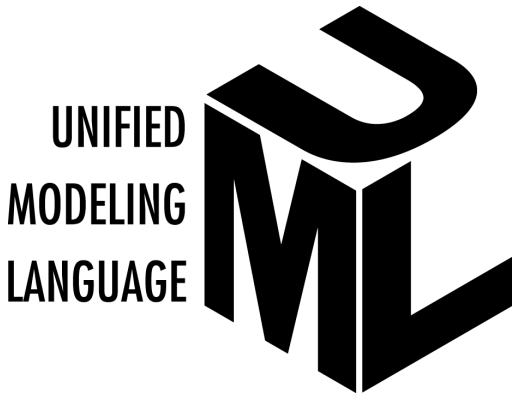


Notation Guide

version 1.0
13 January 1997

ad/97-01-09

RATIONAL
SOFTWARE CORPORATION
2800 San Tomas Expressway
Santa Clara, CA 95051-0951
http://www.rational.com

Contents

Contents

1. DOCUMENT OVERVIEW

This document describes the notation for the visual representation of the Unified Modeling Language (UML). This document should be used in conjunction with the companion *UML Semantics* document. This notation document contains brief summaries of the semantics of UML constructs, but the semantics document must be consulted for full details.

This document is arranged into chapters according to diagram types. Within each diagram type are listed model elements that are found on that diagram and their representation. Note, however, 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 other places than the chapter 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 chapter is divided into numbered 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* document.

Notation: Explains the notational elements of the feature.

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 we do not want to restrict this excessively. In some sense, we are defining the “paper notation” that printed documents show, rather than the “screen notation”. We realize that the ability to extend the notation can lead to unintelligible dialects so we hope that this freedom will be used in intuitive ways. We 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 one of the presentation options and implement it; tools should give the users the options of selecting among various presentation options, including some that are not described in this document.

Style guidelines: Suggestions for the use of stylistic markers, such as fonts, naming conventions, arrangement of symbols, etc., 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 sugges-

tions, but the use of some consistent guidelines of your own choosing is recommended in any case.

2. DIAGRAM ORGANIZATION

2.1 GRAPHS AND THEIR CONTENTS

Most UML diagrams 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 topological relationships that are important: connection (usually of lines to 2-d shapes), containment (of symbols by 2-d shapes with boundaries), and visual attachment (one symbol being "near" another one on a diagram).

Note that UML notation is basic 2-dimensional. 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 but it is not currently practical.

There are basically four kinds of graphical constructs that are used in UML notation: icons, 2-d symbols, paths, and strings.

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 stand-alone symbols that may or may not be connected to paths.

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.

Paths are 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.

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.

2.2 DRAWING PATHS

Path consist of a series of line segments whose endpoints coincide. The entire path is a single topological unit. Line segments may be drawn at any angle (oblique lines). One style option is to restrict all lines to fall on a rectilinear grid, but this 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.

2.3 INVISIBLE HYPERLINKS AND THE ROLE OF TOOLS

A notation on a piece of paper contains no hidden information. A notation on a computer screen, however, 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 document 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.

2.4 BACKGROUND INFORMATION

2.4.1 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 most usefully presented 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, however, that hidden links may exist from graphical items to tabular items.

2.5 NOTE

A note is a comment placed on the diagram. It is attached to the diagram rather than to a model element, unless it is stereotyped to be a constraint.

2.5.1 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 diagrams and may be attached to zero or more modeling elements by dashed lines.

2.5.2 Presentation options

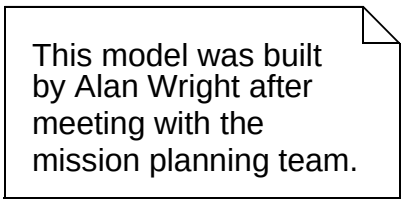
A note may have a stereotype.

A note with the stereotype “constraint” or a more specific form 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). Other kinds of notes are purely notation; they have no underlying model element.

2.5.3 Example

See also Section 2.6.2 for a note symbol containing a constraint.

Figure 1. Note



2.6 CONSTRAINT

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 “or” constraint) are predefined in UML, others may be user-defined. A user-defined constraint is described in words 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.

2.6.1 Notation

A constraint is shown as a text string in braces ({ }). UML does not prescribe the language in which the constraint is written. However, there is an expectation that individual tools may provide one or more languages in which formal constraints may be written. Otherwise the constraint may be written in natural language.

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

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.

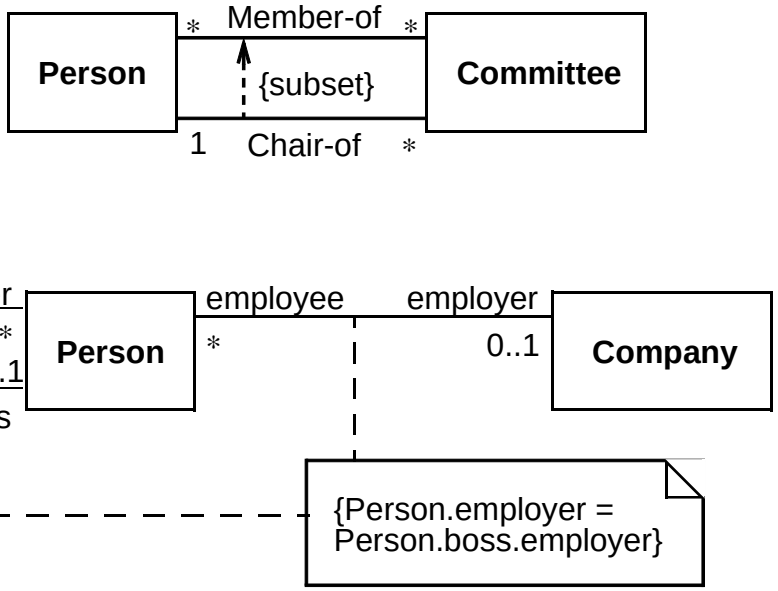
For a single graphical symbol (such as a class or an association path): The constraint string may be placed near the symbol, preferably near the name of the symbol, if any.

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.

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.

2.6.2 Example

Figure 2. Constraints



2.7 PACKAGES AND MODEL ORGANIZATION

A package is a grouping of model elements. Packages themselves may be nested within other packages. A package may contain both subordinate packages and ordinary model elements. The entire system can be thought of as a single high-level package with everything else in it. All kinds of UML model elements and diagrams can be organized into packages.

2.7.1 Notation

A package is shown as a large rectangle with a small rectangle (a “tab”) attached on one corner (preferably the left side of the upper side of the large rectangle). It is a manila folder icon.

If contents of the package are not shown, then the name of the package is placed within the large rectangle.

If contents of the package are shown, then the name of the package may be placed within the tab.

A stereotype string may be placed above the package name.

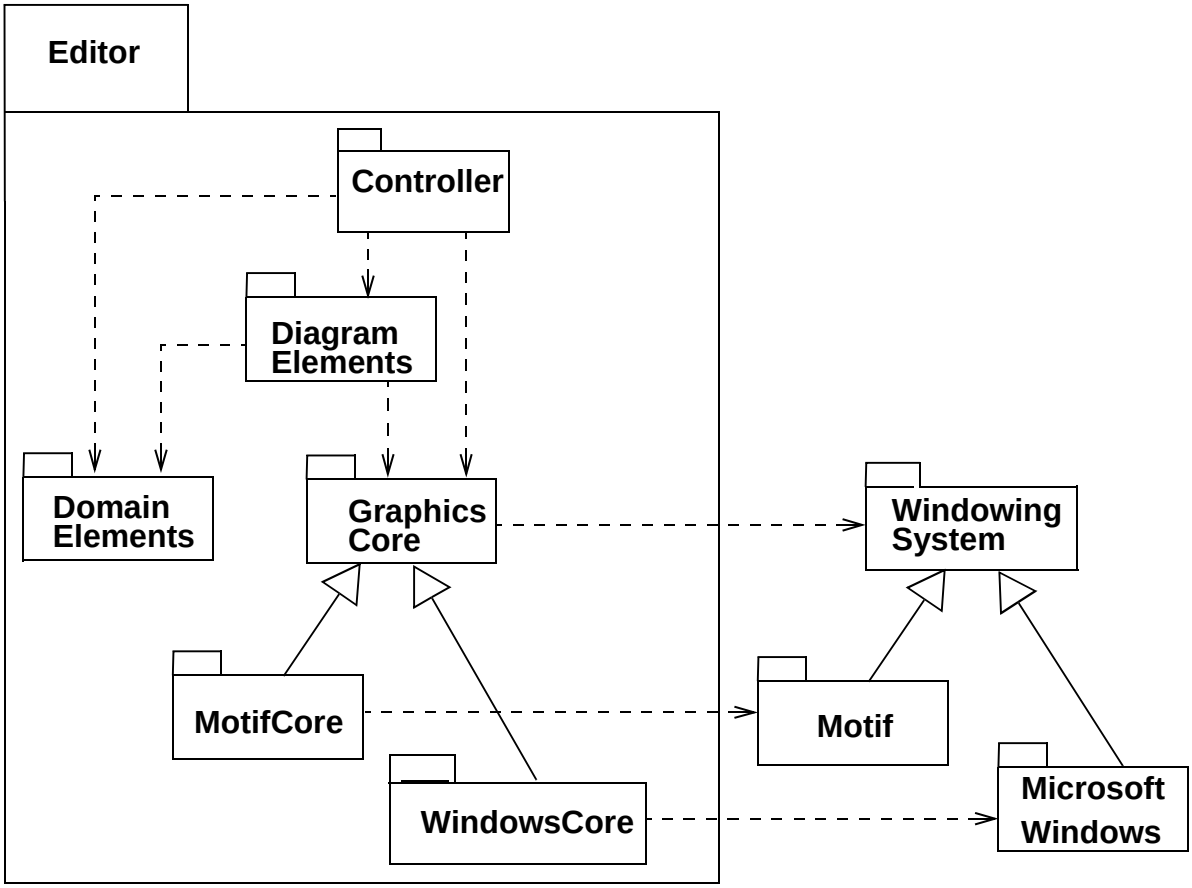
The contents of the package may be shown within the large rectangle.

2.7.2 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.

2.7.3 Example

Figure 3. Packages and their dependencies



3. GENERIC NOTATION

This section describes notation features that apply widely to other notation features.

3.1 TYPE-INSTANCE CORRESPONDENCE

The main purpose of modeling is to prepare generic descriptions that describe many specific particular items. This is often known as the *type-instance dichotomy*. (In this section the words *type* and *instance* are used in a somewhat broader way than the rest of the document.) Many or most of the modeling concepts in UML have this dual character, usually modeled by two paired modeling elements, one of which represents the generic descriptor and the other of which the individual items that it describes. Examples of such pairs in UML include: Class-Object, Association-Link, Parameter-Value, Operation-Call, 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 immediately visually apparent. 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.

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.

3.2 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.

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 Example

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.

3.3 NAME

3.3.1 Semantics

A name is a string that is used to identify a model element 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.

Pathname. A pathname is a series of names linked together by a delimiter. There are various kinds of pathnames described in this document, each in its proper place and with its particular delimiter.

3.3.2 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.

3.3.3 Example

Names:

BankAccount

integrate

controller

abstract

this_is_a_very_long_name_with_underscores

Pathname:

MathPak::Matrices::BandedMatrix.dimension

3.4 LABEL

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

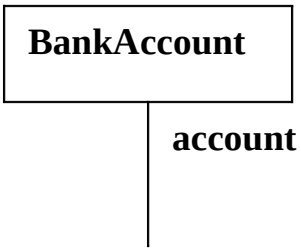
3.4.1 Notation

Visually the attachment is normally 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 can maintain 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.

3.4.2 Example

Figure 4. Attachment by containment and attachment by adjacency



3.5 PROPERTY STRING

A *property string* is a string used to display properties attached to some model element. This is a term for a notation syntax format that may be used for various kinds of model properties (not just tagged values).

3.5.1 Semantics

Note that we use *property* in a general sense to mean any value attached to a model element, including built-in attributes, associations, and tagged values. In this sense it can include indirectly reachable values that can be found starting at a given element.

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 alter their semantics in flexible ways.

3.5.2 Notation

A property string is displayed as a comma-delimited sequence of *property specifications* all inside a pair of braces ({ }).

A *property specification* has the form

keyword = *value*

where *keyword* is the name of a property and *value* is an arbitrary string that denotes its value. If the property is a Boolean flag, then the default value is **true** if the value is omitted. (That is, to specify a value of true you include the keyword; to specify a value of false you omit it 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.

3.5.3 Presentation options

A tool may present property specifications one per line with or without the enclosing braces, provided they are appropriately marked 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.

For a text item that is presented as a string parsable into various fields, certain property values may be included in the string provided an appropriate syntax is defined to distinguish them. For example, certain language-dependent properties might be included in the string presenting an attribute.

3.5.4 Example

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

{ abstract }

3.6 TYPE EXPRESSION

3.6.1 Semantics

Programming languages have a variety of rules for constructing type expressions for declaring variables and parameters. Some languages (such as C++) permit types to be composed into complex expressions without names; others (such as Ada) require composed types to be name before they can be used in declarations. Programming languages have both predefined and user-defined types. UML uses the word *type* in a more general way, roughly corresponding to the concept of *abstract data type* in computer science. In most programming languages the word *type* corresponds most closely to the UML concept of *class,* as the programming-language types include both data structure and operation structure.

UML avoids specifying the rules for constructing type expressions because they are so language-dependent. Rather, UML assumes that *type expression* strings will appear in the declarations of attributes, variables, and parameters, but UML leaves undefined the actual syntax of the type expressions. It is the responsibility of a tool to verify and parse type expressions (if desired, otherwise they can be left as strings). Programming-language type definitions do not explicitly occur in the UML, but type expressions can be generated from UML types and classes (and code generation is a major use of models). At the very least, UML assumes that there is a type expressions corresponding to a single type or class, and that each type expression contains within it references to one or more UML types (including primitive types). Programming-language type expressions appear in the specification of UML attributes and parameters. These appear in the UML as class *TypeExpression,* whose detailed form is unspecified and programming-language-dependent UML *does* assume that a *TypeExpression* contains embedded within it references to actual UML types, but the exact mapping is highly dependent on the syntax of a particular programming language and the UML definition does not attempt to impose a single mapping.

3.6.2 Notation

A type expression is displayed as a string; the syntax of the string is the responsibility of a tool, possibly by reference to an appropriate programming language, possibly with some encoding into fields; the UML does not prescribe the language. At the very least, however, a reference to a class corresponds to a type expression. Some languages, however, may provide a more complicated syntax for implementation types that may be difficult to express simply as class references.

3.6.3 Example

BankAccount

BankAccount * (*) (Person*, int)

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

3.7 STEREOTYPES

3.7.1 Semantics

A stereotype is, in effect, a new class of modeling element that is introduced at modeling time within a model. There are certain restrictions on what they can be: they must be based on certain existing classes in the metamodel and they may extend those classes only in certain predefined ways. They represent a built-in extensibility mechanism of UML.

Stereotypes themselves may have a classification hierarchy. Because the root of such a hierarchy is a metamodel class in the UML metamodel, such classifications are probably best left for experts with a detailed knowledge of UML.

3.7.2 Notation

The general presentation of a stereotype is to place the name of the stereotype within matched *guillemets*, which are the quotation mark symbols used in French and certain other languages, as for example: «foo». (Note that a guillemet looks like a double angle-bracket but it is a single character in most extended fonts. Double angle-brackets may be used as a substitute by the typographically challenged.) The stereotype string is generally placed above or in front of the name of the model element being described. The stereotype 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.

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: 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. Alternately, the entire base model element symbol may be “collapsed” into an icon containing the element name or with the name above or

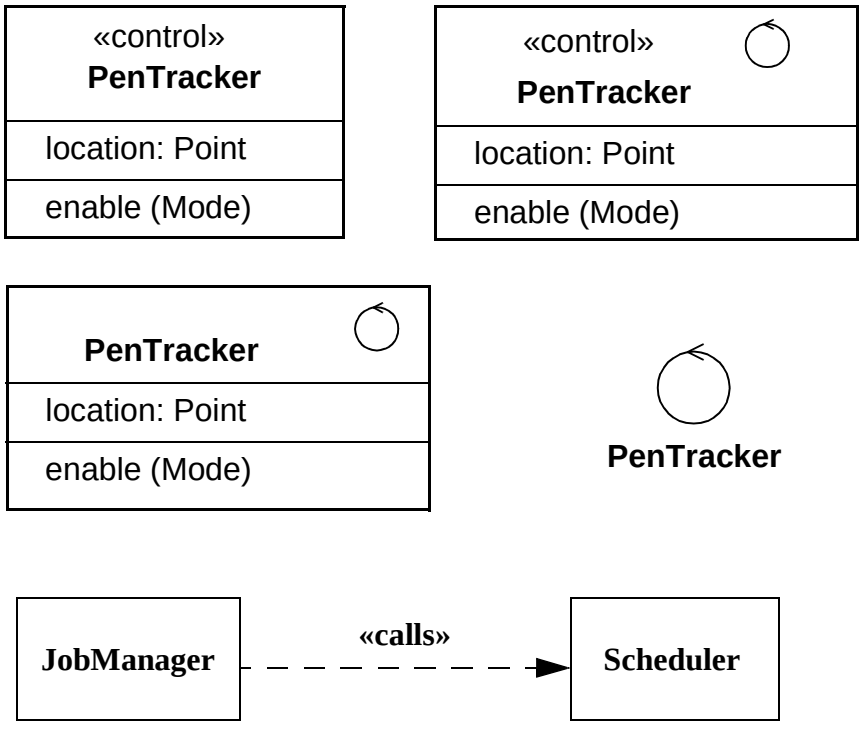
below the icon. Other information contained by the base model element symbol is suppressed. More general forms of icon specification and substitution are conceivable but we leave these to the ingenuity of tool builders, with the warning that excessive use of extensibility capabilities may lead to loss of portability among tools.

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). Users may use graphic markers freely in their personal work (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. Each stereotype is a stereotype «stereotype» of a class (yes, this is a self-referential usage!). Generalization relationships show the extended metamodel hierarchy. Because of the danger of extending the internal metamodel hierarchy, a tool may, but need not, expose this capability on class diagrams.

3.7.3 Example

Figure 5. Varieties of stereotype notation



4. STATIC STRUCTURE DIAGRAMS

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

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

4.1 CLASS DIAGRAM

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

4.1.1 Notation

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

4.2 OBJECT DIAGRAM

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

There is no need for tools to support a separate format for object diagrams. Class diagrams can contain objects, so a class diagram with objects and no classes is an “object diagram.” Collaboration diagrams contain objects, so a collaboration diagram with no messages is an “object diagram.” The phrase is useful, however, to characterize a particular usage achievable in various ways.

4.3 CLASS

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

4.3.1 Semantics

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

4.3.2 Basic notation

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

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

Package-name::*Class-name*

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

4.3.3 Presentation options

Either or both of the attribute and operation compartments may be suppressed. A separator line is not drawn for a missing compartment. If a compartment is suppressed, no inference can be drawn about the presence or absence of elements in it.

Additional compartments may be supplied as a tool extension to show other predefined or user-defined model properties, for example, to show business rules, responsibilities, variations, events handled, and so on. Most compartments are simply lists of strings. More complicated formats are possible, but UML does not specify such formats; they are a tool

responsibility. Appearance of each compartment should preferably be implicit based on its contents. Tools may provide explicit markers if needed.

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

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

4.3.4 Style guidelines

Class name in boldface, centered.

Stereotype name in plain face, within guillemets, centered.

Typically class names begin with an uppercase letter.

Attributes and operations in plain face, left justified.

Typically attribute and operation names begin with a lowercase letter.

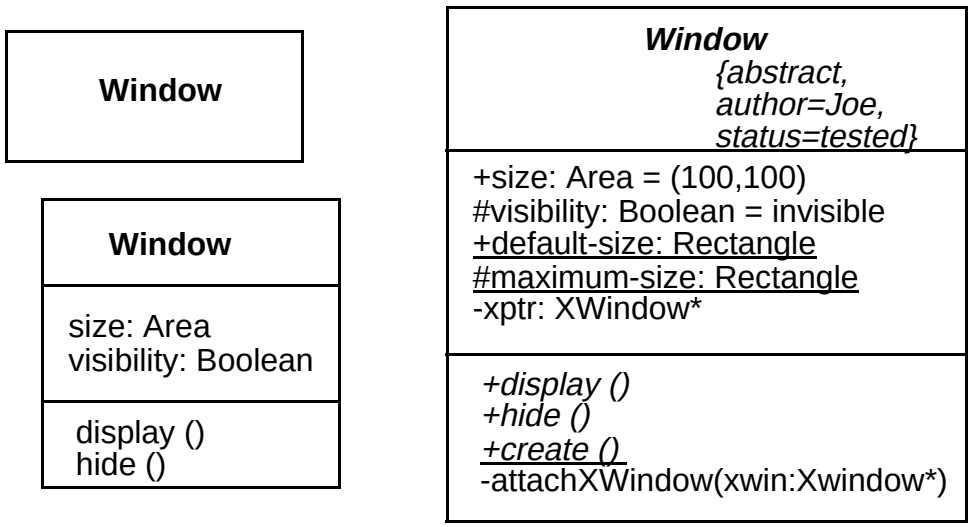
As a tool extension, boldface may be used for marking special list elements, for example, to designate candidate keys in a database design. This might encode some design property modeled as a tagged value, for example.

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

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

4.3.5 Example

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



4.4 NAME COMPARTMENT

4.4.1 Notation

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

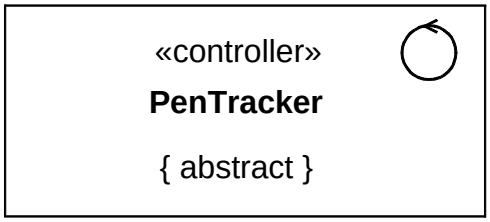
An optional stereotype keyword may be placed above the class name within guillemets, and/or a stereotype icon may be placed in the upper right corner of the compartment.

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

A property list may be placed in braces below the class name. The list may show class-level attributes for which there is no UML notation and it may also show tagged values.

The stereotype and property list are optional.

Figure 7. Name compartment



4.5 LIST COMPARTMENT

4.5.1 Notation

Holds a list of strings, each of which is the encoded representation of an element, such as an attribute or operation. The strings are presented one to a line with overflow to be handled in a tool-dependent manner. In addition to lists of attributes or operations, lists can show other kinds of predefined or user-defined values, such as responsibilities, rules, or modification histories. The manipulation of user-defined lists is tool-dependent.

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

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

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

4.5.2 Presentation options

A tool may present the list elements in a sorted order, in which case the inherent ordering of the elements is not visible. A sort is based on some internal property and does not indicate additional model information. Example sort rules include alphabetical order, ordering by stereotype (such as constructors, destructors, then ordinary methods), ordering by visibility (public, then protected, then private), etc.

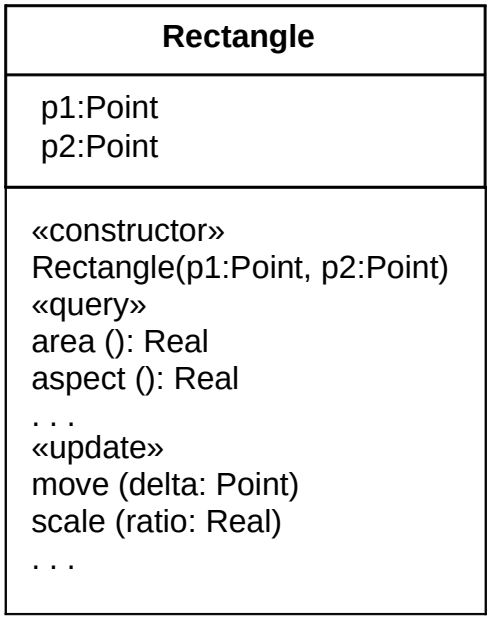
The elements in the list may be filtered according to some selection rule. The specification of selection rules is a tool responsibility. The absence of items from a filtered list indicates that no elements meet the filter criterion, but no inference can be drawn about the presence or absence of elements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

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

Note that attributes may also be shown by composition (see Figure 20).

4.5.3 Example

Figure 8. Stereotype keyword applied to groups of list elements



4.6 TYPE

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

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

4.6.1 Semantics

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

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

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

4.6.2 Notation

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

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

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

4.7 INTERFACES

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

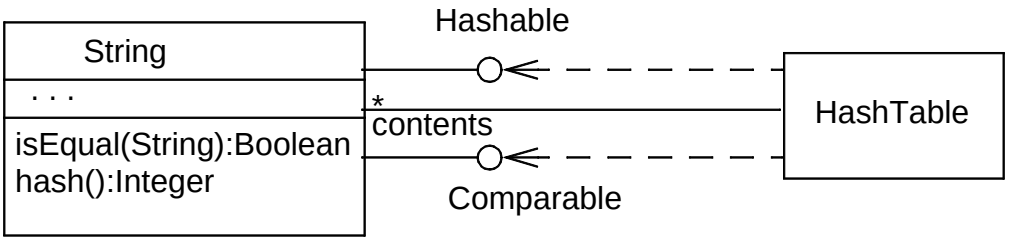
4.7.1 Notation

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

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

4.7.2 Example

Figure 9. Interface notation on class diagram



4.8 PARAMETERIZED CLASS (TEMPLATE)

4.8.1 Semantics

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

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

4.8.2 Notation

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

4.8.3 Presentation options

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

Parameters are restricted attributes, with the syntax

name : *type*

where *name* is an identifier for the parameter with scope inside the template;

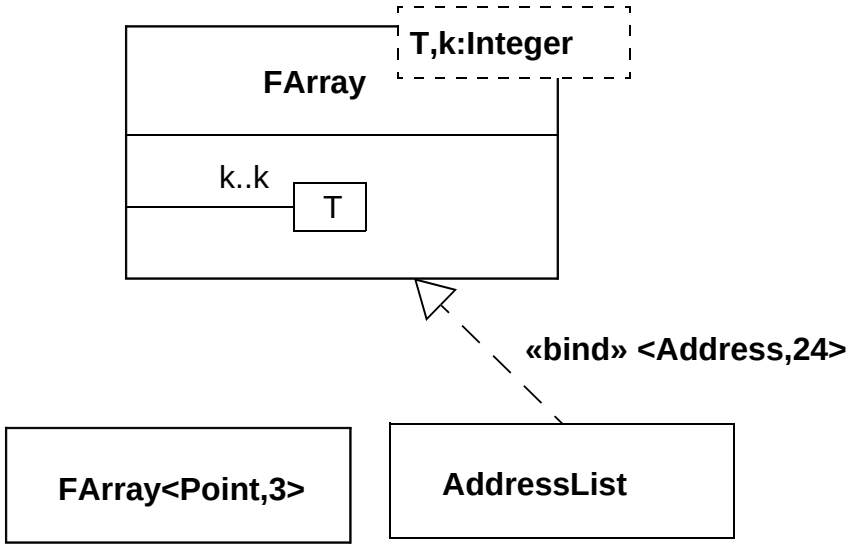
where *type* is a string designating a *TypeExpression* for the parameter.

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

implementation type, such as a class name). Other parameter types (such as Integer) should be explicitly shown.

4.8.4 Example

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



4.9 BOUND ELEMENT

4.9.1 Semantics

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

4.9.2 Notation

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

Template-name ‘<‘ *value-list* ‘>’

where *value-list* is a comma-delimited non-empty list of value expressions;

where *Template-name* is identical to the name of a template.

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

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

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

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

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

4.9.3 Style guidelines

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

4.9.4 Example

See Figure 10.

4.10 UTILITY

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

4.10.1 Semantics

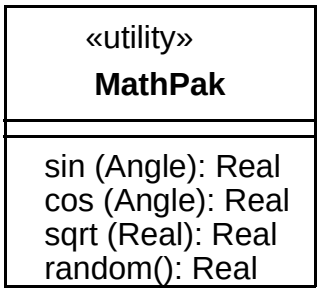
The instance-scope attributes and operations of a utility are interpreted as global attributes and operations. It is inappropriate for a utility to declare class-scope attributes and operations because the instance-scope members are already interpreted as being at class scope.

4.10.2 Notation

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

4.10.3 Example

Figure 11. Notation for utility



4.11 METACLASS

4.11.1 Semantics

A metaclass is a class whose instances are classes.

4.11.2 Notation

Shown as the stereotype «metaclass» of Class.

4.12 CLASS PATHNAMES

4.12.1 Notation

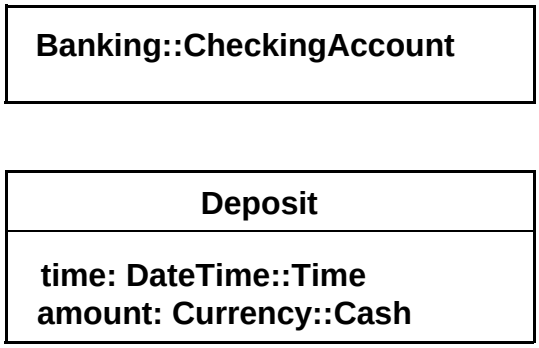
Class symbols (rectangles) serve to define a class and its properties, such as relationships to other classes. A reference to a class in a different package is notated by using a pathname for the class, in the form:

package-name :: *class-name*

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

4.12.2 Example

Figure 12. Pathnames for classes in other packages



4.13 IMPORTING A PACKAGE

4.13.1 Semantics

A class in another package may be referenced. On the package level, the «imports» dependency shows the packages whose classes may be referenced within a given package or packages recursively embedded within it. The target references must be exported by the target package. Note that exports are not recursive; they must be propagated up across each

level of containment. Imports are recursive within inner levels of containment. (See the semantics document for full details.)

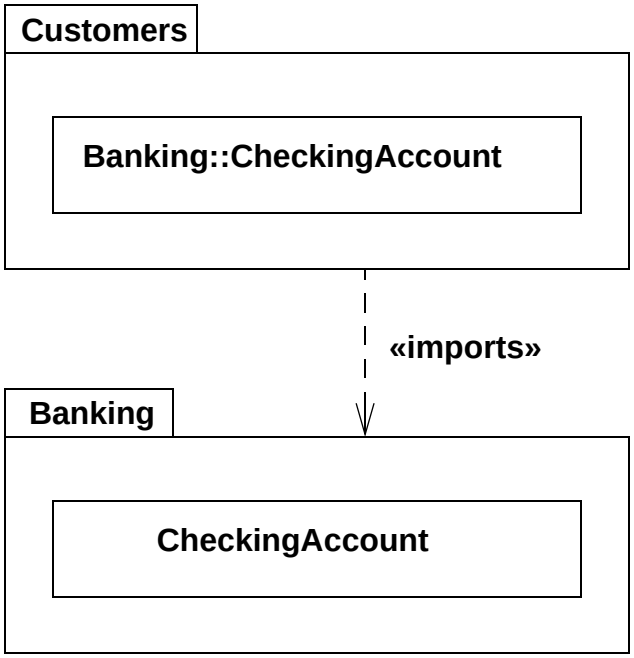
4.13.2 Notation

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

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

4.13.3 Example

Figure 13. Imports dependency among packages



4.14 ATTRIBUTE

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

4.14.1 Semantics

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

The type of an attribute may be complex, such as array[String] of Point. In some specification languages, it may also be expressed as a mapping expressed without a specific commitment to data structure, such as String→Point (where the arrow represents the standard mathematical concept of functional mapping). This form expresses what D'Souza calls "parameterized queries" using the syntax location(String):Point in his Catalysis method. In any case, the details of the attribute type expressions are not specified by UML; they depend on the expression syntax supported by the particular specification or programming language being used.

4.14.2 Notation

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

visibility name : *type-expression* = *initial-value* { *property-string* }

where *visibility* is one of:

+ public visibility

protected visibility

- private visibility

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

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

where *name* is an identifier string;

where *type-expression* is a language-dependent specification of the implementation type of an attribute;

where *initial-value* is a language-dependent expression for the initial value of a newly created object. The initial value is optional (the equal sign is also omitted).

An explicit constructor for a new object may augment or modify the default initial value;

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

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

class-scope-attribute

4.14.3 Presentation options

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

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

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

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

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

Particular attributes within a list may be suppressed (see List Compartment).

4.14.4 Style guidelines

Attribute names typically begin with a lowercase letter.

Attribute names in plain face.

4.14.5 Example

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

#maximum-size: Rectangle
-xptr: XWindow*

4.15 OPERATION

Used to show operations in classes.

4.15.1 Notation

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

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

where *visibility* is one of:

+ public visibility

protected visibility

- private visibility

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

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

where *name* is an identifier string;

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

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

name : *type-expression = default-value*

where *name* is the name of a formal parameter;

where *type-expression* is the (language-dependent) specification of an implementation type;

where *default-value* is an optional value expression for the parameter, expressed in and subject to the limitations of the eventual target language;

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

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

4.15.2 Presentation options

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

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

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

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

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

4.15.3 Style guidelines

Attribute names typically begin with a lowercase letter.

Attribute names in plain face.

An abstract operation may be shown in italics.

4.15.4 Example

Figure 14. Operation list with a variety of operations

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

4.16 ASSOCIATION

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

4.17 BINARY ASSOCIATION

4.17.1 Notation

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

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

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

association name

Designates the (optional) name of the association.

Shown as a name string near the path. The string may have an optional small black solid triangle in it; the point of the triangle indicates the direction in

which to read the name. The name-direction arrow has no semantics significance; it is purely descriptive. The classes in the association are ordered as indicated by the name-direction arrow. (Note that there is no need for a *name direction* property on the association model; the ordering of the classes within the association *is* the name direction. This convention works even with n-ary associations.) A stereotype keyword within guillemets may be placed above or in front of the association name. A property string may be placed after or below the association name.

association class symbol

Designates an association that has class-like properties, such as attributes, operations, and other associations. This is present if and only if the association is an association class.

Shown as a class symbol attached to the association path by a dashed line.

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

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

4.17.2 Presentation options

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

4.17.3 Style guidelines

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

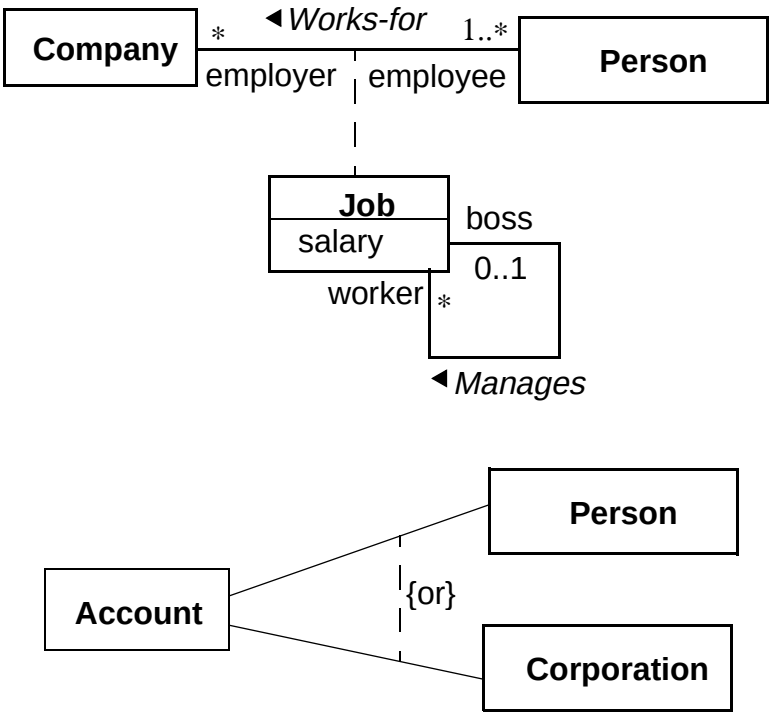
4.17.4 Options

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

the constraint string “{or}” labeling the dashed line. Any instance of the class may only participate in at most one of the associations at one time. (This is simply a particular use of the constraint notation.)

4.17.5 Example

Figure 15. Association notation



4.18 ASSOCIATION ROLE

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

4.18.1 Notation

The path may have graphical adornments at each end where the path connects to the class symbol. The end of an association attached to a class is called a *role*. These adornments indicate properties of the role. The adornments are part of the association symbol, not part

of the class symbol. The role adornments are either attached to the end of the line or near the end of the line and must drag with it. The following kinds of adornments may be attached to a role:

multiplicity – see detail section. Multiplicity may be suppressed on a particular role or for an entire diagram. In an incomplete model the multiplicity may be unspecified in the model itself, in which case it must be suppressed in the notation.

ordering – if the multiplicity is greater than one, then the set of related elements can be ordered or unordered. The default is unordered (they form a set). Various kinds of ordering can be specified as a constraint on the role. The declaration does not specify how the ordering is established or maintained; operations that insert new elements must make provision for specifying their position either implicitly (such as at the end) or explicitly. Possible values include:

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

ordered — the elements are ordered into a list. This generic specification includes all kinds of ordering. This may be specified by a keyword constraint: “{ordered}”.

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

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

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

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

navigability

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

Presentation option 1: Show all arrows. The absence of an arrow indicates navigation is not supported.

Presentation option 2: Suppress all arrows. No inference can be drawn about navigation. This is similar to any situation in which information is suppressed from a view.

Presentation options 3: Suppress arrows for associations with navigability in both directions; show arrows only for associations with one-way navigability. In this case the two-way navigability cannot be distinguished from no-way navigation, but the latter case is normally rare or nonexistent in practice. This is yet another example of a situation in which some information is suppressed from a view.

aggregation indicator

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

If the diamond is filled, then it signifies the strong form of aggregation known as *composition.*

rolename

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

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

4.18.2 Presentation options

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

4.18.3 Style guidelines

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

qualifier

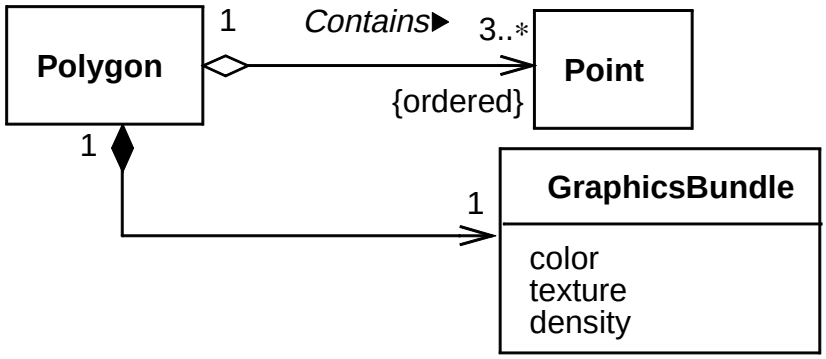
aggregation symbol

navigation arrow

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

4.18.4 Example

Figure 16. Various adornments on association roles



4.19 MULTIPLICITY

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

4.19.1 Notation

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

lower-bound .. upper-bound

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

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

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

4.19.2 Style guidelines

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

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

4.19.3 Example

0..1

1

0..*

*

1..*

1..6

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

4.20 QUALIFIER

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

4.20.1 Notation

A qualifier is shown as a small rectangle attached to the end of an association path between the final path segment and the symbol of the class that it connects to. The qualifier rectangle is part of the association path, not part of the class. The qualifier rectangle drags with the path segments. The qualifier is attached to the source end of the association; that is, an object of the source class together with a value of the qualifier uniquely select a partition in the set of target class objects on the other end of the association.

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

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

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

4.20.2 Presentation options

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

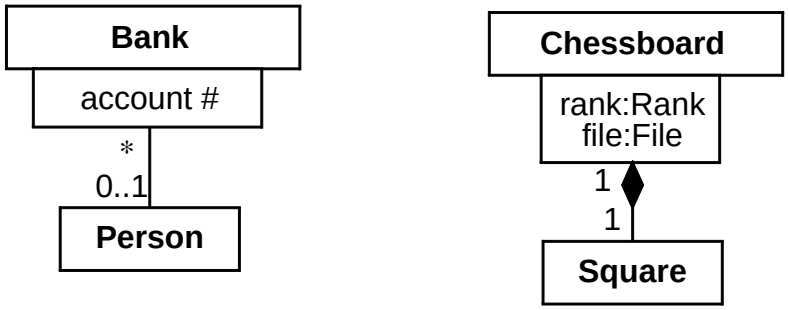
A tool may use a lighter line for qualifier rectangles than for class rectangles to distinguish them clearly.

4.20.3 Style guidelines

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

4.20.4 Example

Figure 17. Qualified associations



4.21 ASSOCIATION CLASS

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

4.21.1 Notation

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

4.21.2 Presentation options

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

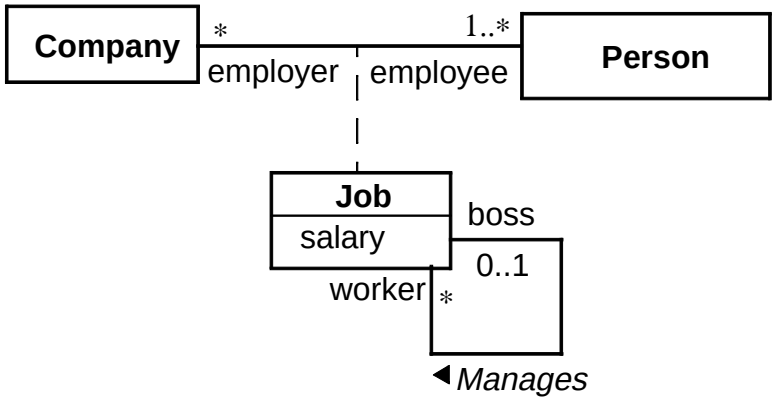
4.21.3 Style guidelines

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

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

4.21.4 Example

Figure 18. Association class



4.22 N-ARY ASSOCIATION

4.22.1 Semantics

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

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

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

4.22.2 Notation

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

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

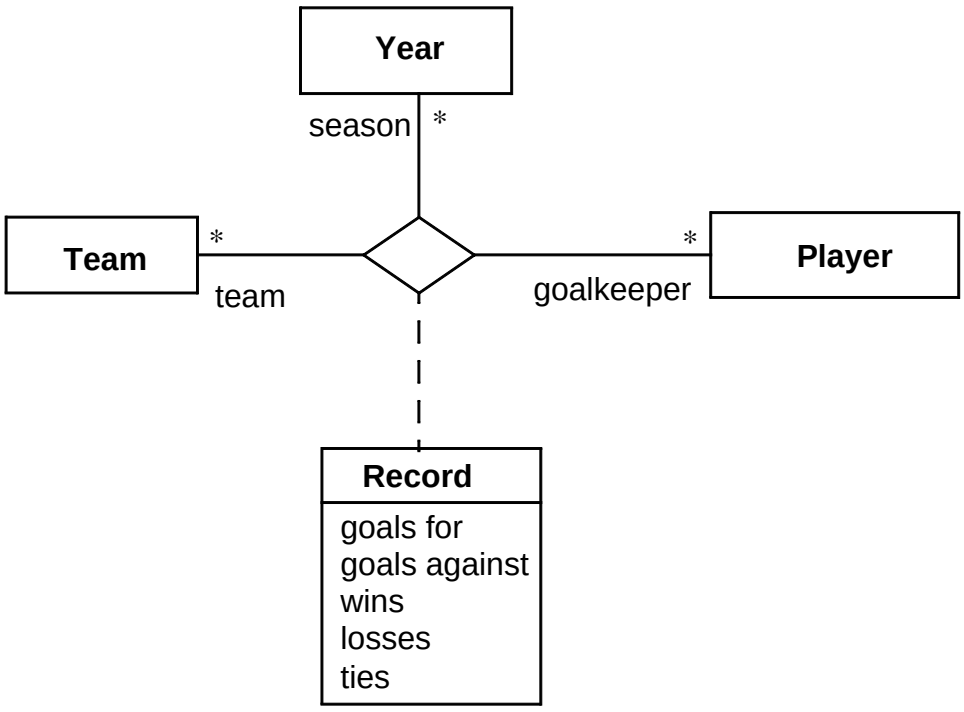
4.22.3 Style guidelines

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

4.22.4 Example

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

Figure 19. Ternary association that is also an association class



4.23 COMPOSITION

Composition is a form of aggregation with strong ownership and coincident lifetime of part with the whole. The multiplicity of the aggregate end may not exceed one (it is unshared). The aggregation is unchangeable (once established the links may not be changed). Parts with multiplicity > 1 may be created after the aggregate itself but once created they live and die with it. Such parts can also be explicitly removed before the death of the aggregate.

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

4.23.1 Semantics

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

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

4.23.2 Notation

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

rolename ':' *classname*

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

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

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

4.23.3 Design guidelines

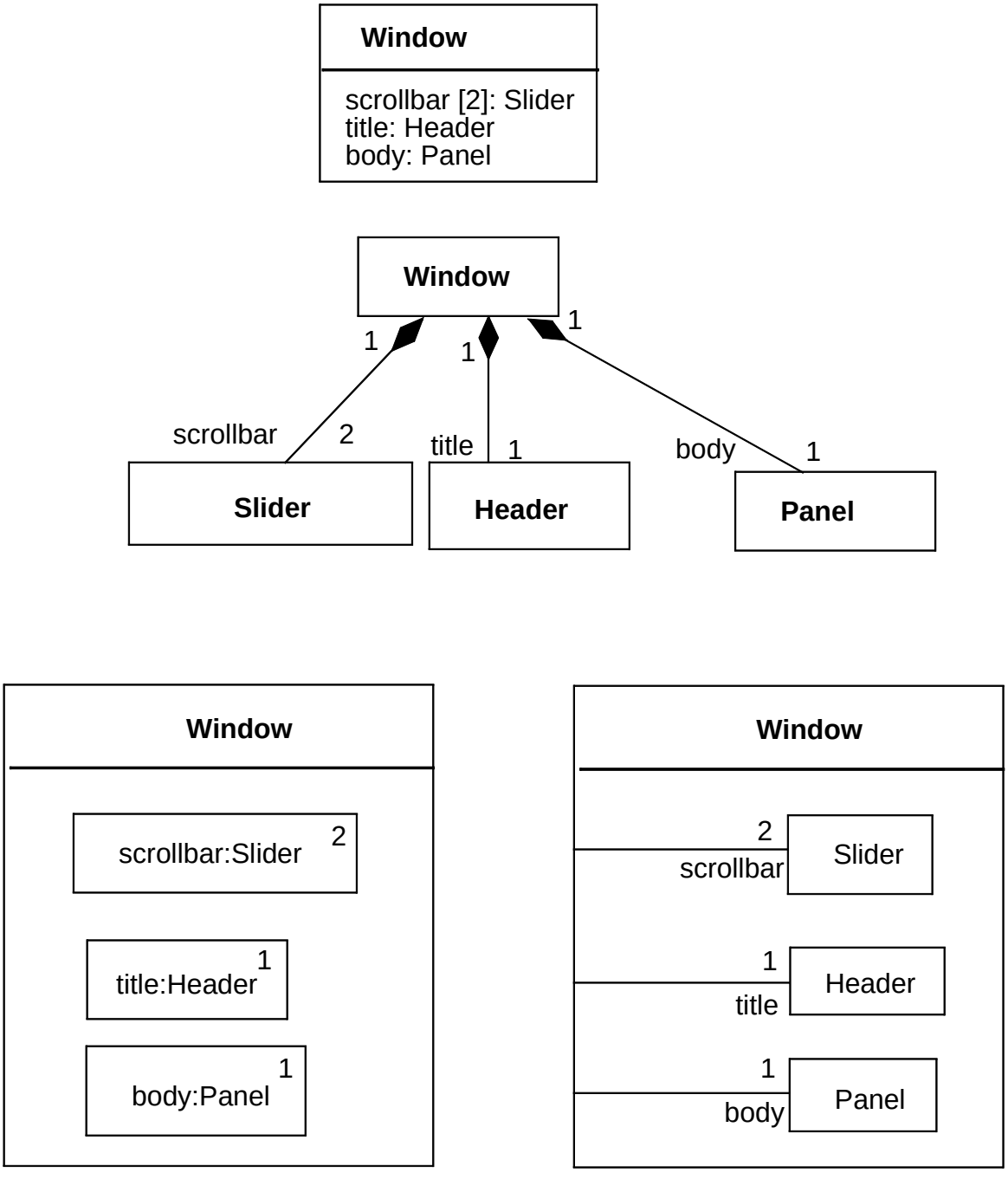
This notation is applicable to “class-like” model elements: classes, types, nodes, processes, etc.

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

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

4.23.4 Example

Figure 20. Different ways to show composition



4.24 GENERALIZATION

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

4.24.1 Notation

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

A generalization path may have a text label in the following format:

discriminator : powertype

where *discriminator* is the name of a partition of the subtypes of the supertype. The subtype is declared to be in the given partition;

where *powertype* is the name of a type whose instances are subtypes of another type, namely the subtypes whose paths bear the powertype name. If a type symbol with the same name appears in the model, it designates the same type; it should be shown with the stereotype «powertype». For example, TreeSpecies is a powertype on the Tree type; consequently instances of TreeSpecies (such as Oak or Birch) are also subtypes of Tree.

Either the discriminator, or the colon and powertype, or both may be omitted.

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

4.24.2 Presentation options

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

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

4.24.3 Details

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

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

overlapping

disjoint

complete

incomplete

4.24.4 Semantics

The following constraints are predefined:

overlapping	A descendent may be descended from more than one of the subclasses.
disjoint	A descendent may not be descended from more than one of the subclasses.
complete	All subclasses have been specified (whether or not shown). No additional subclasses are expected.
incomplete	Some subclasses have been specified but the list is known to be incomplete. There are additional subclasses that are not yet in the model. The is a statement about the model itself. Note that this is not the same as the ellipsis, which states that additional subclasses exist in the model but are not shown on the current diagram.

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

Semantic variation points

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

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

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

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

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

4.24.5 Example

Figure 21. Styles of displaying generalization

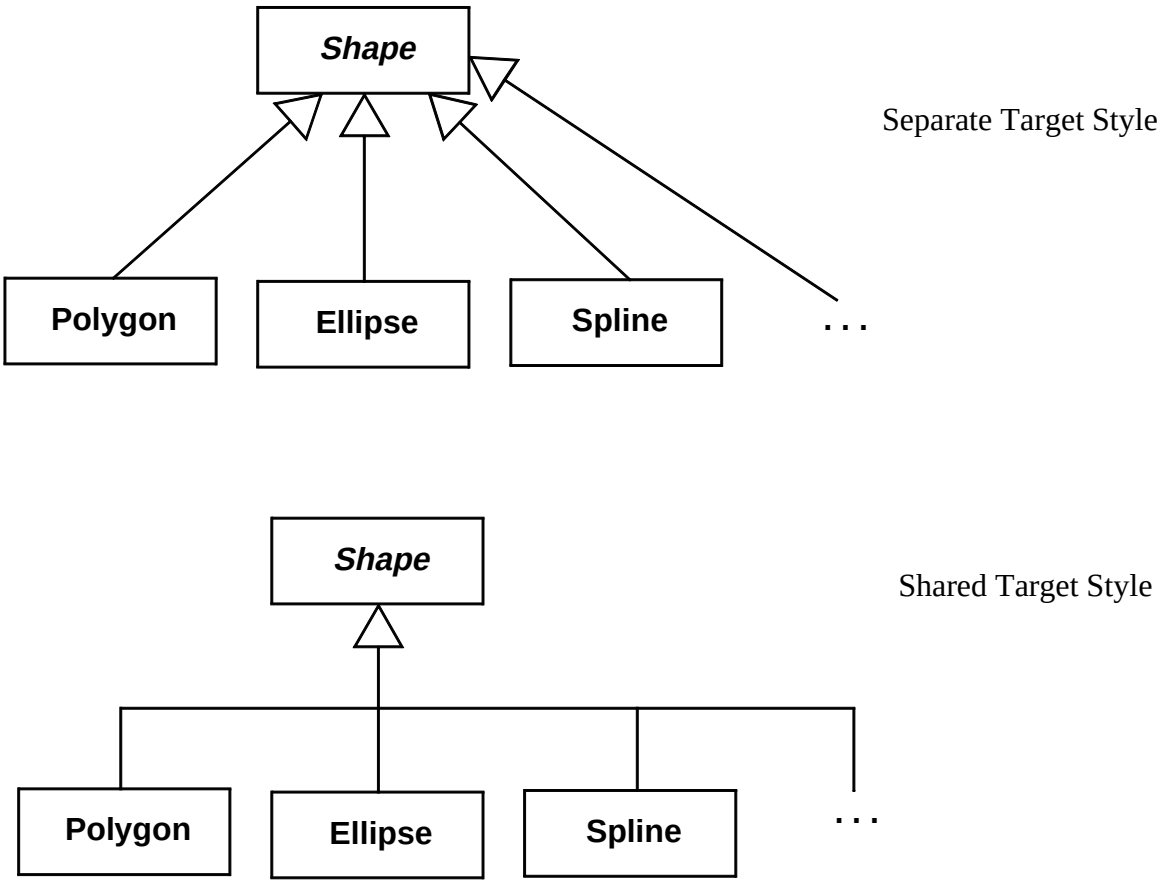


Figure 22. Generalization with discriminators and constraints, separate target style

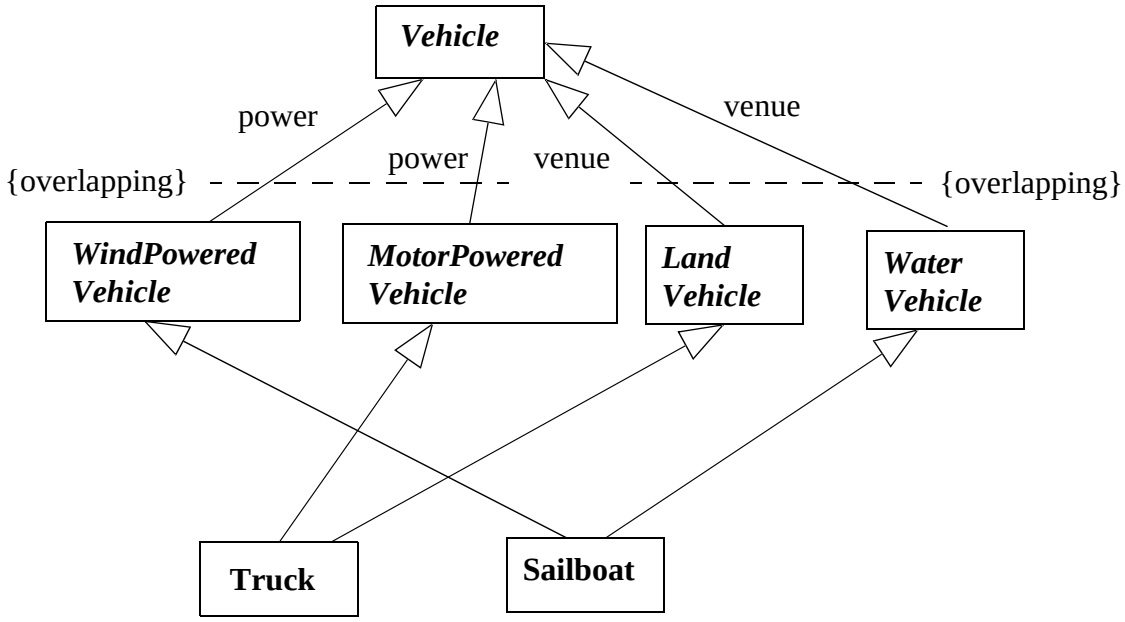
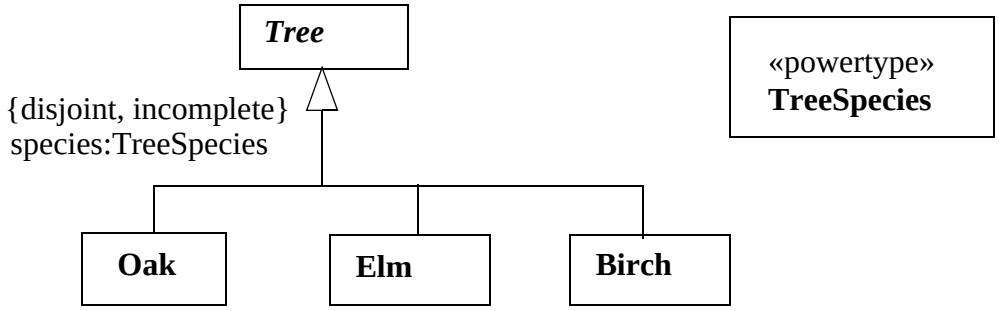


Figure 23. Generalization with power type, shared target style



4.25 DEPENDENCY

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

4.25.1 Notation

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

4.25.2 Presentation options

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

4.25.3 Example

Figure 24. Various dependencies among classes

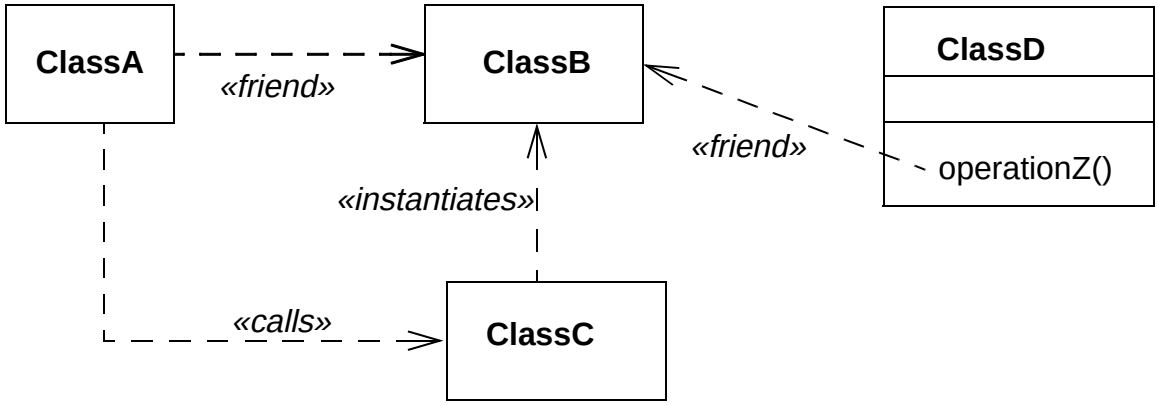
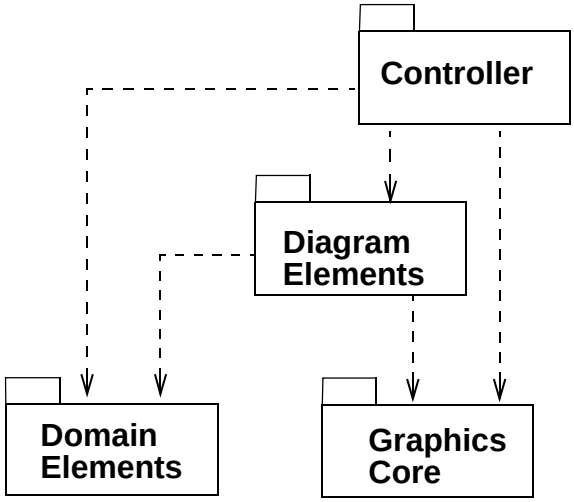


Figure 25. Dependencies among packages



4.26 REFINEMENT RELATIONSHIP

4.26.1 Semantics

The refinement relationship represents the fuller specification of something that has been already specified at a certain level of detail. It is a commitment to certain choices consistent

with the more general specification but not required by it. It is a relationship between two descriptions of the same thing at different levels of abstraction.

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

Relation between a type and a class that realizes it (realization).

Relation between an analysis class and a design class (design trace).

Relation between a high-level construct at a coarse granularity and a lower-level construct at a finer granularity, such as a collaboration at two levels of detail (leveling of detail).

Relation between a construct and its implementation at a lower virtual layer, such as the implementation of a type as a collaboration of lower-level objects (implementation).

Relation between a straightforward implementation of a construct and a more efficient but more obscure implementation that accomplishes the same effect (optimization).

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

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

4.26.2 Notation

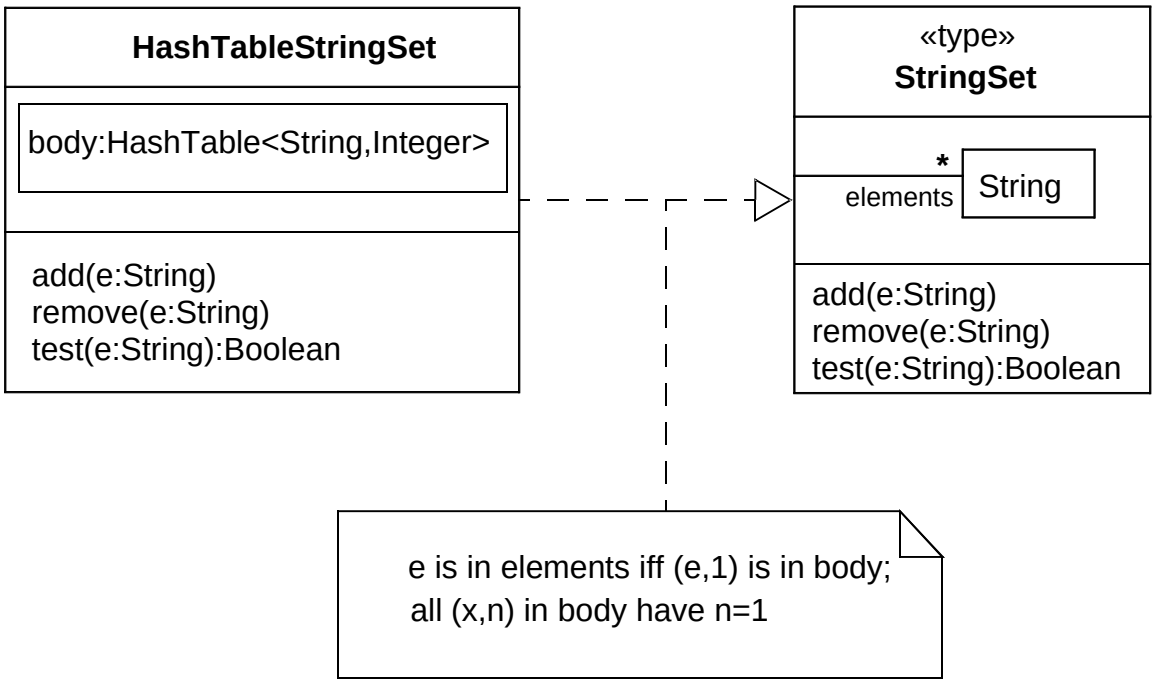
Refinement may be shown as a dashed generalization symbol, that is, a dashed line with a closed hollow triangular arrowhead on the end connected to more general element. A stereotype may be attached to specify a particular kind of refinement. A note may be attached to the line stating the mapping from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

Refinement *within* a given model can be shown as a dependency with the stereotype «refines» or one of its more specific forms, such as «implements». Refinement *between* models may be modeled as an invisible hyperlink supported by a dynamic tool. The refine-

ment relationship may have a mapping attached to it; the mapping will normally be reached via an invisible hyperlink from the relationship path.

4.26.3 Example

Figure 26. Refinement



4.27 DERIVED ELEMENT

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

4.27.1 Notation

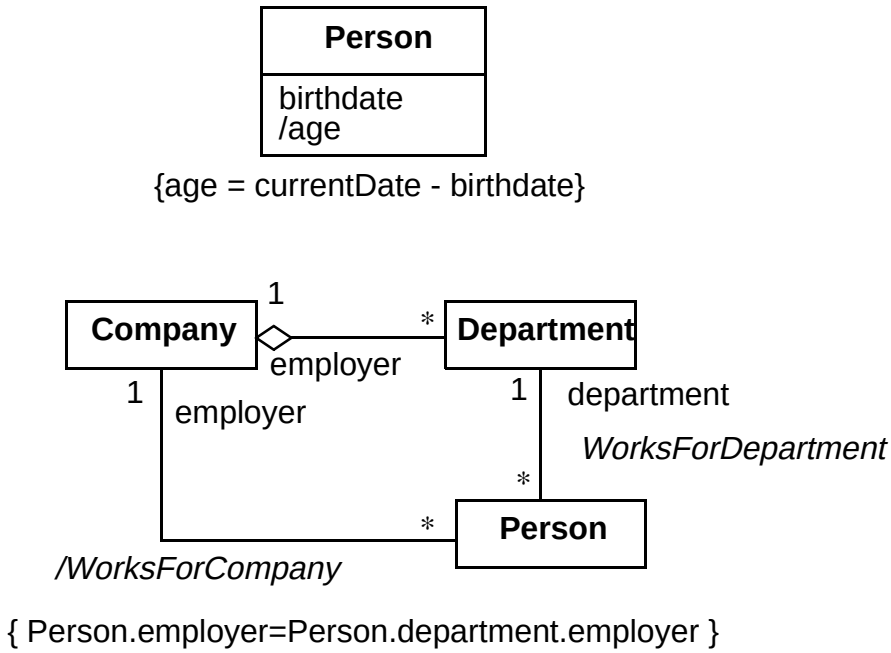
A derived element is shown by placing a slash (/) in front of the name of the derived element, such as an attribute or a rolename.

4.27.2 Style guidelines

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

4.27.3 Example

Figure 27. Derived attribute and derived association



4.28 NAVIGATION EXPRESSION

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

4.28.1 Notation

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

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

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

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

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

4.28.2 Example

flight.pilot.training_hours > flight.plane.minimum_hours

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

5. USE CASE DIAGRAMS

A use case diagram shows the relationship among actors and use cases within a system.

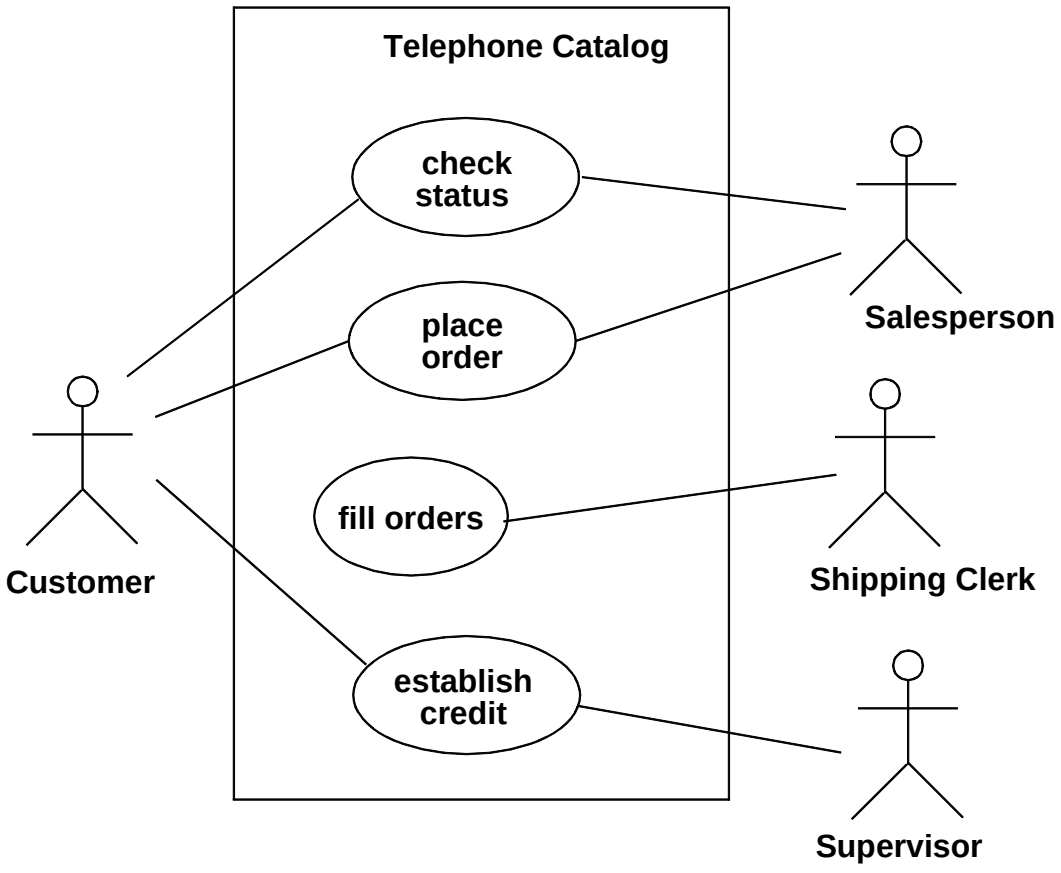
5.1 USE CASE DIAGRAM

5.1.1 Notation

A use case diagram is a graph of actors, a set of use cases enclosed by a system boundary, communication (participation) associations between the actors and the use cases, and generalizations among the use cases.

5.1.2 Example

Figure 28. Use case diagram



5.2 USE CASE

5.2.1 Notation

A use case is shown as an ellipse containing the name of the use case.

5.2.2 Presentation options

The name of the use case may be placed below the ellipse. (This may be viewed as the "stereotype" of the use case, which has the same symbol. According to the rules of stereotypes, the name may be placed above, inside, or below the symbol.)

5.2.3 Style guidelines

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

Use case names should follow capitalization and punctuation guidelines used for behavioral items in the same model.

5.3 ACTOR

5.3.1 Notation

An actor is shown as a class rectangle with the stereotype "actor". The standard stereotype icon for a use case is the "stick man" figure with the name of the actor below the figure.

5.4 USE CASE RELATIONSHIPS

5.4.1 Notation

The following relationships are meaningful within a use case diagram:

Communicates – The participation of an actor in a use case is shown by connecting the actor symbol to the use case symbol by a solid path. The actor is said to "communicate" with the use case.

Extends – An “extends” relationship between use cases is shown by a generalization arrow from the use case providing the extension to the base use case. The arrow is labeled with the stereotype «extends». An extends relationships from use case A to use case B indicates that an instance of use case B may include (subject to specific conditions specified in the extension) the behavior specified by A. Behavior specified by several extenders of a single target use case may occur within a single use case instance.

Uses – A “uses” relationship between use cases is shown by a generalization arrow from the use case doing the use to the use case being used. The arrow is labeled with the stereotype «uses». A uses relationship from use case A to use case B indicates that an instance of the use case A will also include the behavior as specified by B.

The relationship between a use case and its instances (on one hand) are usually shown by an invisible hyperlink. The relationship between a use case and its implementation may be shown as a refinement relationship but may also be shown as an invisible hyperlink. The expectation is that a tool will support the ability to “zoom into” a use case to see its scenarios and/or implementation as an interaction.

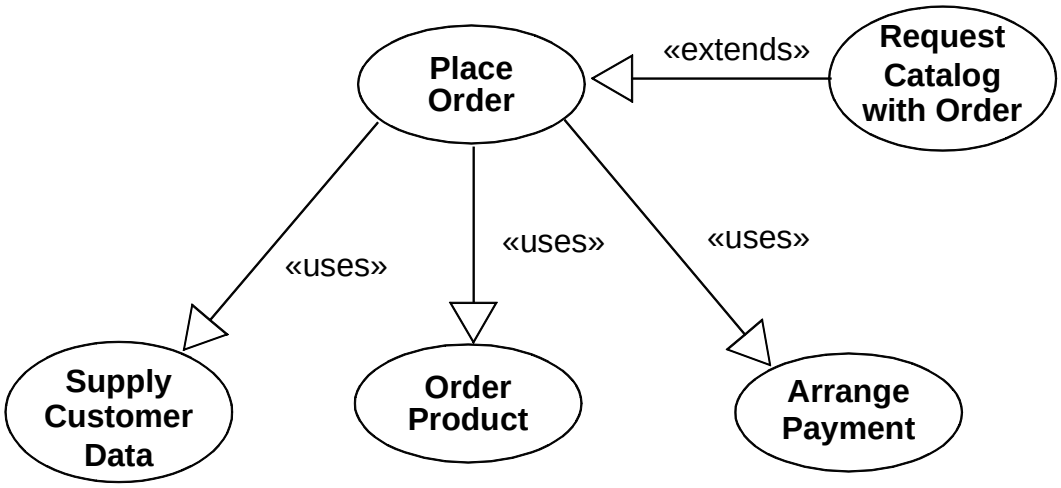
The specification of use case external behavior defines the possible sequences of messages exchanged among the actors and the system. At the use case level, these may be specified by a state machine (including an activity diagram) in which the transitions are labeled by message exchanges. A use case type can be instantiated as a use case instance. Normally at least one scenario should be prepared for each significantly different kind of use case instance. Each scenario shows a sequence of interactions between the actors and the system, with all decisions definite.

The implementation of a use case type can be shown as a collaboration, which is a society of objects and links together with the possible sequences of message flows that produce the effect of the use case. Collaboration diagrams show the sequences of messages among objects that implement the use case.

Both instantiation and implementation of use cases may be shown by invisible hyperlinks from the use case to another diagram.

5.4.2 Example

Figure 29. Use case relationships



6. SEQUENCE DIAGRAMS

A pattern of interaction among objects is shown on an interaction diagram. Interaction diagrams come in two forms based on the same underlying information but each emphasizing a particular aspect of it: sequence diagrams and collaboration diagrams.

A *sequence diagram* shows an interaction arranged in time sequence. In particular, it shows the objects participating in the interaction by their "lifelines" and the messages that they exchanged arranged in time sequence. It does not show the associations among the objects.

Sequence diagrams come in several slightly different formats intended for different purposes.

A sequence diagram can exist in a generic form (describes all the possible sequences) and in an instance form (describes one actual sequence consistent with the generic form). In cases without loops or branches, the two forms are isomorphic.

Sequence diagrams and collaboration diagrams express similar information but show it in different ways. Sequence diagrams show the explicit sequence of messages and are better for real-time specifications and for complex scenarios. Collaboration diagrams show the relationships among objects and are better for understanding all of the effects on a given object and for procedural design.

6.1 SEQUENCE DIAGRAM

6.1.1 Notation

A sequence diagram has two dimensions: the vertical dimension represents time, the horizontal dimension represents different objects. 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 objects. Objects can be grouped into "swimlanes" on a diagram.

(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.)

6.1.2 Presentation options

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 marks, descriptions of actions during an activation, and so one) can be shown either in the margin or near the transitions or activations that they label.

6.1.3 Example

Figure 30. Simple sequence diagram with concurrent objects

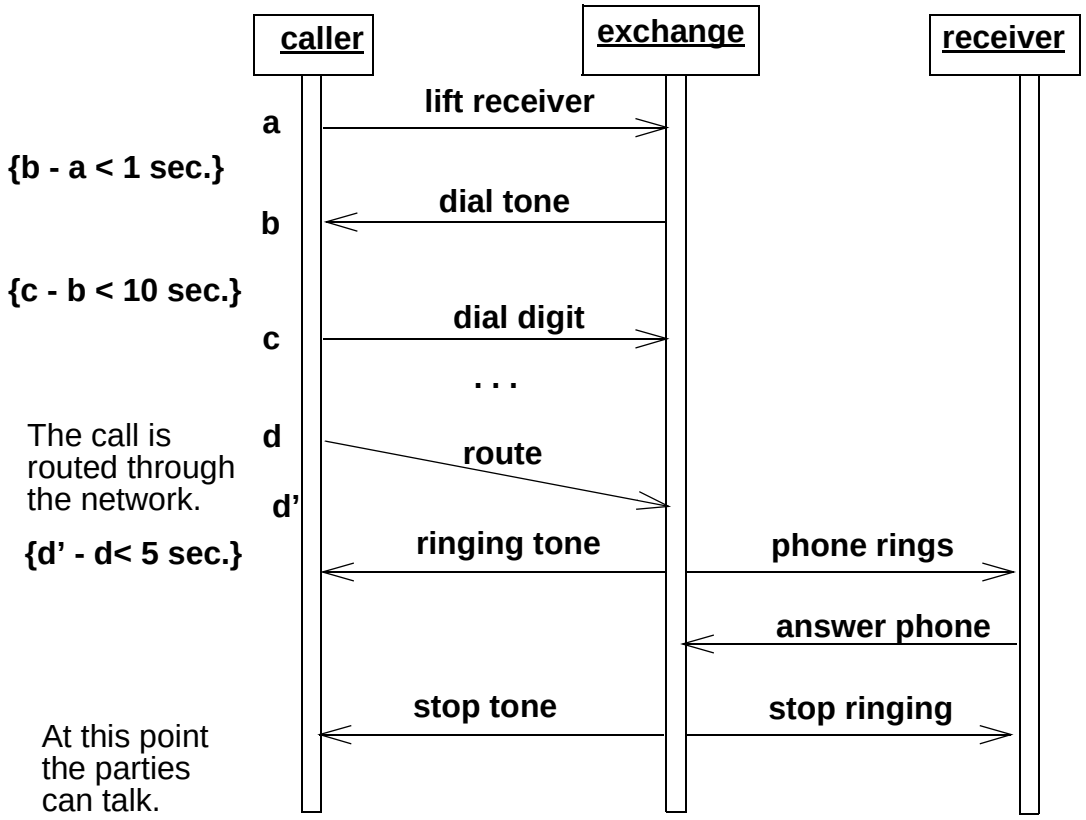
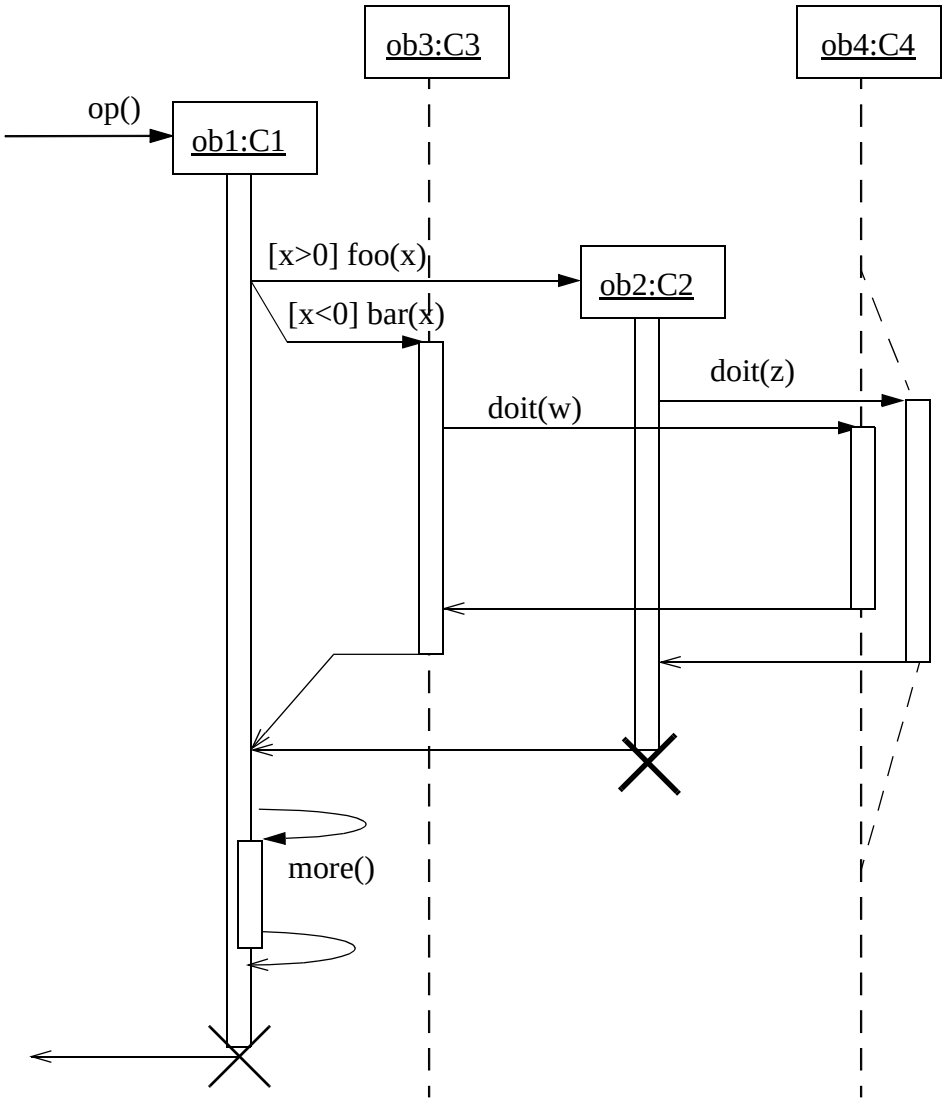


Figure 31. Sequence diagram with focus of control, conditional, recursion, creation, destruction



6.2 OBJECT LIFELINE

6.2.1 Notation

An object is shown as a vertical dashed line called the “lifeline”. The lifeline represents the existence of the object at a particular time. If the object 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 object is created during the diagram, then the message that creates it is drawn with its arrowhead on the object symbol. If the object is destroyed during the diagram, then its destruction is marked by a large “X”, either at the message that causes the destruction or (in the case of self-destruction) at the final return message from the destroyed object. An object that exists when the transaction starts is shown at the top of the diagram (above the first arrow). An object 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 message flow. The lifelines may merge together at some subsequent point.

6.2.2 Example

See Figure 31.

6.3 ACTIVATION

An activation (focus of control) shows the period of time during which an object is performing an action either directly or through a subordinate procedure.

6.3.1 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 action being performed may be labeled in text next to the activation symbol or in the left margin, depending on style; alternately the incoming message may indicate the action, 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 message (the one that initiates the action) and the base of the symbol is at the tail of a return message.

In the case of concurrent objects each with their own threads of control, an activation shows the duration when each object is performing an operation; operations by other objects are not relevant. If the distinction between direct computation and indirect computation (by a nested procedure) is unimportant, the entire lifeline may be shown as an activation.

In the case of procedural code, an activation shows the duration during which a procedure is active in the object or a subordinate procedure is active, possibly in some other object. In other words, all of the active nested procedure activations may be seen at a given time. In the case of a recursive call to an object with an existing activation, the second activation symbol is drawn slightly to the right of the first one, so that they appear to "stack up" visually. (Recursive calls may be nested to an arbitrary depth.)

6.3.2 Example

See Figure 31.

6.4 MESSAGE

A message is a communication between objects that conveys information with the expectation that action will ensue. The receipt of a message is normally considered an event.

6.4.1 Notation

A message is shown as a horizontal solid arrow from the lifeline of one object to the lifeline of another object. In case of a message from an object to itself, the arrow may start and finish on the same object symbol. The arrow is labeled with the name of the message (operation or signal) and its argument values. The arrow may also be labeled with a sequence number to show the sequence of the message in the overall interaction. Sequence numbers are often omitted in sequence diagrams, in which 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. A message may also be labeled with a guard condition.

Variation: Asynchronous. An asynchronous message is drawn with a half-arrowhead, that (one with only one wing instead of two).

Variation: Call. A procedure call is drawn as a full arrowhead. A return is shown as a transverse tick mark (short transverse line) slightly before the end of the line near the target of the return. A call that immediately returns (without any subordinate structure) may be shown as a single line with an arrowhead and a tick mark.

Variation: In a procedural flow of control, the return arrow may be omitted (it is implicit at the end of an activation). For nonprocedural flow of control (including parallel processing and asynchronous messages) returns should be shown explicitly.

Variation: In a concurrent system, a full arrowhead shows the yielding of a thread of control (wait semantics) and a half arrowhead shows the sending of a message without yielding control (no-wait semantics).

Variation: Normally message arrows are drawn horizontally. This indicates the duration required to send the message is “atomic”, that is, it is brief compared to the granularity of the interaction and that nothing else can “happen” during the message transmission. This is the correct assumption within many computers. If the message requires some time to arrive, during which something else can occur (such as a message in the opposite direction) then the message 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 labeled by a guard condition. Depending on whether the guard conditions are mutually exclusive, the construct may represent conditionality or concurrency.

Variation: Iteration. A connected set of messages may be enclosed and marked as an iteration. For a scenario, the iteration indicates that the set of messages 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 messages 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 messages 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.

6.5 TRANSITION TIMES

6.5.1 Notation

A transition instance (such as a message in a sequence diagram or a collaboration diagram or a transition in a state machine) may be given a name. The name represents the time at which a message is sent (example: A). In cases where the delivery of the message in not instantaneous, the time at which the message is received is indicated by the transition name with a prime sign appended (example: A’). The name may be shown in the left margin aligned with the arrow (on a sequence diagram) or near the tail of the message flow arrow (on a collaboration diagram). This name may be used in constraint expressions to designate the time the message was sent. If the message line is slanted, then the primed-name indicates the time at which the message is received.

Constraints may be specified by placing Boolean expressions in braces on the sequence diagram.

6.5.2 Example

See Figure 30.

7. COLLABORATION DIAGRAMS

A collaboration diagram shows an interaction organized around the objects in the interaction and their links to each other. Unlike a sequence diagram, a collaboration diagram shows the relationships among the objects. On the other hand, a collaboration diagram does not show time as a separate dimension, so the sequence of messages and the concurrent threads must be determined using sequence numbers.

7.1 COLLABORATION

7.1.1 Semantics

Behavior is implemented by sets of objects that exchange messages within an overall interaction to accomplish a purpose. To understand the mechanisms used in a design, it is important to see only the objects and the messages involved in accomplishing a purpose or a related set of purposes, projected from the larger system of which they are part. Such a construct is called a *collaboration.*

A collaboration is a modeling unit that describes a set of interactions among types. A collaboration involves two kinds of model constructs: a description of the static structure of the affected objects, including their relevant relationships, attributes, and operations; and a description of the sequences of messages exchanged among the objects to perform work. The first aspect is called the *context* supplied by the collaboration; the second aspect is called the *interactions* supported by the collaboration. Both are needed for a full specification of behavior, but each can be used separately for some design purposes.

A collaboration may be attached to a type, an operation, or a use case to describe their external effects; this is a not an implementation but a specification that describes the changes in the external environment caused by the item. A collaboration may also be attached to a class, to a method (an implemented operation), or to a use case realization (via an «implements» refinement) to describe how they are implemented internally; this collaboration shows the internal constituents of the item and how they interact to achieve the desired external behavior. A collaboration used for implementation is at a finer granularity than one used for specification of the same item.

A parameterized collaboration represents a design construct that can be used repeatedly in different designs. The participants in the collaboration, including the classes, relationships, attributes, and operations can be parameters of the generic collaboration. The parameters are bound to particular model elements in each instantiation of generic collaboration. Such a parameterized collaboration is called a *design pattern.* Whereas most collaborations can be anonymous because they are attached to a named entity, patterns are free standing design constructs and 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.

7.1.2 Notation

The description of a collaboration involves two aspects: the structural description of its participants and the behavioral description of its execution. The two aspects are often described together on a single diagram but at times it is useful to describe the structural and behavioral aspects separately. The description of the structure of objects playing roles in a collaboration and their relationships is called a *context.* The description of the dynamic behavior of the message sequences exchanged among objects to accomplish a specific purpose is called an *interaction.* The remainder of this chapter discusses the notation for contexts and interactions.

7.2 DESIGN PATTERN

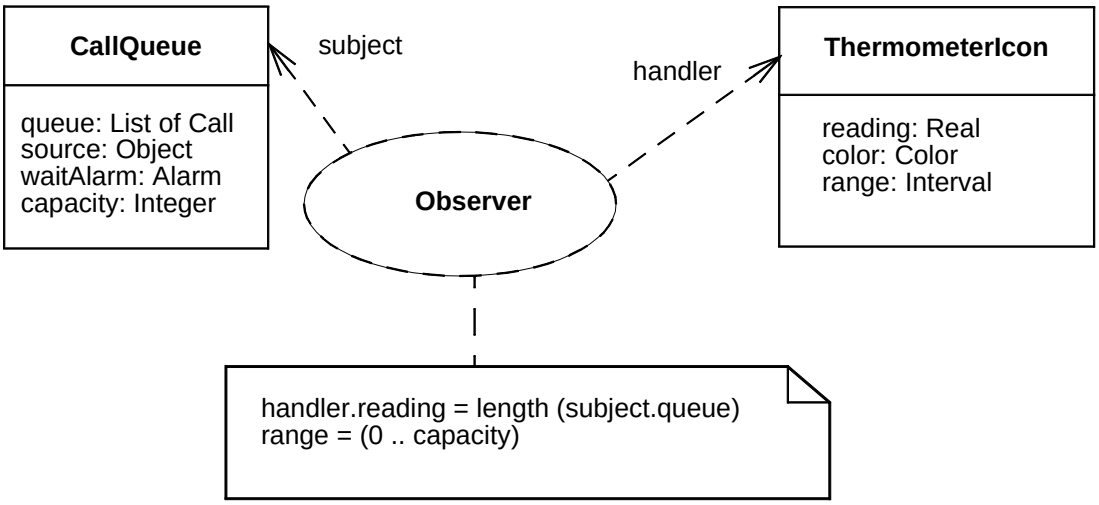
A collaboration can be used to specify the implementation of design constructs. For this purpose it is necessary to specify its context and interactions. It is also possible to view a collaboration as a single entity from the "outside." For example, this could be used to identify the presence of design patterns within a system design.

7.2.1 Notation

A collaboration (as a complete entity representing a design pattern) is shown as a dotted ellipse containing the name of the pattern. A dotted arrow is drawn from the collaboration symbol to each of the objects or classes (depending on whether it appears within an object diagram or a class diagram) that participate in the collaboration. Each arrow 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.

Figure 32. Occurrence of a pattern



7.3 CONTEXT

A context is a view of one or more modeling elements that are related for a particular purpose, such as performing an operation. A context may be a projection from a more complete model, from which details irrelevant to the particular purpose have been suppressed. A context is not itself a modeling element; it is the term for the fragment of the static model that underlies a collaboration.

7.3.1 Semantics

A *context* is a model fragment that shows one or more classes together with their contents, associations, and neighbor classes, plus additional relationships and classes as needed to define operations on the class. Any classes not shown are not affected by operations on the class (or by a particular operation).

Since each context shows a local view of the entire system, classes may appear slightly differently in different contexts. Each context may show the attributes and relationships important to its purposes and suppress the others. Ultimately each context must be a projection from a consistent model of the entire system, but within a single local view the scope of each element in the context is not specified.

7.3.2 Notation

The context of a collaboration is shown as an object diagram—a graph of objects and links. The names of the objects represent their roles within the collaboration. A collaboration is a prototype, so the objects in its context are also prototypes; in each execution of the collaboration they are bound to actual objects. There are several ways to show the diagram:

Methods. If the collaboration shows the implementation of an operation, then it is usually drawn as a separate collaboration diagram including both context and message flow. The context for the operation includes the target object of the operation and any other objects that it calls on, directly or indirectly, to implement the operation. The context includes the objects present before the operation, the objects present after the operation (these may be the same or mostly the same as the ones before), and objects that exist only during the operation; these may be marked as «new», «destroyed», and «transient». Only objects involved in the operation implementation need be shown. To show the execution of the operation, message flows are superimposed on the context objects (see Section 7.10).

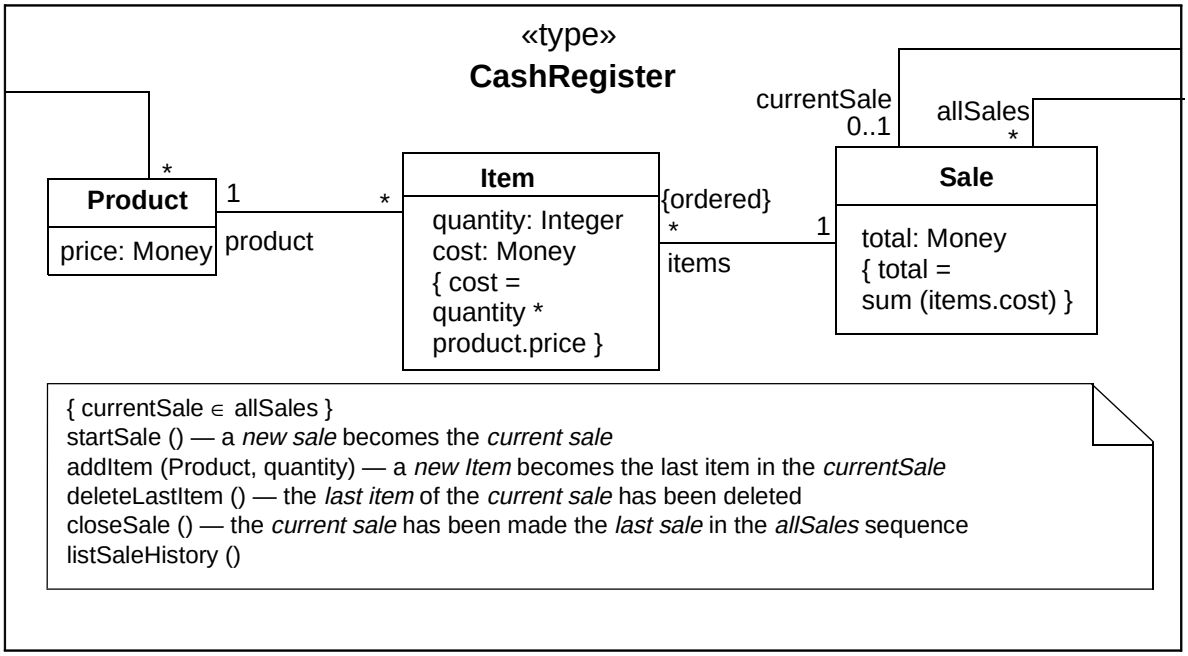
Types and operations. If the collaboration shows the definition of a type as a whole, then its context is an object diagram that shows the constituents of the type together with related objects affected by operations on the type. Because a type does not have operation implementation, there are no message flows. Instead the collaboration is a declarative specification of the behavior of the type. It may contain a list of invariants, that is, constraints that remain true in spite of the performance of operations on the type. For each operation, it may contain a declarative specification of the operation in terms of the values of the objects in the type context: the value of each object after performing an operation is specified as a function of the values of the set of objects before performing the operation (a “before-after” specification). Such specifications *may* be shown on the diagram, but often they are lengthy and are stored in the background, accessible by hidden hyperlinks.

In both cases the usual assumption is that objects and classes not shown on the context are not affected by the operation. (It is not always safe to assume that all of the objects on a context diagram are used by the operation, however.)

Different contexts may be devised for the same type for different purposes. Each context may have a somewhat different set of attributes, operators, and related objects that are relevant to each purpose. Inasmuch as actual operations often fall into related groups, each context might specify a consistent view shared by several operations that is somewhat different from the view needed by other operations on the same type. Similarly, the model of types in a business organization can often be divided into several contexts, each from the point of view of a particular stakeholder.

7.3.3 Example

Figure 33. Type definition using context and before-after conditions



7.4 INTERACTIONS

A collaboration of objects interacts to accomplish a purpose (such as performing an operation) by exchanging messages. The messages may include both signals and calls, as well as more implicit interaction through conditions and time events. A specific pattern of message exchange to accomplish a specific purpose is called an *interaction.*

7.4.1 Semantics

An *interaction* is a behavioral specification that comprises a sequence of message exchanges among a set of objects within a context to accomplish a specific purpose, such as the implementation of an operation. To specify an interaction, it is first necessary to specify a context, that is, the establish the objects 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.

7.4.2 Notation

Interactions are shown as sequence diagrams or as collaboration diagrams. Both diagram formats show the execution of collaborations. However, sequence diagrams only show the participating objects and do not show their relationships to other objects or their attributes, 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 message and explicit representation of method activations. Sequence diagrams are described in Chapter 6. Collaboration diagrams show the full context of an interaction, including the objects and their relationships relevant to a particular interaction, so they are often better for design purposes. Collaboration diagrams are described in the following sections.

7.5 COLLABORATION DIAGRAM

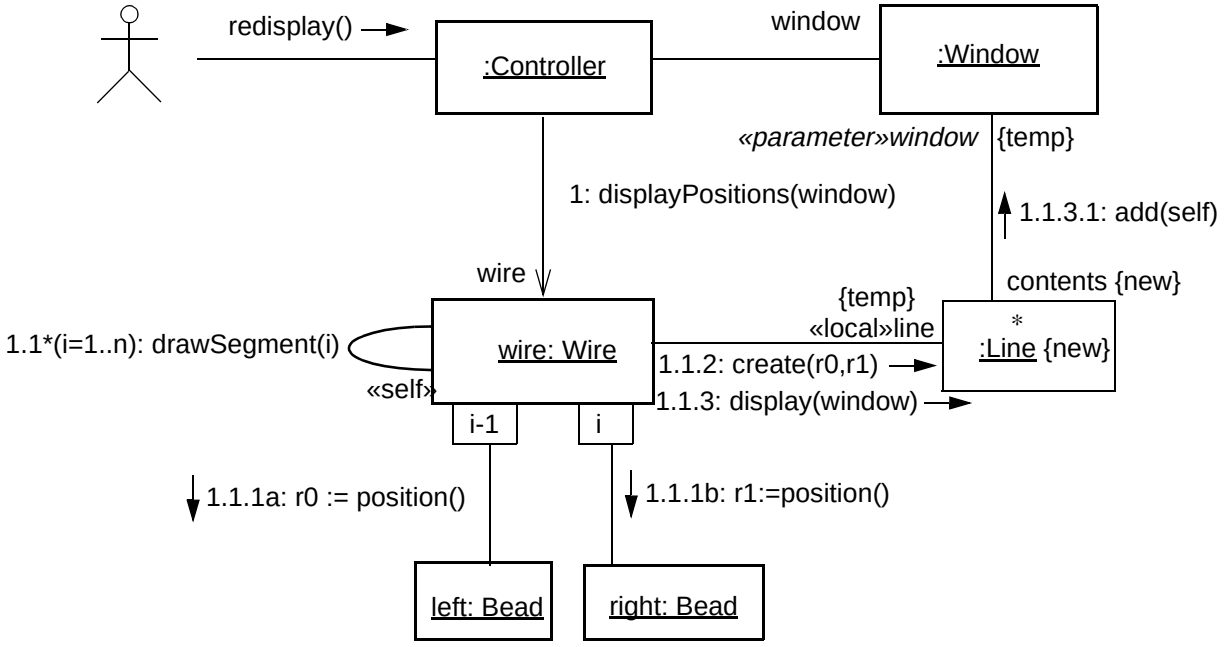
7.5.1 Notation

A collaboration diagram is a context, that is, a graph of objects and links with message flows attached to its links. The context of the diagram shows the objects relevant to the performance of an operation, including objects indirectly affected or accessed during the operation. The context for an operation includes its arguments and local variables created during its execution as well as ordinary associations. Objects created during the execution may be designated as «new»; objects destroyed during the execution maybe designated as «destroyed»; objects created during the execution and then destroyed may be designated as «transient».

The invoker of an interaction may be shown on a collaboration diagram as an actor symbol. The internal messages that implement an operation are number starting with number 1. For a procedural flow of control the subsequent message numbers are nested in accordance with call nesting. For a nonprocedural sequence of messages exchanged among concurrent objects all the sequence numbers are at the same level (that is, they are not nested).

7.5.2 Example

Figure 34. Collaboration diagram



7.6 OBJECT

7.6.1 Notation

(The object notation is derived from the class notation by underlining instance-level elements, as explained in the general comments in Section 3.1.)

An object is 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.

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 but it is expected that a tool will specify such a syntax using some programming language.

7.6.2 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 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. This is a good opportunity for the use of animation by a tool (the values would change dynamically). An alternate notation is to show the same object more than once with a «becomes» relationship between them.

7.6.3 Style guidelines

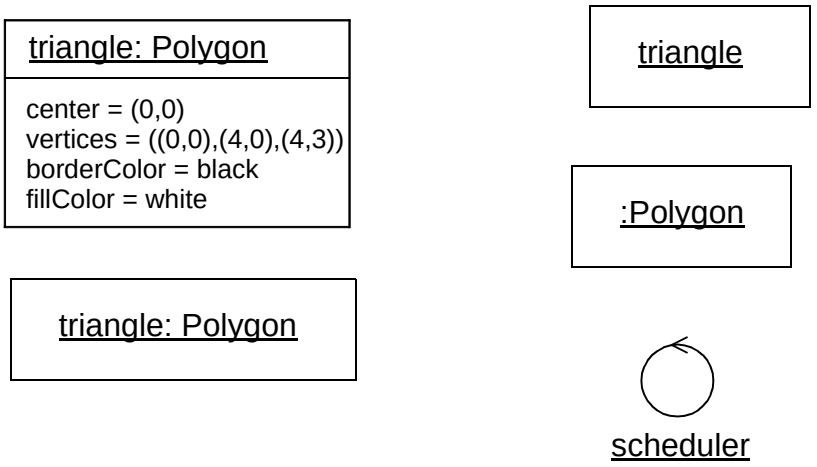
Objects may be shown on (static) object diagrams as well as (dynamic) collaboration diagrams and sequence diagrams. Static object diagrams serve mainly to show examples of data structures.

7.6.4 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 permissible, although the UML does not currently support those semantics.

7.6.5 Example

Figure 35. Objects



7.7 COMPOSITE OBJECT

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.

7.7.1 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 attributes values may be regarded as the parts of a composite object, so there is not such a difference.)

7.7.2 Presentation options

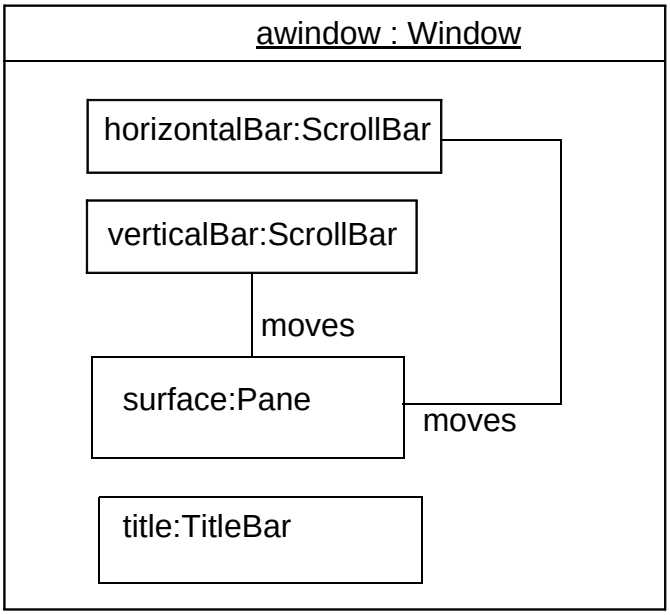
The contents of a composite object may be suppressed and messages to the parts may be subsumed to the composite object itself. Internal messages among the parts may be suppressed in such a high-level view.

7.7.3 Style guidelines

Messages are normally shown either to the composite or to its parts on one diagram, but they are not normally mixed on one diagram. In other words, the composite may be viewed at two different levels of abstraction, but it is desirable to only use one level at a time.

7.7.4 Example

Figure 36. Composite object



7.8 ACTIVE OBJECT

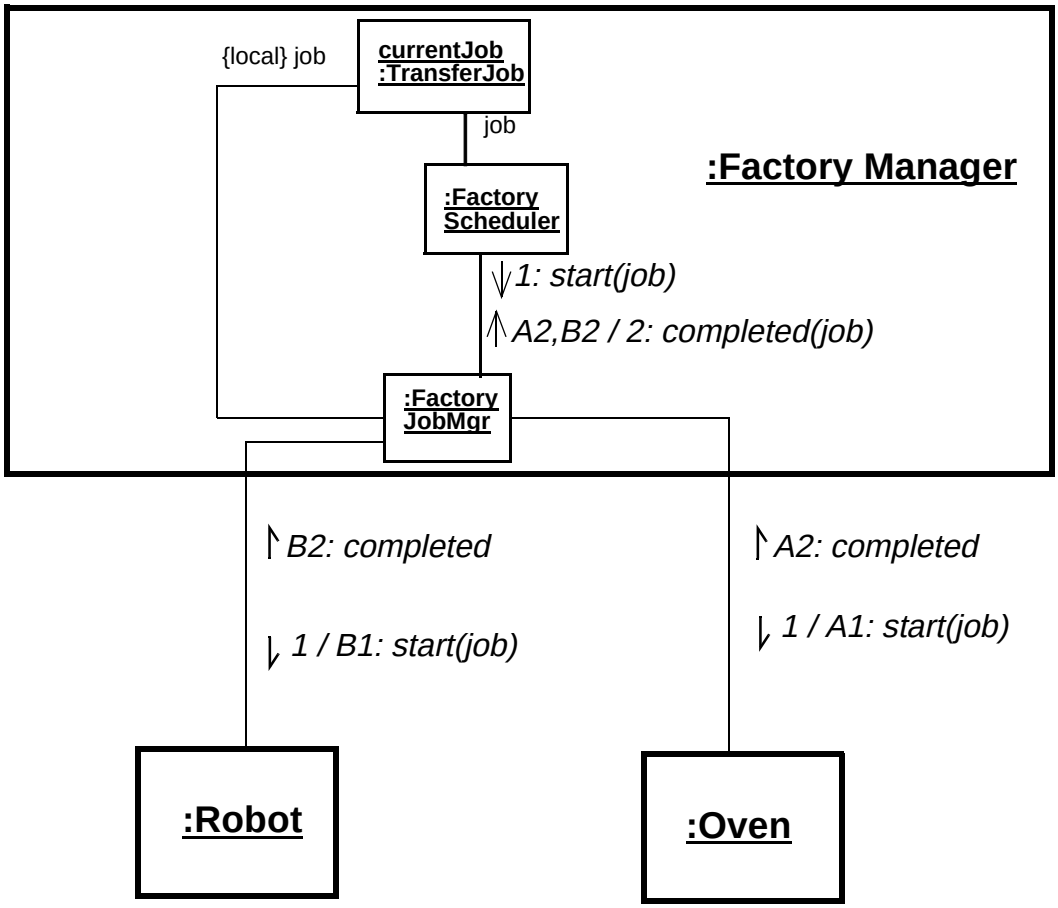
An active object is one that owns a thread of control and may initiate control activity. A passive object is one that holds data but that does not initiate control. However, a passive object may send messages in the process of processing a request that it has received.

7.8.1 Notation

An active object is an object that owns a thread of control. It is shown as an object with a heavy border. Frequently active objects are shown as composites with embedded parts.

7.8.2 Example

Figure 37. Composite active object



7.9 LINKS

A link is a a tuple (list) of object references. In the most normal case, it is a pairing of object references. It is an instance of an association.

7.9.1 Notation

A binary link is shown as a path between two objects. In the case of a reflexive association, it may involve a loop with a single object. See Association 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 objects 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 roles.

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

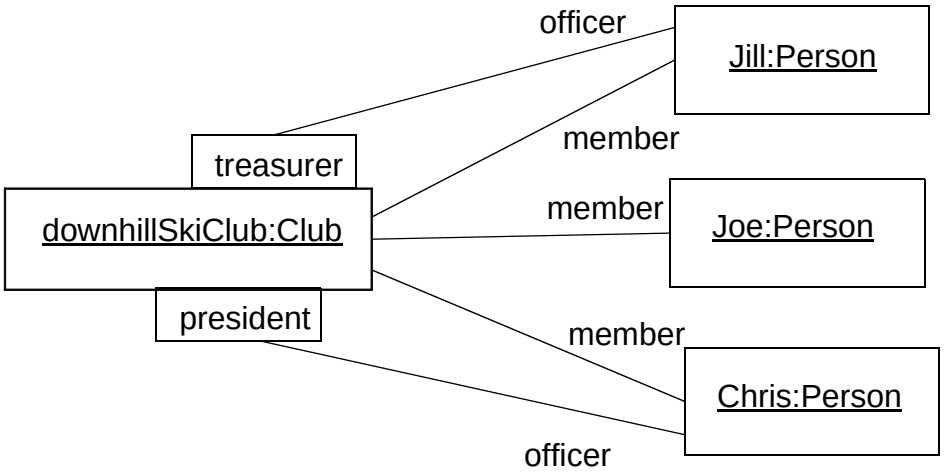
Implementation stereotypes. A stereotype may be attached to the link role to indicate various kinds of implementation. The following stereotypes may be used:

«association»	association (default, unnecessary to specify except for emphasis)
«parameter»	procedure parameter
«local»	local variable of a procedure
«global»	global variable
«self»	self link (the ability of an object to send a message to itself)

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

7.9.2 Example

Figure 38. Links



7.10 MESSAGE FLOWS

A *message flow* in the notation that shows the sending of a message from one object to another. The implementation of a message may take various forms, such as a procedure call, the sending of a signal between active threads, the explicit raising of events, and so on.

7.10.1 Notation

A message flow is shown as a labeled arrow placed near a link. The meaning is that the link is used to transport or otherwise implement the delivery of the message to the target object. The arrow points along the link in the direction of the target object (the one that receives the message).

Control flow type

The following arrowhead variations may be used to show different kinds of messages:

filled solid arrowhead
: procedure call or other nested flow of control. The entire nested sequence is completed before the outer level sequence resumes. May be used with ordinary procedure calls. May also be used with concurrently active objects when one of them sends a signal and waits for a nested sequence of behavior to complete.

stick arrowhead
: Flat flow of control. Each arrow shows the progression to the next step in sequence without. May be combined with procedure calls, or procedure calls can be flattened into a linear sequence.

half stick arrowhead
: asynchronous flow of control. Used instead of the stick arrowhead to explicitly show an asynchronous message between two objects.

other variations
: other kinds of control may be shown, such as “balking” or “time-out”, but these are treated as extensions to the UML core

Message label. The label has the following syntax:

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

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

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 flow is not enabled until all of the message flows whose sequence numbers are listed have occurred (a thread can go beyond the required message flow and the guard remains satisfied). Therefore the guard condition represents a synchronization of threads.

Sequence expression. The sequence-expression is a dot-separated list of sequence-terms followed by a colon (‘:’). 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 that 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 tentative notation for this is to follow the star by a double vertical line (for parallelism): *||.

Signature. A signature is a string that indicates the name, the arguments, and the return value of an operation, message, or signal. These have the following properties:

Return-value. This is a list of names that designates the values returned by the message 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 event raised in the target object (which is often the event of requesting an operation to be performed). It may be implemented in various ways, *one* of which is an operation call. If it is implemented as a procedure call, then this is the name of the operation and the operation must be defined on the class of the receiver or inherited by it. In other cases it may be the name of an event that is raised on the receiving object. In normal practice with procedural overloading, both the message name and the argument list types are required to identify a particular operation.

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 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 object (that is, attributes of it and links from it and paths reachable from them).

7.10.2 Presentation options

Instead of text expressions for arguments and return values, data tokens may be shown near a message. 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.

7.10.3 Example

See Figure 34 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

[x < 0] 4: invert (x, color)conditional message

A3,B4/ C3.1*: update ()synchronization with other threads, iteration

7.11 CREATION/DESTRUCTION MARKERS

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

7.11.1 Notation

An object or link that is created during an interaction has the keyword *new* as a constraint. An object or link that is destroyed during an interaction has the keyword *destroyed* as a constraint. The keyword may be used even if the element has no name. Both keywords may be used together, but the keyword *transient* may be used in place of *new destroyed.*

7.11.2 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.

7.11.3 Example

See Figure 34.

8. STATE DIAGRAM

A state diagram shows the sequences of states that an object or an interaction goes through during its life in response to received stimuli, together with its responses and actions.

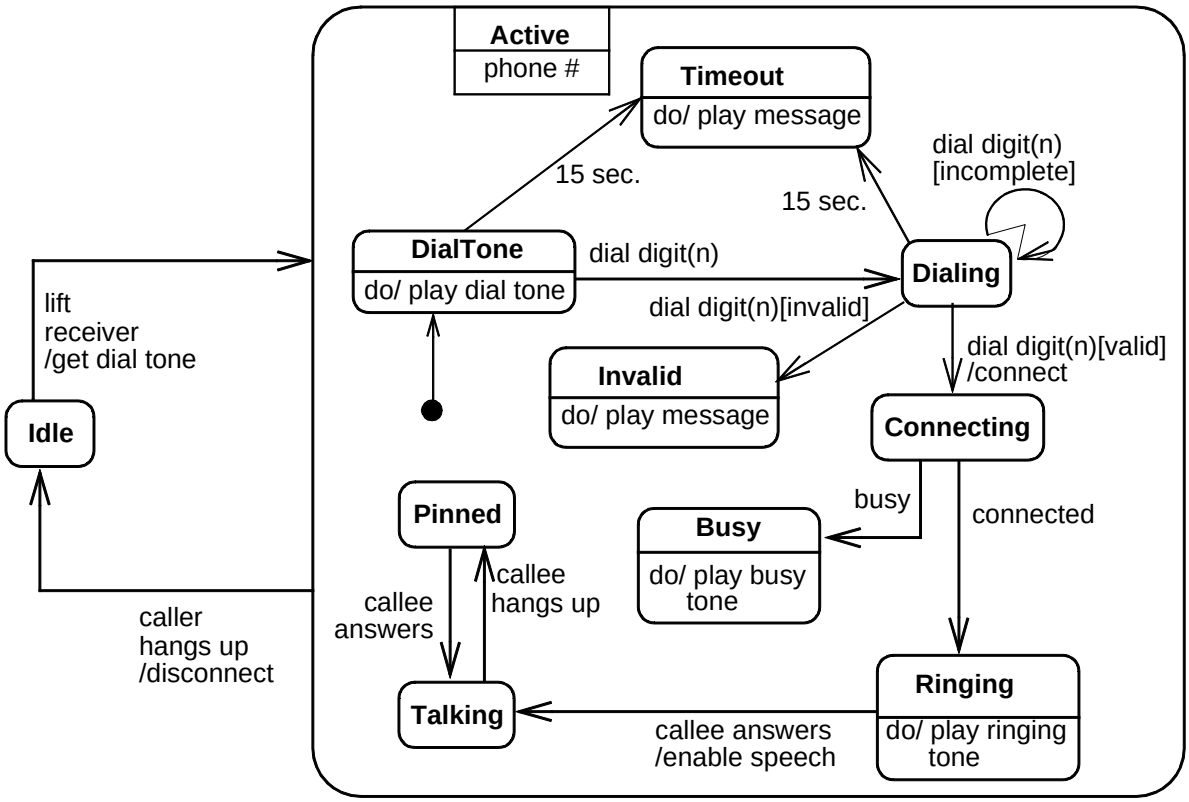
The semantics and notation described in this chapter are substantially those of David Harel's statecharts with some minor modifications. His work was a major advance on the traditional flat state machines.

8.1 STATE DIAGRAM

8.1.1 Notation

A state diagram is a bipartite graph of states and transitions. It is also a graph of states connected by physical containment and tiling. The entire state diagram is attached (through the model) to a class or a method (an operation implementation).

Figure 39. State diagram



8.2 STATES

8.2.1 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. An object remains in a state for a finite (non-instantaneous) time.

An internal “do” action is an ongoing process performed while the object is in the given state. The action need not be atomic; it is interruptible by outside events. It is initiated when the state is entered (after any incoming transition actions and entry actions). It may terminate by itself, in which case the termination represents an implicit “action complete” event. Otherwise it is externally terminated whenever the state is exited (before any exit action or outgoing transition actions). Nested state machines are equivalent to do-actions.

Each subregion of a state may have initial states and final states. A transition to the enclosing state represents a transition to the initial state. 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 of activity” event” on the enclosing state.

8.2.2 Notation

A state is shown as a rectangle with rounded corners. It may have one or more compartments. The compartments are all optional. They are as follows:

Name compartment. Holds the (optional) name of the state as a string. States without names are “anonymous” and are all distinct. Two state symbols with the same non-empty name designate the same state; multiple symbols with the same state name might be used for graphical convenience to avoid routing lines to a single state symbol.

State variable compartment. Holds a list of state variables that are defined within the state or any of its nested substates. State variables have the form of attributes. Their initial value expressions may include attributes or links of the owning object, state variables of enclosing states, and parameters of incoming transitions (if they appear on all incoming paths). State variable *are* attributes of the owning class but are distinguished because they are affected by or used by actions in the state diagram.

Internal activity compartment. Holds a list of internal actions or activities performed while the object is in the state. These have the format:

event-name argument-list ‘/’ *action-expression*

Each event name or pseudo-event name may appear at most once in a single state.

The following special actions have the same form but represent reserved words that cannot be used for event names:

‘entry’ ‘/’ *action-expression*An atomic action performed on entry to the state

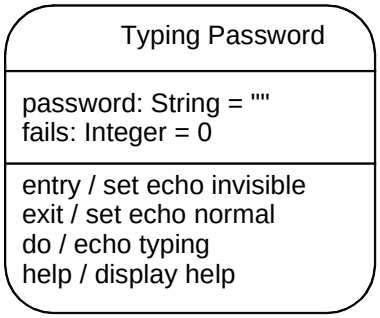
‘exit’ ‘/’ *action-expression*An atomic action performed on exit from the state

‘do’ ‘/’*action-expression*An ongoing action performed while in the state.

Action expressions may use state variables of the current or enclosing states, attributes and links of the owning object, and parameters of incoming transitions (if they appear on all incoming paths).

8.2.3 Example

Figure 40. State



8.3 SUBSTATES

A state can be refined using *and*-relationships into concurrent substates or using *or*-relationships into mutually exclusive disjoint substates. A given state may only be refined in one of these two ways. Its substates can may be refined in the same way or the other way.

A newly-created object starts in its initial state. The event that creates the object may be used to trigger a transition from the initial state symbol.

An object that transitions to its outermost final state ceases to exist.

8.3.1 Notation

An expansion of a state shows its fine structure. In addition to the (optional) name, state variable, 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 subregions. Each subregion is a concurrent substate. Each subregion 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.

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

An ongoing “do” action should not be specified in an enclosing state, as the decomposition of the state into substates shows its internal behavior.

An initial (pseudo)state is shown as a small solid filled circle. A transition from an initial state may be labeled with the name of an event; if so, it represents a transition to the enclosing state triggered by the given event. If it is unlabeled, it represents any transition to the enclosing state (and is therefore incompatible with another labeled initial state). The initial transition may have an action. The initial state is a notational device; an object may not be *in* such a state but must transition to an actual state.

A final (pseudo)state is shown as a circle surrounding a small solid filled circle (a bull’s eye). It may be labeled by a *send-event-expression*; if so, it represents the occurrence of an event at the level of the enclosing state. (In effect, reaching the state causes a subsequent transition on the enclosing state.) If the state is unlabeled, it represents the completion of activity in the enclosing state which triggers any transition on the implicit activity completion event.

8.3.2 Example

Figure 41. Sequential substates

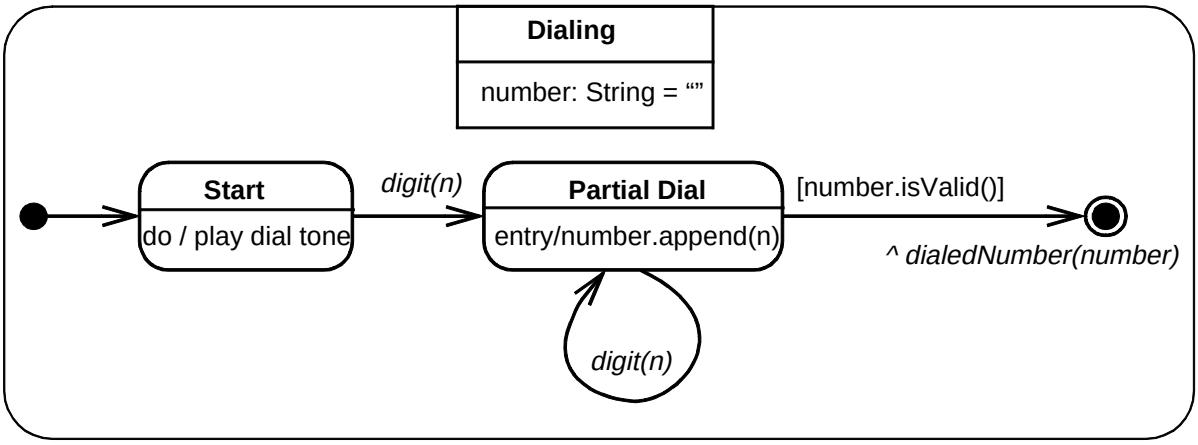
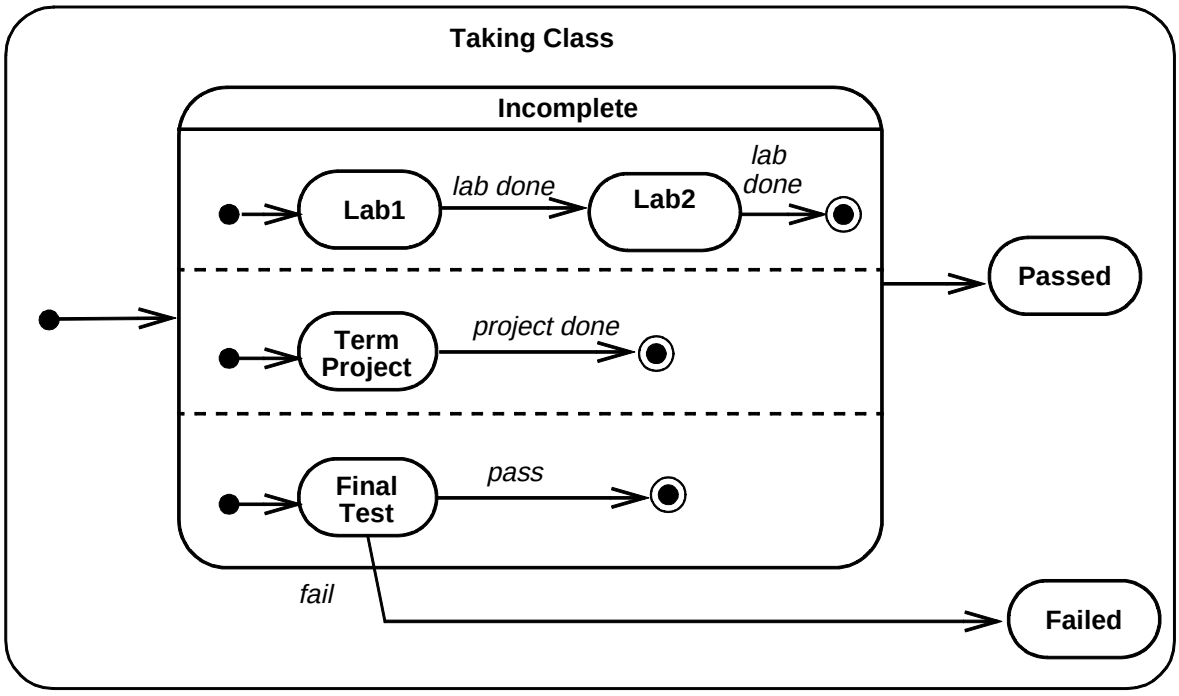


Figure 42. Concurrent substates



8.4 EVENTS

8.4.1 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 (usually described as a boolean expression). These are notated as guard conditions on transitions without event names.

receipt of an explicit signal from one object to another. These are notated as named events as triggers on transitions.

receipt of a call for an operation by an object. These are notated as named events as triggers on transitions.

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. These are notated as time expressions as triggers on transitions.

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.

8.4.2 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 event can be specified using the «signal» stereotype of a class 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 as an expression that evaluates (at modeling time) to an amount of time, such as “5 seconds”. By default, this indicates the amount of time

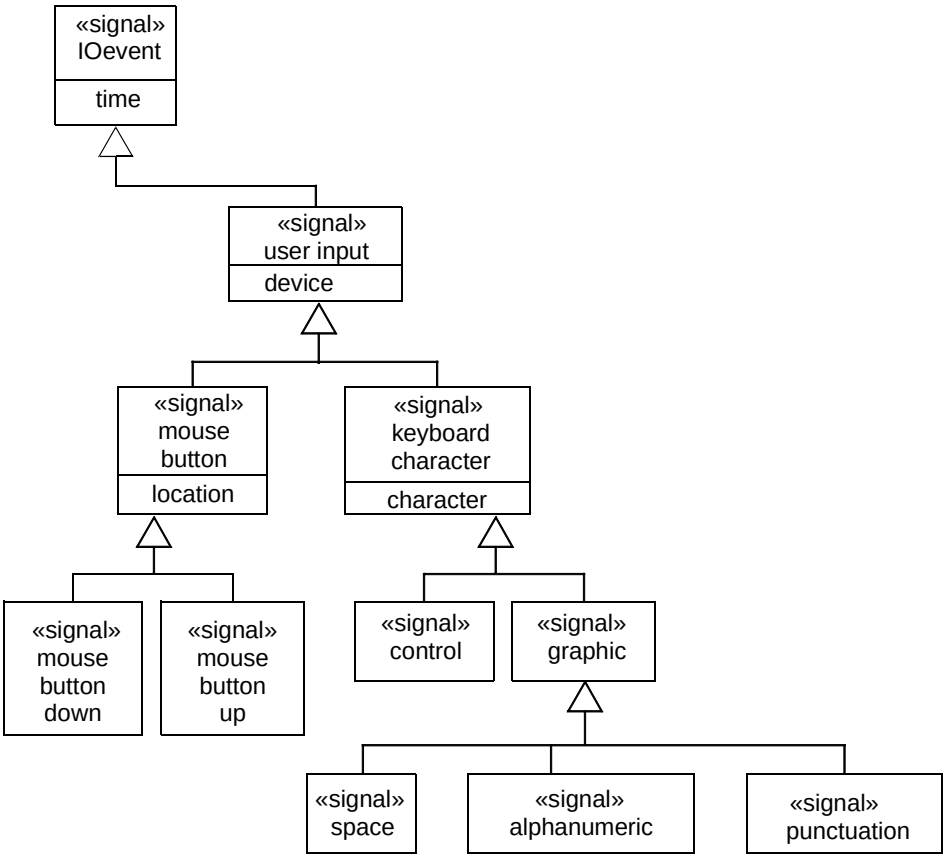
after the current state was entered. Other time events can be specified as conditions, such as [date = Jan. 1, 2000] or [10 seconds since exit from state A].

A condition becoming true is shown as a guard condition with no event. 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.

Events can be declared on a class diagram with the stereotype «event».

8.4.3 Example

Figure 43. Signal event declaration



8.5 SIMPLE TRANSITIONS

8.5.1 Semantics

A simple transition is a relationship between two states indicating that an object in the first state will enter the second state and perform certain specified actions when a specified event occurs if 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 available within actions specified on the transition or within actions initiated in the subsequent state. Events are processed one at a time. If an event does not trigger any transition, it is simply ignored. If it triggers more than one transition, only one will fire; the choice may be nondeterministic if a firing priority is not specified.

8.5.2 Notation

A transition is shown as a solid arrow from one state (the *source* state) to another state (the *target* state) labeled by a *transition string.* The string has the following format:

event-signature ‘[’ guard-condition] ‘/’ action-expression ‘^’ send-clause

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

event-name ‘(‘ *parameter* ‘,’ . . . ‘)’

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 *action-expression* is a procedural expression that 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. The action-clause must be an atomic operation, that is, it may not be interruptible; it must be executed entirely before any other actions are considered. The transition may contain more than one action clause (with delimiter).

‘The *send-clause* is a special case of an action, with the format:

destination-expression ‘.’ *destination-event-name* ‘(‘ *argument* ‘.’ . . . ‘)’

The transition may contain more than one send clause (with delimiter). The relative order of action clauses and send clauses is significant and determines their execution order.

The *destination-expression* is an expression that evaluates to an object or a set of objects.

The *destination-event-name* is the name of an event meaningful to the destination object(s).

The *destination-expression* and the arguments may be written in terms of the parameters of the triggering event and the attributes and links of the owning object.

Transition times. Names may be placed on transitions to designate the times at which they fire. See the section on transition times within Section 6.5.

8.5.3 Example

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

8.6 COMPLEX TRANSITIONS

A general 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.

8.6.1 Semantics

If the owning object is concurrently in all of the source states of a transition, then the transition is enabled. If the guard condition for the transition is true, then the transition fires and performs its actions. Then the object ceases to be in all of the source states and becomes in all of the target states. Normally this involves crossing out of or into a concurrent state region.

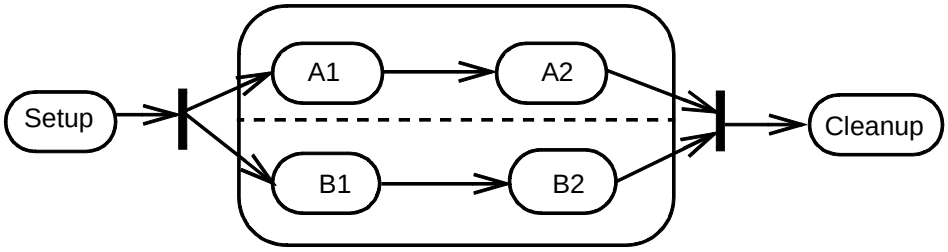
Normally all of the source states must be occupied before a complex transition is enabled. In more complicated situations, the guard condition may be expanded to permit firing when some subset of the states is occupied. Note that the concept of simultaneous event occurrence is nonphysical and is not supported; each transition is enabled by a single event.

8.6.2 Notation

A complex transition is shown as a short heavy vertical bar. The bar may have one or more solid arrows from states to the bar (these are the *source states*); the bar may have one or more solid 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.

8.6.3 Example

Figure 44. Complex transition



8.7 TRANSITIONS TO NESTED STATES

8.7.1 Semantics

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

A transition from a complex 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.

8.7.2 Notation

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

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

A transition drawn from a complex state boundary indicates a transition of the complex 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 complex 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 complex state, transition arrows may be specified from one or more concurrent states to a synchronization bar; specific states in the other regions are therefore 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 not have any outgoing transitions. If transition to the history indicator fires it indicates that the object resumes the state it last had within the complex region; any necessary entry actions are performed.

8.7.3 Presentation options

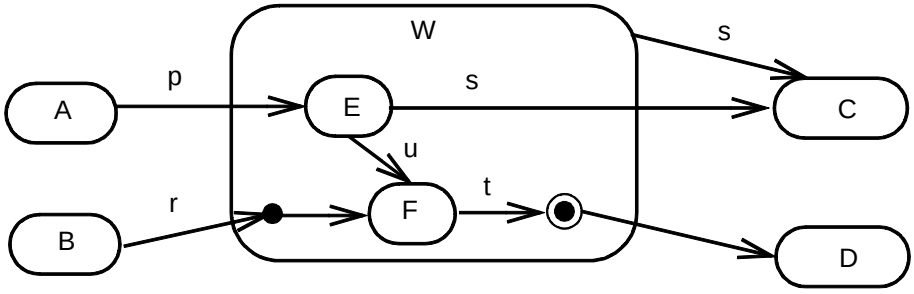
Stubbed transitions. Nested states may be suppressed. Transitions to nested states are subsumed to the most specific visible enclosing state of the suppressed state. Subsumed transitions that do not come from an unlabeled final state or go to an unlabeled initial state may (but need not) be shown as coming from or going to *stubs*. A *stub* is shown as a small vertical line 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 contour or to an internal substate, including a transition to a stubbed state. Events should not normally be shown on transitions leading from a stubbed state to an external state, however. 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 complex state contour (denoting the implicit event "action complete" for the corresponding state).

8.7.4 Example

See Figure 42 for an example of complex transitions.

Figure 45. Stubbed transitions



may be abstracted as

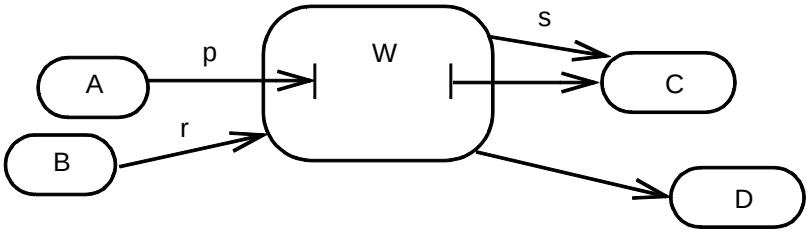
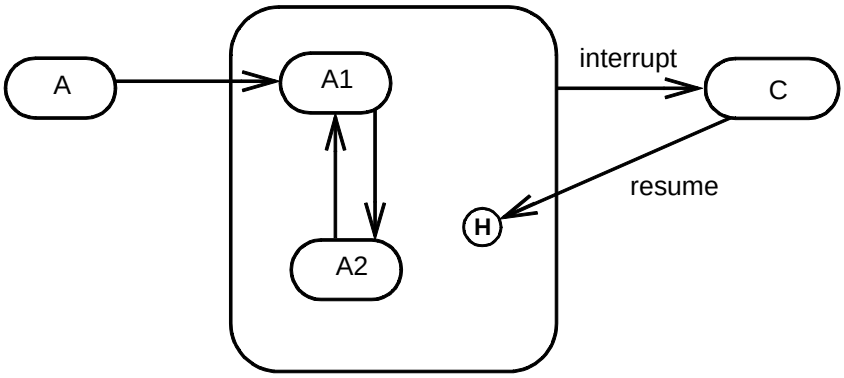


Figure 46. History indicator



8.8 SENDING MESSAGES

8.8.1 Semantics

Messages are sent by an action in an object to a target set of objects; the target set can be degenerate as a single object or the entire system. The sender can be subsumed to an object, a composite object, or a class.

8.8.2 Notation

See Section 8.5 for the text syntax of sending messages that cause events for other objects.

Sending such a message can also be shown visually. See Section 6.4 and Section 7.10 for details of showing messages in sequence diagrams and collaboration diagrams.

Sending a message between state diagrams may be shown by drawing a dashed arrow from the sender to the receiver. Messages must be sent between objects, so this means that the diagram must be some form of object diagram containing objects (not classes). The arrow is labeled with the event name and arguments of the event that is caused by the reception of the event. Each state diagram must be contained within an object symbol representing a collaborating object; graphically the state diagrams may be nested physically within an object symbol, or the object enclosing *one* state diagram may be implicit (being the object owning the main state diagram at issue). The state diagrams represent the states of the collaborating objects.

The sender symbol may be one of:

A transition. The message is sent as part of the action of firing the transition. This is an alternate presentation to the text syntax for sending messages.

An object. The message is sent by an object of the class at some point in its life, but the details are unspecified.

The receiver may be one of:

An object, including a class reference symbol containing a state diagram. The message is received by the object and may trigger a transition on the corresponding event. There may be many transitions involving the event. This notation may not be used when the target object is computed dynamically; in that case a text expression must be used.

A transition. The transition must be the only transition in the object involving the given event, or at least the only transition that could possibly be triggered by the

particular sending of the message. This notation may not be used when the transition triggered depends on the state of the receiving object and not just on the sender.

A class designation. This notation would be used to model the invocation of class-scope operations, such as the creation of a new instance. The receipt of such a message causes the instantiation of a new object in its default initial state. The event seen by the receiver may be used to trigger a transition from its default initial state and therefore represents a way to pass information from the creator to the new object.

8.8.3 Example

Figure 47. Sending messages

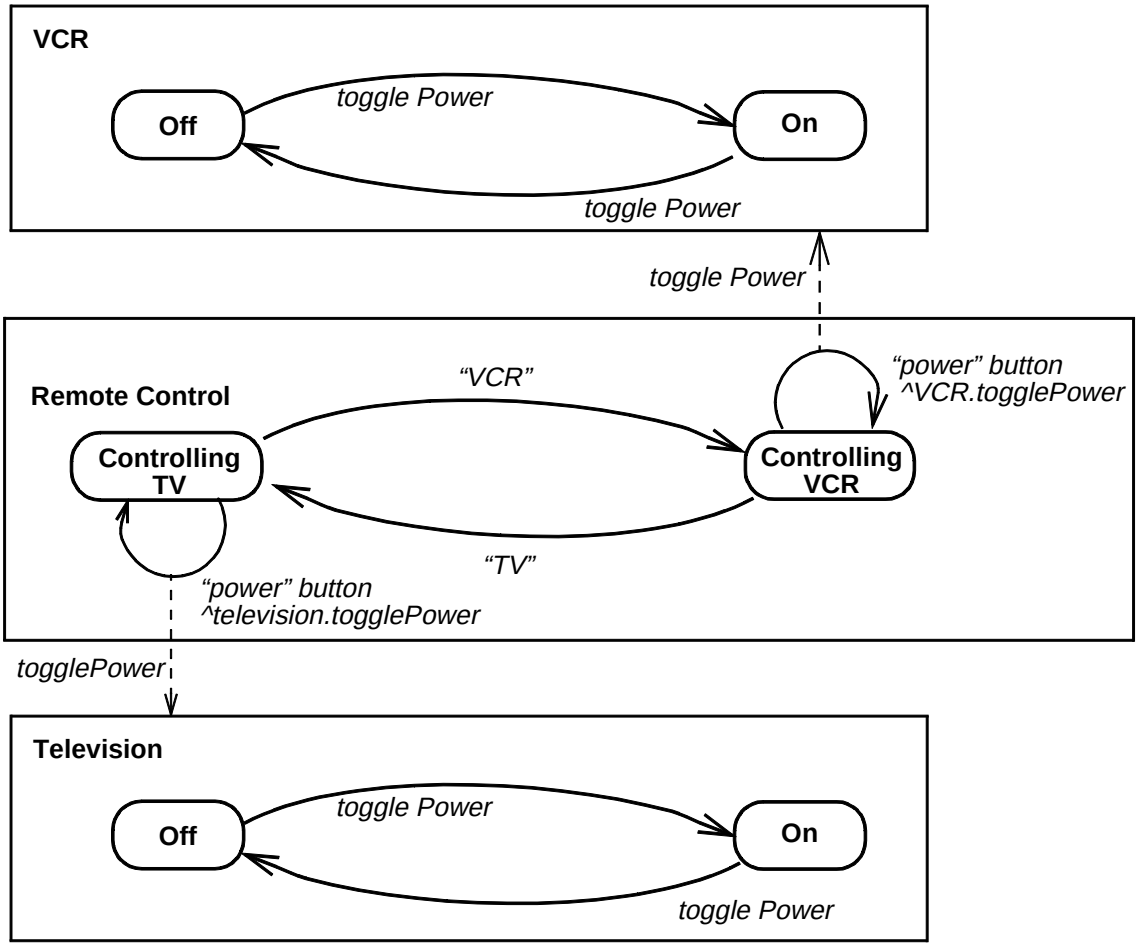
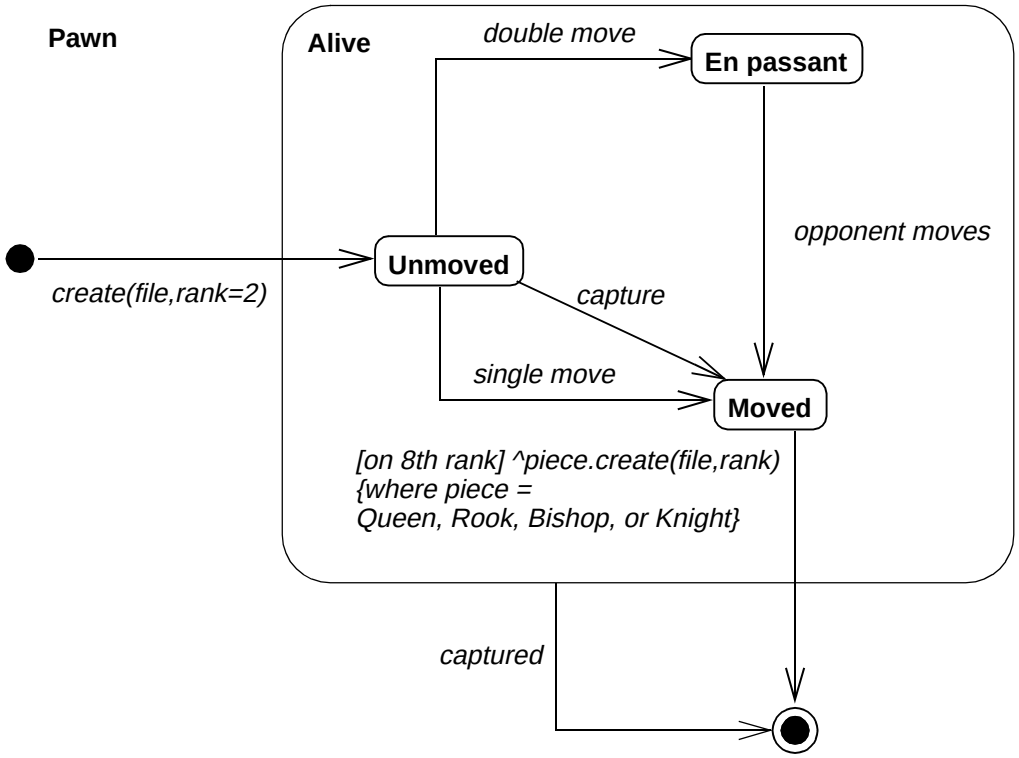


Figure 48. Creating and destroying objects



8.9 INTERNAL TRANSITIONS

8.9.1 Semantics

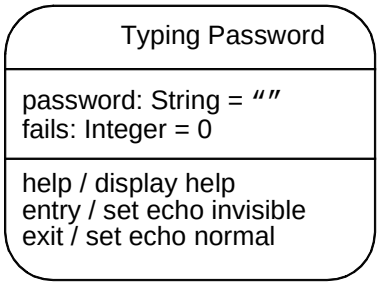
An internal transition is a transition that remains within a single state rather than a transition that involves two states. It represents the occurrence of an event that does not cause a change of state. By analogy it is also used for the pseudoevents of entering the state (from any other state not nested in the particular state), exiting the state (to any other state not nested in the particular state), and performing an action while in the state.

Note that an internal transition is not equivalent to a self-transition from a state back to the same state. The self-transition causes the exit and entry actions on the state to be executed and the initial state to be entered, whereas the internal transition does not invoke the exit and entry actions and does not cause a change of state (including a nested state).

8.9.2 Notation

An internal transition is attached to the state rather than a transition. Graphically it is shown as a text string within the internal transition compartment on a state symbol. The syntax of an internal transition string is the same as for an external transition. See Section 8.5 for details.

Figure 49. State with state variables and internal transitions



9. ACTIVITY DIAGRAM

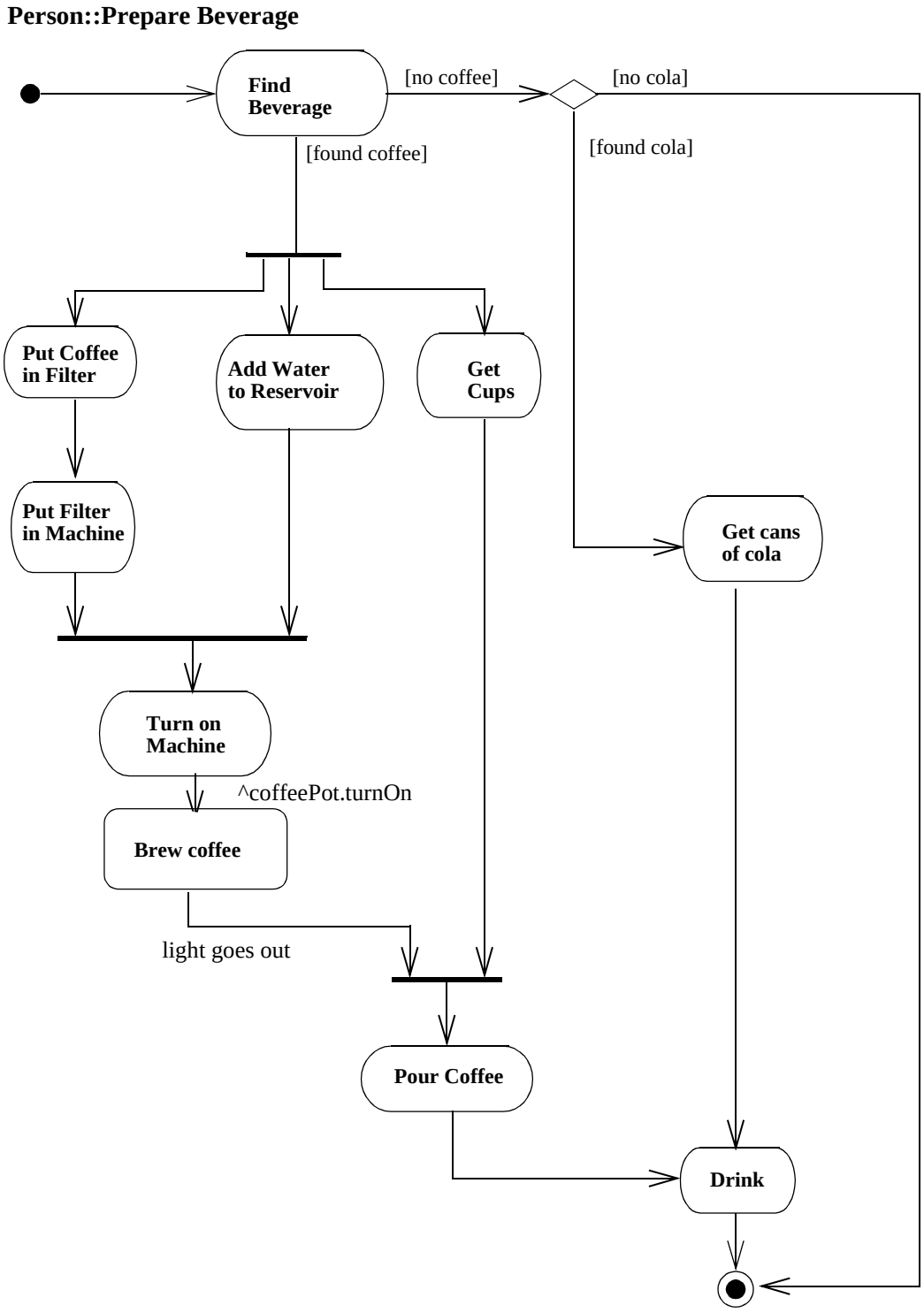
9.1 ACTIVITY DIAGRAM

9.1.1 Notation

An activity diagram is a special case of a state diagram in which all (or at least most) of the states are action states and in which all (or at least most) of the transitions are triggered by completion of the actions in the source states. The entire activity diagram is attached (through the model) to a class or to the implementation of an operation or a use case. 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.

9.1.2 Example

Figure 50. Activity diagram



9.2 ACTION STATE

9.2.1 Semantics

An *action state* is a shorthand for a state with an internal action and at least one outgoing transition involving the implicit event of completing the internal action (there may be several such transitions if they have guard conditions). Action states should not have internal transitions or outgoing transitions based on explicit events; use normal states for this situation. The normal use of an action state is to model a step in the execution of an algorithm (a procedure).

9.2.2 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.

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.

9.2.3 Presentation options

The action may be described by natural language, pseudocode, 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 but they are more commonly used with activity diagrams, which are special cases of state diagrams.

9.2.4 Example

Figure 51. Activities

9.3 DECISIONS

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 shorthand for showing decisions.

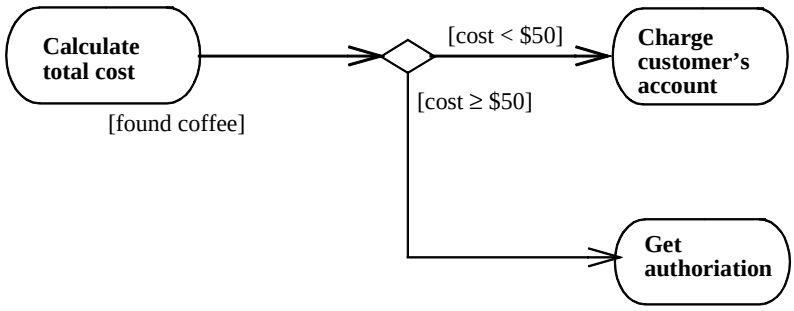
9.3.1 Notation

A decision may be shown by labeling multiple output transitions of an action with different guard conditions.

For convenience a stereotype is provided for a decision: the traditional diamond shape, with one or more incoming arrows and with two or more outgoing arrows, each labeled by a distinct guard condition with no event trigger. All possible outcomes should appear on one of the outgoing transitions.

9.3.2 Example

Figure 52. Decision



9.4 SWIMLANES

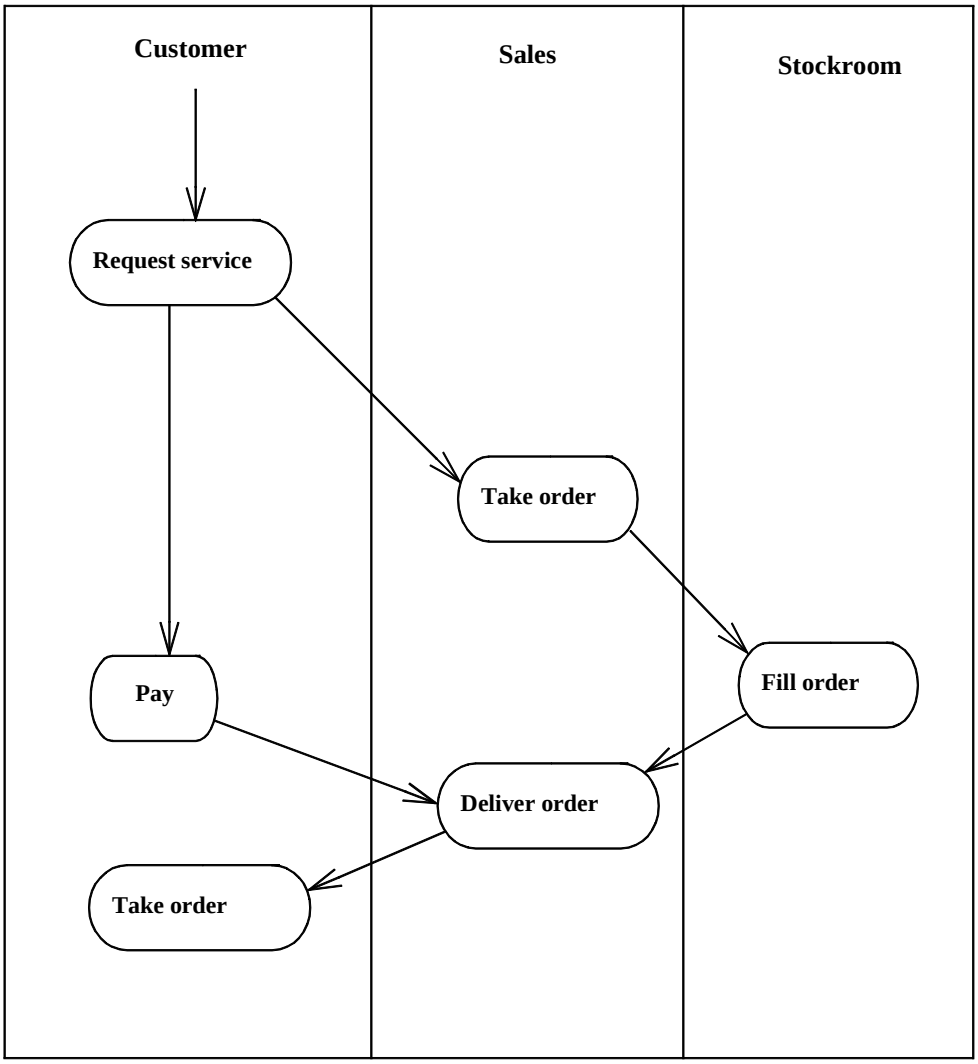
Actions may be organized into *swimlanes.* Swimlanes are a kind of package for organizing responsibility for activities within a class. They often correspond to organizational units in a business model.

9.4.1 Notation

An activity diagram may be divided visually into “swimlanes” each separated from neighboring swimlanes by vertical solid lines on both sides. Each swimlane represents responsibility for part of the overall activity, and may eventually be implemented by one or more objects. The relative ordering of the swimlanes has no semantic significance but might indicate some affinity. Each action is assigned to one swimlane. Transitions may cross lanes; there is no significance to the routing of a transition path.

9.4.2 Example

Figure 53. Swimlanes in activity diagram



9.5 ACTION-OBJECT FLOW RELATIONSHIPS

Activities operate by and on objects. Two kinds of relationships can be shown: The kinds of objects that have primary responsibility for performing an action and the other objects whose values are used or determined by the action.

9.5.1 Notation

Object responsible for an action. The object responsible for performing an action can be shown by drawing a lifeline and placing actions on lifelines Each lifeline represents a distinct object. There may be multiple lifelines for different objects of the same or different kinds. If this approach is chosen, usually a sequence diagram should be used. See Section 6.1. If an object lifeline is not shown, then some object within the swimlane package is responsible for the action but the object is not shown. Multiple actions within a single swimlane can be handled by the same or different objects.

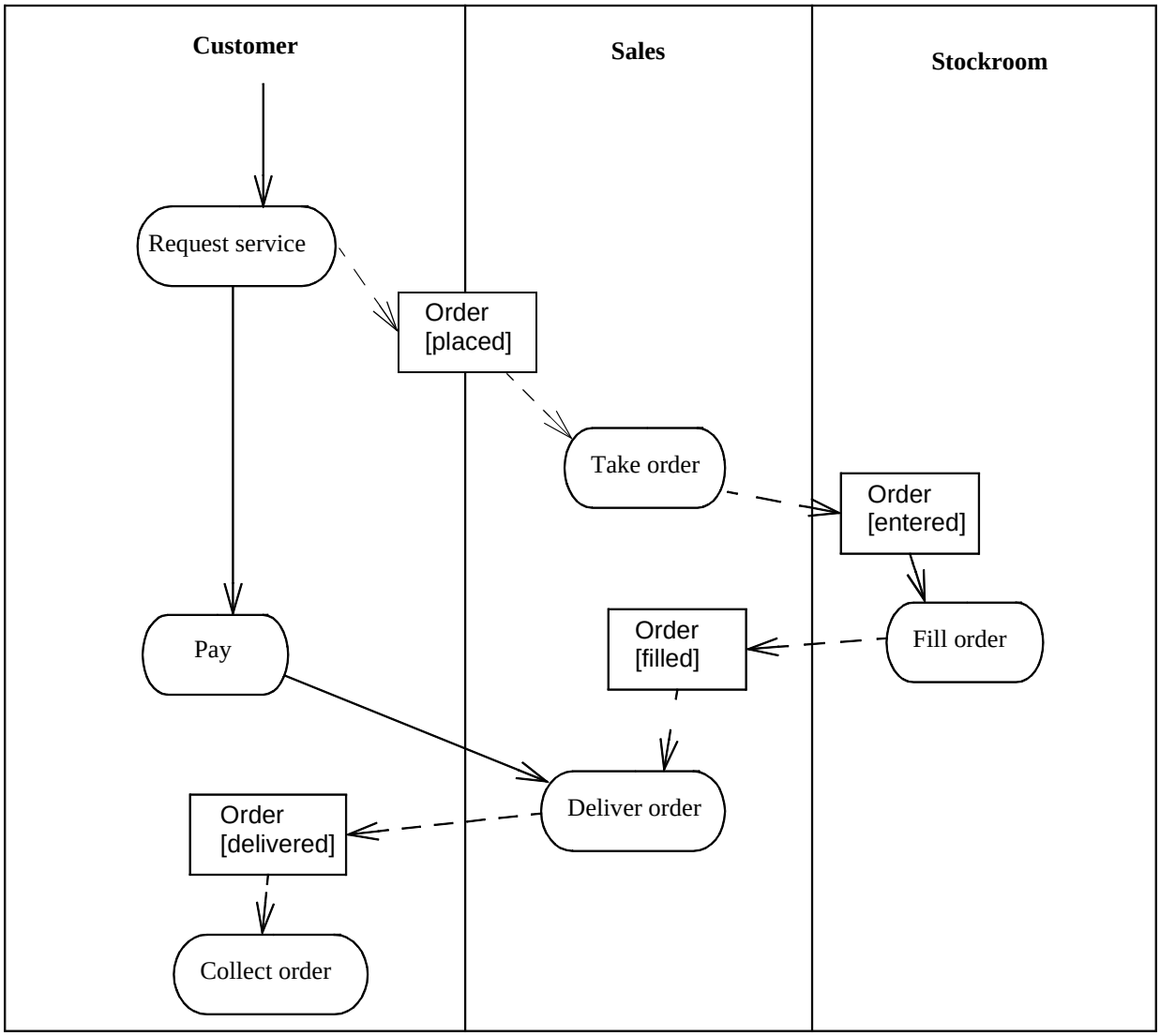
Object flow. Objects that are input to or output by an action may be shown as object symbols. A dashed arrow is drawn from an action to an output object, and a dashed arrow is drawn from an input object to its action. The same object may be (and usually is) the output of one action and the input of one or more subsequent activities.

The control flow (solid) arrows may be omitted when the object flow (dashed) arrows supply a redundant constraint. In other words, when an action produces an output that is input by a subsequent action, that object flow relationship implies a control constraint.

Object state. Frequently the same object is manipulated by a number of successive activities. It is possible to show the arrows to and from all of the relevant activities. For greater clarity, however, the object may be displayed multiple times on a diagram, each appearance denoting a different point during its 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].

9.5.2 Example

Figure 54. Actions and object flow



9.6 OPTIONAL STEREOTYPES

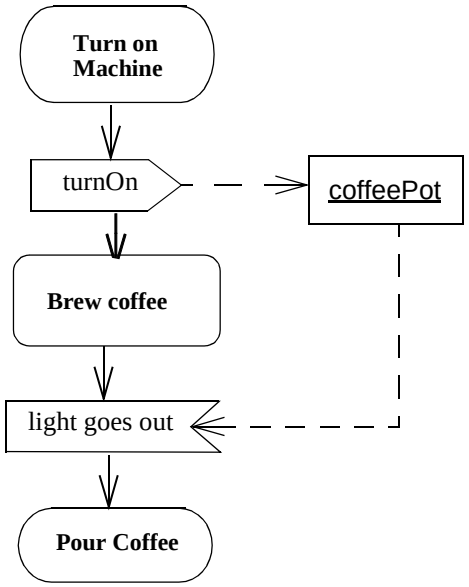
The following stereotypes provide explicit symbols for certain kinds of information that can be specified on transitions. These stereotypes are not necessary for constructing activity diagrams but some users may prefer the added impact that they provide.

9.6.1 Stereotypes

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. This symbol replaces the event label on the transition. 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.

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. This symbol replaces the send-signal label on the transition. 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 55. Stereotypes for signal receipt and sending



10. IMPLEMENTATION DIAGRAMS

Implementation diagrams show aspects of implementation, including source code structure and run-time implementation structure. They come in two forms: component diagrams show the structure of the code itself and deployment diagrams show the structure of the run-time system.

10.1 COMPONENT DIAGRAMS

10.1.1 Semantics

A component diagram shows the dependencies among software components, including source code components, binary code components, and executable components. A software module may be represented as a component type. Some components exist at compile time, some exist at link time, and some exist at run time; some exist at more than one time. A compile-only component is one that is only meaningful at compile time; the run-time component in this case would be an executable program.

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).

10.1.2 Notation

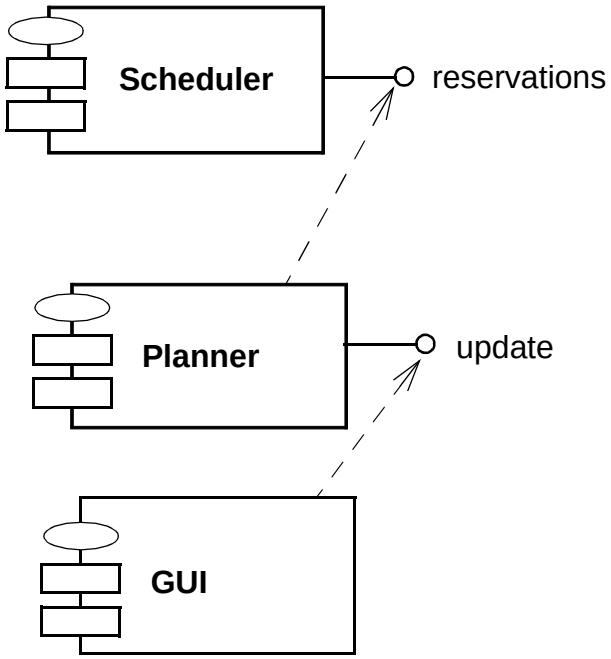
A component diagram is a graph of components connected by dependency relationships. Components may also be connected to components by physical containment representing composition relationships.

A diagram containing component types and node types may be used to show compiler dependencies, 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 language-specific and may be shown as stereotypes of the dependencies.

The diagram may also be used to show interfaces and calling dependencies among components, using dashed arrows from components to interfaces on other components.

10.1.3 Example

Figure 56. Component diagram



10.2 DEPLOYMENT DIAGRAMS

10.2.1 Semantics

Deployment diagrams show the configuration of run-time processing elements and the software components, processes, and objects that live on them. Software component instances represent run-time manifestations of 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.

10.2.2 Notation

A deployment diagram is a graph of nodes connected by communication associations. Nodes may contain component instances; this indicates that the component lives or runs on the node. Components may contain objects; this indicates that the object is part of 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.

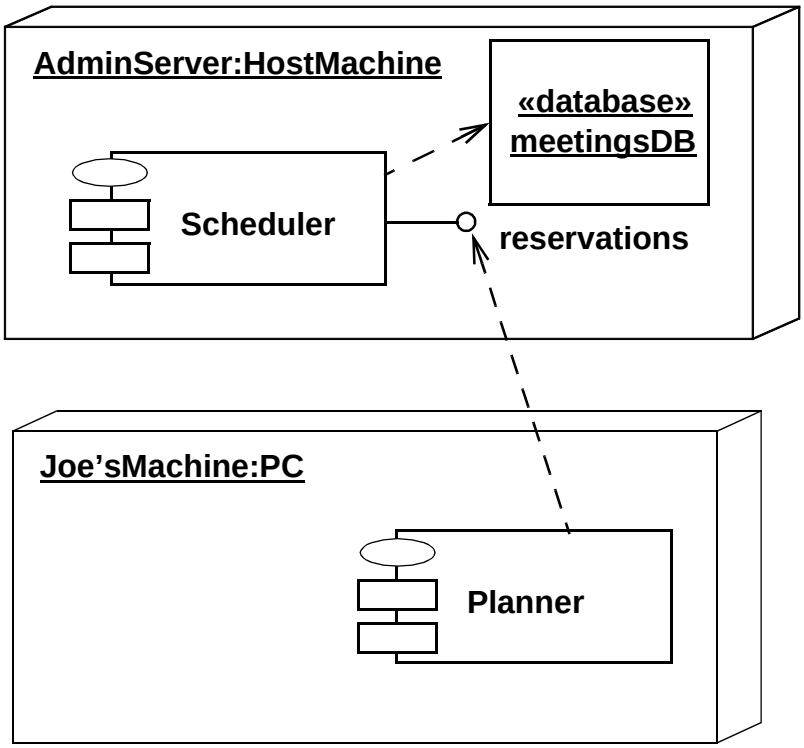
The deployment type diagram may also be used to show which components may run on which nodes, by using dashed arrows with the stereotype «supports».

Migration of components from node to node or objects from component to component may be shown using the «becomes» stereotype of the dependency relationship. In this case the component or object is resident on its node or component only part of the entire time.

Note that a process is just a special kind of object (see Active Object).

10.2.3 Example

Figure 57. Nodes



10.3 NODES

10.3.1 Semantics

A node is a run-time physical object that represents a computational resource, generally having at least a memory and often processing capability as well. Nodes may be represented as type and as instances. Run time computational instances, both objects and component instances, may reside on node instances.

10.3.2 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.

Dashed-arrow dependency arrows show the capability of a node type to support a component type. A stereotype may be used to state the precise kind of dependency.

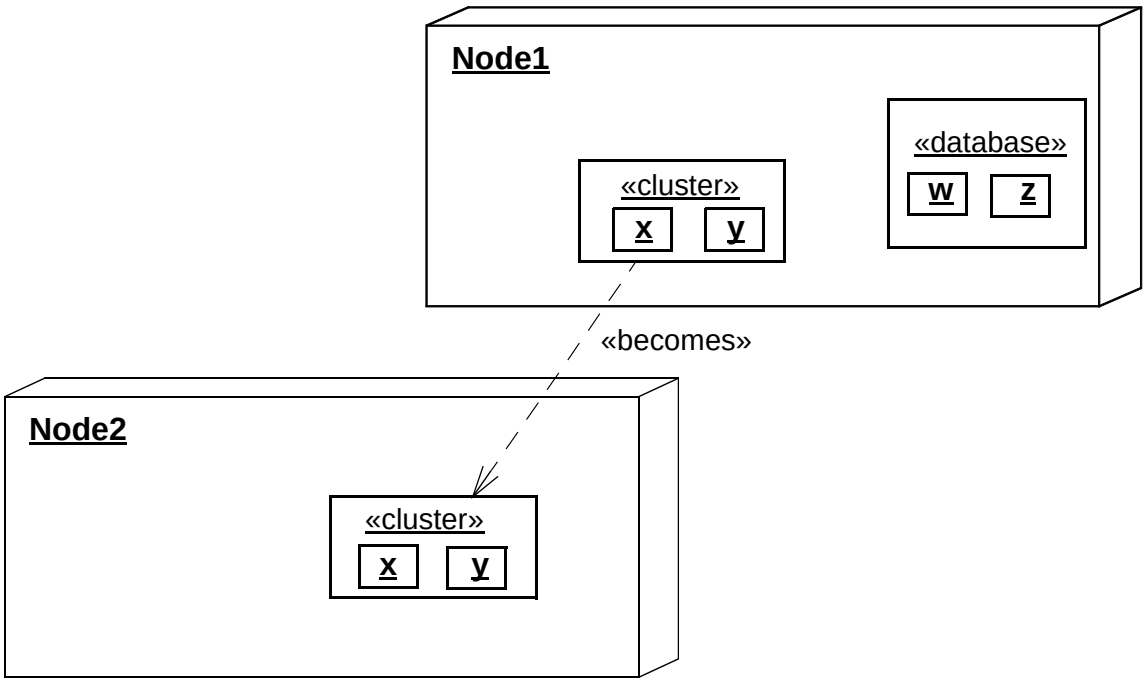
Component instances and objects may be contained within node instance symbols. This indicates that the items reside on the node instances.

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).

10.3.3 Example

This example shows two nodes containing an object (cluster) that migrates from one node to another and also an object that remains in place.

Figure 58. Use of nodes to hold objects



10.4 COMPONENTS

10.4.1 Semantics

A component type represents a piece of software code (source, binary, or executable) and may be used to show compiler and run-time dependencies. A component instance represents a run-time code unit and may be used to show code units that have identity at run time, including their location on nodes.

10.4.2 Notation

A component is shown as a rectangle with one small ellipse and 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:

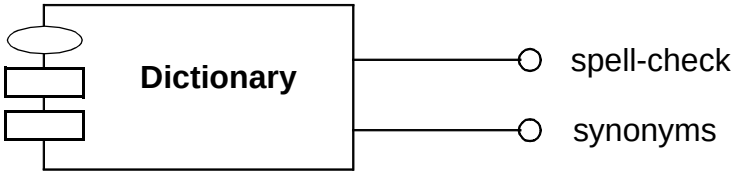
component-name ':' *component-type*

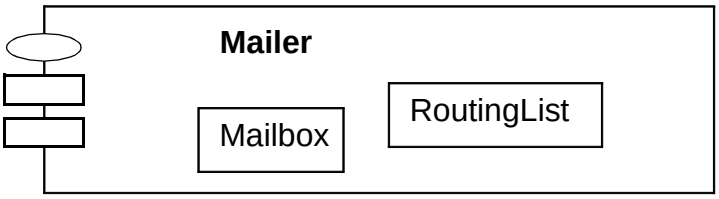
A property may be used to indicate the life-cycle stage that the component describes (source, binary, executable, or more than one of those). Only executable components (including programs, DLLs, run-time linkable images, etc.) may be located on nodes.

10.4.3 Example

The example shows a component with interfaces and also a component that contains objects at run time.

Figure 59. Component





10.5 LOCATION OF COMPONENTS AND OBJECTS WITHIN OBJECTS

Instances may be located within other instances. For example, objects may live in processes that live in components that live on nodes. '

10.5.1 Notation

The location of an instance (including objects, component instances, and node instances) within another instance may be shown by physical nesting. Alternately, an instance may have a property tag “location” whose value is the name of the containing instance.

If an object moves during an interaction, then it may be as two or more occurrences with a “becomes” dependency between the occurrences. The dependency may have a time property attached to it to show the time when the object moves. Each occurrence represents the object during a period of time. Messages should be directed to the correct occurrence of the object.

10.5.2 Example

See the other diagrams in this section for examples of objects and components located on nodes as well as migration.

Index

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

Index