UML Notation Guide

Chapter 4

■

4. STATIC STRUCTURE DIAGRAMS

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

This chapter includes classes and their variations, including templates and instantiated classes, and the relationships between classes: association and generalization. It includes the contents of classes: attributes and operations. It also includes the organizational unit of class diagrams: packages.

4.1 Class diagram

A class diagram is a graph of modeling elements shown on a two-dimensional surface. (Note that a "class" diagram may also contain types, packages, relationships, and even instances, such as objects and links. Perhaps a better name would be "static structural diagram" but "class diagram" sounds better.)

4.1.1 Notation

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

4.2 Object diagram

An object diagram is a graph of instances. A static object diagram is an instance of a class diagram; it shows a snapshot of the detailed state of a system at a point in time. A dynamic object diagram shows the detailed state of a system over some period of time, including the changes that occur over time; dynamic object diagrams are manifested as collaboration diagrams.

There is no need for tools to 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." Collaboration diagrams contain objects, so a collaboration diagram with no messages is an "object diagram." The phrase is useful, however, to characterize a particular usage achievable in various ways.

4.3 Class

A class is the descriptor for a set of objects with similar structure, behavior, and relationships. 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 types, signals, or utilities) are notated as stereotypes of classes. Classes are declared in class diagrams and used in most other diagrams. UML provides a graphical notation for declaring and using classes, as well as a textual notation for referencing classes within the descriptions of other model elements.

4.3.1 Semantics

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.

4.3.2 Basic notation

A class is drawn as a solid-outline rectangle with 3 compartments separated by horizontal lines. The top name compartment holds the class name and other general properties of the class (including stereotype); the middle list compartment holds a list of attributes; the bottom list compartment holds a list of operations.

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

> *Package-name::Class-name*

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

4.3.3 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.

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, and so on. Most compartments are simply lists of strings. More complicated formats are possible, but UML does not specify such formats; they are a tool responsibility. Appearance of each compartment should preferably be implicit based on its contents. Tools may provide explicit markers if needed.

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

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

4.3.4 Style guidelines

Class name in boldface, centered.

Stereotype name in plain face, within guillemets, centered.

Typically class names begin with an uppercase letter.

Attributes and operations in plain face, left justified.

Typically attribute and operation names begin with a lowercase letter.

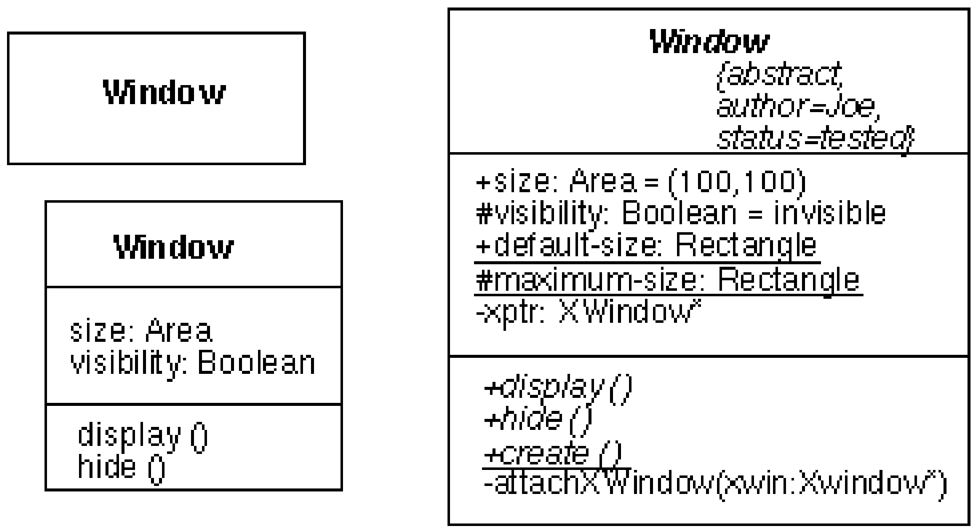
As a tool extension, boldface may be used for marking special list elements, for example, to designate candidate keys in a database design. This might encode some design property modeled as a tagged value, for example.

Strings for the names of abstract classes or the signatures of abstract operations in italics.

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

4.3.5 Example

Figure 6. Class notation: details suppressed, analysis-level details, implementation-level details



4.4 Name Compartment

4.4.1 Notation

Displays the name of the class and other properties in up to 3 sections:

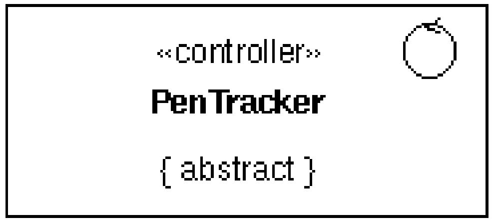
An optional stereotype keyword may be placed above the class name within guillemets, and/or a stereotype icon may be placed in the upper right corner of the compartment.

The name of the class appears next. (Style: centered, leading capital, boldface.)

A property list 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 stereotype and property list are optional.

Figure 7. Name compartment



4.5 List Compartment

4.5.1 Notation

Holds a list of strings, each of which is the encoded representation of an element, 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, lists can show other kinds of predefined or user-defined values, such as responsibilities, rules, or modification histories. 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, however, to achieve some other purpose. For example, they may be sorted in some way. Even if the list is sorted, however, 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 there exist additional elements in the model that meet the selection condition but that are not shown in that list. Such elements may appear in a different view of the list.

Group properties: A property string may be shown as a 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. Stereotype strings may also be used in a similar way to qualify subsequent list elements.

4.5.2 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), etc.

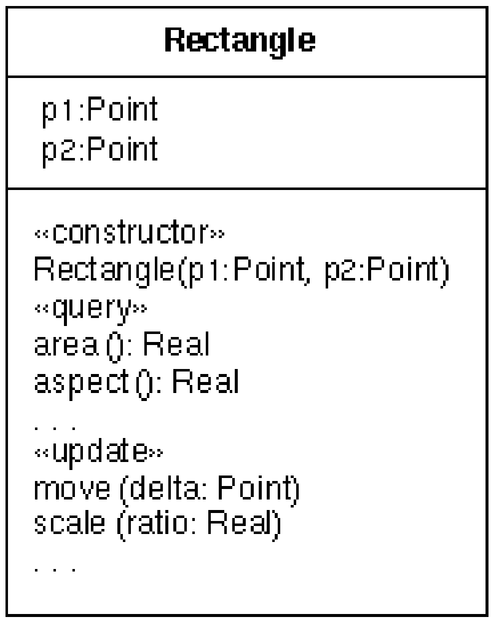
The elements in the list may be filtered according to some selection rule. The specification of selection rules is a tool responsibility. The absence of items from a filtered list indicates that no elements meet the filter criterion, but no inference can be drawn about the presence or absence of elements 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 20).

4.5.3 Example

Figure 8. Stereotype keyword applied to groups of list elements



4.6 Type

A type is descriptor for objects with abstract state, concrete external operation specifications, and no operation implementations. A class is a descriptor for objects with concrete state and concrete operation implementation.

Classes implement types. A type provides a specification of external behavior. A class provides an implementation data structure and a procedural implementation of methods that together implement the specified behavior.

4.6.1 Semantics

A type may contain attributes and operations, but neither of them represents an implementation commitment. Attributes in a type define the *abstract state* of the type. These represent the state information supported by objects of the type, but an actual class implementing the type may represent the information in a different way, as long as the representation maps to the abstract attributes of the type. Type attributes can be used to define the effects of type operations. A type may contain specifications for operations, including their signatures and a description of their effects, but the operations do not contain implementations. The effect of an operation is defined in terms of the changes it makes to the abstract attributes of the type.

It is sometimes helpful to describe abstract properties that represent structured information. For example, a type might contain a *PriceList* attribute that maps product names to money. The types of these attributes can be treated as mathematical functional mappings, such as **ProductName-->Money**.

A type establishes a behavioral specification for classes. A class that supports the operations defined by a type is said to *implement* the type; this relationship can be shown as a form of *refinement* relationship from the class to the type that it implements.

4.6.2 Notation

A type shown as a stereotype of a class symbol with the stereotype «type».

A type may contain lists of abstract attributes and of operations.

A type may contain a context and specifications of its operations accordingly.

4.7 Interfaces

An interface is the use of a type to describe the externally-visible behavior of a class, component, or other entity (including summarization units such as packages).

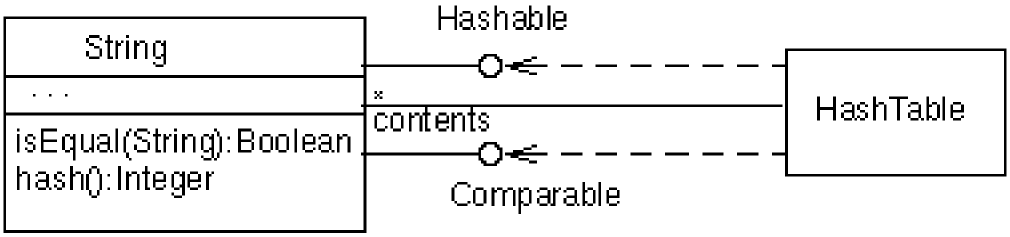
4.7.1 Notation

An interface may be displayed using a small circle with the name of the type. This notation stresses the operations provided by the type. An interface may supply one or more operations. The circle may be attached to classes (or higher-level containers, such as packages that contain the classes) that support it by a solid line. 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 requires the operations in the interface may be attached to the circle by a dashed arrow. The dashed arrow indicates a sufficiency test: if the type provides at least these operations then a class that realizes it will work. The dependent class is not required to actually use all of the operations.

An interface is a type and may also be shown using the full rectangle symbol with compartments. The circle form may be regarded as a shorthand notation.

4.7.2 Example

Figure 9. Interface notation on class diagram



4.8 Parameterized Class (Template)

4.8.1 Semantics

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

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

4.8.2 Notation

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

4.8.3 Presentation options

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

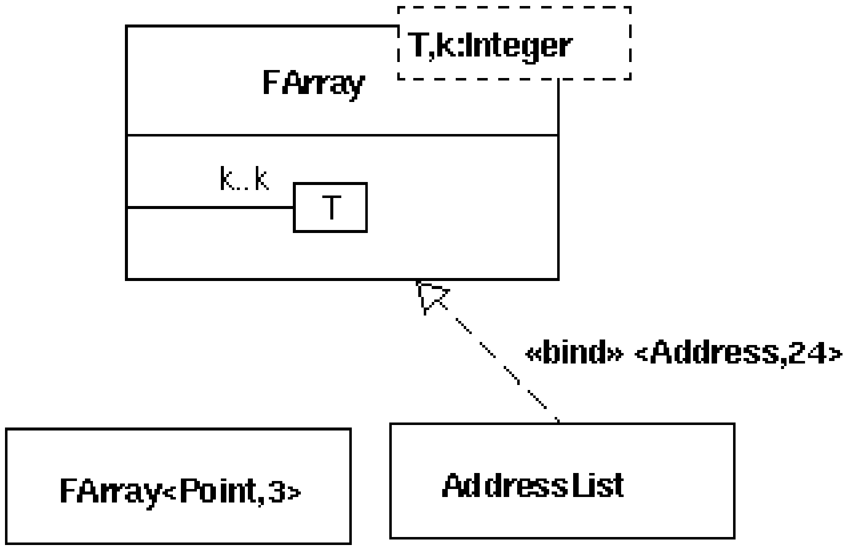
Parameters are restricted attributes, with the syntax

> *name : type*
>
> where *name* is an identifier for the parameter with scope inside the template;
>
> where *type* is a string designating a *TypeExpression* for the parameter.

The default type of a parameter is **TypeExpression** (or **class** as it is somewhat confusingly declared in C++, even though they allow **int's** and other non-classes). If the type name is omitted, it is assumed to be **TypeExpression** (that is, the argument itself must be an implementation type, such as a class name). Other parameter types (such as **Integer**) should be explicitly shown.

4.8.4 Example

Figure 10. Template notation with use of parameter as a reference



4.9 Bound Element

4.9.1 Semantics

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

4.9.2 Notation

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

> *Template-name* '<' *value-list* '>'
>
> where *value-list* is a comma-delimited non-empty list of value expressions;
>
> where *Template-name* is identical to the name of a template.

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

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

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

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

The relationship between the bound element and its template may alternatively be shown by a refinement relationship with the stereotype «bind». The arguments are shown on the relationship. In this case the bound form may be given a name distinct from the template.

4.9.3 Style guidelines

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

4.9.4 Example

See Figure 10.

4.10 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.

4.10.1 Semantics

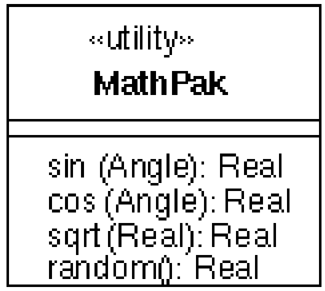
The instance-scope attributes and operations of a utility are interpreted as global attributes and operations. It is inappropriate for a utility to declare class-scope attributes and operations because the instance-scope members are already interpreted as being at class scope.

4.10.2 Notation

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

4.10.3 Example

Figure 11. Notation for utility



4.11 Metaclass

4.11.1 Semantics

A metaclass is a class whose instances are classes.

4.11.2 Notation

Shown as the stereotype «metaclass» of Class.

4.12 Class Pathnames

4.12.1 Notation

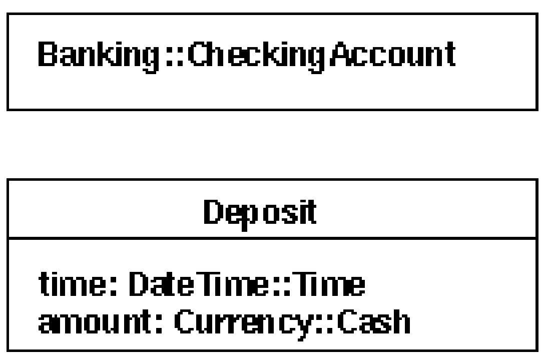
Class symbols (rectangles) serve to define a class and its properties, such as relationships to other classes. A reference to a class in a different package is notated by using a pathname for the class, in the form:

> *package-name :: class-name*

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

4.12.2 Example

Figure 12. Pathnames for classes in other packages



4.13 Importing a package

4.13.1 Semantics

A class in another package may be referenced. On the package level, the \xc7 imports\xc8 dependency shows the packages whose classes may be referenced within a given package or packages recursively embedded within it. The target references must be exported by the target package. Note that exports are not recursive; they must be propagated up across each level of containment. Imports are recursive within inner levels of containment. (See the semantics document for full details.)

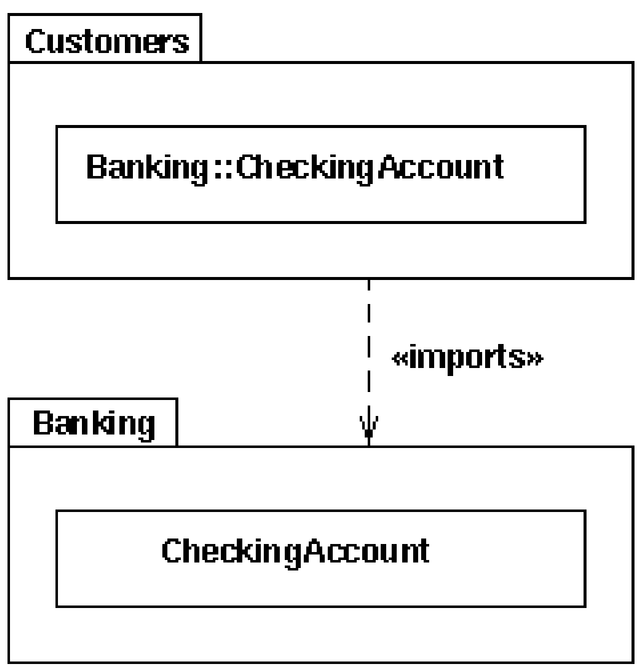
4.13.2 Notation

The imports dependency is displayed as a dependency arrow from the referencing package to the package containing the target of the references. The arrow has the stereotype «imports».

A package controls the external visibility of its contents. An item can be imported into package if it is made visible ("exported") by its declaring package. There is no special UML notation for the visibility of items within a package. Rather a view can be constructed showing the publicly available items from a package.

4.13.3 Example

Figure 13. Imports dependency among packages



4.14 Attribute

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).

4.14.1 Semantics

Note that an attribute is semantically equivalent to a composition association.

The type of an attribute may be complex, such as **array[String] of Point**. In some specification languages, it may also be expressed as a mapping expressed without a specific commitment to data structure, such as **String -->Point** (where the arrow represents the standard mathematical concept of functional mapping). This form expresses what D'Souza calls "parameterized queries" using the syntax **location(String):Point** in his Catalysis method. 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.

4.14.2 Notation

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

> *visibility name* : *type-expression* = *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). 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.
> >
> > 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 *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 entire string. 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.

class-scope-attribute

4.14.3 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 List Compartment).

4.14.4 Style guidelines

Attribute names typically begin with a lowercase letter.

Attribute names in plain face.

4.14.5 Example

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

4.15 Operation

Used to show operations in classes.

4.15.1 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). 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.

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 of the value returned by the operation. If the return-type is omitted if the operation does not return a value (C++ **void**);

where *parameter-list* is a comma-separated list of formal parameters, each specified using the syntax:

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

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 entire string. An instance-scope operation is the default and is not marked.

4.15.2 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.

4.15.3 Style guidelines

Attribute names typically begin with a lowercase letter.

Attribute names in plain face.

An abstract operation may be shown in italics.

4.15.4 Example

Figure 14. Operation list with a variety of operations

4.16 Association

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

4.17 Binary Association

4.17.1 Notation

A binary association is drawn as a solid path connecting two class symbols (both ends may be connected to the same class, 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 sequences of segments is called a *path*.

The end of an association where it connects to a class is called an *association role*. Most of the interesting information about an association is attached to its roles. See the section on Association Role for details.

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 a role so that confusion does not occur. The following kinds of adornments may be attached to a path:

association name

Designates the (optional) name of the association.

Shown as a name string near the path. The 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 classes in the association are ordered as indicated by the name-direction arrow. (Note that there is no need for a *name direction* property on the association model; the ordering of the classes 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.

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.

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, but they are graphically distinct. The association class symbol can be dragged away from the line but the dotted line must remain attached to both the path and the class symbol.

4.17.2 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).

4.17.3 Style guidelines

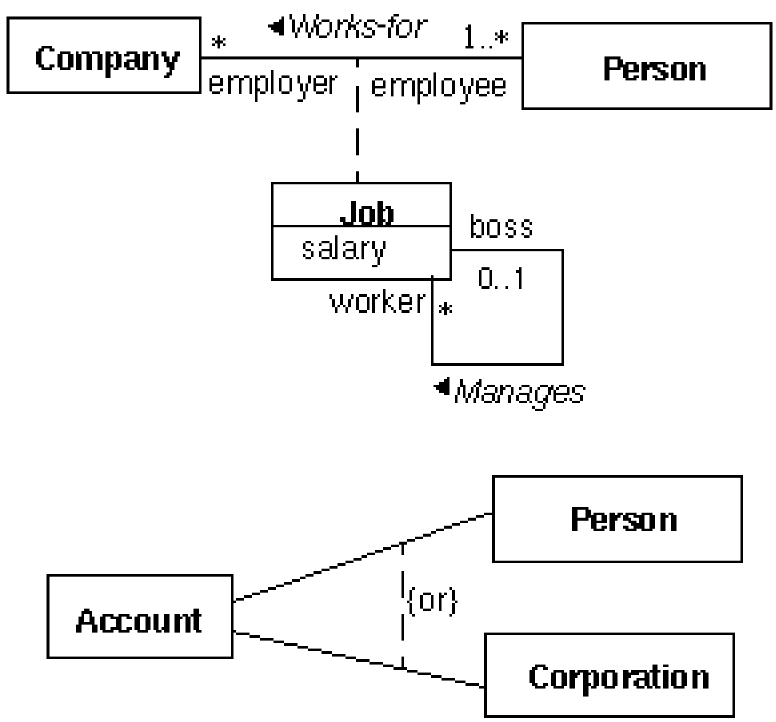
Lines may be drawn at any angle. One popular style is to draw straight paths between icons whenever possible. Another popular style is to have all lines be horizontal or vertical (orthogonal grid), using multiple segments to compose paths when necessary. In any case the user should be consistent.

4.17.4 Options

Or-association. An or-constraint indicates a situation in which only one of several potential associations may be instantiated at one time for any single object. This is shown as a dashed line connecting two or more associations, all of which must have a class in common, with the constraint string "{or}" labeling the dashed line. Any instance of the class may only participate in at most one of the associations at one time. (This is simply a particular use of the constraint notation.)

4.17.5 Example

Figure 15. Association notation



4.18 Association Role

An association role is simply an end of an association where it connects to a class. The role is part of the association, not part of the class. Each association has two or more roles. Most of the interesting details about an association are attached to its roles.

4.18.1 Notation

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

> multiplicity - see detail section. Multiplicity may be suppressed on a particular role or for an entire diagram. In an incomplete model the multiplicity may be unspecified in the model itself, in which case it must be suppressed in the notation.
>
> ordering - if the multiplicity is greater than one, then the set of related elements can be ordered or unordered. The default is unordered (they form a set). Various kinds of ordering can be specified as a constraint on the role. The declaration does not specify how

the ordering is established or maintained; operations that insert new elements must make provision for specifying their position either implicitly (such as at the end) or explicitly. Possible values include:

> unordered -- the elements form an unordered set. This is the default and need not be shown explicitly.
>
> ordered -- the elements are ordered into a list. This generic specification includes all kinds of ordering. This may be specified by a keyword constraint: "{ordered}".

An ordered relationship may be implemented in various ways but this is normally specified as a language-specified code generation property to select a particular implementation.

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

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

qualifier - see detail section. Qualifier is optional but not suppressible.

navigability

> An arrow may be attached to the end of the path to indicate that navigation is supported toward the class attached to the arrow. Arrows may be attached to zero, one, or two ends of the path. In principle arrows could be shown whenever navigation is supported in a given direction. In practice it is sometimes convenient to suppress some of the arrows and just show exceptional situations. Here are some options on showing navigation arrows:
>
> 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 options 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, but 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.

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*.

rolename

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

Other properties can be specified for association roles 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 such other properties include mutability.

4.18.2 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.

4.18.3 Style guidelines

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

qualifier

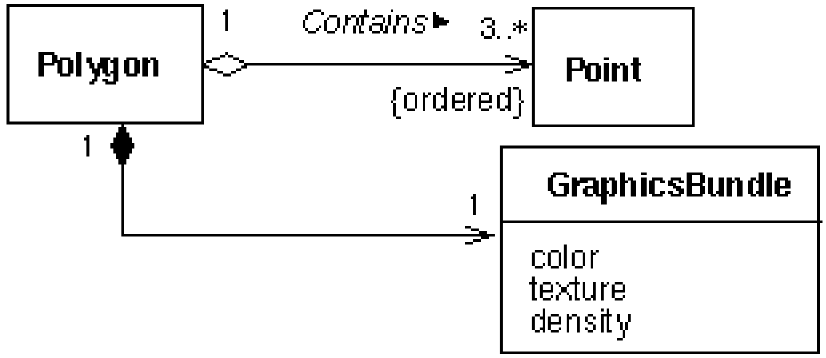
aggregation symbol

navigation arrow

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

4.18.4 Example

Figure 16. Various adornments on association roles



4.19 Multiplicity

A multiplicity string 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 is a subset of the nonnegative open integers.

4.19.1 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).

4.19.2 Style guidelines

Intervals should preferably 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".

4.19.3 Example

0..1

1

0..*

*

1..*

1..6

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

4.20 Qualifier

A qualifier is an association attribute or tuple of attributes whose values serve to partition the set of objects associated with an object across an association.

4.20.1 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 class that it connects to. The qualifier rectangle is part of the association path, not part of the class. The qualifier rectangle drags with the path segments. The qualifier is attached to the source end of the association; that is, an object of the source class together with a value of the qualifier uniquely select a partition in the set of target class objects on the other end of the association.

The multiplicity attached to the target role denotes the possible cardinalities of the set of target objects selected by the pairing of a source object and a qualifier value. Common values include "0..1" (a unique value may be selected, but every possible qualifier value does not necessarily select a value), "1" (every possible qualifier value selects a unique target object, therefore the domain of qualifier values must be finite), and "*" (the qualifier value is an index that partitions the target objects into subsets).

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

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

4.20.2 Presentation options

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

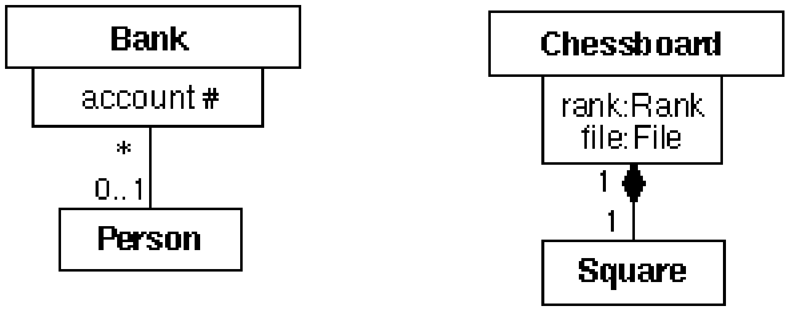
A tool may use a lighter line for qualifier rectangles than for class rectangles to distinguish them clearly.

4.20.3 Style guidelines

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

4.20.4 Example

Figure 17. Qualified associations



4.21 Association Class

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.

4.21.1 Notation

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

4.21.2 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.

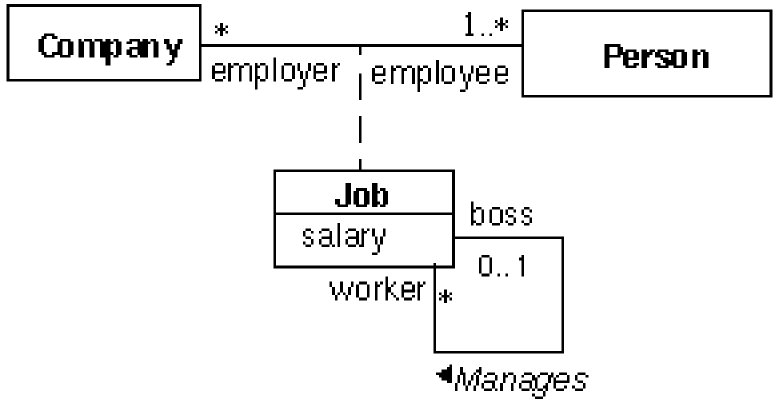
4.21.3 Style guidelines

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 role adornments.

Note that the association path and the association class are a single model element and therefore have a single name. The name can be shown on the path or 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.

4.21.4 Example

Figure 18. Association class



4.22 N-ary association

4.22.1 Semantics

An n-ary association is an association among 3 or more classes (a single class may appear more than once). Each instance of the association is an n-tuple of values from the respective classes. A binary association is a special case with its own notation.

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

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

4.22.2 Notation

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

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

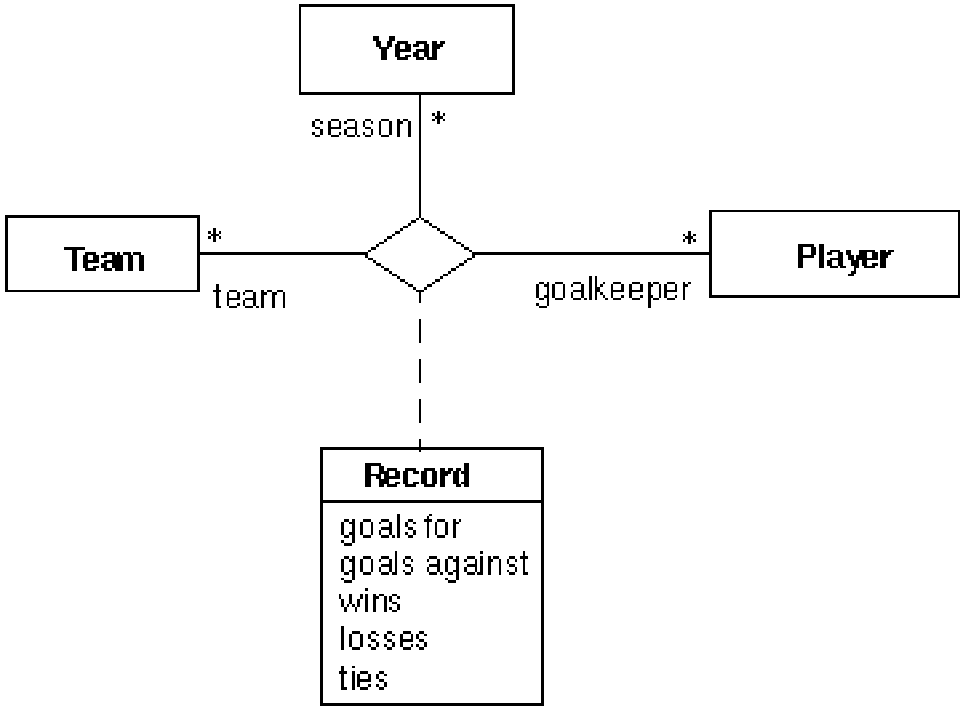
4.22.3 Style guidelines

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

4.22.4 Example

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

Figure 19. Ternary association that is also an association class



4.23 Composition

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). The aggregation is unchangeable (once established the links may not be changed). Parts with multiplicity > 1 may be created after the aggregate itself but once created they live and die with it. Such parts can also be explicitly removed before the death of the aggregate.

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

4.23.1 Semantics

Within a composite additional associations can be defined that are not meaningful within the system in general. These represent patterns of connection that are meaningful only within the context of the composite. Such associations can be thought of as generating *quasiclasses* (or *qua-types* as Bock and Odell call them) that are specializations of the general classes; the specializations are defined only inside the composite. In actual practice it often happens that one of the classes in the association does not know about the association or the other class, so that the implementation need not actually use the qua-class.

The entire system may be thought of as an implicit composite, so that any multiplicity specifications within top-level classes restrict the cardinality of the classes in a particular execution; Embley's singleton classes can be seen in that light.

4.23.2 Notation

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. A nested element may have a rolename within the composition; the name is shown in front of its type in the syntax:

rolename ':' classname

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.

Another alternative is to show the composite as a graphical symbol containing its parts, but to draw an association line from the composition symbol boundary to each of the parts within it. Rolenames and multiplicity of the parts may be indicated for each of the parts; using this notation it is unnecessary to display the aggregation diamond because the composition aggregation is specified by the nesting.

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

4.23.3 Design guidelines

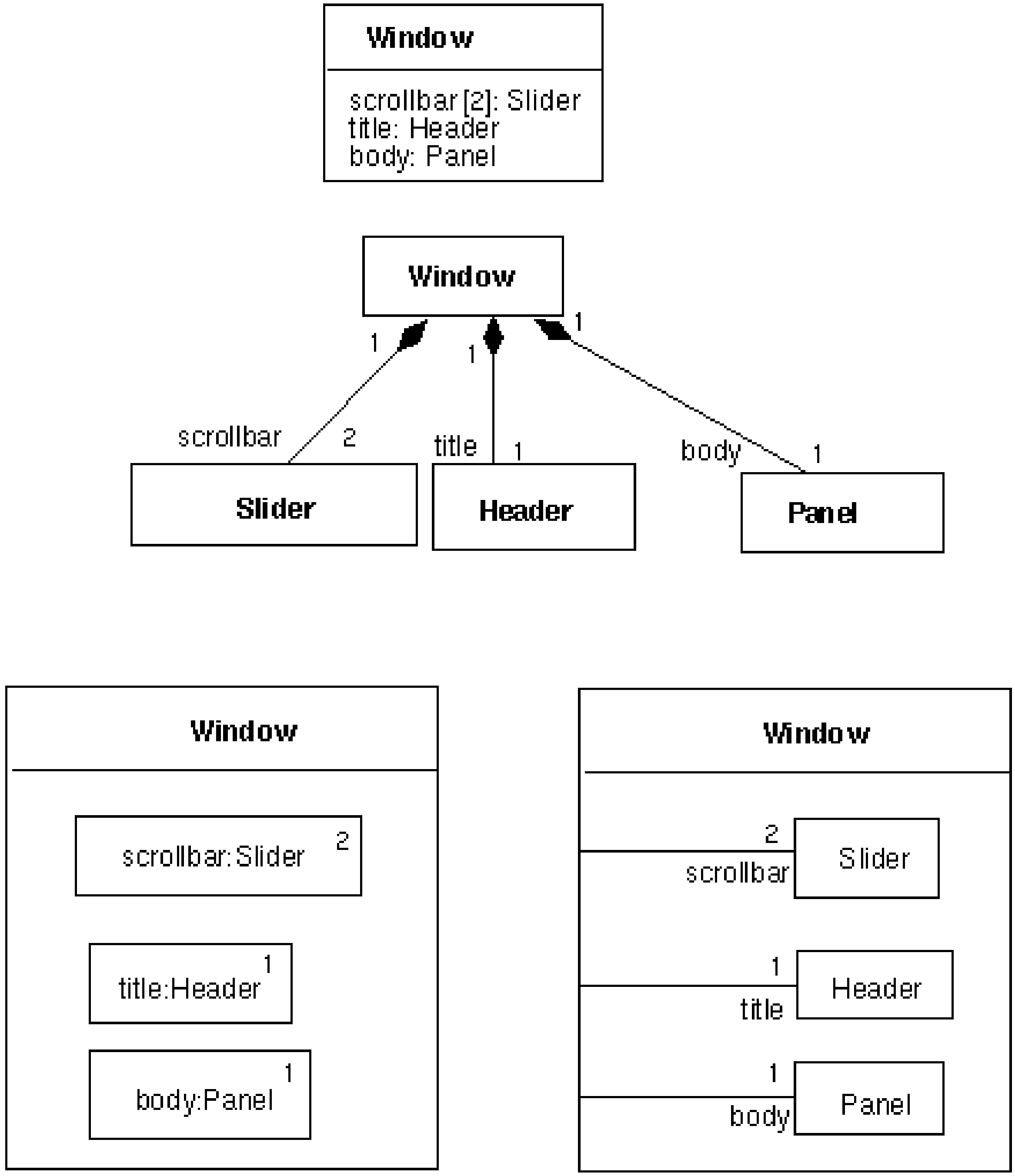
This notation is applicable to "class-like" model elements: classes, types, nodes, processes, etc.

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, so it should be used when the structure and identity of the attribute objects themselves is unimportant outside the class.

Be aware that state diagrams use different notation for composition than class diagrams. The composition of a state from two or more substates is shown by partitioning the state region into subregions by dashed lines. The simple nesting of states indicates state generalization.

4.23.4 Example

Figure 20. Different ways to show composition



4.24 Generalization

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

4.24.1 Notation

Generalization is shown as a solid-line path from the more specific element (such as a subclass) to 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 in the following format:

> discriminator : powertype
>
> where *discriminator* is the name of a partition of the subtypes of the supertype. The subtype is declared to be in the given partition;
>
> where *powertype* is the name of a type whose instances are subtypes of another type, namely the subtypes whose paths bear the powertype name. If a type symbol with the same name appears in the model, it designates the same type; it should be shown with the stereotype «powertype». For example, TreeSpecies is a powertype on the Tree type; consequently instances of TreeSpecies (such as Oak or Birch) are also subtypes of Tree.
>
> Either the discriminator, or the colon and powertype, or both may be omitted.

Note that the word *type* also includes both types and classes.

4.24.2 Presentation options

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

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

4.24.3 Details

The existence of additional subclasses in the model that are not shown on a particular diagram may be shown using an ellipsis (. . .) in place of a subclass. (Note: this does not indicate that additional classes may be added in the future. It indicates that additional classes exist right now but are not being seen.)

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

> overlapping
>
> disjoint
>
> complete
>
> incomplete

4.24.4 Semantics

The following constraints are predefined:

> overlapping A descendent may be descended from more than one of the subclasses.

disjoint A descendent may not be descended from more than one of the subclasses.

complete All subclasses have been specified (whether or not shown). No additional subclasses are expected.

incomplete Some subclasses have been specified but the list is known to be incomplete. There are additional subclasses that are not yet in the model. The is a statement about the model itself. Note that this is not the same as the ellipsis, which states that additional subclasses 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 superclass. Multiple occurrences of the same discriminator name are permitted and indicate that the subclasses belong to the same partition.

Semantic variation points

There are different possible ways to interpret the semantics of generalization (as with other constructs). Although there is a standard UML interpretation consistent with the operation of the major object-oriented languages, there are purposes and languages that require a different interpretation. Different semantics can be permitted by identifying *semantic variation points* and giving them names, so that different users and tools could understand the variation being used (it is not assumed that all tools will support this concepts). These are some semantic variations applicable to generalization:

Multiple inheritance. Whether a class may have more than one superclass.

Multiple classification. Whether an object may belong directly to more than one class.

Dynamic classification. Whether an object may change class during execution.

The ordinary UML semantics assumes multiple inheritance, no multiple classification, and no dynamic classification, but most parts of the semantics and notation are not affected if these assumptions are change.

4.24.5 Example

Figure 21. Styles of displaying generalization

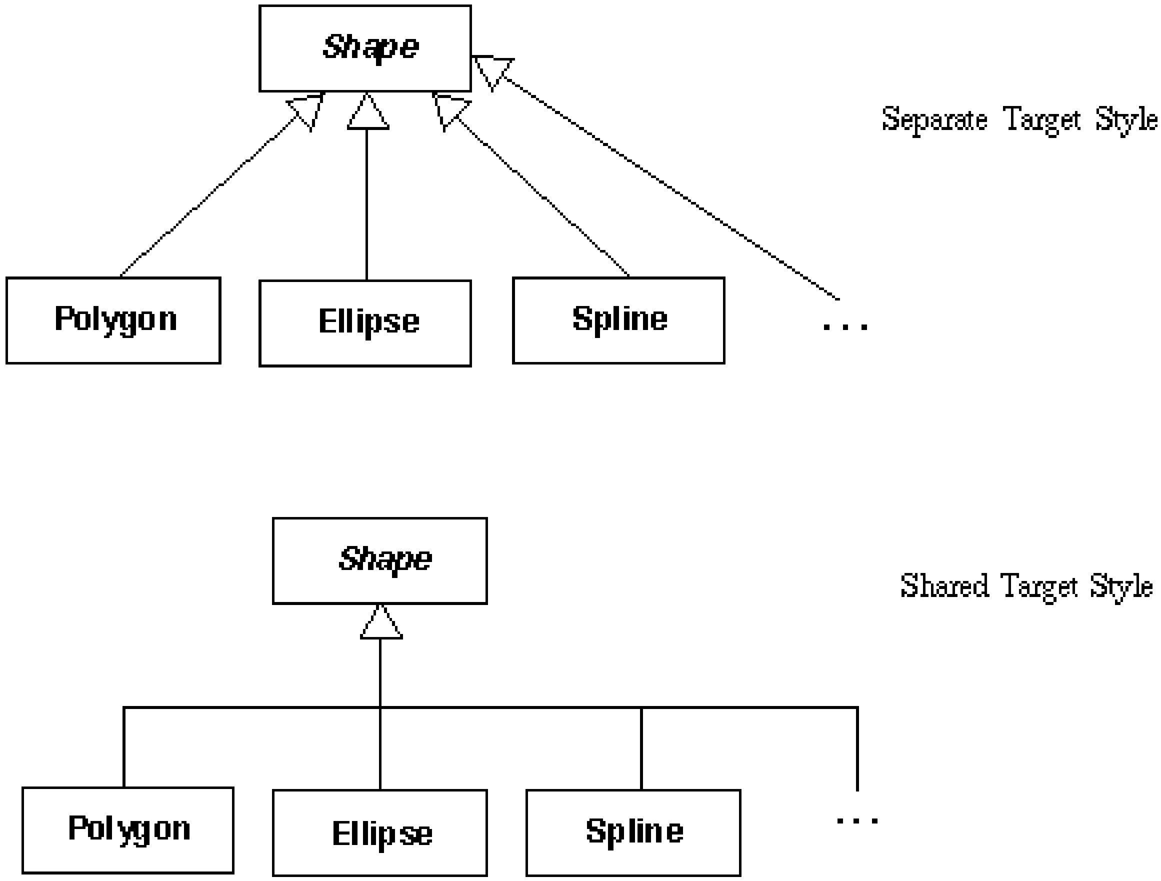


Figure 22. Generalization with discriminators and constraints, separate target style

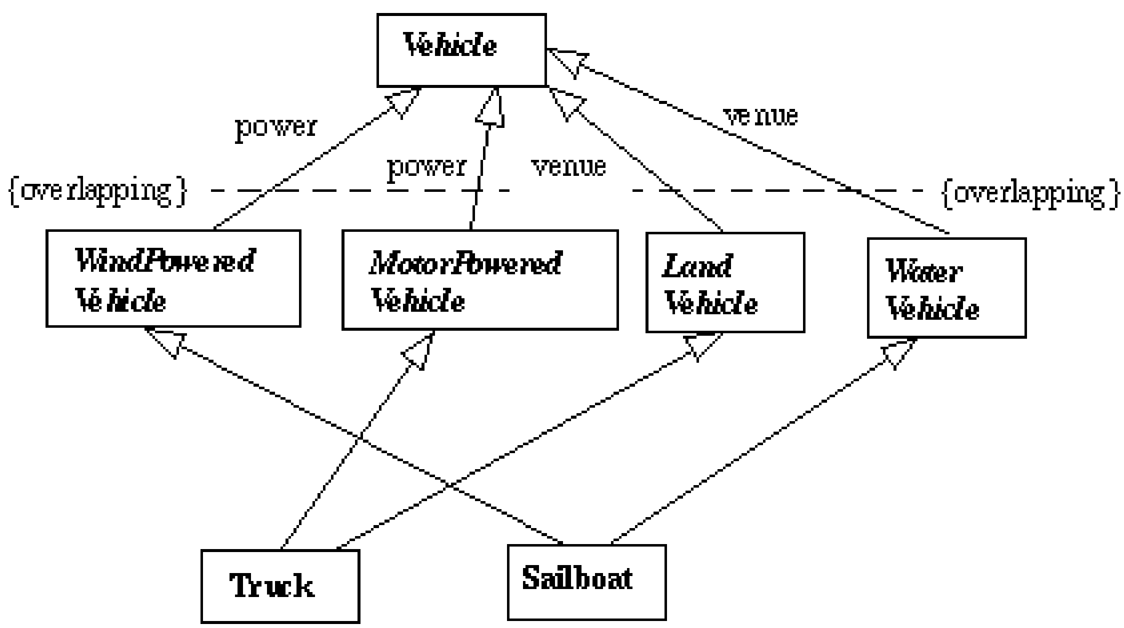
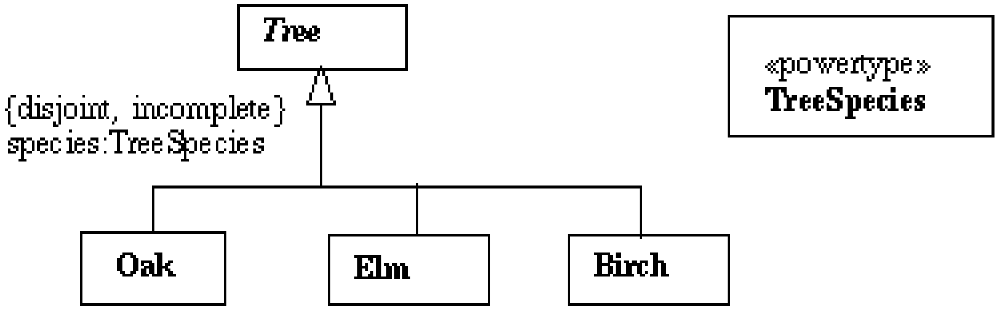


Figure 23. Generalization with power type, shared target style



4.25 Dependency

A dependency indicates a semantic relationship between two (or more) 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.

4.25.1 Notation

A dependency is shown as a dashed arrow from one model element to another model element that the first element is dependent on. The arrow may be labeled with an optional stereotype and an optional name.

4.25.2 Presentation options

If one of the elements is a note or constraint then the arrow may be suppressed (the note or constraint is the source of the arrow).

4.25.3 Example

Figure 24. Various dependencies among classes

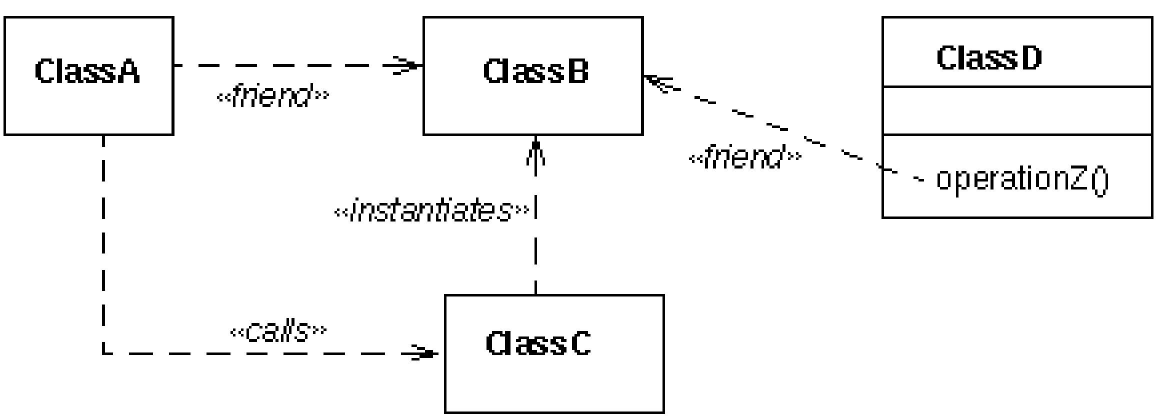
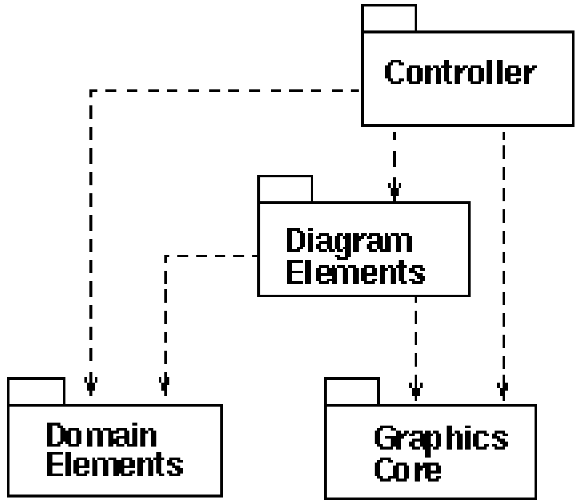


Figure 25. Dependencies among packages



4.26 Refinement Relationship

4.26.1 Semantics

The refinement relationship represents the fuller specification of something that has been already specified at a certain level of detail. It is a commitment to certain choices consistent with the more general specification but not required by it. It is a relationship between two descriptions of the same thing at different levels of abstraction.

The evolution of a design may be described by refinement relationships. entire process of design is a process of refinement. Note that refinement is a relationship between development artifacts and does not imply any top-down development process. Refinement includes the following kinds of things (not necessarily complete):

> Relation between a type and a class that realizes it (realization).
>
> Relation between an analysis class and a design class (design trace).
>
> Relation between a high-level construct at a coarse granularity and a lower-level construct at a finer granularity, such as a collaboration at two levels of detail (leveling of detail).
>
> Relation between a construct and its implementation at a lower virtual layer, such as the implementation of a type as a collaboration of lower-level objects (implementation).
>
> Relation between a straightforward implementation of a construct and a more efficient but more obscure implementation that accomplishes the same effect (optimization).

Note that refinement shows a relationship between two different views of something. You can use either view but they are alternate ways of expressing the same thing under different conditions. Examples include the relationship between an analysis type and a design class, between a scenario at a high level and the same scenario broken into finer steps, and between a simple implementation of an operation and an optimized implementation of the same operation.

A refinement relationship may also have a specification of how the more detailed version maps into the more abstract version.

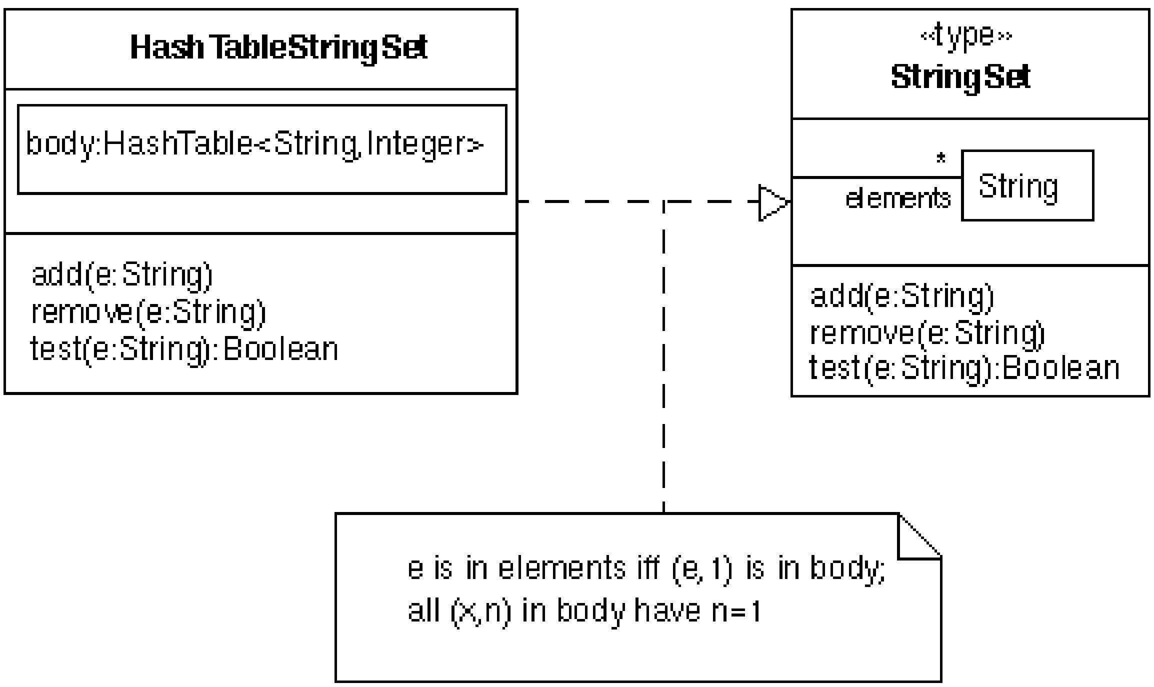
4.26.2 Notation

Refinement may be shown as a dashed generalization symbol, that is, a dashed line with a closed hollow triangular arrowhead on the end connected to more general element. A stereotype may be attached to specify a particular kind of refinement. A note may be attached to the line stating the mapping from the more specific form to the more general form.

Refinement *within* a given model can be shown as a dependency with the stereotype «refines» or one of its more specific forms, such as «implements». Refinement *between* models may be modeled as an invisible hyperlink supported by a dynamic tool. The refinement relationship may have a mapping attached to it; the mapping will normally be reached via an invisible hyperlink from the relationship path.

4.26.3 Example

Figure 26. Refinement



4.27 Derived Element

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.

4.27.1 Notation

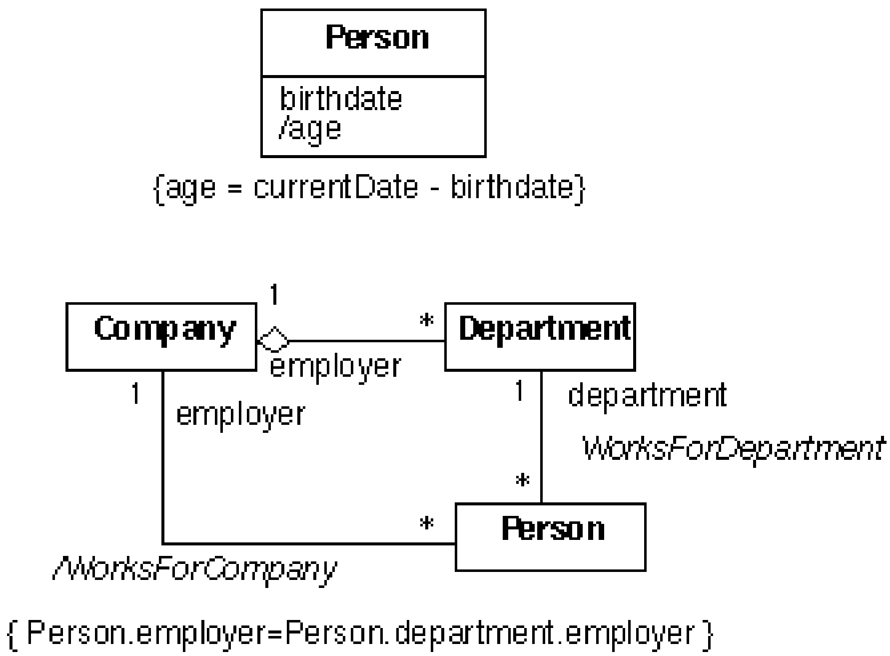
A derived element is shown by placing a slash (/) in front of the name of the derived element, such as an attribute or a rolename.

4.27.2 Style guidelines

The details of computing a derived element can be specified by a dependency with the stereotype «derived». 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.

4.27.3 Example

Figure 27. Derived attribute and derived association



4.28 Navigation Expression

UML notation provides a small language for expressing navigation paths in class models.

4.28.1 Notation

These forms can be chained together. The leftmost element must be an expression for an object or a set of objects. The expressions are meant to work on sets of values when applicable.

set '.' *selector*

> the *selector* is the name of an attribute in the objects of the set or the name of a role of the target end of a link attached to the objects in the set. The result is the value of the attribute or the related object(s). The result is a value or a set of values depending on the multiplicity of the set and the association.

set '.' '~' *selector*

the *selector* is the name of a role on the source end of an association attached to the *set* of objects. The result is the object(s) attached to the other side. This represents an inverse relationships, that is, the use of the rolename in the "wrong way."

set '[' *boolean-expression* ']'

the *boolean-expression* is written in terms of objects within the set and values accessible from them. The result is the subset of objects for which the boolean expression is true.

set '.' *selector* '[' *qualifier-value* ']'

the *selector* designates a qualified association that qualifies the *set*. The *qualifier-value* is a value for the qualifier attribute. The result is the related object selected by the qualifier. Note that this syntax is applicable to array indexing as a form of qualification.

4.28.2 Example

flight.pilot.training_hours > flight.plane.minimum_hours

company.employee [title = "Manager" and count (employee) > 10]