UNIFIED
MODELING
LANGUAGE

UML Notation Guide

version 1.1
1 September 1997

Rational Software ■ Microsoft ■ Hewlett-Packard ■ Oracle

Sterling Software ■ MCI Systemhouse ■ Unisys ■ ICON Computing

IntelliCorp ■ i-Logix ■ IBM ■ ObjecTime ■ Platinum Technology ■ Ptech

Taskon ■ Reich Technologies ■ Softeam

ad/97-08-05

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 semantic concepts subdivided by 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 representation of the semantic concept ("forward mapping to notation").
>
> Presentation options: Describes various options in presenting the model information, such as the ability to suppress or filter information, alternate ways of showing things, and suggestions for alternate ways of presenting information within a tool. Dynamic tools need the freedom to present information in various ways and we do not want to restrict this excessively. In some sense, we are defining the "canonical notation" that printed documents show, rather than the "screen notation". 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 just one of the presentation options and implement it; tools should offer users the options of selecting among various presentation options, including some that are not described in this document.
>
> Style guidelines: 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 suggestions, but the use of some consistent guidelines of your own choosing is recommended in any case.

Example: Shows samples of the notation. String and code examples are given in the following font: `This is a string sample.`

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

2. DIAGRAM ELEMENTS

This chapter discusses mechanisms of the notation. These are generic mechanisms that are used in various ways in subsequent chapters to represent semantics.

2.1 GRAPHS AND THEIR CONTENTS

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

UML notation is intended to be drawn on 2-dimensional surfaces. Some shapes are 2-dimensional projections of 3-d shapes (such as cubes) but they are still rendered as icons on a 2-dimensional surface. In the near future true 3-dimensional layout and navigation may be possible on desktop machines 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

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

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

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 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.

2.5.1 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.

2.5.2 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.

2.5.3 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.

2.5.4 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.

2.5.5 Mapping

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

2.6 NAME

2.6.1 Semantics

A name is a string that is used to uniquely 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.

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

2.6.2 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.

2.6.3 Example

Names:

BankAccount

integrate

controller

abstract

this_is_a_very_long_name_with_underscores

Pathname:

MathPak::Matrices::BandedMatrix.dimension

2.6.4 Mapping

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

2.7 LABEL

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

2.7.1 Semantics

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

2.7.2 Notation

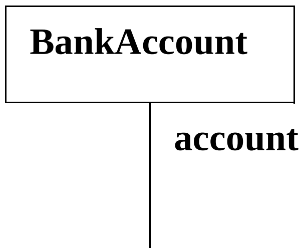
A label is a string that is graphically attached to another symbol on a diagram. Visually the attachment is normally by containment of the string (in a closed region) or by placing the string near the symbol. Sometimes the string is placed in a definite position (such as below a symbol) but most of the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

2.7.3 Presentation options

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

2.7.4 Example

Figure 1. Attachment by containment and attachment by adjacency



2.8 KEYWORDS

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

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

«*keyword***»**

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

2.9 EXPRESSION

2.9.1 Semantics

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

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

2.9.2 Notation

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

2.9.3 Example

BankAccount

BankAccount * (*) (Person*, int)

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

[i > j and self.size > i]

2.9.4 Mapping

An expression string maps to an Expression element (possibly a particular subclass of Expression, such as ObjectSetExpression or TimeExpression).

2.9.5 OCL Expressions

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

2.9.6 Selected OCL Notation

Syntax for some common navigational expressions are shown below. These forms can be chained together. The leftmost element must be an expression for an object or a set of objects. The expressions are meant to work on sets of values when applicable. For more details and syntax see the OCL description.

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

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

set '–>' 'select' '(' *boolean-expression* ')'
the *boolean-expression* is written in terms of objects within the set. The result is the subset of objects in the set for which the boolean expression is true.

2.9.7 Example

flight.pilot.training_hours > flight.plane.minimum_hours

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

2.10 NOTE

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

2.10.1 Semantics

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

2.10.2 Notation

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

2.10.3 Presentation options

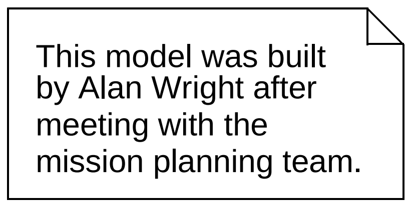
A note may have a stereotype.

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.

2.10.4 Example

See also Section 4.1.3 for a note symbol containing a constraint.

Figure 2. Note



2.10.5 Mapping

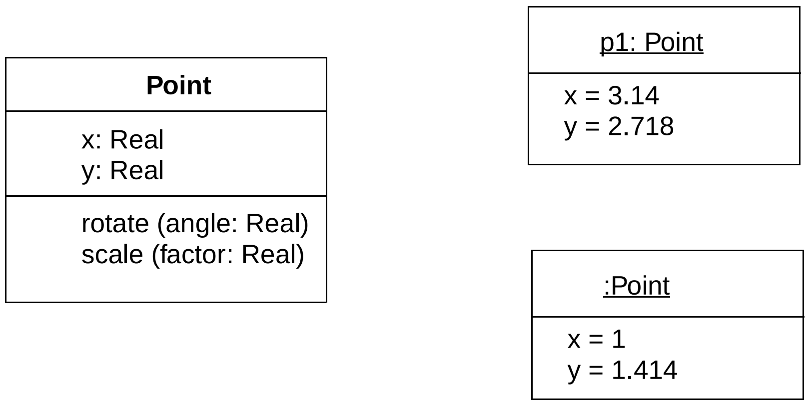
A note may represent the textual information in several possible metamodel constructs; it must be created in context that is known to a tool, and the tool must maintain the mapping. The string in the note maps to the body of the corresponding modeling element. A note may represent: a constraint; a tagged value; the body of a method; or other string values within modeling elements. It may also represent a comment attached directly to a diagram element.

2.11 TYPE-INSTANCE CORRESPONDENCE

A major purpose of modeling is to prepare generic descriptions that describe many specific particular items. This is often known as the *type-instance dichotomy.* Many or most of the modeling concepts in UML have this dual character, usually modeled by two paired modeling elements, one 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.

Figure 3. Classes and objects



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

3. MODEL MANAGEMENT

3.1 PACKAGES AND MODEL ORGANIZATION

3.1.1 Semantics

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. Some packages may be Subsystems or Models. The entire system description can be thought of as a single high-level subsystem package with everything else in it. All kinds of UML model elements and diagrams can be organized into packages.

Note that packages *own* model elements and model fragments and are the basis for configuration control, storage, and access control. Each element can be directly owned by a single package, so the package hierarchy is a strict tree. However, packages can reference other packages, so the usage network is a graph.

There are several predefined stereotypes of Model and Subsystem. See the metamodel document for details. In particular, the stereotype «system» of Subsystem denotes the entire set of models for the complete system being modeled; it is the root of the package hierarchy and the only model element that is not owned by some other model element.

3.1.2 Notation

A package is shown as a large rectangle with a small rectangle (a "tab") attached on one corner (usually the left side of the upper side of the large rectangle). It is a manila folder shape.

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 keyword string may be placed above the package name. The keywords *subsystem* and *model* indicate that the package is a metamodel Subsystem or Model. The predefined stereotypes *system*, *facade*, *framework*, and *top package* are also notated with keywords. User-defined stereotypes of one of these predefined kinds of package are also notated with keywords, but they must not conflict with the predefined keywords.

A list of properties as may be placed in braces after or below the package name. Example: {abstract}. See Section 4.2.2 for details of property syntax.

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

The visibility of a package element outside the package may be indicated by preceding the name of the element by a visibility symbol ('+' for public, '-' for private, '#' for protected). If the element is an inner package, the visibilities of its elements as exported by the outer package are obtained by combining the visibilities of an element within the package with the visibility of the package itself: the most restrictive visibility results.

Relationships may be drawn between package symbols to show relationships between at least some of the elements in the packages. In particular, dependency between packages implies that there exist one or more dependencies among the elements.

3.1.3 Presentation options

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

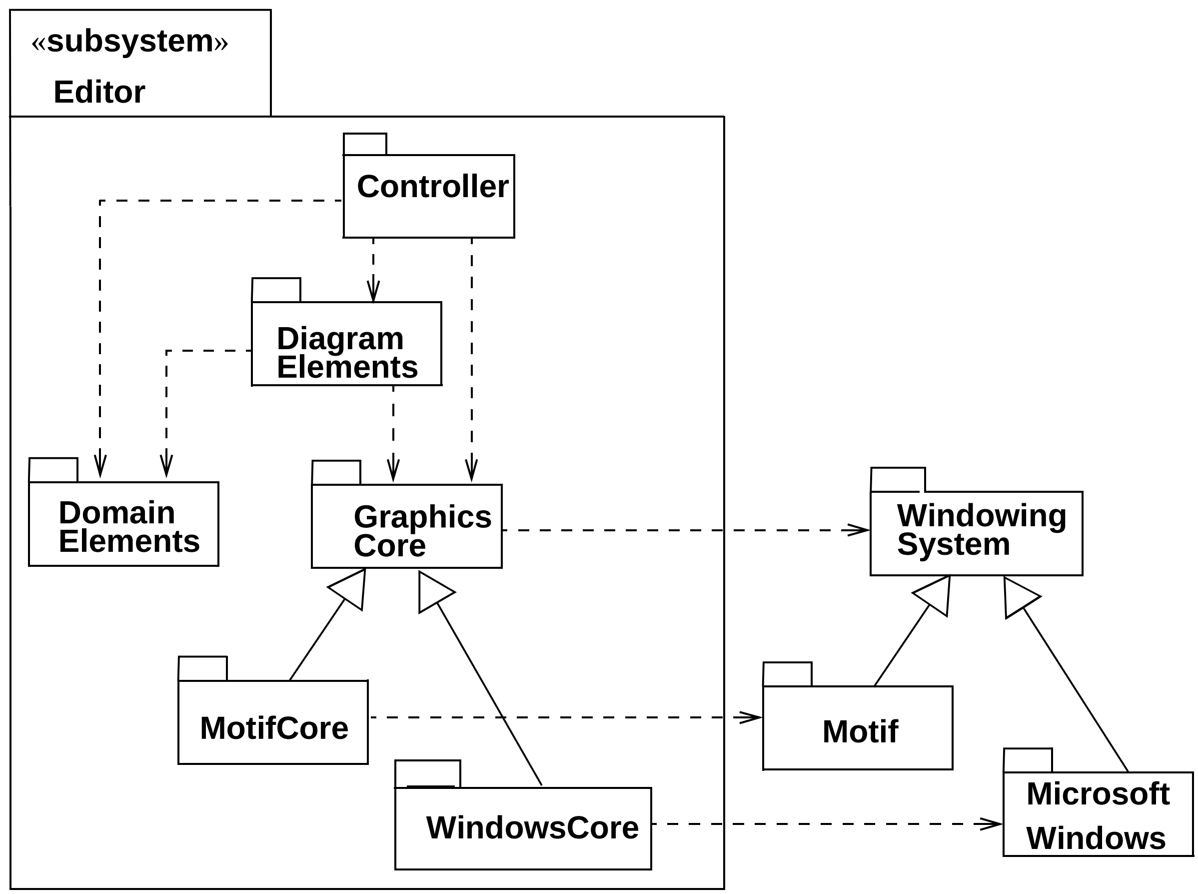
A tool may show visibility by a graphic marker, such as color or font.

3.1.4 Style guidelines

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

3.1.5 Example

Figure 4. Packages and their dependencies



3.1.6 Mapping

A package symbol maps into a Package element. The name on the package symbol is the name of the Package element. If the package has a keyword that is a predefined keyword, then the package symbol maps into the corresponding subclass of Package or into the corresponding stereotype of Package; otherwise it maps into a user-defined stereotype of Package.

A symbol directly contained within the package symbol (i.e., not contained within another symbol) maps into a model element owned by the package element. However, a symbol whose name is a pathname maps into a reference to a model element owned by another package; only the reference is owned by the current package. Relationships from the package symbol boundary map into relationships to the package element.

4. GENERAL EXTENSION MECHANISMS

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

4.1 CONSTRAINT AND COMMENT

4.1.1 Semantics

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

A *comment* is a text string (including references to human-readable documents) attached directly to a model element. This is syntactically equivalent to a constraint written in the language “text” whose meaning is significant to humans but which is not conceptually executable (except inasmuch as humans are regarded as the instruments of interpretation). A comment can therefore attach arbitrary textual information to any model element of presumed general importance.

4.1.2 Notation

A constraint is shown as a text string in braces ({ }). There is an expectation that individual tools may provide one or more languages in which formal constraints may be written. One predefined language for writing constraints is OCL (defined in a companion document). Otherwise the constraint may be written in natural language. A constraint may be a “comment”; it that case it is written in text (possibly including pictures or other viewable documents) for “interpretation” by a human. Each constraint is written in a specific language, although the language is not generally displayed on the diagram (the tool must keep track of it).

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

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

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

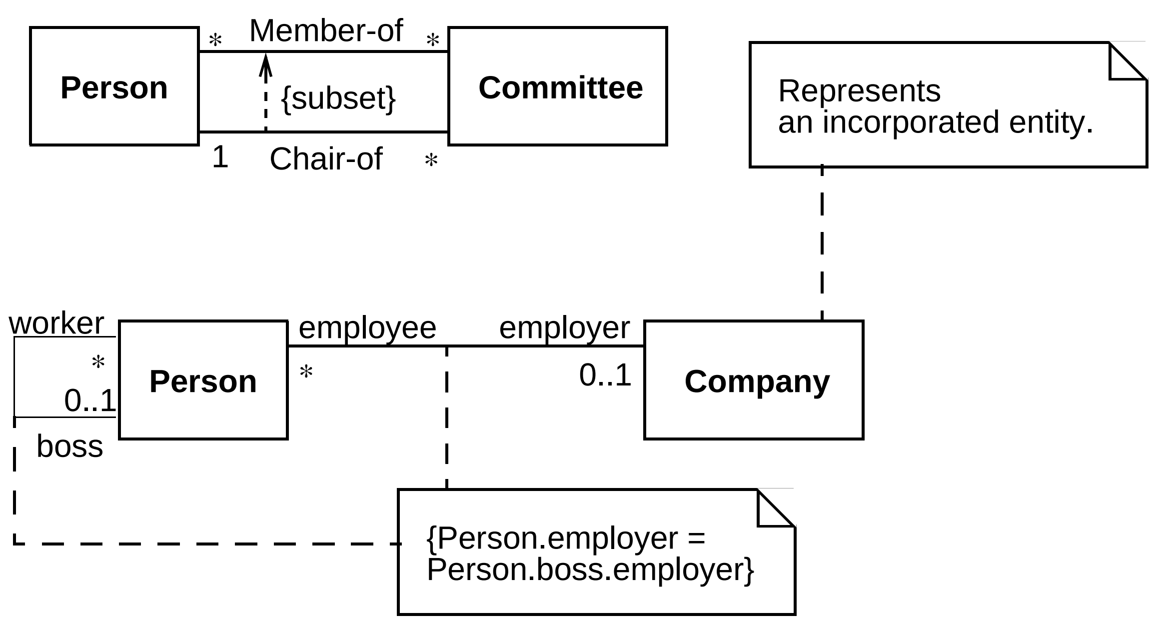
For two graphical symbols (such as two classes or two associations): The constraint is shown as a dashed arrow from one element to the other element labeled by the constraint string (in braces). The direction of the arrow is relevant information within the constraint.

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

A comment is shown by a text string placed within a note symbol that is attached to a model element. The braces are omitted to show that this is purely a textual comment. (The braces therefore indicate a constraint expressed in some interpretable constraint language.)

4.1.3 Example

Figure 5. Constraints



4.1.4 Mapping

The constraint string maps into the *body* expression in a Constraint element. The mapping depends on the language of the expression, which is known to a tool but generally not displayed on a diagram. If the string lacks braces (i.e., a Comment), then it maps into an expression in the language "text".

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

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

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

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

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

4.2 ELEMENT PROPERTIES

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

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

4.2.1 Semantics

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

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.

4.2.2 Notation

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

A *property specification* has the form

keyword = *value*

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

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

4.2.3 Presentation options

A tool may present property specifications on separate lines 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.

4.2.4 Style guidelines

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

4.2.5 Example

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

{ abstract }

4.2.6 Mapping

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

4.3 STEREOTYPES

4.3.1 Semantics

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

4.3.2 Notation

The general presentation of a stereotype is to use the symbol for the base element but to place a keyword string above the name of the element (if any); the keyword string is 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. Most computers have a Character Map utility. Double angle-brackets may be used as a substitute by the typographically challenged.) The keyword string is generally placed above or in front of the name of the model element being described. The keyword string may also be used as an element in a list, in which case it applies to subsequent list elements until another stereotype string replaces it, or an empty stereotype string («») nullifies it. Note that a stereotype name should not be identical to a predefined keyword applicable to the same element type.

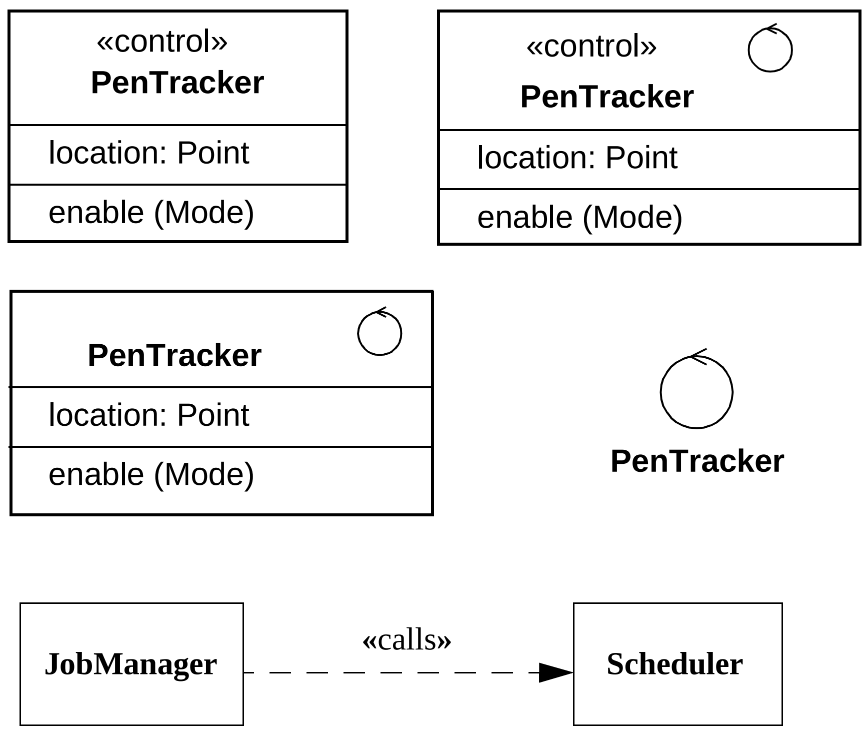
To permit limited graphical extension of the UML notation as well, a graphic icon or a graphic marker (such as texture or color) can be associated with a stereotype. The UML does not specify the form of the graphic specification, but many bitmap and stroked formats exist (and their portability is a difficult problem). The icon can be used in one of two ways: 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). None of the UML symbols *require* the use of such graphic markers. Users *may* use graphic markers freely in their personal work for their own purposes (such as for highlighting within a tool) but should be aware of their limitations for interchange and be prepared to use the canonical forms when necessary.

The classification hierarchy of the stereotypes themselves could be displayed on a class diagram; however, this would be a metamodel diagram and must be distinguished (by user and tool) from an ordinary model diagram. In such a diagram each stereotype is shown as a class with the stereotype «stereotype» (yes, this is a self-referential usage!). Generalization relationships may 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; this is not a capability required by ordinary modelers

4.3.3 Example

Figure 6. Varieties of stereotype notation



4.3.4 Mapping

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

5. STATIC STRUCTURE DIAGRAMS

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

This 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.

5.1 CLASS DIAGRAM

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

5.1.1 Semantics

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

5.1.2 Notation

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

5.1.3 Mapping

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

5.2 OBJECT DIAGRAM

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

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

5.3 CLASSIFER

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

5.4 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 interfaces, signals, or utilities) are notated using keywords on class symbols; some of these are separate metamodel classes and some are stereotypes of Class. Classes are declared in class diagrams and used in most other diagrams. UML provides a graphical notation for declaring and using classes, as well as a textual notation for referencing classes within the descriptions of other model elements.

5.4.1 Semantics

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

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

5.4.2 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.

See the sections on Name Compartment and List Compartment for more details.

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. Compartment names can be used to remove ambiguity, if necessary (Section 5.6.1). A full pathname can be specified by chaining together package names separated by double colons (::).

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

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

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

5.4.4 Style guidelines

(Note that these are recommendations, not mandates.)

Center class name in boldface.

Center stereotype name in plain face within guillemets above class name.

Being class names with an uppercase letter.

Left justify attributes and operations in plain face.

Begin attribute and operation names with a lowercase letter.

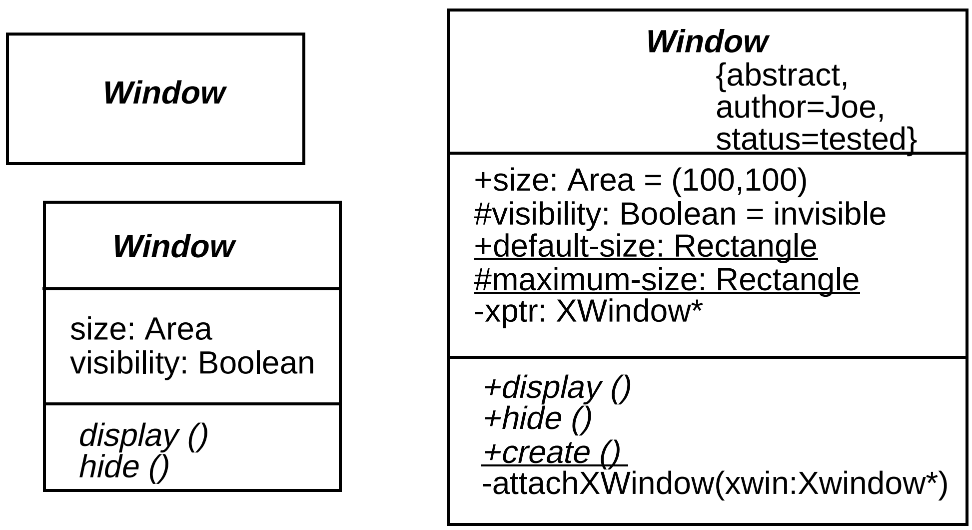
Show the names of abstract classes or the signatures of abstract operations in italics

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

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

5.4.5 Example

Figure 7. Class notation: details suppressed, analysis-level details, implementation-level details



5.4.6 Mapping

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

5.5 NAME COMPARTMENT

5.5.1 Notation

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

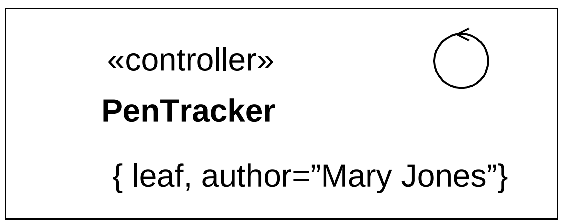
An optional stereotype keyword may be placed above the class name within guillemets, and/or a stereotype icon may be placed in the upper right corner of the compartment. The stereotype name must not match a predefined keyword.

The name of the class appears next. If the class is abstract, its name appears in italics. But note that any explicit specification of generalization status take precedence over the name font.

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

The stereotype and property list are optional.

Figure 8. Name compartment



5.5.2 Mapping

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

5.6 List Compartment

5.6.1 Notation

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

The items in the list are ordered and the order may be modified by the user. The order of the elements is meaningful information and must be accessible within tools. For example, it may be used by a code generator in generating a list of declarations. The list elements may be presented in a different order, 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. Keyword strings may also be used in a similar way to qualify subsequent list elements.

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

5.6.2 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 25).

5.6.3 Example

Figure 9. Stereotype keyword applied to groups of list elements

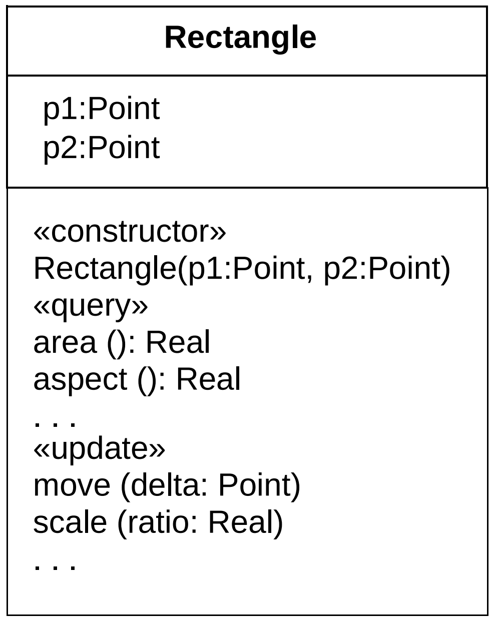
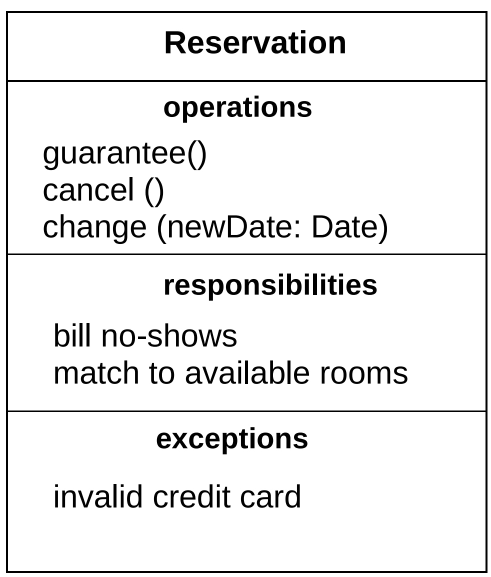


Figure 10. Compartments with names



5.6.4 Mapping

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

5.7 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).

5.7.1 Semantics

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

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

5.7.2 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 or public). A tool should assign visibilities to new attributes even if the visibility is not shown. The visibility marker is a shorthand for a full *visibility* property specification string.

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

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

where *name* is an identifier string that represents the name of the attribute;

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

where *initial-value* is a language-dependent expression for the initial value of a newly created object. The initial value is optional (the equal sign is also omitted). An explicit constructor for a new object may augment or modify the default initial value;

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

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

<u>*class-scope-attribute*</u>

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

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

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

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

name [0..1]: String

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

5.7.3 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).

5.7.4 Style guidelines

Attribute names typically begin with a lowercase letter.

Attribute names in plain face.

5.7.5 Example

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

5.7.6 Mapping

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

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

5.8 OPERATION

Used to show operations defined on classes. Also used to show methods supplied by classes.

5.8.1 Operation

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

5.8.2 Notation

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

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

where *visibility* is one of:

+ public visibility

protected visibility

- private visibility

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

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

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

where *name* is an identifier string;

where *return-type-expression* is a language-dependent specification of the implementation type or types of the value returned by the operation. If the return-type is omitted if the operation does not return a value (C++ void). A list of expressions may be supplied to indicate multiple return values.

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

kind name : *type-expression* = *default-value* where *kind* is **in, out,** or **inout**, with the default **in** if absent;

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

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

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

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

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

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

The concurrency semantics of an operation are specified by a property string with one of the names: *sequential*, *guarded*, *concurrent*. In the absence of a specification the concurrency semantics are undefined and must be assumed to be sequential in the worst case.

The top-most appearance of an operation signature declares the operation on the class (and therefore inherited by all of its descendents). If this class does not implement the operation (i.e., does not supply a method) then the operation may be marked as "{abstract}" or the operation signature may be italicized to indicate that it is abstract. Any subordinate appearances of the operation signature indicate that the subordinate class implements a method on the operation. (The specification of "{abstract}" or italics on a subordinate class would not indicate a method but this usage of the notation would be poor form.)

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

An operation entry with the stereotype «signal» indicates that the class accepts the given signal. The syntax is identical to that of an operation.

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

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

5.8.3 Presentation options

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

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

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

5.8.4 Style guidelines

Operation names typically begin with a lowercase letter.

Operation names in plain face.

An abstract operation may be shown in italics.

5.8.5 Example

Figure 11. Operation list with a variety of operations

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

5.8.6 Mapping

A string entry within the operation compartment maps into an Operation or a Method within the Class representing the class symbol. The properties of the operation map in accord with the preceding descriptions. See the description of Attribute for additional details.

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

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

5.8.7 Signal reception

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

5.9 TYPE VS. IMPLEMENTATION CLASS

5.9.1 Semantics

Classes can be specialized by stereotypes into Types and Implementation Classes (although they can be left undifferentiated as well). A Type characterizes a changeable role that an object may adopt and later abandon. An Implementation Class defines the physical data structure and procedures of an object as implemented in traditional languages (C++, Smalltalk, etc.). An object may have multiple Types (which may change dynamically) but only one ImplementationClass (which is fixed). Although the usage of Types and ImplementationClasses is different, their internal structure is the same, so they are modeled as stereotypes of Class. All kinds of Class require that a subclass fully support the features of the superclass, including support for all inherited attributes, associations, and operations.

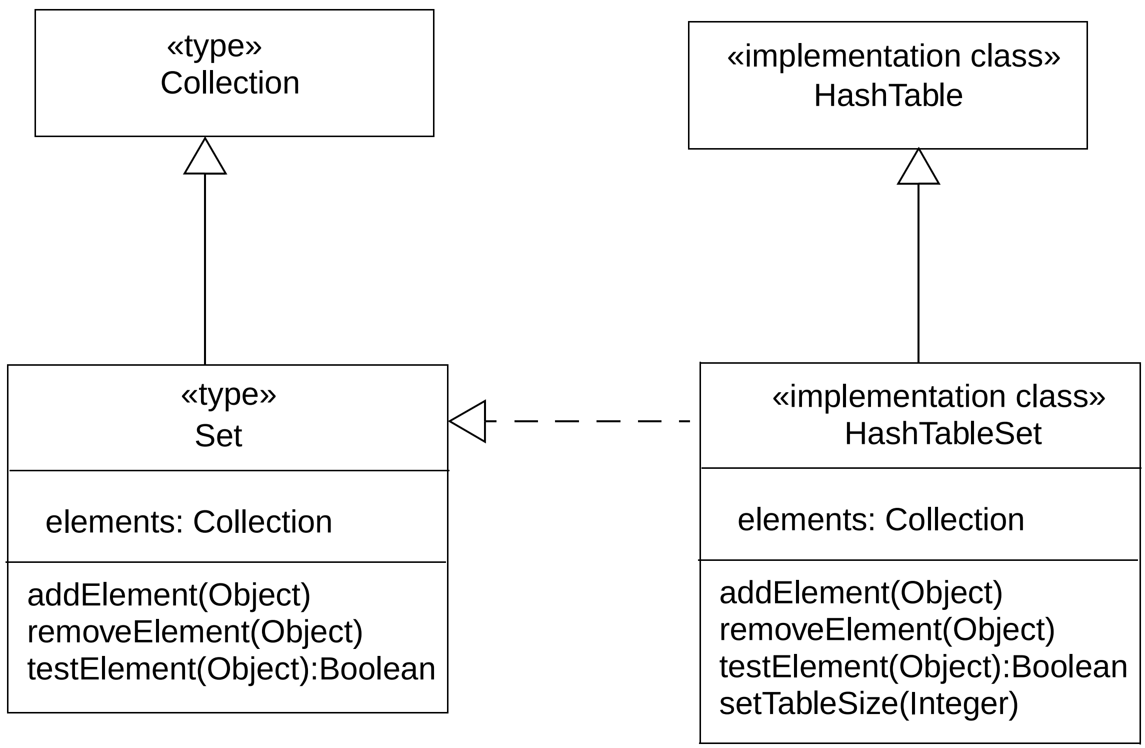
5.9.2 Notation

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

The implementation of a type by an implementation class is modeled as the Realizes relationship, shown as a dashed line with a solid triangular arrowhead (a dashed "generalization arrow"). This symbol implies inheritance of operations but not of structure (attributes or associations).

5.9.3 Example

Figure 12. Notation for types and implementation classes



5.9.4 Mapping

A class symbol with a stereotype (including “type” and “implementation class”) maps into a Class with the corresponding stereotype. A class symbol without a stereotype maps into a Class with the default stereotype for the diagram (if a default has been defined by the modeler or tool), otherwise it maps into a Class with no stereotype. This symbol is normally used between a class and an interface but may also be used between any two classifiers to show inheritance of operations only without inheritance of attributes or associations.

5.10 INTERFACES

5.10.1 Semantics

An interface is a specifier for the externally-visible operations of a class, component, or other entity (including summarization units such as packages) without specification of internal structure. Each interface often specifies only a limited part of the behavior of an actual class. Interfaces do not have implementation; they lack attributes, states, or associations; they only have operations. Interfaces may have generalization relationships. An interface is formally equivalent to an abstract class with no attributes and no methods and only abstract operations, but Interface is a peer of Class within the UML metamodel; both are Classifiers.

5.10.2 Notation

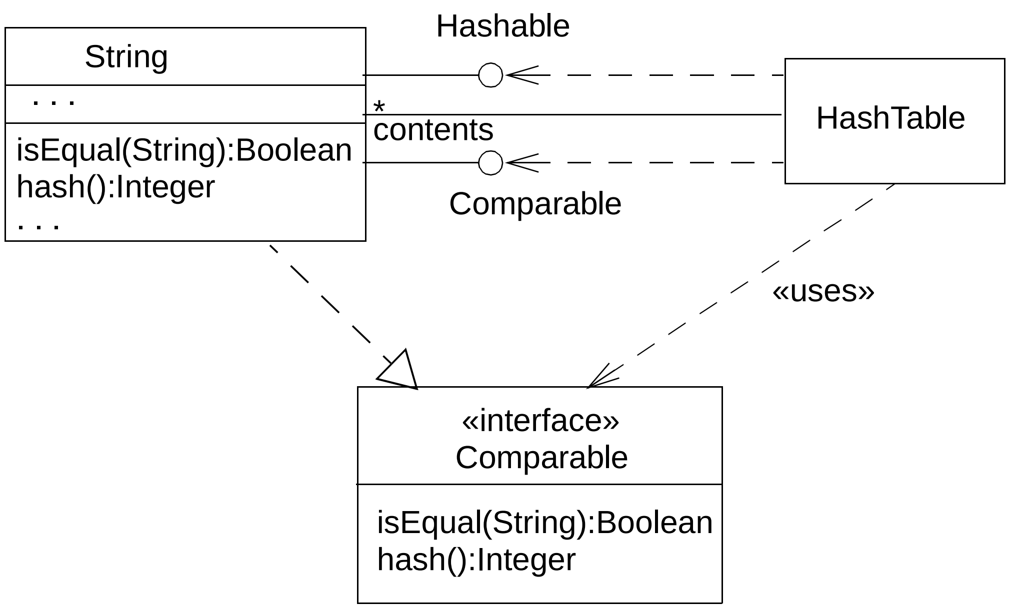
An interface is a Classifer and may also be shown using the full rectangle symbol with compartments and the keyword «interface». A list of operations supported by the interface is placed in the operation compartment. The attribute compartment may be omitted because it is always empty.

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

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

5.10.3 Example

Figure 13. Interface notation on class diagram



5.10.4 Mapping

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

A dashed generalization arrow from a class symbol to an interface symbol or a solid line connecting a class symbol and an interface circle maps into a realization-specification relationship between the corresponding Class and Interface elements. A dependency arrow from a class symbol to an interface symbol maps into a «uses» dependency between the corresponding Class and Interface.

5.11 PARAMETERIZED CLASS (TEMPLATE)

5.11.1 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 directly-usable class because it has unbound parameters. Its parameters must be bound to actual values to create a bound form that is a class. Only a class can be s superclass or the target of an association (a one-way association *from* the template *to* another class is permissible, however). A template may be a subclass of an ordinary class; this implies that all classes formed by binding it are subclasses of the given superclass.

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

5.11.2 Notation

A small dashed rectangle is superimposed on the upper right-hand corner of the rectangle for the class (or to the symbol for another modeling element). The dashed rectangle contains 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 also include occurrences of the formal parameters. Occurrences of the formal parameters can also occur inside of a context for the class, for example, to show a related class identified by one of the parameters

5.11.3 Presentation options

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

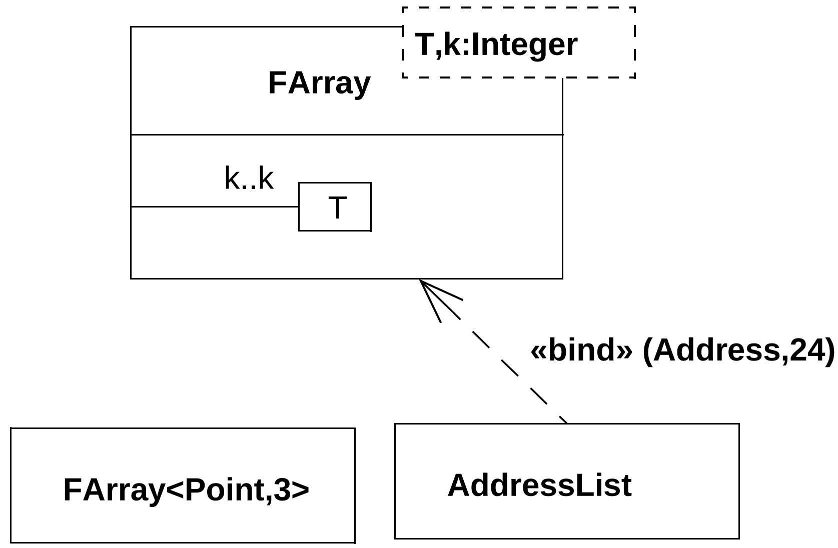
Parameters are restricted attributes, shown as strings with the syntax

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

If the type name is omitted, it is assumed to be a type expression that resolves to a classifier, such as a class name or a data type. Other parameter types (such as Integer) must be explicitly shown; they must resolve to valid type expressions.

5.11.4 Example

Figure 14. Template notation with use of parameter as a reference



5.11.5 Mapping

The addition of the template dashed box to a symbol causes the addition of the parameter names in the list as ModelElements within the Namespace of the ModelElement corresponding to the base symbol. Each of the parameter ModelElements has the templateParameter association to the Namespace.

5.12 BOUND ELEMENT

5.12.1 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.

5.12.2 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 Dependency relationship with the keyword «bind». The arguments are shown in parentheses after the keyword. In this case the bound form may be given a name distinct from the template.

5.12.3 Style guidelines

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

5.12.4 Example

See Figure 14.

5.12.5 Mapping

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

5.13 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.

5.13.1 Semantics

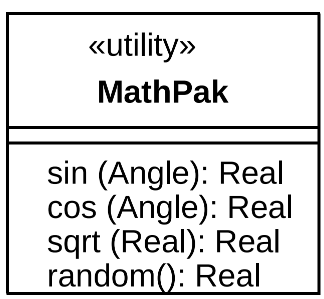
The instance-scope attributes and operations of a utility are interpreted as global attributes and operations. It is inappropriate for a utility to declare class-scope attributes and operations because the instance-scope members are already interpreted as being at class scope.

5.13.2 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.

5.13.3 Example

Figure 15. Notation for utility



5.13.4 Mapping

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

5.14 METACLASS

5.14.1 Semantics

A metaclass is a class whose instances are classes.

5.14.2 Notation

Shown as the stereotype «metaclass» of Class.

5.14.3 Mapping

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

5.15 CLASS PATHNAMES

5.15.1 Notation

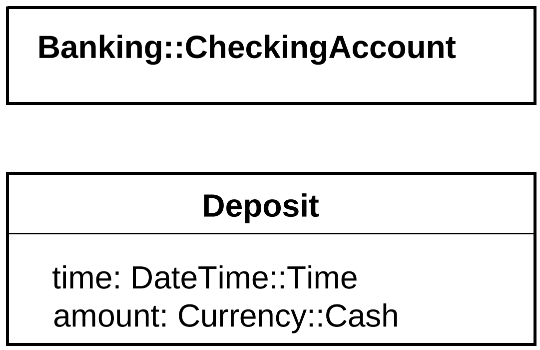
Class symbols (rectangles) serve to define a class and its properties, such as relationships to other classes. A reference to a class in a different package is notated by using a pathname for the class, in the form:

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.

5.15.2 Example

Figure 16. Pathnames for classes in other packages



5.15.3 Mapping

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

5.16 IMPORTING A PACKAGE

5.16.1 Semantics

A class in another package may be referenced. On the package level, the «imports» dependency indicates that the contents of the target packages may be referenced by the client package or packages recursively embedded within it. The target references must have visibility sufficient for the referents. Visibilities may be specified on model elements and on packages. If a model element is nested inside one or more packages, the visibilities of the element and all of its containers combine according to the rule that the most restrictive visibility in the set is obtained. It is not possible to selectively export certain elements from within a nested package; the visibility of the outer package is applied to each element exported by an inner package. Imports are recursive within nested levels of packages. A descendent of a class requires at least “protected” visibility; any other class requires “public” visibility. (See the semantics document for full details.)

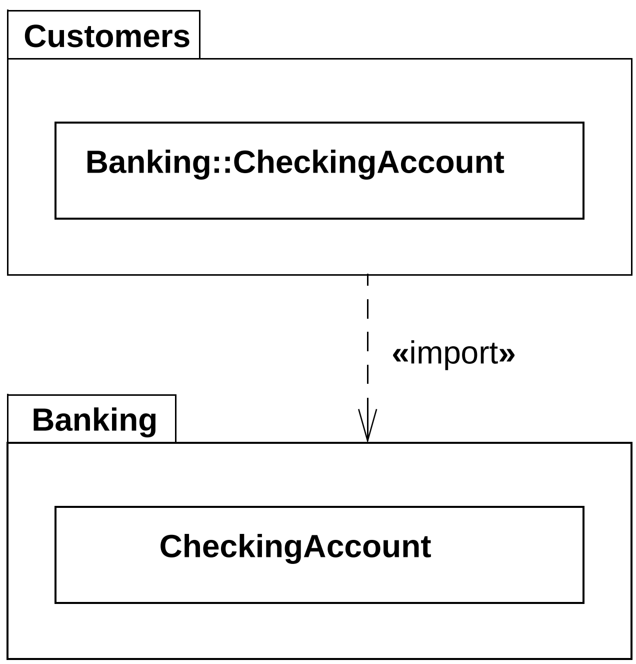
Note that an imports dependency does not modify the namespace of the client or in any other way automatically create references; it merely grants permission to establish references. Note also that a tool could automatically create imports dependencies for users if desired when references are created.

5.16.2 Notation

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

5.16.3 Example

Figure 17. Imports dependency among packages



5.16.4 Mapping

This is not a special symbol. It maps into a Dependency with the stereotype «import» between the two packages.

5.17 OBJECT

5.17.1 Semantics

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

5.17.2 Notation

The object notation is derived from the class notation by underlining instance-level elements, as explained in the general comments in Section 2.11.

An object shown as a rectangle with two compartments.

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

objectname : *classname*

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

`display_window: WindowingSystem::GraphicWindows::Window`

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

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

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

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

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

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

attributename : *type* = *value*

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

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

5.17.3 Presentation options

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

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

The attribute value compartment as a whole may be suppressed.

Attributes whose values are not of interest may be suppressed.

Attributes whose values change during a computation may show their values as a list of values held over time. 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.

5.17.4 Style guidelines

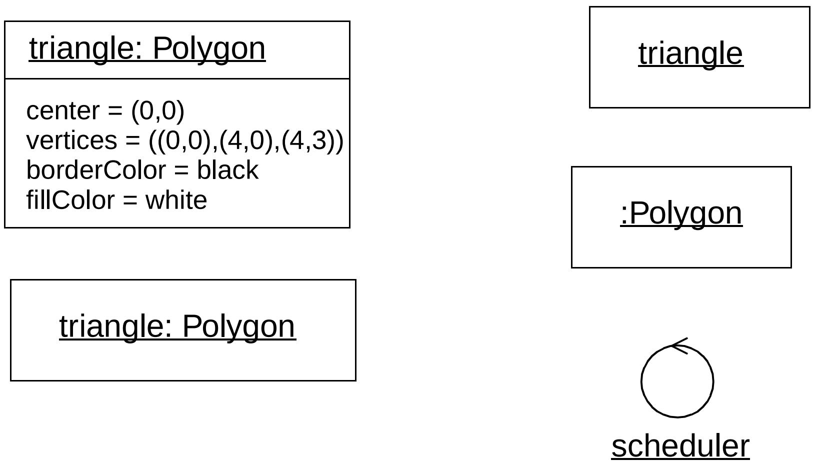
Objects may be shown on class diagrams. The elements on collaboration diagrams are not objects, because they describe many possible objects; they are instead roles that may be held by object. Objects in class diagrams serve mainly to show examples of data structures.

5.17.5 Variations

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

5.17.6 Example

Figure 18. Objects



5.17.7 Mapping

The mapping of an object symbol depends on the diagram:

Within a collaboration, it maps into a ClassifierRole of the corresponding Collaboration. The role has the name specified by the *objectname* portion of the symbol name string. The ClassifierRole has a type association to the Class whose name appears in the *classname* part of the symbol name string.

In an object diagram or within an ordinary class diagram, it maps into an Object of the Class given by the *classname* part of the name string. The values of the attributes are given by the value expressions in the attribute list in the symbol.

5.18 COMPOSITE OBJECT

5.18.1 Semantics

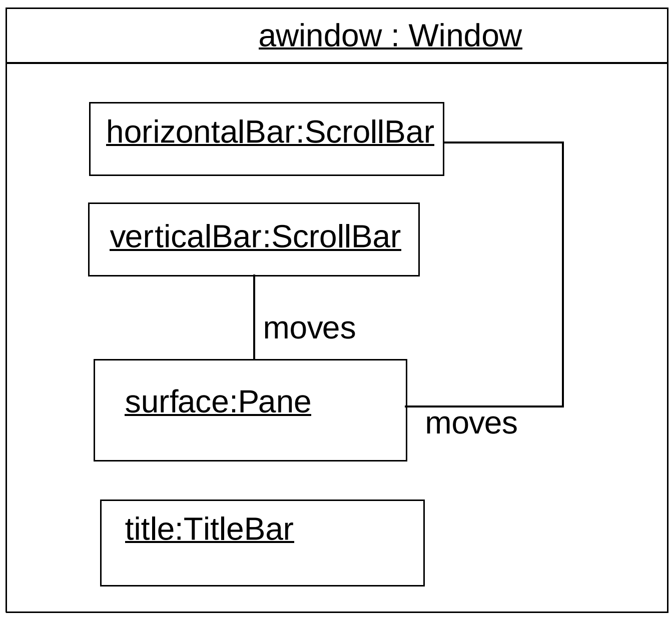
A composite object represents a high-level object made of tightly-bound parts. This is an instance of a composite class, which implies the composition aggregation between the class and its parts. A composite object is similar to (but simpler and more restricted than) a collaboration, but it is defined completely by composition in a static model.

5.18.2 Notation

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

5.18.3 Example

Figure 19. Composite object



5.18.4 Mapping

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

5.19 ASSOCIATION

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

5.20 BINARY ASSOCIATION

5.20.1 Semantics

A binary association is an association among exactly two classes (including the possibility of a reflexive association from a class to itself).

5.20.2 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*.

Un a binary association both ends may attach to the same class. The links of such an association may connect two different objects from the same class or one object to itself. The latter case is a *reflexive* association; it may be forbidden by a constraint if necessary.

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 (but not near enough to an end to be confused with a rolename). The name string may have an optional small black solid triangle in it; the point of the triangle indicates the direction in which to read the name. The name-direction arrow has no semantics significance; it is purely descriptive. The 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.

5.20.3 Presentation options

When two paths cross, the crossing may optionally be shown with a small semicircular jog to indicate that the paths do not intersect (as in electrical circuit diagrams). Alternately crossing can be unmarked but connections might be shown by small dots.

5.20.4 Style guidelines

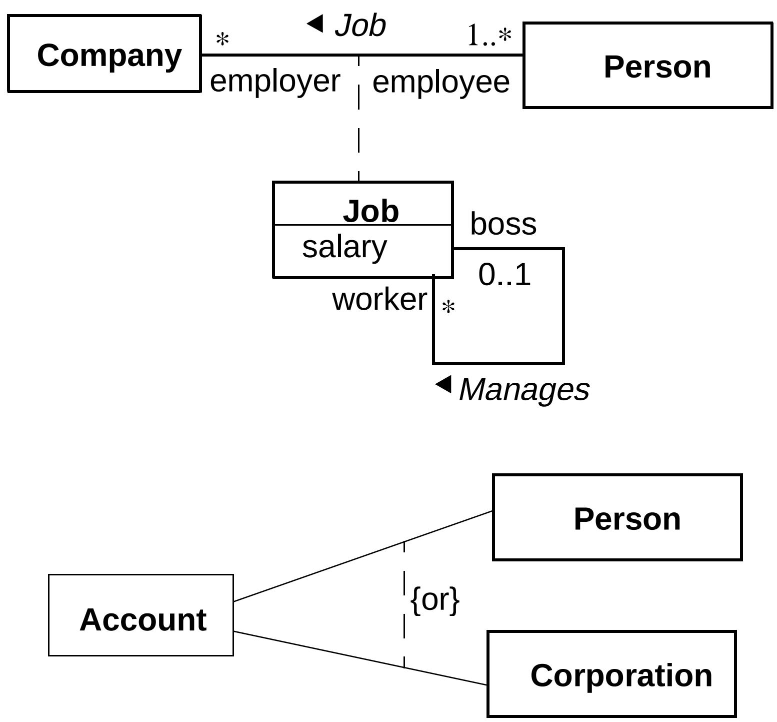
Lines may be drawn using various styles, including orthogonal segments, oblique segments, and curved segments. The choice of a particular set of line styles is a user choice.

5.20.5 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. Each rolename must be different. (This is simply a predefined use of the constraint notation.)

5.20.6 Example

Figure 20. Association notation



5.20.7 Mapping

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

5.21 ASSOCIATION END

5.21.1 Semantics

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

5.21.2 Notation

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

> multiplicity – specified by a text syntax, see detail section. Multiplicity may be suppressed on a particular association or for an entire diagram. In an incomplete model the multiplicity may be unspecified in the model itself, in 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. If no indication is given, then it is unordered (the elements form a set). Various kinds of ordering can be specified as a constraint on the association end. The declaration does not specify how the ordering is established or maintained; operations that insert new elements must make provision for specifying their position either implicitly (such as at the end) or explicitly. Possible values include:
>
> > unordered — the elements form an unordered set. This is the default and need not be shown explicitly.
> >
> > ordered — the elements of the set are ordered into a list. It is still a set and duplicates are prohibited. This generic specification includes all kinds of ordering. This may be specified by the keyword syntax: “{ordered}”.
>
> An ordered relationship may be implemented in various ways but this is normally specified as a language-specified code generation property to select a particular implementation. An implementation extension might substitute the data structure to hold the elements for the generic specification “ordered”.
>
> At implementation level, sorting may also be specified. It does not add new semantic information but it expresses a design decision:
>
> > sorted — the elements are sorted based on their internal values. The actual sorting rule is best specified as a separate constraint.
>
> 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. To be totally explicit arrows may be shown whenever navigation is supported in a given direction. In practice it is often convenient to sup-

press some of the arrows and just show exceptional situations. See the presentation options for details.

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.

interface specifier

The name of a Classifier with the syntax

‘:’ *classifiername*

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

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

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

changeability

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

visibility

Specified by a visibility indicator (‘+’, ‘#’, ‘-’ or explicit keyword such as {public}) in front of the rolename. Specifies the visibility of the association traversing in the direction toward the given rolename. See Section 5.7 for details of visibility specification.

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.

5.21.3 Presentation options

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

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

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

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

Presentation 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.

5.21.4 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

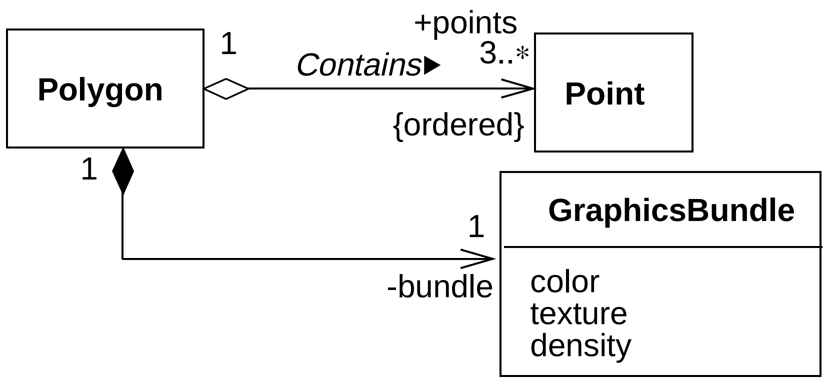
aggregation symbol

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").

5.21.5 Example

Figure 21. Various adornments on association roles



5.21.6 Mapping

The adornments on the end of an association path map into properties of the corresponding role of the Association. In general, implications cannot be drawn from the absence of an adornment (it may simply be suppressed) but see the preceding descriptions for details.

5.22 MULTIPLICITY

5.22.1 Semantics

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

5.22.2 Notation

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

lower-bound .. *upper-bound*

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

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

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

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

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

5.22.3 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”.

5.22.4 Example

0..1

1

0..*

*

1..*

1..6

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

5.22.5 Mapping

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

5.23 QUALIFIER

5.23.1 Semantics

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

5.23.2 Notation

A qualifier is shown as a small rectangle attached to the end of an association path between the final path segment and the symbol of the 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 (i.e., every target falls into exactly one partition).

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.

5.23.3 Presentation options

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

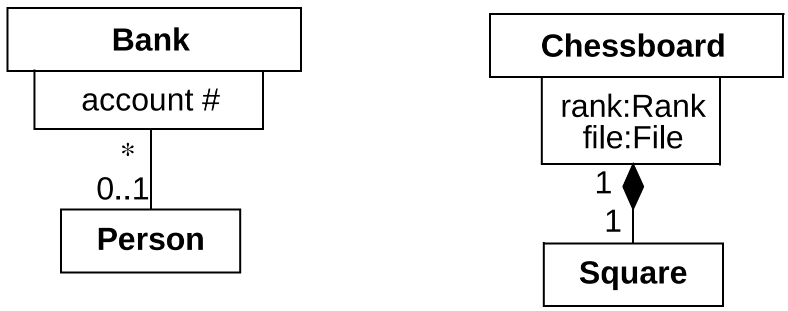
A tool may use a lighter line for qualifier rectangles than for class rectangles to distinguish them clearly.

5.23.4 Style guidelines

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

5.23.5 Example

Figure 22. Qualified associations



5.23.6 Mapping

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

5.24 ASSOCIATION CLASS

5.24.1 Semantics

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

5.24.2 Notation

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

5.24.3 Presentation options

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

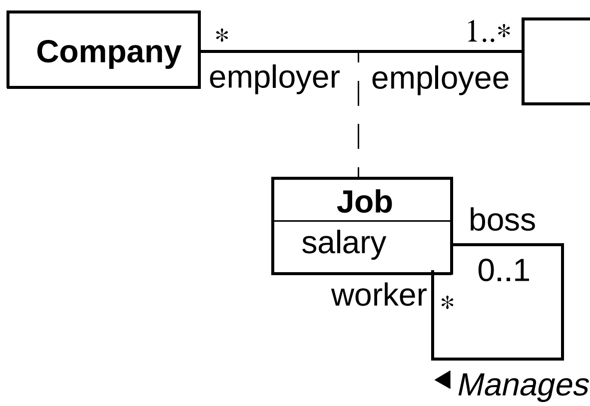
5.24.4 Style guidelines

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

5.24.5 Example

Figure 23. Association class



5.24.6 Mapping

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

5.25 N-ARY ASSOCIATION

5.25.1 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.

5.25.2 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.

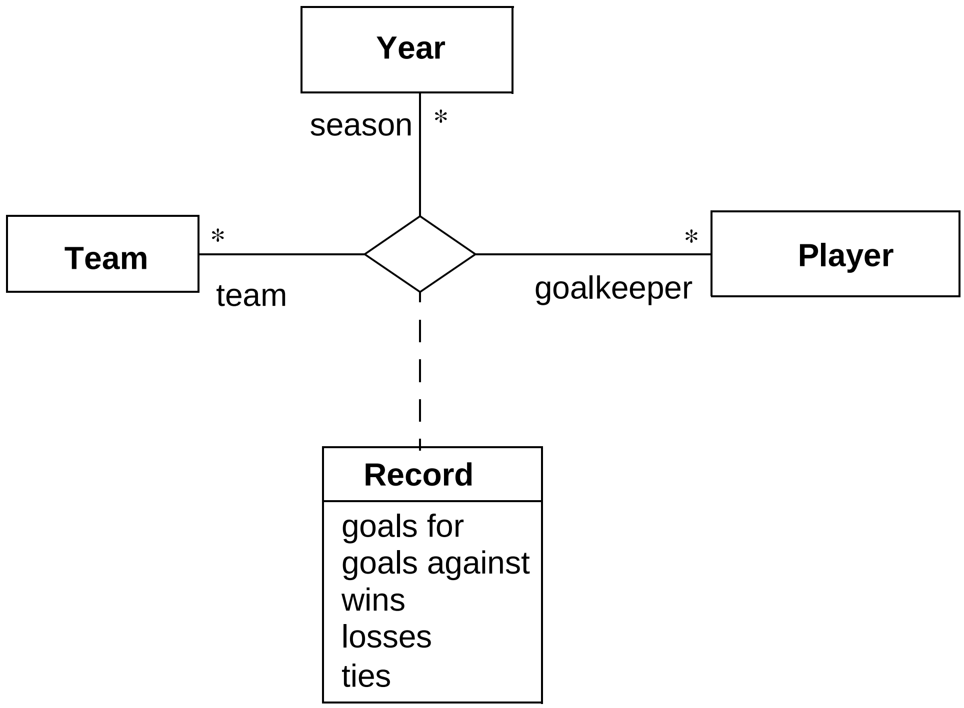
5.25.3 Style guidelines

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

5.25.4 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 24. Ternary association that is also an association class



5.25.5 Mapping

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

5.26 COMPOSITION

5.26.1 Semantics

Composition is a form of aggregation with strong ownership and coincident lifetime of part with the whole. The multiplicity of the aggregate end may not exceed one (it is unshared). See the semantics document for further details.

The parts of a composition may include classes and associations. The meaning of an association in a composition is that any tuple of objects connected by a single link must all belong to the *same* container object.

5.26.2 Notation

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.

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

rolename ‘:’ *classname*

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

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

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

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

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

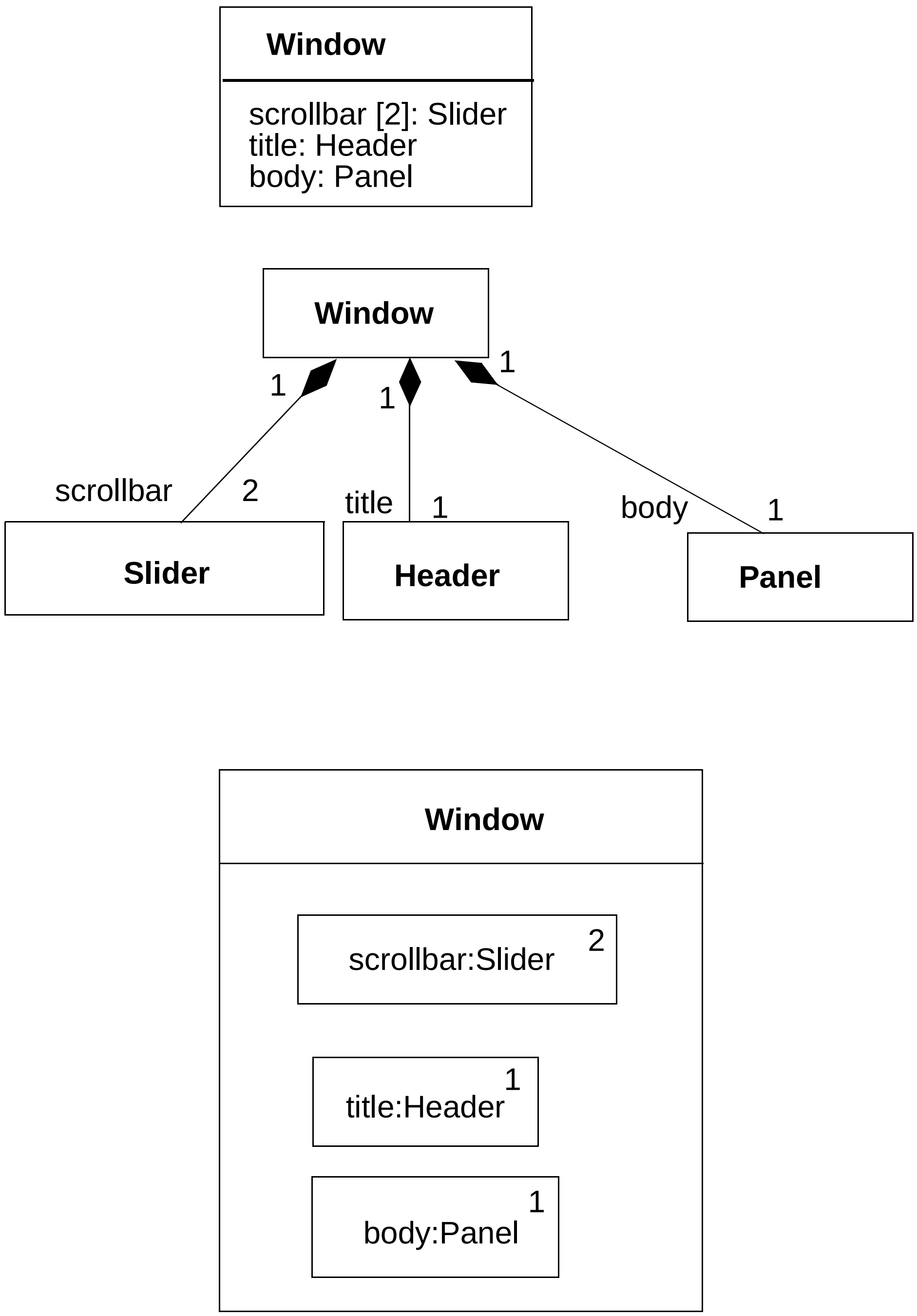
5.26.3 Design guidelines

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.

5.26.4 Example

Figure 25. Different ways to show composition



5.26.5 Mapping

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

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

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

5.27 LINKS

5.27.1 Semantics

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

5.27.2 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:

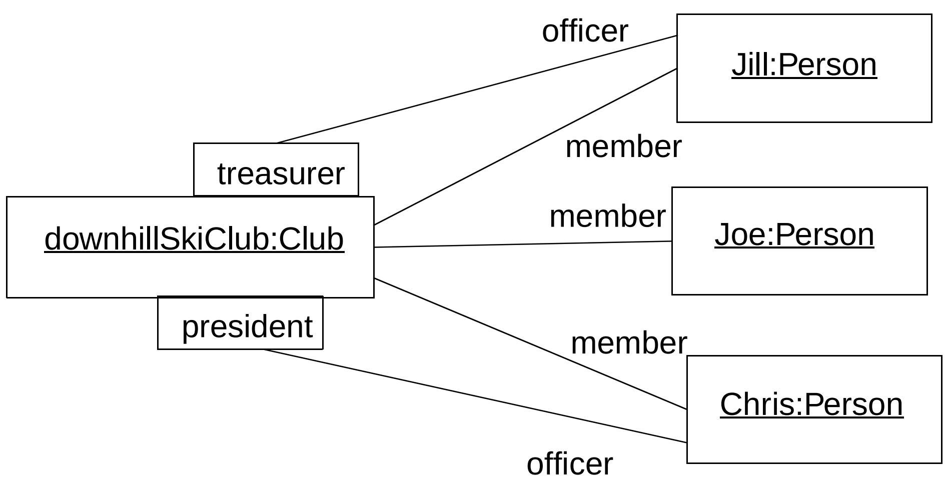
«association»	association (default, unnecessary to specify except for emphasis)
«parameter»	procedure parameter

«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.

5.27.3 Example

Figure 26. Links



5.27.4 Mapping

The mapping depends on the kind of diagram:

Within a collaboration diagram, each link path maps to an AssociationRole between the ClassifierRoles corresponding to the connected class boxes. If a name is placed on the link path, then it is the name of the Association that is the type of the AssociationRole. Stereotypes on the path indicate the form of the relationship within the collaboration.

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

5.28 GENERALIZATION

5.28.1 Semantics

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.

5.28.2 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
>
> where *discriminator* is the name of a partition of the subtypes of the superclass. The subclass is declared to be in the given partition. The absence of a discriminator label indicates the “empty string” discriminator which is a valid value (the “default” discriminator).

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

5.28.3 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.

5.28.4 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. This is a notational convention that information has been suppressed, not a semantic statement)

Predefined constraints may be used to indicate semantic constraints among the 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:

The following constraints are predefined:

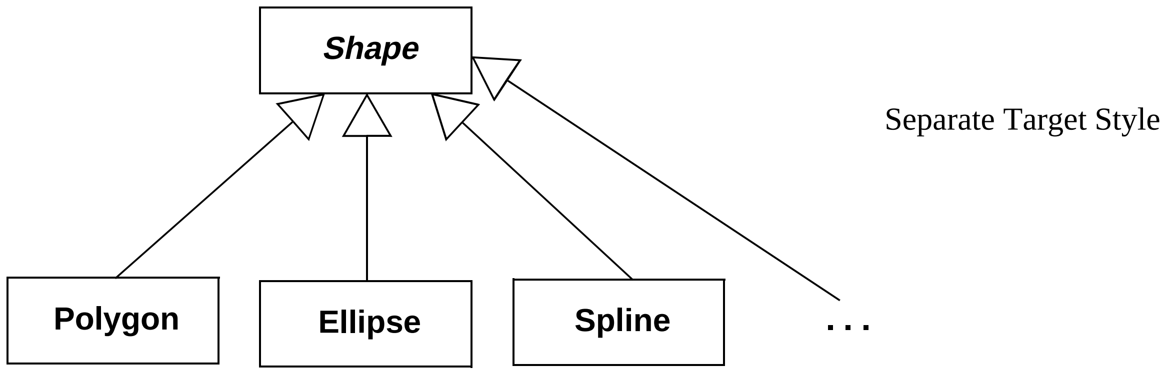
overlapping	A descendent may be descended from more than one of the subclasses.
disjoint	A descendent may not be descended from more than one of the subclasses.
complete	All subclasses have been specified (whether or not shown). No additional subclasses are expected.
incomplete	Some subclasses have been 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.

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

5.28.5 Example

Figure 27. Styles of displaying generalization



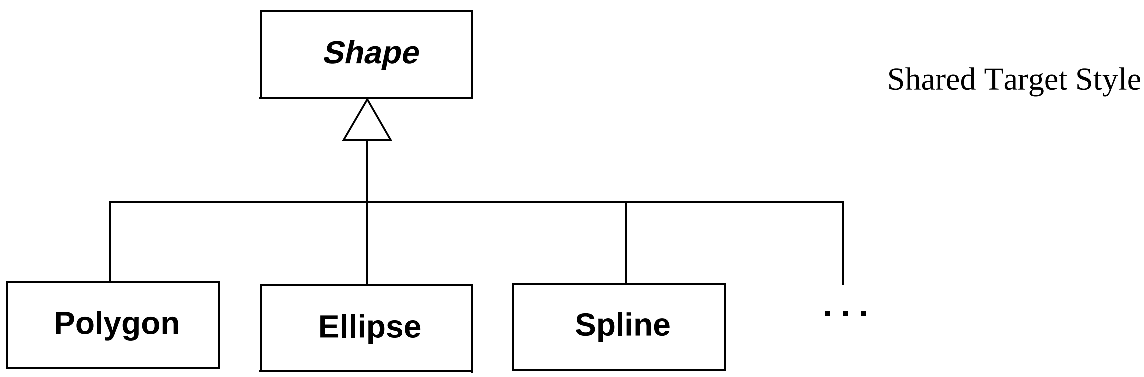


Figure 28. Generalization with discriminators and constraints, separate target style

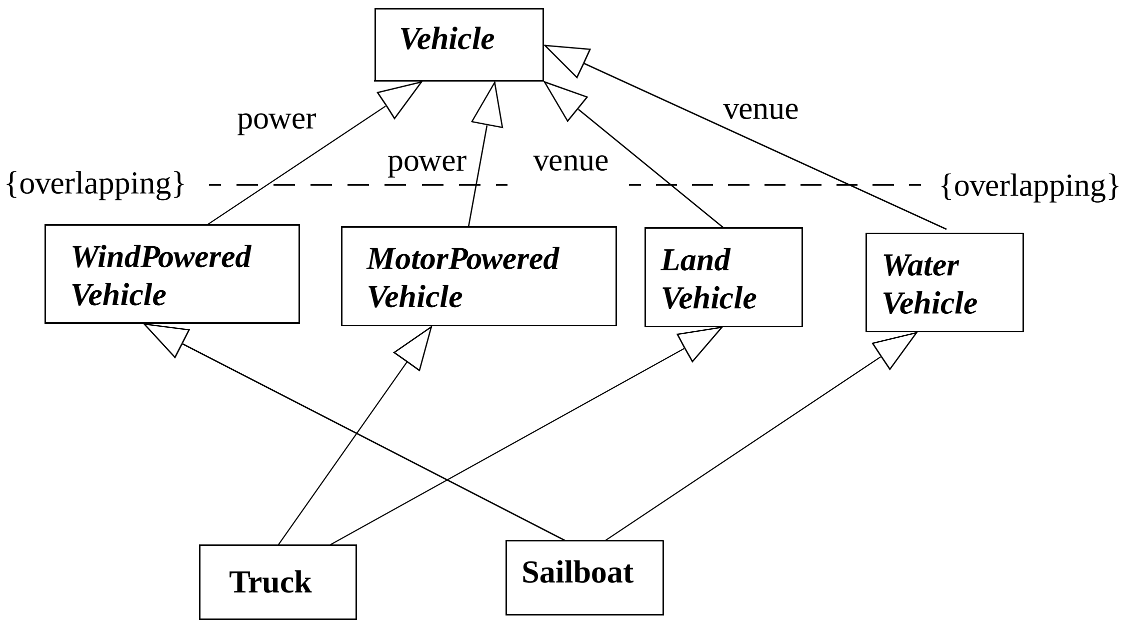
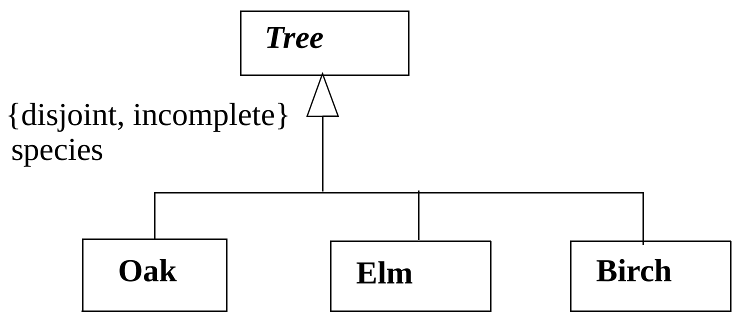


Figure 29. Generalization with shared target style



5.28.6 Mapping

Each generalization path between two class boxes maps into a Generalization between the corresponding Classes. A generalization tree with one arrowhead and many tails maps into a set of Generalizations, one between each Class corresponding to a class box on a tail and the single Class corresponding to the class box on the head. That is, a tree is semantically indistinguishable from a set of distinct arrows; it is purely a notational convenience.

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

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

5.29 DEPENDENCY

5.29.1 Semantics

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.

5.29.2 Notation

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

The following kinds of Dependency are predefined and may be indicated with keywords:

trace – Trace: a historical connection between two elements that represent the same concept at different levels of meaning

refine – Refinement: a historical or derivation connection between two elements with a mapping (not necessarily complete) between them. A description of the mapping may be attached to the dependency in a note. Various kinds of refinement have been proposed and can be indicated by further stereotyping.

uses – Usage: a situation in which one element requires the presence of another element for its correct implementation or functioning. May be stereotyped further to indicate the exact nature of the dependency, such as calling an operation of another class, granting permission for access, instantiating an object of another class, etc.

bind – Binding: a binding of template parameters to actual values to create a nonparameterized element. See Section 5.12 for more details.

5.29.3 Presentation options

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

5.29.4 Example

Figure 30. Various usage dependencies among classes

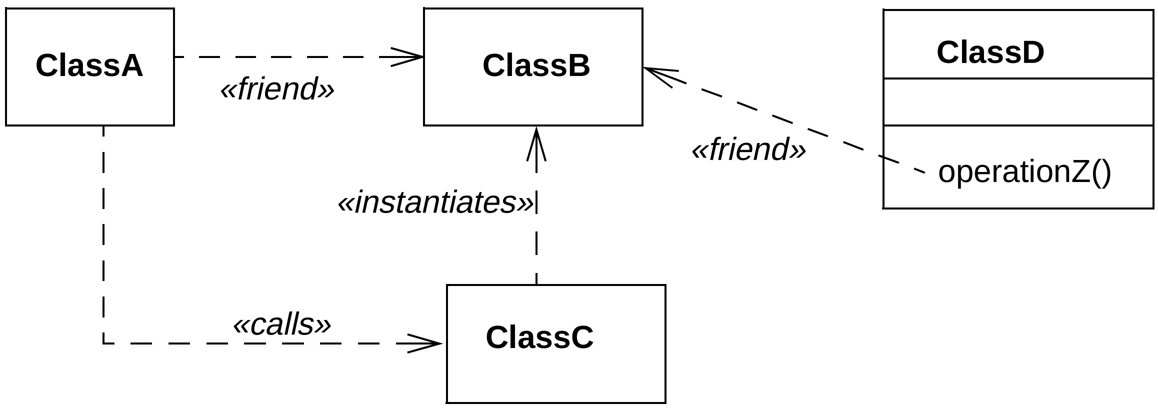
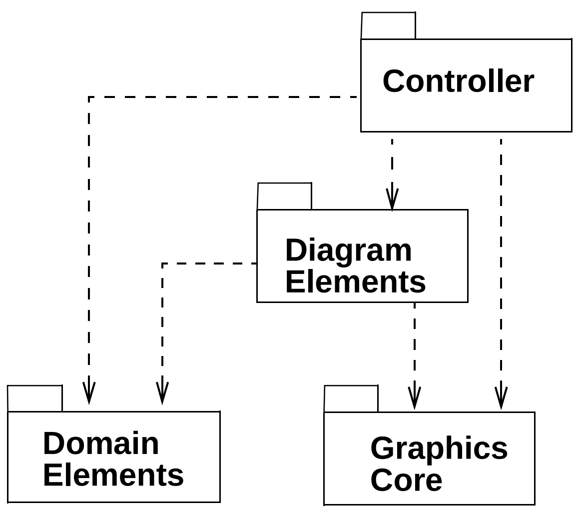


Figure 31. Dependencies among packages



5.29.5 Mapping

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

5.30 DERIVED ELEMENT

5.30.1 Semantics

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

5.30.2 Notation

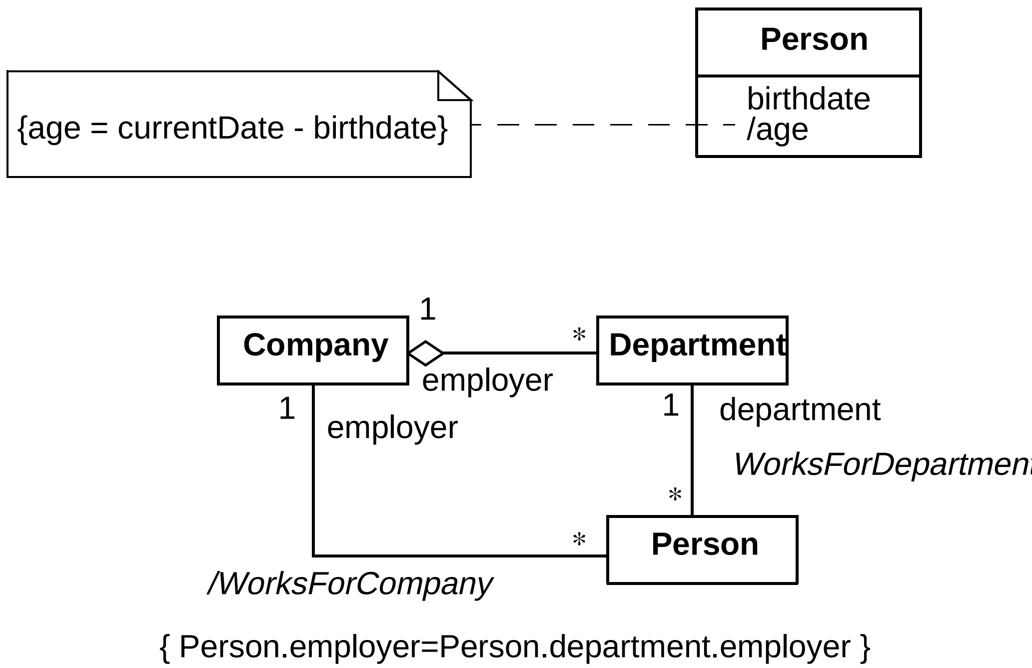
A derived element is shown by placing a slash (/) in front of the name of the derived element, such as an attribute or a rolename.

5.30.3 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.

5.30.4 Example

Figure 32. Derived attribute and derived association



5.30.5 Mapping

The presence of a derived adornment (a leading “/” on the symbol name) on a symbol maps into the setting of the “derived” property of the corresponding Element.

6. USE CASE DIAGRAMS

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

6.1 USE CASE DIAGRAM

6.1.1 Semantics

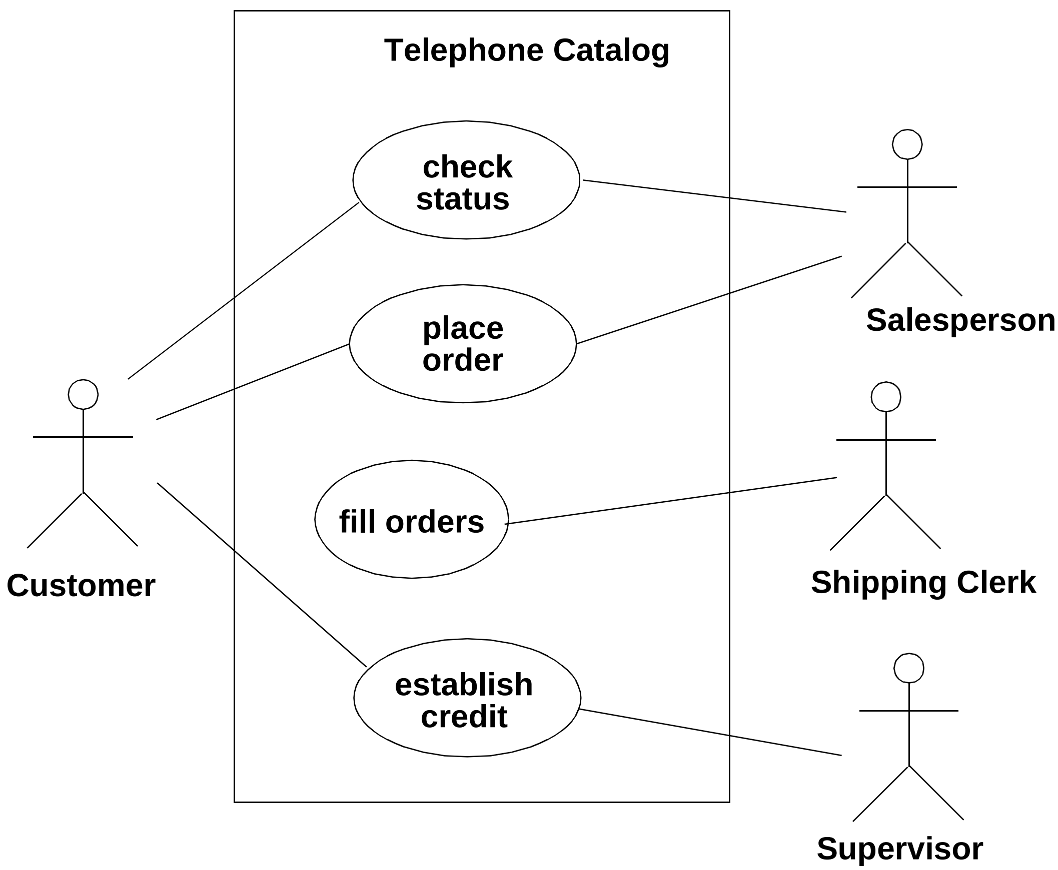
Use case diagrams show elements from the use case model. The use case model represents functionality of a system or a class as mainfested to external interactors with the system.

6.1.2 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.

6.1.3 Example

Figure 33. Use case diagram



6.1.4 Mapping

A set of use case ellipses within a box with connections to actor symbols maps to a single UseCase-Model package containing a set of UseCases and Actors with relationships among them.

6.2 USE CASE

6.2.1 Semantics

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

6.2.2 Notation

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

An *extension point* is a location within a use case at which action sequences from other use cases may be inserted. Each extension point must have a unique name within a use case. Extension points may be listed in a compartment of the use case with the heading **extension points**.

6.2.3 Presentation options

The name of the use case may be placed below the ellipse.

6.2.4 Style guidelines

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

6.2.5 Mapping

A use case symbol maps to a UseCase with the given name (if any).

An extension point maps into an ExtensionPoint within the UseCase.

6.3 ACTOR

6.3.1 Semantics

An actor is a role of object or objects outside of a system that interacts directly with it as part of a coherent work unit (a use case). An Actor element characterizes the role played by an outside object; one physical object may play several roles and therefore be modeled by several actors.

6.3.2 Notation

An actor may be shown as a class rectangle with the stereotype “actor”. The standard stereotype icon for an actor is the “stick man” figure with the name of the actor below the figure.

6.3.3 Style guidelines

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

6.3.4 Mapping

An actor symbol maps to an Actor with the given name.

6.4 USE CASE RELATIONSHIPS

6.4.1 Semantics

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

Communicates – The participation of an actor in a use case. This is the only relationship between actors and use cases.

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 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.

6.4.2 Notation

The communication relationship between an actor and a use case is shown as a solid line between the actor and the use case.

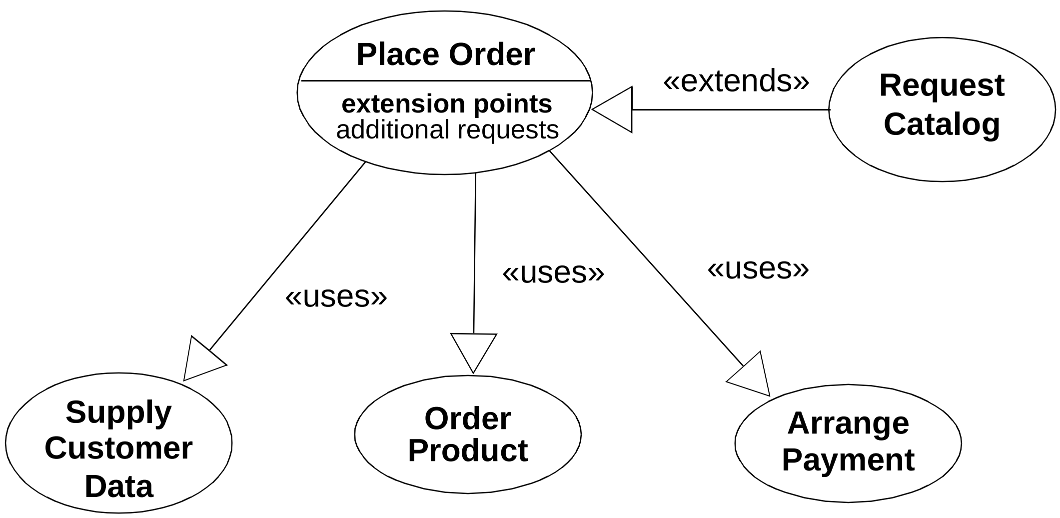
An “extends” relationship between use cases is shown by a generalization arrow from the use case providing the extension to the base use case. The arrow is labeled with the stereotype «extends».

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».

The relationship between a use case and its external interaction sequences are usually shown by an invisible hyperlink to sequence diagrams. The relationship between a use case and its implementation may be shown as a refinement relationship to a collaboration 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.

6.4.3 Example

Figure 34. Use case relationships



6.4.4 Mapping

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

7. SEQUENCE DIAGRAMS

7.1 KINDS OF INTERACTION 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 exchange 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.

7.2 SEQUENCE DIAGRAM

7.2.1 Semantics

A sequence diagram represents an Interaction, which is a set of messages exchanged among objects within a collaboration to effect a desired operation or result.

7.2.2 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.

See subsequent sections for details of the contents of a sequence 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.)

7.2.3 Presentation options

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

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

Various labels (such as timing 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.

7.2.4 Example

Figure 35. Simple sequence diagram with concurrent objects

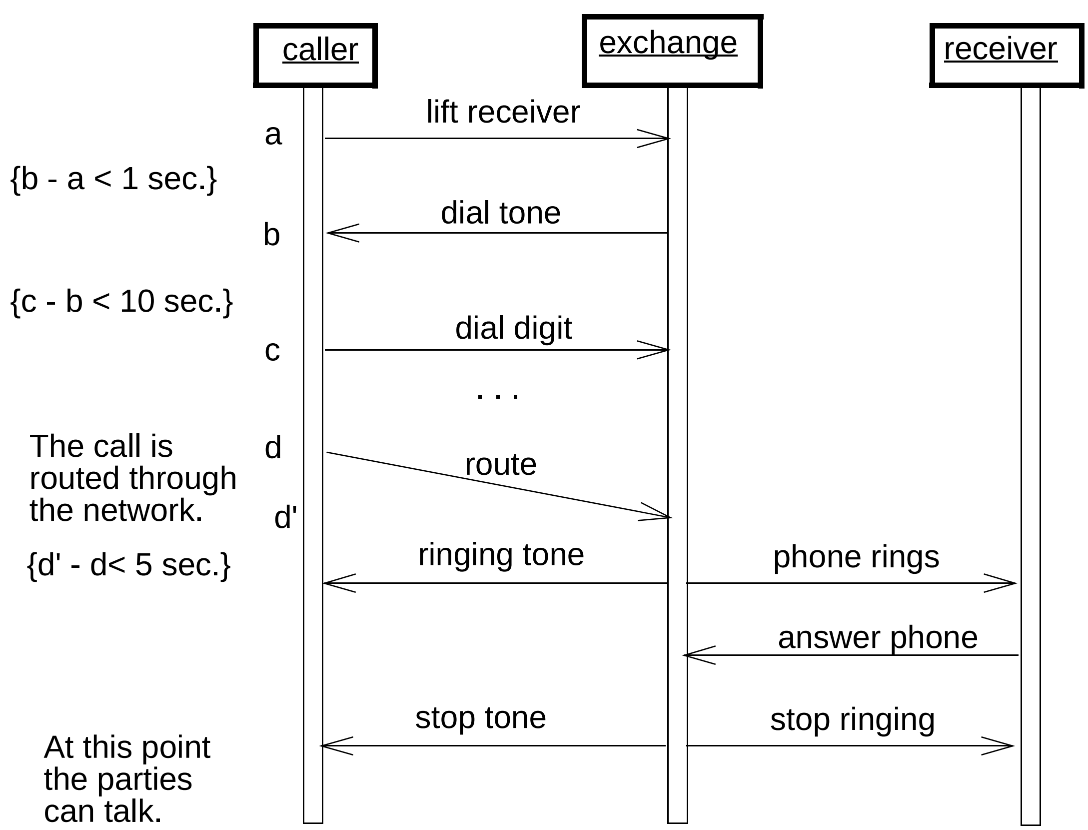
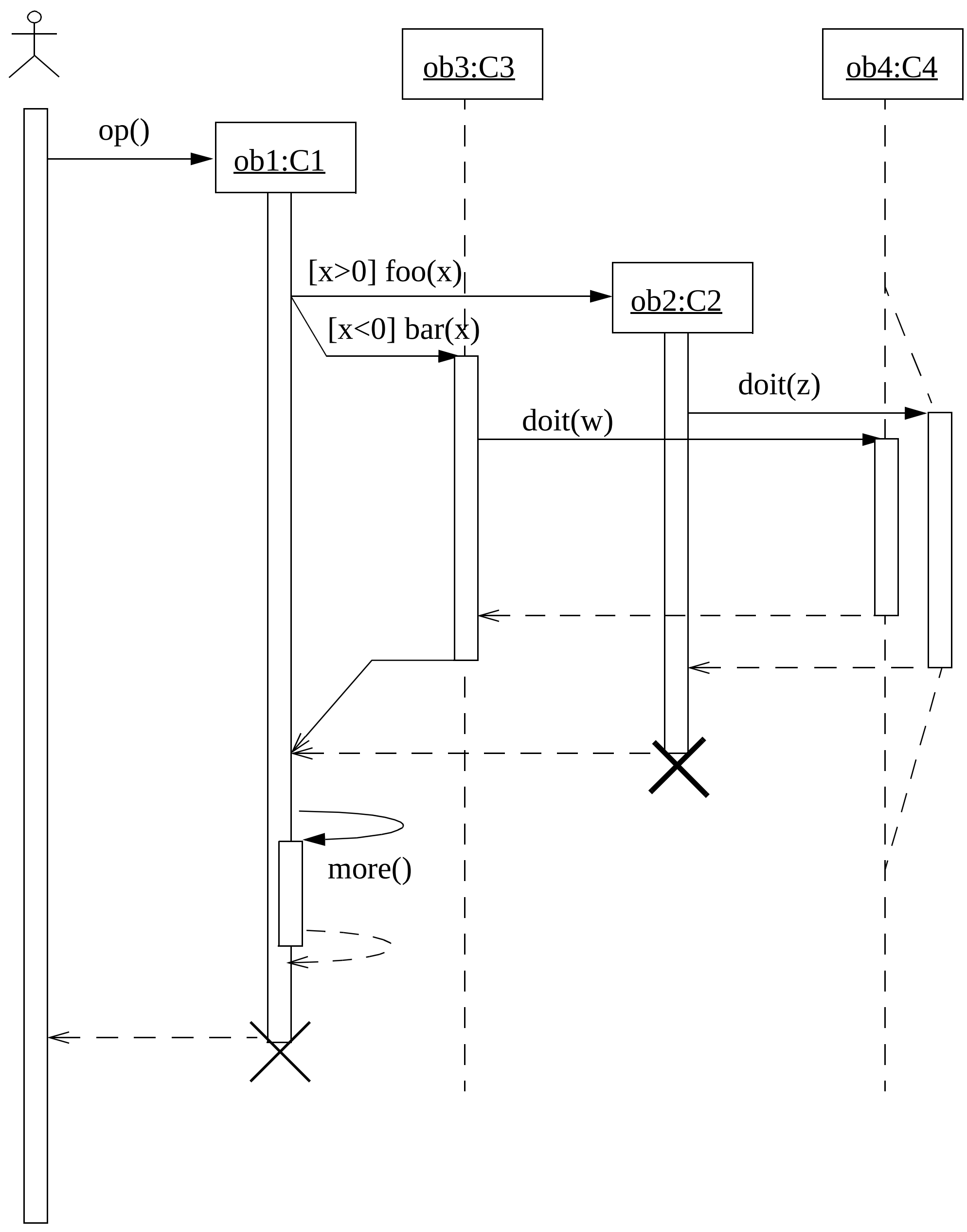


Figure 36. Sequence diagram with focus of control, conditional, recursion, creation, destruction



7.2.5 Mapping

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

A sequence diagram maps into an Interaction and an underlying Collaboration. Each object box with its lifeline maps into a ClassifierRole; the name field maps into the ClassifierRole name and the type field maps into the *type* association from the role to the Classifier with the given name. The associations among roles are not shown on the sequence diagram; they must be obtained in the model from a complementary collaboration diagram or other means. A message arrow maps into a Message between the ClassifierRoles corresponding to the two lifelines that the arrow connects; unless the correct AssociationRole can be determined from a complementary collaboration diagram or other means, the Message must be attached to a dummy AssociationRole implied between the two ClassifierRoles for lack of complete information. A timing label placed on the level of an arrow endpoint maps into the name of the corresponding Message. A constraint placed on the diagram maps into a Constraint on the entire Interaction.

An object symbol placed within the frame of the diagram maps into a CreateAction attached to the Message corresponding to the incoming arrow. If an object termination symbol ("X") is the target of an arrow, it maps into a DestroyAction attached to the Message corresponding to the arrow; otherwise it maps into a TerminateAction

On a diagram with concurrent objects, a *predecessor* association is established between Messages corresponding to successive arrows in the vertical sequence. In case of concurrent arrows, the mapping to a *predecessor* sequence may be ambiguous and may require additional information.

On a procedural sequence diagram (one with focus of control and calls) subsequent arrows on the same lifeline map into Messages obeying the *predecessor* association. An arrow to the head of a focus of control region establishes a nested activation; it maps into a Message (synchronous, activation) with associated CallAction (holding the arguments and referencing the target Operation) between the ClassifierRoles corresponding to the lifelines. All arrows departing the nested activation map into Messages with an *activation* Association to the Message corresponding to the arrow at the head of the activation. A return arrow departing the end of the activation maps into a Message (synchronous, reply) with an *activation* Association to the Message corresponding to the arrow at the head of the activation and a *predecessor* association to the previous message within the same activation. A return must be the final message within a predecessor chain; it is not the predecessor of any message. Any guard conditions or iteration conditions attached to a message arrow become *recurrence* values of the Message. The operation name is used to select the target Operation with the given name. The operation arguments become *argument* Expressions on the Action.

7.3 OBJECT LIFELINE

7.3.1 Semantics

A Role is a slot for an object within a collaboration that describes the type of object that may play the role and describes its relationships to other Roles. Within a sequence diagram the existence and duration of the object in a role is shown, but the relationships among the roles is not shown. There are ClassifierRoles and AssociationRoles.

7.3.2 Notation

An object role 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.

7.3.3 Example

See Figure 36.

7.3.4 Mapping

See Section 7.2.5.

7.4 ACTIVATION

7.4.1 Semantics

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

7.4.2 Notation

An activation is shown as a tall thin rectangle whose top is aligned with its initiation time and whose bottom is aligned with its completion time. The 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.)

7.4.3 Example

See Figure 36.

7.4.4 Mapping

See Section 7.2.5.

7.5 MESSAGE

7.5.1 Semantics

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

7.5.2 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.

7.5.3 Presentation options

Variation: Asynchronous. An asynchronous message is drawn with a half-arrowhead (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 dashed arrow.

Variation: In a procedural flow of control, the return arrow may be omitted (it is implicit at the end of an activation). It is assumed that every call has a paired return after any subordinate messages; the return value can be shown on the initial message line. 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.

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

7.5.4 Mapping

See Section 7.2.5.

7.6 TRANSITION TIMES

7.6.1 Semantics

A message may have a sending time and a receiving time. These are formal names that may be used within constraint expressions. The two may be the same (if the message is considered atomic) or different (if its delivery is nonatomic).

7.6.2 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.

7.6.3 Example

See Figure 35.

7.6.4 Mapping

See Section 7.2.5.

8. 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 object roles. 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.

8.1 COLLABORATION

8.1.1 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 for other purposes. Such a static construct is called a *collaboration.*

A collaboration is a set of participants and relationships that are meaningful for a given set of purposes. The identification of participants and their relationships does not have global meaning.

A collaboration may be attached to an operation or a use case to describe the context in which their behavior occurs. The actual behavior may be specified in interactions, such as sequence diagrams or collaboration diagrams. A collaboration may also be attached to a class to define the class's static structure.

A parameterized collaboration represents a design construct that can be used repeatedly in different designs. The participants in the collaboration, including the classes and relationships, 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 can capture the structure of a *design pattern* (note that a design pattern involves more than structural aspects). Whereas most collaborations can be anonymous because they are attached to a named entity, patterns are free standing design constructs that must have names.

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

8.1.2 Notation

The description of behavior 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 structure of objects playing roles in a behavior and their relationships is called a *collaboration.* A collaboration shows the context in which interaction occurs. The dynamic behavior of the message sequences exchanged among objects to accomplish a specific purpose is called an *interaction.* A collaboration is shown by a collaboration diagram without messages. By adding messages, an interaction is shown. Different sets of messages may be applied to the same collaboration to yield different interactions.

8.2 COLLABORATION DIAGRAM

8.2.1 Semantics

A collaboration diagram represents a Collaboration, which is a set of objects related in a particular context, and an Interaction, which is a set of messages exchanged among the objects within a collaboration to effect a desired operation or result.

8.2.2 Notation

A collaboration diagram is a graph of references to objects and links with message flows attached to its links. The diagram shows the objects relevant to the performance of an operation, including objects indirectly affected or accessed during the operation. The collaboration used to describe 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}. These changes in life state are derivable from the detailed messages sent among the objects; the are provided as notational conveniences.

The diagram also shows the links among the objects, including transient links representing procedure arguments, local variables, and *self* links. Because collaboration diagrams are often used to help design procedures, they typically show navigability using arrowheads on links. (An arrowhead on a line between object boxes indicates a link with one-way navigability. An arrow next to a line indicates a message flowing in the given direction over the link. Obviously a message arrow cannot flow backwards over a one-way link.)

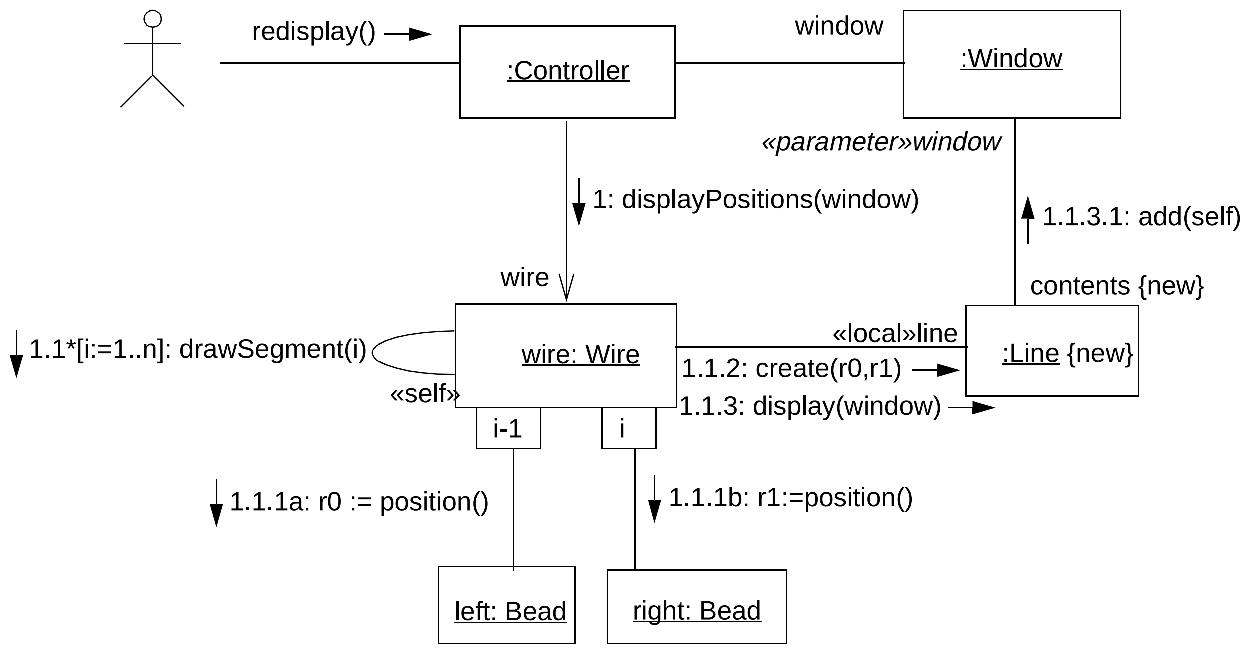
Individual attribute values are usually not shown explicitly. If messages must be sent to attribute values, the attributes should be modeled using associations instead.

The internal messages that implement a method are numbered 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).

A collaboration diagram without messages shows the *context* in which interactions can occur, without showing any specific interactions. It might be used to show the context for a single operation or even for all of the operations of a class or group of classes.

8.2.3 Example

Figure 37. Collaboration diagram



8.2.4 Mapping

A collaboration diagram maps to a Collaboration with a superimposed Interaction.

8.3 PATTERN STRUCTURE

8.3.1 Semantics

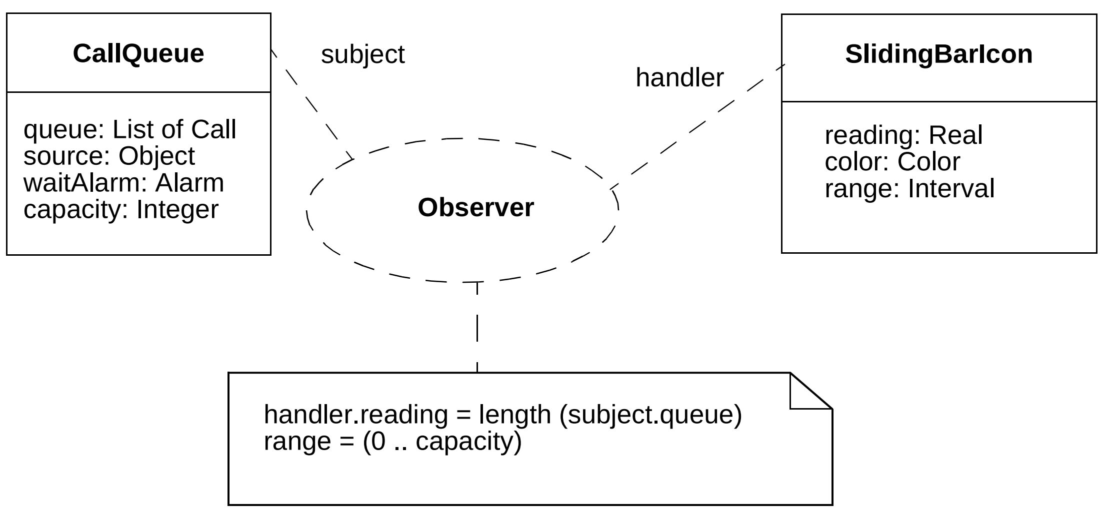
A collaboration can be used to specify the implementation of design constructs. For this purpose it is necessary to specify its context and interactions. It is also possible to view a collaboration as a single entity from the "outside." For example, this could be used to identify the presence of design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

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

8.3.2 Notation

A use of a collaboration is shown as a dashed ellipse containing the name of the collaboration. A dashed line is drawn from the collaboration symbol to each of the objects or classes (depending on whether it appears within an object diagram or a class diagram) that participate in the collaboration. Each line is labeled by the *role* of the participant. The roles correspond to the names of elements within the context for the collaboration; such names in the collaboration are treated as parameters that are bound to specify elements on each occurrence of the pattern within a model. Therefore a collaboration symbol can shown the use of a design pattern together with the actual classes that occur in that particular use of the pattern.

Figure 38. Use of a collaboration



8.3.3 Mapping

A collaboration usage symbol maps into a Collaboration. For each class symbol attached by an arrow to the pattern occurrence symbol, the corresponding Class is bound to the template parameter that is the *type* association target of the ClassifierRole in the Pattern with the name equal to the name on the arrow.

8.4 COLLABORATION CONTENTS

The contents of a collaboration are modeling elements that interact within a given context for a particular purpose, such as performing an operation or a use case; it is a “society of objects”. A collaboration is a fragment of a larger complete model that is intended for a particular purpose.

8.4.1 Semantics

A *collaboration* shows one or more roles together with their contents, associations, and neighbor roles, plus additional relationships and classes as needed. To use a collaboration, each role must be bound to an actual class that can support the operations required of the role.

8.4.2 Notation

A collaboration is shown as a graph of class references and association references. Each reference is a *role* of the collaboration; that is, each entity is playing a role within the context of the collaboration, a role that is only part of its full description. The names of the objects represent their roles within the collaboration. A collaboration is a prototype; in each use of the collaboration the roles are bound to actual objects. There are several ways to show the diagram:

Methods. If the collaboration shows the implementation of an operation (a method), then it is usually drawn as a separate collaboration diagram including context to which message flow is added to obtain an interaction. The collaboration 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 collaboration 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 implementation of an operation, message flows are superimposed on the links between objects in the collaboration; each flow shows a step within the method for the operation (see Section 8.9).

Classes. A collaboration is normally defined for a single operation. By taking the union of all of the collaborations for all of the operations of a class, an overall collaboration for the entire class can be shown. This collaboration shows all of the context for the implementation of the class.

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

Different collaborations may be devised for the same class for different purposes. Each collaboration may show a somewhat different subset of attributes, operators, and related objects that are relevant to each purpose. Inasmuch as actual operations often fall into related groups, each collaboration 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 collaborations, each from the point of view of a particular stakeholder.

8.5 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 exchanges to accomplish a specific purpose is called an *interaction.*

8.5.1 Semantics

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

8.5.2 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 7. 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.

8.5.3 Example

See Collaboration Diagram section for a collaboration underlying an interaction.

8.6 COLLABORATION ROLES

8.6.1 Semantics

A Role is a slot for an object within a collaboration that describes the type of object that may play the role and describes its relationships to other Roles. There are ClassifierRoles and AssociationRoles.

8.6.2 Notation

A collaboration role is shown using the notation for an object or a link. Keep in mind, however, that in the context of a collaboration these represent roles that *bind* to actual objects or links when the collaboration is used, not actual objects and links.

A class role is shown as a class rectangle symbol. Normally only the name compartment is shown. The name compartment contains the string:

classRoleName : Classifiername

The classname can include a full pathname of enclosing packages, if necessary (a tool will normally permit shortened pathnames to be used when they are unambiguous). The package names precede the 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.

A class role representing a set of objects includes a multiplicity indicator (such as “*”) in the upper right corner of the class box.

An association role is shown as a path between two class role symbols. If the name of the corresponding association is included it is underlined. Rolenames are not underlined. Even in absence of underlining a line connecting class roles is an association role.

If one end of the association role path is connected to a multiple class role, then a multiplicity indicator may be placed on that end to emphasize the multiplicity.

8.6.3 Presentation options

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

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

8.6.4 Example

See Figure 37.

8.6.5 Mapping

The object symbol in a collaboration diagram maps to a ClassifierRole whose name matches the *object* part of the name string; the role has a *type* Association to a Classifier whose name matches the *type* part of the name string.

8.7 MULTIOBJECT

8.7.1 Semantics

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

8.7.2 Notation

A multiobject is shown as two rectangles in which the top rectangle is shifted slightly vertically and horizontally to suggest a stack of rectangles. A message arrow to the multiobject symbol indicates a message to the set of objects, for example, a selection operation to find an individual object.

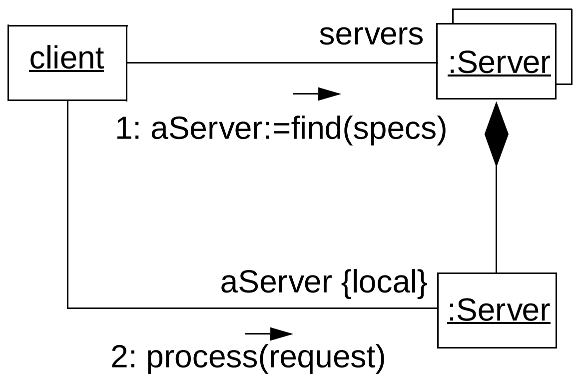
To perform an operation on each object in a set of associated objects requires two messages: an iteration to the multiobject to extract links to the individual objects, then a message sent to each individual object using the (temporary) link. This may be elided on a diagram by combining the messages into a single message that includes an iteration and an application to each individual object; the target rolename takes a “many” indicator (*) to show that many individual links are implied. Although this may be written as a single message, in the underlying model (and in any actual code) it requires the two layers of structure (iteration to find links, message using each link) mentioned previously.

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

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

8.7.3 Example

Figure 39. Multiobject



8.7.4 Mapping

A multiobject symbol maps to a ClassifierRole with multiplicity “many” (or whatever is explicitly specified). In other respects it maps the same as an object symbol.

8.8 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. In a collaboration diagram, a ClassifierRole that is an active class represents the active objects that occur during execution.

8.8.1 Semantics

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

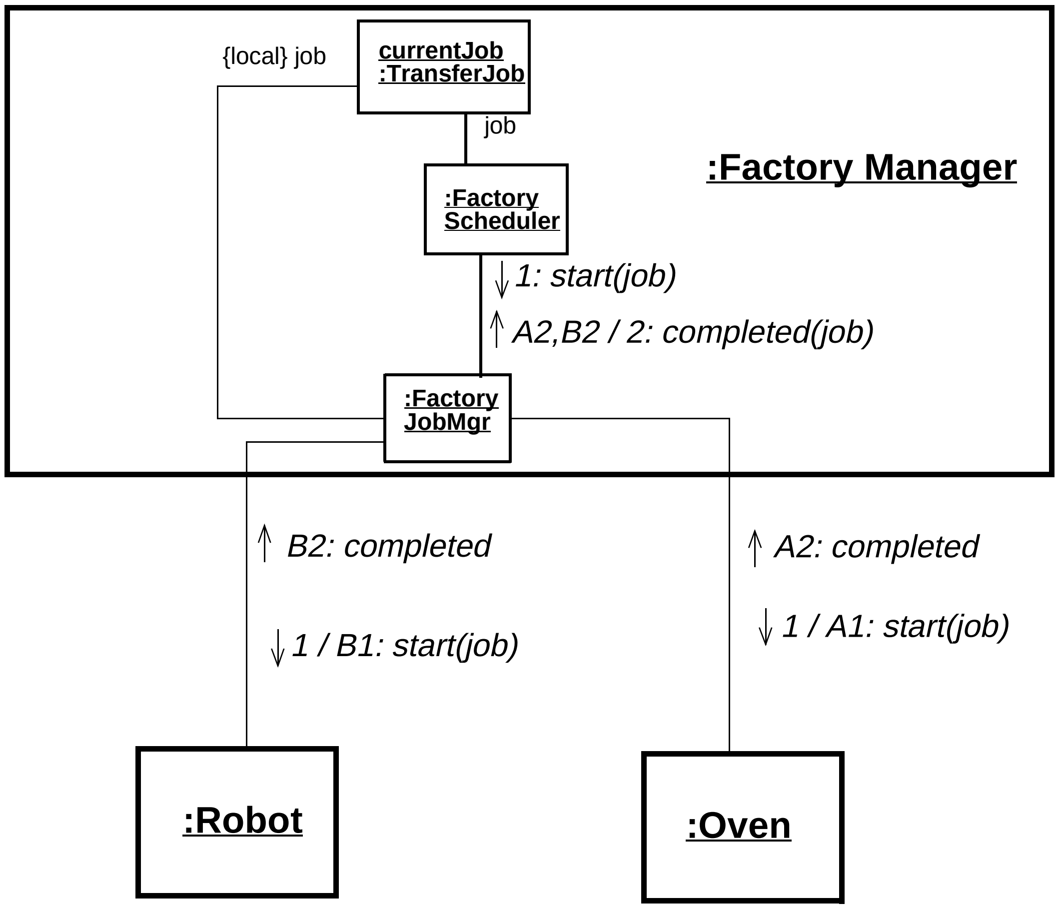
8.8.2 Notation

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

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

8.8.3 Example

Figure 40. Composite active object



8.8.4 Mapping

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

A nested object symbol (active or not) maps into a Classifierrole that has a composition association to the roles corresponding to its contents, as described under Composition.

8.9 MESSAGE FLOWS

8.9.1 Semantics

A *message flow* is 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.

8.9.2 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. Normally all of the messages are asynchronous.

half stick arrowhead

asynchronous flow of control. Used instead of the stick arrowhead to explicitly show an asynchronous message between two objects in a procedural sequence.

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.

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

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 notation for this is to follow the star by a double vertical line (for parallelism): *||.

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

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).

8.9.3 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.

8.9.4 Example

See Figure 37 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

8.9.5 Mapping

A message flow symbol maps into a Message between the ClassifierRoles corresponding to the boxes connected by the association path bearing the message flow symbol. The control flow type sets the corresponding Message properties.

The predecessor expression together with the sequence expression determine the *predecessor* and *activation* (caller) associations between the Message and other messages. The predecessors of the Message are the messages corresponding to the sequence numbers in the predecessor list as well as the message corresponding to the immediate preceding sequence number as the Message (i.e., 1.2.2 is the one preceding 1.2.3). The caller of the Message is the Message whose sequence number is truncated by one position (i.e., 1.2 is the caller of 1.2.3).

The return value maps into a message from the called object to the caller with direction *return*. Its *predecessor* is the final message within the procedure. Its *activation* is the message that called the procedure.

The recurrence expression, the iteration clause, and the condition clause determine the recurrence value in the Message.

The operation name and the form of the signature determine the Operation attached to the CallAction associated with the Message.

The arguments of the signature determine the arguments associated with the CallAction.

In a procedural interaction, each message flow symbol also maps into a second Message with the properties (synchronous, reply) representing the return flow. This Message has an *activation* Association to the original call Message. Its associated Action is a ReturnAction bearing the return values as arguments (if any).

8.10 CREATION/DESTRUCTION MARKERS

8.10.1 Semantics

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.

8.10.2 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.*

8.10.3 Presentation options

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

8.10.4 Example

See Figure 37.

8.10.5 Mapping

Creation or destruction indicators map into CreateActions or DestroyActions actions on the target ClassifierRoles or into TerminateActions. The actions correspond to messages that cause the changes. These status indicators are merely summaries of the total actions.

9. STATECHART DIAGRAMS

A statechart 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 modifications to make them object-oriented. His work was a major advance on the traditional flat state machines. Statechart notation also implements aspects of both Moore machines and Mealy machines, traditional state machine models.

9.1 STATECHART DIAGRAM

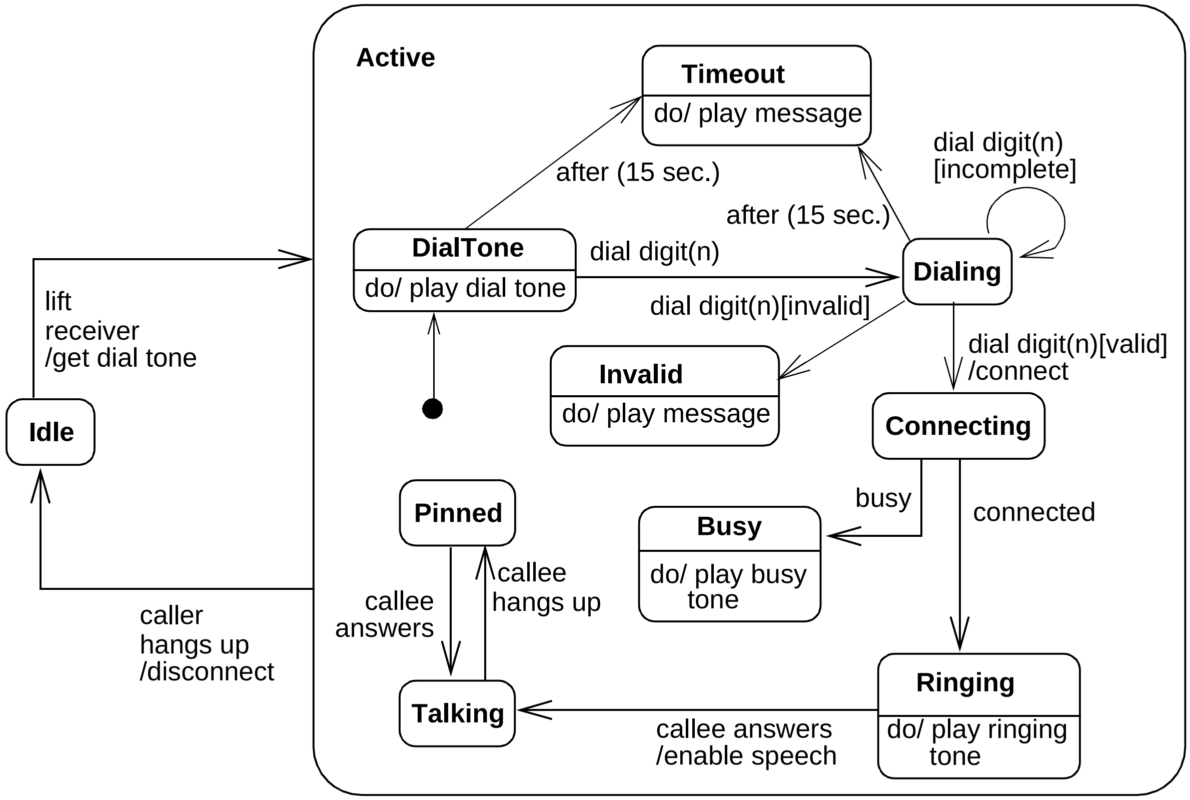
9.1.1 Semantics

A state machine is a graph of states and transitions that describes the response of an object of a given class to the receipt of outside stimuli. A state machine is attached to a class or a method

9.1.2 Notation

A statechart diagram represents a state machine. The states are represented by state symbols and the transitions are represented by arrows connecting the state symbols. States may also contain subdiagrams by physical containment and tiling. .

Figure 41. State diagram



9.1.3 Mapping

A statechart diagram maps into a StateMachine. That StateMachine may be attached to a Class or a Method but there is no explicit notation for this.

9.2 STATES

9.2.1 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.

Actions are atomic and non-interruptible. A state may correspond to ongoing activity. Such activity is expressed as a nested state machine. Alternately, ongoing activity may be represented by a pair of actions, one that starts the activity on entry to the state and one that terminates the activity on exit from the state.

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. Completion of the outermost state of an object corresponds to its death.

9.2.2 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 It is undesirable to show the same named state twice in the same diagram, however, as confusion may ensue.

Internal transition compartment. Holds a list of internal actions or activities performed in response to events received while the object is in the state, without changing state. These have the format:

> *event-name argument-list* ‘[’ guard-condition ‘]’‘/’ *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:

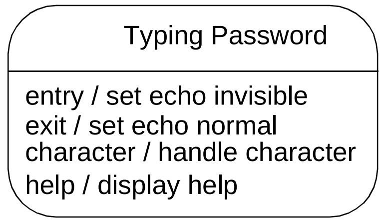
> ‘entry’ ‘/’ *action-expression*An atomic action performed on entry to the state
>
> ‘exit’ ‘/’ *action-expression*An atomic action performed on exit from the state
>
> Entry and exit actions may not have arguments or guard conditions (because they are invoked implicitly, not explicitly). However, the entry action at the top level of the state machine for a class may have parameters that represent the arguments that it receives when it is created.
>
> Action expressions may use attributes and links of the owning object and parameters of incoming transitions (if they appear on all incoming transitions).

The following keyword represents the invocation of a nested state machine:

> ‘do’ ‘/’ *machine-name* (*argument-list*)
>
> The *machine-name* must be the name of a state machine that has an initial and final state. If the nested machine has parameters, then the argument list must match correctly. When this state is entered, after any entry action then execution of the nested state machine begins with its initial state. When the nested state machine reaches its final state, then any exit action in the current state is performed and the current state is considered completed and may take a transition based on implicit completion of activity.

9.2.3 Example

Figure 42. State



9.2.4 Mapping

A state symbol maps into a State. See the next section for further details on which kind of state.

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

An internal action string with the name “entry” or “exit” maps into an association: the source is the State corresponding to the enclosing state symbol; the target is an ActionSequence that maps the action expression; the association is the Entry action or the Exit action association.

An internal action string with the name “do” maps into the invocation of a nested state machine.

Any other internal action maps into an *internal* Association from the corresponding State to a Transition. The action expression maps into the ActionSequence and Guard for the Transition. The event name and arguments map into an Event corresponding to the event name and arguments; the Transition has a *trigger* Association to the Event

9.3 Composite States

9.3.1 Semantics

A state can be decomposed 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.

9.3.2 Notation

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

An expansion of a state into concurrent substates is shown by tiling the graphic region of the state using dashed lines to divide it into 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 initial (pseudo)state is shown as a small solid filled circle. In a top-level state machine, the transition from an initial state may be labeled with the event that creates the object; otherwise it must be unlabeled. If it is unlabeled, it represents any transition to the enclosing state. The initial transition may have an action. 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 represents the completion of activity in the enclosing state and it triggers a transition on the enclosing state labeled by the implicit activity completion event (usually displayed as an unlabeled transition).

9.3.3 Example

Figure 43. Sequential substates

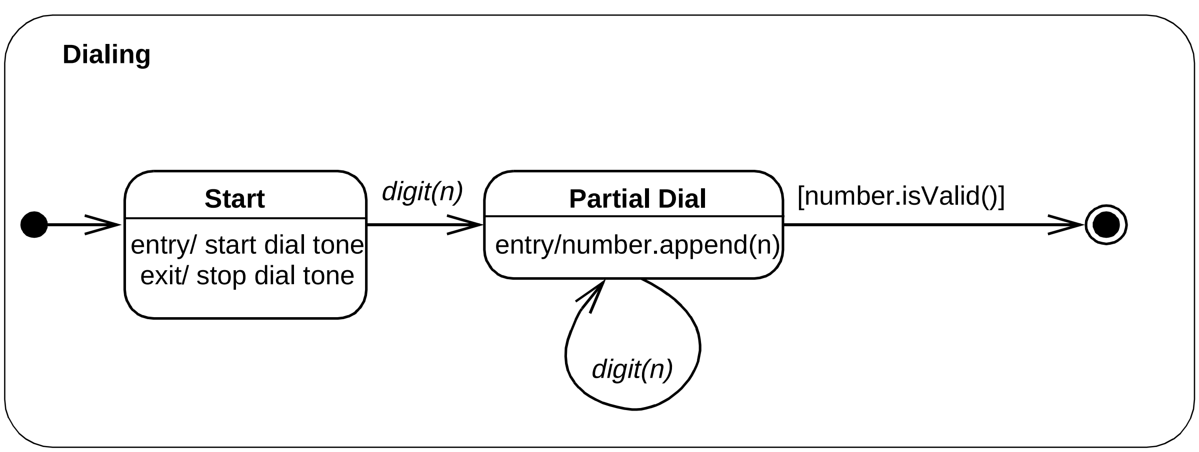
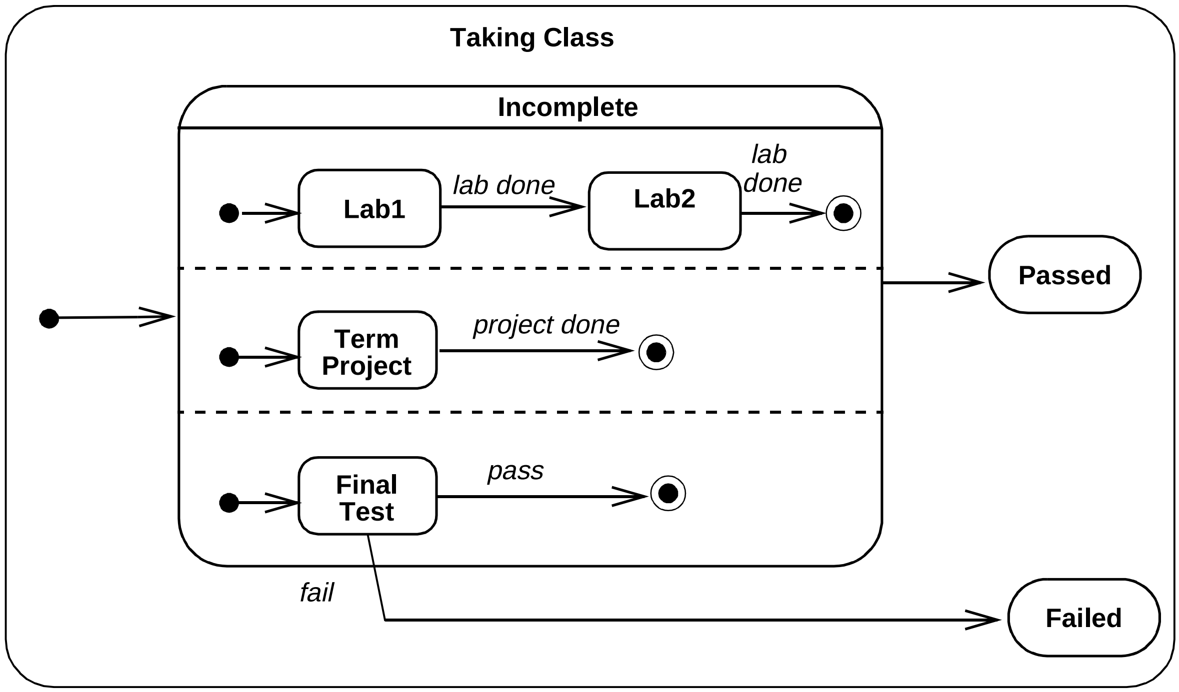


Figure 44. Concurrent substates



9.3.4 Mapping

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

An initial state symbol or a final state symbol map into a Pseudostate of kind *initial* or *final.*

9.4 EVENTS

9.4.1 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). This is a ChangeEvent. These are notated with the keyword **when** followed by a boolean expression in parentheses. The event occurs whenever the value of the expression changes from false to true. Note that this is different from a guard condition: A guard condition is evalu-

ated *once* whenver its event fires; if it is false then the transition does not occur and the event is lost. Example: **when** (balance < 0).

receipt of an explicit signal from one object to another. This is a SignalEvent. One of these is notated by the signature of the event as a trigger on a transition.

receipt of a call for an operation by an object. This is a CallEvent. These are notated by the signature of the operation as a trigger on a transitions. There is no visual difference from a signal event; it is assumed that the names with distinguish them.

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. This is a TimeEvent. These are notated as time expressions as triggers on transitions. One common time expression is the passage of time since the entry to the current state; this is notated with the keyword **after** followed by an amount of time in parentheses. Example: **after** (10 seconds).

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.

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

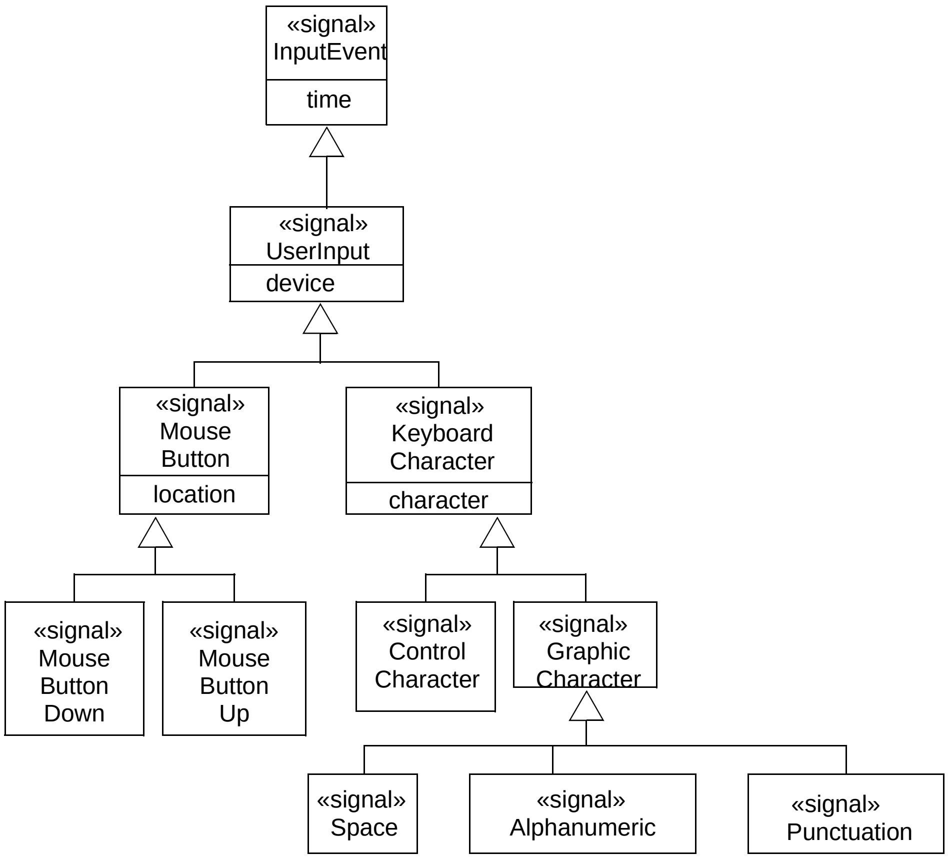
An elapsed-time event can be specified with the keyword **after** followed by an expression that evaluates (at modeling time) to an amount of time, such as “**after** (5 seconds)” or **after** (10 seconds since exit from state A)”. If no starting point is indicated, then it is the time since the entry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

A condition becoming true is shown with the keyword **when** followed by a boolean expression. This may be regarded as a continuous test for the condition until it is true, although in practice it would only be checked on a change of values (and there are ways to determine when it must be check). This is mapped into a ChangeEvent in the model.

Signals can be declared on a class diagram with the keyword «signal» on a rectangle symbol. These define signal names that may be used to trigger transitions. Their parameters are shown in the attribute compartment. They have no operations. They may appear in a generalization hierarchy. Note that they are *not* real classes and may not appear in relationships to real classes.

9.4.3 Example

Figure 45. Signal declaration



9.4.4 Mapping

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

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

9.5 SIMPLE TRANSITIONS

9.5.1 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 within the same sequential region (i.e., not in different concurrent regions), only one will fire; the choice may be nondeterministic if a firing priority is not specified.

9.5.2 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 guard condition may also involve tests of concurrent states of the current machine (or explicitly designated states of some reachable object); for example, “**in** State1” or “**not in** State2”. State names may be fully qualified by the nested states that contain them, yielding path names of the form “State1::State2::State3”; this may be used in case same state name occurs in different composite state regions of the overall machine.

The *action-expression* is 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-message-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-message-name* is the name of a message (operation or signal) 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.

Branches. A simple transition may be extended to include a tree of decision symbols (see Section 10.3). This is equivalent to a set of individual transitions, one for each path through the tree, whose guard condition is the “and” of all of the conditions along the path.

Transition times. Names may be placed on transitions to designate the times at which they fire. See the section on transition times within Section 7.6.

9.5.3 Example

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

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

9.5.4 Mapping

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

The event name and parameters map into an Event element, which may be a SignalEvent, a CallEvent, or a TimeExpression (if it has the proper syntax). The event is attached as a *trigger* Association to the Transition.

The guard condition maps into a Guard element attached to the Transition.

An action expression maps into an ActionSequence attached as an *effect* Association to the Transition; the target object expression (if any) in the expression maps into a *target* ObjectSetExpression. Each term in the action expression maps into an Action that is a part of the ActionSequence. A send clause maps into a RaiseAction with an ObjectSetExpression for the destination.

A transition time label on a transition maps into a TimingMark attached to the Transition.

9.6 COMPLEX TRANSITIONS

A complex 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.

9.6.1 Semantics

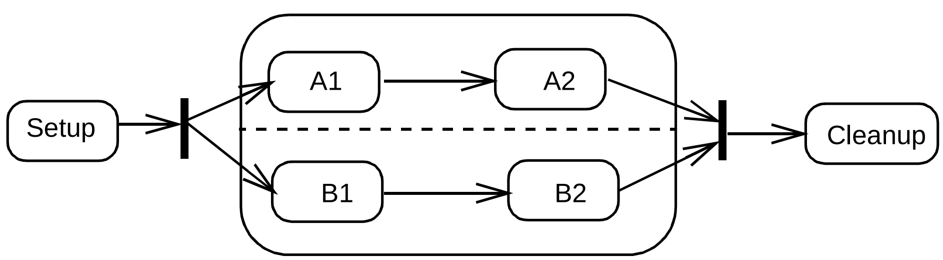
A complex transition is enabled when all of the source states are occupied. After a complex transition fires all of its destination states are occupied.

9.6.2 Notation

A complex transition is shown as a short heavy bar (a *synchronization* bar, which can represent synchronization, forking, or both). 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.

9.6.3 Example

Figure 46. Complex transition



9.6.4 Mapping

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

9.7 TRANSITIONS TO NESTED STATES

9.7.1 Semantics

A transition drawn to the boundary of a complex state is equivalent to a transition to its initial state (or to a complex transition to the initial states 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.

9.7.2 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 are 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 have at most one outgoing unlabeled transition; this identifies the default “previous state” if the region has never been entered. If a transition to the history indicator fires it indicates that the object resumes the state it last had within the complex region; any necessary entry actions are performed. The history indicator may also be ‘H*’ for *deep history.* This indicates that the object resumes the state it last had at any depth within the complex region, rather than being restricted to the state at the same level as the history indicator. A region may have both shallow and deep history indicators.

9.7.3 Presentation options

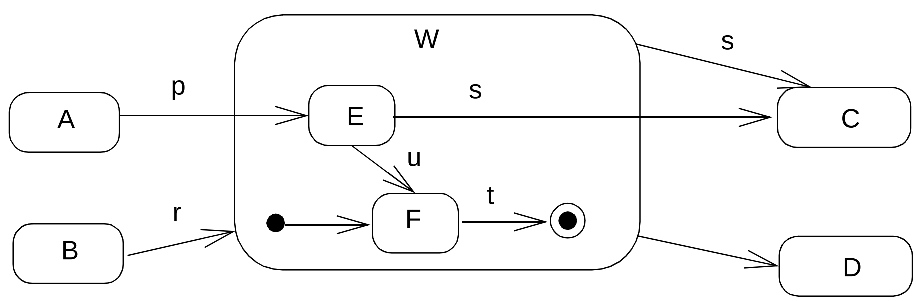
Stubbed transitions. Nested states may be suppressed. Transitions to nested states are subsumed to the most specific visible enclosing state of the suppressed state. Subsumed transitions that do not come from an unlabeled final state or go to an unlabeled initial state may (but need not) be shown as 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).

9.7.4 Example

See Figure 44 and Figure 46 for examples of complex transitions. Following are examples of stubbed transitions and the history indicator.

Figure 47. Stubbed transitions



may be abstracted as

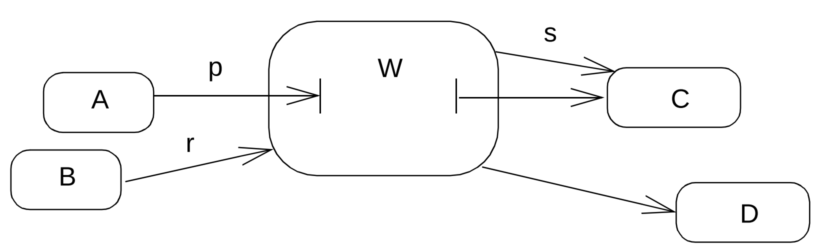
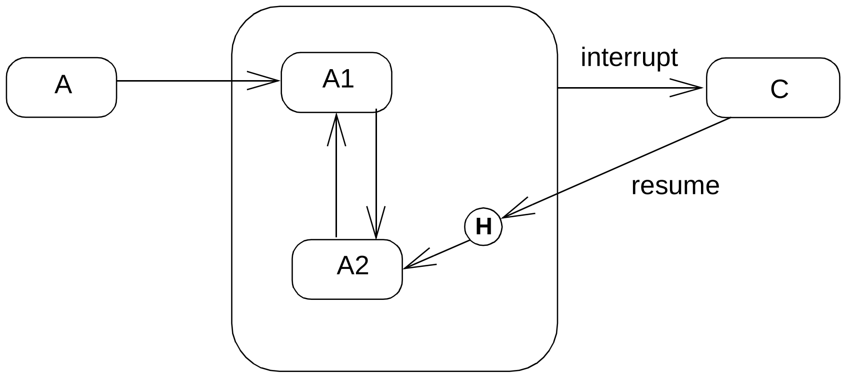


Figure 48. History indicator



9.7.5 Mapping

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

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

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

9.8 SENDING MESSAGES

9.8.1 Semantics

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

9.8.2 Notation

See Section 9.5 for the text syntax of sending messages that cause events for other objects.

Sending such a message can also be shown visually. See Section 7.5 and Section 8.9 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.

Note that this notation may also be used on other kinds of diagrams to show sending of events between classes or 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.

9.8.3 Example

Figure 49. Sending messages

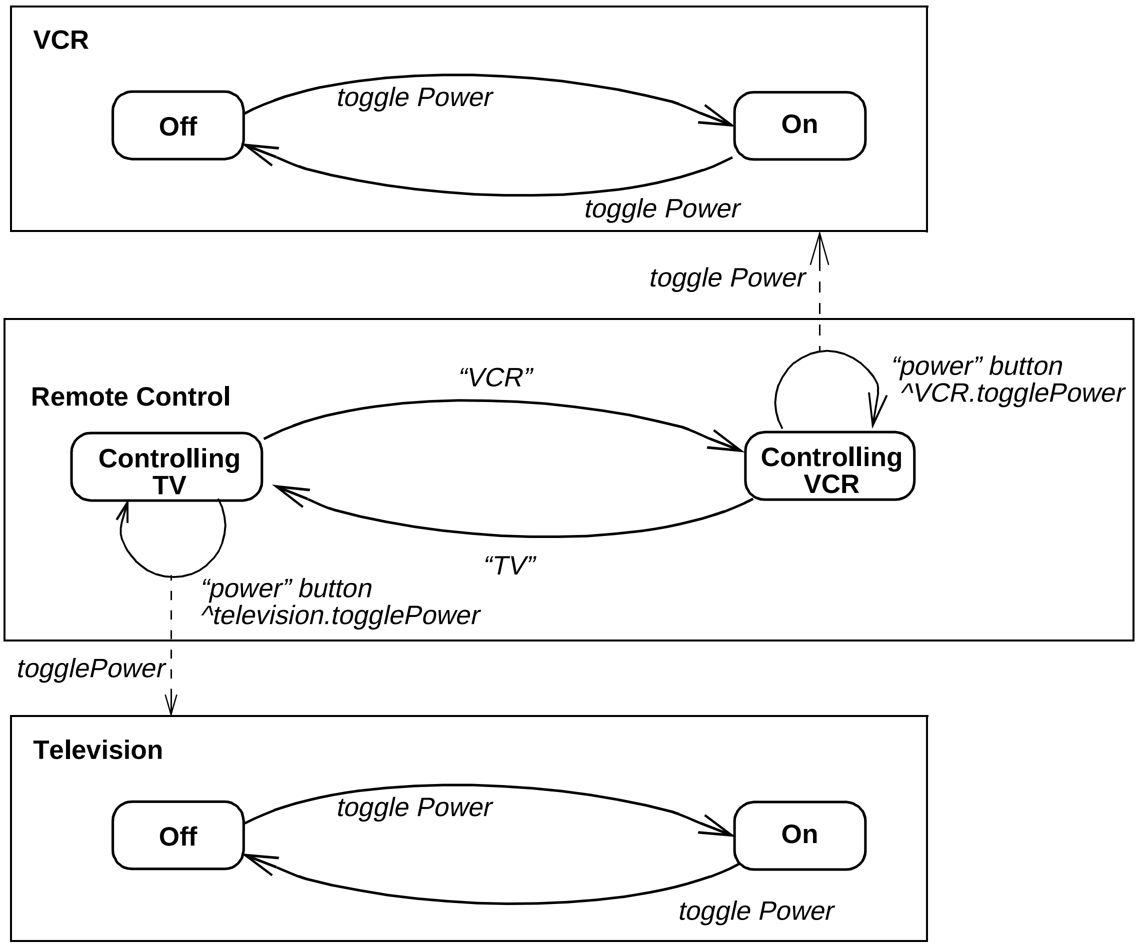
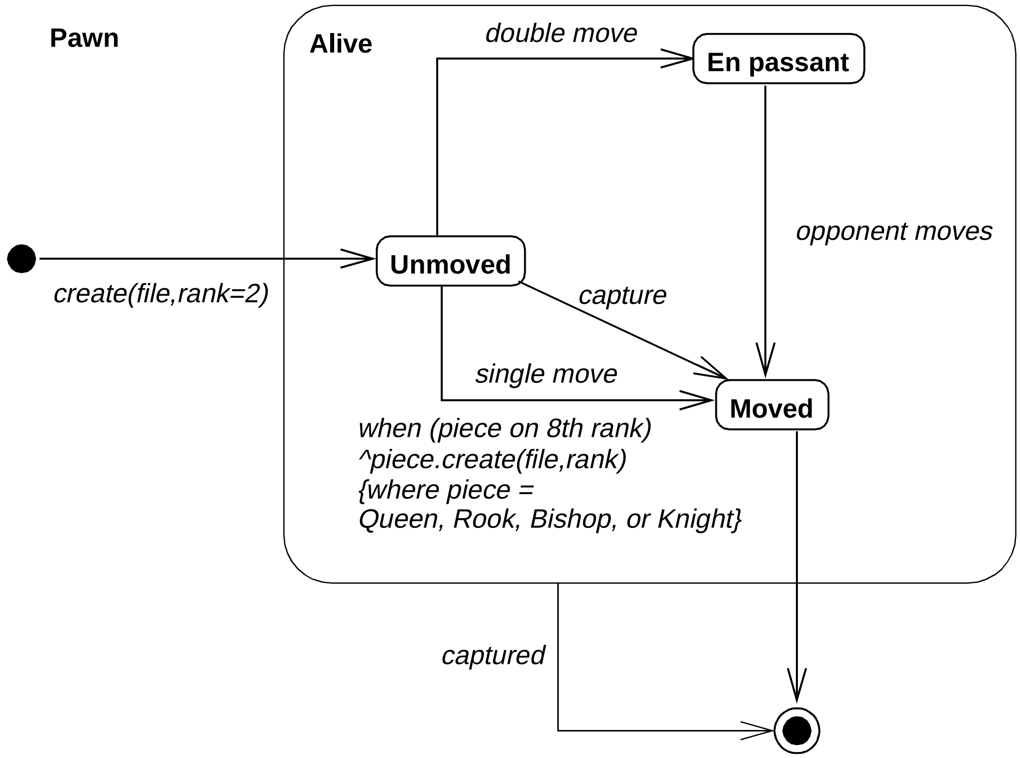


Figure 50. Creating and destroying objects



9.8.4 Mapping

A send arrow to an object maps into a SendAction whose *message* is a Signal that corresponds to the name on the arrow and whose *target* ObjectSetExpression corresponds to the target object.

If the arrow goes directly to a transition in the target object statechart, then the *target* ObjectSetExression corresponds to the object owning the statechart containing the transition. In addition, the transition in the target statechart implicitly triggers on the event being sent (i.e., the name of the sent event is effectively written on the target transition).

If the sender symbol is an object, then the diagram is suggestive of the sender but has no actual semantic mapping.

9.9 INTERNAL TRANSITIONS

9.9.1 Semantics

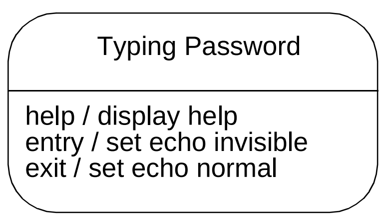
An internal transition is a transition that remains within a single state rather than a transition that involves two states. It represents the occurrence of an event that does not cause a change of state. Entering the state (from any other state not nested in the particular state) and exiting the state (to any other state not nested in the particular state) are treated notationally as internal transitions with the reserved words “entry” and “exit”, but they are not really internal transitions in the internal model.

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).

9.9.2 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 9.5 for details.

Figure 51. State with internal transitions



9.9.3 Mapping

The mapping for internal transitions has been given in Section 9.2.4.

10. ACTIVITY DIAGRAM

10.1 ACTIVITY DIAGRAM

10.1.1 Semantics

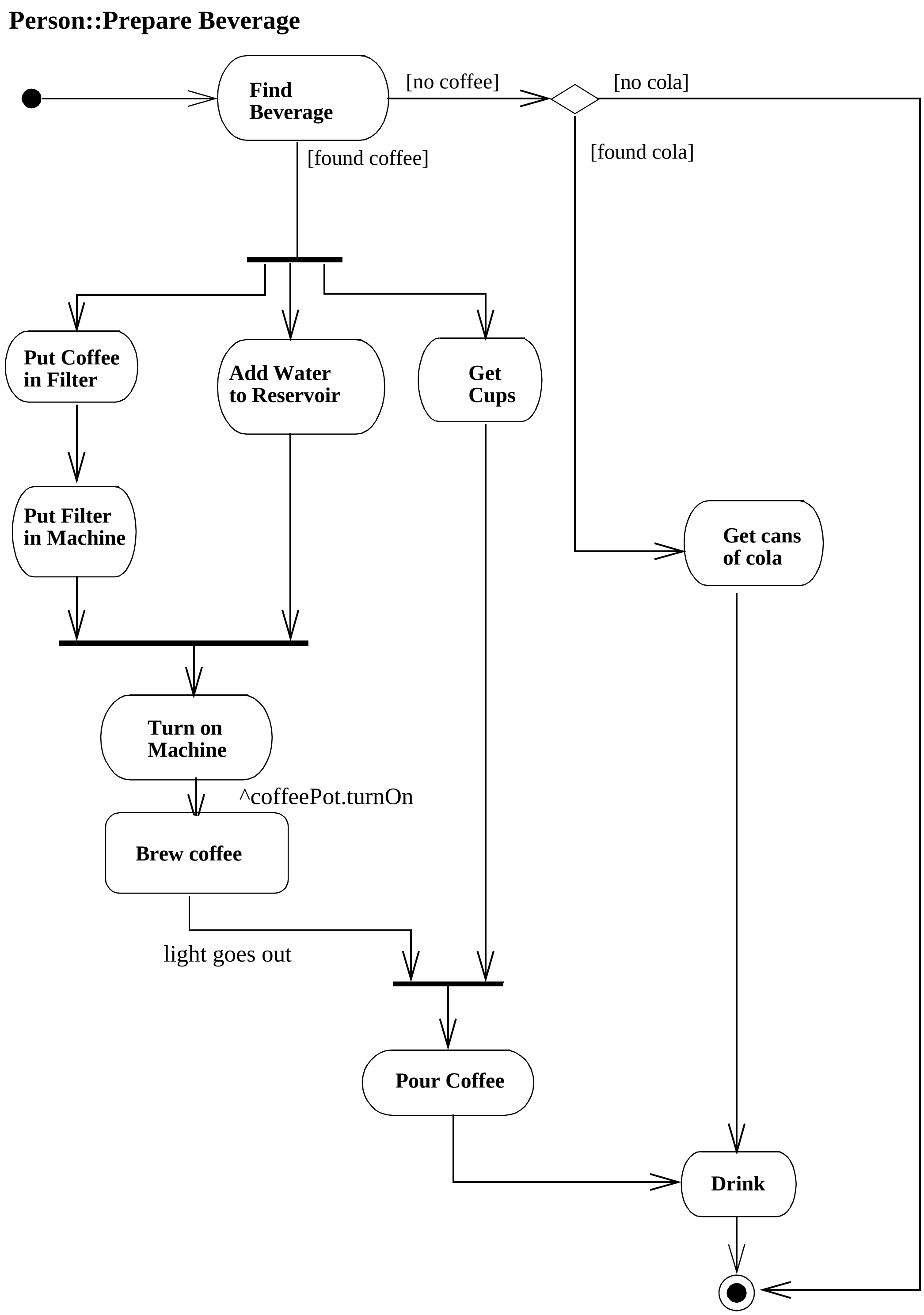
An activity model is a variation of a state machine in which the states are Activities representing the performance of operations and the transitions are triggered by the completion of the operations. It represents a state machine of a procedure itself; the procedure is the implementation of an operation on the owning class.

10.1.2 Notation

An activity diagram is a special case of a state diagram in which all (or at least most) of the states are action 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.

10.1.3 Example

Figure 52. Activity diagram



10.1.4 Mapping

An activity diagram maps into an ActivityModel.

10.2 ACTION STATE

10.2.1 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).

10.2.2 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.

10.2.3 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.

10.2.4 Example

Figure 53. Activities

10.2.5 Mapping

An action state symbol maps into an ActionState invoking a CallAction. This is equivalent to an *entry* action on a regular state. There is no *exit* nor any internal transitions. The State is normally anonymous.

10.3 DECISIONS

10.3.1 Semantics

A state diagram (and by derivation an activity diagram) expresses a decision when guard conditions are used to indicate different possible transitions that depend on Boolean conditions of the owning object. UML provides shorthand for showing decisions.

10.3.2 Notation

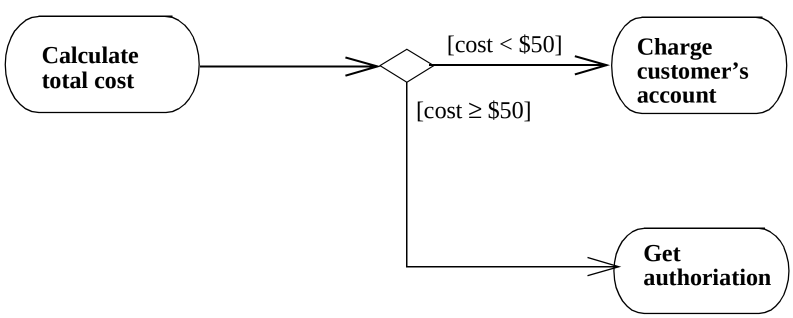
A decision may be shown by labeling multiple output transitions of an action with different guard conditions.

The icon provided for a decision is the traditional diamond shape, with one 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.

Note that a chain of decisions may be part of a complex transition, but only the first segment in such a chain may contain an event trigger label. All segments may have guard expressions.

10.3.3 Example

Figure 54. Decision



10.3.4 Mapping

A decision symbol maps into a Pseudostate of kind *branch*. Each label on an outgoing arrow maps into a Guard on the corresponding Transition leaving the Pseudostate.

10.4 SWIMLANES

10.4.1 Semantics

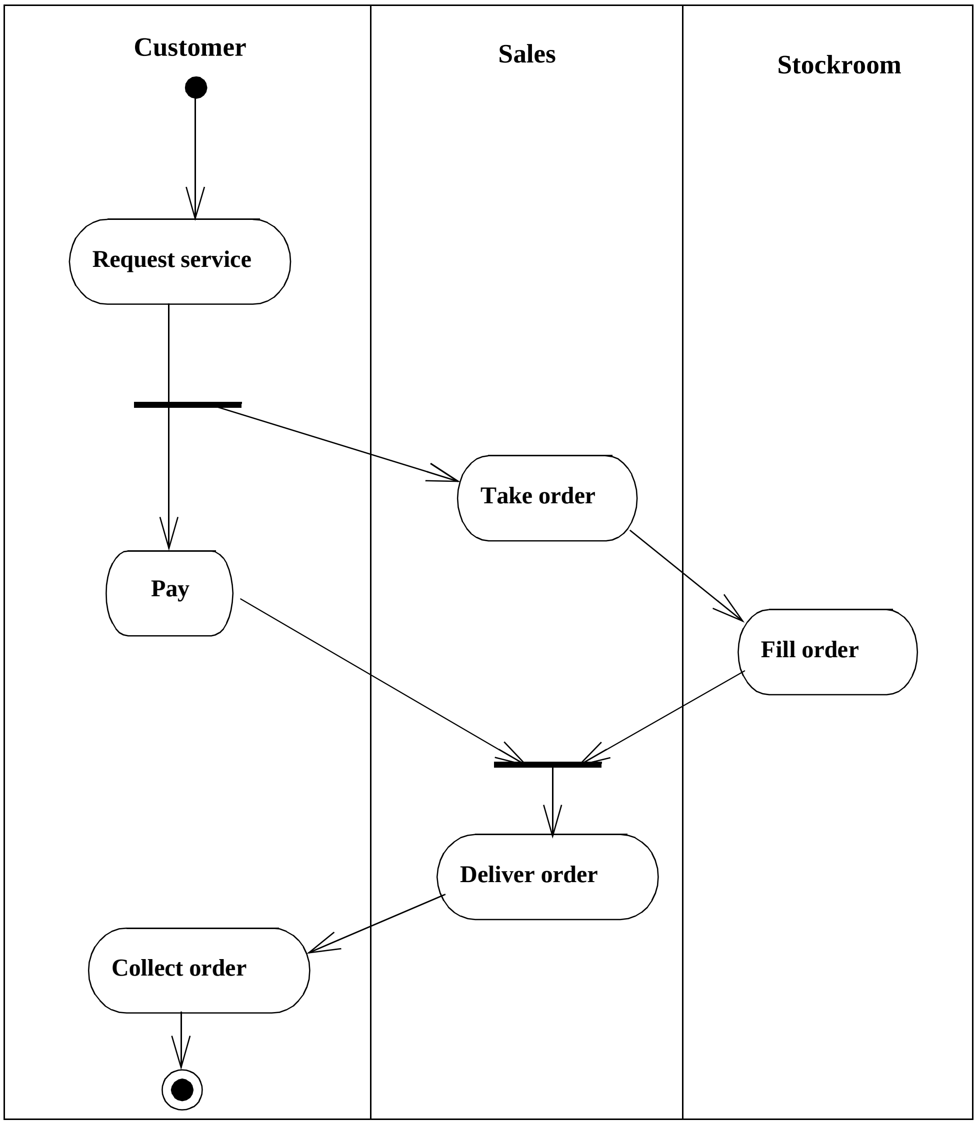
Actions may be organized into *swimlanes.* Swimlanes are a kind of package for organizing responsibility for activities within a class. They often correspond to organizational units in a business model.

10.4.2 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.

10.4.3 Example

Figure 55. Swimlanes in activity diagram



10.4.4 Mapping

A swimlane maps into a Partition of the States in the ActivityModel. A state symbol in a swimlane causes the corresponding State to belong to the corresponding Partition.

10.5 ACTION-OBJECT FLOW RELATIONSHIPS

10.5.1 Semantics

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. These are modeled as messages sent between the object owning the activity model and the objects that are input or output by the actions in the model.

10.5.2 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 7.2. 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.

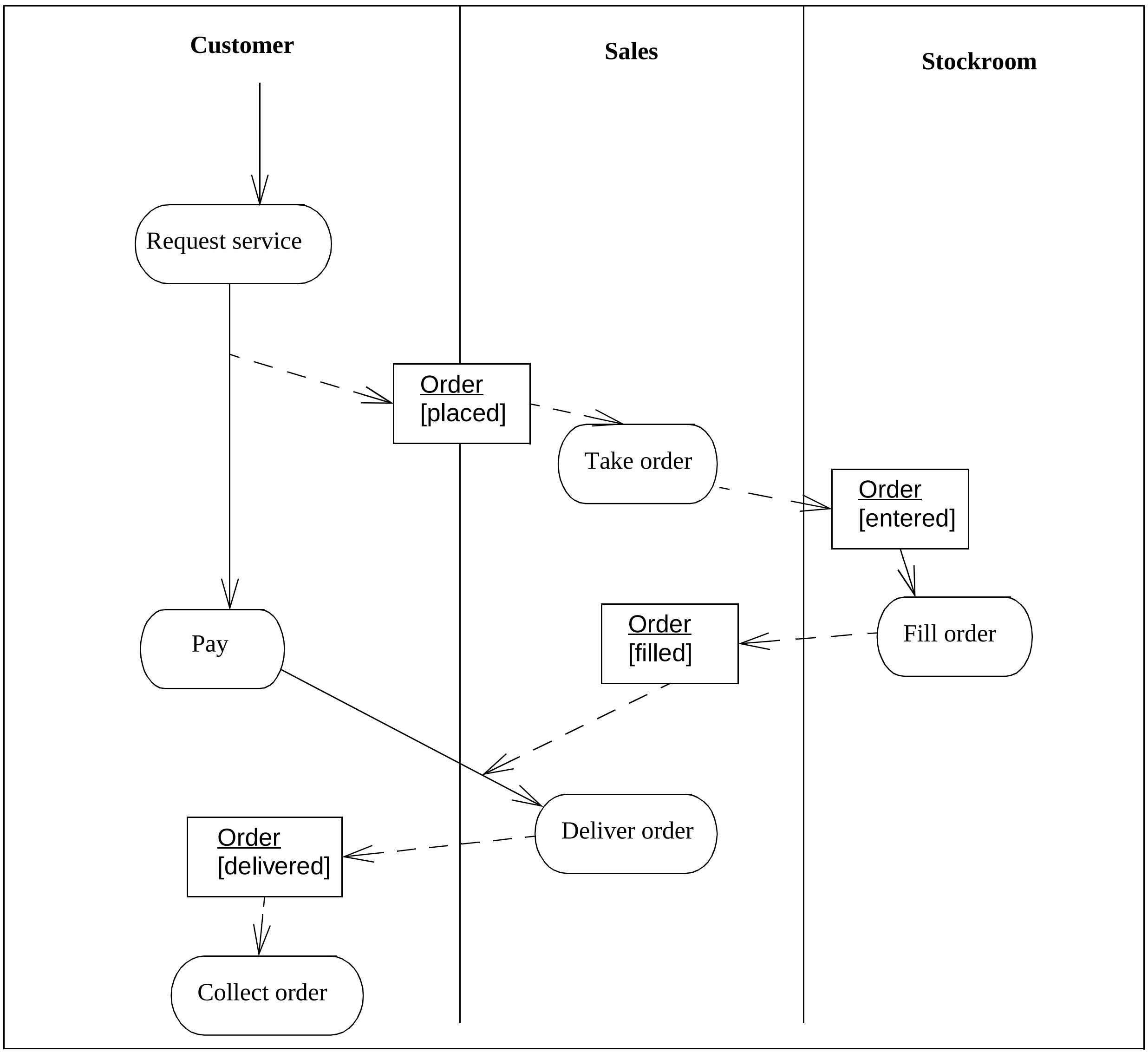
Object flow. Objects that are input to or output by an action may be shown as object symbols. A dashed arrow is drawn from an action outgoing transition to an output object, and a dashed arrow is drawn from an input object to an incoming arrow of an action. The same object may be (and usually is) the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

The control flow (solid) arrows 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 in 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]. This notation may also be used in collaboration diagrams.

10.5.3 Example

Figure 56. Actions and object flow



10.5.4 Mapping

An object flow symbol maps into an ObjectFlowState whose incoming and outgoing Transitions correspond to the incoming and outgoing arrows. The Transitions have no attachments. The class name and (optional) state name of the object flow symbol map into a Class or a ClassifierInState with the given name(s).

10.6 CONTROL ICONS

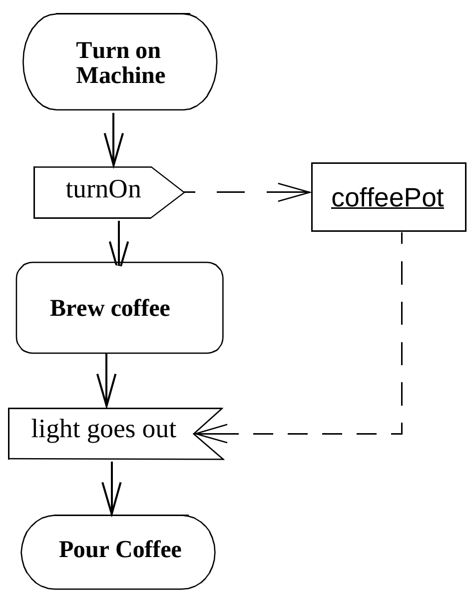
The following icons provide explicit symbols for certain kinds of information that can be specified on transitions. These icons are not necessary for constructing activity diagrams but many users prefer the added impact that they provide.

10.6.1 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 57. Symbols for signal receipt and sending

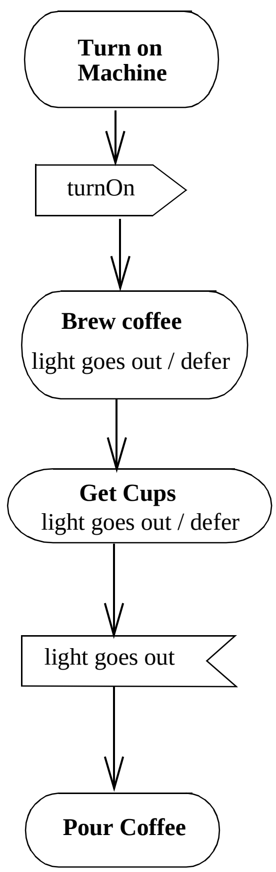


Deferred events. A frequent situation is when an event that occurs must be “deferred” for later use while some other activity is underway. (Normally an event that is not handled immediately is lost.) This may be thought of as having an internal transition that handles the event and places it on an internal queue until it is needed or until it is discarded. Each state or activity specifies a set of events that are deferred if they occur during the state or activity. If an event is not included in the set of deferred events for a state, then it is discarded from the queue even if it has already occurred. If a transition depends on an event, the transition fires immediately if the event is already on the internal queue. If several transitions are possible, the leading event in the queue takes precedence.

A deferred event is shown by listing it within the state followed by a slash and the special operation *defer.* If the event occurs, it is saved and it recurs when the object transitions to another state, where it may be deferred again. When the object reaches a state in which the event is not deferred, it must be accepted or lost. The indication may be placed on a composite state, in which case it remains deferred throughout the composite state.

When used in conjunction with an action state, a deferred event that occurs during the action state is deferred until the action is completed, when it may trigger a transition. This means that the transition will occur correctly regardless of the relative order of the event and the action completion.

Figure 58. Deferred event



10.6.2 Mapping

An input event symbol maps into an event trigger on the Transition between the States corresponding to the connected state symbols.

An output event symbols maps into a RaiseAction on the Transition between the States corresponding to the connected state symbols.

An input event symbol whose successor is a join symbol maps into an event trigger on a Transition to an implicit dummy State; the outgoing Transition from the dummy State enters the join Pseudostate.

A deferred event attached to a state maps into a *deferredEvent* association from the State to the Event.

11. 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.

11.1 COMPONENT DIAGRAMS

11.1.1 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).

11.1.2 Notation

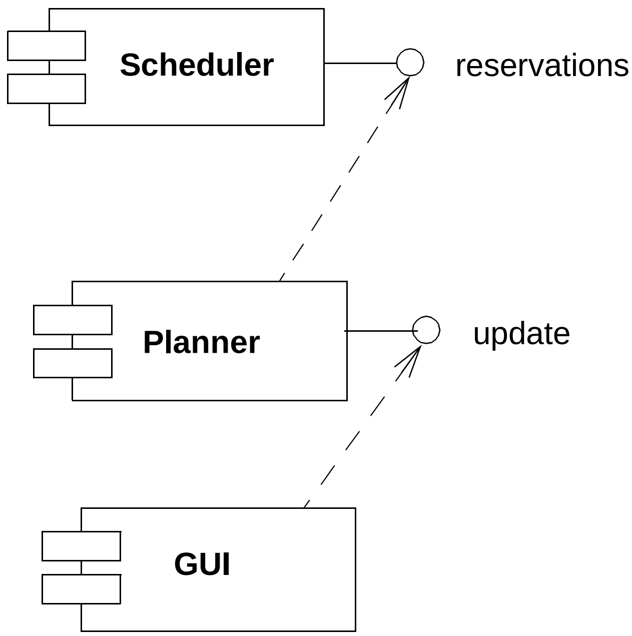
A component diagram is a graph of components connected by dependency relationships. Components may also be connected to components by physical containment representing composition relationships.

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.

11.1.3 Example

Figure 59. Component diagram



11.1.4 Mapping

A component diagram maps to a static model whose elements include Components.

11.2 DEPLOYMENT DIAGRAMS

11.2.1 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.

11.2.2 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.

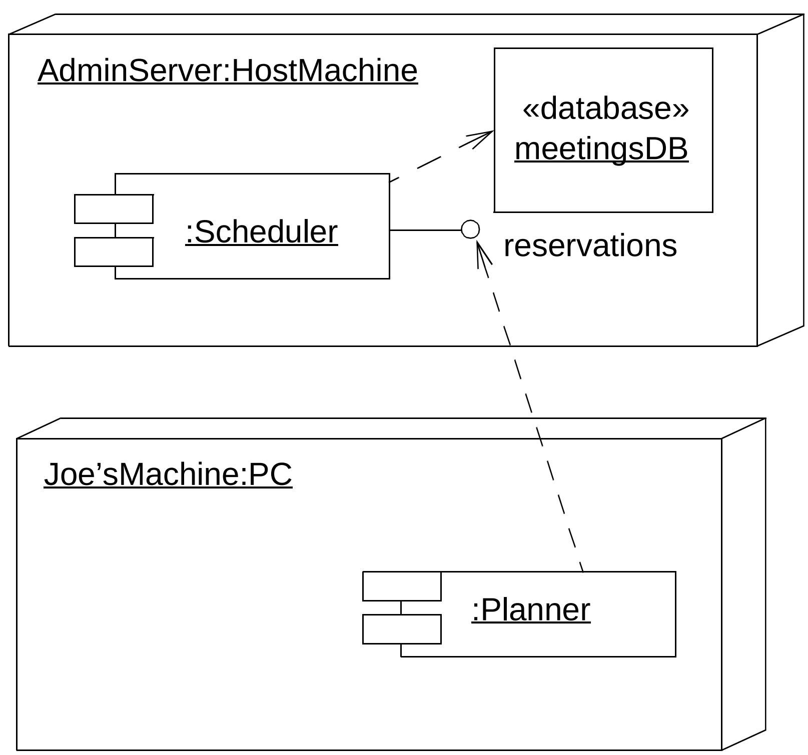
The deployment type diagram may also be used to show which components may run on which nodes, by using dashed arrows with the stereotype «supports».

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).

11.2.3 Example

Figure 60. Nodes



11.2.4 Mapping

A deployment diagram maps to a static model whose elements include Nodes. It is not particularly distinguished in the model.

11.3 NODES

11.3.1 Semantics

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

11.3.2 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.

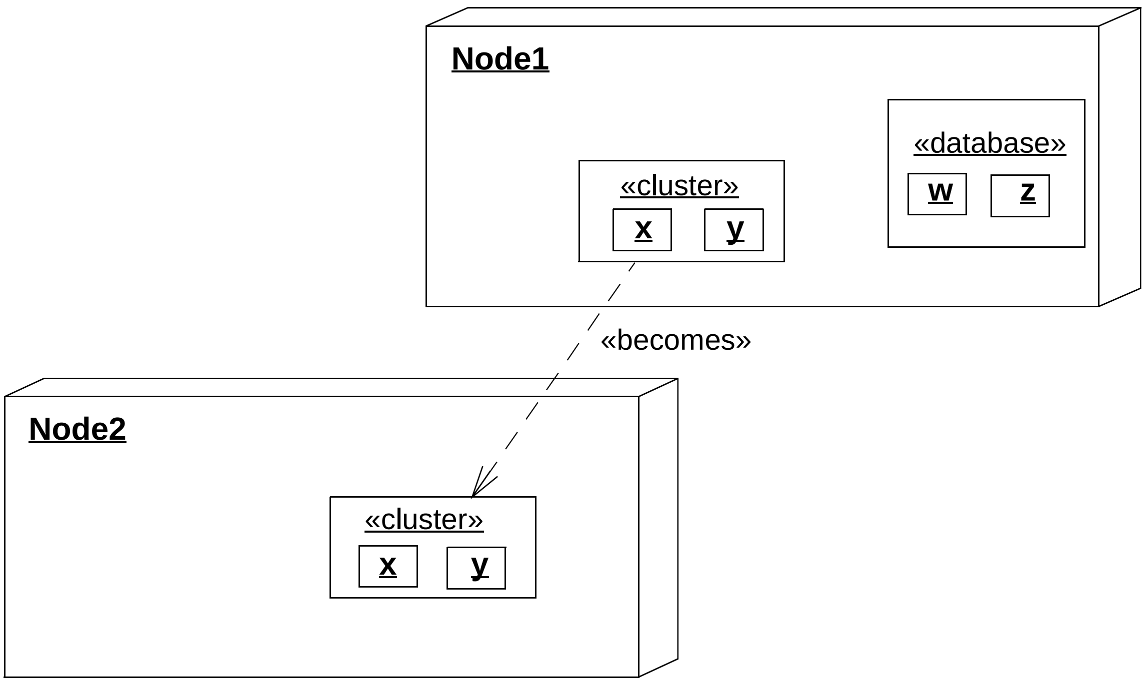
Component instances and objects may be contained within node instance symbols. This indicates that the items reside on the node instances. Containment may also be shown by aggregation or composition association paths.

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).

11.3.3 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 61. Use of nodes to hold objects



11.3.4 Mapping

A node maps to a «node» stereotype of a Class or Object. The nesting of symbols within the node symbol maps into a composition association between a node class and constituent classes or a composition link between a node object and constituent objects.

11.4 COMPONENTS

11.4.1 Semantics

A component type represents a distributable piece of implementation of a system, including software code (source, binary, or executable) but also including business documents, etc., in a human system. Components may be used to show dependencies, such as compiler and run-time dependencies or information dependencies in a human orgzanization. A component instance represents a run-time implementation unit and may be used to show implementation units that have identity at run time, including their location on nodes.

11.4.2 Notation

A component is shown as a rectangle with two small rectangles protruding from its side.

A component type has a type name:

component-type

A component instance has a name and a type. The name of the component and its type may be shown as an underlined string either within the component symbol or above or below it, with the syntax:

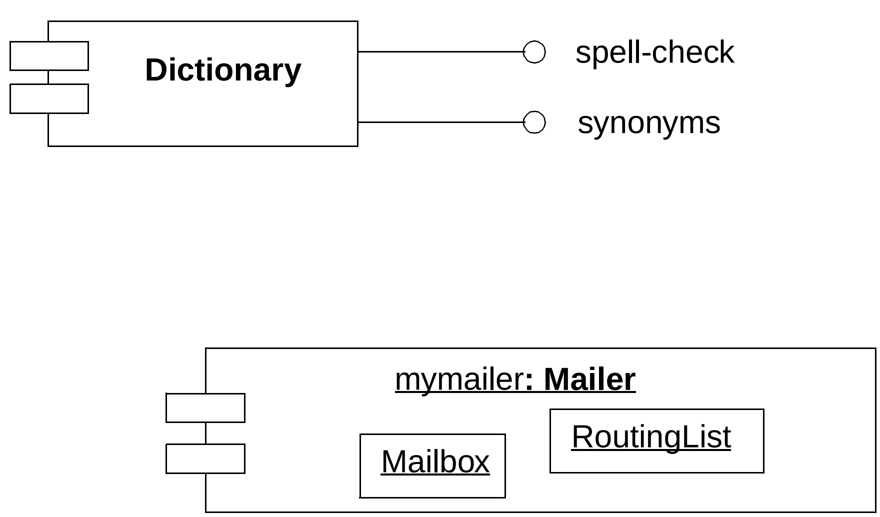
component-name ':' component-type

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

11.4.3 Example

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

Figure 62. Component



11.4.4 Mapping

A component symbol maps into a «component» stereotype of a Class or an Object. Graphical nesting of other symbols maps into composition association of the Component to Classes or Objects in it.

Interface circles attached to the component symbol by solid lines map into *supports* Dependencies to Interfaces.

11.5 Location of Components and objects within objects

11.5.1 Semantics

Instances may be located within other instances. For example, objects may live in processes that live in components that live on nodes. In more complicated situations processes may migrate from node to node, so a process may live in many nodes and deal with many components over time.

11.5.2 Notation

The location of an instance (including objects, component instances, and node instances) within another instance may be shown by physical nesting. Containment may also be shown by aggregation or composition association paths. 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.

11.5.3 Example

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

11.5.4 Mapping

Physical nesting of symbols maps into composition association from the Element corresponding to the outer symbol to the Elements corresponding to the contents.

Index

Index