context in which a tool implies a co rresponding model element and a Stere otype represented by the icon. The element and the stereotype have the stereotype relationship.

Part 5 - Static Structure Diagrams

Class diagrams show the static structu re of the model, in particular, the things that exist (such as classes and types), their internal structure, and their r elationships to other things. Clas s diag rams do not show temporal information, alth ough th ey may contain reified occurrences of things that have or things that describe temporal behavior. An object diagram shows instan ces co mpatible with a p articular class diag ram.

This section discusses classes and their variations, including templates and instantiated classes, and the r elationships be tween class es (association and generalization) and the contents of classes (attr ibutes and operations).

3.19   Class Diagram

A class diagram is a g raph of Classifier elements co nnected by their various static relationships. Note that a "class" d iagram may also contain interfaces, packages, relationships, and even instances, such as objects and links. Perhaps a better name would be "static structu ral diagram" but "class diagram" is s horter 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 di visions 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 diag rams may be o rganized into pac kages eith er with the ir underlying models or as separate packages th at build u pon the underlying mo del packages.

3.19.3 Mapping

A class diagram does not necessarily match a single semantic entity. A package within the static stru ctural model may be represented by one or more class diag rams. The division of the presentation into sep arate diag rams is for graphical c onvenience an d does no t imply a p artitioning of the m odel its elf. The conten ts of a diag ram map in to elements in the s tatic sem antic m odel. If a diag ram is part of a package, then its contents map into elements in th e same packag e (including possible re ferences to elements accessed o r 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 in stance of a class dia gram; it sho ws a sn apshot 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 with out a k eyword represents a class, and the oth er subclasses of *Classifier* are indicated with ke ywords. In th e sections th at f ollow, the discu ssion will focus on *Class,* but most of the notation applies to th e other element kind s as sem antically appropriate and as de scribed later under their own sectio ns.

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 intention 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 g raphical notation for declaring and using classes, as well as a te xtual notation for referencing classes within the descriptions of other model elements.

3.22.1 Semantics

A class represents a c oncept with in the sys tem being modeled. Classes h ave 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 d rawn as a solid -outline rectangle with three c ompartments s eparated b y horizontal lines. The top name compartment holds the class name and other general properties o f the class (including s tereotype); the middle list co mpartment ho lds a list of attributes; th e bottom list com partment holds a list o f operations.

See Section 3.23, "N ame Compartment," on page 3-35 and Section 3.24, "List Compartment," on p age 3-35 for more de tails.

3.22.2.1 References

By default a class shown within a package is assumed to be defined within that package. To show a r eference to a class de fined in another package, use th e syn tax

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

as the na me strin g in the nam e compartment. A fu ll path name ca n be speci fied by chaining to gether pack age names separ ated by double colo ns (::).

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 (Section 3.24, "List Compartment," on p age 3-35).

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 co mpartments ar e simply lists of string s. More complicated formats are possible, but UML does not specify such formats; th ey are a to ol responsibility. Appearance of each compartment should preferably be implicit based on its con tents. Co mpartment 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 "collaps ed" to sh ow jus t the ster eotype icon, with the nam e of the class either insid e the cla ss or belo w the ic on. Other contents of the class are su ppressed.

3.22.4 Style Guidelines

- Center cla ss name in boldface.
- Center keyword (including stereotype names) in p lain f ace with in g uillemets above class name.
- Begin class names with an uppercase letter.

- 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.
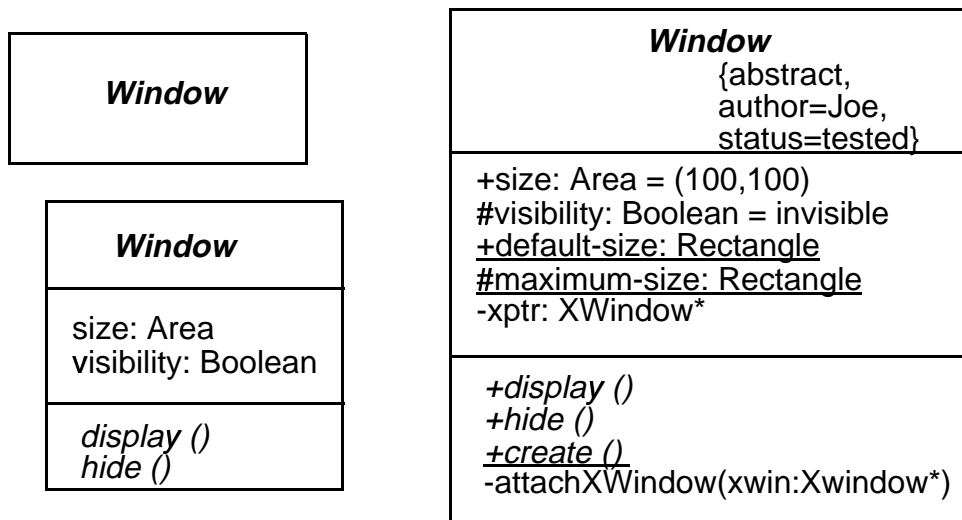
3.22.5 *Example*

Figure 3-17   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, its name appears in italics. 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}".

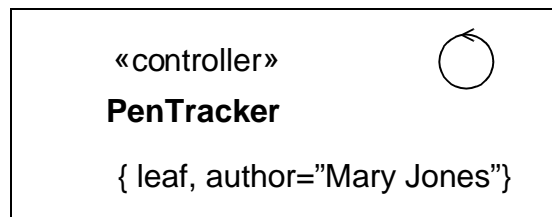The stereotype and property list are optional.

```
«controller»            ◯
PenTracker

{ leaf, author="Mary Jones"}
```

Figure 3-18 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 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 3-36 on page 3-75).
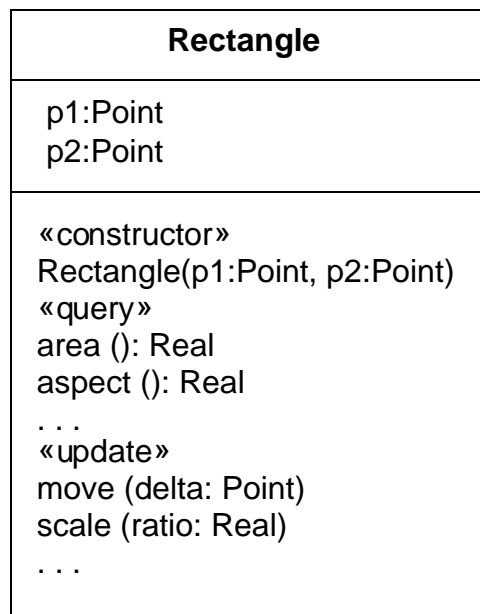
3.24.3 Example

Rectangle

p1:Point
p2:Point

«constructor»
Rectangle(p1:Point, p2:Point)
«query»
area (): Real
aspect (): Real
. . .
«update»
move (delta: Point)
scale (ratio: Real)
. . .

Figure 3-19 Stereotype Keyword Applied to Groups of List Elements

Reservation
operations
guarantee()
cancel ()
change (newDate: Date)
responsibilities
bill no-shows
match to available rooms
exceptions
invalid credit card

Figure 3-20 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 TypeExpression. It may resolve to a class name or it may be complex, such as array[String] of Point. In any case, the details of the attribute type expressions are not specified by UML. They depend on the expression syntax supported by the particular specification or programming language being used.

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

- Where *visibility* is one of:

  + public visibility

  # protected visibility

  - private 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*). 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*] shows the multiplicity of the attribute (Section 3.42, "Multiplicity," on page 3-68). The term may be omitted, in which case the multiplicity is 1..1 (exactly one).

- Where *type-expression* is a language-dependent specification of the implementation type of an attribute.

- 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 attribute name, for example:

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

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 [0..1]: String

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.

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

Particular attributes within a list may be suppressed (see Section 3.24, "List Compartment," on page 3-35).

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
+<u>default-size: Rectangle</u>
#<u>maximum-size: Rectangle</u>
-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 accordance 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 (e.g., 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

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*). 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 (i.e., 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 algorithm of a method may be indicated in a note attached to the operation entry.

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 method body may be shown in a note attached to the operation entry within the compartment (Figure 3-21). The line is drawn to the string within the compartment. This approach is useful mainly for showing small method bodies.
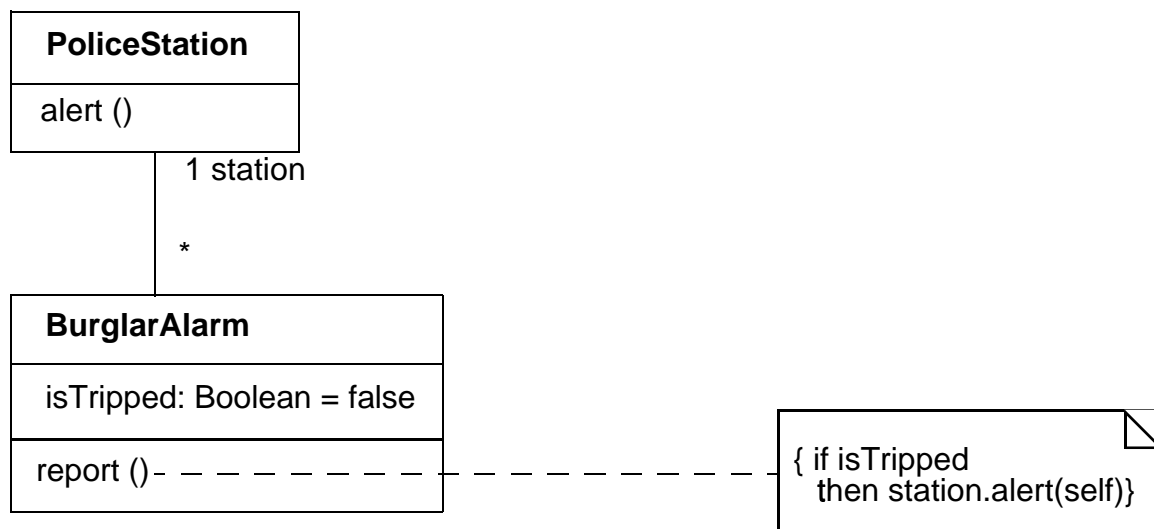

Figure 3-21 Note showing method body

3.26.4 Style Guidelines

Operation names typically begin with a lo wercase lette r. Operation names are in p lain face. An abstract operation may be shown in italics.

3.26.5 Example

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

Figure 3-22 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 t he class symbol. The properties of the operation map in accordance with the preceding descriptions. See the description of "Attribute" on page 3-38 for additional details. Parameters without keywords map into Parameters with kind=in, otherwise according to the keyword. Return value names may 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 th e corresponding Class o r 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 o f conventions such as italics fo r abstract o perations). If an abstract o peration entry ap pears with in a h ierarchy in wh ich the sam e operation has already been defined in a n ancestor, it h as no effect but is n ot an error un less the declarations are inconsistent.

Note that the operation string entry does not specify the b ody of a m ethod.

3.27 Type vs. Implementation Class

3.27.1 Semantics

Classes can be stereotyped as Types or Implementation Classes (alth ough they can be left undifferentiated as well). A Type is used to specify a domain of objects together with operations applicab le to the o bjects with out defining the physical implementation of those objects. A Type may not include any methods, but it may provide behavioral specifications for its o perations. It m ay also h ave attributes and associations th at are defined solely for the purpose of specifying the behavior of the type's operations and do not represent any actual im plementation of state da ta.

An Implementation Class defines the physical data structure (for attributes and associations) and methods of an object as implemented in traditional languages (e.g., C++, Smalltalk). 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. Note that the physical attributes and associations of the Implementation Class do not have to be the same as those of any Type it realizes and that the Implementation Class may provide methods for its operations in terms of its physical attributes and associations.

An object may have at most one Implementation Class, since this specifies the physical implementation of the object. However, an object may conform to multiple different Types. If the object has an Implementation Class, then that Implementation Class should realize the Types to which the object conforms. If dynamic classification is used, then the Types to which an object conforms may actually change dynamically. A Type may be used in this way to characterize a changeable role that an object may adopt and later abandon.

Although the use of types and implementation classes is different, their internal structure is the same and they are both classifiers of objects. Therefore they are modeled as stereotypes of classes. As such, they both fully support the usual generalization/specialization and the inheritance of attributes, associations, and operations. Note, however, the types may only specialize other types and implementation classes may only specialize other implementation classes. Types and implementation classes can be related only be realization.

3.27.2 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.
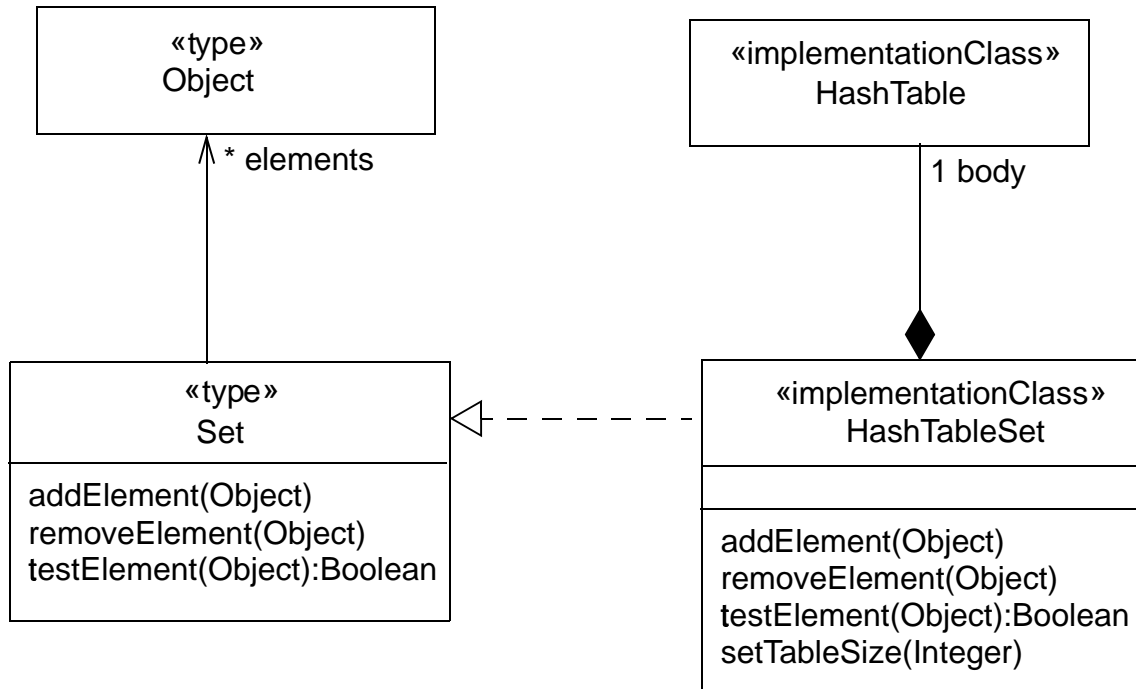
3.27.3 Example


```
┌─────────────────────┐          ┌─────────────────────────┐
│      «type»          │          │  «implementationClass»  │
│      Object          │          │       HashTable         │
└─────────────────────┘          └─────────────────────────┘
          ↑ * elements                        │ 1 body
                                               │
                                               ◆
┌─────────────────────┐          ┌─────────────────────────┐
│      «type»          │ ◁----    │  «implementationClass»  │
│       Set            │          │      HashTableSet       │
├─────────────────────┤          ├─────────────────────────┤
│ addElement(Object)   │          │                         │
│ removeElement(Object)│          ├─────────────────────────┤
│ testElement(Object):Boolean│    │ addElement(Object)      │
└─────────────────────┘          │ removeElement(Object)   │
                                  │ testElement(Object):Boolean│
                                  │ setTableSize(Integer)   │
                                  └─────────────────────────┘
```

Figure 3-23 Notation for Types and Implementation Classes

3.27.4 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.

3.28 Interfaces

3.28.1 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, 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).

3.28.2 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).
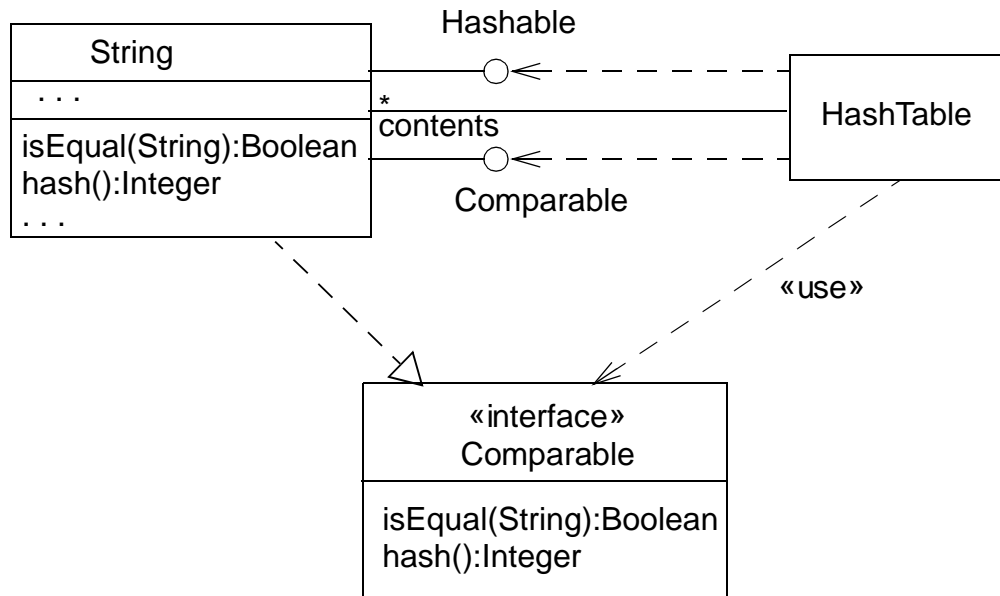
3.28.3 Example

Figure 3-24 Interface Notation on Class Diagram

3.28.4 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.

3.29 Parameterized Class (Template)

3.29.1 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 n ot a d irectly u sable class b ecause it has u nbound p arameters. Its parameters must be bound to actual values to create a bound form that is a class. On ly a class can be a sup erclass or the target of an associa tion (a o ne-way ass ociation *from* the template *to* another class is permissible, however). A template may be a subclass of an ordinary class. T his im plies th at all classes form ed by binding it ar e su bclasses o f the gi ven s uperclass.

Parameterization can be applied to other ModelElements, such as Collaborations or even entire Packages. The de scription gi ven here f or classes app lies to other kinds of modeling elements in the obvious way.

3.29.2 Notation

A small dashed rectangle is superimposed on the upper right-hand corner of the rectangle for the class (or to th e symbol for another modeling element). The dashed rectangle contains a p arameter list of formal parameters fo r the class an d 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 c lass rectangle; however, they may also inclu de occurrences of the formal parameters. Occu rrences of the formal parameters can also occ ur inside of a context fo r the class, fo r example, to show a re lated class iden tified by one of th e parameters.

3.29.3 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 = defau lt-value*

- Where *name* is an identifier for the parameter with scope inside the template.

- Where *type* is a string designating a *TypeExpression* for the parameter.

- Where *default-value* is a string designating an Ex pression for a default value that is used when the corr esponding argument is om itted in a Binding. The e qual 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 a rgument in a Binding must be the n ame of a Cl assifier (including a class or a data type). Other parameter ty pes (such as Integer) must be e xplicitly shown. The value sup plied for an argument in a B inding must be an ac tual in stance value of the given kind.
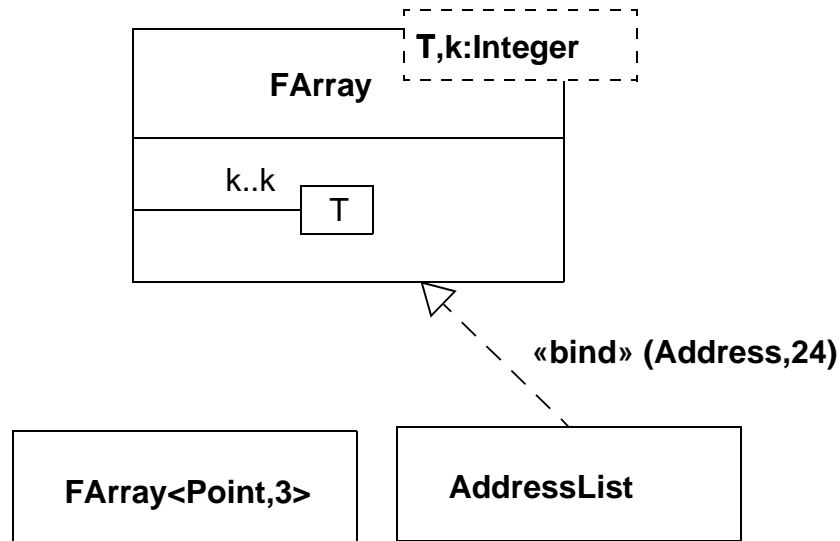
3.29.4 Example

Figure 3-25 Template Notation with Use of Parameter as a Reference

3.29.5 Mapping

The addition of the template dashed box to a symbol causes the addition of the parameter names in the list as Mo delElements within th e Namespace of th e 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 th e base Mod elElement.

3.30   Bound Element

3.30.1 Semantics

A template cannot be used directly in a n ordinary relationship such as generalization or association, because it ha s a free par ameter th at is no t mea ningful outside of a scop e that declares the param eter. To be us ed, a tem plate's par ameters must be *bound* to actual v alues. Th e ac tual v alue f or e ach parameter is an expression defined within the scope of use. I f the ref erencing scope is itself a tem plate, then the p arameters of the referencing tem plate ca n be u sed as actual v alues in bin ding the referenced tem plate. The para meter n ames in the tw o tem plates cann ot be assumed to correspond because they have n o scope outside of their r espective templates.

3.30.2 Notation

A bound element is in dicated by a te xt syn tax in th e na me strin g of an ele ment, 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 descr ibed by the te mplate Varray.

The number and type of v alues must match the nu mber and type of the tem plate parameters for the template of the g iven name.

The bound element name may be used anywhere that an element name of the parameterized kind could be used. For example, a b ound class name co uld 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 sp ecified by its te mplate; theref ore, its content m ay not be extended. Declaration of new attributes or operations for classes is not permitted, for e xample, but a bo und class co uld be su bclassed an d the subclass extended in th e u sual way.

The relation ship between th e bo und elem ent and its template alter natively may be shown by a Depen dency relationship wi th 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.

3.30.3 Style Guidelines

The attribute and operation compartments are normally suppressed within a bound class, because they m ust not be modified in a bound te mplate.

3.30.4 Example

See Figure 3-25 on page 3-50.

3.30.5 Mapping

The use of the bound element syntax for the name of a symbol maps into a Binding dependency between the de pendent ModelElement (su ch as Class) co rresponding to the bound elemen t symbol and the pro vider Mo delElement (again, such as C lass) whose name matches the nam e part of the bo und element without the arguments. If the name does not m atch a template element or if the number of arguments in the bound element does not match the n umber of p arameters in the template, then the model is ill formed. Each argument in the bound element maps into a ModelElement bearing an argument link to the Bin ding dependency. An e xplicitly drawn «bind» dependency symbol maps to a Binding dependency with arguments as de scribed above.

3.31   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 class.

3.31.1 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.

3.31.2 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.

3.31.3 Example

```
┌──────────────────────┐
│      «utility»       │
│      MathPak         │
├──────────────────────┤
│ sin (Angle): Real    │
│ cos (Angle): Real    │
│ sqrt (Real): Real    │
│ random(): Real       │
└──────────────────────┘
```
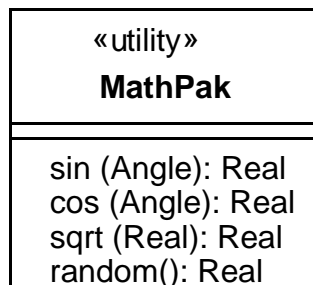
Figure 3-26 Notation for Utility

3.31.4 Mapping

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

3.32   Metaclass

3.32.1 Semantics

A metaclass is a class whose instances are classes.

3.32.2 Notation

A metaclass is s hown as the ster eotype «metaclass» o f Class.

3.32.3 Mapping

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

3.33   Enumeration

3.33.1 Semantics

An Enumeration is a user-defined data type whose instances are a set of user-specified named enumeration liter als. The li terals ha ve a relative order but n o algebra is defined on the m.

3.33.2 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 liter als may be placed, one to a line, in the m iddle compartment. Operations defined on the liter als may be placed in th e lower compartment. The lower and mid dle compartments may be suppressed.

3.33.3 Mapping

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

3.34   Stereotype

3.34.1 Semantics

A Stereotype is a user-defined metaelement whose structure matches an existing UML metaelement.

3.34.2 Notation

A Stereotype is shown us ing the Classif ier notation (a r ectangle) with th e keyword «stereotype». The name of the Stereotype is placed in the upper compartment. Constraints on eleme nts de scribed by the ster eotype may be place d in a na med compartment called **Constraints**. Required tags may be placed in a named compartment called **Tags**.

The base element may be indicated by a property string of the form {baseElement
= name}.

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.

3.34.3 Mapping

Maps into a Stereotype with the given constraints and base element.

3.35   Powertype

3.35.1 Semantics

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

3.35.2 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.

3.35.3 Mapping

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

3.36   Class Pathnames

3.36.1 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 c lasses also appear in te xt expressions, most n otably in ty pe specifications for attributes an d variables. In these plac es a ref erence t o a cla ss is indicated by simply including the name of the class itself, including a possible package name, subject to the sy ntax ru les of the e xpression.
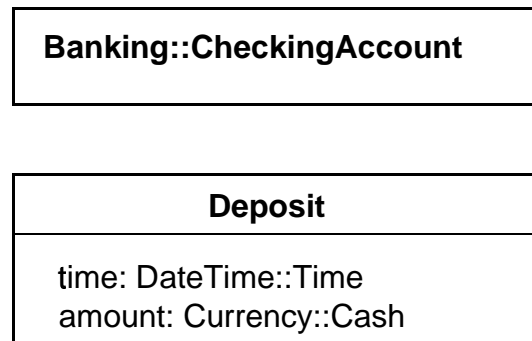
3.36.2 Example

```
┌─────────────────────────────┐
│                             │
│   Banking::CheckingAccount   │
│                             │
└─────────────────────────────┘
```

```
┌─────────────────────────────┐
│           Deposit           │
├─────────────────────────────┤
│   time: DateTime::Time       │
│   amount: Currency::Cash     │
└─────────────────────────────┘
```

Figure 3-27 Pathnames f or Classes in Other Packages

3.36.3 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 gi ven nam e. The name is a ssumed to be defined in th e target package; o therwise, the model is ill form ed. A R elationship from a symbol in the current package (i.e., the package containing the diagram and its mapped elements) to a symbol in another package is part of the current package.

3.37 Accessing or Importing a Package

3.37.1 Semantics

An element may reference an element contained in a different package. On the package level, the «access» dependency indicates tha t the contents of the target pack age may be referenced by the clien t package or packages recursively embedded within it. The target references must have visibility sufficient for the ref erents: public visibility for a n unrelated package, public or protected v isibility for a descenden t of the target package, or any visibil ity for a package nested inside the tar get 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 acce ss dependency does not modify the namespace of the clien t or in an y other way automatically create references; it m erely g rants perm ission to establis h references. Note also that a tool could automatically create access dependencies for users if d esired when references are crea ted.

An import dependency grants access and also lo ads the names with appropriate visibility in the target namespace into the acce ssing package (i.e., a pathname is n ot necessary to reference them). Such names are not automatically reexported; however, a name must be explicitly re exported (and may be gi ven a new name and visibility at the same time).

3.37.2 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 ster eotype keyword «access». T his dependency indicates that elements within the clie nt package m ay legally reference ele ments with in the sup plier. The references m ust also satisfy v isibility con straints specified b y the su pplier. Note that the dependency does no t au tomatically cre ate an y references. It me rely grants permission for th em to be establish ed.

The im port dep endency has th e s ame notation as the acc ess depen dency except it h as the stereotype keyword «import».
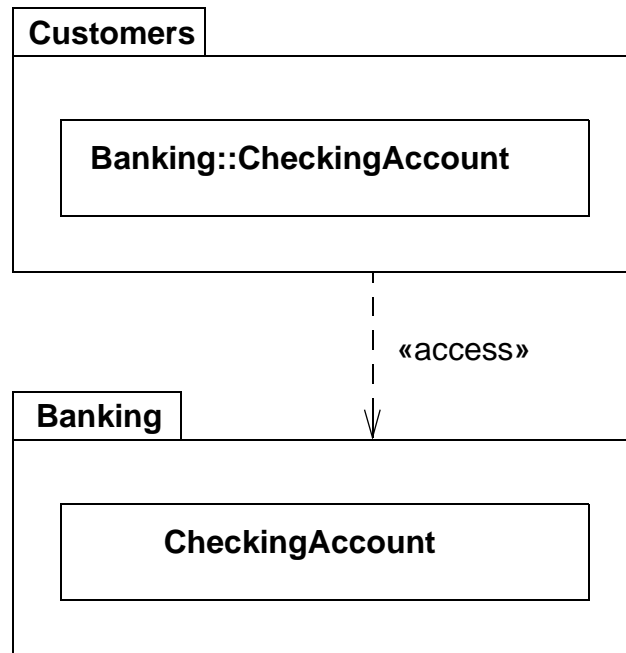
3.37.3 Example

Figure 3-28 Access Dependency Among Packages

3.37.4 Mapping

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

3.38 Object

3.38.1 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.

3.38.2 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 u nderlined, 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 (i.e., only one implementation class permitted, but multiple types permitted).

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

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

The list must be a comma-separated list of names of states that ca n 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 e xpected th at a tool will sp ecify such a s yntax 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.

3.38.3 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.

3.38.4 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.

3.38.5 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.

3.38.6 Example
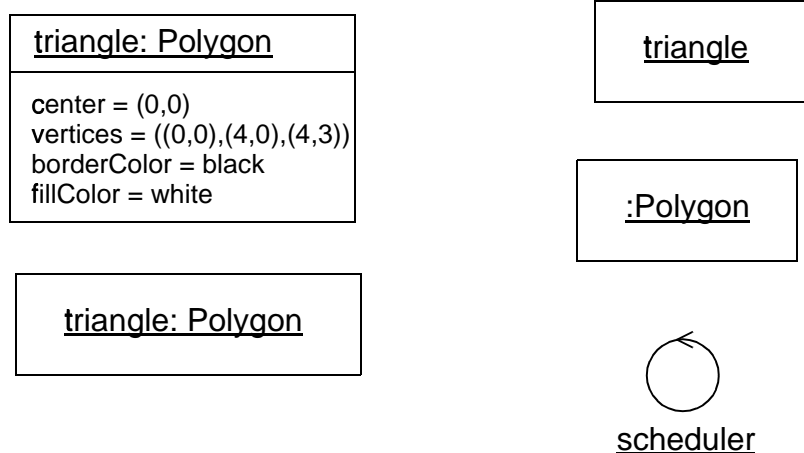
Figure 3-29 Objects

3.38.7 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.

3.39 Composite Object

3.39.1 Semantics

A composite object represents a high-level object made of tightly-bound parts. This is an instance of a c omposite class, wh ich implies the co mposition ag gregation between the class and its parts. A composite object is sim ilar to (but simpler and more restricted than) a collabo ration; however, it is def ined completely b y composition in a static model. See Section 3.46, "Composition," on page 3-73.

3.39.2 Notation

A composite object is shown as an object symbol. The name string of the composite object is placed in a co mpartment near th e top of the rectangle (as with an y object). The lower compartment holds the parts of the composite object in stead of a lis t of attribute values. (However, even a list o f attribute values m ay be regarded as the parts of a composite o bject, so there is n ot a great difference.) It is p ossible for so me of the parts to be composite objects with further nesting.

3.39.3 Example
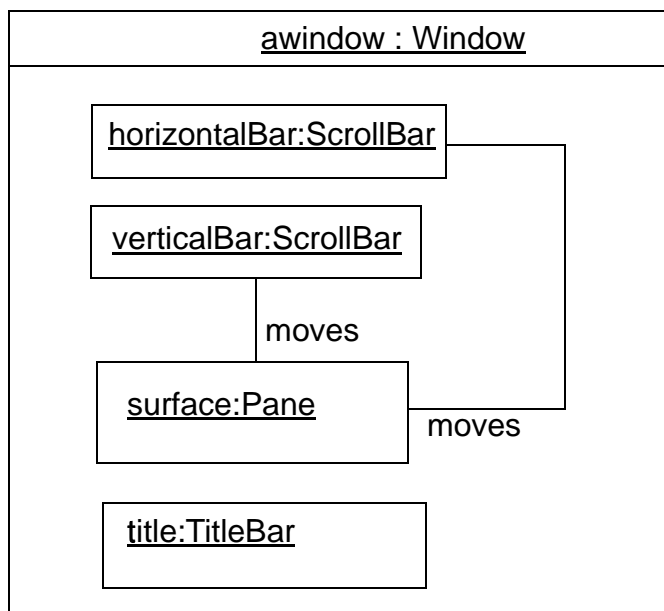
Figure 3-30 Composite Objects

3.39.4 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).

3.39.5 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.

3.40   Binary Association

3.40.1 Semantics

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

3.40.2 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.

3.40.2.1 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.

3.40.2.2 *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.

3.40.3 *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).

3.40.4 *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.

3.40.5 *Options*

3.40.5.1 *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 th e constraint s tring "{xo r}" labelin g the d ashed li ne. Any ins tance of the classif ier may only p articipate in o ne of th e assoc iations at on e time. Eac h rolename must be different. (This is simply a predefined use of the constraint notation.)
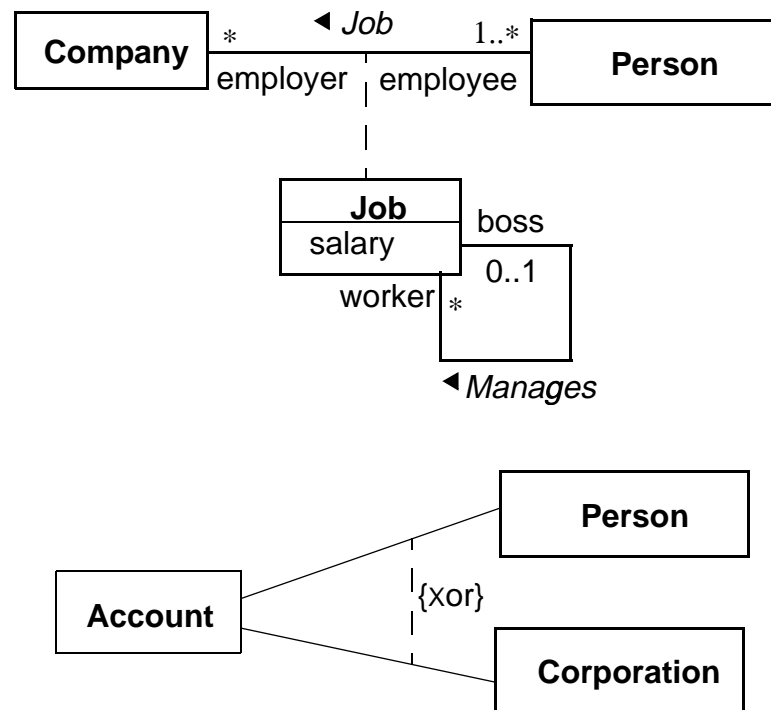
3.40.6 Example

Figure 3-31 Association Notation

3.40.7 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 th e tail of the a rrow is the first class and th e Classifier corresponding to the h ead of the arrow is the second Classifier in the o rdering of ends of th e Association; otherwise, the ordering of ends in the asso ciation is un determined. The adornments on the path map into properties of the Association as described above. The Association is owned by the package containing the diagram.

3.41 Association End

3.41.1 Semantics

An association end is simply an end of an association where it connects to a classifier. It is part of the asso ciation, not part of the classifier. Each association has two or more ends. Most of the interesting details ab out an asso ciation are attached to its en ds. An association end is not a separable element, it is ju st a mechanical part of an associati on.

3.41.2 Notation

The path may have graphical ado rnments at eac h end where the p ath con nects to th e classifier symbol. These adornments indicate properties of the association related to the classifier. The adornments a re part of the association sy mbol, not part of the classifier symbol. The end adornments are either attac hed to the en d of the lin e, or near the en d of the lin e, and must dr ag with it. The following kinds of adornments may be attach ed to an a ssociation end.

3.41.2.1 multiplicity

Specified by a text sy ntax, multiplicity m ay be suppressed on a particular ass ociation or for an entire diagram. In an inco mplete model the multiplicity may be un specified in the model its elf. In this case, it m ust be s uppressed in the no tation. See Section 3.42, "Multiplicity," on page 3-68.

3.41.2.2 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 s et). Various kin ds of ordering can be specified as a constraint on the association en d. The declaration does not specify how the ordering is established or maintained. Opera tions that insert new elements must make pro vision for specifying their position either implicitly (such as at the end) or explicitly. Possible values in clude:

- unordered - the elements form an unordered set. This is th e default and need not be shown explicitly.

- ordered - the elements of the set ha ve an ordering, but dup licates are still prohibited. This ge neric spe cification includes all kinds of ordering. This ma y be specified by the keyword syn tax "{ordered}".

An ordered relation ship may be im plemented in v arious ways; however, this is normally specified as a language-specified code generation property to select a particular implementation. An im plementation extension might s ubstitute the d ata structure to hold the ele ments for the generic specification "ordered."

At implementation le vel, sorting m ay also be sp ecified. It do es 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.

3.41.2.3 *qualifier*

A qualifier is optional, but not suppressible. See Section 3.43, "Qualifier," on page 3-69.

3.41.2.4 *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 3-33 for details.

3.41.2.5 *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 3.46, "Composition," on page 3-73.

3.41.2.6 *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.

3.41.2.7 *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.

3.41.2.8 *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.

3.41.2.9 *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 "Attribute" on page 3-38 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.

3.41.3 *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.

3.41.4 Style Guidelines

If there a re multiple ado rnments on a sing le association end, they are presented in the following order, reading from the end of the path attached to th e 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 associatio n. They may be place d 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 cou nterclockwise), but this is s ometimes overridden by the n eed for clarity in a cr owded layout. A r olename and a mu ltiplicity may be placed on opposite sides of the same association end, or they may be p laced together (for example, "* employee").

3.41.5 Example
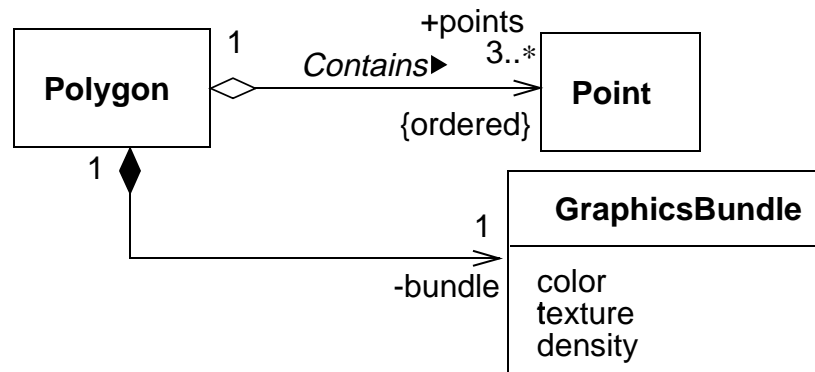
Figure 3-32 Various A dornments on A ssociation R oles

3.41.6 Mapping

The adornments o n the end of an associa tion path map into p roperties of the corresponding role of the Ass ociation. In g eneral, implication s cannot be d rawn from the absence of an adornment (it may simply be suppressed) but see the p receding descriptions for details. The interface spec ifier maps into the "specification" rolename in the AssociationEnd-Classifier association.

3.42 *Multiplicity*

3.42.1 *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 no n-negative integers.

3.42.2 *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 (i.e., no dynamic evaluation of expressions, essentially the same rule on literal values as most programming languages).

3.42.3 *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".

3.42.4 *Example*

0..1

1

0..*

*

1..*

1..6

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

3.42.5 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 Multip licity is a v alue and not an o bject. It
cannot stand on its o wn, but is the value of some element property.

3.43 Qualifier

3.43.1 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 associa tion.

3.43.2 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 associa tion. An instan ce of the source cla ssifier, together with a v alue of the
qualifier, uniquely select a partition in the set of ta rget classifier instan ces on the other
end of the asso ciation (i.e., every target falls into e xactly o ne partition).

The multiplicity attached to th e target end denotes the possible cardin alities o f the set
of target instances selected by the pairing of a sou rce instance and a q ualifier value.
Common values in clude:

- "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 qu alifier v alues m ust 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 s hown on e to a line. Q ualifier attr ibutes h ave the sam e notation as classif ier
attributes, e xcept that in itial value expressions are not meaningful.

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

3.43.3 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.

3.43.4 Style Guidelines

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

3.43.5 Example
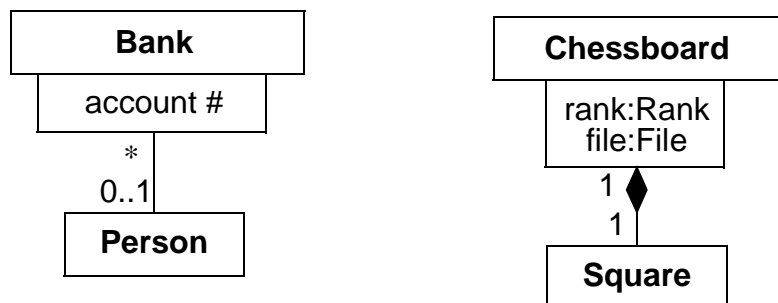
Figure 3-33 Qualified Associations

3.43.6 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.

3.44 Association Class

3.44.1 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.

3.44.2 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 they 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.

3.44.3 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.

3.44.4 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.
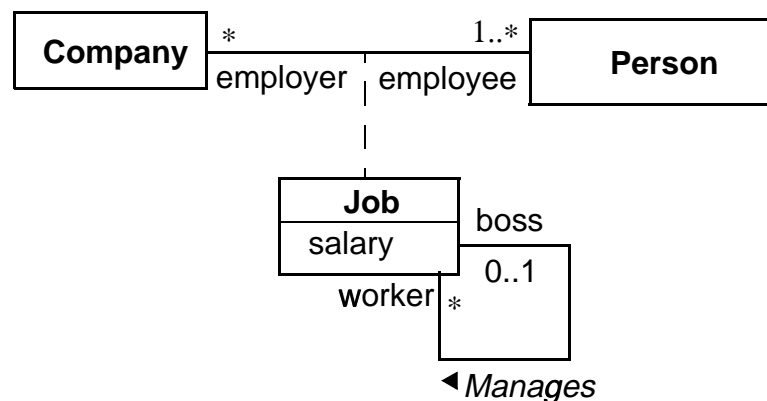
3.44.5 Example

Figure 3-34 Association Class

3.44.6 Mapping

An association path connecting two class bo xes connected by a da shed line to an other class box maps into a sin gle Ass ociationClass elem ent. The name of the AssociationClass element is taken f rom the association path, the attac hed class bo x, or both (they must be consistent if both are present). The Association properties map from the association path, as sp ecified previously. The Class pro perties m ap from the cla ss box, as sp ecified previously. Any con straints or p roperties p laced on either the association path or atta ched class b ox apply to th e Association Class itself ; they must not conflict.

3.45 N-ary Association

3.45.1 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 associa tion is an n-tuple of values from the re spective 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 th e association when the oth er N-1 v alues are fixed.

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

3.45.2 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 n ame of the associati on (if any) is sh own near the diamond. Role adornments may appear on each path as with a bin ary association. Mu ltiplicity m ay be ind icated; however, qualifiers and aggregation are n ot permitted.

An association class symbol may be attached to the diamond by a dashed line. This indicates an n -ary association th at ha s attributes, o perations, an d/or associations.

3.45.3 Style Guidelines

Usually the lines are drawn from the p oints on the diam ond or the midpoint of a sid e.

3.45.4 Example

This example shows the record of a team in each season with a particular goalkeeper. It is assumed that the g oalkeeper m ight be trad ed during the season and can ap pear with different team s
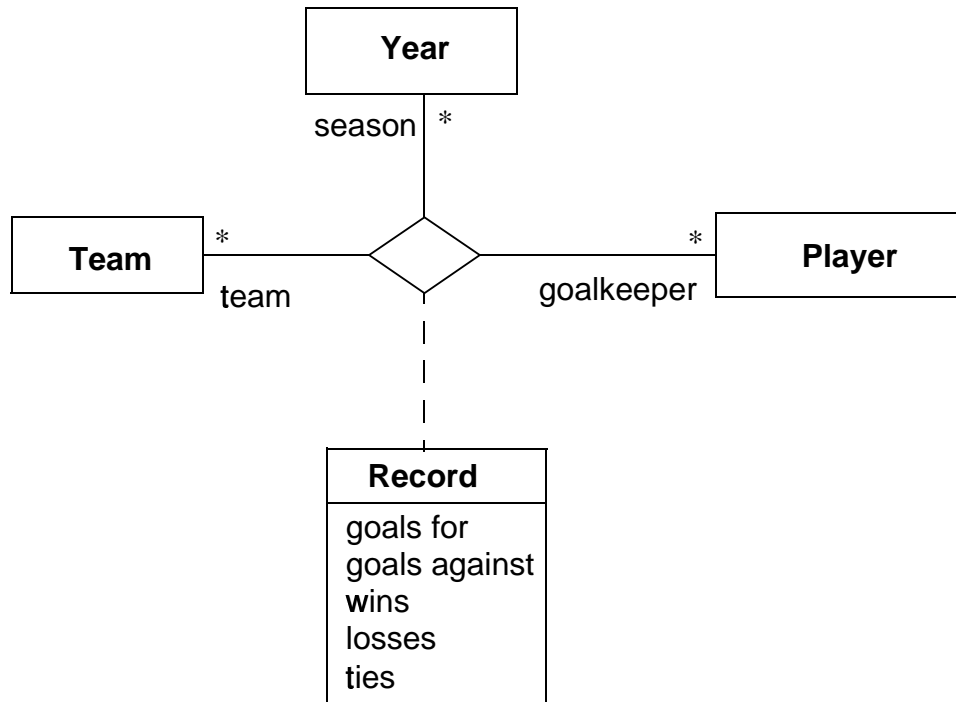
.

Figure 3-35 Ternary association that is also an association class

3.45.5 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.

3.46 Composition

3.46.1 Semantics

Composition is a form of aggregation with strong ownership and coincident lifetime of part with the whole. The multiplicity of the aggregate end may not exceed one (it is unshared). See Section 3.41, "Association End," on page 3-64 for further details.

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 composition is that any tuple of objects connected by a single link must all belong to the *same* container object.

3.46.2 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.

3.46.3 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.

3.46.4 Example

Window

scrollbar [2]: Slider
title: Header
body: Panel

Window

1

1

1

scrollbar   2

title   1

body   1

Slider

Header

Panel

Window

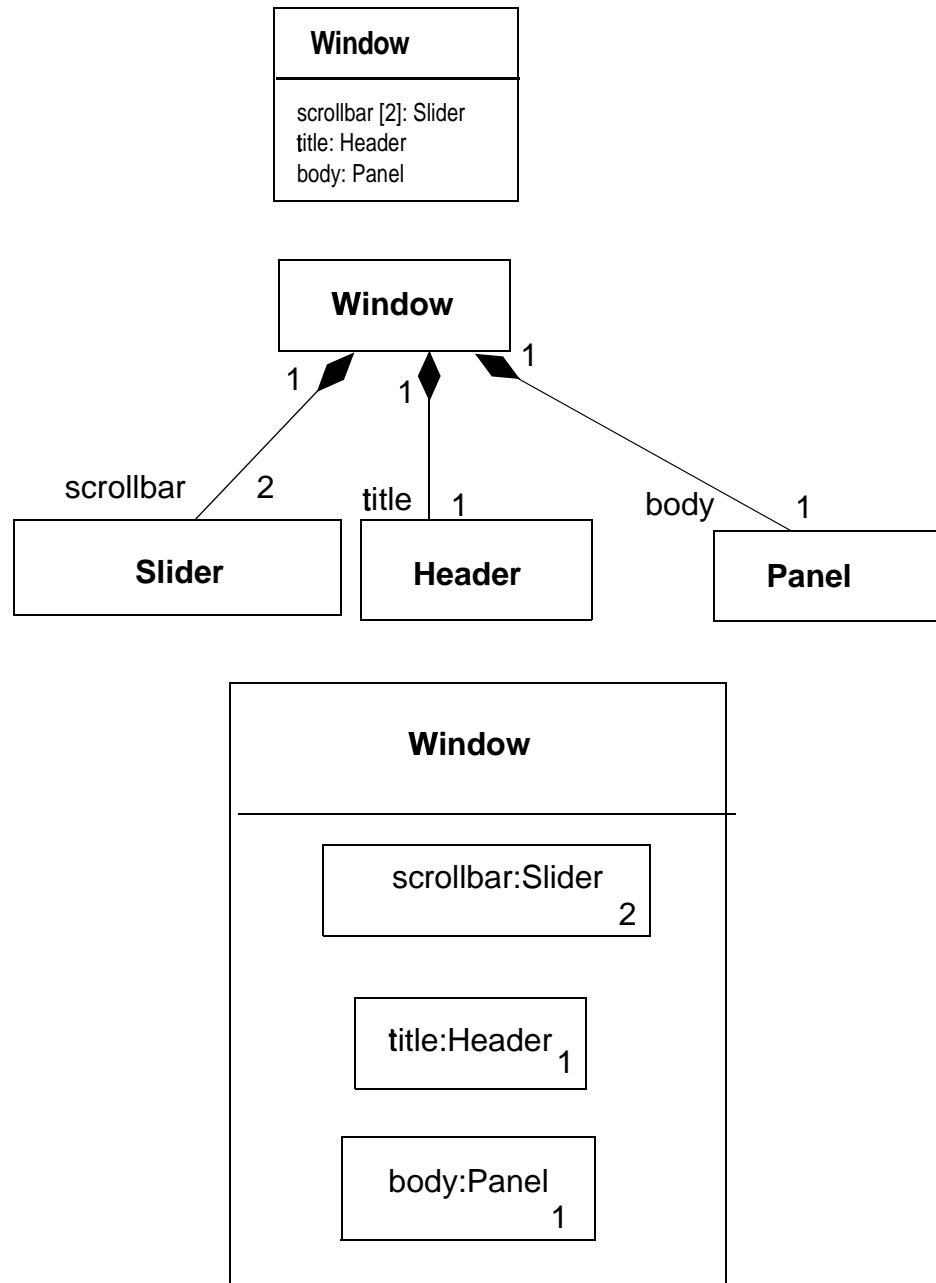scrollbar:Slider

2

title:Header   1

body:Panel

1

Figure 3-36 Different Ways to Show Composition

3.46.5 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.

3.47 Link

3.47.1 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.

3.47.2 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 Section 3.39.5, "Association," on page 3-61 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.

3.47.2.1 *Implementation stereotypes*

A stereotype may be attached to the link end to indicate various kinds of implementation. The follo wing stereotyp es ma y be u sed:

«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)

3.47.2.2 *N-ary link*

An n-ary link is shown as a diamond with a path to each participating instance. The other adornments o n the assoc iation, and the adornments on the associa tion end s, have the same po ssibilities as th e bin ary li nk.

3.47.3 *Example*
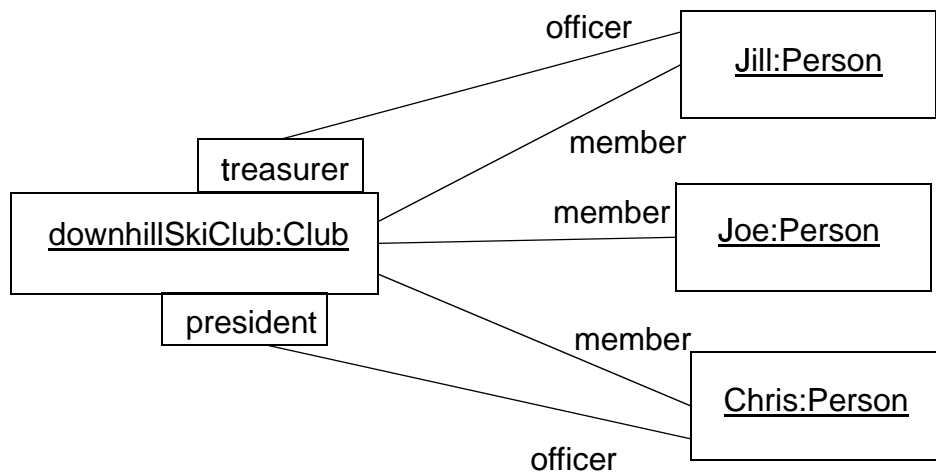
Figure 3-37 Links

3.47.4 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).

3.48 Generalization

3.48.1 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.

3.48.2 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 classes, although the notation may be messy because of the multiple lines. An association can be shown as an association class for the purpose of attaching generalization arrows.

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.

3.48.3 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.
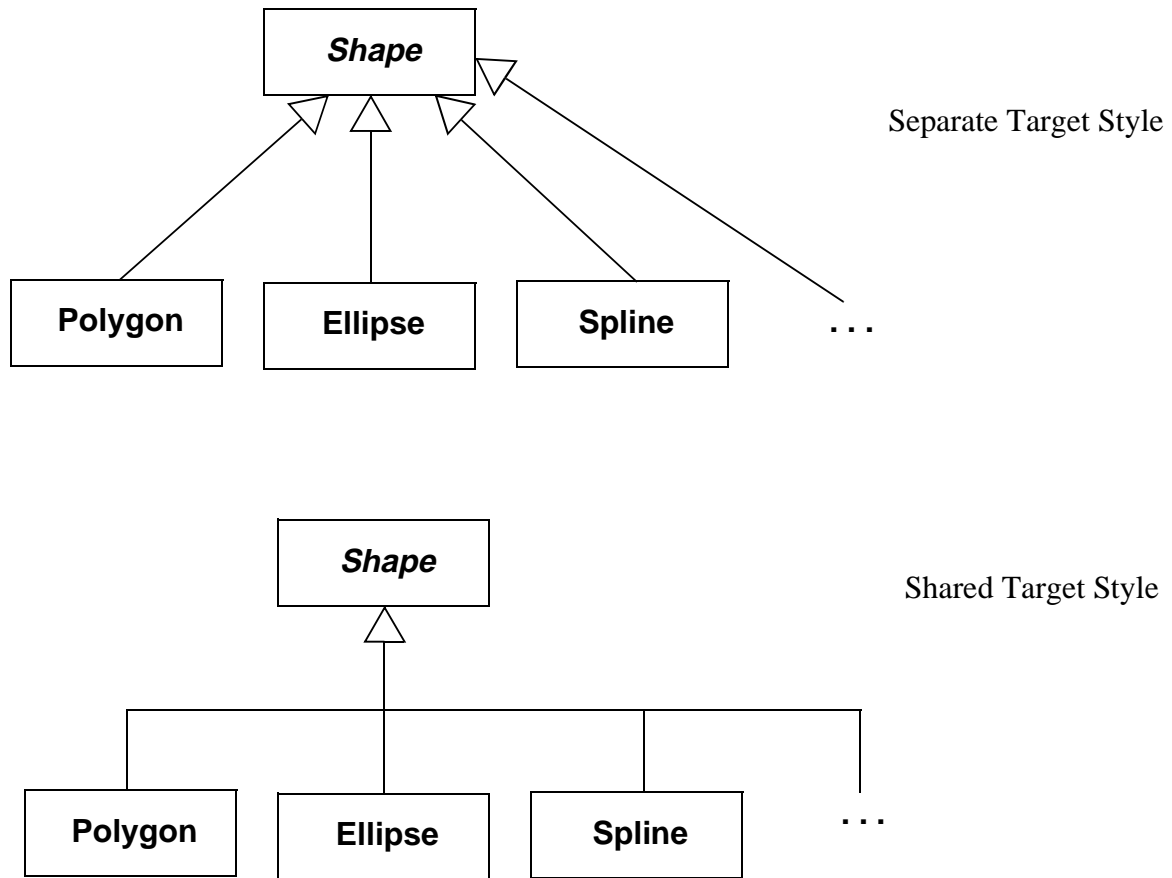
3.48.4 Example

Shape

Polygon   Ellipse   Spline   . . .

Separate Target Style

Shape

Polygon   Ellipse   Spline   . . .

Shared Target Style

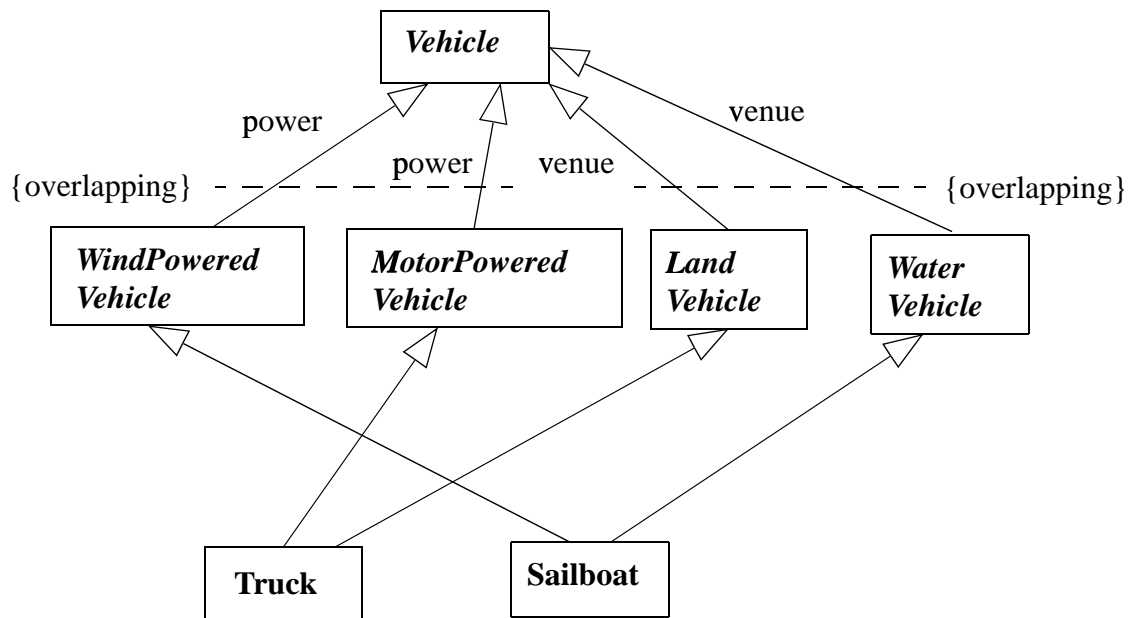*Figure 3-38* Styles of Displaying Generalizations

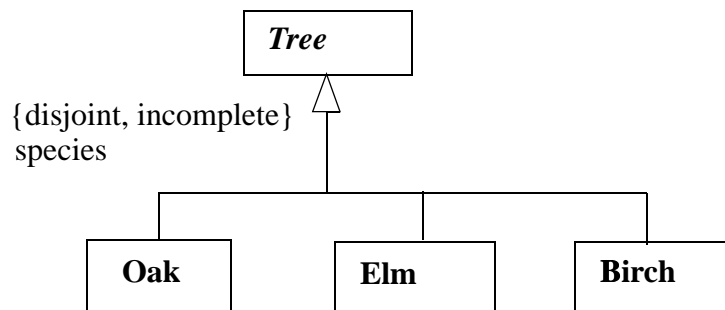*Figure 3-39* Generalization with Discriminators and Constraints, Separate Target Style

Figure 3-40 Generalization with Shared Target Style

3.48.5 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.

3.49   Dependency

3.49.1   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.

3.49.2   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.

.

Table 3-1   Keyword Descriptions

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 3.30, "Bound Element," on page 3-50 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 represent 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, and instantiating an object of another class. 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.

3.49.3 Presentation Options

The connection between a note or constraint and the element it applies to is shown by a dashed line with out an arrowhead. This is not a Dependency.
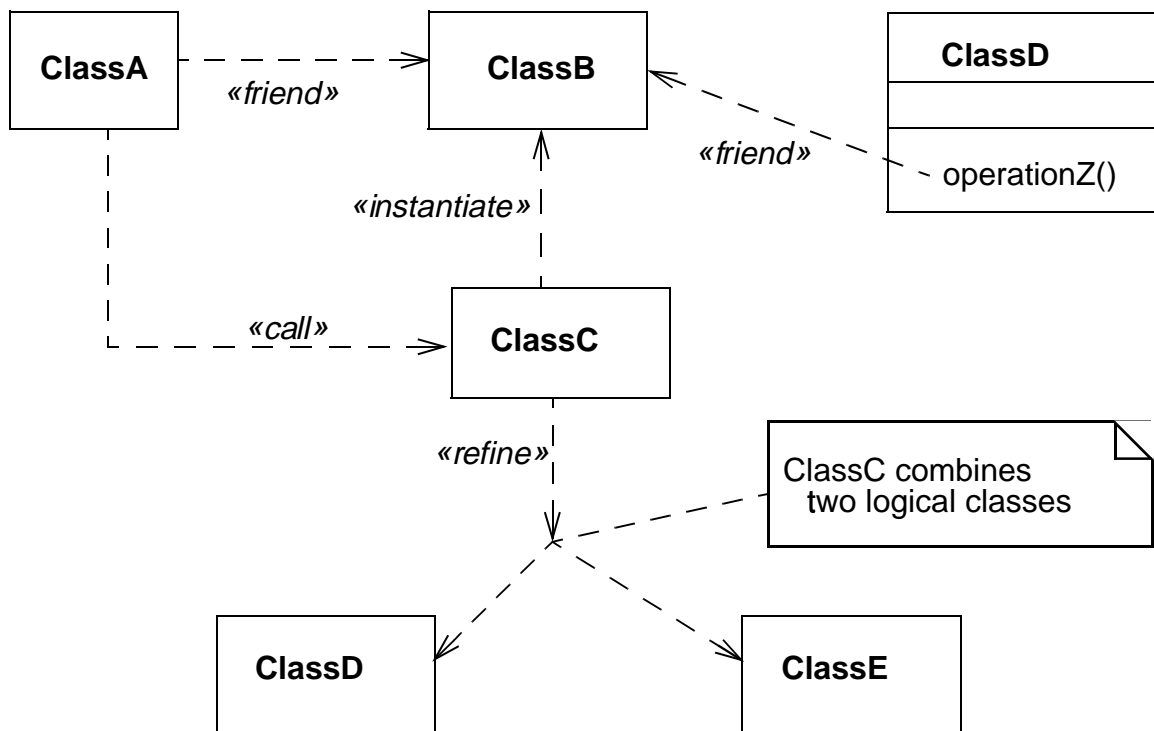
3.49.4 Example

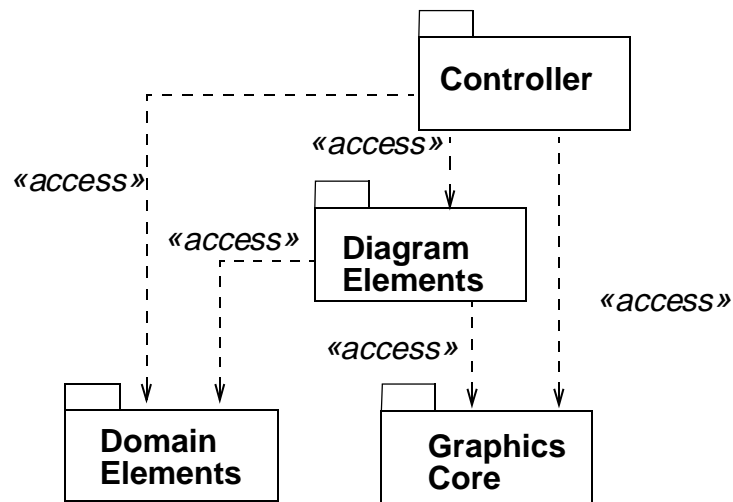*Figure 3-41* Various Dependencies Among Classes

Figure 3-42 Dependencies Among Packages

3.49.5 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.

3.50   Derived Element

3.50.1 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.

3.50.2 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.

3.50.3 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.
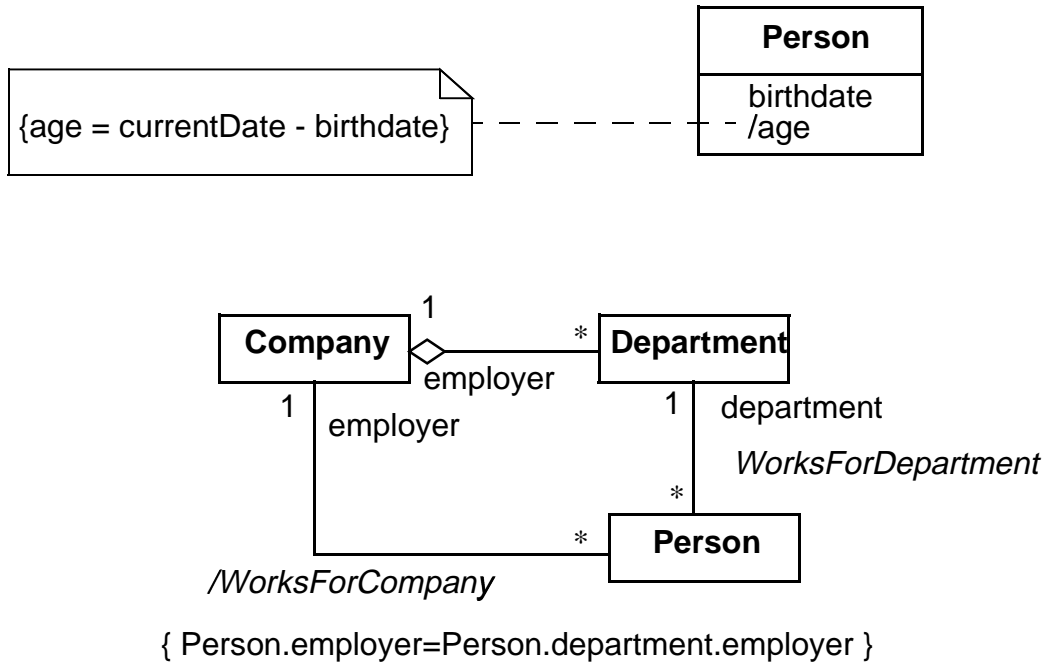
3.50.4 Example

Figure 3-43 Derived Attribute and Derived Association

3.50.5 Mapping

The presence of a derived adornment (a leading "/" on the symbol name) on a symbol maps into the attachment of the "derived" tag to the corresponding Element.

3.51 InstanceOf

3.51.1 Semantics

Shows the connection between an instance and its classifier.

3.51.2 Notation

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

3.51.3 Mapping

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