UML Notation Guide 3

This guide describes the notation for the visual representation of the Unified Modeling Language (UML). This notation document contains brief summaries of the semantics of UML constructs, but the UML Semantics chapter must be consulted for full details.

Contents

This chapter contains the following topics.

Part 1 - Background

3.1 Introduction

This chapter is arranged in parts according to semantic concepts subdivided by diagram types. Within each diagram type, model elements that are found on that diagram and their representation are listed. Note that many model elements are usable in more than one diagram. An attempt has been made to place each description where it is used the most, but be aware that the document involves implicit cross-references and that elements may be useful in places other than the section in which they are described. Be aware also that the document is nonlinear: there are forward references in it. It is not intended to be a teaching document that can be read linearly, but a reference document organized by affinity of concept.

Each part of this chapter is divided into sections, roughly corresponding to important model elements and notational constructs. Note that some of these constructs are used within other constructs; do not be misled by the flattened structure of the chapter. Within each section the following subsections may be found:

- Semantics: Brief summary of semantics. For a fuller explanation and discussion of fine points, see the *UML Semantics* chapter in this specification.
- Notation: Explains the notational representation of the semantic concept (“forward mapping to notation”).
- Presentation options: Describes various options in presenting the model information, such as the ability to suppress or filter information, alternate ways of showing things, and suggestions for alternate ways of presenting information within a tool.

  Dynamic tools need the freedom to present information in various ways and the authors do not want to restrict this excessively. In some sense, we are defining the “canonical notation” that printed documents show, rather than the “screen notation.” The ability to extend the notation can lead to unintelligible dialects, so we hope this freedom will be used in intuitive ways. The authors have not sought to eliminate all the ambiguity that some of these presentation options may introduce, because the presence of the underlying model in a dynamic tool serves to easily disambiguate things. Note that a tool is not supposed to pick just one of the presentation options and implement it. Tools should offer users the options of selecting among various presentation options, including some that are not described in this document.
- Style guidelines: Include suggestions for the use of stylistic markers, such as fonts, naming conventions, arrangement of symbols that are not explicitly part of the notation, but that help to make diagrams more readable. These are similar to text indentation rules in C++ or Smalltalk. Not everyone will choose to follow these suggestions, but the use of some consistent guidelines of your own choosing is recommended in any case.
- Example: Shows samples of the notation. String and code examples are given in the following font: `This is a string sample.`

- Mapping: Shows the mapping of notation elements to metamodel elements (“reverse mapping from notation”). This indicates how the notation would be represented as semantic information. Note that, in general, diagrams are interpreted in a particular context in which semantic and graphic information is gathered simultaneously. The assumption is that diagrams are constructed by an editing tool that internalizes the model as the diagram is constructed. Some semantic constructs have no graphic notation and would be shown to a user within a tool using a form or table.

Part 2 - Diagram Elements

3.2 Graphs and Their Contents

Most UML diagrams and some complex symbols are graphs containing nodes connected by paths. The information is mostly in the topology, not in the size or placement of the symbols (there are some exceptions, such as a sequence diagram with a metric time axis). There are three kinds of visual relationships that are important:

1. connection (usually of lines to 2-d shapes),
2. containment (of symbols by 2-d shapes with boundaries), and
3. visual attachment (one symbol being “near” another one on a diagram).

These visual relationships map into connections of nodes in a graph, the parsed form of the notation.

UML notation is intended to be drawn on 2-dimensional surfaces. Some shapes are 2-dimensional projections of 3-d shapes (such as cubes), but they are still rendered as icons on a 2-dimensional surface. In the near future, true 3-dimensional layout and navigation may be possible on desktop machines; however, it is not currently practical.

There are basically four kinds of graphical constructs that are used in UML notation:

1. Icons - An icon is a graphical figure of a fixed size and shape. It does not expand to hold contents. Icons may appear within area symbols, as terminators on paths or as standalone symbols that may or may not be connected to paths.
2. 2-d Symbols - Two-dimensional symbols have variable height and width and they can expand to hold other things, such as lists of strings or other symbols. Many of them are divided into compartments of similar or different kinds. Paths are connected to two-dimensional symbols by terminating the path on the boundary of the symbol. Dragging or deleting a 2-d symbol affects its contents and any paths connected to it.
3. Paths - Sequences of line segments whose endpoints are attached. Conceptually a path is a single topological entity, although its segments may be manipulated graphically. A segment may not exist apart from its path. Paths are always attached to other graphic symbols at both ends (no dangling lines). Paths may have *terminators*; that is, icons that appear in some sequence on the end of the path and that qualify the meaning of the path symbol.

4. Strings - Present various kinds of information in an “unparsed” form. UML assumes that each usage of a string in the notation has a syntax by which it can be parsed into underlying model information. For example, syntaxes are given for attributes, operations, and transitions. These syntaxes are subject to extension by tools as a presentation option. Strings may exist as singular elements of symbols or compartments of symbols, as elements in lists (in which case the position in the list conveys information), as labels attached to symbols or paths, or as stand-alone elements on a diagram.

3.3 Drawing Paths

A path consists of a series of line segments whose endpoints coincide. The entire path is a single topological unit. Line segments may be orthogonal lines, oblique lines, or curved lines. Certain common styles of drawing lines exist: all orthogonal lines, or all straight lines, or curves only for bevels. The line style can be regarded as a tool restriction on default line input. When line segments cross, it may be difficult to know which visual piece goes with which other piece; therefore, a crossing may optionally be shown with a small semicircular jog by one of the segments to indicate that the paths do not intersect or connect (as in an electrical circuit diagram).

In some relationships (such as aggregation and generalization) several paths of the same kind may connect to a single symbol. In some circumstances (described for the particular relationship) the line segments connected to the symbol can be combined into a single line segment, so that the path from that symbol branches into several paths in a kind of tree. This is purely a graphical presentation option; conceptually the individual paths are distinct. This presentation option may not be used when the modeling information on the segments to be combined is not identical.

3.4 Invisible Hyperlinks and the Role of Tools

A notation on a piece of paper contains no hidden information. A notation on a computer screen may contain additional invisible hyperlinks that are not apparent in a static view, but that can be invoked dynamically to access some other piece of information, either in a graphical view or in a textual table. Such dynamic links are as much a part of a *dynamic* notation as the visible information, but this guide does not prescribe their form. We regard them as a tool responsibility. This document attempts to define a *static* notation for the UML, with the understanding that some useful and interesting information may show up poorly or not at all in such a view. On the other hand, we do not know enough to specify the behavior of all dynamic tools, nor do we want to stifle innovation in new forms of dynamic presentation. Eventually some of the dynamic notations may become well enough established to standardize them, but we do not feel that we should do so now.

3.5 Background Information

3.5.1 Presentation Options

Each appearance of a symbol for a class on a diagram or on different diagrams may have its own presentation choices. For example, one symbol for a class may show the attributes and operations and another symbol for the same class may suppress them. Tools may provide style sheets attached either to individual symbols or to entire diagrams. The style sheets would specify the presentation choices. (Style sheets would be applicable to most kinds of symbols, not just classes.)

Not all modeling information is presented most usefully in a graphical notation. Some information is best presented in a textual or tabular format. For example, much detailed programming information is best presented as text lists. The UML does not assume that all of the information in a model will be expressed as diagrams; some of it may only be available as tables. This document does not attempt to prescribe the format of such tables or of the forms that are used to access them, because the underlying information is adequately described in the UML metamodel and the responsibility for presenting tabular information is a tool responsibility. It is assumed that hidden links may exist from graphical items to tabular items.

3.6 String

A string is a sequence of characters in some suitable character set used to display information about the model. Character sets may include non-Roman alphabets and characters.

3.6.1 Semantics

Diagram strings normally map underlying model strings that store or encode information about the model, although some strings may exist purely on the diagrams. UML assumes that the underlying character set is sufficient for representing multibyte characters in various human languages; in particular, the traditional 8-bit ASCII character set is insufficient. It is assumed that the tool and the computer manipulate and store strings correctly, including escape conventions for special characters, and this document will assume that arbitrary strings can be used without further fuss.

3.6.2 Notation

A string is displayed as a text string graphic. Normal printable characters should be displayed directly. The display of nonprintable characters is unspecified and platform-dependent. Depending on purpose, a string might be shown as a single-line entity or as a paragraph with automatic line breaks.

Typeface and font size are graphic markers that are normally independent of the string itself. They may code for various model properties, some of which are suggested in this document and some of which are left open for the tool or the user.

3.6.3 Presentation Options

Tools may present long strings in various ways, such as truncation to a fixed size, automatic wrapping, or insertion of scroll bars. It is assumed that there is a way to obtain the full string dynamically.

3.6.4 Examples

BankAccount

integrate (f: Function, from: Real, to: Real)

{ author = “Joe Smith”, deadline = 31-March-1997, status = analysis }

The purpose of the shuffle operation is nominally to put the cards into a random configuration. However, to more closely capture the behavior of physical decks, in which blocks of cards may stick together during several riffles, the operation is actually simulated by cutting the deck and merging the cards with an imperfect merge.

3.6.5 Mapping

A graphic string maps into a string within a model element. The mapping depends on context. In some circumstances, the visual string is parsed into multiple model elements. For example, an operation signature is parsed into its various fields. Further details are given with each kind of symbol.

3.7 Name

3.7.1 Semantics

A name is a string that is used to identify a model element uniquely within some scope. A pathname is used to find a model element starting from the root of the system (or from some other point). A name is a selector (qualifier) within some scope—the scope is made clear in this document for each element that can be named.

A pathname is a series of names linked together by a delimiter (such as ‘::’). There are various kinds of pathnames described in this document, each in its proper place and with its particular delimiter.

3.7.2 Notation

A name is displayed as a text string graphic. Normally a name is displayed on a single line and will not contain nonprintable characters. Tools and languages may impose reasonable limits on the length of strings and the character set they use for names, possibly more restrictive than those for arbitrary strings, such as comments.

3.7.3 Example

Names:

BankAccount

integrate

controller

abstract

this_is_a_very_long_name_with_underscores

Pathname:

MathPak::Matrices::BandedMatrix

3.7.4 Mapping

Maps to the name of a model element. The mapping depends on context, as with String. Further details are given with the particular element.

3.8 Label

A label is a string that is attached to a graphic symbol.

3.8.1 Semantics

A label is a term for a particular use of a string on a diagram. It is purely a notational term.

3.8.2 Notation

A label is a string that is attached graphically to another symbol on a diagram. Visually the attachment normally is by containment of the string (in a closed region) or by placing the string near the symbol. Sometimes the string is placed in a definite position (such as below a symbol) but most of the time the statement is that the string must be “near” the symbol. A tool maintains an explicit internal graphic linking between a label and a graphic symbol, so that the label drags with the symbol, but the final appearance of the diagram is a matter of aesthetic judgment and should be made so that there is no confusion about which symbol a label is attached to. Although the attachment may not be obvious from a visual inspection of a diagram, the attachment is clear and unambiguous at the graphic level (and poses no ambiguity in the semantic mapping).

3.8.3 Presentation Options

A tool may visually show the attachment of a label to another symbol using various aids (such as a line in a given color, flashing of matched elements, etc.) as a convenience.

3.8.4 Example

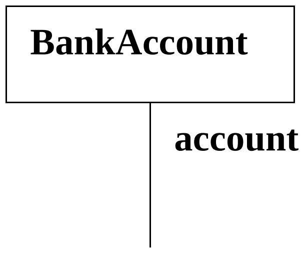


Figure 3-1 Attachment by Containment and Attachment by Adjacency

3.9 Keywords

The number of easily-distinguishable visual symbols is limited. The UML notation makes use of text keywords in places to distinguish variations on a common theme, including metamodel subclasses of a base class, stereotypes of a metamodel base class, and groups of list elements. From the user's perspective, the metamodel distinction between metamodel subclasses and stereotypes is often unimportant, although it is important to tool builders and others who implement the metamodel.

The general notation for the use of a keyword is to enclose it in guillemets («»):

«*keyword*»

Certain predefined keywords are described in the text of this document. These must be treated as reserved words in the notation. Others are available for users to employ as stereotype names. The use of a stereotype name that matches a predefined keyword is ill formed.

3.10 Expression

3.10.1 Semantics

Various UML constructs require expressions, which are linguistic formulas or procedures that yield values when evaluated at run-time. These include expressions for types, boolean values, and numbers. UML does not include an explicit linguistic analyzer for expressions. Rather, expressions are expressed as strings in a particular language or using procedures, or both. The OCL constraint language is used within the UML semantic definition and may also be used at the user level; other languages (such as programming languages) may also be used.

UML avoids specifying the syntax for constructing type expressions because they are so language-dependent. It is assumed that the name of a class or simple data type will map into a simple Classifier reference, but the syntax of complicated language-dependent type expressions, such as C++ function pointers, is the responsibility of the specification language.

3.10.2 Notation

An expression is displayed as a string defined in a particular language. The syntax of the string is the responsibility of a tool and a linguistic analyzer for the language. The assumption is that the analyzer can evaluate strings at run-time to yield values of the appropriate type, or can yield a procedure to capture the meaning of the expression. For example, a type expression evaluates to a Classifier reference, and a boolean expression evaluates to a true or false value. The language itself is known to a modeling tool but is generally implicit on the diagram, under the assumption that the form of the expression makes its purpose clear.

3.10.3 Examples

```
BankAccount

BankAccount * (*) (Person*, int)

array [1..20] of reference to range (-1.0..1.0) of Real

[ i > j and self.size > i ]
```

3.10.4 Mapping

An expression string maps to an Expression element (possibly a particular subclass of Expression, such as BooleanExpression or TimeExpression). If an analyzer yields a procedure for calculating the value of the expression, then the body association from Expression to Procedure is used to record this.

3.10.5 OCL Expressions

UML includes a definition of the OCL language, which is used to define constraints within the UML metamodel itself. The OCL language may be supported by tools for user-written expressions as well. Other possible languages include various computer languages as well as plain text (which cannot be parsed by a tool, of course, and is therefore only for human information). The OCL language is defined in the “Object Constraint Language Specification” chapter.

3.10.6 Selected OCL Notation

Syntax for some common navigational expressions are shown below. 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.

item '.' *selector*	The *selector* is the name of an attribute in the item or the name of the target end of a link attached to the item. 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 multiplicities of the item and the association.
item '.' *selector* '[' *qualifier-value* ']'	The *selector* designates a qualified association that qualifies the *item*. 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.
set '->' 'select' '(' *boolean-expression* ')'	The *boolean-expression* is written in terms of objects within the set. The result is the subset of objects in the set for which the boolean expression is true.

3.10.7 Examples

```
flight.pilot.training_hours > flight.plane.minimum_hours

company.employees–>select (title = “Manager” and self.reports–>size > 10)
```

3.11 Note

A note is a graphical symbol containing textual information (possibly including embedded images). It is a notation for rendering various kinds of textual information from the metamodel, such as constraints, comments, method bodies, and tagged values.

3.11.1 Semantics

A note is a notational item. It shows textual information within some semantic element.

3.11.2 Notation

A note is shown as a rectangle with a “bent corner” in the upper right corner. It contains arbitrary text. It appears on a particular diagram and may be attached to zero or more modeling elements by dashed lines.

3.11.3 Presentation Options

A note may have a stereotype.

A note with the keyword “constraint” or a more specific stereotype of constraint (such as the code body for a method) designates a constraint that is part of the model and not just part of a diagram view. Such a note is the view of a model element (the constraint).

3.11.4 Example

See also Figure 3-17 on page 3-28 for a note symbol containing a constraint.

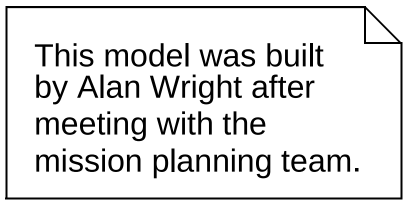

This model was built by Alan Wright after meeting with the mission planning team.

Figure 3-2 Note

3.11.5 Mapping

A note may represent the textual information in several possible metamodel constructs; it must be created in context that is known to a tool, and the tool must maintain the mapping. The string in the note maps to the body of the corresponding modeling element. A note may represent:

- a constraint,
- a tagged value,
- the body of a procedure of a method, or
- other string values within modeling elements.

It may also represent a comment attached directly to a diagram element.

3.12 Type-Instance Correspondence

A major purpose of modeling is to prepare generic descriptions that describe many specific items. This is often known as the *type-instance dichotomy.* Many or most of the modeling concepts in UML have this dual character, usually modeled by two paired modeling elements, one represents the generic descriptor and the other the individual items that it describes. Examples of such pairs in UML include: Class-Object, Association-Link, UseCase-UseCaseInstance, Message-Stimulus, and so on.

Although diagrams for type-like elements and instance-like elements are not exactly the same, they share many similarities. Therefore, it is convenient to choose notation for each type-instance pair of elements such that the correspondence is visually apparent immediately. There are a limited number of ways to do this, each with advantages and disadvantages. In UML, the type-instance distinction is shown by employing the same geometrical symbol for each pair of elements and by underlining the name string (including type name, if present) of an instance element. This visual distinction is generally easily apparent without being overpowering even when an entire diagram contains instance elements.

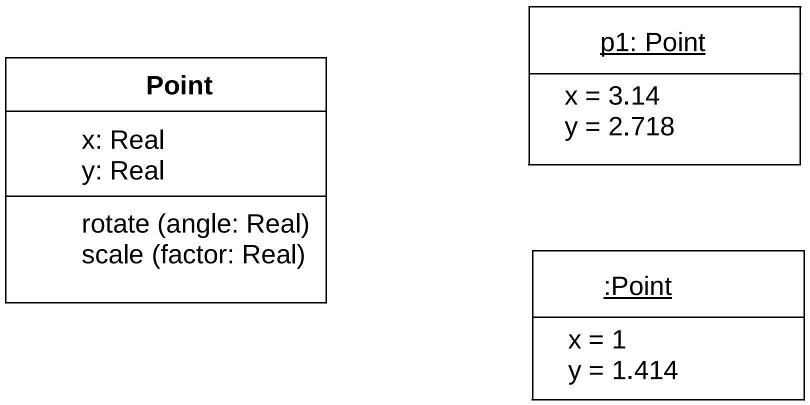


Figure 3-3 Classes and Objects

A tool is free to substitute a different graphic marker for instance elements at the user's option, such as color, fill patterns, or so on.

Roles (in collaborations) are somewhat between types and instances. Like instances, they identify distinct occurrences of a single classifier. Like types, they describe a reusable element that can have many distinct instances. A role is a distinguishable use of a classifier, but one that is still part of a general description (a collaboration) that can be used to create many instances. A run-time object may correspond to zero or more classes and to zero or more roles. The notation for a role permits indication of its base classifiers. The notation for an instance permits specification of its classifiers, its roles, or both.

A role is indicated by a name, colon, and type, not underlined and part of a collaboration. An instance is indicated by an optional name, optional slash followed by list of roles, colon, and list of types.

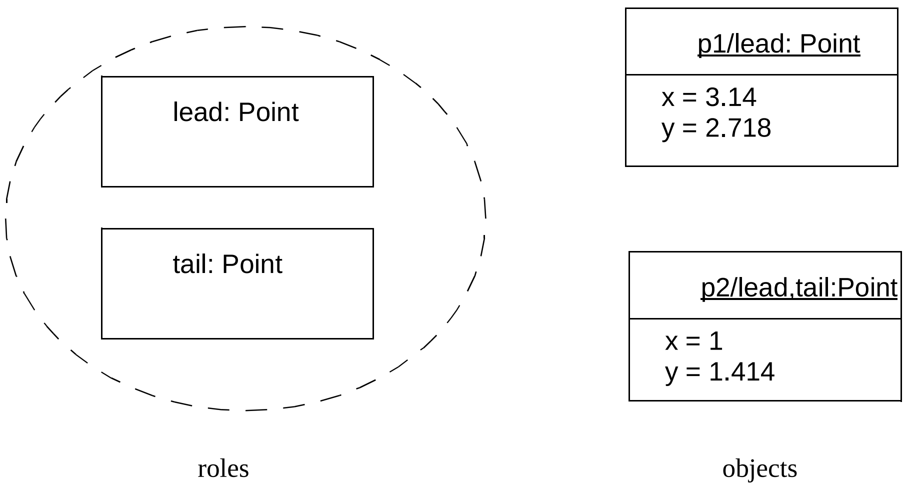


Figure 3-4 Roles and objects

Part 3 - Model Management

3.13 Package

3.13.1 Semantics

A *package* is a grouping of model elements. Packages themselves may be nested within other packages. A package may contain subordinate packages as well as other kinds of model elements. All kinds of UML model elements can be organized into packages.

Note that packages *own* model elements and are the basis for configuration control, storage, and access control. Each element can be directly owned by a single package, so the package hierarchy is a strict tree. However, packages can reference other packages, modeled by using one of the stereotypes «import» and «access» of Permission dependency, so the usage network is a graph. Other kinds of dependencies between packages usually imply that one or more dependencies among the elements exists.

3.13.2 Notation

A package is shown as a large rectangle with a small rectangle (a “tab”) attached to the left side of the top of the large rectangle. It is the common folder icon.

The contents of the package may be shown within the large rectangle. Contents may also be shown by branching lines to contained elements, drawn outside of the package (see Figure 3-5 on page 3-18). A plus sign (+) within a circle is drawn at the end attached to the container.

- If the contents of the package are not shown within the large rectangle, then the name of the package may be placed within the large rectangle.
- If the contents of the package are shown within the large rectangle, then the name of the package may be placed within the tab.

A keyword string may be placed above the package name. The predefined stereotypes *facade, framework, stub,* and *topLevel* are notated within guillemets.

A list of properties may be placed in braces after or below the package name. Example: {abstract}. See Section 3.17, “Element Properties,” on page3-29 for details of property syntax.

The visibility of a package element outside the package may be indicated by preceding the name of the element by a visibility symbol (‘+’ for public, ‘-’ for private, ‘#’ for protected, ‘~’ for package).

Relationships may be drawn between package symbols to show relationships between some of the elements in the packages. An import or access relationship between two packages is drawn as a dashed arrow with open arrowhead, labeled with the string «import» or «access», respectively.

Elements from imported or accessed packages may be shown outside the package symbol. As (public) elements in imported packages are added to the client namespace, they may alternatively be drawn inside the package symbol.

3.13.3 Presentation Options

A tool may show visibility by a graphic marker, such as color or font.

A tool may also show visibility by selectively displaying those elements that meet a given visibility level; for example, all of the public elements only.

A diagram showing a package with contents must not necessarily show all its contents; it may show a subset of the contained elements according to some criterion.

The contents of a package may also be shown using tree notation. The namespace ownership relationships between the package and its elements are marked with a circle with a cross in it at the owning end.

3.13.4 Style Guidelines

It is expected that packages with large contents will be shown as simple icons with names, in which the contents may be dynamically accessed by “zooming” to a detailed view.

3.13.5 Example

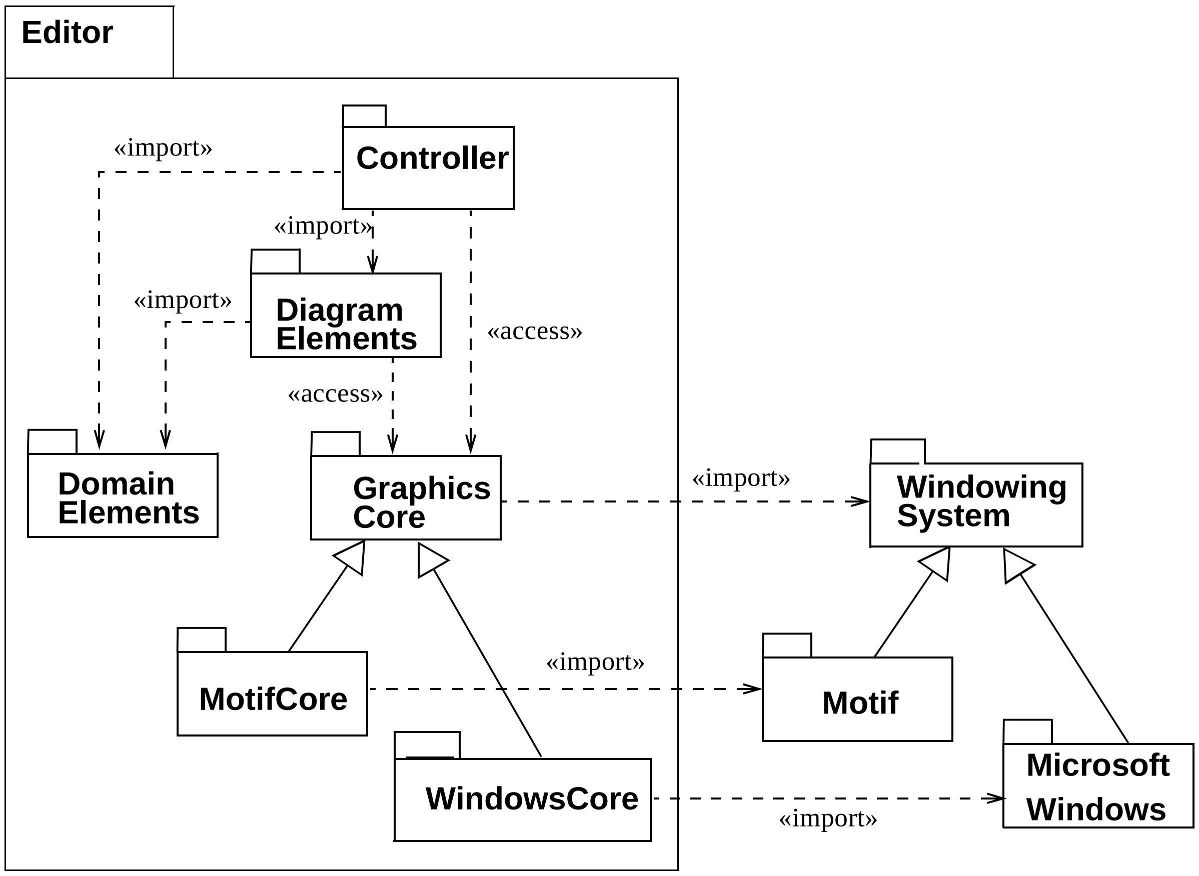


Figure 3-5 Packages and their access and import relationships.

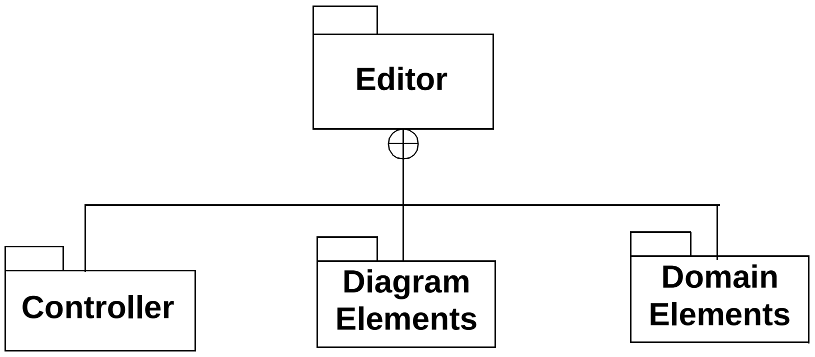


Figure 3-6 Some of the contents of the Editor package shown in a tree structure.

3.13.6 Mapping

A package symbol maps into a Package element. The name on the package symbol is the name of the Package element. If there is a string above the package name other than «model» or «subsystem», then it maps into a Package element with the corresponding stereotype. If there is a string «model» or «subsystem», then it maps into a Model or Subsystem element, respectively.

A relationship icon drawn from the package symbol boundary to another package symbol maps into a corresponding relationship to the other package element.

A symbol directly contained within the package symbol; that is, not contained within another symbol maps into a model element either owned or referenced by the package element. The alias used for a referenced element is often its pathname, in which case it is directly visible from the diagram that the element is not owned by the package. Only the reference is owned by the current package. Alternatively, a symbol shown outside the package symbol, attached to one of the symbols within the package symbol, denotes a referenced model element.

Symbols connected to the package symbol by branching lines with a plus sign at the end attached to the package symbol, map to elements in the package.

3.14 Subsystem

3.14.1 Semantics

Whereas a package is a generic mechanism for organizing model elements, a *subsystem* represents a behavioral unit in the physical system, and hence in the model. A subsystem offers interfaces and has operations, and its contents are partitioned into specification and realization elements. The specification of the subsystem consists of operations on the subsystem, together with specification elements such as use cases, state machines.

Apart from defining a namespace, a subsystem serves as a specification unit for the behavior of its contained model elements. A subsystem may or may not be instantiable.

3.14.2 Notation

A subsystem is notated basically in the same way as a package, with the addition of a fork symbol placed in the upper right corner of the large rectangle. The name of the subsystem (together with optional keyword, stereotype) is placed within the large rectangle. Optionally, especially if contents of the subsystem are shown within the large rectangle, the subsystem name and the fork are placed within the tab (the small rectangle).

An instantiable subsystem has the string «instantiable» above its name.

The large rectangle has three compartments, one for operations and one for each of the subsets specification elements and realization elements. These are usually shown by dividing the rectangle by a vertical line, and then dividing the area to the left of this line into two compartments by a horizontal line. The operations are shown in the upper left compartment, the specification elements in the compartment below, and the realization elements in the right compartment. The latter two compartments are labeled 'Specification Elements' and 'Realization Elements,' respectively, to avoid potential ambiguity. The operations compartment is unlabeled. This is the general pattern for subsystem notation, although there are many different ways to customize it in a particular diagram, see Section 3.14.3, "Presentation Options," on page 3-20 and Section 3.14.4, "Example," on page 3-21.

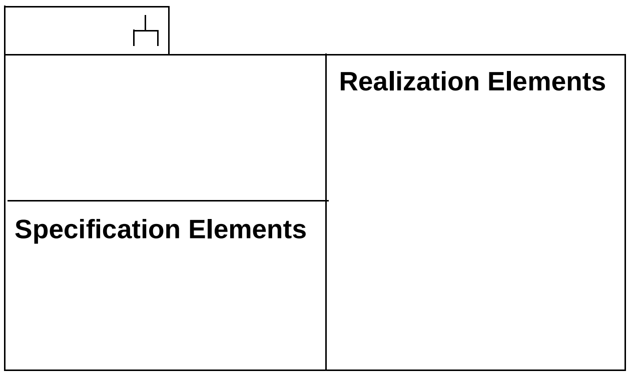


Figure 3-7 The general pattern for subsystem notation, with three compartments.

The mapping from the realization part to the specification part; that is, to operations and specification elements, is drawn using dashed arrows with closed, hollow arrowheads. For collaborations, the mapping may also be expressed textually.

When a subsystem is shown together with other, peer elements in a diagram, it is often shown without contents, in which case there are no compartments in the large rectangle. See Section 3.14.4, "Example," on page 3-21.

3.14.3 Presentation Options

The fork symbol may be replaced by the keyword «subsystem» placed above the name of the subsystem.

The compartments may be rearranged within the subsystem symbol.

One or more of the compartments may be collapsed or suppressed. In cases where more than one diagram is used to show all information about a particular subsystem, each diagram shows a subset of the subsystem's features and/or contents. Hence, compartments not relevant in a particular diagram are suppressed.

All contained elements in a subsystem may be shown together in one, non-labeled compartment; that is, no visual differentiating between specification elements and realization elements is done.

Tools may provide alternative ways to differentiate specification elements from realization elements, such as different colors, using the keyword «specification» for specification elements, etc.

As with packages, the contents of a subsystem may be shown using tree notation. Distinction between specification and realization elements may then be done; for example, by having two separate, labeled branches, or by showing the category separately for each element in the tree as suggested above.

3.14.4 Example

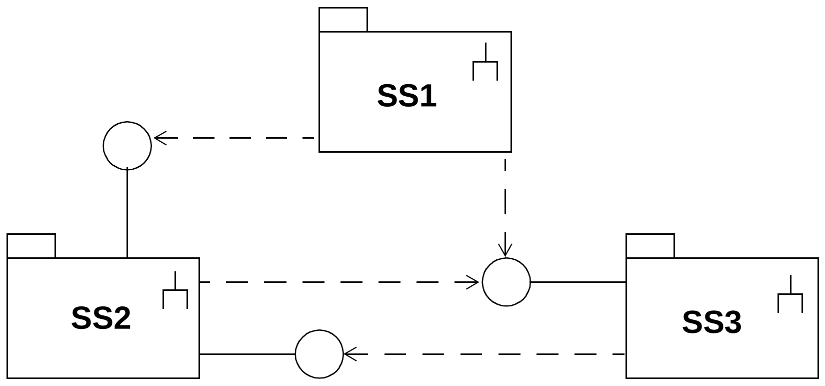


Figure 3-8 An overview diagram showing subsystems with interfaces and their dependencies.

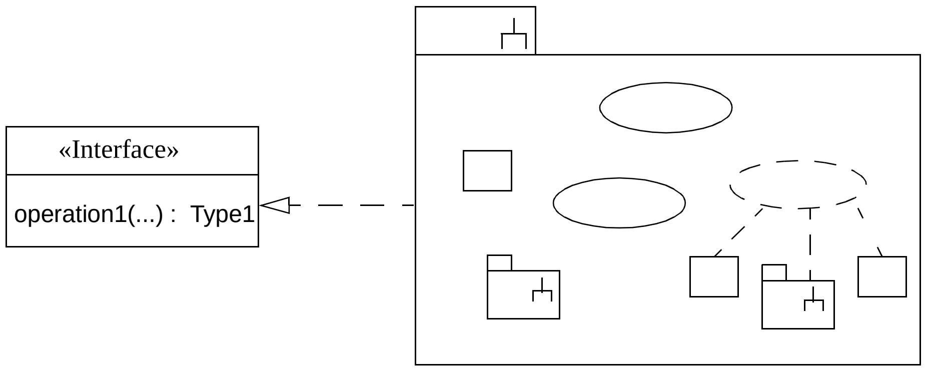


Figure 3-9 All contained elements of a subsystem shown together without division into compartments. Here, the subsystem offers operation1(...) although this is not explicitly shown.

In Figure 3-9 no visual separation between specification and realization elements is made. The following three figures are schematic examples where the specification/realization distinction is explicit. Together these figures constitute an example of how the basic notation for subsystem can be used to show different “views” of a subsystem in different diagrams, together giving the whole picture of the subsystem.

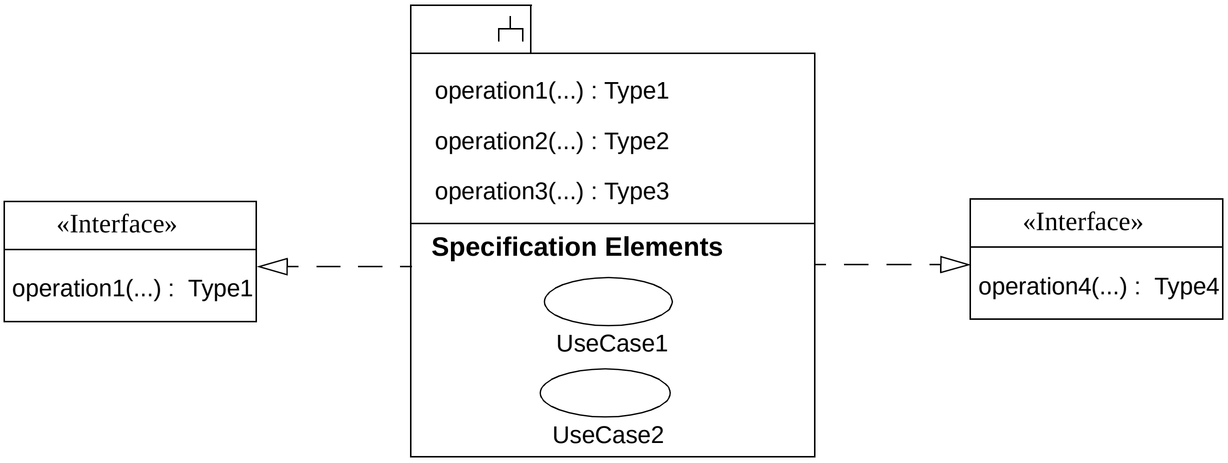


Figure 3-10 The specification part of a subsystem; compartment for realization part is suppressed. Implicit from the diagram is that the operation4(...) is either an operation of a specification element (UseCase1 or UseCase2) or of the subsystem itself. Furthermore, in cases where no operations are used for the specification but only contained specification elements, there is no operations compartment, and *vice versa*.

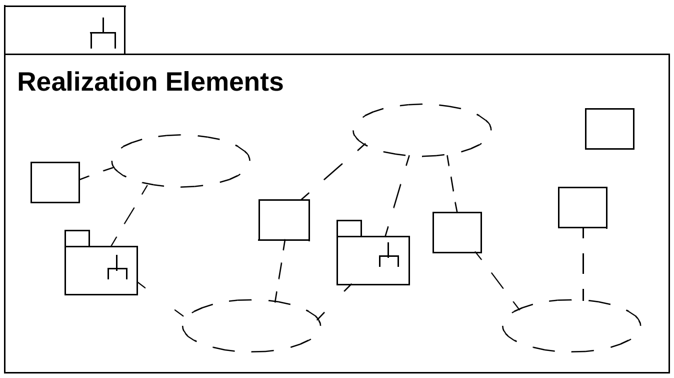


Figure 3-11 The realization part of a subsystem; compartments for specification part; that is, operations and specification elements are suppressed. Alternatively, collaborations could be shown in a separate diagram.

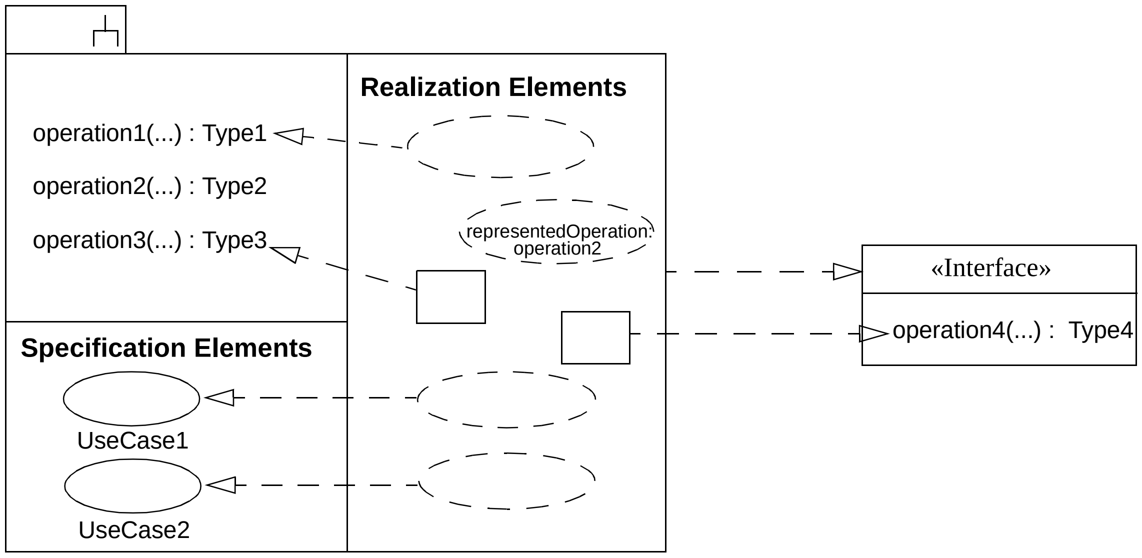


Figure 3-12 The mapping between specification part and realization part shown using all three compartments, but only those realization elements with relevance to the mapping are shown. The figure also shows examples of different ways to express the mapping.

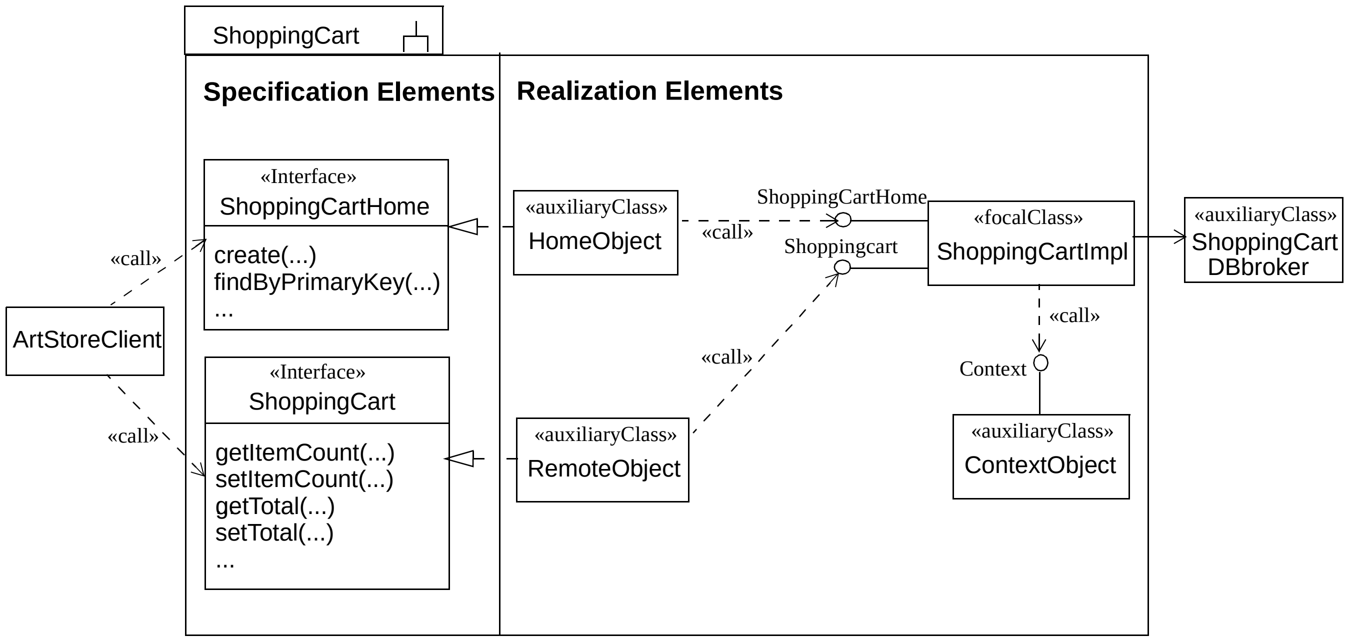


Figure 3-13 A component modeled using a subsystem and classes stereotyped «focalClass» or «auxiliaryClass», respectively.

3.14.5 Mapping

A subsystem symbol maps into a Subsystem with the given name. The mapping is analogous to that of package symbols, with the following addition:

A symbol within a compartment of the large rectangle labeled ‘Specification Elements’ or ‘Realization Elements’ is mapped to a specification or realization element of the subsystem, respectively. An operation signature string within a non-labeled compartment maps to an operation of the subsystem. Note that a compartment may coincide with the whole rectangle.

A symbol, that is not an operation signature string, within a non-labeled compartment maps to an element contained in the subsystem.

A dashed arrow with closed, hollow arrowhead from a symbol denoting a realization element to a symbol denoting a specification element or an operation maps to a «realize» relationship between the corresponding elements.

3.15 Model

3.15.1 Semantics

A model captures a view of a physical system. Hence, it is an abstraction of the physical system with a certain purpose; for example, to describe behavioral aspects of the physical system to a certain category of stakeholders. A model contains all the model elements needed to represent a physical system completely according to the purpose of this particular model. The model elements in a model are organized into a package/subsystem hierarchy, where the top-most package/subsystem represents the boundary of the physical system.

Different models of the same physical system show different aspects of the system. The pre-defined stereotype «systemModel» can be applied to a model containing the entire set of models for a physical system.

Relationships between elements in different models have no semantic impact on the contents of the models because of the self-containment of models. However, they are useful for tracing refinements and for keeping track of requirements between models.

Relationships between models express refinement, import, etc.

3.15.2 Notation

A model is notated using the ordinary package symbol with a small triangle in the upper right corner of the large rectangle. Optionally, especially if contents of the model is shown within the large rectangle, the triangle may be drawn to the right of the model name in the tab.

Relationships between models as well as relationships between elements in different models are shown using the notation for the given kind of relationship. In particular, trace dependencies are notated with a dashed line, with an optional open arrowhead, and the keyword «trace».

3.15.3 Presentation Options

A model may be notated as a package, using the ordinary package symbol with the keyword «model» placed above the name of the model.

3.15.4 Example

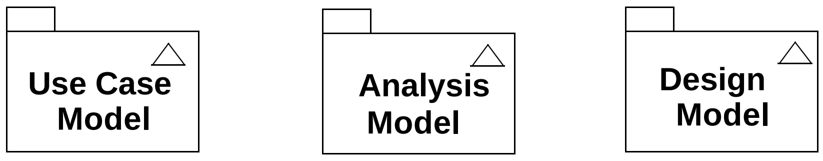


Figure 3-14 Three views of a physical system, each represented by a model.

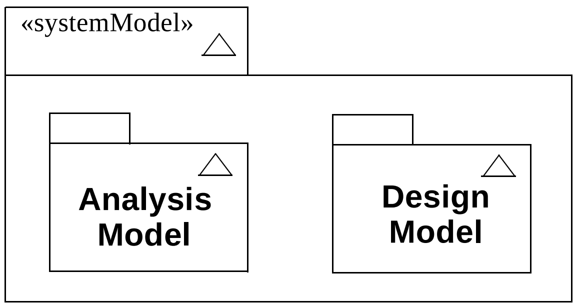


Figure 3-15 A «systemModel» containing an analysis model and a design model.

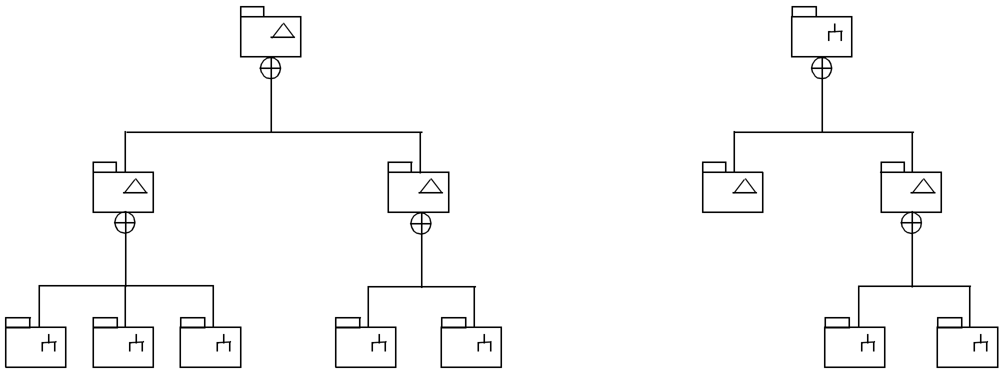


Figure 3-16 Two examples of containment hierarchies with models and subsystems shown using branching lines. The left hierarchy is based on Model, whereas the right one is based on Subsystem.

3.15.5 Mapping

A model symbol maps to a Model with the given name. The mapping is analogous to that of package symbols.

Part 4 - General Extension Mechanisms

The elements in this section are general purpose mechanisms that may be applied to any modeling element. The semantics of a particular use depends on a convention of the user or an interpretation by a particular constraint language or programming language; therefore, they constitute an extensibility device for UML.

3.16 Constraint and Comment

3.16.1 Semantics

A *constraint* is a semantic relationship among model elements that specifies conditions and propositions that must be maintained as true; otherwise, the system described by the model is invalid (with consequences that are outside the scope of UML). Certain kinds of constraints (such as an association "xor" constraint) are predefined in UML, others may be user-defined. A user-defined constraint is described in words in a given language, whose syntax and interpretation is a tool responsibility. A constraint represents semantic information attached to a model element, not just to a view of it.

A *comment* is a text string (including references to human-readable documents) attached directly to a model element. A comment can attach arbitrary textual information to any model element of presumed general importance but it has no semantic force. Comments may be used for explaining the reasons for decisions, among other things.

3.16.2 Notation

A constraint is shown as a text string in braces ({ }). There is an expectation that individual tools may provide one or more languages in which formal constraints may be written. One predefined language for writing constraints is OCL (see the Object Constraint Language Specification chapter); otherwise, the constraint may be written in natural language. Each constraint is written in a specific language, although the language is not generally displayed on the diagram (the tool must keep track of it, however).

For an element whose notation is a text string (such as an attribute, etc.), the constraint string may follow the element text string in braces.

For a list of elements whose notation is a list of text strings (such as the attributes within a class), a constraint string may appear as an element in the list. The constraint applies to all succeeding elements of the list until another constraint string list element or the end of the list. A constraint attached to an individual list element does not supersede the general constraint, but may augment or modify individual constraints within the constraint string.

For a single graphical symbol (such as a class or an association path), the constraint string may be placed near the symbol, preferably near the name of the symbol, if any.

For two graphical symbols (such as two classes or two associations), the constraint is shown as a dashed arrow from one element to the other element labeled by the constraint string (in braces). The direction of the arrow is relevant information within the constraint. The client (tail of the arrow) is mapped to the first position and the supplier (head of the arrow) is mapped to the second position in the constraint.

For three or more graphical symbols, the constraint string is placed in a note symbol and attached to each of the symbols by a dashed line. This notation may also be used for the other cases. For three or more paths of the same kind (such as generalization paths or association paths), the constraint may be attached to a dashed line crossing all of the paths.

A comment is shown as a text string (not enclosed in braces) within a note icon. Syntax for including comments within other elements (such as expressions or constraints) are not specified by UML but may be provided by a tool as part of the expression syntax for a particular language.

3.16.3 Example

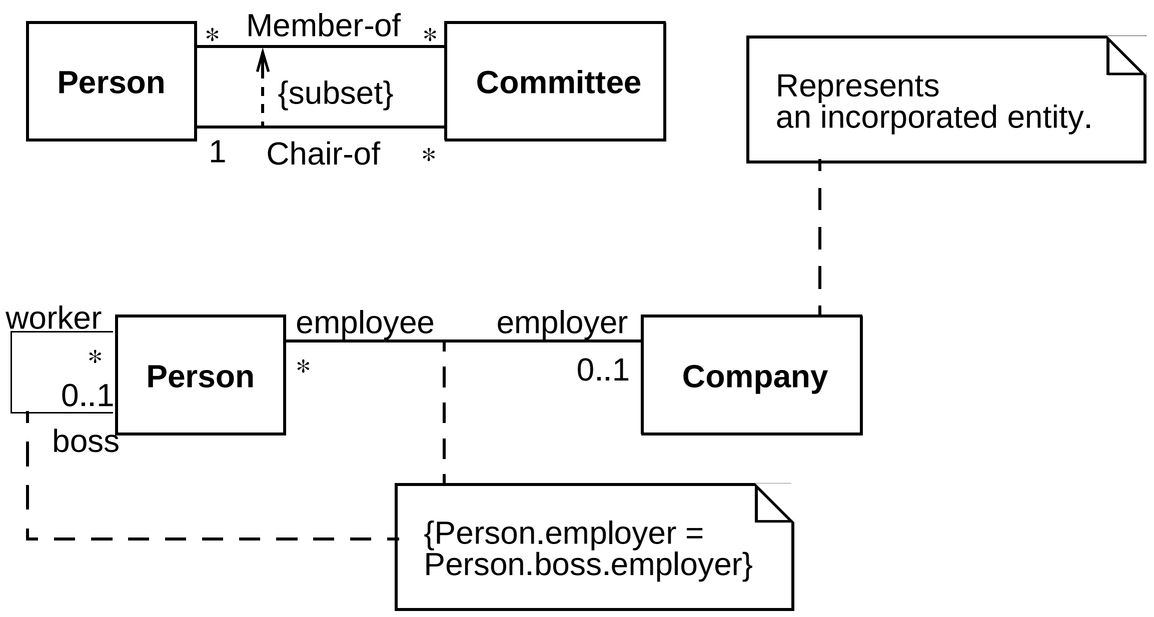


Figure 3-17 Constraints and comment

3.16.4 Mapping

A constraint string is a string enclosed in braces ({ }).

The constraint string maps into the *body* expression in a Constraint element. The mapping depends on the language of the expression, which is known to a tool but generally not displayed on a diagram.

A constraint string following a list entry maps into a Constraint attached to the element corresponding to the list entry.

A constraint string represented as a stand-alone list element maps into a separate Constraint attached to each succeeding model element corresponding to subsequent list entries (until superseded by another constraint or property string).

A constraint string placed near a graphical symbol must be attached to the symbol by a hidden link by a tool operating in context. The tool must maintain the graphical linkage implicitly. The constraint string maps into a Constraint attached to the element corresponding to the symbol.

A constraint string attached to a dashed arrow maps into a constraint attached to the two elements corresponding to the symbols connected by the arrow.

A string enclosed in braces in a note symbol maps into a Constraint attached to the elements corresponding to the symbols connected to the note symbol by dashed lines.

A string (not enclosed in braces) in a note attached to the symbol for an element maps into a Comment attached to the corresponding element.

3.17 Element Properties

Many kinds of elements have detailed properties that do not have a visual notation. In addition, users can define new element properties using the *tagged value* mechanism.

A string may be used to display properties attached to a model element. This includes properties represented by attributes in the metamodel as well as both predefined and user-defined tagged values.

3.17.1 Semantics

Note that we use *property* in a general sense to mean any value attached to a model element, including attributes, associations, and tagged values. In this sense it can include indirectly reachable values that can be found starting at a given element. Some kinds of properties would have syntax within expressions (not specified by UML) but no explicit UML notation.

A *tagged value* is a keyword-value pair that may be attached to any kind of model element (including diagram elements as well as semantic model elements). The keyword is called a *tag*. Each tag represents a particular kind of property applicable to one or many kinds of model elements. Both the tag and the value are encoded as strings. Tagged values are an extensibility mechanism of UML permitting arbitrary information to be attached to models. It is expected that most model editors will provide basic facilities for defining, displaying, and searching tagged values as strings but will not otherwise use them to extend the UML semantics. It is expected, however, that back-end tools such as code generators, report writers, and the like will read tagged values to guide their semantics in flexible ways.

3.17.2 Notation

A property (either a metamodel attribute or a tagged value) is displayed as a comma-delimited sequence of *property specifications* all inside a pair of braces ({ }).

A *property specification* has the form

name = *value*

where *name* is the name of a property (metamodel attribute or arbitrary tag) and *value* is an arbitrary string that denotes its value. If the type of the property is Boolean, then the default value is **true** if the value is omitted. That is, to specify a value of true you may include just the keyword. To specify a value of false, you omit the name completely. Properties of other types require explicit values. The syntax for displaying the value is a tool responsibility in cases where the underlying model value is not a string or a number.

Note that property strings may be used to display built-in attributes as well as tagged values.

Boolean properties frequently have the form is*Name*, where *name* is the name of some condition that may be true or false. In these cases, the form "*name*" may usually appear by itself, without a value, to mean "is*Name* = true." For example, {abstract} is the same as {isAbstract = true}.

Tagged values can sometimes refer to other model elements (see Section 2.6.2.5, "TaggedValue," on page 2-79). In that case, the usual tagged value format is used except that the value is the name of the model element that is referenced. Alternatively, it may be represented graphically using a «taggedValue» relationship, which uses the dependency notation. The direction of the dependency arrow is towards the referenced element. These two cases are illustrated in Figure 3-18

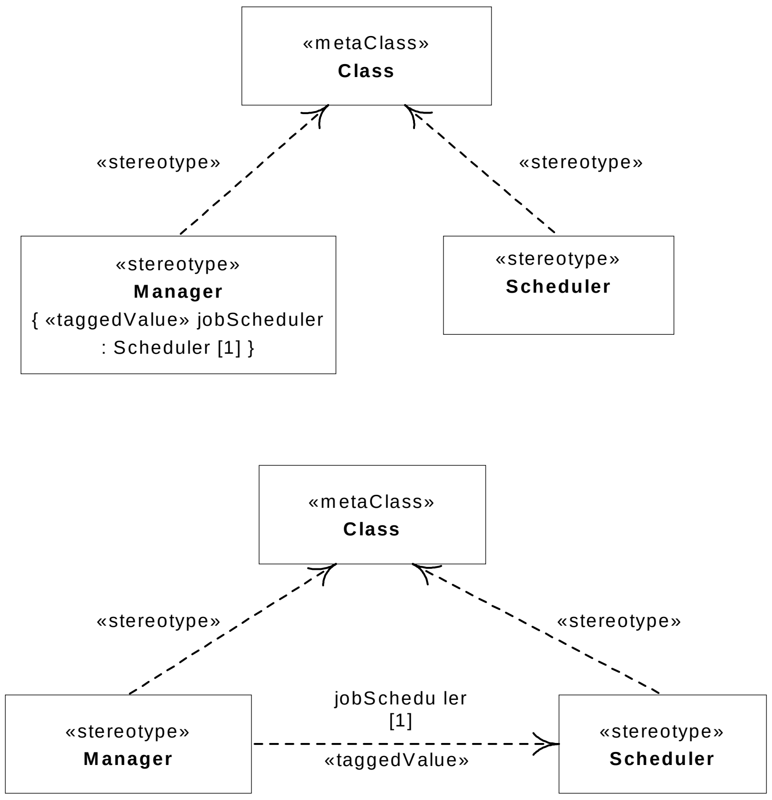


Figure 3-18 Alternative notations for tagged values as references

3.17.3 Presentation Options

A tool may present property specifications on separate lines with or without the enclosing braces, provided they are marked appropriately to distinguish them from other information. For example, properties for a class might be listed under the class name in a distinctive typeface, such as italics or a different font family.

3.17.4 Style Guidelines

It is legal to use strings to specify properties that have graphical notations; however, such usage may be confusing and should be used with care.

3.17.5 Example

{ author = “Joe Smith”, deadline = 31-March-1997, status = analysis }

{ abstract }

3.17.6 Mapping

Each term within a string maps to either a built-in attribute of a model element or a tagged value (predefined or user-defined). A tool must enforce the correspondence to built-in attributes.

3.18 Stereotypes

3.18.1 Semantics

A stereotype is, in effect, a new class of metamodel element that is introduced at modeling time. It represents a subclass of an existing metamodel element with the same form (attributes and relationships) but with a different intent. Generally a stereotype represents a usage distinction. A stereotyped element may have additional constraints on it from the base metamodel class. It may also have required tagged values that add information needed by elements with the stereotype. It is expected that code generators and other tools will treat stereotyped elements specially. Stereotypes represent one of the built-in extensibility mechanisms of UML.

3.18.2 Notation

The general presentation of a stereotype is to use the symbol for the metamodel base element but to place a keyword string above the name of the element (if any). The keyword string (Section 3.9, “Keywords,” on page 3-11) is the name of the stereotype within matched *guillemets*, which are the quotation mark symbols used in French and certain other languages (for example, «foo»).

Note – A guillemet looks like a double angle-bracket, but it is a single character in most extended fonts. Most computers have a Character Map utility. Double angle-brackets may be used as a substitute by the typographically challenged.

The keyword string is generally placed above or in front of the name of the model element being described. If multiple stereotypes are defined for the same model element, they are placed vertically one below the other. The keyword string may also be used as an element in a list, in which case it applies to subsequent list elements until another stereotype string replaces it, or an empty stereotype string («») nullifies it. Note that a stereotype name should not be identical to a predefined keyword applicable to the same element type.

To permit limited graphical extension of the UML notation as well, a graphic icon or a graphic marker (such as texture or color) can be associated with a stereotype. The UML does not specify the form of the graphic specification, but many bitmap and stroked formats exist (and their portability is a difficult problem). The icon can be used in one of two ways:

1. It may be used instead of, or in addition to, the stereotype keyword string as part of the symbol for the base model element that the stereotype is based on. For example, in a class rectangle it is placed in the upper right corner of the name compartment. In this form, the normal contents of the item can be seen.
2. The entire base model element symbol may be "collapsed" into an icon containing the element name or with the name above or below the icon. Other information contained by the base model element symbol is suppressed. More general forms of icon specification and substitution are conceivable, but we leave these to the ingenuity of tool builders, with the warning that excessive use of extensibility capabilities may lead to loss of portability among tools.

If multiple stereotypes are defined, the graphical icons or markers are omitted.

UML avoids the use of graphic markers, such as color, that present challenges for certain persons (the color blind) and for important kinds of equipment (such as printers, copiers, and fax machines). None of the UML symbols *require* the use of such graphic markers. Users *may* use graphic markers freely in their personal work for their own purposes (such as for highlighting within a tool) but should be aware of their limitations for interchange and be prepared to use the canonical forms when necessary.

The classification hierarchy of the stereotypes themselves can be displayed on a class diagram, as described in Section 3.35, "Stereotype Declaration," on page 3-57. This capability is not required by many modelers who must use existing stereotypes but not define new kinds of stereotypes.

3.18.3 Examples

Figure 3-19 on page 3-33 illustrates various notational forms of the stereotype notation. Note that the top four shapes are alternatives of each other. The next one shows how a dependency can be stereotyped and the bottom example illustrates a model element with multiple stereotypes.

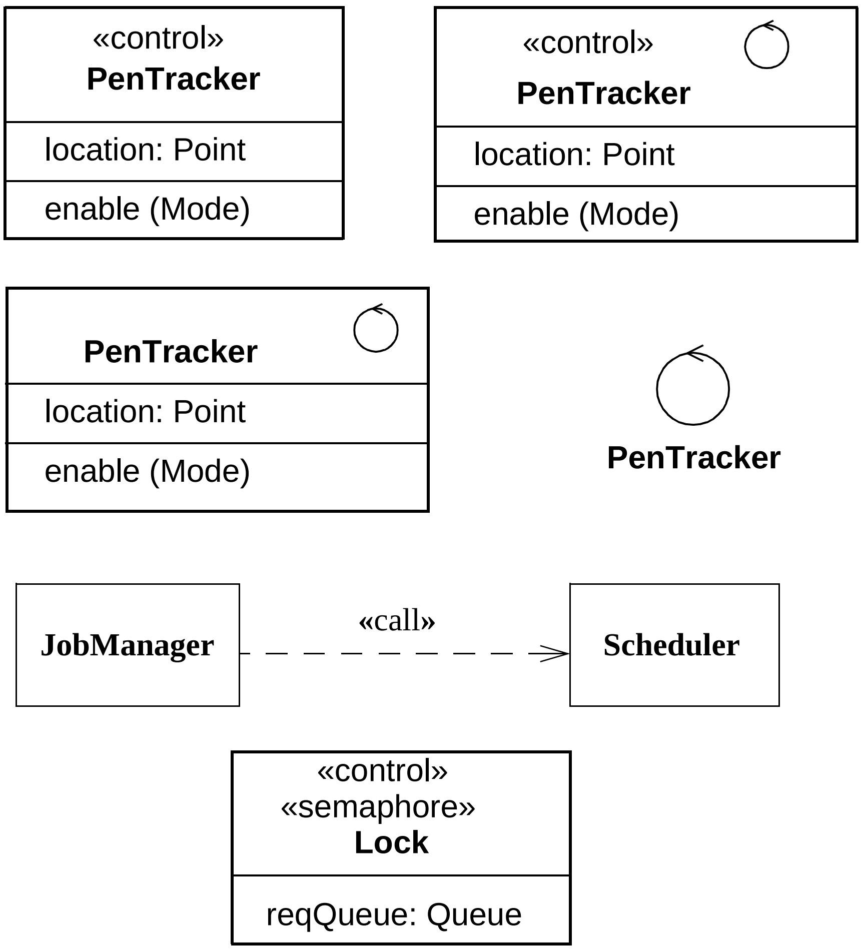


Figure 3-19 Varieties of Stereotype Notation

3.18.4 Mapping

The use of a stereotype keyword maps into the stereotype relationship between the Element corresponding to the symbol containing the name and the Stereotype of the given name. The use of a stereotype icon within a symbol maps into the stereotype relationship between the Element corresponding to the symbol containing the icon and the Stereotype represented by the symbol. A tool must establish the connection when the symbol is created and there is no requirement that an icon represent uniquely one stereotype. The use of a stereotype icon, instead of a symbol, must be created in a context in which a tool implies a corresponding model element and a Stereotype represented by the icon. The element and the stereotype have the stereotype relationship.

Part 5 - Static Structure Diagrams

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

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

3.19 Class Diagram

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

3.19.1 Semantics

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

3.19.2 Notation

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

3.19.3 Mapping

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

3.20 Object Diagram

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

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

3.21 Classifier

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

3.22 Class

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

3.22.1 Semantics

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

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

3.22.2 Basic Notation

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

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

3.22.2.1 References

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

Package-name::*Class-name*

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

3.22.3 Presentation Options

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

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

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

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

3.22.4 Style Guidelines

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

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

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

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

3.22.5 Example

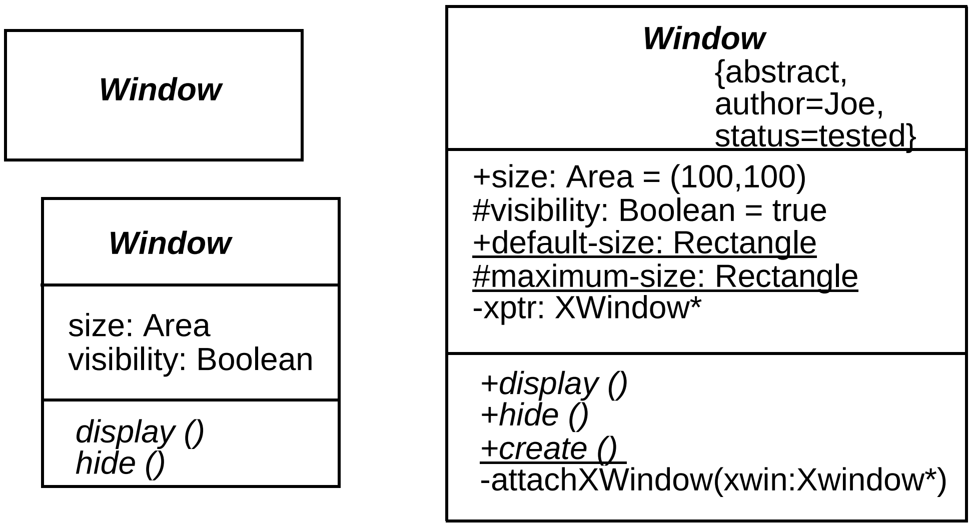


Figure 3-20 Class Notation: Details Suppressed, Analysis-level Details, Implementation-level Details

3.22.6 Mapping

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

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

3.23 Name Compartment

3.23.1 Notation

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

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

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

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

The stereotype and property list are optional.

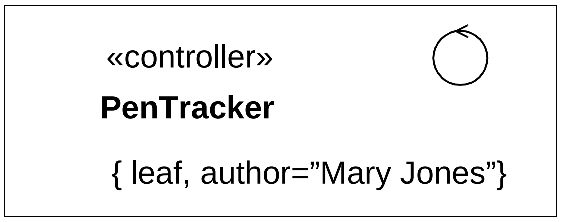


Figure 3-21 Name Compartment

3.23.2 Mapping

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

3.24 List Compartment

3.24.1 Notation

A list compartment holds a list of strings, each of which is the encoded representation of a feature, such as an attribute or operation. The strings are presented one to a line with overflow to be handled in a tool-dependent manner. In addition to lists of attributes or operations, optional lists can show other kinds of predefined or user-defined values, such as responsibilities, rules, or modification histories. UML does not define these optional lists. The manipulation of user-defined lists is tool-dependent.

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

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

3.24.1.1 Group properties

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

3.24.1.2 Compartment name

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

3.24.2 Presentation Options

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

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

The elements in the list may be filtered according to some selection rule. The specification of selection rules is a tool responsibility. The absence of items from a filtered list indicates that no elements meet the filter criterion, but no inference can be drawn about the presence or absence of elements that do not meet the criterion. However, the ellipsis notation is available to show that invisible elements exist. It is a tool responsibility whether and how to indicate the presence of either local or global filtering, although a stand-alone diagram should have some indication of such filtering if it is to be understandable.

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

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

3.24.3 Example

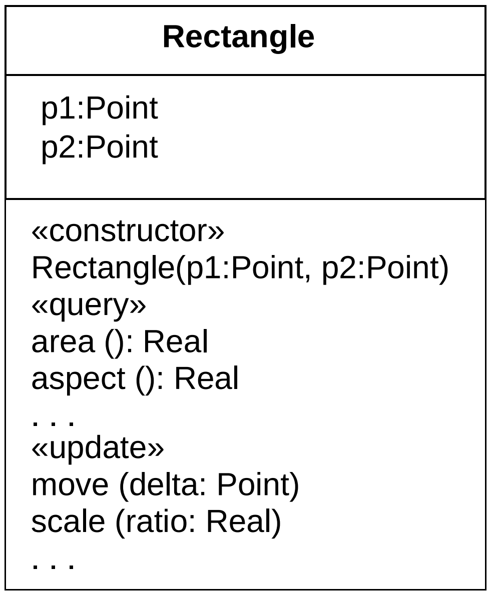


Figure 3-22 Stereotype Keyword Applied to Groups of List Elements

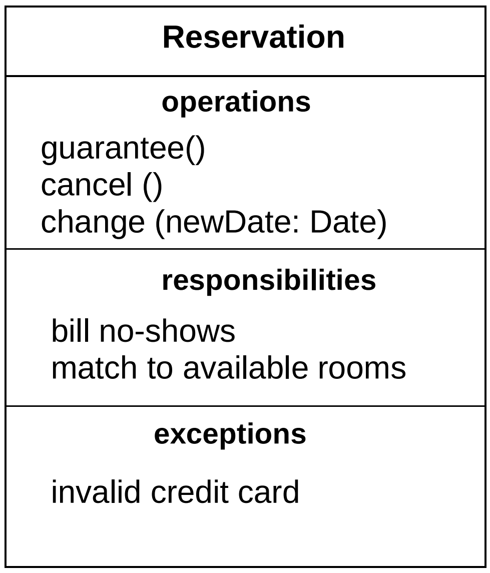


Figure 3-23 Compartments with Names

3.24.4 Mapping

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

3.25 Attribute

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

3.25.1 Semantics

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

The type of an attribute is a Classifier.

3.25.2 Notation

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

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

- Where *visibility* is one of:

  + public visibility

  # protected visibility

  - private visibility

  ~ .package visibility

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

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

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

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

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

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

class-scope-attribute

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

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

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

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

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

name : String [0..1]

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

3.25.3 Presentation Options

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

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

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

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

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

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

3.25.4 Style Guidelines

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

3.25.5 Example

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

3.25.6 Mapping

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

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

3.26 Operation

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

3.26.1 Semantics

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

3.26.2 Notation

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

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

- Where *visibility* is one of:

  + public visibility

  # protected visibility

- - private visibility
- ~ package visibility

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.26.3 Presentation Options

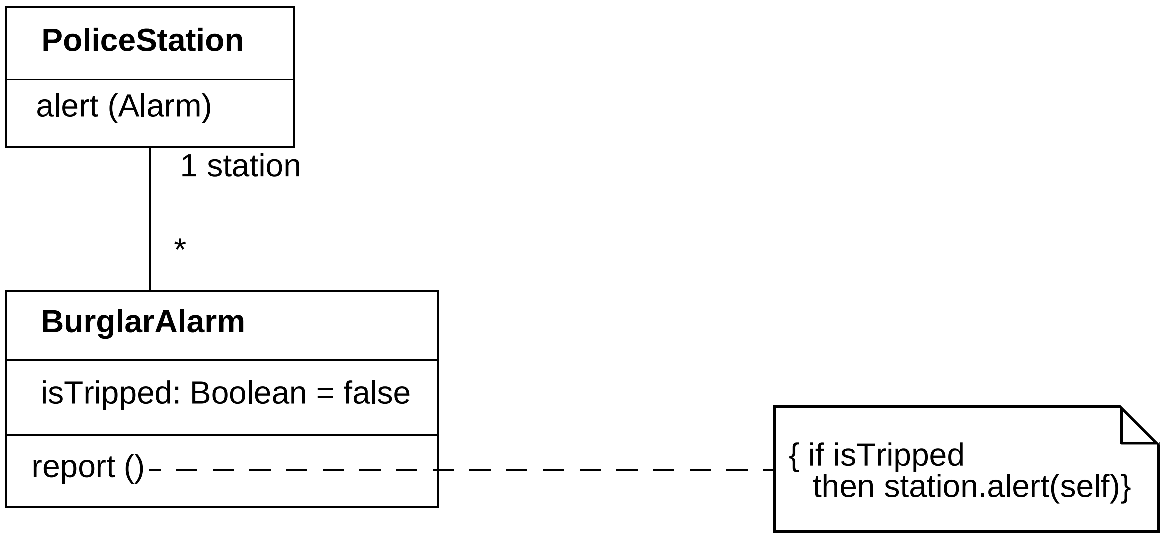
The argument list and return type may be suppressed (together, not separately).

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

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

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

.



PoliceStation
alert (Alarm)
1 station
*
BurglarAlarm
isTripped: Boolean = false
report ()
{ if isTripped then station.alert(self)}

Figure 3-24 Note showing method body

3.26.4 Style Guidelines

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

3.26.5 Example

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

Figure 3-25 Operation List with a Variety of Operations

3.26.6 Mapping

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

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

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

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

3.27 Nested Class Declarations

3.27.1 Semantics

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

3.27.2 Notation

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

3.27.3 Mapping

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

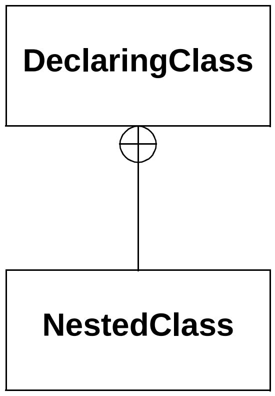


Figure 3-26 Nested class declaration

3.28 Type and Implementation Class

3.28.1 Semantics

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

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

3.28.2 Notation

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

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

3.28.3 Example

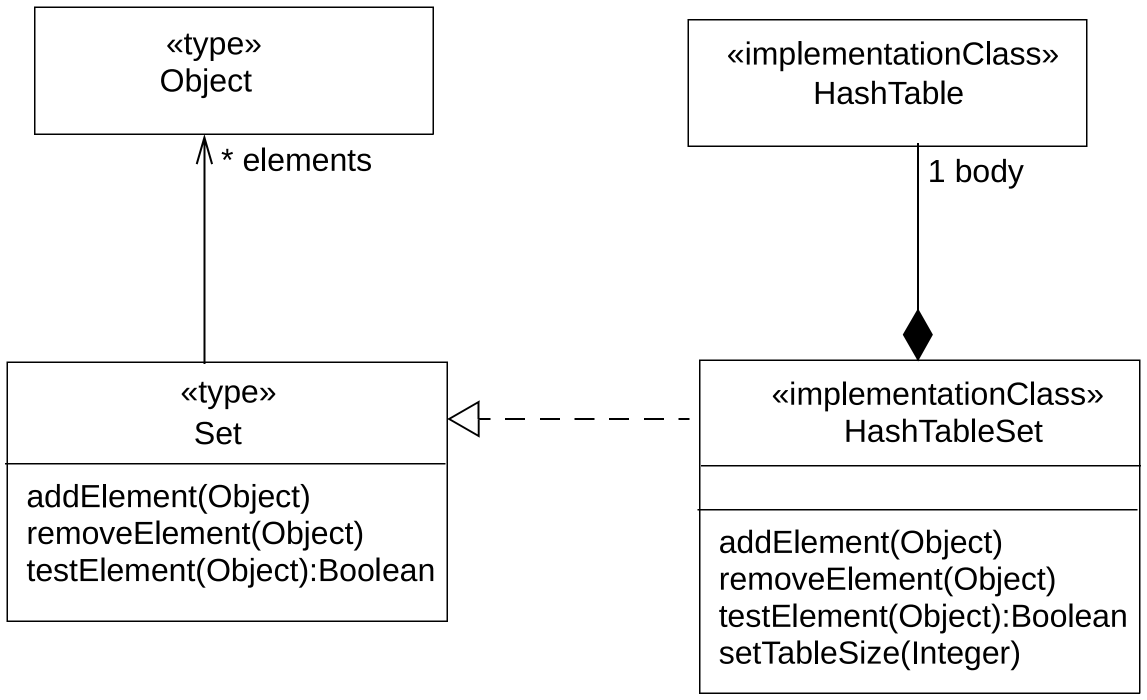


Figure 3-27 Notation for Types and Implementation Classes

3.28.4 Mapping

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

3.29 Interfaces

3.29.1 Semantics

An interface is a specifier for the externally-visible operations of a class, component, or other classifier (including subsystems) without specification of internal structure. Each interface often specifies only a limited part of the behavior of an actual class. Interfaces do not have implementation. They lack attributes, states, or associations; they only have operations. (An interface may be the target of a one-way association, however, but it may not have an association that it can navigate.) Interfaces may have generalization relationships. An interface is formally equivalent to an abstract class with no attributes and no methods and only abstract operations, but Interface is a peer of Class within the UML metamodel (both are Classifiers).

3.29.2 Notation

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

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

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

3.29.3 Example

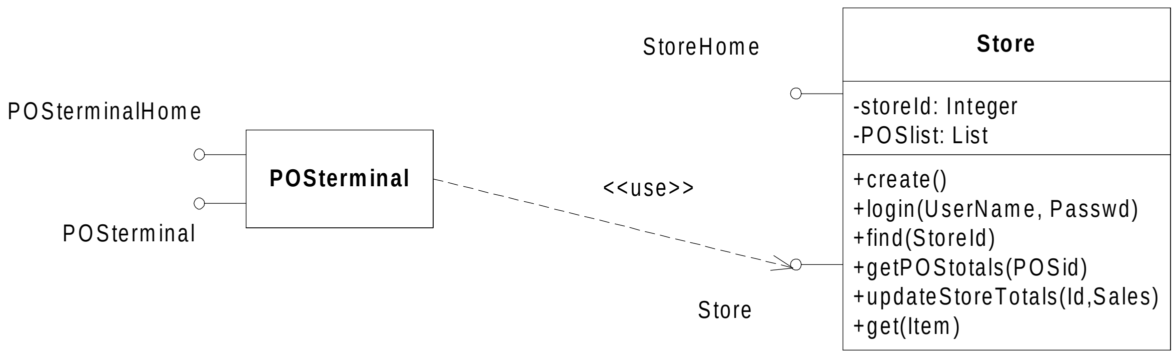


Figure 3-28 Shorthand Version of Interface Notation

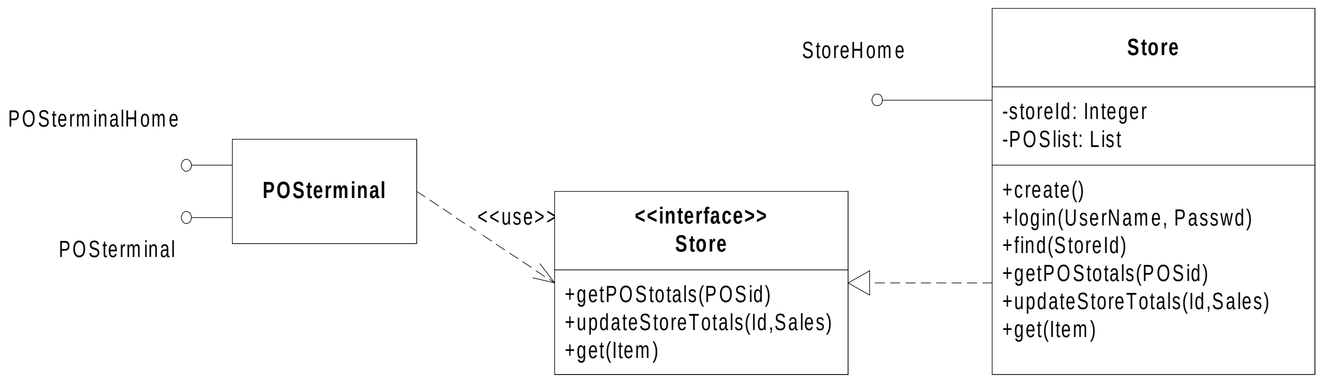


Figure 3-29 Longhand Version of Interface Notation

3.29.4 Mapping

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

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

3.30 Parameterized Class (Template)

3.30.1 Semantics

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

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

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

3.30.2 Notation

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

3.30.3 Presentation Options

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

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

name : *type* = *default-value*

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

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

3.30.4 Example

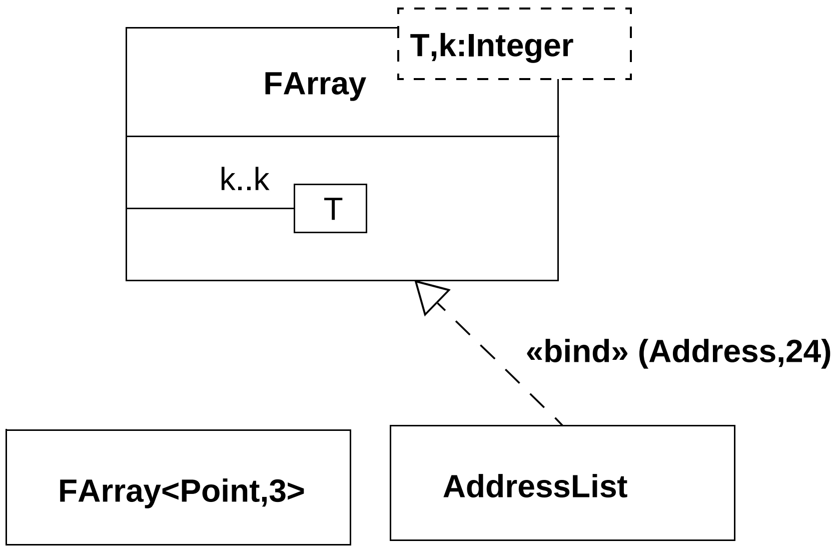


Figure 3-30 Template Notation with Use of Parameter as a Reference

3.30.5 Mapping

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

3.31 Bound Element

3.31.1 Semantics

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

3.31.2 Notation

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

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

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

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

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

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

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

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

3.31.3 Style Guidelines

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

3.31.4 Example

See Figure 3-30 on page 3-54.

3.31.5 Mapping

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

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

3.32 Utility

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

3.32.1 Semantics

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

3.32.2 Notation

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

3.32.3 Example

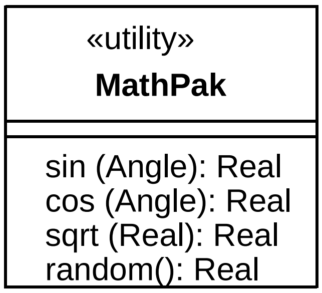


Figure 3-31 Notation for Utility

3.32.4 Mapping

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

3.33 Metaclass

3.33.1 Semantics

A metaclass is a class whose instances are classes.

3.33.2 Notation

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

3.33.3 Mapping

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

3.34 Enumeration

3.34.1 Semantics

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

3.34.2 Notation

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

3.34.3 Mapping

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

3.35 Stereotype Declaration

3.35.1 Semantics

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

3.35.2 Notation

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

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

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

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

«metaclass» metaclassName

or

«stereotype» stereotypeName

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

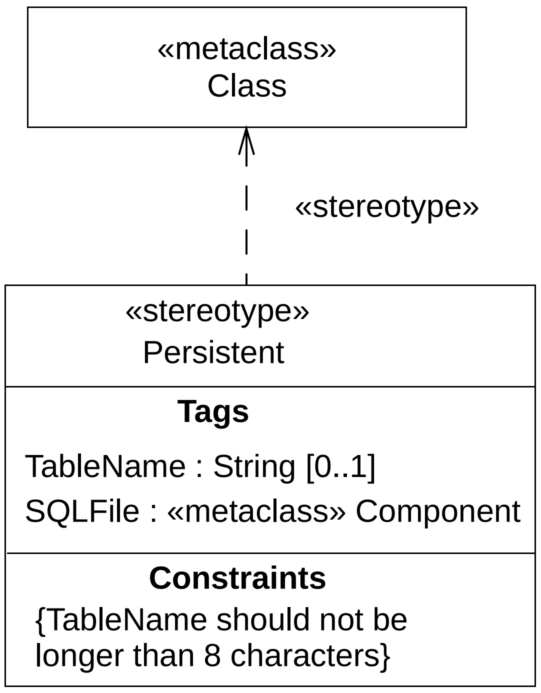

«metaclass»
Class
«stereotype»
«stereotype»
Persistent
Tags
TableName : String [0..1]
SQLFile : «metaclass» Component
Constraints
{TableName should not be longer than 8 characters}

Figure 3-32 Notational form for declaring a stereotype

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

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

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

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

Figure 3-33 Tabular form for specifying stereotypes

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

Figure 3-34 Tabular form for specifying tags

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

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

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

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

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

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

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

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

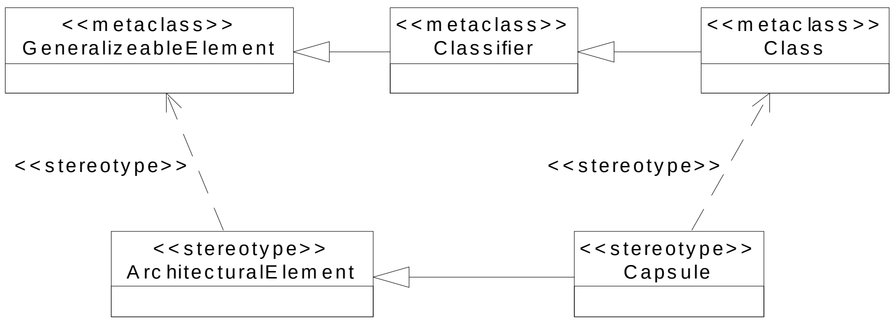


Figure 3-35 Graphical equivalent of the stereotype declarations shown in Figure 3-34

3.35.3 Mapping

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

3.36 Powertype

3.36.1 Semantics

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

3.36.2 Notation

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

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

`discriminatorName: powertypeName`,

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

3.36.3 Mapping

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

3.37 Class Pathnames

3.37.1 Notation

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

package-name :: *class-name*

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

3.37.2 Example

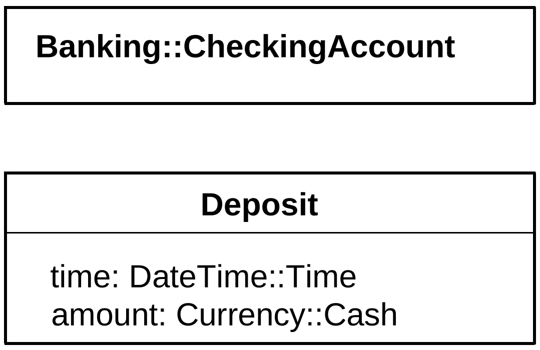


Figure 3-36 Pathnames for Classes in Other Packages

3.37.3 Mapping

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

3.38 Accessing or Importing a Package

3.38.1 Semantics

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

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

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

3.38.2 Notation

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

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

3.38.3 Example

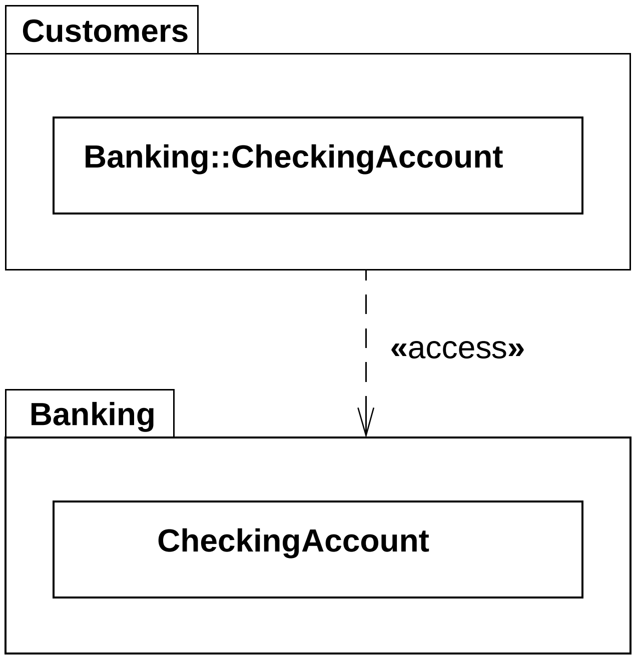


Figure 3-37 Access Dependency Among Packages

3.38.4 Mapping

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

3.39 Object

3.39.1 Semantics

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

3.39.2 Notation

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

An object shown as a rectangle with two compartments.

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

objectname : *classname*

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

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

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

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

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

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

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

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

attributename : *type* = *value*

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

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

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

3.39.3 Presentation Options

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

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

The attribute value compartment as a whole may be suppressed.

Attributes whose values are not of interest may be suppressed.

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

3.39.4 Style Guidelines

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

3.39.5 Variations

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

3.39.6 Example

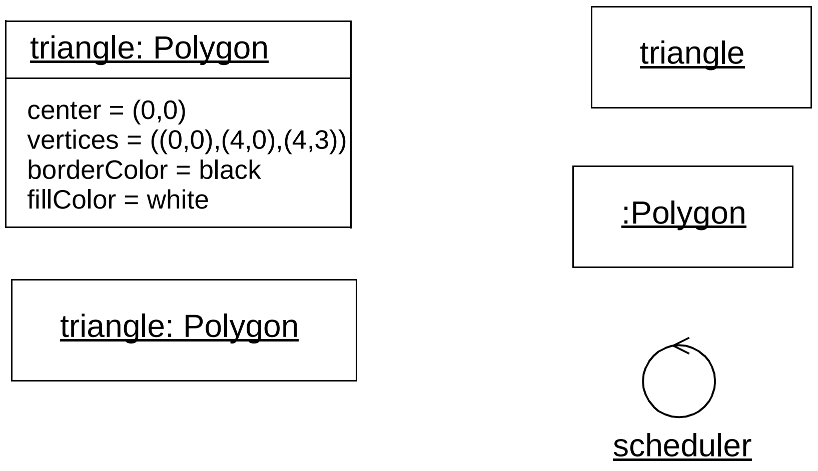


Figure 3-38 Objects

3.39.7 Mapping

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

3.40 Composite Object

3.40.1 Semantics

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

3.40.2 Notation

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

3.40.3 Example

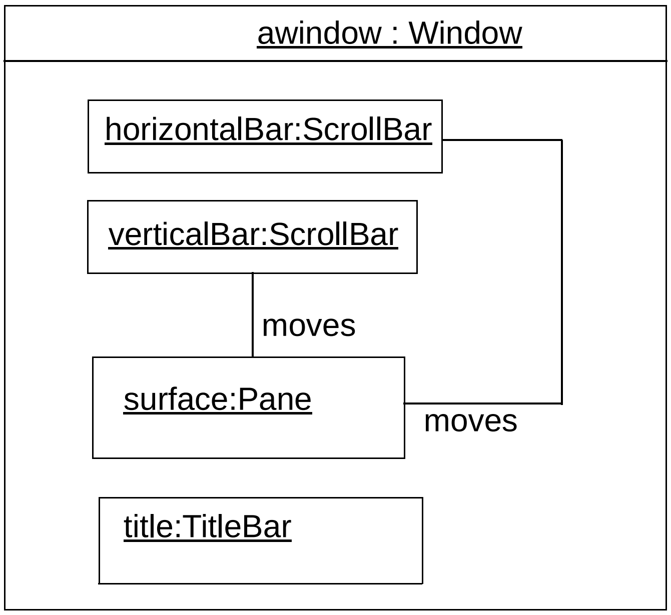


Figure 3-39 Composite Objects

3.40.4 Mapping

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

3.41 Association

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

3.42 Binary Association

3.42.1 Semantics

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

3.42.2 Notation

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

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

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

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

3.42.2.1 association name

Designates the (optional) name of the association.

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

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

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

3.42.2.2 association class symbol

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

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

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

3.42.3 Presentation Options

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

3.42.4 Style Guidelines

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

3.42.5 Options

3.42.5.1 Xor-association

An xor-constraint indicates a situation in which only one of several potential associations may be instantiated at one time for any single instance. This is shown as a dashed line connecting two or more associations, all of which must have a classifier in common, with the constraint string “{xor}” labeling the dashed line. Any instance of the classifier may only participate in one of the associations at one time. Each rolename must be different. (This is simply a predefined use of the constraint notation.)

3.42.6 Example

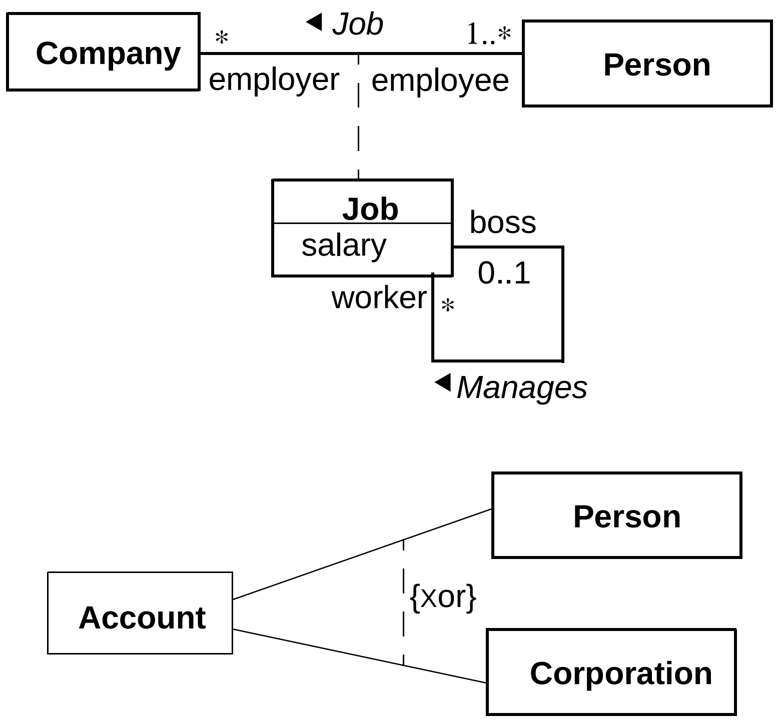


Figure 3-40 Association Notation

3.42.7 Mapping

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

3.43 Association End

3.43.1 Semantics

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

3.43.2 Notation

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

3.43.2.1 multiplicity

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

3.43.2.2 ordering

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

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

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

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

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

3.43.2.3 qualifier

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

3.43.2.4 navigability

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

3.43.2.5 aggregation indicator

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

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

3.43.2.6 rolename

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

3.43.2.7 interface specifier

The name of a Classifier with the syntax:

‘:’ *classifiername*, . . .

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

The use of a rolename and interface specifier are equivalent to creating a small collaboration that includes just an association and two roles, whose structure is defined by the rolename and attached classifier on the original association. Therefore, the original association and classifiers are a use of the collaboration. The original classifier must be compatible with the interface specifier (which can be an interface or a type, among other kinds of classifiers).

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

3.43.2.8 changeability

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

3.43.2.9 visibility

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

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

3.43.3 Presentation Options

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

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

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

3.43.4 Style Guidelines

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

- qualifier
- aggregation symbol
- navigation arrow

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

3.43.5 Example

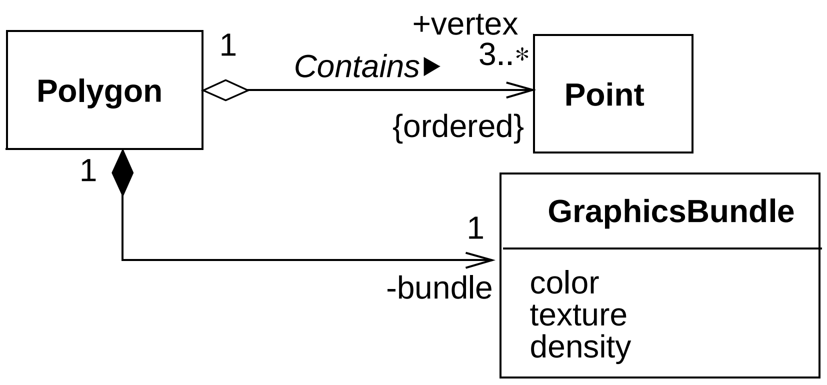


Figure 3-41 Various Adornments on Association Roles

3.43.6 Mapping

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

3.44 Multiplicity

3.44.1 Semantics

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

3.44.2 Notation

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

lower-bound .. *upper-bound*

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

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

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

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

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

3.44.3 Style Guidelines

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

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

3.44.4 Example

0..1

1

0..*

*

1..*

1..6

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

3.44.5 Mapping

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

3.45 Qualifier

3.45.1 Semantics

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

3.45.2 Notation

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

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

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

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

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

3.45.3 Presentation Options

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

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

3.45.4 Style Guidelines

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

3.45.5 Example

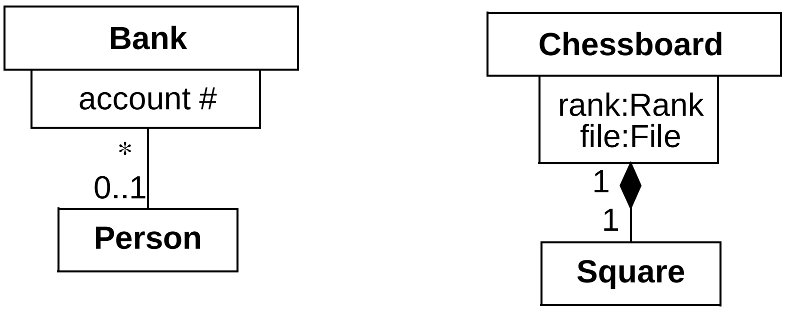


Figure 3-42 Qualified Associations

3.45.6 Mapping

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

3.46 Association Class

3.46.1 Semantics

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

3.46.2 Notation

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

3.46.3 Presentation Options

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

3.46.4 Style Guidelines

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

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

3.46.5 Example

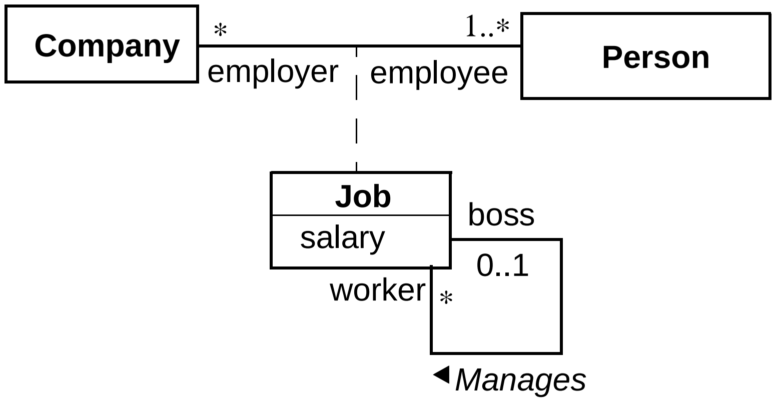


Figure 3-43 Association Class

3.46.6 Mapping

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

3.47 N-ary Association

3.47.1 Semantics

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

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

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

3.47.2 Notation

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

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

3.47.3 Style Guidelines

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

3.47.4 Example

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

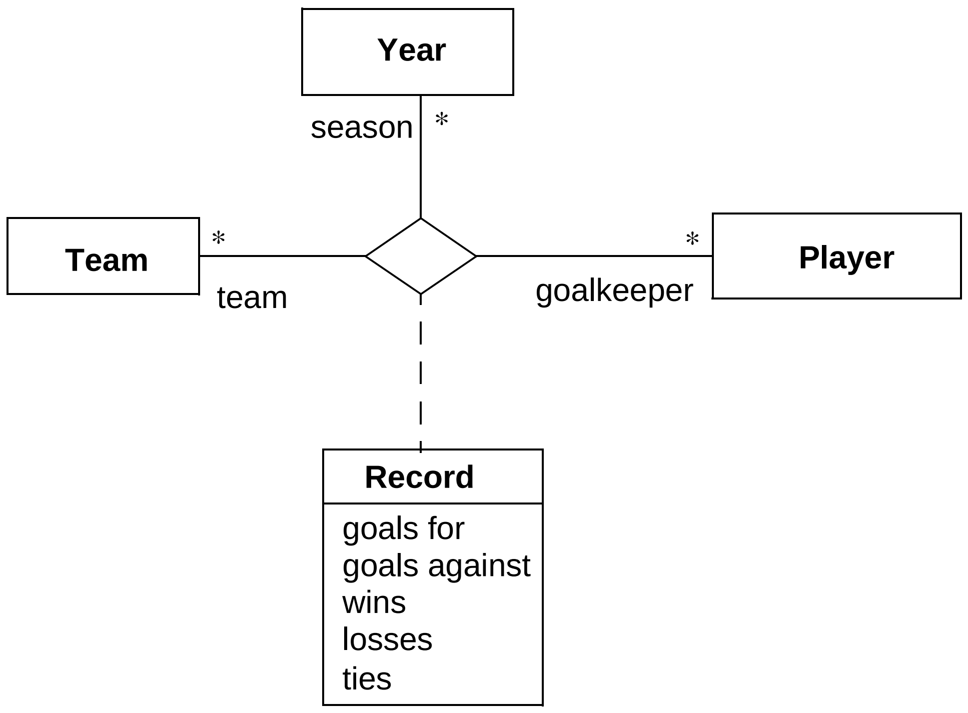


Figure 3-44 Ternary association that is also an association class

3.47.5 Mapping

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

3.48 Composition

3.48.1 Semantics

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

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

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

3.48.2 Notation

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

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

rolename ‘:’ *classname*

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

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

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

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

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

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

3.48.3 Design Guidelines

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

3.48.4 Example

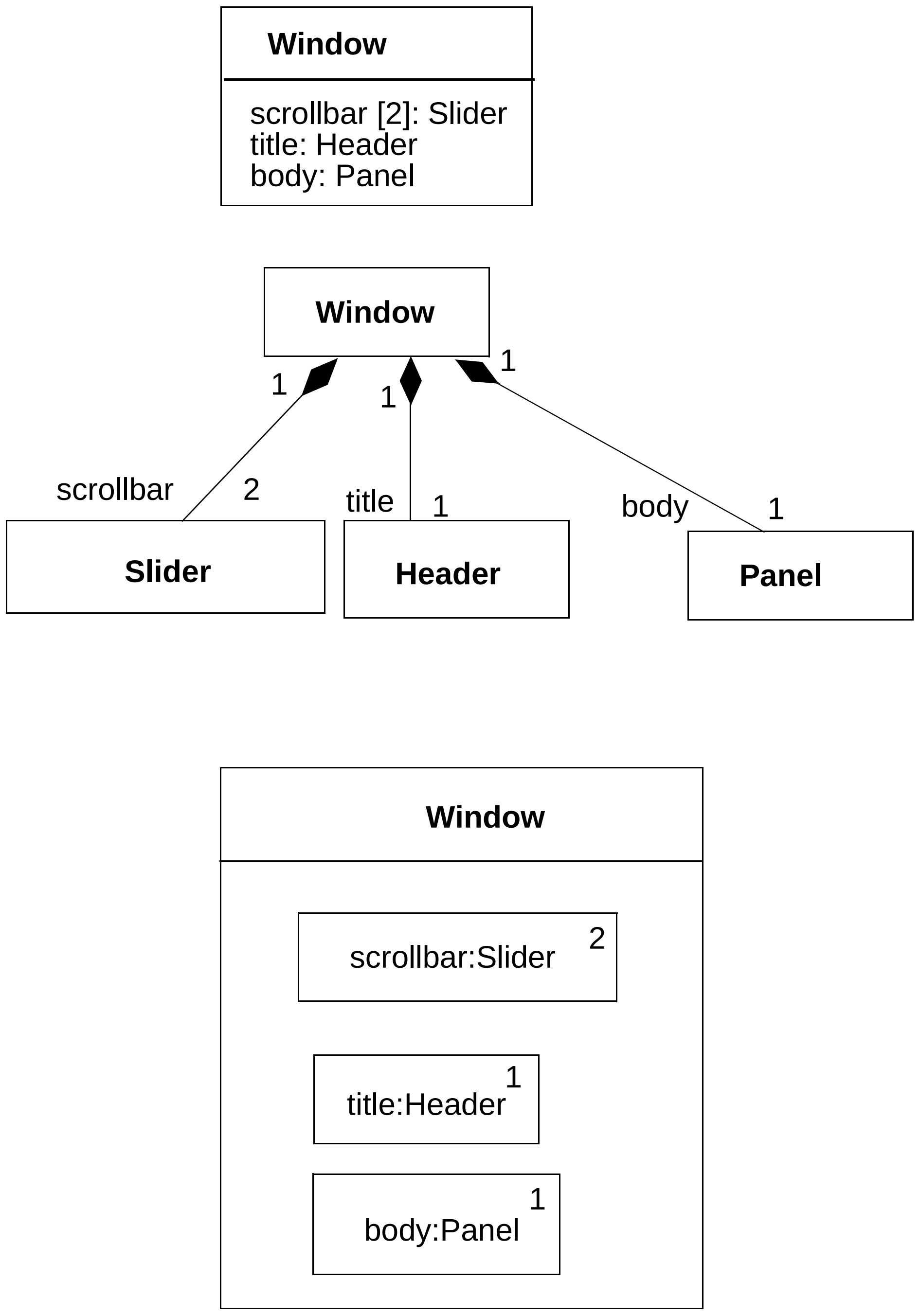


Figure 3-45 Different Ways to Show Composition

3.48.5 Mapping

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

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

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

3.49 Link

3.49.1 Semantics

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

3.49.2 Notation

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

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

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

3.49.2.1 Implementation stereotypes

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

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

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

3.49.2.2 N-ary link

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

3.49.3 Example

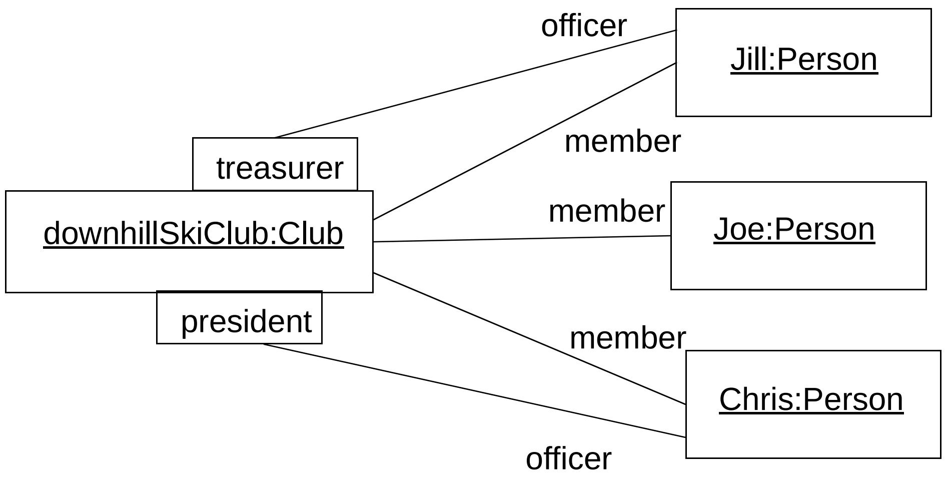


Figure 3-46 Links

3.49.4 Mapping

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

3.50 Generalization

3.50.1 Semantics

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

3.50.2 Notation

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

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

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

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

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

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

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

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

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

3.50.3 Presentation Options

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

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

3.50.4 Example

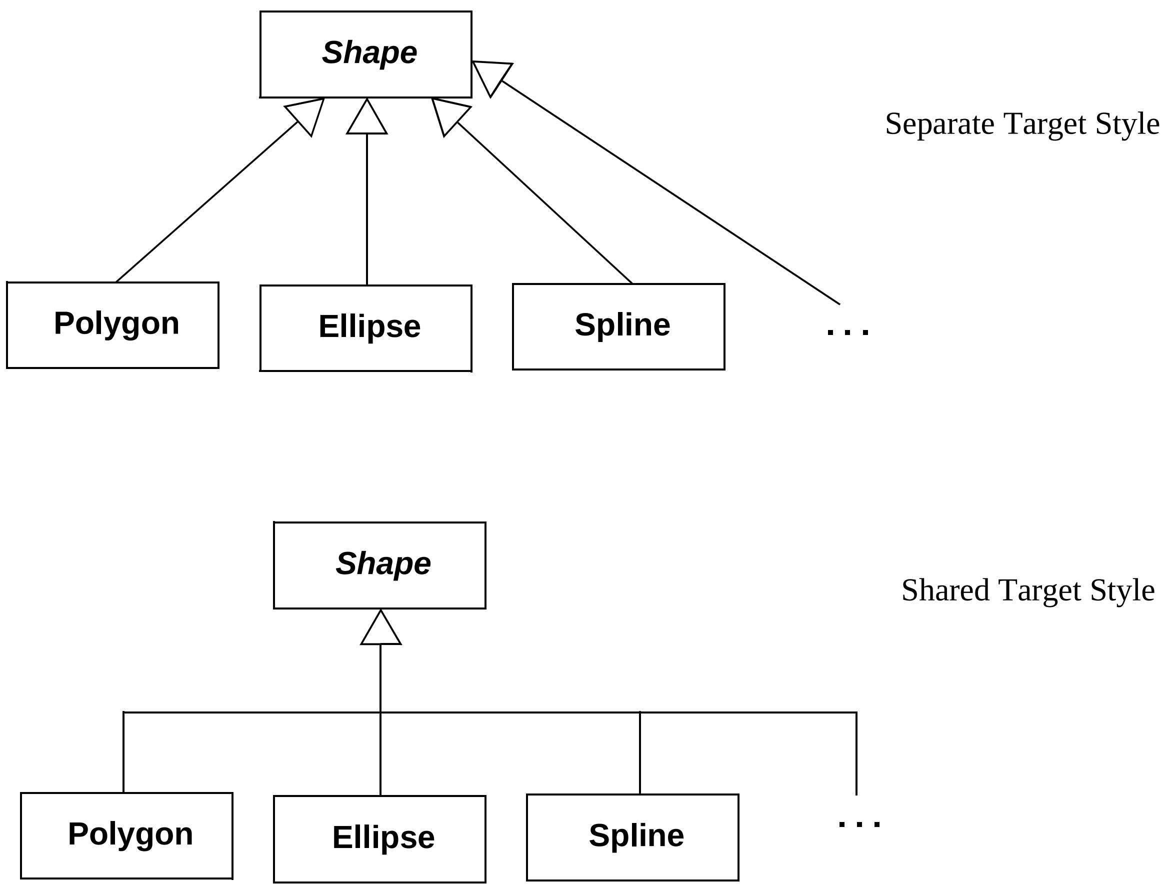


Figure 3-47 Styles of Displaying Generalizations

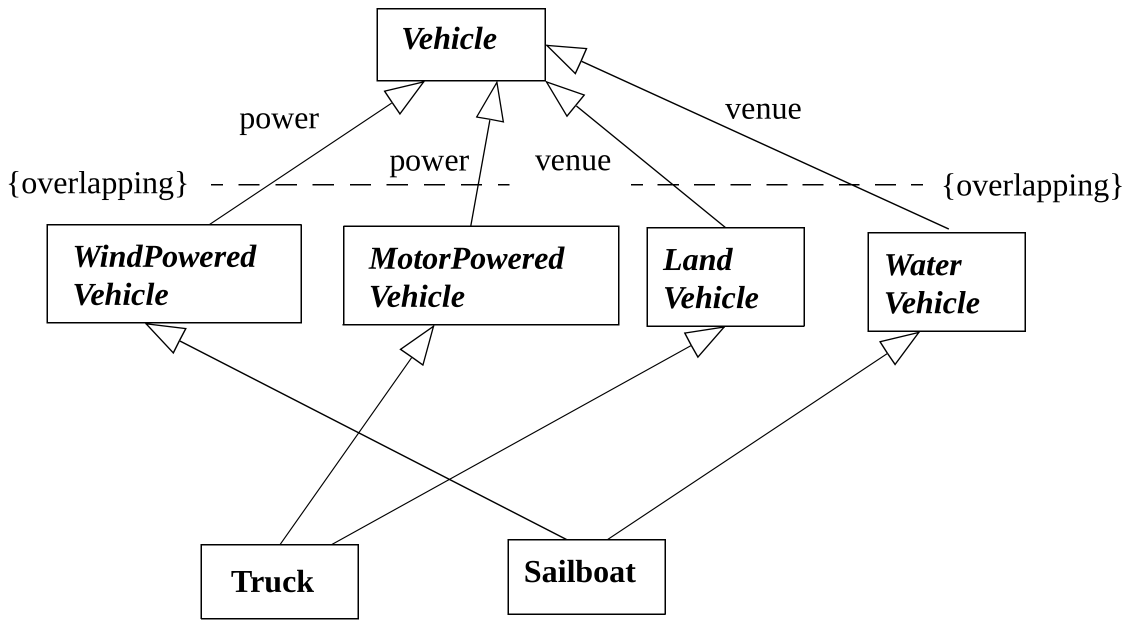


Figure 3-48 Generalization with Discriminators and Constraints, Separate Target Style

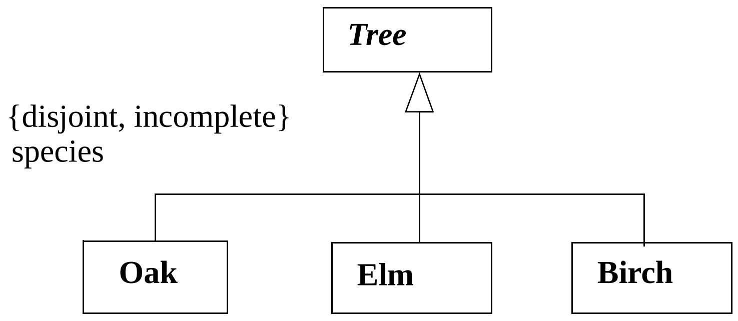


Figure 3-49 Generalization with Shared Target Style

3.50.5 Mapping

Each generalization path between two element symbols maps into a Generalization between the corresponding GeneralizableElements. A generalization tree with one arrowhead and many tails maps into a set of Generalizations, one between each element corresponding to a symbol on a tail and the single GeneralizableElement corresponding to the symbol on the head. That is, a tree is semantically indistinguishable from a set of distinct arrows, it is purely a notational convenience.

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

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

3.51 Dependency

3.51.1 Semantics

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

3.51.2 Notation

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

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

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

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

3.51.3 *Presentation Options*

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

3.51.4 Example

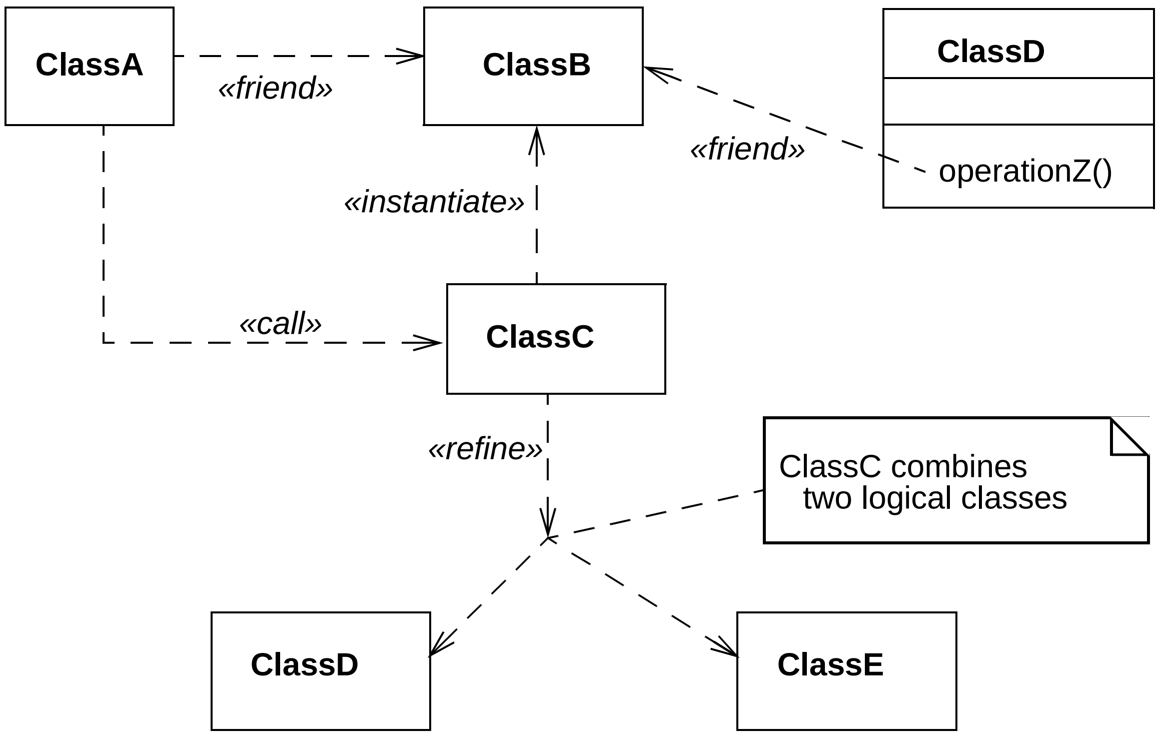


Figure 3-50 Various Dependencies Among Classes

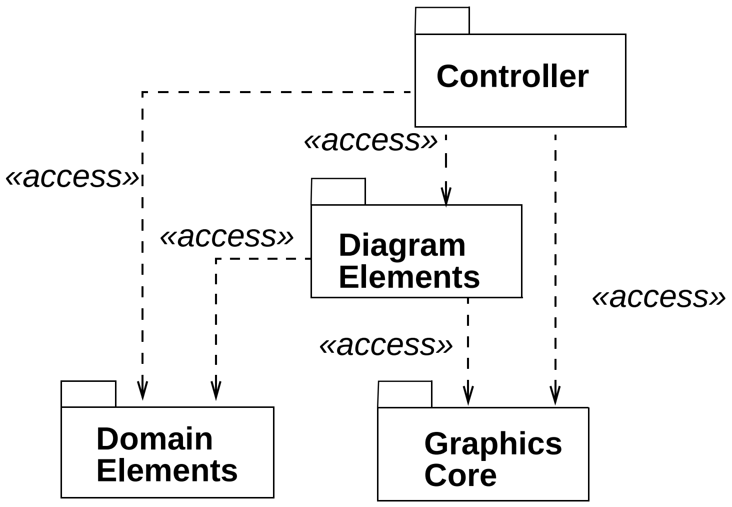


Figure 3-51 Dependencies Among Packages

3.51.5 Mapping

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

3.52 Derived Element

3.52.1 Semantics

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

3.52.2 Notation

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

3.52.3 Style Guidelines

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

3.53 InstanceOf

3.53.1 Semantics

Shows the connection between an instance and its classifier.

3.53.2 Notation

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

3.53.3 Mapping

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

Part 6 - Use Case Diagrams

A use case diagram shows the relationship among use cases within a system or other semantic entity and their actors.

3.54 Use Case Diagram

3.54.1 Semantics

Use case diagrams show actors and use cases together with their relationships. The use cases represent functionality of a system or a classifier, like a subsystem or a class, as manifested to external interactors with the system or the classifier.

3.54.2 Notation

A use case diagram is a graph of actors, a set of use cases, possibly some interfaces, and the relationships between these elements. The relationships are associations between the actors and the use cases, generalizations between the actors, and generalizations, extends, and includes among the use cases. The use cases may optionally be enclosed by a rectangle that represents the boundary of the containing system or classifier.

3.54.3 Example

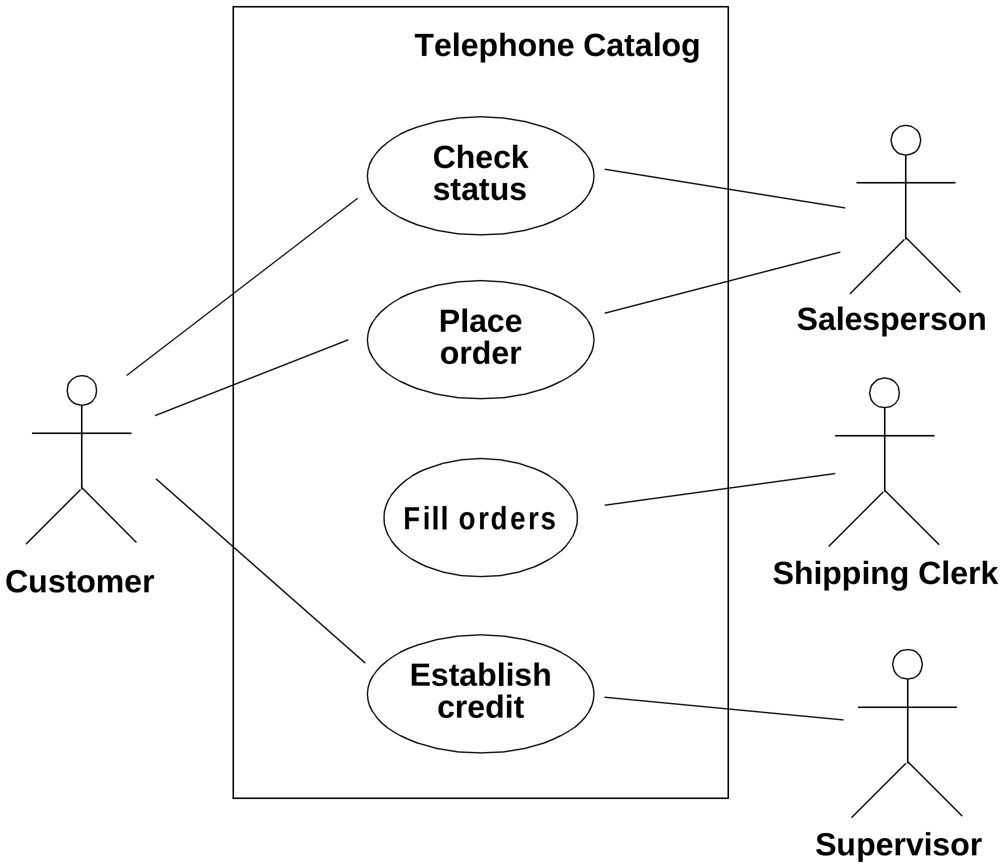


Figure 3-52 Use Case Diagram

3.54.4 Mapping

A set of use case ellipses with connections to actor symbols maps to a set of UseCases and Actors corresponding to the use case and actor symbols, respectively. The optional rectangle maps onto either a Model with the stereotype «useCaseModel» containing the set of UseCases and Actors, or to a Classifier, like Subsystem or Class, containing the set of UseCases. An interface in the diagram is mapped onto an Interface in the Model, and the connection between the interface and the actor or use case icons is mapped onto a realization Dependency (an Abstraction dependency being stereotyped «realize») between the Classifiers. Each generalization arrow maps onto a Generalization in the model, and each line between an actor symbol and a use case ellipse maps to an Association between the corresponding Classifiers. A dashed arrow with the keyword «include» or «extend» maps to an Include or Extend relationship between UseCases.

3.55 Use Case

3.55.1 Semantics

A *use case* is a kind of classifier representing a coherent unit of functionality provided by a system, a subsystem, or a class as manifested by sequences of messages exchanged among the system (subsystem, class) and one or more outside interactors (called *actors*) together with actions performed by the system (subsystem, class).

An *extension point* is a reference to one location within a use case at which action sequences from other use cases may be inserted. Each extension point has a unique name within a use case, and a description of the location within the behavior of the use case.

3.55.2 Notation

A use case is shown as an ellipse containing the name of the use case. An optional stereotype keyword may be placed above the name and a list of properties included below the name. As a classifier, a use case may also have compartments displaying attributes and operations.

Extension points may be listed in a compartment of the use case with the heading **extension points**. The description of the locations of the extension point is given in a suitable form, usually as ordinary text, but can also be given in other forms, like the name of a state in a state machine, or a precondition or a postcondition.

The behavior of a use case can be described in several different ways, depending on what is convenient. Plain text is used often, but state machines, operations, and methods are examples of other ways of describing the behavior of the use case. Sequence diagrams can be used for describing the interaction between use cases and their actors.

3.55.3 Presentation Options

The name of the use case may be placed below the ellipse. The name of an abstract use case may be shown in italics.

The ellipse may contain or suppress compartments presenting the attributes, the operations, and the extension points of the use case.

3.55.4 Style Guidelines

Use case names should follow capitalization and punctuation guidelines used for Classifiers in the model.

3.55.5 Mapping

A use case symbol maps to a UseCase with the given name. An extension point maps into an ExtensionPoint within the UseCase.

3.56 Actor

3.56.1 Semantics

An actor defines a coherent set of roles that users of an entity can play when interacting with the entity. An actor may be considered to play a separate role with regard to each use case with which it communicates.

3.56.2 Notation

The standard stereotype icon for an actor is a “stick man” figure with the name of the actor below the figure.

3.56.3 Presentation Options

An actor may also be shown as a class rectangle with the keyword «actor», with the usual notation for all compartments. Other icons that convey the kind of actor may also be used to denote an actor.

3.56.4 Style Guidelines

Actor names should follow capitalization and punctuation guidelines used for types and classes in the model.

3.56.5 Mapping

An actor symbol maps to an Actor with the given name. The names of abstract actors may be shown in italics.

3.57 Use Case Relationships

3.57.1 Semantics

There are several standard relationships among use cases or between actors and use cases.

- Association – The participation of an actor in a use case; that is, instances of the actor and instances of the use case communicate with each other. This is the only relationship between actors and use cases.

- Extend – An extend relationship from use case A to use case B indicates that an instance of use case B may be augmented (subject to specific conditions specified in the extension) by the behavior specified by A. The behavior is inserted at the location defined by the extension point in B, which is referenced by the extend relationship.
- Generalization – A generalization from use case C to use case D indicates that C is a specialization of D.
- Include – An include relationship from use case E to use case F indicates that an instance of the use case E will also contain the behavior as specified by F. The behavior is included at the location which defined in E.

3.57.2 Notation

An association between an actor and a use case is shown as a solid line between the actor and the use case. It may have end adornments such as multiplicity.

An extend relationship between use cases is shown by a dashed arrow with an open arrow-head from the use case providing the extension to the base use case. The arrow is labeled with the keyword «extend». The condition of the relationship is optionally presented close to the key-word.

An include relationship between use cases is shown by a dashed arrow with an open arrow-head from the base use case to the included use case. The arrow is labeled with the keyword «include».

A generalization between use cases is shown by a generalization arrow; that is, a solid line with a closed, hollow arrow head pointing at the parent use case.

The relationship between a use case and its external interaction sequences is usually defined by an invisible hyperlink to sequence diagrams. The relationship between a use case and its realization may be shown as dashed arrow with the keyword «representedClassifier» to collaborations, but may also be defined as invisible hyperlinks.

3.57.3 Example

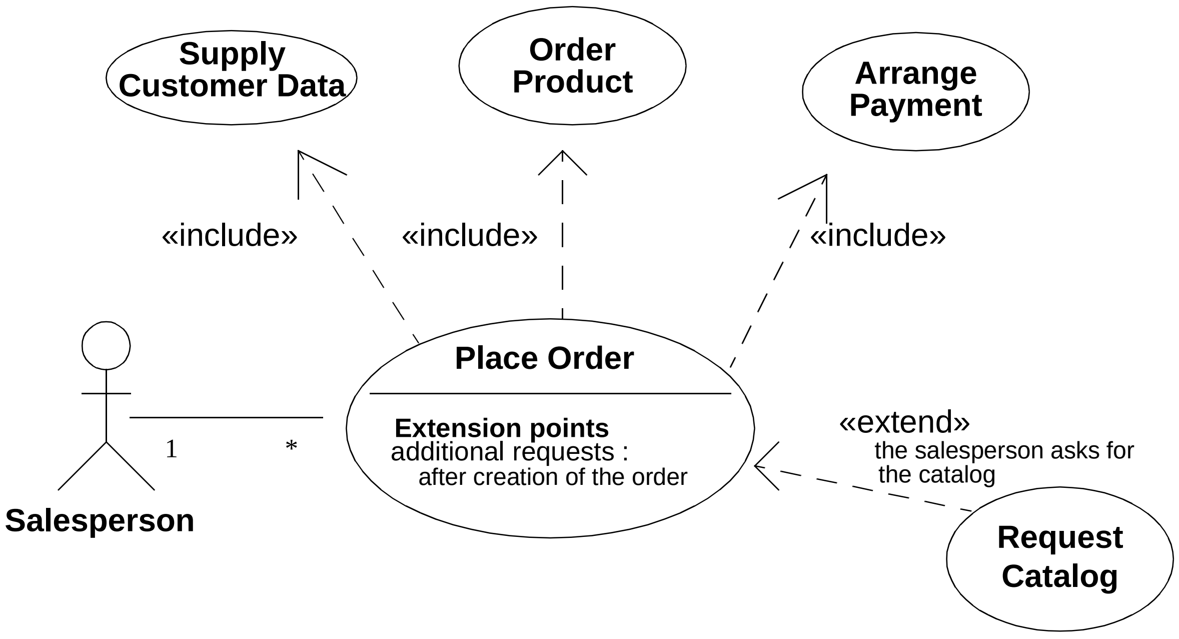


Figure 3-53 Use Case Relationships

3.57.4 Mapping

A path between use case and/or actor symbols maps into the corresponding relationship between the corresponding Elements, as described above.

3.58 Actor Relationships

3.58.1 Semantics

There is one standard relationship among actors and one between actors and use cases.

- Association – The participation of an actor in a use case; that is, instances of the actor and instances of the use case communicate with each other. This is the only relationship between actors and use cases.
- Generalization – A generalization from an actor A to an actor B indicates that an instance of A can communicate with the same kinds of use-case instances as an instance of B.

3.58.2 Notation

An association between an actor and a use case is shown as a solid line between the actor and the use case.

A generalization between actors is shown by a generalization arrow; that is, a solid line with a closed, hollow arrow head. The arrow head points at the more general actor.

3.58.3 Example

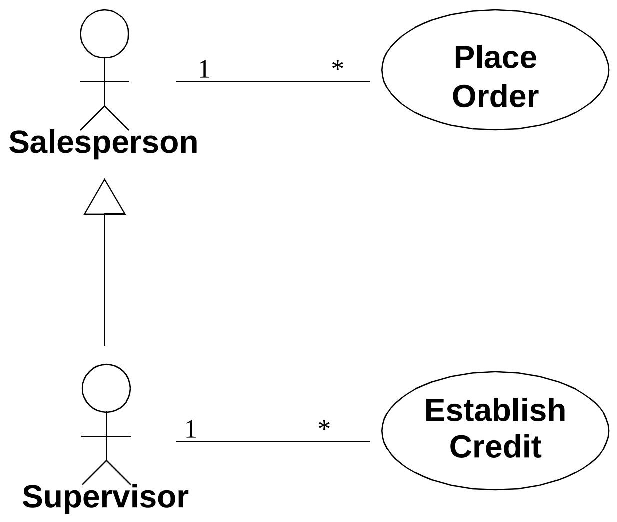


Figure 3-54 Actor Relationships

3.58.4 Mapping

A generalization between two actor symbols and an association between actor symbol and a use case symbol maps into the corresponding relationship between the corresponding Elements, as described above.

Part 7 - Interaction Diagrams

The description of behavior involves two aspects: 1) the structural description of the participants and 2) the description of the communication patterns. The structure of Instances playing roles in a behavior and their relationships is called a *Collaboration*. The communication pattern performed by Instances playing the roles to accomplish a specific purpose is called an *Interaction*. The two aspects of behavior are often described together on a single diagram, but at times it is useful to describe the structural aspects separately.

Interaction diagrams come in two forms based on the same underlying information, specified by a Collaboration and possibly by an Interaction, but each form emphasizes a particular aspect of it. The two forms are *sequence diagrams* and *collaboration diagram*s. A sequence diagram shows the explicit sequence of communications and is better for real-time specifications and for complex scenarios. A collaboration diagram shows an Interaction organized around the roles in the Interaction and their relationships. It does not show time as a separate dimension, so the sequence of communications and the concurrent threads must be determined using sequence numbers.

3.59 Collaboration

3.59.1 Semantics

Behavior is implemented by ensembles of Instances that exchange Stimuli within an overall interaction to accomplish a task. To understand the mechanisms used in a design, it is important to see only those Instances and their cooperation involved in accomplishing a purpose or a related set of purposes, projected from the larger system of which they are part of. Such a static construct is called a *Collaboration*.

A Collaboration includes an ensemble of ClassifierRoles and AssociationRoles that define the participants needed for a given set of purposes. Instances conforming to the ClassifierRoles play the roles defined by the ClassifierRoles, while Links between the Instances conform to AssociationRoles of the Collaboration. ClassifierRoles and AssociationRoles define a usage of Instances and Links, and the Classifiers and Associations declare all required properties of these Instances and Links.

An *Interaction* is defined in the context of a Collaboration. It specifies the communication patterns between the roles in the Collaboration. More precisely, it contains a set of partially ordered *Messages*, each specifying one communication; for example, what Signal to be sent or what Operation to be invoked, as well as the roles to be played by the sender and the receiver, respectively.

A *CollaborationInstanceSet* references an ensemble of Instances that jointly perform the task specified by the CollaborationInstanceSet's Collaboration. These Instances play the roles defined by the ClassifierRoles of the Collaboration; that is, the Instances have all the properties declared by the ClassifierRoles (the Instances are said to *conform to* the ClassifierRoles). The Stimuli sent between the Instances when performing the task are participating in the *InteractionInstanceSet* of the CollaborationInstanceSet. These Stimuli conform to the Messages in one of the Interactions of the Collaboration. Since an Instance can participate in several CollaborationInstanceSets at the same time, all its communications are not necessarily referenced by only one InteractionInstanceSet. They can be interleaved.

A Collaboration may be attached to an Operation or a Classifier, like a UseCase, to describe the context in which their behavior occurs; that is, what roles Instances play to perform the behavior specified by the Operation or the UseCase. A Collaboration is used for describing the realization of an Operation or a Classifier. A Collaboration that describes a Classifier, like a UseCase, references Classifiers and Associations in general, while a Collaboration describing an Operation includes the arguments and local variables of the Operation, as well as ordinary Associations attached to the Classifier owning the Operation. The Interactions defined within the Collaboration specify the communication pattern between the Instances when they perform the behavior specified in the Operation or the UseCase. These patterns are presented in sequence diagrams or collaboration diagrams. A Collaboration may also be attached to a Class to define the static structure of the Class; that is, how attributes, parameters, etc. cooperate with each other.

A parameterized Collaboration represents a design construct that can be used repeatedly in different designs. The participants in the Collaboration, including the Classifiers and Relationships, can be parameters of the generic Collaboration. The parameters are bound to particular ModelElements in each instantiation of the generic Collaboration. Such a parameterized Collaboration can capture the structure of a *design pattern* (note that a design pattern involves more than structural aspects). Whereas most Collaborations can be anonymous because they are attached to a named ModelElement, Collaboration patterns are free standing design constructs that must have names.

A Collaboration may be expressed at different levels of granularity. A coarse-grained Collaboration may be refined to produce another Collaboration that has a finer granularity.

3.60 Sequence Diagram

3.60.1 Semantics

A sequence diagram presents an Interaction, which is a set of Messages between ClassifierRoles within a Collaboration, or an InteractionInstanceSet, which is a set of Stimuli between Instances within a CollaborationInstanceSet to effect a desired operation or result.

3.60.2 Notation

A sequence diagram has two dimensions: the vertical dimension represents time, and the horizontal dimension represents different instances. Normally time proceeds down the page. (The dimensions may be reversed, if desired.) Usually only time sequences are important, but in real-time applications the time axis could be an actual metric. There is no significance to the horizontal ordering of the instances.

The different kinds of arrows used in sequence diagrams are described in Section 3.63, “Message and Stimulus,” on page 3-111. These are the same kinds as in collaboration diagrams; see Section 3.65, “Collaboration Diagram,” on page 3-114.

Note that much of this notation is drawn directly from the Object Message Sequence Chart notation of Buschmann, Meunier, Rohnert, Sommerlad, and Stal, which is itself derived with modifications from the Message Sequence Chart notation.

3.60.3 Presentation Options

The horizontal ordering of the lifelines is arbitrary. Often call arrows are arranged to proceed in one direction across the page; however, this is not always possible and the ordering does not convey information.

The axes can be interchanged, so that time proceeds horizontally to the right and different objects are shown as horizontal lines.

Various labels (such as timing constraints, descriptions of actions during an activation, and so on) can be shown either in the margin or near the transitions or activations that they label.

Timing constraints may be expressed using time expressions on message or stimuli names. The functions *sendTime* (the time at which a stimulus is sent by an instance) and *receiveTime* (the time at which a stimulus is received by an instance) may be applied to stimuli names to yield a time. The set of time functions is open-ended, so that users can invent new ones as needed for special situations or implementation distinctions (such as *elapsedTime*, *executionStartTime*, *queuedTime*, *handledTime*, etc.)

Construction marks of the kind found in blueprints can be used to indicate a time interval to which a constraint may be attached (see bottom right of Figure 3-55 on page 3-104). This notation is visually appealing but it is ambiguous if the arrow is horizontal, because the send time and the receive time cannot be distinguished. In many cases the transmission time is negligible, so the ambiguity is harmless, but a tool must nevertheless map such a notation unambiguously to an expression on message or stimuli names (as shown in the examples in the left of the diagram) before the information is placed in the semantic model. (A tool may adopt defaults for this mapping.) Similarly, a tool might permit the time function to be elided and use the stimulus name to denote the time of stimulus sending or receipt within a timing expression (such as “b.receiveTime - a.sendTime < 1 sec.” in Figure 3-55), but again this is only a surface notation that must be mapped to a proper time expression in the semantic model).

3.60.4 Example

Simple sequence diagram with concurrent objects.

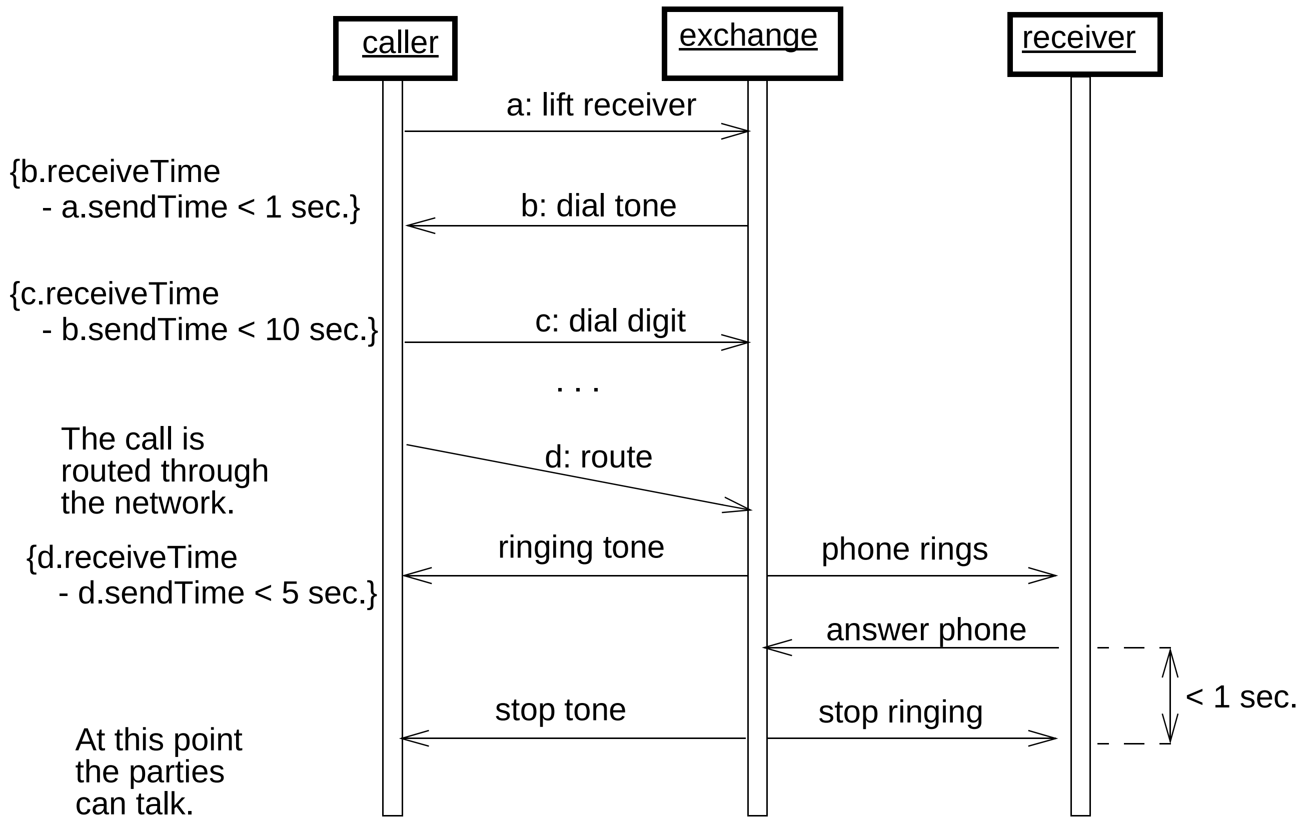


Figure 3-55 Simple Sequence Diagram with Concurrent Objects (denoted by boxes with thick borders).

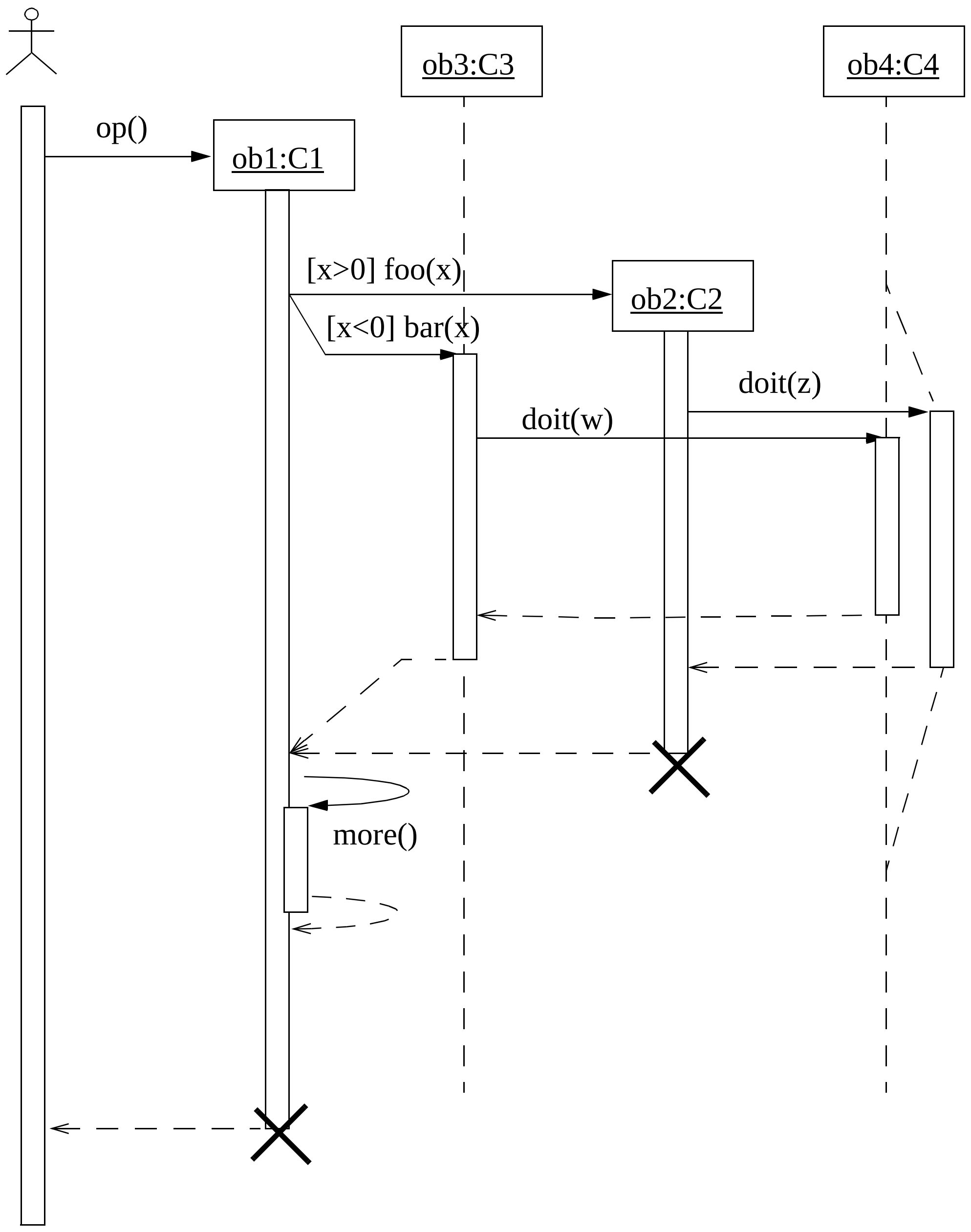


Figure 3-56 Sequence Diagram with Focus of Control, Conditional, Recursion, Creation, and Destruction.

3.60.5 Mapping

This section summarizes the mapping for the sequence diagram and the elements within it, some of which are described in subsequent sections.

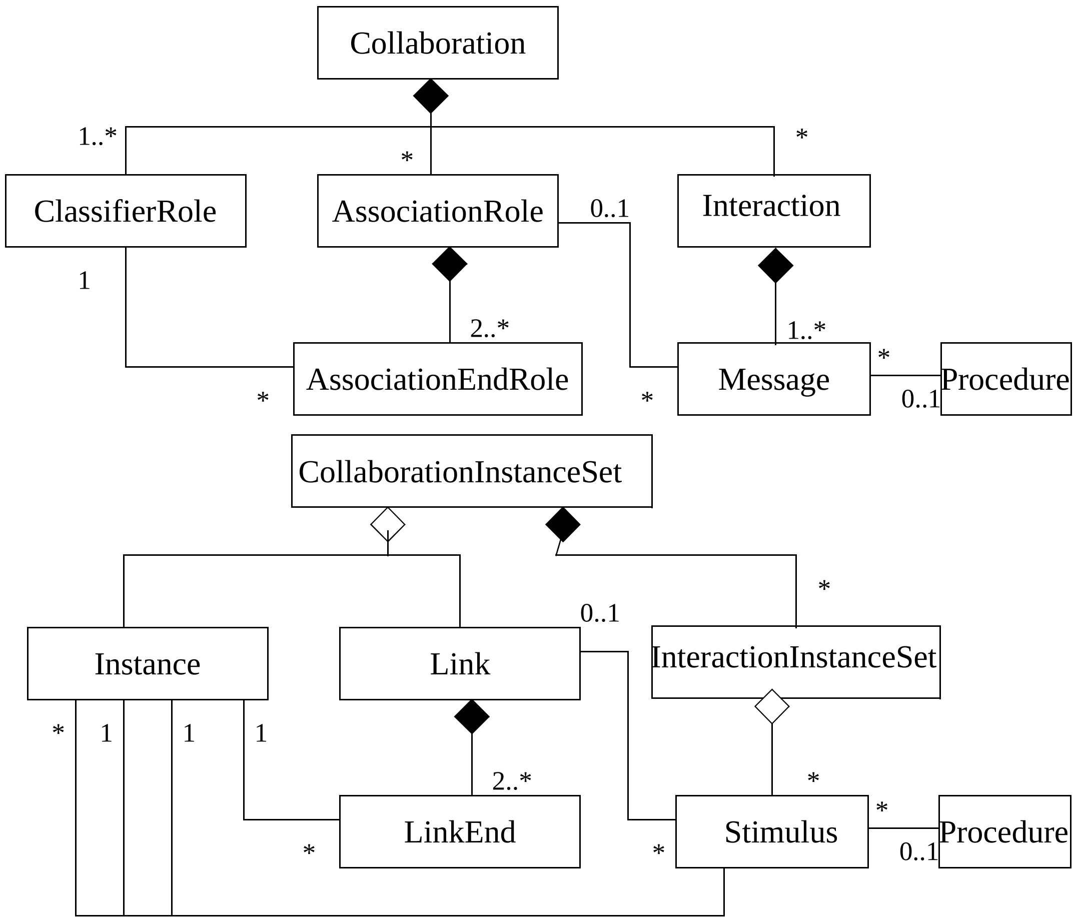


Figure 3-57 A summary of the UML constructs used in the section below.

3.60.5.1 *Sequence diagram*

A sequence diagram maps into an Interaction and an underlying Collaboration or an InteractionInstanceSet and an underlying CollaborationInstanceSet depending on whether the diagram shows Instances or ClassifierRoles. An Interaction specifies a sequence of communications; it contains a collection of partially ordered Messages, each specifying a communication between a sender role and a receiver role. A CollaborationInstanceSet references a collection of Instances that conform to the ClassifierRoles in the Collaboration owning the Interaction. These Instances communicate by dispatching Stimuli that conform to the Messages in the Interaction. The CollaborationInstanceSet has an InteractionInstanceSet that references these Stimuli. A sequence diagram presents either a collection of object symbols and arrows mapping to Instances and Stimuli, or a collection of classifier-role symbols and arrows mapping to ClassifierRoles and Messages. The Instances and Stimuli conform to the ClassifierRoles and Messages.

The sequence diagram presents either a Collaboration or a CollaborationInstanceSet. In the former case, the classifier box with its lifeline maps onto a ClassifierRole in the Collaboration, and the arrows map onto the Messages in one of the Collaboration's Interactions. The name strings in the boxes map onto the names of the ClassifierRoles, while the classifier names map onto the ClassifierRole's *base* Classifiers. The AssociationRoles among the ClassifierRoles are not shown on the sequence diagram. They must be obtained in the model from a complementary collaboration diagram or other means.

If the sequence diagram presents a CollaborationInstanceSet, each object box with its lifeline maps into an Instance, which conforms to a ClassifierRole in the CollaborationInstanceSet's Collaboration. The name field maps into the name of the Instance, the role name into the ClassifierRole's name, and the class field maps into the names of the Classifiers being the *base* Classifiers of the ClassifierRole. An arrow maps into a Stimulus connected to two Instances: the sender and the receiver. The Link used for the communication of the Stimulus plays the role specified by the AssociationRole connected to the Message. Unless the correct Link can be determined from a complementary collaboration diagram or other means, the Stimulus is either not attached to a Link (not a complete model), or it is attached to an arbitrary Link or to a dummy Link between the Instances conforming to the AssociationRole implied by the two ClassifierRoles due to the lack of complete information.

The label of the arrow is mapped into either the body attribute of the Procedure, or into a detailed action model. For the action model, the name of the Operation to be invoked or Signal to be sent is mapped onto the name of the Operation or Signal invoked by the actions in the Procedure connected to the Message. Different alternatives exist for showing the arguments of the Stimulus. If references to the actual Instances being passed as arguments are shown, these are mapped onto the arguments of the Stimulus. If the argument expressions are shown instead, and a detailed action model is used, then these are mapped into CodeActions in the Procedure, or additional actions that compute the values of the expressions. Finally, if the types of the arguments are shown together with the name of the Operation or the Signal, these are mapped onto the parameter types of the Operation or the Attribute types of the Signal, respectively. A timing label placed on the level of an arrow endpoint maps into the name of the corresponding Message or Stimulus. A constraint placed on the diagram maps into a Constraint on the entire Interaction.

An arrow with the arrowhead pointing to an object symbol or role symbol within the frame of the diagram maps into a Stimulus (Message) dispatched by a CreateObjectAction. The interpretation is that an Instance is created by dispatching the Stimulus. If the target of the arrow is a classifier-role symbol, the Instance will conform to the ClassifierRole. (Note, that the diagram does not necessarily show from which Classifier the Instance originates; only that the newly created Instance conform to the ClassifierRole.) After the creation of the Instance, it may immediately start interacting with other Instances. This implies that the creation method (constructor, initializer) of the Instance dispatches these Stimuli. If an object termination symbol

("X") is the target of an arrow, the arrow maps into a Stimulus that will cause the receiving Instance to be removed. If the object termination symbol appears in the diagram without an incoming arrow, it maps into a Procedure containing a DestroyObjectAction.

The order of the arrows in the diagram maps onto pairs of associations between the Stimuli (Messages). A *predecessor* relationship is established between Stimuli (Messages) corresponding to successive arrows in the vertical sequence. In case of concurrent arrows preceding an arrow, the corresponding Stimulus (Message) has a collection of predecessors. Moreover, each Stimulus (Message) has an *activator* association to the Stimulus (Message) corresponding to the incoming arrow of the activation.

Procedural sequence diagram

On a procedural sequence diagram (one with focus of control and calls), subsequent arrows on the same lifeline map into Stimuli (Messages) obeying the *predecessor* association. An arrow to the head of a focus of control region establishes a nested activation. The arrow maps into a Stimulus (Message) with the dispatching Procedure containing a CallOperationAction. The Stimulus holds the sender and receiver Instance, as well as the argument Instances, to be supplied in the invocation and references the target Operation to be invoked. The expressions that evaluate to the arguments of the Operation are, in a detailed action model, mapped into CodeActions in the Procedure connected to the Stimulus, or additional actions that compute the values of the expressions. In the case the arrow maps onto a Message the sender and the receiver are specified by the *sender* and *receiver* ClassifierRoles of the Message. The sender and receiver Instances of a Stimulus conform to these ClassifierRoles. Any condition or iteration expression attached to the arrow becomes, in a detailed action model, the test clause action in a ConditionalAction or LoopAction in the dispatching Procedure. All arrows departing the nested activation map into Stimuli (Messages) with an *activation* Association to the Stimulus (Message) corresponding to the arrow at the head of the activation. A return arrow departing the end of the activation maps into a Stimulus (Message) with:

- an *activation* Association to the Stimulus (Message) corresponding to the arrow at the head of the activation, and
- a *predecessor* association to the previous Stimulus (Message) within the same activation; that is, the last Stimulus (Message) being sent in the activation.

A return must be the final Stimulus (Message) within a predecessor chain. It is not the predecessor of any Stimulus (Message).

3.61 Object Lifeline

3.61.1 Semantics

In a sequence diagram an object lifeline denotes an Instance playing a specific role. Arrows between the lifelines denote communication between the Instances playing those roles. Within a sequence diagram the existence and duration of the Instance in a role is shown, but the relationships among the Instances are not shown. The role is specified by a ClassifierRole; it describes the properties of an Instance playing the role and describes the relationships an Instance in that role has to other Instances.

3.61.2 Notation

An Instance is shown as a vertical dashed line called the “lifeline.” The lifeline represents the existence of the Instance at a particular time. If the Instance is created or destroyed during the period of time shown on the diagram, then its lifeline starts or stops at the appropriate point; otherwise, it goes from the top to the bottom of the diagram. An object symbol is drawn at the head of the lifeline. If the Instance is created during the diagram, then the arrow, which maps onto the Stimulus that creates the Instance, is drawn with its arrowhead on the object symbol. If the Instance is destroyed during the diagram, then its destruction is marked by a large “X,” either at the arrow mapping to the Stimulus that causes the destruction or (in the case of self-destruction) at the final return arrow from the destroyed Instance. An Instance that exists when the transaction starts is shown at the top of the diagram (above the first arrow), while an Instance that exists when the transaction finishes has its lifeline continue beyond the final arrow.

The lifeline may split into two or more concurrent lifelines to show conditionality. Each separate track corresponds to a conditional branch in the communication. The lifelines may merge together at some subsequent point.

3.61.3 Presentation Options

In some cases, it is necessary to link sequence diagrams to each other; for example, it might not be possible to put all lifelines in one diagram, or a sub-sequence is included in several diagrams; hence, it is convenient to put the common sub-sequence in a separate diagram, which is referenced from the other diagrams. In these cases, the cut between the diagrams can be expressed in one of the diagrams with a dangling arrow leaving a lifeline but not arriving at another lifeline, and in the other diagram it is expressed with a dangling arrow arriving at a lifeline from nowhere. In both cases, it is recommended to attach a note stating which diagram the sequence originates from or continues in. This is purely notational. The different diagrams show different parts of the underlying Interaction.

3.61.4 Example

See also Figure 3-56 on page 3-105.

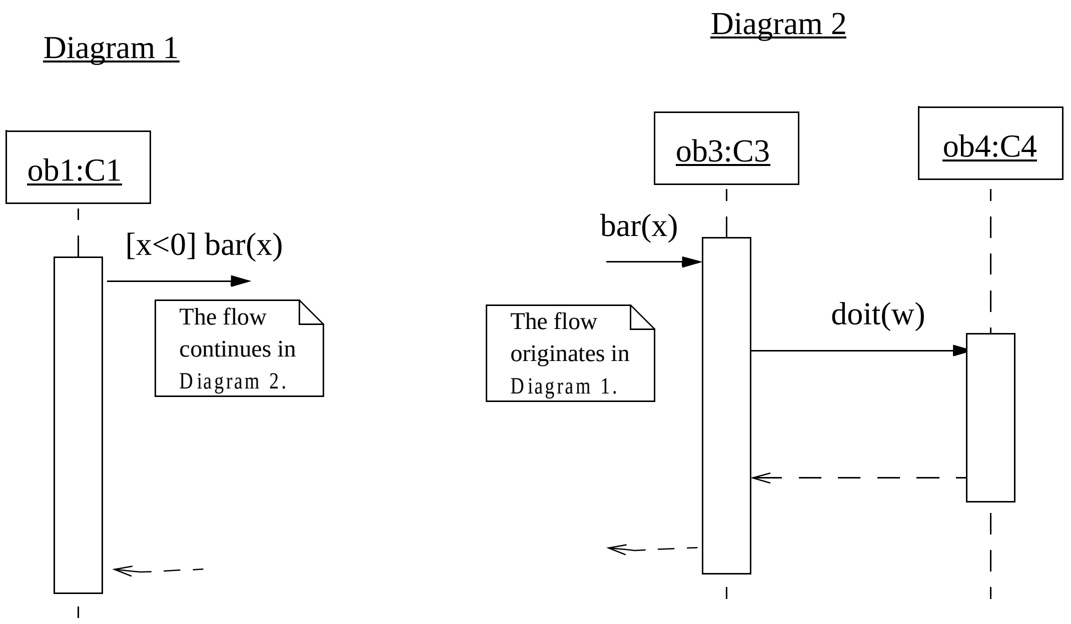


Diagram 1
ob1:C1
[x<0] bar(x)
The flow continues in Diagram 2.
Diagram 2
ob3:C3
ob4:C4
bar(x)
doit(w)
The flow originates in Diagram 1.

Figure 3-58 The flow shown in the sequence diagram to the left continues in the diagram to the right.

3.61.5 Mapping

See Section 3.60.5, “Mapping,” on page 3-106.

3.62 Activation

3.62.1 Semantics

An activation (focus of control) shows the period during which an Instance is performing a Procedure either directly or through a subordinate procedure. It represents both the duration of the performance of the Procedure in time and the control relationship between the activation and its callers (stack frame).

3.62.2 Notation

An activation is shown as a tall thin rectangle whose top is aligned with its initiation time and whose bottom is aligned with its completion time. The Procedure being performed may be labeled in text next to the activation symbol or in the left margin, depending on style. Alternately, the incoming arrow may indicate the Procedure, in which case it may be omitted on the activation itself. In procedural flow of control, the top of the activation symbol is at the tip of an incoming arrow (the one that initiates the procedure) and the base of the symbol is at the tail of a return arrow.

In the case of concurrent Instances each with their own threads of control, an activation shows the duration when each Instance is performing an Operation or transition in a state machine. Operations by other Instances are not relevant. If the distinction between direct computation and indirect computation (by a nested operation call) is unimportant, the entire lifeline may be shown as an activation.

3.62.3 Example

See Figure 3-55 on page 3-104 and Figure 3-56 on page 3-105.

3.62.4 Mapping

See Section 3.60.5, “Mapping,” on page 3-106.

3.63 Message and Stimulus

3.63.1 Semantics

A Stimulus is a communication between two Instances that conveys information with the expectation that action will ensue. A Stimulus will cause an Operation to be invoked, raise a Signal, or cause an Instance to be created or destroyed.

A Message is a specification of Stimulus, i.e. it specifies the roles that the sender and the receiver Instances must conform to, as well as the Procedure which will, when executed, dispatch a Stimulus that conforms to the Message.

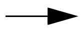
3.63.2 Notation

In a sequence diagram a Stimulus as well as a Message is shown as a horizontal solid arrow from the lifeline of one Instance or ClassifierRole to the lifeline of another Instance or ClassifierRole. In case of a Stimulus from an Instance to itself, the arrow may start and finish on the same lifeline. The arrow is labeled with the name of the Operation to be invoked or the name of the Signal. Its argument values or argument expressions may be presented, as well.

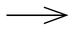
The arrow may also be labeled with a sequence number to show the sequence of the Stimulus (Message) in the overall interaction. However, sequence numbers are often omitted in sequence diagrams, as the physical location of the arrow shows the relative sequences, but they are necessary in collaboration diagrams. Sequence numbers are useful on both kinds of diagrams for identifying concurrent threads of control. An arrow may also be labeled with a condition and/or iteration expression.

3.63.3 Presentation options

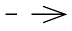
The following arrowhead variations may be used to show different kinds of communications.

filled solid arrowhead 

Operation call or other nested flow of control. The entire nested sequence is completed before the outer level sequence resumes. The arrowhead may be used to denote ordinary operation calls, but it may also be used to denote concurrently active instances when one of them sends a Signal and waits for a nested sequence of behavior to complete before it continues.

stick arrowhead 

Asynchronous communication; that is, no nesting of control. The sender dispatches the Stimulus and immediately continues with the next step in the execution.[1]

dashed arrow with stick arrowhead 

Return from operation call.

Variation:

In a procedural flow of control, the return arrow may be omitted (it is implicit at the end of an activation). It is assumed that every call has a paired return after any subordinate stimuli. The return value can be shown on the initial arrow. For nonprocedural flow of control (including parallel processing and asynchronous messages) returns should be shown explicitly.

Variation:

Normally message arrows are drawn horizontally. This indicates the duration required to send the stimulus is “atomic;” that is, it is brief compared to the granularity of the interaction and that nothing else can “happen” during the transmission of the stimulus. This is the correct assumption within many computers. If the stimulus requires some time to arrive, during which something else can occur (such as a stimulus in the opposite direction), then the arrow may be slanted downward so that the arrowhead is below the arrow tail.

Variation: Branching

A branch is shown by multiple arrows leaving a single point, each possibly labeled by a condition. Depending on whether the conditions are mutually exclusive, the construct may represent conditionality or concurrency.

1.UML 1.3 and previous versions included a half-stick arrowhead notation in addition to the stick arrowhead notation. This notation has been removed because the semantic distinction between the two was subtle and confusing.

Variation: Iteration

A connected set of arrows may be enclosed and marked as an iteration. For a generic sequence diagram, the iteration indicates that the dispatch of a set of stimuli can occur multiple times. For a procedure, the continuation condition for the iteration may be specified at the bottom of the iteration. If there is concurrency, then some arrows in the diagram may be part of the iteration and others may be single execution. It is desirable to arrange a diagram so that the arrows in the iteration can be enclosed together easily.

Variation:

A lifeline may subsume an entire set of objects on a diagram representing a high-level view.

Variation:

A distinction may be made between a period during which an Instance has a live activation and a period in which the activation is actually computing. The former (during which it has control information on a stack but during which control resides in something that it called) is shown with the ordinary double line. The latter (during which it is the top item on the stack) may be distinguished by shading the region.

3.63.4 Example

See Figure 3-56 on page 3-105.

3.63.5 Mapping

See Section 3.60.5, “Mapping,” on page 3-106.

3.64 Transition Times

3.64.1 Semantics

A Message may specify several different times; for example, a sending time and a receiving time. These are formal names that may be used within Constraint expressions. The set of different kinds of times is open-ended so that users can invent new ones as needed for special situations, such as *elapsedTime* and *startExecutionTime*. These expressions may be used in Constraints to designate specific time constraints valid for the Message.

3.64.2 Notation

A transition instance (such as a Stimulus or Message in a sequence diagram, a collaboration diagram, or a Transition in a state machine) may be given a name. A timing constraint is formed as an expression based on the name of the transition. For example, if the name of a Stimulus is *stim*, its send-time is expressed by *stim.sendTime ()*, and its receive-time by *stim.receiveTime ()*. The timing constraint may be shown in the left margin aligned with the arrow (on a sequence diagram) or near the tail of the arrow (on a collaboration diagram). Constraints may be specified by placing Boolean expressions, possibly including time expressions, in braces on the sequence diagram.

3.64.3 Presentation Options

When it is clear from the context, the name of a Message or the name of a Stimulus may itself be used to denote the time at which the transition started. In cases where the performance of the transition is not instantaneous, the time at which the transition is ended may be indicated by the same name with a prime sign appended to the name.

3.64.4 Example

See Figure 3-55 on page 3-104.

3.64.5 Mapping

See Section 3.60.5, “Mapping,” on page 3-106.

Part 8 - Collaboration Diagrams

3.65 Collaboration Diagram

3.65.1 Semantics

A collaboration diagram presents either a Collaboration, which contains a set of roles to be played by Instances, as well as their required relationships given in a particular context, or it presents a CollaborationInstanceSet with a collection of Instances and their relationships. The diagram may also present an Interaction (InteractionInstanceSet), which defines a set of Messages (Stimuli) specifying the interaction between the Instances playing the roles within a Collaboration to achieve the desired result.

A Collaboration is used for describing the realization of an Operation or a Classifier. A Collaboration that describes a Classifier, like a UseCase, references Classifiers and Associations in general, while a Collaboration describing an Operation includes the arguments and local variables of the Operation, as well as ordinary Associations attached to the Classifier owning the Operation.

3.65.2 Notation

A collaboration diagram shows a graph of either Instances linked to each other, or ClassifierRoles and AssociationRoles; it may also include the communication stated by an Interaction or InteractionInstanceSet.

Because collaboration diagrams often are used to help design procedures, they typically show navigability using arrowheads on the lines representing Links or AssociationRoles. (An arrowhead on a line between boxes indicates a Link or AssociationRole with one-way navigability. An arrow next to a line indicates Stimuli or Message flowing in the given direction. Obviously such an arrow cannot point backwards over a one-way line.)

The order of the interaction is described with a sequence of numbers, usually starting with number *1*. For a procedural flow of control, the subsequent communication numbers are nested in accordance with call nesting. For a nonprocedural sequence of interactions among concurrent instances, all the sequence numbers are at the same level (that is, they are not nested).

A collaboration diagram without any interaction shows the *context* in which interactions can occur. It might be used to show the context for a single Operation or even for all of the Operations of a Class or group of Classes.

A collection of standard constraints may be used to show whether an Instance or a Link is created or destroyed during the execution:

- Instances and Links created during the execution may be designated as {new}.
- Instances and Links destroyed during the execution may be designated as {destroyed}.
- Instances and Links created during the execution and then destroyed may be designated as {transient}.

These changes in life state are derivable from the detailed interaction among the Instances, they are provided as notational conveniences.

3.65.2.1 Collaboration Instance

A collaboration diagram given at instance level shows a CollaborationInstanceSet; that is, a collection of object boxes and lines mapping to Instances and Links, respectively. These instances conform to the ClassifierRoles and AssociationRoles of the CollaborationInstanceSet's Collaboration. The diagram may also include arrows attached to the lines that correspond to Stimuli communicated over the Links. The diagram shows the Instances relevant to the realization of an Operation or Classifier, including Instances indirectly affected or accessed during the performance. The diagram also shows the Links among the Instances, including transient ones representing procedure arguments, local variables, and *self* links. Individual attribute values are usually not shown explicitly. If Stimuli must be sent to attribute values, the Attributes should be modeled using Associations instead.

3.65.2.2 Collaboration

A collaboration diagram given at specification level shows a Collaboration; that is, the roles defined within a Collaboration. Together, these roles form a realization of the attached Operation or Classifier of the Collaboration. The diagram contains a collection of class boxes and lines corresponding to ClassifierRoles and AssociationRoles in the Collaboration. In this case the arrows attached to the lines map onto Messages.

3.65.3 Example

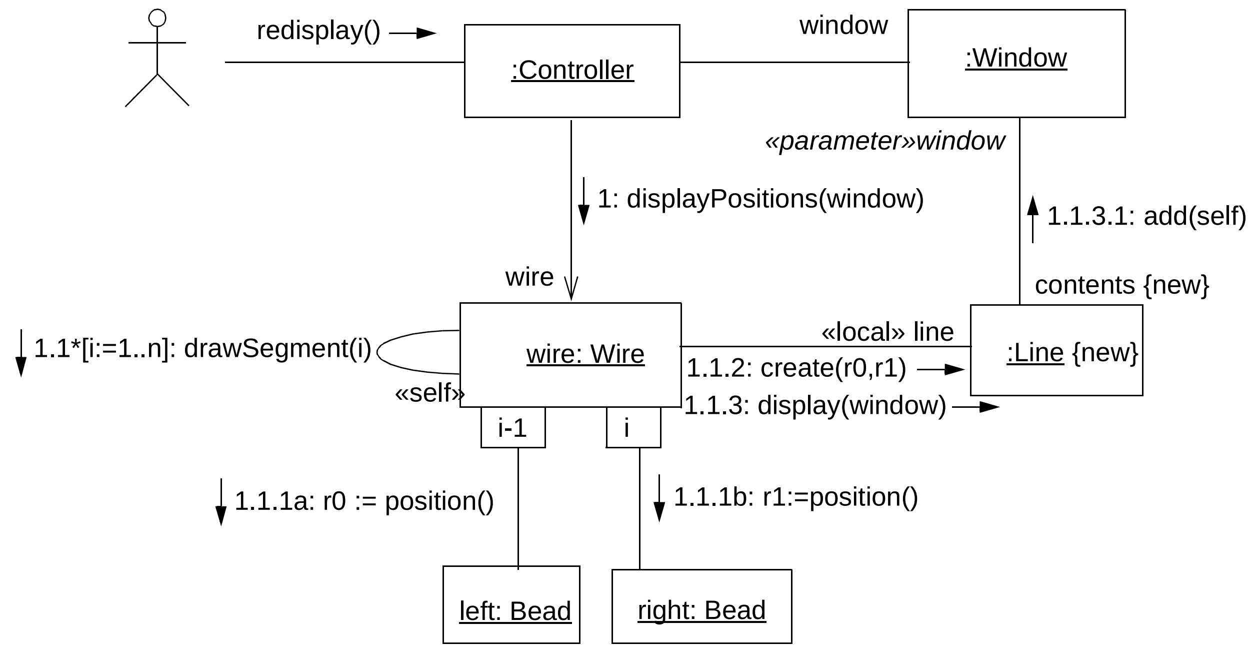


Figure 3-59 Collaboration Diagram at instance level, presenting Objects, Links, and Stimuli referenced by a CollaborationInstanceSet and its InteractionInstanceSet.

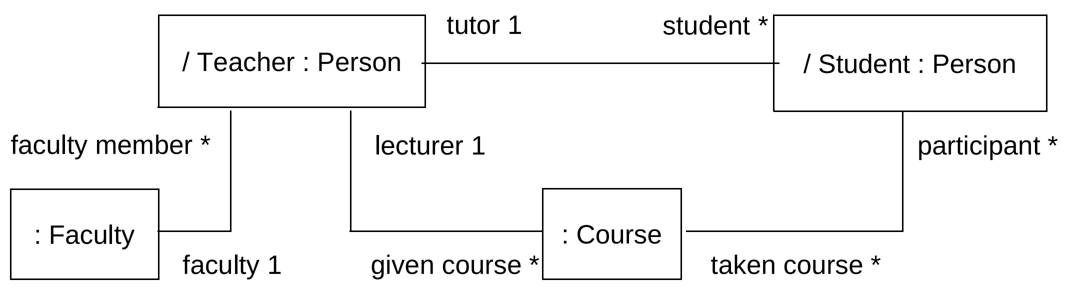


Figure 3-60 Collaboration Diagram at specification level, presenting the ClassifierRoles and the AssociationRoles that belong to the Collaboration.

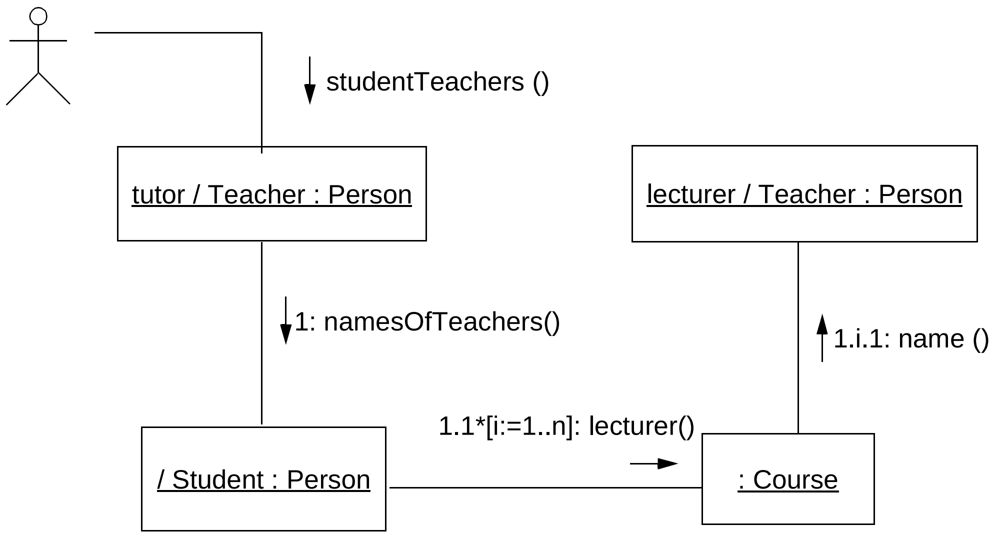


Figure 3-61 Collaboration Diagram presenting a CollaborationInstanceSet in which some of the Objects play the same role. The instances conform to the Collaboration shown in Figure 3-60 on page 3-116.

3.65.4 *Mapping*

A collaboration diagram maps either to a Collaboration, possibly together with an Interaction, or to a CollaborationInstanceSet possibly together with its InteractionInstanceSet. The mapping of each kind of icon is described in Section 3.69, “Collaboration Roles,” on page 3-124. The mapping of the stereotypes is explained in Section 3.49, “Link,” on page 3-84.

3.66 *Pattern Structure*

3.66.1 *Semantics*

A Collaboration can be used to specify the implementation of design constructs. For this purpose, it is necessary to specify its context and interactions. It is also possible to view a Collaboration as a single entity from the “outside.” For example, this could be used to identify the presence of design patterns within a system design. A pattern is a parameterized Collaboration; that is, a Collaboration template. In each use of the pattern, actual Classifiers are substituted for the parameters in the pattern definition.

Note that *patterns* as defined in *Design Patterns* by Gamma, Helm, Johnson, and Vlissides include much more than structural descriptions. UML describes the structural aspects and some behavioral aspects of design patterns; however, UML notation does not include other important aspects of patterns, such as usage trade-offs or examples. These must be expressed by other means, such as in text or tables.

A Collaboration can be defined in terms of other, so-called subordinate, Collaborations. Each role in the former Collaboration, the so-called superordinate Collaboration, is either a new role that is defined in the superordinate Collaboration or it is a role defined in one or several of the subordinate Collaborations and reused in the definition of the superordinate Collaboration. In the latter case, the role is often renamed so it better suits the purpose of the superordinate Collaboration. If so, the original names of the roles are shown within curly brackets after the name used within the superordinate Collaboration (see Figure 3-66 on page 3-120).

3.66.2 Notation

A use of a Collaboration is shown as a dashed ellipse containing the name of the Collaboration. A dashed line is drawn from the collaboration symbol to each of the symbols denoting Classifiers that participate in the Collaboration. Each line is labeled by the *role* of the participant. The roles correspond to the names of elements within the context for the Collaboration; such names in the Collaboration are treated as parameters that are bound to specify elements on each occurrence of the pattern within a model. Therefore, a collaboration symbol can show the use of a design pattern together with the actual Classifiers and Associations that occur in that particular use of the pattern.

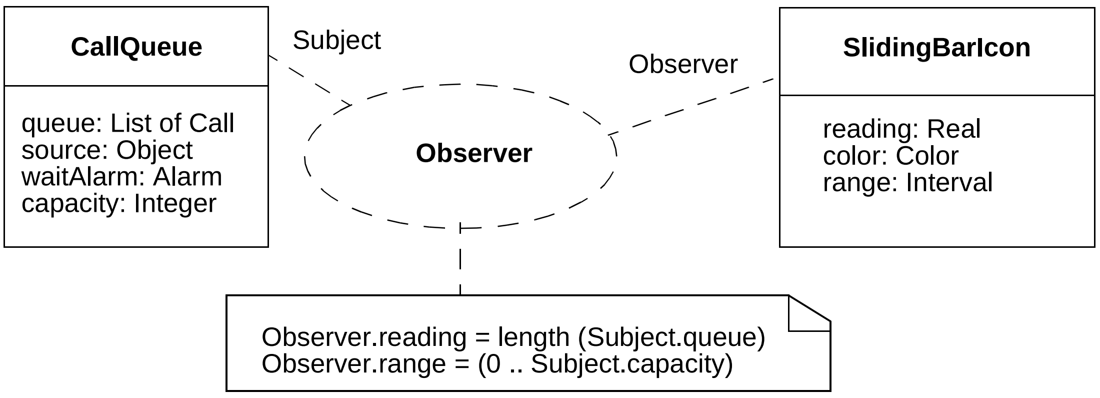


Figure 3-62 Use of a Collaboration.

As a Collaboration is a GeneralizableElement, it may have Generalization relationships to other Collaborations. In this way it is possible to define one Collaboration to be a specialization of another Collaboration. It is depicted by the ordinary Generalization arrow from the dashed ellipse representing the child Collaboration to the icon of the parent Collaboration. The roles of the child Collaborations may be specializations of roles in the parent Collaboration. This is shown by redefining the role name of the parent collaboration in the child collaboration.

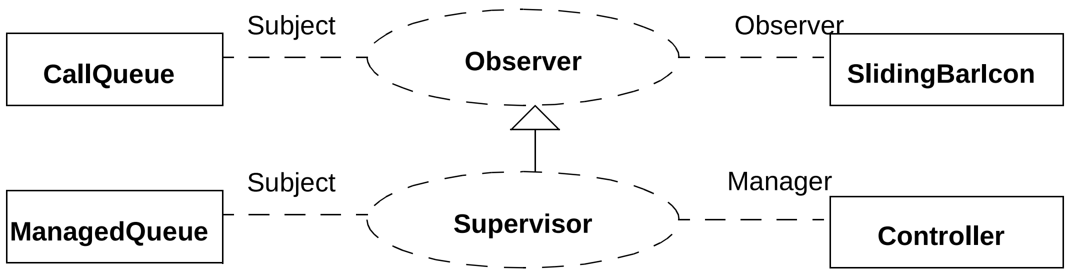


Figure 3-63 Specialization of a Collaboration. As the Subject role of the Supervisor collaboration is a specialization (an extension) of the Subject role defined in the Observer collaboration, the ManagedQueue class is used instead of the CallQueue class as the base of the Subject role.

A dashed arrow with a stick arrowhead is used to show that a Collaboration is a realization of an Operation or a Classifier. This relationship can also be presented in textual form within the Collaboration symbol.

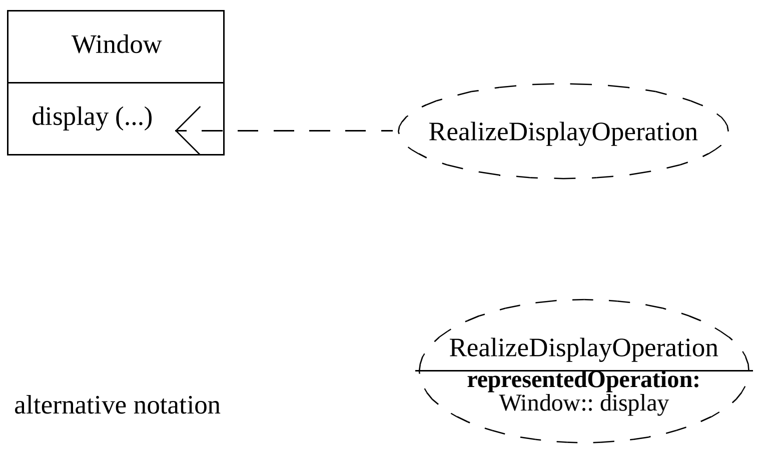


Figure 3-64 The relationship between a Collaboration and the element it is realizing can be shown either as a dashed arrow with a stick arrowhead from the Collaboration to the realized element, or in text.

The usual convention is used to show a CollaborationInstanceSet; that is, it is shown as a dashed ellipse with the name underlined. The Instances and the Links that participate in the CollaborationInstanceSet are connected to the ellipse with dashed lines. The name of the role an instance is playing is shown close to the line and the instance.

In some cases it is convenient to show the static structure of a Collaboration within the collaboration icon (the dashed ellipse).

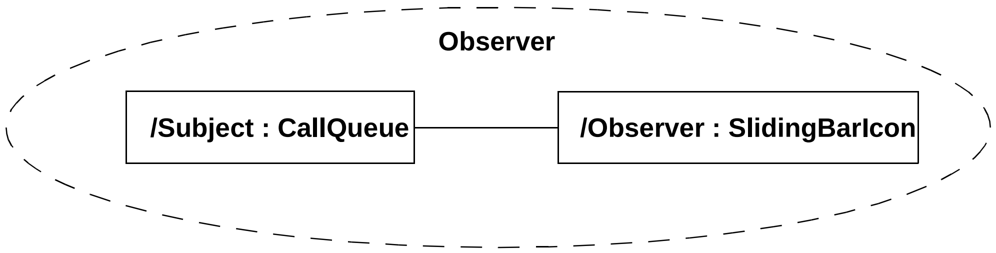


Figure 3-65 The static structure of a Collaboration shown within the collaboration icon.

It is possible to denote that a Collaboration is defined in terms of other Collaborations in two different ways, either using dashed ellipses showing the Collaborations and their relationships, or using ordinary collaboration diagrams. The former way has the advantage that it explicitly shows the relationship between the Collaborations, while the latter shows the structure of the new Collaboration.

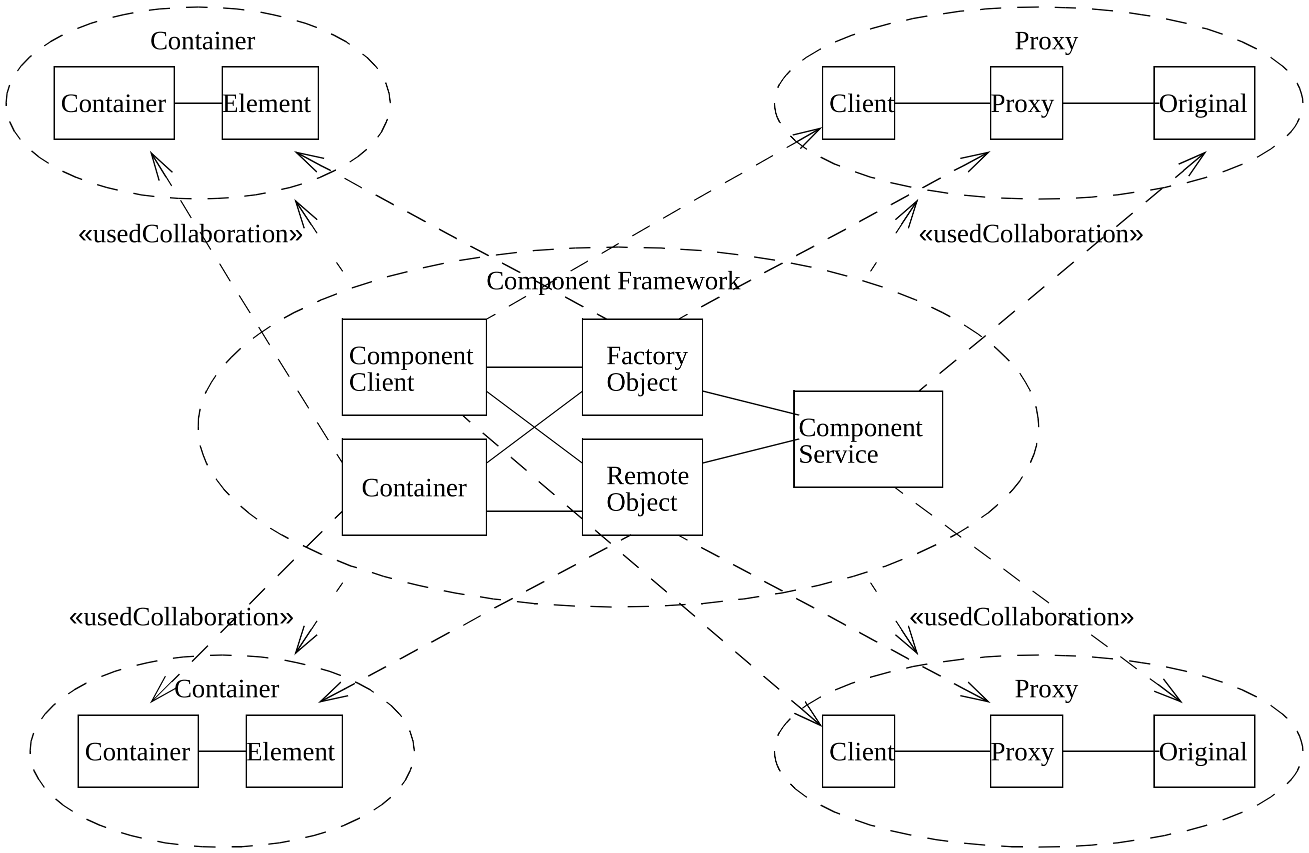


Figure 3-66 The ComponentFramework Collaboration uses two occurrences of the Proxy Collaboration and two occurrences of the Container Collaboration. Note that each role in the Component Framework corresponds to a role in two of the used Collaborations.

3.66.3 Mapping

A collaboration usage symbol maps into a Collaboration. For each class symbol and lines attached by a dashed line to the pattern occurrence symbol, the corresponding Classifier or Association is bound to the template parameter that is the *base* association target of the ClassifierRole or AssociationRole in the Collaboration template with the name equal to the name on the dashed line.

A dashed arrow with a closed hollow arrowhead from a Collaboration symbol to a Classifier or to an Operation is mapped onto the *representedClassifier* and onto the *representedOperation* association of the Collaboration, respectively.

A collaboration usage symbol with its name underlined is mapped onto a CollaborationInstanceSet. The object box symbols and the lines attached to the ellipse by dashed lines are mapped onto Instances and Links, respectively.

3.67 Collaboration Contents

The contents of a Collaboration is a collection of roles specifying how Instances and Links cooperate within a given context for a particular purpose, such as performing an Operation or a Use case. A Collaboration is a fragment of a larger complete model that is intended for a particular purpose.

3.67.1 Semantics

A *Collaboration diagram* shows either a Collaboration or a CollaborationInstanceSet. In the former case, the diagram shows one or more roles together with their contents, relationships, and neighbor roles, plus additional relationships and Classes as needed. When the diagram shows a CollaborationInstanceSet, it shows instances participating in the CollaborationInstanceSet, playing the roles defined in the Collaboration. To use a Collaboration, each role must be bound to an actual Classifier (or collection of Classifiers, if multiple classification is used) that (jointly) support the Features required by the role. The additional elements express additional requirements that cannot be modeled with roles, such as Generalizations between roles.

3.67.2 Notation

A collaboration diagram presents a graph of class boxes or object boxes together with connecting lines. These icons map onto ClassifierRoles and AssociationRoles, or Instance, and Links, respectively (see Section 3.69, “Collaboration Roles,” on page 3-124).

However, a collaboration diagram may also contain other elements, like different kinds of Classifiers, Generalizations, and Constraints, to express additional information. These elements are shown using their ordinary icons.

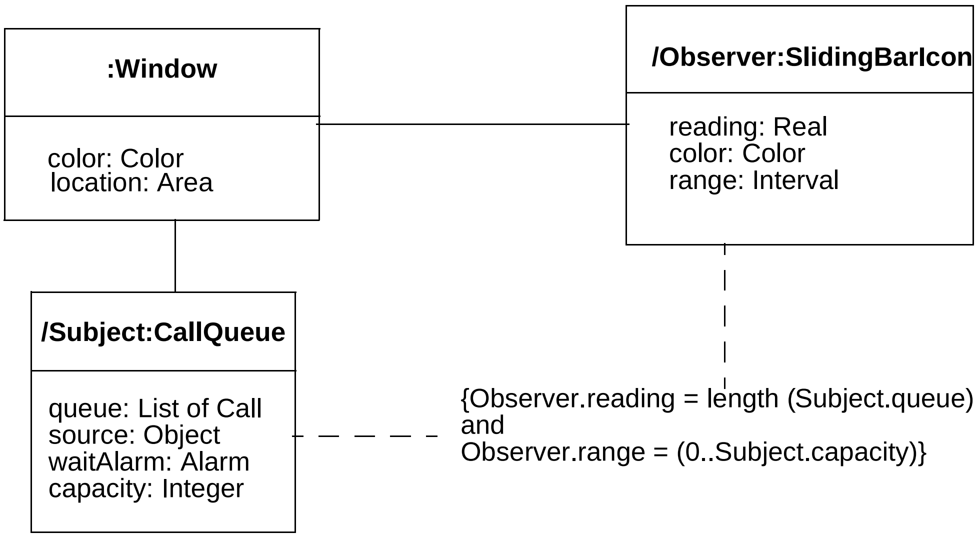


Figure 3-67 A collaboration diagram showing a Collaboration with a Constraint as a constraining element of the Collaboration.

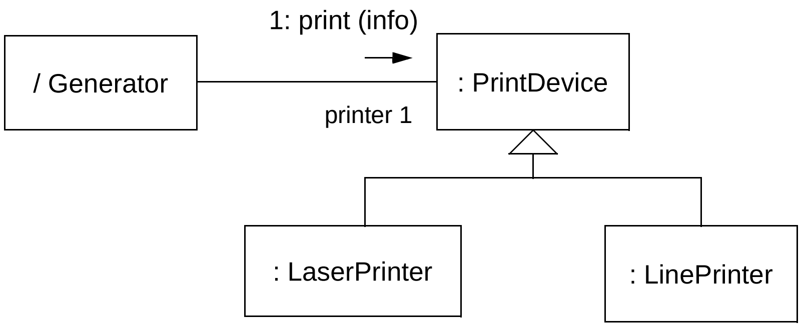


Figure 3-68 A collaboration diagram showing different roles, together with two additional Generalization relationships as constraining elements.

3.67.3 *Mapping*

The mapping of roles and instances are described in Section 3.69, “Collaboration Roles,” on page 3-124. Any constraining element, like a generalization arrow, is mapped onto its usual model element, such as Generalization. These elements are referenced by the Collaboration as its *constraining elements*.

3.68 Interactions

A collaboration of Instances interacts to accomplish a purpose (such as performing an Operation) by exchanging Stimuli. These may include both sending Signals and invocations of Operations, as well as more implicit interaction through conditions and time events. A specific pattern of communication exchanges to accomplish a specific purpose is called an *Interaction*. The collection of Stimuli sent between the Instances that participate in a CollaborationInstanceSet when they perform the task of the Collaboration is called an *InteractionInstanceSet*.

3.68.1 Semantics

An *Interaction* is a behavioral *specification* that comprises a sequence of communications exchanged among a set of Instances within a Collaboration to accomplish a specific purpose, such as the implementation of an Operation. To specify an Interaction, it is first necessary to specify a Collaboration; that is, to establish the roles that interact and their relationships. Then, the possible interaction sequences are specified. These can be specified in a single description containing conditionals (branches or conditional signals), or they can be specified by supplying multiple descriptions, each describing a particular path through the possible execution paths.

One communication is specified with a Message; it specifies the sender and the receiver roles, as well as the Procedure that will cause the communication to take place. The Procedure specifies what kind of communication that should take place, such as sending a Signal or invoking an Operation, and determines the actual arguments to be supplied. The Procedure may also state conditions or iterations of the communication.

When the Procedure is performed, a Stimulus is dispatched conforming to the Message. The Stimulus contains references to the sender and the receiver Instances playing the sender role and the receiver role of the Message, as well as a sequence of references to Instances being the actual arguments determined by the Procedure. An InteractionInstanceSet is a collection of Stimuli that conform to the Messages of an Interaction, i.e. the Stimuli are sent between the Instances participating an a CollaborationInstanceSet when they perform the task defined by the Collaboration.

3.68.2 Notation

Interactions are shown as sequence diagrams or as collaboration diagrams. Both diagram formats show the execution of collaborations. However, sequence diagrams do not show the relationships between the Instances or the Attribute values of the Instances; therefore, they do not fully show the context aspect of a Collaboration. Sequence diagrams do show the behavioral aspect of Collaborations explicitly, including the time sequence of Stimuli and explicit representation of method activations. Sequence diagrams are described in “Part 7 - Interaction Diagrams” on page 3-100. Collaboration diagrams show the full context of an interaction, including the Instances and their relationships relevant to a particular interaction. The sequencing of the Stimuli is done using sequence numbers, since distributing them along a time axis, like in Sequence diagrams, is not possible in this kind of diagram. (In fact, in some cases it is convenient to use sequence numbers in combination with a time axis.) The contents of collaboration diagrams are described in the following section.

3.68.3 Mapping

The mapping of roles and instances are described below, while the mapping of messages and stimuli are described in Section 3.72, “Message and Stimulus,” on page 3-130.

3.68.4 Example

See Section 3.65, “Collaboration Diagram,” on page 3-114 for examples of Interactions and InteractionInstanceSets and their Collaborations and CollaborationInstanceSets, respectively.

3.69 Collaboration Roles

3.69.1 Semantics

A ClassifierRole defines a role to be played by an Instance within a Collaboration. The role describes the kind of Instance that may play the role, such as required Operations and Attributes, and describes its relationships to Instances playing other roles. The relationships to other roles are defined by AssociationRoles. These describe the required Links between the Instances; that is, a subset of the existing Links.

3.69.2 Notation

A ClassifierRole is shown using a class rectangle symbol. Normally, only the name compartment is shown, but the attribute and operation compartments may also be shown when needed. The name compartment contains the string:

‘/’ ClassifierRoleName ‘:’ ClassifierName [‘,’ ClassifierName]*

The name of the Classifier (or Classifiers if multiple classification is used) can include a full pathname of enclosing Packages, if necessary. A tool will normally permit shortened pathnames to be used when they are unambiguous. The Package names precede the Classifier name and are separated by double colons. For example:

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

A stereotype may be shown textually (in guillemets above the name string) or as an icon in the upper right corner. A ClassifierRole representing a set of Instances can include a multiplicity indicator (such as “*”) in the upper right corner of the class box.

An AssociationRole is shown with the usual association line. The name string of the AssociationRole follows the same syntax as for the ClassifierRole. If the name is omitted, a line connected to ClassifierRole symbols denotes an AssociationRole. The information attached to the ends of the AssociationRole; that is, to the AssociationEndRoles, are shown using the same notation as for AssociationEnds.

An Instance playing the role defined by a ClassifierRole is depicted by an object box, normally without an attribute compartment. The name of the Instance is shown as a string:

ObjectName ‘/’ ClassifierRoleName ‘:’ ClassifierName [‘,’ ClassifierName]*

That is it starts with the name of the Instance, followed by the complete name of the ClassifierRole, all underlined. If the attribute compartment is shown, it contains the names of the Attributes required by an Instance playing the role. If some Attributes are required to have certain values, this is shown in the same way as in object diagrams; that is, the name of the attribute followed by an equal sign and the relevant values.

A Link is shown by a line between object boxes. Its name string follows the syntax of an Object playing a specific role.

3.69.3 Presentation options

The name of a ClassifierRole may be omitted. In this case, the colon is kept together with the Classifier name. The role name may be omitted only if there is only *one* role to be played by Instances of the base Classifier in the Collaboration.

The name of the Classifier may be omitted together with the colon.

At least one of the Classifier name (together with the colon) or the ClassifierRole name (together with the slash) must be present to denote a ClassifierRole. Otherwise, the rectangle denotes an ordinary Classifier or Instance depending on whether the name is underlined or not.

If the role is to be played by an Instance originating from multiple Classifiers, the names of the Classifiers are shown in a comma separated list after the colon.

In an object box the Instance name, the role name and / or the classifier name may be omitted. However, the colon should be kept in front of the classifier name, and the slash should be kept in front of the role name. The notation used is the same for Instances in general, with the possible addition of the name of the ClassifierRole that the Instance conforms to.

Note, the name of an Instance is always underlined, whereas the name of a Classifier (including ClassifierRole) is never underlined. Furthermore, an un-named line between icons representing Instances is always a Link, and between icons representing Classifiers (except ClassifierRoles) it is always an Association.

These tables summarize the different combinations of names:

syntax	explanation
: C	un-named Instance originating from the Classifier C
/ R	un-named Instance playing the role R
/ R : C	un-named Instance originating from the Classifier C playing the role R
O / R	an Instance named O playing the role R
O : C	an Instance named O originating from the Classifier C
O / R : C	an Instance named O originating from the Classifier C playing the role R
O	an Instance named O
/ R	a role named R
: C	an un-named role with the *base* Classifier C
/ R : C	a role named R with the *base* Classifier C

3.69.4 Example

See figures in Section 3.65, “Collaboration Diagram,” on page 3-114.

3.69.5 Mapping

A classifier role rectangle maps onto one ClassifierRole. The role name is the name of the ClassifierRole and the sequence of classifier names are the names of the *base* Classifiers. An association role line maps onto an AssociationRole attached to the ClassifierRoles corresponding to the rectangles at the end points of the line.

An object symbol maps onto an Instance whose name is the *object* part of the name string. The Classifiers of the Instance are those named according to the sequence of names in the *class* part of the string (or children of these Classifiers). The Instance conforms to the ClassifierRole, whose name is the *role* part of the string.

A Collaboration can also be used for describing the internal structure of a Classifier. In such case, the names of the roles are the same as the names of the attributes of the Classifier. In this way, the connection between the roles and the Attributes they represent are established. (The base of the roles are not enough for uniquely identifying this mapping, since several Attributes may have the same type.)

3.70 Multiobject

3.70.1 Semantics

A multiobject represents a set of Instances on the “many” end of an Association. This is used to show Operations and Signals that address the entire set, rather than a single Instance in it. The underlying static model is unaffected by this grouping. This corresponds to an Association with multiplicity “many” used to access a set of associated Instances.

3.70.2 Notation

A multiobject is shown as two rectangles in which the top rectangle is shifted slightly vertically and horizontally to suggest a stack of rectangles. A message arrow to the multiobject symbol indicates a Stimulus to the set of Instances (for example, a selection Operation to find an individual Object).

To perform an Operation on each Instance in a set of associated Instances requires two Stimuli: an iteration to the multiobject to extract Links to the individual Instances and then a Stimulus sent to each individual Instance using the (temporary) Link. This may be elided on a diagram by combining the arrows into a single arrow that includes an iteration and an application to each individual Instance. The target rolename takes a “many” indicator (*) to show that many individual Links are implied. Although this may be written as a single Stimulus, in the underlying model (and in any actual code) it requires the two layers of structure (iteration to find Links, communication using each Link) mentioned previously.

An Instance from the set is shown as a normal object symbol, but it may be attached to the multiobject symbol using a composition Link to indicate that it is part of the set. A communication arrow to the simple object symbol indicates a Stimulus to an individual Instance.

Typically a selection Stimulus to a multiobject returns a reference to an individual Instance, to which the original sender then sends a Stimulus.

3.70.3 Example

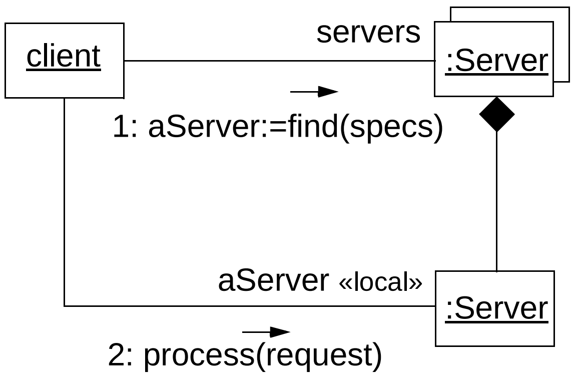


Figure 3-69 Multiobject

3.70.4 Mapping

A multiobject symbol maps to a collection of Instances in which each Instance conforms to the ClassifierRole and this role has the multiplicity “many” (or whatever is explicitly specified). In other respects, it maps the same as an object symbol. (The stereotype is explained in Section 3.49, “Link,” on page 3-84.)

3.71 Active object

An *active object* is one that owns a thread of control and may initiate control activity. A passive object is one that holds data, but does not initiate control. However, a passive object may send Stimuli in the process of processing a request that it has received. In a collaboration diagram, a ClassifierRole that is an active class represents the active objects that occur during execution.

3.71.1 Semantics

An active object is an Instance that owns a thread of control. Processes and tasks are traditional kinds of active objects.

3.71.2 Notation

A role for an active object is shown as a rectangle with a heavy border. Frequently, active object roles are shown as composites with embedded parts.

The property keyword *{active}* may also be used to indicate an active object.

3.71.3 Example

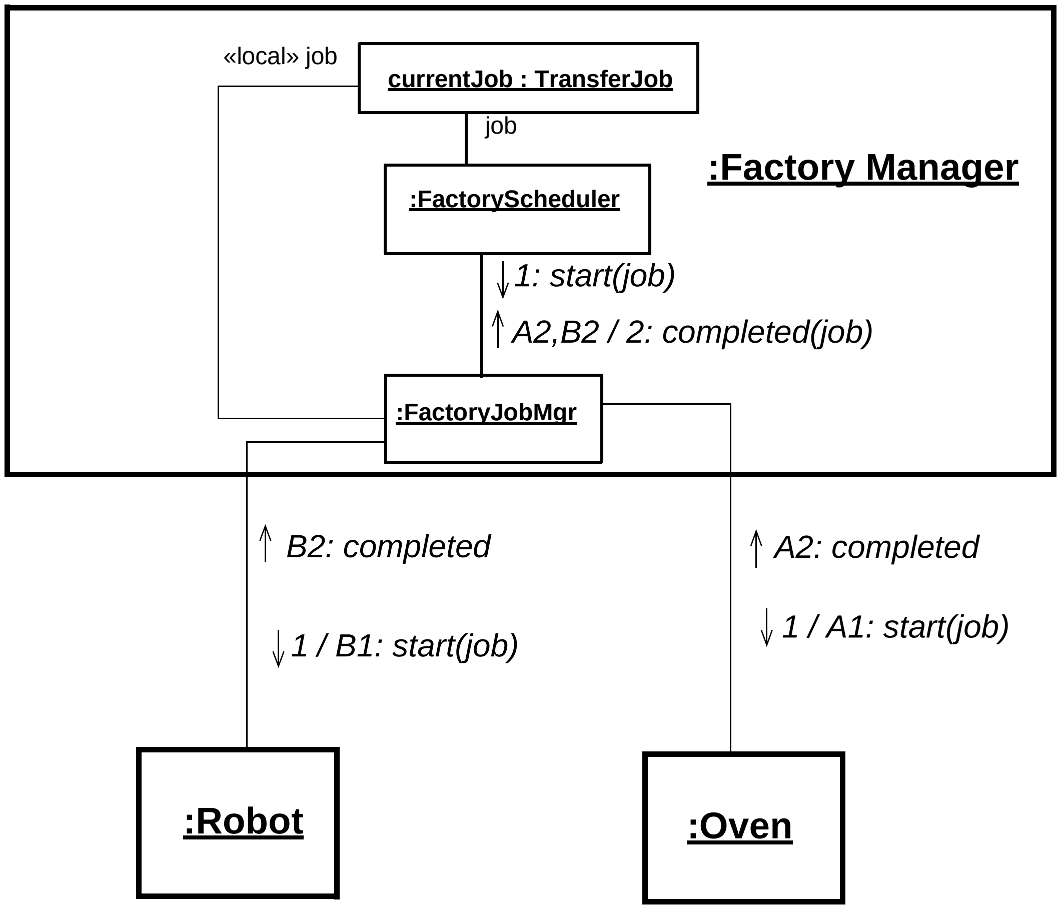


Figure 3-70 Composite Active Object

3.71.4 Mapping

An active object symbol maps as an object symbol does, with the addition that the class of the object has the *active* property set.

3.72 Message and Stimulus

3.72.1 Semantics

In a collaboration diagram a Stimulus is a communication between two Instances that conveys information with the expectation that action will ensue. A Stimulus will cause an Operation to be invoked, raise a Signal, or an Instance to be created or destroyed.

A Message is a specification of Stimulus; i.e., it specifies the roles that the sender and the receiver Instances should conform to, as well as the Procedure which will, when executed, dispatch a Stimulus that conforms to the Message.

3.72.2 Notation

Messages and Stimuli are shown as labeled arrows placed near an AssociationRole or a Link, respectively. The meaning is that the Link is used for transportation of the Stimulus to the target Instance. The arrow points along the line in the direction of the receiving Instance.

3.72.2.1 Control flow type

The following arrowhead variations may be used to show different kinds of communications.

filled solid arrowhead ──►

Operation call or other nested flow of control. The entire nested sequence is completed before the outer level sequence resumes. The arrowhead may be used to denote ordinary operation calls, but it may also be used to denote concurrently active instances when one of them sends a Signal and waits for a nested sequence of behavior to complete before it continues.

stick arrowhead $\longrightarrow$

Asynchronous communication; that is, no nesting of control. The sender dispatches the Stimulus and immediately continues with the next step in the execution.

dashed arrow with stick arrowhead - $\rightarrow$

Return from an operation call. The return arrow may be suppressed as it is implicit at the end of an activation.

other variations

Other kinds of control may be shown, such as “balking” or “time-out;” however, these are treated as extensions to the UML core.

A half stick arrowhead can be used to show asynchronous communication. This alternative is included for backwards compatibility. UML 1.3 and previous versions, included both half stick arrowhead and stick arrowhead with a very small (and not well-understood) distinction.

3.72.2.2 Arrow label

In the following the term *Message* is used, but the text applies to *Stimulus*, as well.

The label has the following syntax:

predecessor sequence-expression return-value := *message-name argument-list*

The label indicates the Message being sent, its arguments and return values, and the sequencing of the Message within the larger interaction, including call nesting, iteration, branching, concurrency, and synchronization.

3.72.2.3 Predecessor

The predecessor is a comma-separated list of sequence numbers followed by a slash (‘/’):

sequence-number ‘,’ . . . ‘/’

The clause is omitted if the list is empty.

Each sequence-number is a sequence-expression without any recurrence terms. It must match the sequence number of another Message.

The meaning is that the Message is not enabled until all of the communications whose sequence numbers appear in the list have occurred. Therefore, the list of predecessors represents a synchronization of threads.

Note that the Message corresponding to the numerically preceding sequence number is an implicit predecessor and need not be explicitly listed. All of the sequence numbers with the same prefix form a sequence. The numerical predecessor is the one in which the final term is one less. That is, number 3.1.4.5 is the predecessor of 3.1.4.6.

3.72.2.4 Sequence expression

The sequence-expression is a dot-separated list of sequence-terms followed by a colon (‘:’).

sequence-term ‘.’ . . . ‘:’

Each term represents a level of procedural nesting within the overall interaction. If all the control is concurrent, then nesting does not occur. Each sequence-term has the following syntax:

[*integer* | *name*] [*recurrence*]

The *integer* represents the sequential order of the Message within the next higher level of procedural calling. Messages that differ in one integer term are sequentially related at that level of nesting. Example: Message 3.1.4 follows Message 3.1.3 within activation 3.1. The *name* represents a concurrent thread of control. Messages that differ in the final name are concurrent at that level of nesting. Example: Message 3.1a and Message 3.1b are concurrent within activation 3.1. All threads of control are equal within the nesting depth.

The recurrence represents conditional or iterative execution. This represents zero or more Messages that are executed depending on the conditions involved. The choices are:

‘*’ ‘[’ iteration-clause ‘]’an iteration

‘[’ condition-clause ‘]’a branch

An iteration represents a sequence of Messages at the given nesting depth. The iteration clause may be omitted (in which case the iteration conditions are unspecified). The iteration-clause is meant to be expressed in pseudocode or an actual programming language, UML does not prescribe its format. An example would be: **[i := 1..n]*.

A condition represents a Message whose execution is contingent on the truth of the condition clause. The condition-clause is meant to be expressed in pseudocode or an actual programming language; UML does not prescribe its format. An example would be: *[x > y]*.

Note that a branch is notated the same as an iteration without a star. One might think of it as an iteration restricted to a single occurrence.

The iteration notation assumes that the Messages in the iteration will be executed sequentially. There is also the possibility of executing them concurrently. The notation for this is to follow the star by a double vertical line (for parallelism): *||.

Note that in a nested control structure, the recurrence is not repeated at inner levels. Each level of structure specifies its own iteration within the enclosing context.

3.72.2.5 Signature

A signature is a string that indicates the name, the arguments, and the return value of an Operation or a Reception. The signature of a Message is derived from (is the same as) the signature of the Operation invoked by the Message’s dispatching Procedure, or the Reception for the Signal sent by the Procedure. These have the following properties.

Return-value

This is a list of names that designates the values returned at the end of the communication within the subsequent execution of the overall interaction. These identifiers can be used as arguments to subsequent Messages. If the Message does not return a value, then the return value and the assignment operator are omitted.

Message-name

This is the name of the Operation to be applied on the receiver, or the Signal that is sent to the receiver.

Argument list

This is a comma-separated list of arguments (actual parameters) enclosed in parentheses. The parentheses can be used even if the list is empty. Each argument is either a reference to an Instance, or an expression in pseudocode or an appropriate programming language (UML does not prescribe). The expressions may use return values of previous messages (in the same scope) and navigation expressions starting from the source Instance; that is, Attributes of it and Links from it and paths reachable from them.

3.72.3 Presentation Options

Instead of text expressions for arguments and return values, data tokens may be shown near a message label. A token is a small circle labeled with the argument expression or return value name. It has a small arrow on it that points along the Message (for an argument) or opposite the Message (for a return value). Tokens represent arguments and return values. The choice of text syntax or tokens is a presentation option.

The syntax of Messages may instead be expressed in the syntax of a programming language, such as C++ or Smalltalk. All of the expressions on a single diagram should use the same syntax, however.

A return flow may be explicitly shown with a dashed arrow.

3.72.4 Example

See Figure 3-59 on page 3-116 for examples within a diagram.

Samples of control message label syntax:

2: display (x, y)	simple Message
1.3.1: p:= find(specs)	nested call with return value
4 [x < 0] : invert (x, color)	conditional Message
A3,B4/ C3.1*: update ()	synchronization with other threads, iteration

3.72.5 Mapping

An arrow symbol maps either onto a Message or a Stimulus. If the arrow is attached to a line corresponding to an AssociationRole, it maps onto a Message, with the ClassifierRoles corresponding to the end-points of the line as the sender and the receiver roles. If the line corresponds to a Link, the arrow maps onto a Stimulus, with the Instances corresponding to the end-points of the line as the sender and the receiver Instances. The line is the *communication connection* or the *communication link* of the Message or the Stimulus, respectively.

The control flow type sets the corresponding properties:

- *solid arrowhead*: a synchronous operation invocation
- *stick arrowhead*: an asynchronous operation invocation
- *dashed arrow with stick arrowhead:* return from an synchronous operation invocation

The predecessor expression, together with the sequence expression, determines the *predecessor* and *activation* (caller) relationships of a Message or a Stimulus. The predecessors of a Message (Stimulus) are those Messages (Stimuli) corresponding to the sequence numbers in the predecessor list as well as the Message (Stimulus) corresponding to the immediate preceding sequence number as the Message (Stimulus); that is, 1.2.2 is the one preceding 1.2.3. The caller is the ClassifierRole (Instance) receiving the Message (Stimulus) whose sequence number is truncated by one position; that is, 1.2 is the caller of 1.2.3. The thread-of-control name maps onto a Classifier stereotyped *thread*; that is, an active class.

The label of the arrow is mapped into either the body attribute of the Procedure, or into a detailed action model starting with recurrence. The return of a value maps into a Message from the called Instance to the caller with the dispatching Procedure that outputs the return value. Its *predecessor* is the final Message within the procedure. Its *activation* is the Message that called the procedure.

The recurrence expression, the iteration clause, and the condition clause determine if a ConditionalAction or LoopAction is used in the Procedure attached to the Message.

The operation name and the form of the signature determine the Operation attached to the CallOperationAction in the Procedure of the Message. Similarly for a Signal and SendSignalAction. The arguments of the signature determine the arguments associated with the CallOperationAction and SendSignalAction, respectively

In a procedural interaction, each arrow symbol also maps into a second Message representing the return flow, unless the return flow is explicitly shown. This Message has an *activation* Association to the original call Message. Its associated Procedure outputs the return values as arguments (if any).

3.73 Creation/Destruction Markers

3.73.1 Semantics

During the execution of an interaction some Instances and Links are created and some are destroyed. The creation and destruction of elements can be marked.

3.73.2 Notation

An Instance or a Link that is created during an interaction has the standard constraint *new* attached to it. An Instance or a Link that is destroyed during an interaction has the standard constraint *destroyed* attached. These constraints may be used even if the element has no name. Both constraints may be used together, but the standard constraint *transient* may be used in place of *new destroyed*.

3.73.3 Presentation options

Tools may use other graphic markers in addition to or in place of the keywords. For example, each kind of lifetime might be shown in a different color. A tool may also use animation to show the creation and destruction of elements and the state of the system at various times.

3.73.4 Example

See Figure 3-59 on page 3-116.

3.73.5 Mapping

Creation or destruction indicators map either into procedures containing CreateObjectActions or DestroyObjectActions in the corresponding ClassifierRoles. The former two Actions dispatch the Stimuli that cause the changes. These status indicators are merely summaries of the total actions.

Part 9 - Statechart Diagrams

A statechart diagram can be used to describe the behavior of instances of a model element such as an object or an interaction. Specifically, it describes possible sequences of states and actions through which the element instances can proceed during its lifetime as a result of reacting to discrete events (for example, signals, operation invocations).

The semantics and notation described in this chapter are substantially those of David Harel's statecharts with modifications to make them object-oriented. His work was a major advance on the traditional flat state machines. Statechart notation also implements aspects of both Moore machines and Mealy machines, traditional state machine models.

3.74 Statechart Diagram

3.74.1 Semantics

Statechart diagrams represent the behavior of entities capable of dynamic behavior by specifying its response to the receipt of event instances. Typically, it is used for describing the behavior of class instances, but statecharts may also describe the behavior of other entities such as use-cases, actors, subsystems, operations, or methods.

3.74.2 Notation

A statechart diagram is a graph that represents a state machine. States and various other types of vertices (pseudostates) in the state machine graph are rendered by appropriate state and pseudostate symbols, while transitions are generally rendered by directed arcs that interconnect them. States may also contain subdiagrams by physical containment or tiling. Note that every state machine has a top state that contains all the other elements of the entire state machine. The graphical rendering of this top state is optional.

The association between a state machine and its context does not have a special notation.

An example statechart diagram for a simple telephone object is depicted in Figure 3-71 on page 3-137.

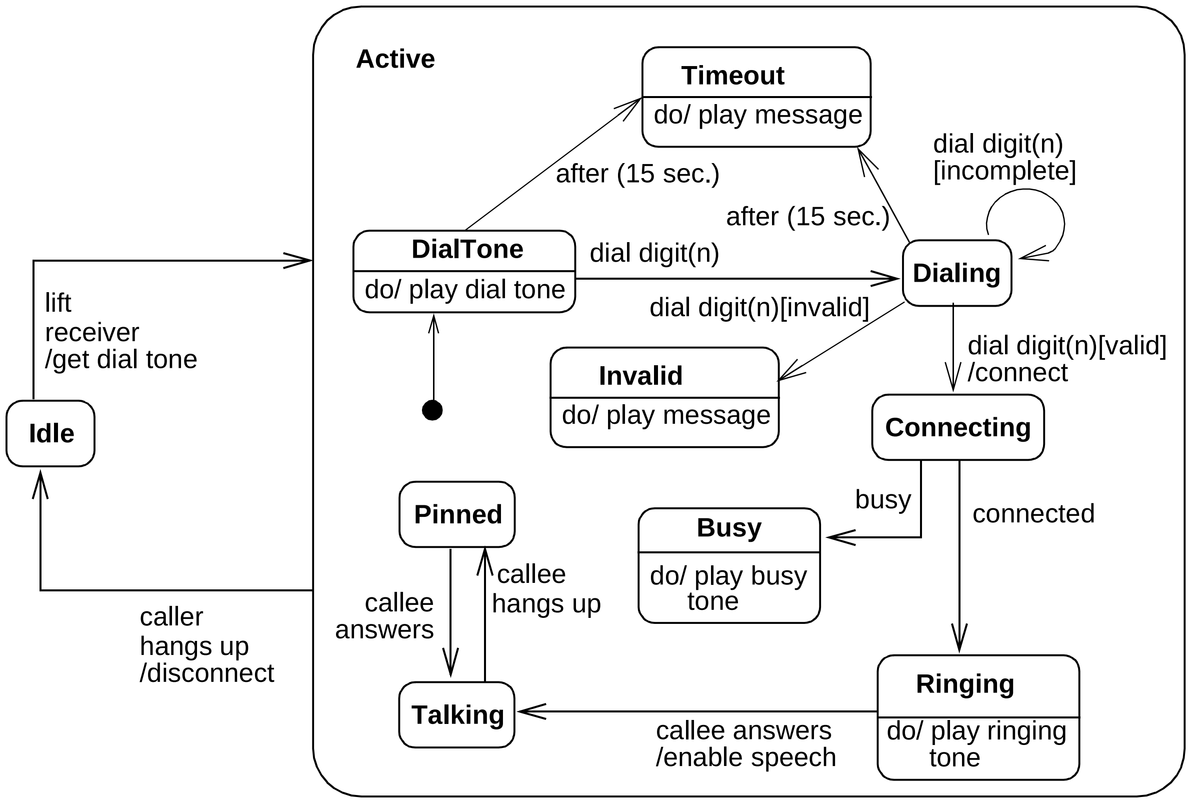


Figure 3-71 State Diagram

3.74.3 Mapping

A statechart diagram maps into a StateMachine. That StateMachine may be owned by an instance of a model element capable of dynamic behavior, such as classifier or a behavioral feature, which provides the context for that state machine. Different contexts may apply different semantic constraints on the state machine.

3.75 State

3.75.1 Semantics

A state is a condition during the life of an object or an interaction during which it satisfies some condition, performs some action, or waits for some event. A *composite* state is a state that, in contrast to a *simple* state, has a graphical decomposition. (Composite states and their notation are described in more detail in Section 3.76, “Composite States,” on page 3-140.) Conceptually, an object remains in a state for an interval of time. However, the semantics allow for modeling “flow-through” states that are instantaneous, as well as transitions that are not instantaneous.

A state may be used to model an ongoing activity. Such an activity is specified either by a nested state machine or by a computational expression.

3.75.2 Notation

A state is shown as a rectangle with rounded corners (Figure 3-72 on page 3-139). Optionally, it may have an attached name tab. The name tab is a rectangle, usually resting on the outside of the top side of a state and it contains the name of that state. It is normally used to keep the name of a composite state that has concurrent regions, but may be used in other cases as well (the Process state in Figure 3-77 on page 3-147 illustrates the use of the name tab).

A state may be optionally subdivided into multiple compartments separated from each other by a horizontal line. They are as follows:

- Name compartment

  This compartment holds the (optional) name of the state as a string. States without names are anonymous and are all distinct. It is undesirable to show the same named state twice in the same diagram, as confusion may ensue. Name compartments should not be used if a name tab is used and vice versa.

- Internal transitions compartment

  This compartment holds a list of internal actions or activities that are performed while the element is in the state.

The action label identifies the circumstances under which the action specified by the action expression will be invoked. The action expression may use any attributes and links that are in the scope of the owning entity. For list items where the action expression is empty, the backslash separator is optional.

A number of action labels are reserved for various special purposes and, therefore, cannot be used as event names. The following are the reserved action labels and their meaning.

entry	This label identifies an action, specified by the corresponding action expression, which is performed upon entry to the state (entry action).
exit	This label identifies an action, specified by the corresponding action expression, that is performed upon exit from the state (exit action).
do	This label identifies an ongoing activity (“do activity”) that is performed as long as the modeled element is in the state or until the computation specified by the action expression is completed (the latter may result in a completion event being generated).
include	This label is used to identify a submachine invocation. The action expression contains the name of the submachine that is to be invoked. Submachine states and the corresponding notation are described in Section 3.82, “Submachine States,” on page3-152.

In all other cases, the action label identifies the event that triggers the corresponding action expression. These events are called internal transitions and are semantically equivalent to self transitions *except that the state is not exited or re-entered*. This means that the corresponding exit and entry actions are not performed. The general format for the list item of an internal transition is:

event-name '(' *comma-separated-parameter-list* ')' '[' *guard-condition*']' '/' *action-expression*

Each event name may appear more than once per state if the guard conditions are different. The event parameters and the guard conditions are optional. If the event has parameters, they can be used in the action expression through the current event variable.

3.75.3 Example

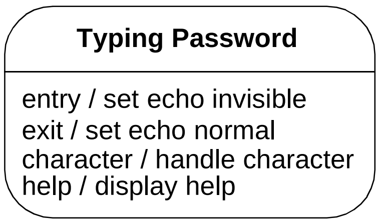


Figure 3-72 State

3.75.4 Mapping

A state symbol maps into a State. See Section 3.76, "Composite States," on page 3-140 for further details on which kind of state.

The name string in the symbol maps to the name of the state. Two symbols with the same name map into the same state. However, each state symbol with no name (or an empty name string) maps into a distinct anonymous State.

A list item in the internal transition compartment maps into a corresponding Action associated with a state. An "entry" list item; that is, an item with the "entry" label maps to the "entry" role, an "exit" list item maps to the "exit" role, and a "do" item maps to the "doActivity" role. (The mapping of "include" items is discussed in Section 3.82, "Submachine States," on page 3-152.)

A list item with an event name maps to a Transition associated with the "internal" role relative to the state. The action expression maps into the ActionSequence and Guard for the Transition. The event name and arguments map into an Event corresponding to the event name and arguments. The Event plays the role of a *trigger* to the Transition.

3.76 Composite States

3.76.1 Semantics

A composite state is decomposed into two or more concurrent substates (called *regions*) or into mutually exclusive disjoint substates. A given state may only be refined in one of these two ways. Naturally, any substate of a composite state can also be a composite state of either type.

A newly-created object takes its topmost default transition, originating from the topmost initial pseudostate. An object that transitions to its outermost final state is terminated.

Each region of a state may have initial pseudostates and final states. A transition to the enclosing state represents a transition to the initial pseudostate. A transition to a final state represents the completion of activity in the enclosing region. Completion of activity in all concurrent regions represents completion of activity by the enclosing state and triggers a completion event on the enclosing state. Completion of the top state of an object corresponds to its termination.

3.76.2 Notation

An expansion of a state shows its internal state machine structure. In addition to the (optional) name and internal transition compartments, the state may have an additional compartment that contains a region holding a nested diagram. For convenience and appearance, the text compartments may be shrunk horizontally within the graphic region.

An expansion of a state into concurrent substates is shown by tiling the graphic region of the state using dashed lines to divide it into regions. Each region is a concurrent substate. Each region may have an optional name and must contain a nested state diagram with disjoint states. The text compartments of the entire state are separated from the concurrent substates by a solid line. It is also possible to use a tab notation to place the name of a concurrent state. The tab notation is more space efficient.

An expansion of a state into disjoint substates is shown by showing a nested state diagram within the graphic region.

An initial pseudostate is shown as a small solid filled circle. In a top-level state machine, the transition from an initial pseudostate may be labeled with the event that creates the object; otherwise, it must be unlabeled. If it is unlabeled, it represents any transition to the enclosing state. The initial transition may have an action.

A final state is shown as a circle surrounding a small solid filled circle (a bull's eye). It represents the completion of activity in the enclosing state and it triggers a transition on the enclosing state labeled by the implicit activity completion event (usually displayed as an unlabeled transition), if such a transition is defined.

In some cases, it is convenient to hide the decomposition of a composite state. For example, the state machine inside a composite state may be very large and may simply not fit in the graphical space available for the diagram. In that case, the composite state may be represented by a simple state graphic with a special "composite" icon, usually in the lower right-hand corner. This icon, consisting of two horizontally placed and connected states, is an *optional* visual cue that the state has a decomposition that is not shown in this particular statechart diagram (Figure 3-74 on page 3-141). Instead, the contents of the composite state are shown in a separate diagram. Note that the "hiding" here is purely a matter of graphical convenience and has no semantic significance in terms of access restrictions.

3.76.3 Examples

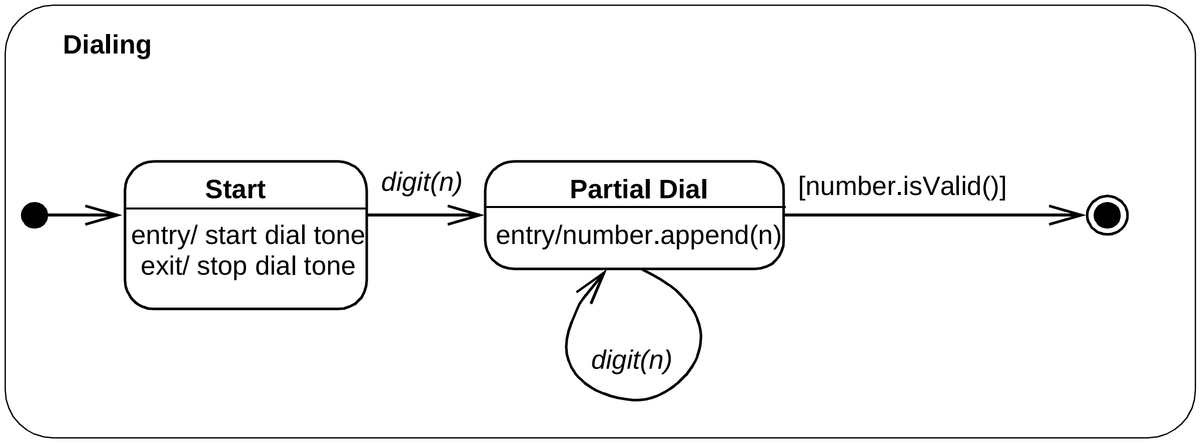


Figure 3-73 Sequential Substates

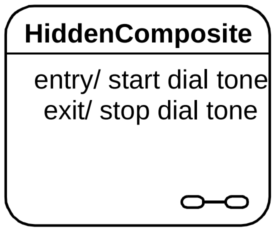


Figure 3-74 Composite State with hidden decomposition indicator icon

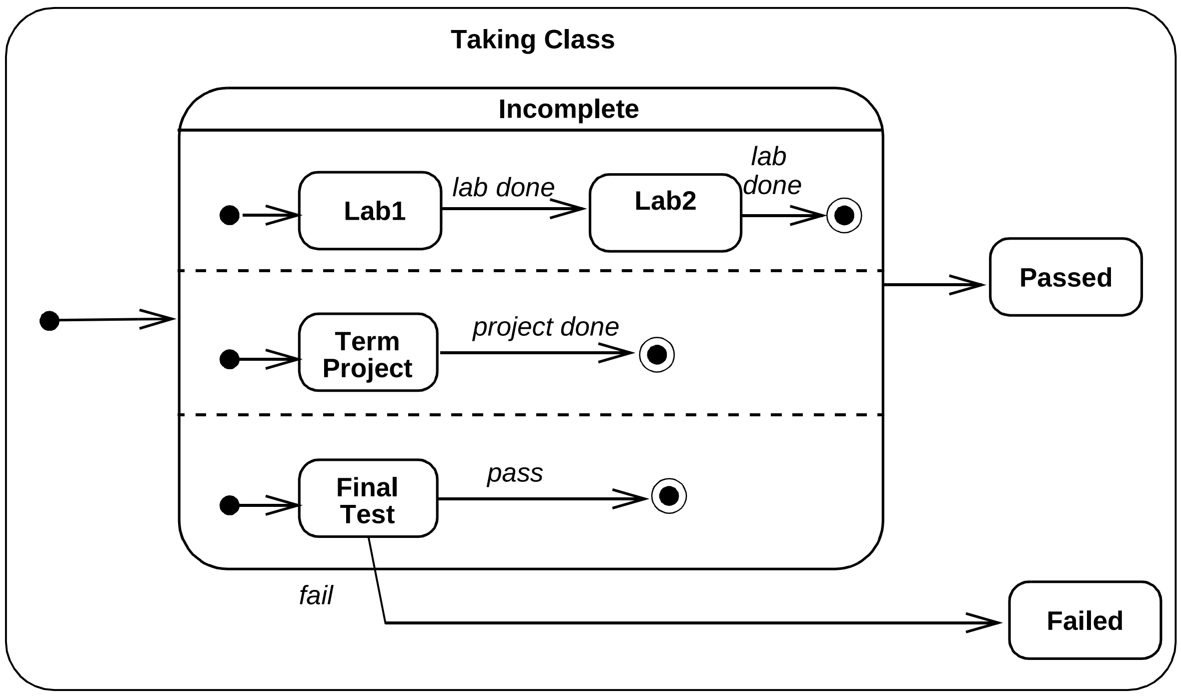


Figure 3-75 Concurrent Substates

3.76.4 Mapping

A state symbol maps into a State. If the symbol has no subdiagrams in it, it maps into a SimpleState. If it is tiled by dashed lines into regions, then it maps into a CompositeState with the *isConcurrent* value true; otherwise, it maps into a CompositeState with the *isConcurrent* value false. A region maps into a CompositeState with the *isRegion* value true and the *isConcurrent* value false.

An initial pseudostate symbol maps into a Pseudostate of kind *initial*. A final state symbol maps to a *final* state.

3.77 Events

3.77.1 Semantics

An event is a noteworthy occurrence. For practical purposes in state diagrams, it is an occurrence that may trigger a state transition. Events may be of several kinds (not necessarily mutually exclusive).

- A designated condition becoming true (described by a Boolean expression) results in a change event instance. The event occurs whenever the value of the expression changes from false to true. Note that this is different from a guard condition. A guard condition is evaluated *once* whenever its event fires. If it is false, then the transition does not occur and the event is lost.
- The receipt of an explicit signal from one object to another results in a signal event instance. It is denoted by the signature of the event as a trigger on a transition.
- The receipt of a call for an operation implemented as a transition by an object represents a call event instance.
- The passage of a designated period of time after a designated event (often the entry of the current state) or the occurrence of a given date/time is a TimeEvent.

The event declaration has scope within the package it appears in and may be used in state diagrams for classes that have visibility inside the package. An event is *not* local to a single class.

3.77.2 Notation

A signal or call event can be defined using the following format:

event-name '(' *comma-separated-parameter-list* ')

A parameter has the format:

parameter-name ':' *type-expression*

A signal can be declared using the «signal» keyword on a class symbol in a class diagram. The parameters are specified as attributes. A signal can be specified as a subclass of another signal. This indicates that an occurrence of the subevent triggers any transition that depends on the event or any of its ancestors.

An elapsed-time event can be specified with the keyword **after** followed by an expression that evaluates (at modeling time) to an amount of time, such as "**after** (5 seconds)" or **after** (10 seconds since exit from state A)." If no starting point is indicated, then it is the time since the entry to the current state. Other time events can be specified as conditions, such as **when** (date = Jan. 1, 2000).

A condition becoming true is shown with the keyword **when** followed by a Boolean expression. This may be regarded as a continuous test for the condition until it is true, although in practice it would only be checked on a change of values.

Signals can be declared on a class diagram with the keyword «signal» on a rectangle symbol. These define signal names that may be used to trigger transitions. Their parameters are shown in the attribute compartment. They have no operations. They may appear in a generalization hierarchy.

3.77.3 Example

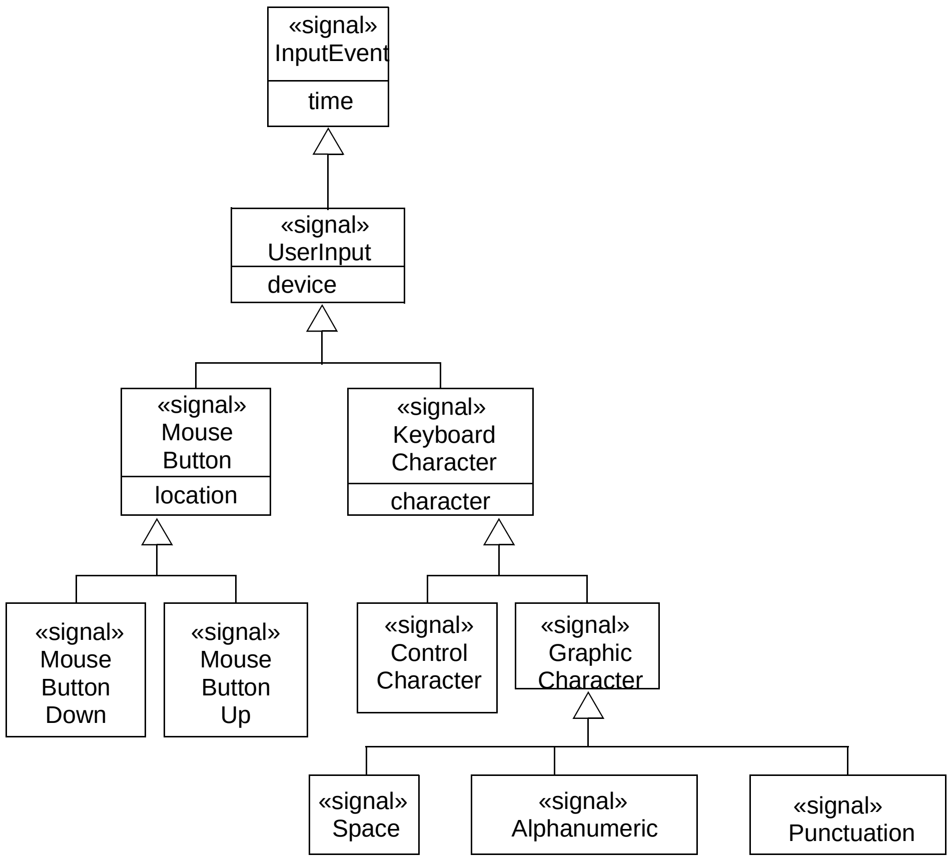


Figure 3-76 Signal Declaration

3.77.4 Mapping

A class box with stereotype «signal» maps into a Signal. The name and parameters are given by the name string and the attribute list of the box. Generalization arrows between signal class boxes map into Generalization relationships between the Signal.

The usage of an event string expression in a context requiring an event maps into an implicit reference of the Event with the given name. It is an error if various uses of the same name (including any explicit declarations) do not match.

3.78 Simple Transitions

3.78.1 Semantics

A simple transition is a relationship between two states indicating that an instance in the first state will enter the second state and perform specific actions when a specified event occurs provided that certain specified conditions are satisfied. On such a change of state, the transition is said to “fire.” The trigger for a transition is the occurrence of the event labeling the transition. The event may have parameters, which are accessible by the actions specified on the transition as well as in the corresponding exit and entry actions associated with the source and target states respectively. Events are processed one at a time. If an event does not trigger any transition, it is discarded. If it can trigger more than one transition within the same sequential region; that is, not in different concurrent regions, only one will fire. If these conflicting transitions are of the same priority, an arbitrary one is selected and triggered.

3.78.2 Notation

A transition is shown as a solid line originating from the *source* state and terminated by an arrow on the *target* state. It may be labeled by a *transition string* that has the following general format:

event-signature ‘[’ guard-condition ‘]’ ‘/’ *action-expression*

The *event-signature* describes an event with its arguments:

event-name ‘(’ *comma-separated-parameter-list* ‘)’

The *guard-condition* is a Boolean expression written in terms of parameters of the triggering event and attributes and links of the object that owns the state machine. The guard condition may also involve tests of concurrent states of the current machine, or explicitly designated states of some reachable object (for example, “**in** State1” or “**not in** State2”). State names may be fully qualified by the nested states that contain them, yielding pathnames of the form “State1::State2::State3.” This may be used in case same state name occurs in different composite state regions of the overall machine.

The *action-expression* is executed if and when the transition fires. It may be written in terms of operations, attributes, and links of the owning object and the parameters of the triggering event, or any other features visible in its scope. The corresponding action must be executed entirely before any other actions are considered. This model of execution is referred to as *run-to-completion* semantics. The action expression may be an action sequence comprising a number of distinct actions including actions that explicitly generate events, such as sending signals or invoking operations. The details of this expression are dependent on the action language chosen for the model.

3.78.2.1 Transition times

Names may be placed on transitions to designate the times at which they fire. See Section 3.64, “Transition Times,” on page 3-113.

3.78.3 Example

```
right-mouse-down (location) [location in window] / object := pick-object (location);
object.highlight ()
```

The event may be any of the standard event types. Selecting the type depends on the syntax of the name (for time events, for example); however, SignalEvents and CallEvents are not distinguishable by syntax and must be discriminated by their declaration elsewhere.

3.78.4 Mapping

A transition string and the transition arrow that it labels together map into a Transition and its attachments. The arrow connects two state symbols. The Transition has the corresponding States as its source (the state at the tail) and destination (the state at the head) States in associations to the Transition.

The event name and parameters map into an Event element, which may be a SignalEvent, a CallEvent, a TimeExpression (if it has the proper syntax), or a ChangeEvent (if it is expressed as a Boolean expression). The event is attached as a “trigger” role in the association to the transition.

The guard condition maps into a Guard element attached to the Transition. Note that a guard condition is distinguished graphically from a change event specification by being enclosed in brackets.

An action expression maps into an Action attached as an “effect” role relative to the Transition.

3.79 Transitions to and from Concurrent States

A concurrent transition may have multiple source states and target states. It represents a synchronization and/or a splitting of control into concurrent threads without concurrent substates.

3.79.1 Semantics

A concurrent transition is enabled when all the source states are occupied. After a compound transition fires, all its destination states are occupied.

3.79.2 Notation

A concurrent transition includes a short heavy bar (a *synchronization* bar, which can represent synchronization, forking, or both). The bar may have one or more arrows from states to the bar (these are the *source states*). The bar may have one or more arrows from the bar to states (these are the *destination states*). A transition string may be shown near the bar. Individual arrows do not have their own transition strings.

3.79.3 Example

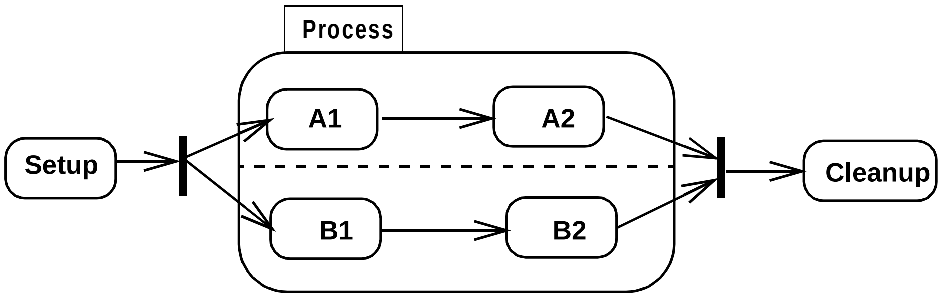


Figure 3-77 Concurrent Transitions

3.79.4 Mapping

A bar with multiple transition arrows leaving it maps into a fork pseudostate. A bar with multiple transition arrows entering it maps into a join pseudostate. The transitions corresponding to the incoming and outgoing arrows attach to the pseudostate as if it were a regular state. If a bar has multiple incoming and multiple outgoing arrows, then it maps into a join connected to a fork pseudostate by a single transition with no attachments.

3.80 Transitions to and from Composite States

3.80.1 Semantics

A transition drawn to the boundary of a composite state is equivalent to a transition to its initial point (or to a complex transition to the initial point of each of its concurrent regions, if it is concurrent). The entry action is always performed when a state is entered from outside.

A transition from a composite state indicates a transition that applies to each of the states within the state region (at any depth). It is “inherited” by the nested states. Inherited transitions can be masked by the presence of nested transitions with the same trigger.

3.80.2 Notation

A transition drawn to a composite state boundary indicates a transition to the composite state. This is equivalent to a transition to the initial pseudostate within the composite state region. The initial pseudostate must be present. If the state is a concurrent composite state, then the transition indicates a transition to the initial pseudostate of each of its concurrent substates.

Transitions may be drawn directly to states within a composite state region at any nesting depth. All entry actions are performed for any states that are entered on any transition. On a transition within a concurrent composite state, transition arrows from the synchronization bar may be drawn to one or more concurrent states. Any other concurrent regions start with their default initial pseudostate.

A transition drawn from a composite state boundary indicates a transition of the composite state. If such a transition fires, any nested states are forcibly terminated and perform their exit actions, then the transition actions occur and the new state is established.

Transitions may be drawn directly from states within a composite state region at any nesting depth to outside states. All exit actions are performed for any states that are exited on any transition. On a transition from within a concurrent composite state, transition arrows may be specified from one or more concurrent states to a synchronization bar; therefore, specific states in the other regions are irrelevant to triggering the transition.

A state region may contain a *history state indicator* shown as a small circle containing an 'H.' The history indicator applies to the state region that directly contains it. A history indicator may have any number of incoming transitions from outside states. It may have at most one outgoing unlabeled transition. This identifies the default "previous state" if the region has never been entered. If a transition to the history indicator fires, it indicates that the object resumes the state it last had within the composite region. Any necessary entry actions are performed. The history indicator may also be 'H*' for *deep history*. This indicates that the object resumes the state it last had at any depth within the composite region, rather than being restricted to the state at the same level as the history indicator. A region may have both shallow and deep history indicators.

3.80.3 Presentation Options

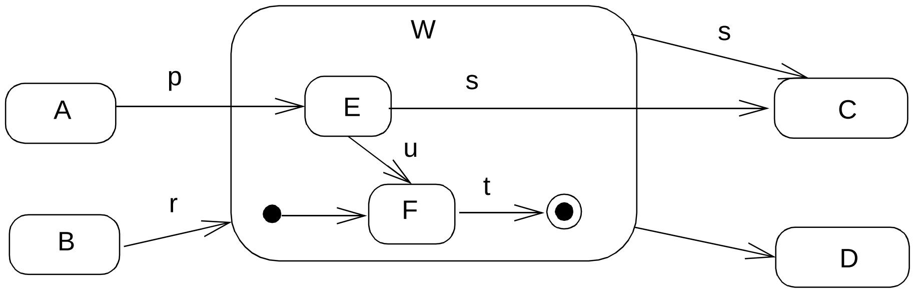
3.80.3.1 Stubbed transitions

Nested states may be suppressed. Transitions to nested states are subsumed to the most specific visible enclosing state of the suppressed state. Subsumed transitions that do not come from an unlabeled final state or go to an unlabeled initial pseudostate may (but need not) be shown as coming from or going to *stubs*. A *stub* is shown as a small vertical line (bar) drawn inside the boundary of the enclosing state. It indicates a transition connected to a suppressed internal state. Stubs are not used for transitions to initial or from final states.

Note that events should be shown on transitions leading into a state, either to the state boundary or to an internal substate, including a transition to a stubbed state. Normally events should not be shown on transitions leading from a stubbed state to an external state. Think of a transition as belonging to its source state. If the source state is suppressed, then so are the details of the transition. Note also that a transition from a final state is summarized by an unlabeled transition from the composite state contour (denoting the implicit event "action complete" for the corresponding state).

3.80.4 Example

See Figure 3-76 on page 3-144 and Figure 3-77 on page 3-147 for examples of composite transitions. The following are examples of stubbed transitions and the history indicator.



may be abstracted as

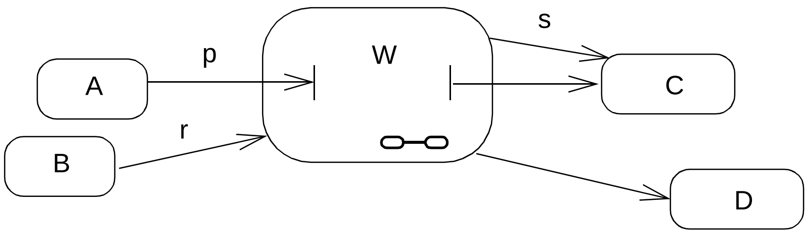


Figure 3-78 Stubbed Transitions

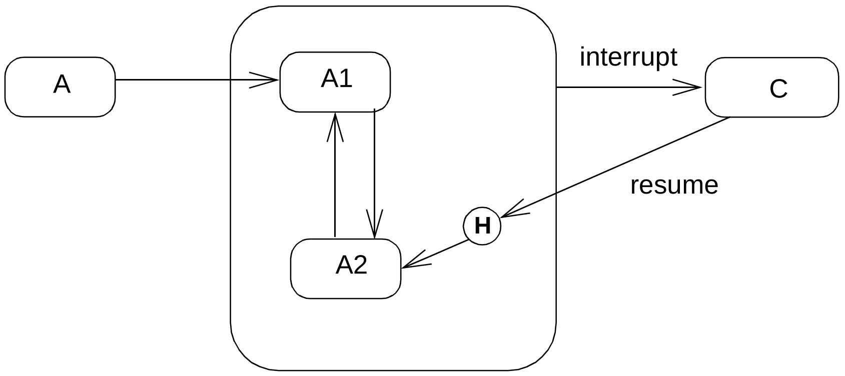


Figure 3-79 History Indicator

3.80.5 Mapping

An arrow to any state boundary, nested or not, maps into a Transition between the corresponding States and similarly for transitions directly to history states.

A history indicator maps into a Pseudostate of kind *shallowHistory* or *deepHistory*.

A stubbed transition does not map into anything in the model. It is a notational elision that indicates the presence of transitions to additional states in the model that are not visible in the diagram.

3.81 Factored Transition Paths

3.81.1 Semantics

By definition, a transition connects exactly two vertices in the state machine graph. However, since some of these vertices may be pseudostates—which are transient in nature—there is a need for describing chains of transitions that may be executed in the context of a single run-to-completion step. Such a transition is known as a *compound transition*.

As a practical measure, it is often useful to share segments of a compound transition. For example, two or more distinct compound transitions may come together and continue via a common path, sharing its action, and possibly terminating on the same target state. In other cases, it may be useful to split a transition into separate mutually exclusive; that is, non-concurrent paths.

Both of these examples of graphical factoring in which some transitions are shared result in simplified diagrams. However, factoring is also useful for modeling dynamically adaptive behavior. An example of this occurs when a single event may lead to any of a set of possible target states, but where the final target state is only determined as the result of an action (calculation) performed after the triggering of the compound transition.

Note that the splitting and joining of paths due to factoring is different from the splitting and joining of concurrent transitions described in Sect i on3.79, "Transitions to and from Concurrent States," on page 3-146. The sources and targets of these factored transitions are not concurrent.

3.81.2 Notation

Two or more transitions emanating from different non-concurrent states or pseudostates can terminate on a common junction point. This allows their respective compound transitions to share the path that emanates from that junction point. A junction point is represented by a small black circle. Alternatively, it may be represented by a diamond shape (see Section 3.87, "Decisions," on page 3-159).

Two or more guarded transitions emanating from the same junction point represent a *static branch point*. Normally, the guards are mutually exclusive. This is equivalent to a set of individual transitions, one for each path through the tree, whose guard condition is the “and” of all of the conditions along the path. Note that the semantics of static branches is that all the outgoing guards are evaluated *before* any transition is taken.

Two or more guarded transitions emanating from a common *dynamic choice point* are used to model dynamic choices. In this case, the guards of the outgoing transitions are evaluated at the time the choice point has been reached. The value of these guards may be a function of some calculations performed in the actions of the incoming transition (s). A dynamic choice point is represented by a small white circle (reminiscent of a small state icon).

3.81.3 Examples

In Figure 3-80 a single junction point is used to merge and split transitions. Regardless of whether the junction point was reached from state State0 or from state State1, the outgoing paths are the same for both cases.

If the state machine in this example is in state State1 and b is less than 0 when event e1 occurs, the outgoing transition will be taken only if one of the three downstream guards is true. Thus, if a is equal to 6 at that point, no transition will be triggered.

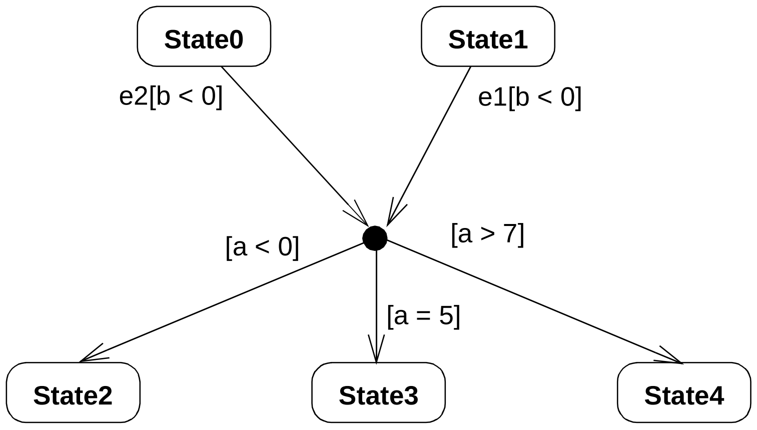


Figure 3-80 Junction points

In the dynamic choice point example in Figure 3-81 on page 3-152, the decision on which branch to take is only made after the transition from State1 is taken and the choice point is reached. Note that the action associated with that incoming transition computes a new value for a. This new value can then be used to determine the outgoing transition to be taken. The use of the predefined condition[else] is recommended to avoid run-time errors.

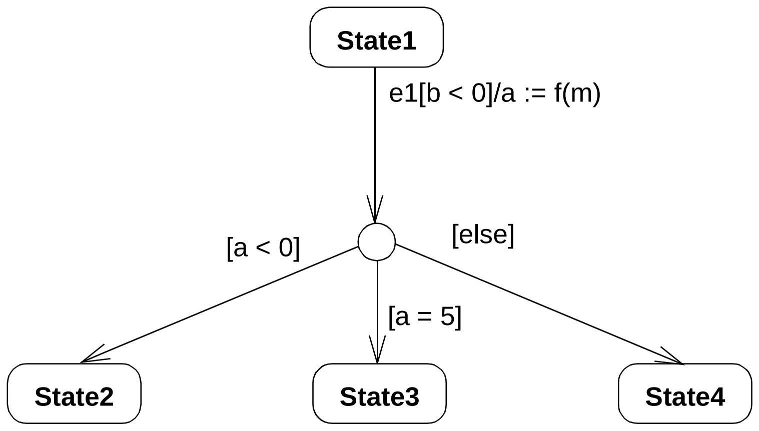


Figure 3-81 Dynamic choice points

3.82 Submachine States

3.82.1 Semantics

A submachine state represents the *invocation* of a state machine defined elsewhere. It is similar to a macro call in the sense that it represents a (graphical) shorthand that implies embedding of a complex specification within another specification. The submachine must be contained in the same context as the invoking state machine.

In the general case, an invoked state machine can be entered at any of its substates or through its default (initial) pseudostate. Similarly, it can be exited from any substate or as a result of the invoked state machine reaching its final state or by an “inherited” or “group” transition that applies to all substates in the submachine.

The non-default entry and exits are specified through special stub states.

3.82.2 Notation

The submachine state is depicted as a normal state with the appropriate “include” declaration within its internal transitions compartment (see Section 3.75, “State,” on page 3-137). The expression following the include reserved word is the name of the invoked submachine.

Optionally, the submachine state may contain one or more entry stub states and one or more exit stub states. The notation for these is similar to that used for stub ends of stubbed transitions, except that the ends are labeled. The labels represent the names of the corresponding substates within the invoked submachine. A pathname may be used if the substate is not defined at the top level of the invoked submachine. Naturally, this name must be a valid name of a state in the invoked state machine.

If the submachine is entered through its default pseudostate or if it is exited as a result of the completion of the submachine, it is not necessary to use the stub state notation for these cases. Similarly, a stub state is not required if the exit occurs through an explicit "group" transition that emanates from the boundary of the submachine state (implying that it applies to all the substates of the submachine).

Submachine states invoking the same submachine may occur multiple times in the same state diagram with different entry and exit configurations and with different internal transitions and exit and entry action specifications in each case.

3.82.3 Example

The following diagram shows a fragment from a statechart diagram in which a submachine (the FailureSubmachine) is invoked in a particular way. The actual submachine is presumably defined elsewhere and is not shown in this diagram. Note that the same submachine could be invoked elsewhere in the same statechart diagram with different entry and exit configurations.

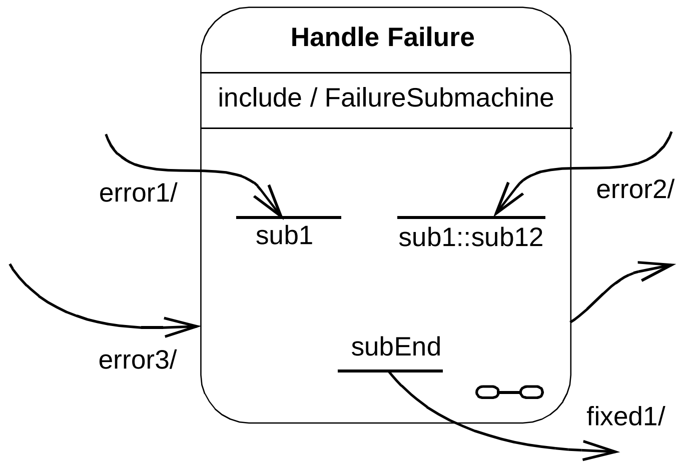


Figure 3-82 Submachine State

In the above example, the transition triggered by event "error1" will terminate on state "sub1" of the FailureSubmachine state machine. Since the entry point does not contain a path name, this means that "sub1" is defined at the top level of that submachine. In contrast, the transition triggered by "error2" will terminate on the "sub12" substate of the "sub1"substate (as indicated by the path name), while the "error3" transition implies taking of the default transition of the FailureSubmachine.

The transition triggered by the event "fixed1" emanates from the "subEnd" substate of the submachine. Finally, the transition emanating from the edge of the submachine state is taken as a result of the completion event generated when the FailureSubmachine reaches its final state.

3.82.4 Mapping

A submachine state in a statechart diagram maps directly to a SubmachineState in the metamodel. The name following the “include” reserved action label represents the state machine indicated by the “submachine” attribute. Stub states map to the Stub State concept in the metamodel. The label on the diagram corresponds to the pathname represented by the “referenceState” attribute of the stub state.

3.83 Synch States

3.83.1 Semantics

A synch state is for synchronizing concurrent regions of a state machine. It is used in conjunction with forks and joins to insure that one region leaves a particular state or states before another region can enter a particular state or states. The firing of outgoing transitions from a synch state can be limited by specifying a bound on the difference between the number of times outgoing and incoming transitions have fired.

3.83.2 Notation

A synch state is shown as a small circle with the upper bound inside it. The bound is either a positive integer or an asterisk ('*') for unlimited. Synch states are drawn on the boundary between two regions when possible.

3.83.3 Example

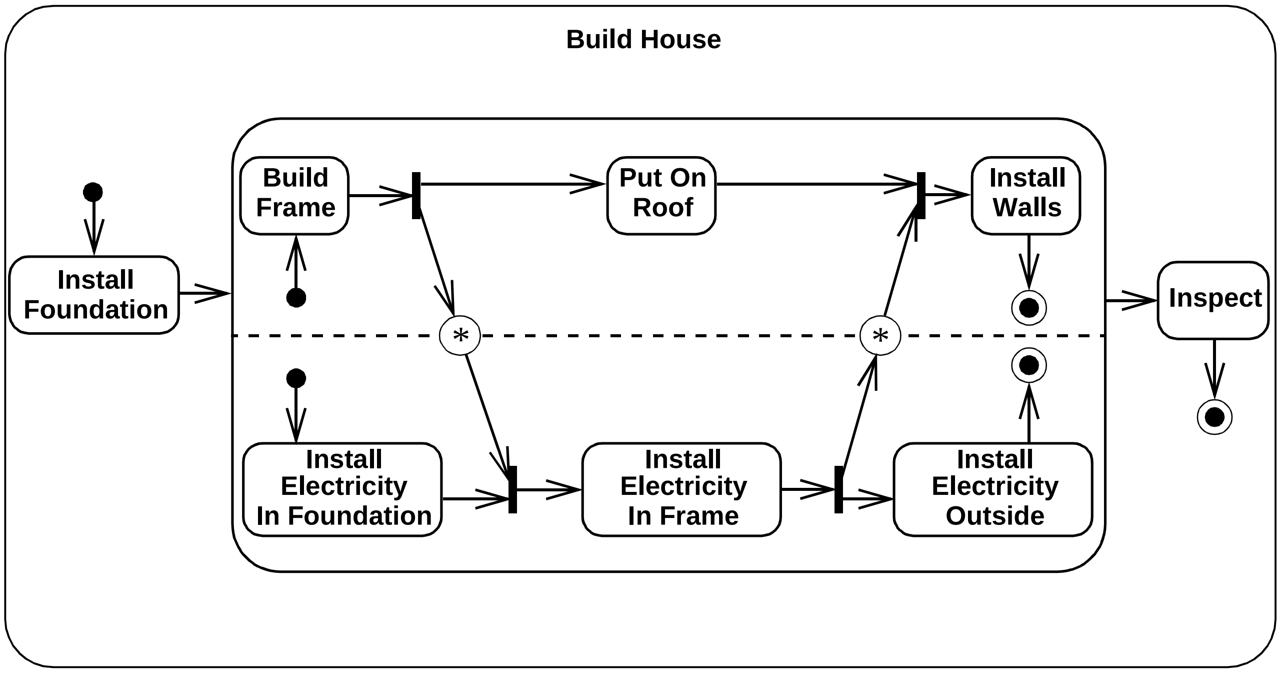

Build House
Build Frame
Put On Roof
Install Walls
Install Foundation
Inspect
*
*
Install Electricity In Foundation
Install Electricity In Frame
Install Electricity Outside

Figure 3-83 Synch states

3.83.4 Mapping

A synch state circle maps into a SynchState, contained by the least common containing state of the regions it is synchronizing. The number inside it maps onto the bound attribute of the synch state. A star ('*') inside the synch state circle maps to a value of Unlimited for the bound attribute.

Part 10 - Activity Diagrams

3.84 Activity Diagram

3.84.1 Semantics

An activity graph is a variation of a state machine in which the states represent the performance of actions or subactivities and the transitions are triggered by the completion of the actions or subactivities. It represents a state machine of a computation itself.

3.84.2 Notation

An activity diagram is a special case of a state diagram in which all (or at least most) of the states are action or subactivity states and in which all (or at least most) of the transitions are triggered by completion of the actions or subactivities in the source states. The entire activity diagram is attached (through the model) to a classifier, such as a use case, or to a package, or to the implementation of an operation. The purpose of this diagram is to focus on flows driven by internal processing (as opposed to external events). Use activity diagrams in situations where all or most of the events represent the completion of internally-generated actions (that is, procedural flow of control). Use ordinary state diagrams in situations where asynchronous events occur.

3.84.3 Example

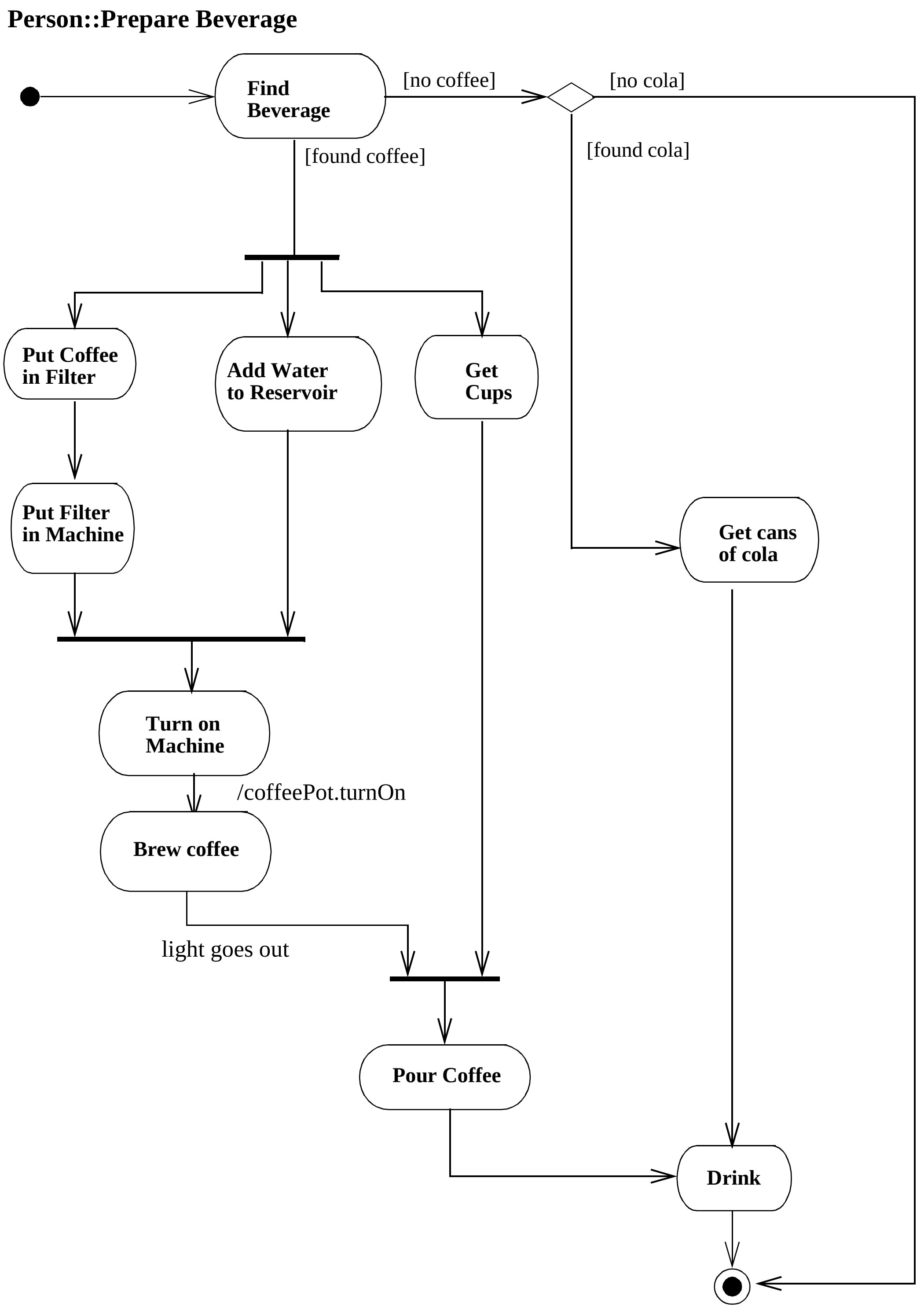


Figure 3-84 Activity Diagram

3.84.4 Mapping

An activity diagram maps into an ActivityGraph.

3.85 Action state

3.85.1 Semantics

An *action state* is a shorthand for a state with an entry action and at least one outgoing transition involving the implicit event of completing the entry action (there may be several such transitions if they have guard conditions). Action states should not have internal transitions, outgoing transitions based on explicit events, or exit actions, use normal states for this situation. Transitions leaving an action state should not include an event signature. Such transitions are implicitly triggered by the completion of the action in the state. The transitions may include guard conditions and actions. A common use of an action state is to model a step in the execution of a workflow process.

3.85.2 Notation

An action state is shown as a shape with straight top and bottom and with convex arcs on the two sides. The *action-expression* is placed in the symbol. The action expression need not be unique within the diagram.

3.85.3 Presentation options

The action may be described by natural language, pseudocode, action language, or programming language code. It may use only attributes and links of the owning object.

Note that action state notation may be used within ordinary state diagrams; however, they are more commonly used with activity diagrams, which are special cases of state diagrams.

3.85.4 Example

matrix.invert (tolerance:Real)
drive to work

Figure 3-85 Action States

3.85.5 Mapping

An action state symbol maps into an ActionState with the action-expression mapped to either the body of the entry action procedure of the State, or to a detailed action model within the procedure. The State is normally anonymous.

3.86 Subactivity state

3.86.1 Semantics

A *subactivity state* invokes an activity graph. When a subactivity state is entered, the activity graph “nested” in it is executed as any activity graph would be. The subactivity state is not exited until the final state of the nested graph is reached, or when trigger events occur on transitions coming out of the subactivity state. Since states in activity graphs do not normally have trigger events, subactivity states are normally exited when their nested graph is finished. A single activity graph may be invoked by many subactivity states.

3.86.2 Notation

A subactivity state is shown in the same way as an action state with the addition of an icon in the lower right corner depicting a nested activity diagram. The name of the subactivity is placed in the symbol. The subactivity need not be unique within the diagram.

This notation is applicable to any UML construct that supports “nested” structure. The icon must suggest the type of nested structure.

3.86.3 Example

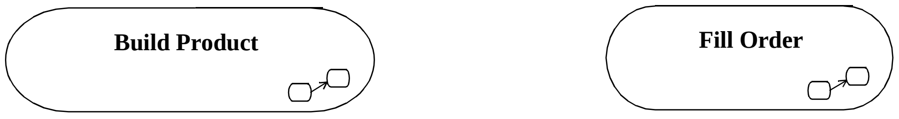


Figure 3-86 Subactivity States

3.86.4 Mapping

A subactivity state symbol maps into a SubactivityState. The name of the subactivity maps to a submachine link between the SubactivityState and an ActivityGraph of that name. The SubactivityState is normally anonymous.

3.87 Decisions

3.87.1 Semantics

A state diagram (and by derivation an activity diagram) expresses a decision when guard conditions are used to indicate different possible transitions that depend on Boolean conditions of the owning object. UML provides a shorthand for showing decisions and merging their separate paths back together. Each possible outcome should appear on one of the outgoing transitions. A predefined guard denoted “else” may be defined for at most one outgoing transition. This transition is enabled if all the guards labeling the other transitions are false.

3.87.2 Notation

A decision may be shown by labeling multiple output transitions of an action with different guard conditions.

The icon provided for a decision is the traditional diamond shape, with one incoming arrow and with two or more outgoing arrows, each labeled by a distinct guard condition with no event trigger.

The same icon can be used to merge decision branches back together, in which case it is called a merge. A merge has two or more incoming arrows and one outgoing arrow.

Note that a chain of decisions may be part of a complex transition, but only the first segment in such a chain may contain an event trigger label. All segments may have guard expressions. The transition coming from a merge may not have a trigger label or guard expressions.

3.87.3 Example

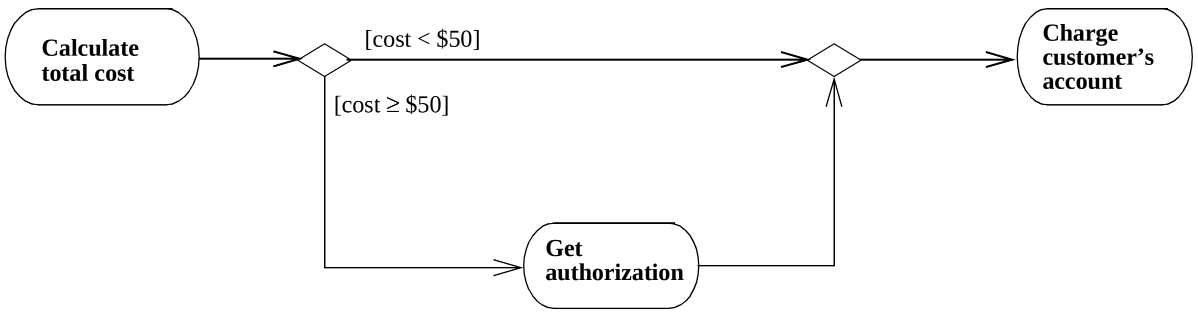


Figure 3-87 Decision and merge

3.87.4 Mapping

A decision symbol maps into a Pseudostate of kind *junction*. Each label on an outgoing arrow maps into a Guard on the corresponding Transition leaving the Pseudostate. A merge symbol maps also maps into a Pseudostate of kind *junction*.

3.88 Call States

3.88.1 Semantics

A call state is an action state that calls a single operation. It is useful in object flow modeling to reduce notational ambiguity over which action is taking input or providing output.

3.88.2 Notation

A call state is shown in the same way as an action state, except that the name of the operation being called is put in the symbol, along with the name of the classifier that hosts the operation in parentheses under it.

3.88.3 Example

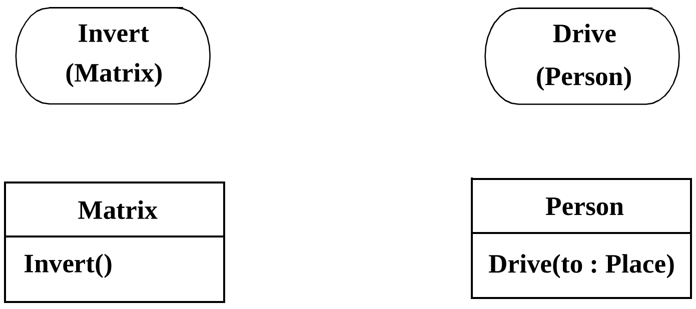


Figure 3-88 Call states and the operations they invoke

3.88.4 Mapping

The top name maps into the operation being called in the entry action of the call state. The name in parentheses maps into the classifier hosting the operation.

3.89 Swimlanes

3.89.1 Semantics

Actions and subactivities may be organized into *swimlanes.* Swimlanes are used to organize responsibility for actions and subactivities. They often correspond to organizational units in a business model.

3.89.2 Notation

An activity diagram may be divided visually into “swimlanes,” each separated from neighboring swimlanes by vertical solid lines on both sides. The relative ordering of the swimlanes has no semantic significance. Each action is assigned to one swimlane. Transitions may cross lanes. There is no significance to the routing of a transition path.

3.89.3 Example

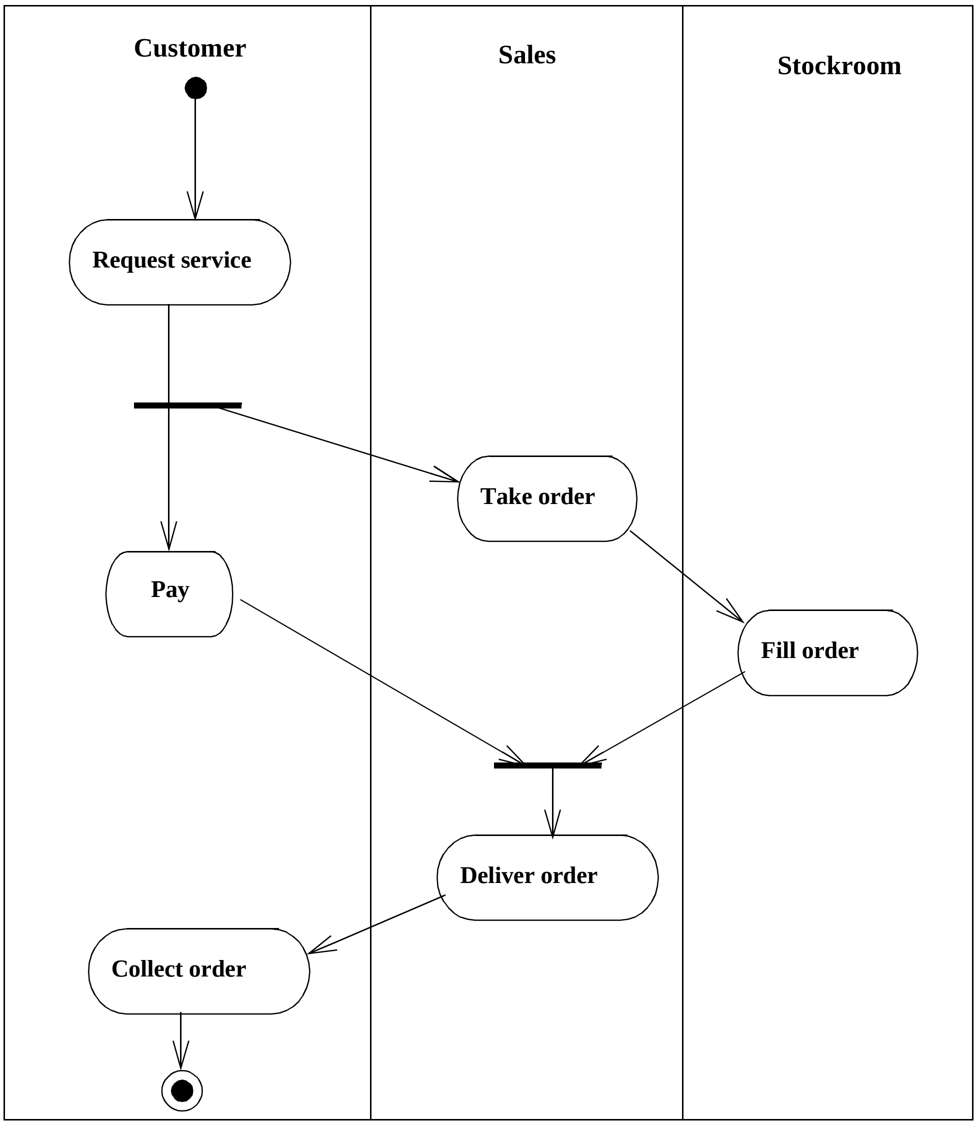


Figure 3-89 Swimlanes in Activity Diagram

3.89.4 Mapping

A swimlane maps into a Partition of the States in the ActivityGraph. A state symbol in a swimlane causes the corresponding State to belong to the corresponding Partition.

3.90 Action-Object Flow Relationships

3.90.1 Semantics

Actions operate by and on objects. These objects either have primary responsibility for initiating an action, or are used or determined by the action. Actions usually specify calls sent between the object owning the activity graph, which initiates actions, and the objects that are the targets of the actions.

3.90.2 Notation

3.90.2.1 Object responsible for an action

In sequence diagrams, the object responsible for performing an action is shown by drawing a lifeline and placing actions on lifelines. See “Sequence Diagram” on page 3-102. Activity diagrams do not show the lifeline, but each action specifies which object performs its operation. These objects may also be related to the swimlane in some way. The actions within a swimlane can all be handled by the same object or by multiple objects.

3.90.2.2 Object flow

Objects that are input to or output from an action may be shown as object symbols. A dashed arrow is drawn from an action state to an output object, and a dashed arrow is drawn from an input object to an action state. The same object may be (and usually is) the output of one action and the input of one or more subsequent actions.

The control flow (solid) arrows must be omitted when the object flow (dashed) arrows supply a redundant constraint. In other words, when a state produces an output that is input to a subsequent state, that object flow relationship implies a control constraint.

3.90.2.3 Object in state

Frequently the same object is manipulated by a number of successive actions or subactivities. It is possible to show one object with arrows to and from all of the relevant actions and subactivities, but for greater clarity, the object may be displayed multiple times on a diagram. Each appearance denotes a different point during the object’s life. To distinguish the various appearances of the same object, the state of the object at each point may be placed in brackets and appended to the name of the object (for example, PurchaseOrder[approved]). This notation may also be used in collaboration and sequence diagrams.

3.90.3 Example

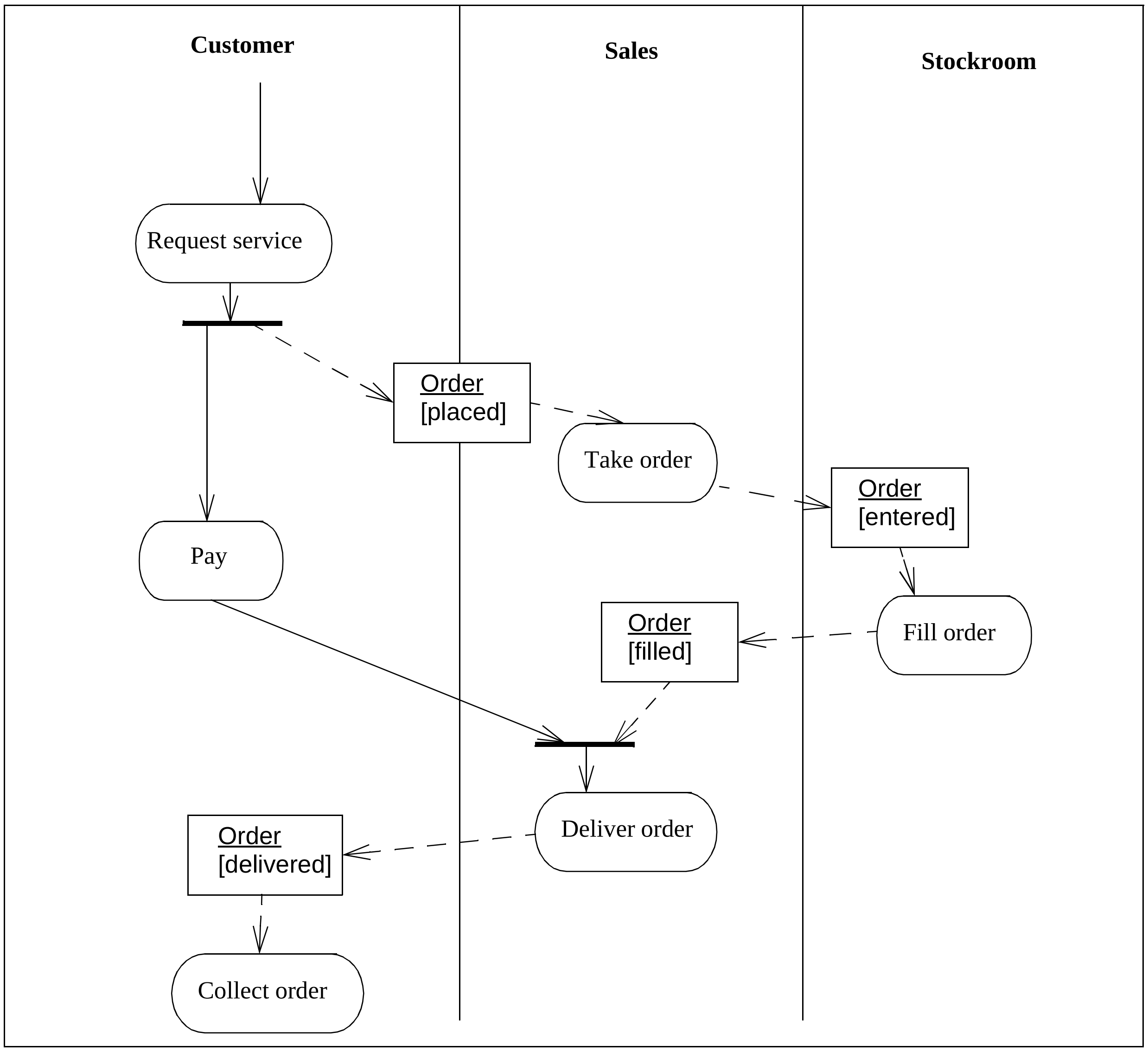


Figure 3-90 Actions and Object Flow

3.90.4 Mapping

An object flow symbol maps into an ObjectFlowState whose incoming and outgoing Transitions correspond to the incoming and outgoing arrows. The Transitions have no attachments. The classifier name and (optional) state name of the object flow symbol map into a Class or a ClassifierInState corresponding to the name(s). Solid and dashed arrows both map to transitions.

3.91 Control Icons

The following icons provide explicit symbols for certain kinds of information that can be specified on transitions. These icons are not necessary for constructing activity diagrams, but many users prefer the added impact that they provide.

3.91.1 Notation

3.91.1.1 Signal receipt

The receipt of a signal may be shown as a concave pentagon that looks like a rectangle with a triangular notch in its side (either side). The signature of the signal is shown inside the symbol. A unlabeled transition arrow is drawn from the previous action state to the pentagon and another unlabeled transition arrow is drawn from the pentagon to the next action state. A dashed arrow may be drawn from an object symbol to the notch on the pentagon to show the sender of the signal; this is optional.

3.91.1.2 Signal sending

The sending of a signal may be shown as a convex pentagon that looks like a rectangle with a triangular point on one side (either side). The signature of the signal is shown inside the symbol. A unlabeled transition arrow is drawn from the previous action state to the pentagon and another unlabeled transition arrow is drawn from the pentagon to the next action state. A dashed arrow may be drawn from the point on the pentagon to an object symbol to show the receiver of the signal, this is optional.

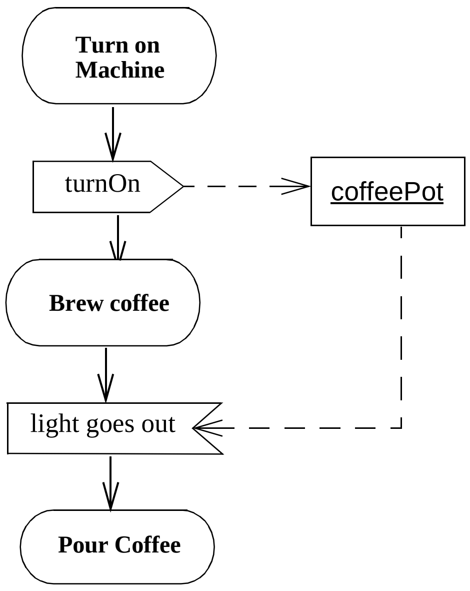


Figure 3-91 Symbols for Signal Receipt and Sending

3.91.1.3 *Deferred events*

A frequent situation is when an event that occurs must be “deferred” for later use while some other action or subactivity is underway. (Normally an event that is not handled immediately is lost.) This may be thought of as having an internal transition that handles the event and places it on an internal queue until it is needed or until it is discarded. Each state specifies a set of events that are deferred if they occur during the state and are not used to trigger a transition. If an event is not included in the set of deferrable events for a state, and it does not trigger a transition, then it is discarded from the queue even if it has already occurred. If a transition depends on an event, the transition fires immediately if the event is already on the internal queue. If several transitions are possible, the leading event in the queue takes precedence.

A deferrable event is shown by listing it within the state followed by a slash and the special operation *defer*. If the event occurs, it is saved and it recurs when the object transitions to another state, where it may be deferred again. When the object reaches a state in which the event is not deferred, it must be accepted or lost. The indication may be placed on a composite state or its equivalents, submachine and subactivity states, in which case it remains deferrable throughout the composite state. A contained transition may still be triggered by a deferrable event, whereupon it is removed from the queue.

It is not necessary to defer events on action states, because these states are not interruptible for event processing. In this case, both deferred and undeferred events that occur during the state are deferred until the state is completed. This means that the timing of the transition will be the same regardless of the relative order of the event and the state completion, and regardless of whether events are deferred.

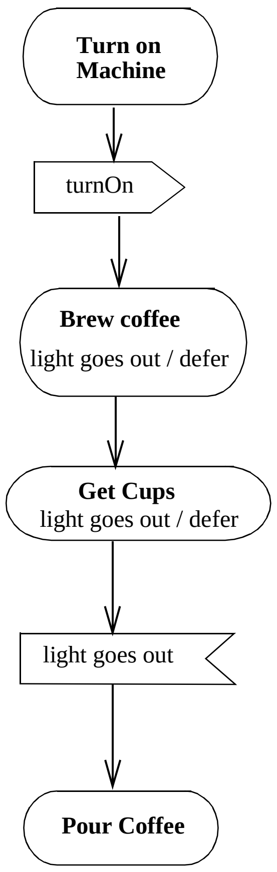


Figure 3-92 Deferred Event

3.91.2 Mapping

A signal receipt symbol maps into a state with no actions or internal transitions. Its specified event maps to a trigger event on the outgoing transition between it and the following state.

A signal send symbol maps into a procedure containing a SendSignalAction on the incoming transition between it and the previous state.

A deferred event attached to a state maps into a *deferrableEvent* association from the State to the Event.

3.92 Synch States

The SynchState notation may be omitted in Activity Diagrams when a SynchState has one incoming and one outgoing transition, and an unlimited bound. The semantics and mapping are the same as if the synch state circles were included, as defined for state machine notation.

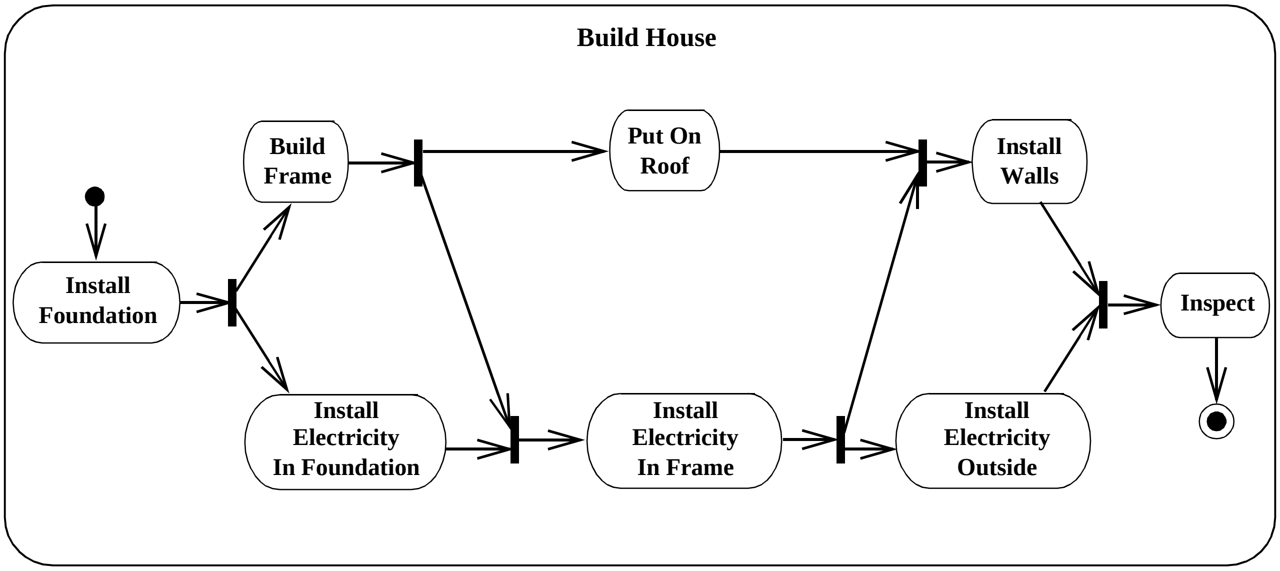

Build House
Build Frame
Put On Roof
Install Walls
Install Foundation
Inspect
Install Electricity In Foundation
Install Electricity In Frame
Install Electricity Outside

Figure 3-93 Synchronizing parallel activities

3.93 Dynamic Invocation

3.93.1 Semantics

The actions of an action state or the activity graph of a subactivity state may be executed more than once concurrently. The number of concurrent invocations is determined at runtime by a concurrency expression, which evaluates to a set of argument lists, one argument list for each invocation.

3.93.2 Notation

If the dynamic concurrency of an action or subactivity state is not always exactly one, its multiplicity is shown in the upper right corner of the state. Otherwise, nothing is shown.

3.93.3 Mapping

A multiplicity string in the upper right corner of an action or subactivity state maps to the same value in the dynamicMultiplicity attribute of the state. The presence of a multiplicity string also maps to a value of true for the isDynamic attribute of the state. If no multiplicity is present, the value of the isDynamic attribute is false.

3.94 Conditional Forks

In Activity Diagrams, transitions outgoing from forks may have guards. This means the region initiated by a fork transition might not start, and therefore is not required to complete at the corresponding join. The usual notation and mapping for guards may be used on the transition outgoing from a fork.

Part 11 - Implementation Diagrams

Implementation diagrams show aspects of physical implementation, including the structure of components and the run-time deployment system. They come in two forms: 1) component diagrams show the structure of components, including the classifiers that specify them and the artifacts that implement them; and 2) deployment diagrams show the structure of the nodes on which the components are deployed. These diagrams can also be applied in a broader way to business modeling where the components represent business procedures and artifacts, and the deployment nodes represent the organization units and resources (human and otherwise) of the business.

3.95 Component Diagram

3.95.1 Semantics

A component diagram shows the dependencies among software components, including the classifiers that specify them (for example, implementation classes) and the artifacts that implement them; such as, source code files, binary code files, executable files, scripts.

A component diagram has only a type form, not an instance form. To show component instances, use a deployment diagram (possibly a degenerate one without nodes).

3.95.2 Notation

A component diagram is a graph of components connected by dependency relationships. Components may also be connected to components by physical containment representing composition relationships.

Classifiers that specify components can be connected to them by physical containment or by a «reside» relationship, which is an instance of the metaassociation between Component and ModelElement. Likewise, artifacts that specify components can be connected to them by physical containment or by an «implement» relationship, which is an instance of the metaassociation between Component and Artifact.

A diagram containing component types may be used to show static dependencies, such as compiler dependencies between programs, which are shown as dashed arrows (dependencies) from a client component to a supplier component that it depends on in some way. The kinds of dependencies are implementation-specific and may be shown as stereotypes of the dependencies.

Although a component does not have its own features (for example, attributes, operations), it acts as a container for other classifiers that are defined with features. Components typically expose a set of interfaces, which represent the services provided by the elements that reside on the component. The diagram may show these interfaces and calling dependencies among components, using dashed arrows from components to interfaces on other components.

3.95.3 Example

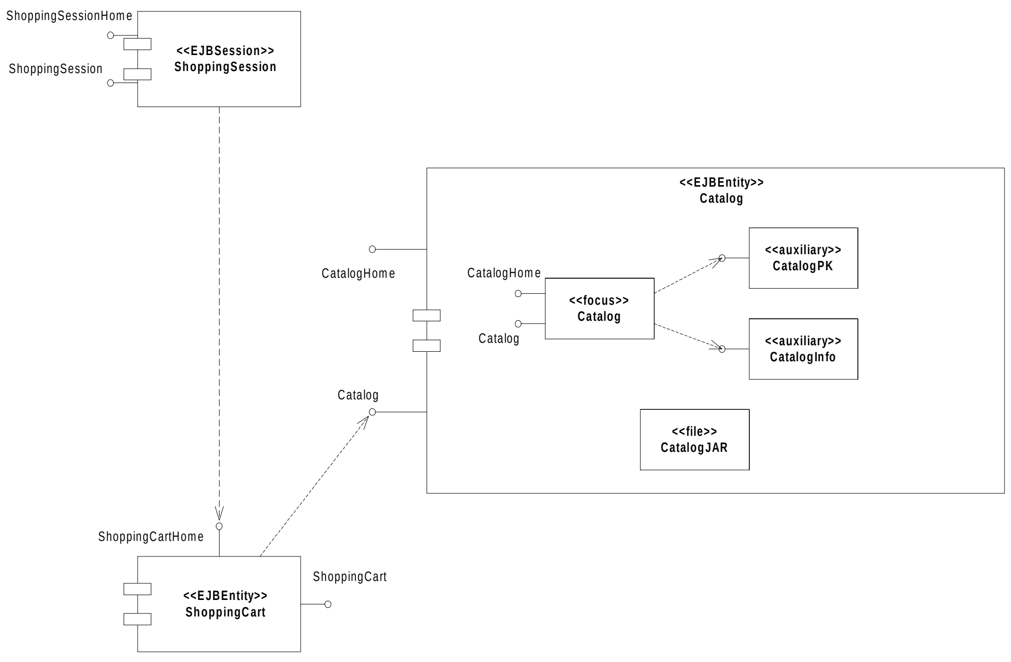


Figure 3-94 Component Diagram

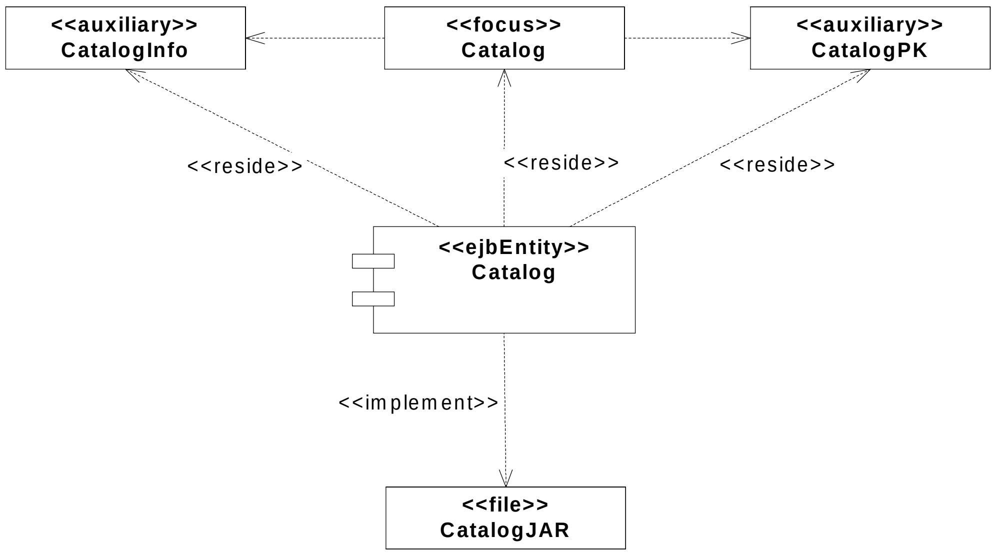


Figure 3-95 Component Diagram Showing Relationships with Classifiers and Artifacts

3.95.4 Mapping

A component diagram maps to a static model whose elements include Components. The physical containment of a Classifier by a Component represents a «reside» relationship, which is an instance of the metaassociation between Component and ModelElement. The physical containment of an Artifact by a Component represents an «implement» relationship, which is an instance of the metaassociation between Component and Artifact.

3.96 Deployment Diagram

3.96.1 Semantics

Deployment diagrams show the configuration of run-time processing elements and the software components, processes, and objects that execute on them. Software component instances represent run-time manifestations of software code units. Components that do not exist as run-time entities (because they have been compiled away) do not appear on these diagrams, they should be shown on component diagrams.

For business modeling, the run-time processing elements include workers and organizational units, and the software components include procedures and documents used by the workers and organizational units.

3.96.2 Notation

A deployment diagram is a graph of nodes connected by communication associations. Nodes may contain component instances. This indicates that the component runs or executes on the node. Components may contain instances of classifiers, which indicates that the instance resides on the component. Components are connected to other components by dashed-arrow dependencies (possibly through interfaces). This indicates that one component uses the services of another component. A stereotype may be used to indicate the precise dependency, if needed.

The deployment type diagram may also be used to show which components may reside on which nodes, by using dashed arrows with the stereotype «deploy» from the component symbol to the node symbol or by graphically nesting the component symbol within the node symbol.

Migration of component instances from node instance to node instance or objects from component instance to component instance may be shown using the «become» stereotype of the dependency relationship. In this case the component instance or object is resident on its node instance or component instance only part of the entire time.

Note that a process is just a special kind of object (see Section 3.71, "Active object," on page 3-128).

3.96.3 Example

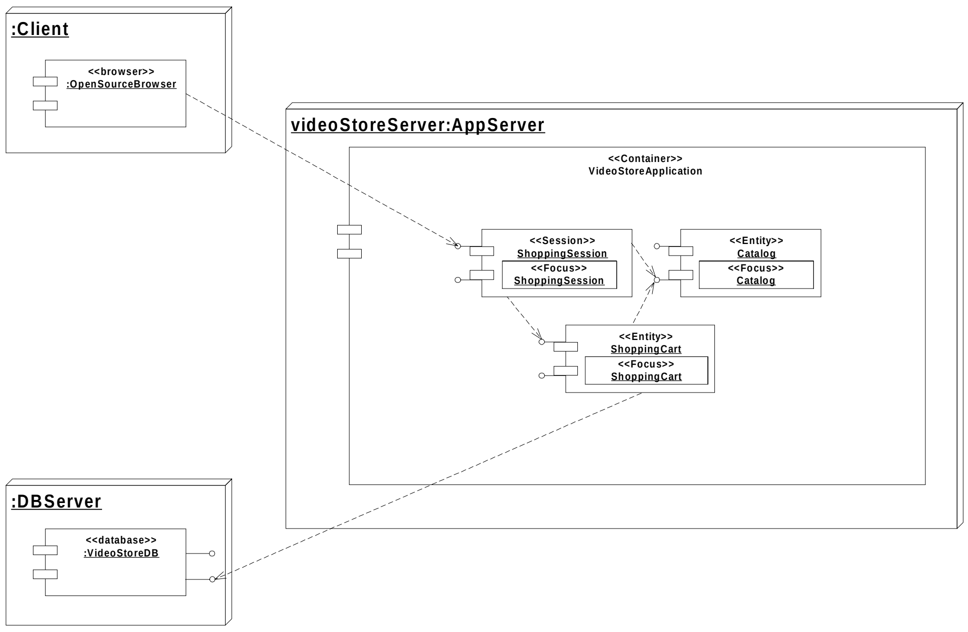


Figure 3-96 Deployment Diagram

3.96.4 Mapping

A deployment diagram maps to a static model whose elements include Nodes. It is not particularly distinguished in the model.

3.97 Node

3.97.1 Semantics

A node is a physical object that represents a processing resource, generally, having at least a memory and often processing capability as well. Nodes include computing devices but also human resources or mechanical processing resources. Nodes may be represented as types and as instances. Run time computational instances, both objects and component instances, may reside on node instances.

3.97.2 Notation

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

node-type

A node instance has a name and a type name. The node may have an underlined name string in it or below it. The name string has the syntax:

name ':' *node-type*

The name is the name of the individual node (if any). The node-type says what kind of a node it is. Either or both elements are optional; if the node-type is omitted, then so is the colon.

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

Component instances and objects may be contained within node instance symbols. This indicates that the items reside on the node instances.

Nodes may be connected by associations to other nodes. An association between nodes indicates a communication path between the nodes. The association may have a stereotype to indicate the nature of the communication path (for example, the kind of channel or network).

3.97.3 Example

This example shows two nodes containing components, where a «become» flow shows the *backupBroker* migrating from the *backupServer* to the *primaryServer* while the other components remain in place.

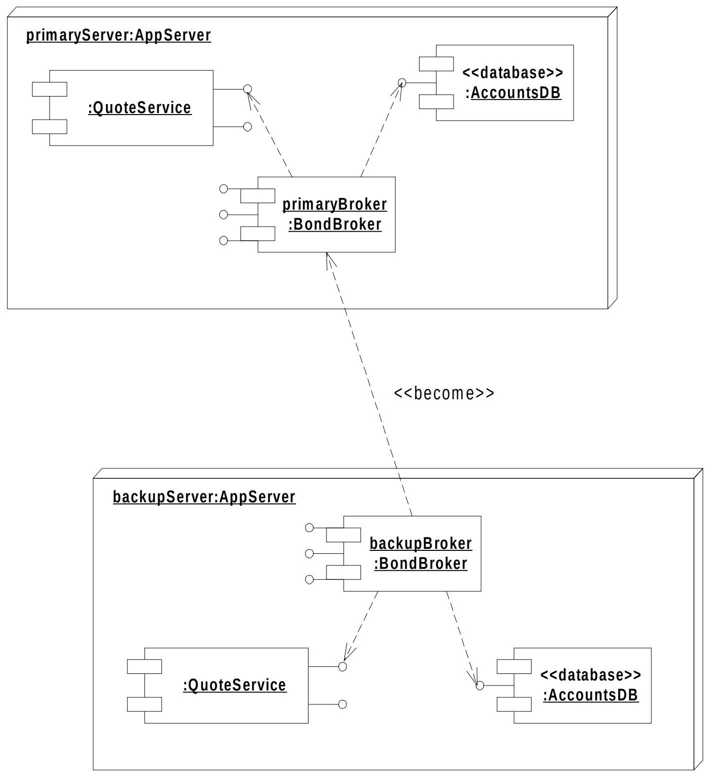


Figure 3-97 Node and Component Instances

3.97.4 Mapping

A node maps to a Node.

A «deploy» arrow or the nesting of a component symbol within a node symbol maps into a residence metassociation between Component and Node. The nesting of a component-instance symbol within a node-instance symbol maps to a residence metaassociation between the ComponentInstance and the NodeInstance.

3.98 Component

3.98.1 Semantics

A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.

A component is typically specified by one or more classifiers that reside on the component. A subset of these classifiers explicitly define the component's external interfaces. A component conforms to the interfaces that it exposes, where the interfaces represent services provided by elements that reside on the component. A component may be implemented by one or more artifacts, such as binary, executable, or script files. A component may be deployed on a node.

3.98.2 Notation

A component is shown as a rectangle with two small rectangles protruding from its side. A component type has a type name:

component-type

A component instance has a name and a type. The name of the component and its type may be shown as an underlined string either within the component symbol or above or below it, with the syntax:

component-name ':' *component-type*

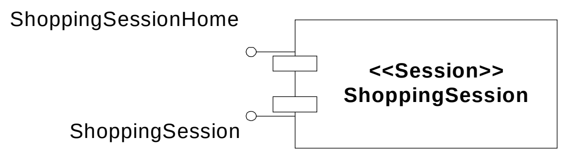
Either or both elements are optional. If the component-type is omitted, then so is the colon.

Objects that reside on a component instance are shown as nested inside the component instance symbol. By analogy, classes that are implemented by a component may be shown as nested within it; this indicates residence and not ownership.

Elements that reside on a component are shown nested inside the component symbol. The visibility of a resident element to other components may be shown using the same notation as for the visibility of the contents of a package (prepending a visibility symbol to the name of the package). The meaning of the visibility depends on the nature of the component. For a source-language component (such as program text), it would control the accessibility of source-language constructs. For a run-time code component (such as executable code), it would control the ability of code in other components to call or otherwise access code in the component.

3.98.3 Example

The example shows a component with interfaces and also a component that contains objects at run time.



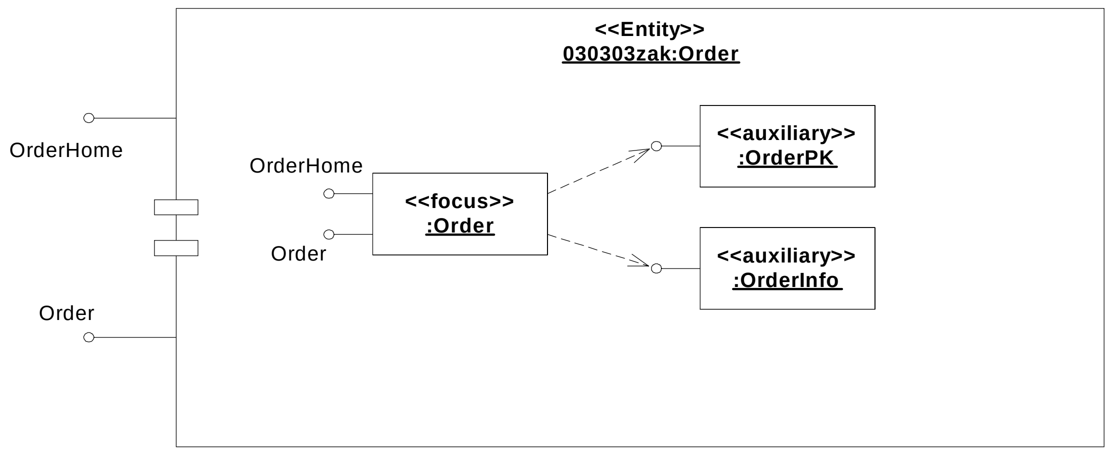


Figure 3-98 Components

3.98.4 *Mapping*

A component symbol maps to a Component.

The graphical nesting of an element (other than a component symbol) in a component symbol maps to an ElementResidence metaassociation class between ModelElement and the Component. Graphical nesting of a component symbol in another component symbol maps to a composition association. The graphical nesting of an instance symbol in a component instance symbol maps to a residence metaassociation between Instance and ComponentInstance.