ellipsis, which states that additional children exist in the model but are not shown on the current diagram. |

The *discriminator* must be unique among the attributes and association roles of the given parent. Multiple occurrences of the same discriminator name are permitted and indicate that the children belong to the same partition.

The use of multiple classification or dynamic classification affects the dynamic execution semantics of the language, but is not usually apparent from a static model.

### 3.50.3 Presentation Options

A group of generalization paths for a given parent may be shown as a tree with a shared segment (including the triangle) to the child, branching into multiple paths to each child.

If a text label is placed on a generalization triangle shared by several generalization paths to children, the label applies to all of the paths. In other words, all of the children share the given properties.

### 3.50.4 Example

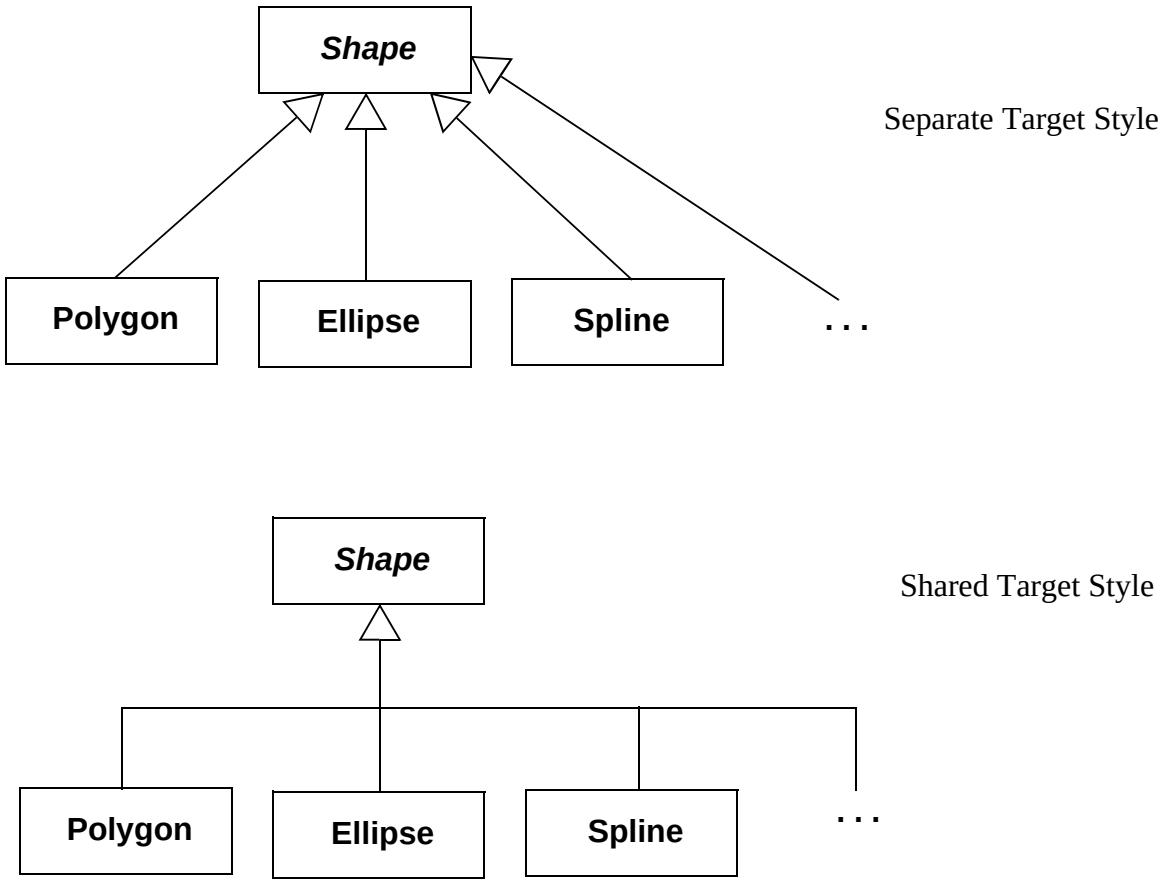


*Figure 3-47* Styles of Displaying Generalizations

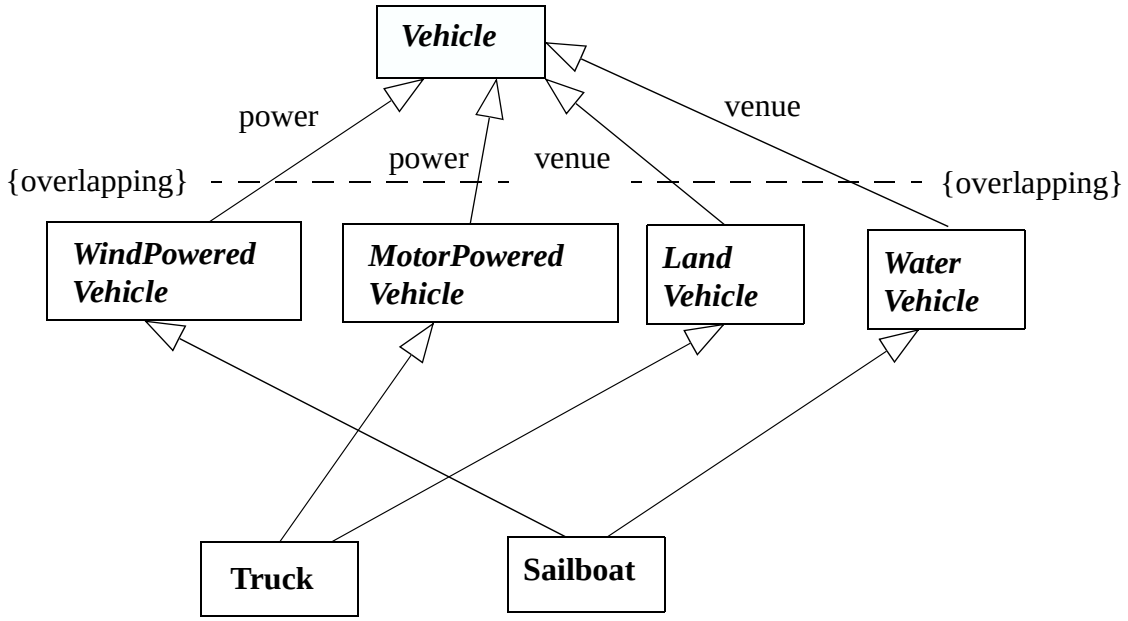


*Figure 3-48* Generalization with Discriminators and Constraints, Separate Target Style

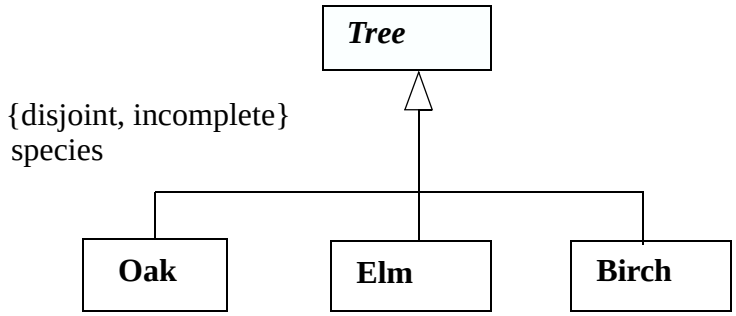


*Figure 3-49* Generalization with Shared Target Style

### 3.50.5 Mapping

Each generalization path between two element symbols maps into a Generalization between the corresponding GeneralizableElements. A generalization tree with one arrowhead and many tails maps into a set of Generalizations, one between each

element corresponding to a symbol on a tail and the single GeneralizableElement corresponding to the symbol on the head. That is, a tree is semantically indistinguishable from a set of distinct arrows, it is purely a notational convenience.

Any property string attached to a generalization arrow applies to the Generalization. A property string attached to the head line segment on a generalization tree represents a (duplicated) property on each of the individual Generalizations.

The presence of an ellipsis ("...") as a child node of a given parent indicates that the semantic model contains at least one child of the given parent that is not visible on the current diagram. Normally, this indicator will be maintained automatically by an editing tool.

## 3.51 Dependency

### 3.51.1 Semantics

A dependency indicates a semantic relationship between two model elements (or two sets of model elements). It relates the model elements themselves and does not require a set of instances for its meaning. It indicates a situation in which a change to the target element may require a change to the source element in the dependency.

### 3.51.2 Notation

A dependency is shown as a dashed arrow between two model elements. The model element at the tail of the arrow (the client) depends on the model element at the arrowhead (the supplier). The arrow may be labeled with an optional stereotype and an optional individual name.

It is possible to have a set of elements for the client or supplier. In this case, one or more arrows with their tails on the clients are connected to the tails of one or more arrows with their heads on the suppliers. A small dot can be placed on the junction if desired. A note on the dependency should be attached at the junction point.

The following kinds of Dependency are predefined and may be indicated with keywords. Note that some of these correspond to actual metamodel classes and others to stereotypes of metamodel classes. All of these are shown as dashed arrows with keywords in guillemets. The name column shows the name of the metamodel class or the informal name of the class with the given keyword stereotype.

| Keyword | Name | Description |
|---|---|---|
| access | Access | The granting of permission for one package to reference the public elements owned by another package (subject to appropriate visibility). Maps into a Permission with the stereotype access. |
| bind | Binding | A binding of template parameters to actual values to create a nonparameterized element. See Section 3.31, “Bound Element,” on page 3-54 for more details. Maps into a Binding. |
| derive | Derivation | A computable relationship between one element and another (one more than one of each). Maps into an Abstraction with the stereotype derivation. |
| import | Import | The granting of permission for one package to reference the public elements of another package, together with adding the names of the public elements of the supplier package to the client package. Maps into a Permission with the stereotype import. |
| refine | Refinement | A historical or derivation connection between two elements with a mapping (not necessarily complete) between them. A description of the mapping may be attached to the dependency in a note. Various kinds of refinement have been proposed and can be indicated by further stereotyping. Maps into an Abstraction with the stereotype refinement. |
| trace | Trace | A historical connection between two elements that represents the same concept at different levels of meaning. Maps into an Abstraction with the stereotype trace. |
| use | Usage | A situation in which one element requires the presence of another element for its correct implementation or functioning. May be stereotyped further to indicate the exact nature of the dependency, such as calling an operation of another class, granting permission for access, instantiating an object of another class, etc. Maps into a Usage. If the keyword is one of the stereotypes of Usage (call, create, instantiate, send), then it maps into a Usage with the given stereotype. |

### 3.51.3 Presentation Options

**Note –** The connection between a note or constraint and the element it applies to is shown by a dashed line without an arrowhead. This is not a Dependency.

### 3.51.4 Example

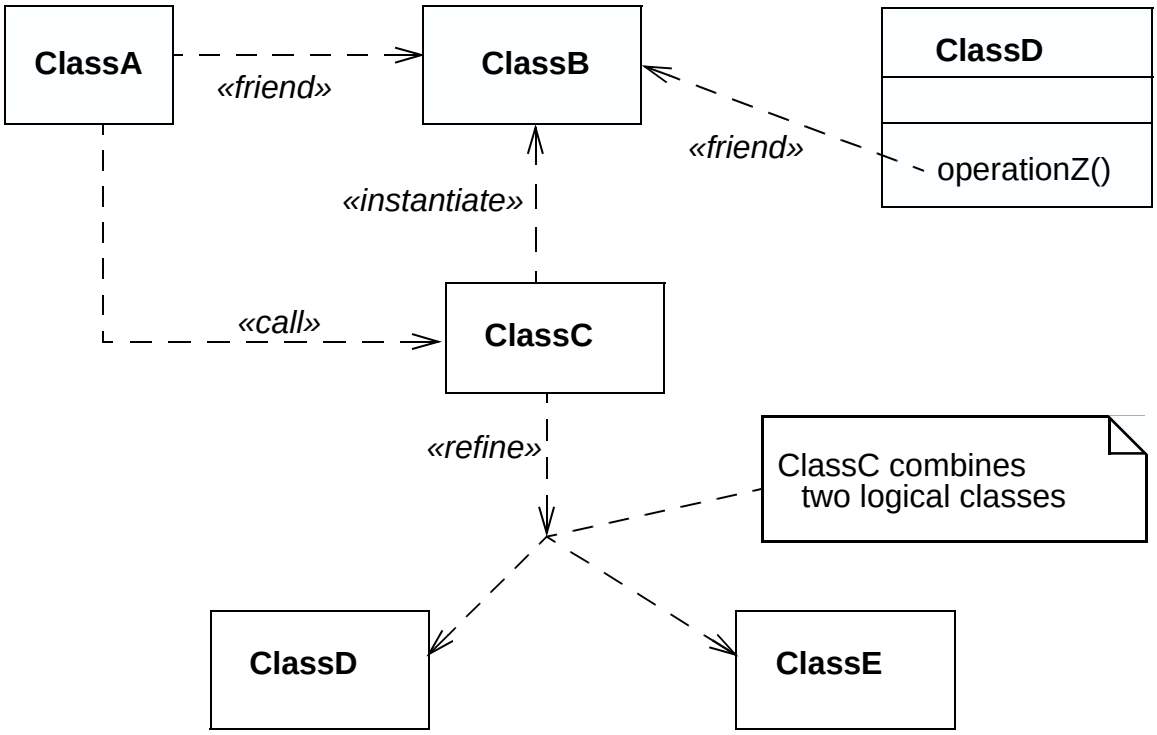


*Figure 3-50* Various Dependencies Among Classes

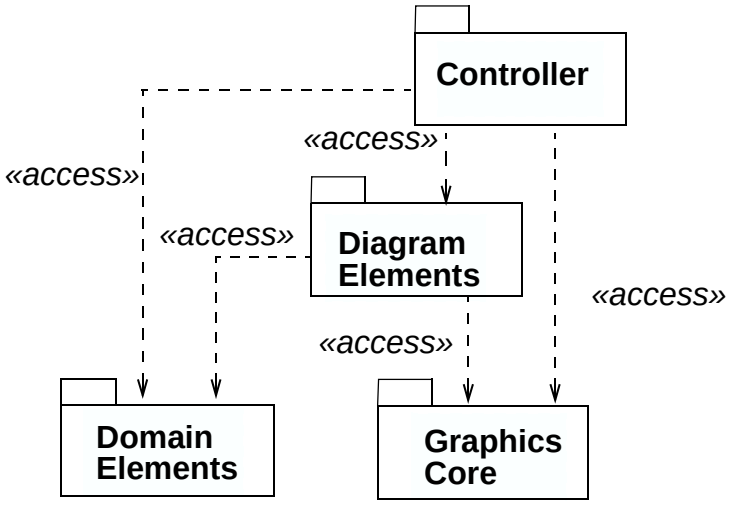


*Figure 3-51* Dependencies Among Packages

### 3.51.5 Mapping

A dashed arrow maps into the appropriate kind of Dependency (based on keywords) between the Elements corresponding to the symbols attached to the ends of the arrow. The stereotype and the name (if any) attached to the arrow are the stereotype and name of the Dependency.

## 3.52 Derived Element

### 3.52.1 Semantics

A derived element is one that can be computed from another one, but that is shown for clarity or that is included for design purposes even though it adds no semantic information.

### 3.52.2 Notation

A derived element is shown by placing a slash (/) in front of the name of the derived element, such as an attribute or a rolename.

### 3.52.3 Style Guidelines

The details of computing a derived element can be specified by a dependency with the stereotype «derive». Usually it is convenient in the notation to suppress the dependency arrow and simply place a constraint string near the derived element, although the arrow can be included when it is helpful.

## 3.53 InstanceOf

### 3.53.1 Semantics

Shows the connection between an instance and its classifier.

### 3.53.2 Notation

Shown as a dashed arrow with its tail on the instance and its head on the classifier. The arrow has the keyword «instanceOf».

### 3.53.3 Mapping

Maps into an instance relationship from the instance to the classifier.