# The
# **Unified Modeling Language**
# for Object-Oriented Development

Documentation Set
Version 0.91Addendum
UML Update

**Grady Booch  Ivar Jacobson  James Rumbaugh**

*Friday, September 27, 1996 1:14 pm*

RATIONAL
SOFTWARE CORPORATION

2800 San Tomas Expressway
Santa Clara, California 95051-0951
Telephone: 408-496-3600
Fax: 408-496-3636
E-mail: product-info@rational.com
URL: http://www.rational.com

Sales (U.S. and Canada)

(800) 728-1212

International offices

Australia +61-2-419-8455
Brazil +55-021-571-2978
Canada 613-599-8581
France +33-1-30-12-09-50
Germany +49-89-797-021
India +91-80-553-8082
Korea +82-2-579-8926
Sweden +46-8-703-4530
Taiwan +886-2-720-1938
UK +44-1273-624814

International representatives

Israel +972-3-531-3333
Japan +81-3-3779-2541

# 1. Overview

## 1.1 Purpose

This document is an addendum to the version 0.8 documentation set for the Unified Method, released in October 1995. The purpose of this version 0.91 document is to provide an update to our work which has evolved due to continued efforts at unification, public feedback, and research addressing new elements of modeling, namely, the issues of distribution, concurrency, and interfaces. This version 0.91 replaces the previous addendum version 0.9; it has the same content with some additional material and some slight revisions.

Much has happened since the release of the version 0.8 documentation set. Most importantly, Ivar has joined our team. The 0.8 documentation set, written by Grady and Jim, had addressed some elements of use case modeling, but now with Ivar as an equal partner we have been able to fully assimilate his work. This effectively means that the scope of our activities has grown to encompass the unification of the Booch, OMT, *and* OOSE methods. As part of this growth we have also renamed our work, changing it from the Unified Method (UM) to the Unified Modeling Language

(UML). This name change reflects the fact that we have chosen to decouple our work on notation and semantics from that of process. Section 2.1 explains this change in more detail.

As part of our continued efforts at unification, we have worked hard to make the UML simpler. This means that we have collapsed some related concepts into more general ones, we've made the notation more regular and even eliminated some symbols, and we've made the metamodel cleaner and smaller. Along the way, we've also found opportunities to reuse existing UML semantics in creative ways. This has enabled us to attack the problems of distribution, concurrency, and interfaces with minimal increase in the size of the UML

## 1.2 Organization

This document is an addendum to the version 0.8 documentation set, and as such does not provide a complete metamodel for the UML. Rather, it concentrates upon what's changed, what's new and different, and what's left to accomplish. Our version 1.0 documentation set, which we expect to deliver in late 1996/early 1997 in conjunction with our OMG submission, will provide a complete metamodel.

The core of this document is organized in three major sections:

| | |
|---|---|
| 2. What's Changed | This section addresses various naming changes (including the change to the UML designation), changes to the syntax of stereotypes, and changes to the syntax and semantics of packages and nodes. |
| 3. What's New | This section addresses the semantics of stereotypes and the integration of use cases (two improvements to the UML) as well as the semantics of interfaces, distribution, and real time modeling (new features to the UML). |
| 4. Tentative Proposals | This section includes some proposals that are not yet final, but are being considered for inclusion in the UML. Readers are invited to comment on these proposals. |
| 5. What's Left | This section provides a schedule of UML developments over the next several months and the work that remains to be done. |

## 1.3 Acknowledgments

Since the publication of the 0.8 documentation set for the Unified Method, we have distributed several thousand copies and have received feedback from hundreds of individuals from around the world. Thank you: your comments have given us much valuable guidance. Not only have you told us what you've liked, but you've told us what we've needed to fix, and what we were missing. Because of the volume of comments, we simply have not been able to respond to every message personally, but be assured that we are still tracking every comment we have received, and we are striving to address all substantial issues. Unfortunately, we can't acknowledge every contribution personally, but we'd like to give special thanks to a few individuals for their detailed feedback, namely, Michael Chonoles, Peter Coad, Bruce Douglas, Don Firesmith, Martin Fowler, Paul Kyzivat, Jim Odell, Dan Tasker, Jeff Sutherland, and various groups of developers at AG Communication Systems and Andersen Consulting.

### 1.4 Points of Contact

Comments on any aspect of the UML should be sent directly to all three of its authors, preferably via e-mail. Our individual addresses are:


| | |
|---|---|
| Grady Booch | egb@rational.com |
| James Rumbaugh | rumbaugh@rational.com |
| Ivar Jacobson | ivar@rational.com |


You can also post a message to all three of us at once using the following address:

amigos@rational.com

Finally, you can send comments to us via snail mail, using the address for Rational's corporate offices as listed on the copyright page of this document.

### 1.5 Copyright

The UML is an open standard; it is not a proprietary Rational language. As such, the UML may be used freely by anyone, anywhere. We are actively encouraging other tool vendors, training firms, consulting firms, authors, and developers to adopt the UML so that there will be wide-spread support for all users of the UML. We have a copyright notice on this and other UML documents simply to prevent commercial for-profit reproduction. If you want to share copies of this document with a colleague, then simply make a complete copy and acknowledge its source. If you want to make hundreds of copies and then sell them or use them for a training course, then please talk to us first. If you want to use this material to build a tool, develop a new training course, write a book, or use in your projects for development, then we encourage you to do so, but please again acknowledge its source, and remember that the UML is almost (but not totally) finished and therefore some things might change. It is in the best interests of the market for there to be consistent support for and use of the UML; if you find holes or areas of ambiguity in using the UML, please contact us. Now is the time to address the remaining loose ends.

### 1.6 Changes from Version 0.9

This document replaces the previous document version 0.9, both of which represent addenda to the base document version 0.8. The following change have been made to version 0.9:

Use of constraints to specify details of generalization and powertypes.
Change in notation for objects and other instance elements.
Clarification of the definition of *event*.
Definition of activity diagrams as a refinement of state diagrams.
Clarification of he relationship among use cases, interactions, and patterns.

## 2. What's Changed

### 2.1 Name Changes

There are two major name changes since the version 0.8 documentation set: the name of the UML itself, and the names of certain diagrams.

In the 0.8 documentation set, we referred to our work as the "Unified Method." We have decided to change the name to the "Unified Modeling Language," or "UML" for short. We made this change for three reasons:

- Feedback from the market indicated that we would add the most value by focusing on stabilizing the artifacts of software development and not the process.
- Our unification efforts were already focusing upon the graphical modeling language and its semantics and far less so on the underlying process.
- The UML is intended to be a universal language for modeling systems, meaning that it can express models of many different kinds and purposes, just as a programming language or a natural language can be used in many different ways. Thus, a single universal process for all styles of development did not seem possible or even desirable: what works for a shrink-wrap software project is probably wrong for a one-of-a-kind globally distributed, human-critical family of systems. However, the UML can be used to express the artifacts of all of these different processes, namely, the models that are produced.

Our move to the UML does not mean that we are ignoring the issues of process. Indeed, the UML assumes a process that is use case driven, architecture-centered, iterative and incremental. It is our observation that the details of this general development process must be adapted to the particular development culture or application domain of a specific organization. We are also working on process issues, but we have chosen to separate the modeling language from the process.

By making the modeling language and its process nearly independent, we therefore give users and other methodologists considerable degrees of freedom to craft a specific process yet still use a common language of expression. This is not unlike blueprints for buildings: there is a commonly understood language for blueprints, but there are a number of different ways to build, depending upon the nature of what is being built and who is doing the building. This is why we say that the UML is essentially the language of blueprints for software.

The names of four diagrams have been changed. In version 0.8, we referred to "message trace diagrams" and "object message diagrams." In the UML, we now refer to these two diagrams as "sequence diagrams" and "collaboration diagrams" respectively. We chose the name "sequence diagram" because it was a more general term (such diagrams involve more than just events) and because it emphasized its focus on the time-ordered sequence of transactions. We chose the name "collaboration diagram" because it was a more general term (the term "object diagram" had gotten terribly overloaded) and because it emphasized its focus on the patterns of collaboration among sets of objects.

Additionally, in version 0.8, we referred to "module diagrams" and "platform diagrams." In the UML, we now refer to these two diagrams as "component diagrams' and "deployment diagrams" respectively. In the case of component diagrams we have made the name change because ongoing use has led us to realize that these diagrams are more generally useful in modeling all sorts of physical components (such as dynamic libraries) as well as more static files. In the case of deployment diagrams, we have made the name change because the term "platform' turned out to be highly overloaded, and the term "deployment" reflected the diagram's true semantics, namely, the topology of the deployed system.

## 2.2 Syntax of Stereotypes

In the UML, the role of stereotypes and properties has been greatly expanded, for they have proven to be powerful mechanisms that are both general and extensible. Section 3.2 explains how the semantics of stereotypes and properties have been improved. Also, the syntax of stereotypes has been changed. In the 0.8 documentation set, we designated a stereotype by enclosing the stereotype name in parenthesis. Because parentheses are used in many other places and are not visually distinctive, we have changed the notation for stereotypes to guillemets (`«»`), which bracket the stereotype name as in `«exception»`. For convenience a guillemet can be typed as two angle-brackets but most typefaces support them as single characters. The stereotype notation can be used to mark an element, such as a class, package, or association. It can also be used within a list (such as a list of attributes or operations) in which case it applies to the succeeding elements in the list until countermanded.

We had considered the use of the exclamation point (!) to introduce a stereotype, but we eventually dropped this idea, for too many people confused it with the not operator found in many programming languages.

Figure 1 provides an example of the syntax for stereotypes.

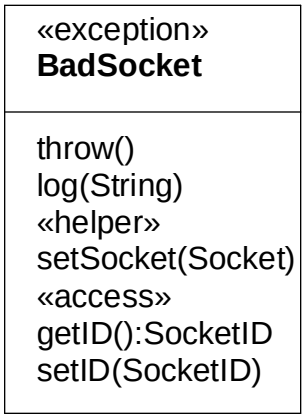


Figure 1: The Syntax of Stereotypes

In this figure, we see a class named `BadSocket` whose stereotype is `exception`. By stylistic convention, we put the name of the class in bold face, while other elements of the class are written in normal face.

The same stereotype notation may be used to group operations. For example, the operations `getID` and `setID` are classified as `access` operations, and the operation `setSocket` is a `helper` operation. `Access` and `helper` both qualify as operation stereotypes, because they are in effect metaclassifications of each operation and describe how the operations are used within the model. For example, access operations can be generated automatically from the attributes and helper operations are meant for internal use only. Operations `throw` and `log` have no stereotype; they are just ordinary public operations.

## 2.3 Generalization

Figure 2 illustrates one minor change to the generalization relationship: we now render this as a directed line with a closed, unfilled triangular arrowhead at the superclass end (in the 0.8 documentation set, we rendered this as a filled triangle).

We have shown another stylistic convention here. Specifically, we have used italic font for abstract classes (and operations) and normal font for concrete classes (and operations). This convention is a shorthand notation for including the property "abstract" with the class name (or operation signature), but it may be particularly convenient for large lists of classes and operations.

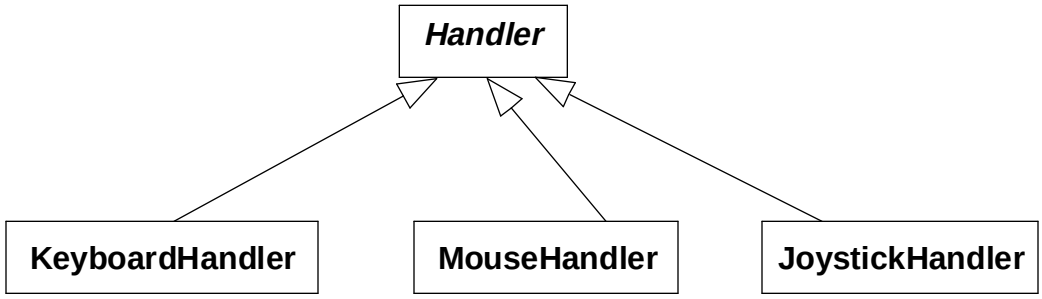


Figure 2: Generalization

Figure 3 shows a change in syntax from the V0.8 definition of "and-generalization." And-generalization occurs when a superclass generalizes more than one independent dimension. Different dimensions represent orthogonal abstract ways of describing an object. Each dimension is an incomplete view of the superclass. Concrete classes are formed by multiple inheritance as a Cartesian product of the different dimensions. In other words, a concrete class has a list of superclasses, one from each dimension of generalization of the original abstract class. For example, the class `Sailboat` is a subclass of `WindPoweredVehicle` and `WaterVehicle`; the class `Truck` is a subclass of `MotorPoweredVehicle` and `LandVehicle`. This construct is semantically equivalent to forming the Cartesian product of the different dimensions, even if all combination subclasses (such as `Sailboat` and `Truck`) are not shown. In V0.8 we required a dummy class for each dimension, which was unnatural and unworkable in a system with many packages. Now we allow a *discriminator* label to be attached to a generalization arc. If several generalization arcs share the same label, then they represent the same dimension of generalization of the superclass; different labels define independent dimensions of generalization. (The "empty" label is just a particular label, so that a generalization without any labels is just a special case of a dimension.)

Note that constraints can be used to indicate relationships among the different subclasses (Figure 4). If all of the subclasses of a given superclass are completely disjoint, use the constraint "{disjoint}". This indicates that no descendent of the superclass may be a descendent of more than one of the subclasses. For example, no Account may be both a BusinessAccount and a PersonalAccount. If the subclasses are not mutually exclusive, then use the constraint "{overlapping}". This indicates that a descendent of the superclass may be an instance of more than one subclass. For example, a Vehicle may be both a LandVehicle and a WaterVehicle. This would require either multiple inheritance (for example, a class AmphibiousVehicle) or else multiple classification within the existing class structure. The constraint is shown by drawing a dotted line across the affected generalization lines and attached the constraint keyword to it. The constraint may be applied independently to the arcs in separate dimensions of and-generalization. If the generalization lines are

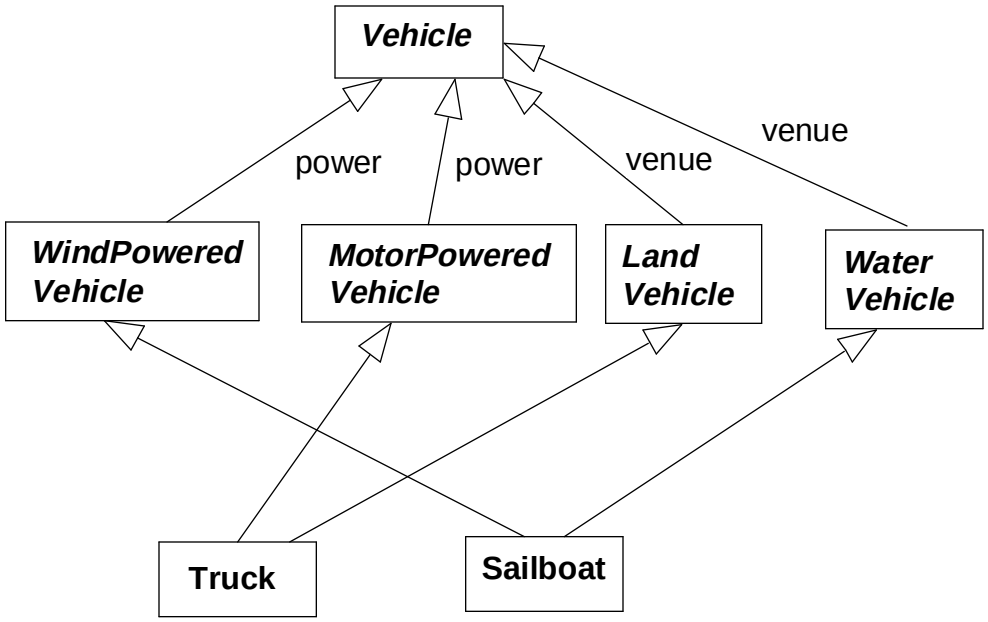


Figure 3: And-generalization with discriminator labels

drawn so that they share a single arrowhead segment (a stylistic option), then the constraint keyword can be attached to the arrowhead segment directly. The absence of a constraint leaves the issue unspecified. (The presence of overlapping classes can be inferred when multiple inheritance occurs in the class diagram.) Other constraints can be applied to sets of generalization lines. For example, the constraint {complete} might indicate that the generalization is complete and further subclasses may not be added.

Further clarifying the semantics of generalization, we do not assume or preclude the concepts of dynamic classification and multiple classification (the terms are due to Odell). These are very useful ideas for analysis, but some users may choose to forgo them because they are not directly supported by the leading OO programming languages. Concepts such as dynamic and multiple classification are properties of the dynamic environment; the same static class model can be used with either assumption. There are many other properties of the dynamic environment that different users may want to vary. We do not feel that a single definition of dynamic semantics will work for all purposes. Accordingly, as part of our work toward formal specification of UML semantics we are investigating mechanisms to support “pluggable semantics” that permit the language to be extended. These mechanisms could be used to tune the execution semantics. (See section 3.4 for an example using the «becomes» relationship to represent dynamic classification in action.)

Stereotypes and constraints can be combined to define “power types” (Odell has used these extensively in his writings). A power type is a metaclass whose instances are a set of sibling subclasses. For example, the class TreeSpecies has as instances the classes Oak, Elm, Birch, etc., which are the subclasses of class Tree. A power type may be indicated with the stereotype «powertype»; it may be attached to a set of generalization arcs or to the shared root of a generalization tree by indicating its name as the type of the discriminator (Figure 5). A discriminator may have an attribute name (it represents an implicit attribute in the superclass), a type name (it represents the name of the power type whose instances are the subclasses themselves), or both.

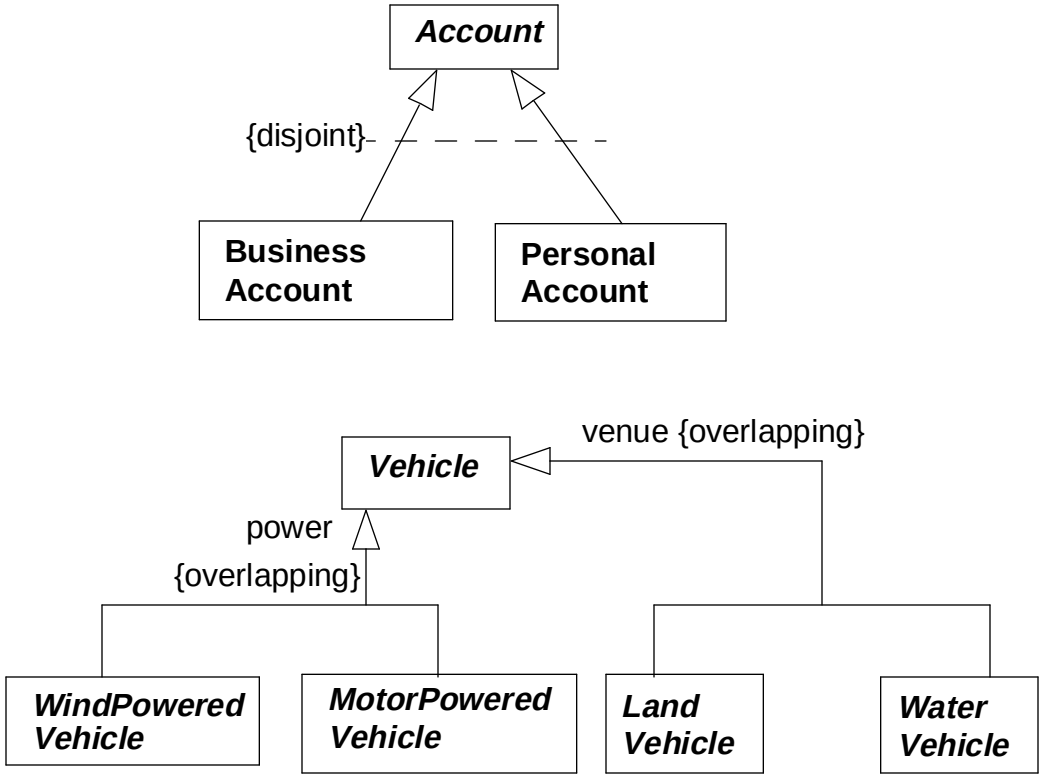


Figure 4: Subclass constraints (various drawing styles)

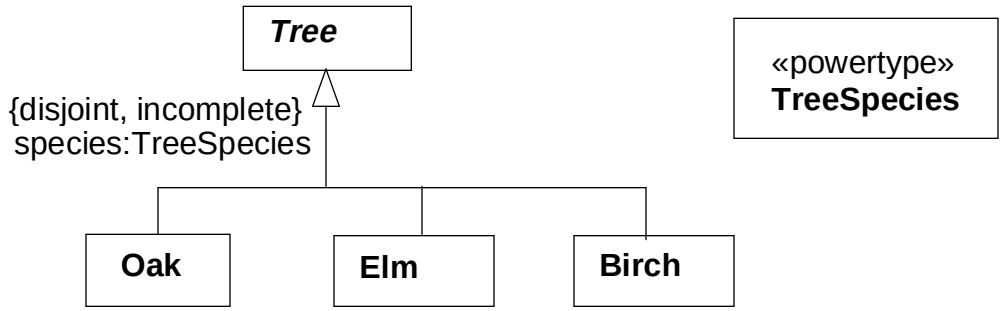


Figure 5: Showing power types

In the final UML document we will suggest some hints to tool builders to provide optional notation to enhance user comprehension of models. One such suggestion is an optional notation for showing an incomplete generalization, i.e., a diagram in which some subclasses of a superclass are shown but others are missing from a particular diagram (presumably they are shown on another diagram). We suggest the ellipsis symbol "…" to explicitly indicate that some subclasses are missing from a diagram. We propose that editing tools automatically generate or suppress this symbol as appropriate (i.e., it is a statement that the *view* omits something, not a statement that the model lacks something). This convention could also be used for associations, attributes, operations, etc. To avoid visual overload such visualization options must be dynamically selectable.

## 2.4 Association Navigation

In the version 0.8 documentation set, we defined *navigability* as an element of the role affiliated with each end of an association relationship; however, we combined its symbol with the by-value/by-reference distinction. In the UML, we have decided to designate navigability with an open arrow, as illustrated in Figure 6. Navigability is visually distinguished from inheritance, which is denoted by an unfilled triangular arrowhead icon near the superclass.

In Figure 6, we have marked the association as navigable only in one direction, namely, from the `Handler` to the `Client`, but not the reverse. This might indicate a design decision, but it might also indicate an analysis decision to partition the system into packages such that the `Client` class is frozen and cannot be extended to know about the `Handler` class, but the `Handler` class can know about the `Client` class.


Figure 6: Association Navigation

We chose this particular notation for navigability for two reasons:

- It was consistent with the use of the open arrowhead to show directionality in all other uses.
- It was visually striking, but not so overwhelming that it stood out as a primary element.

A related change is a reassessment of the "by-value" adornment for associations. A "by-value" implementation always implies aggregation: the value is physically part of the other object, so it must be an aggregation. Therefore it is dangerous and unnecessary to allow "by-value" to be specified separately from aggregation. We have realized that this is really an adornment on an aggregation, a "tightly-bound" implementation of an aggregation. Therefore, the notation for "by-value" implementation is now a solid (filled) diamond on the aggregation symbol; a hollow diamond implies a "by-reference" aggregation, the normal default. The small squares to mark "by-value" and "by-reference" are thus subsumed by this approach. Note that a "by-value" aggregation is semantically equivalent to an attribute, but may be visually more appealing when the part has its own internal structure.

There are certain rules on compatible adornments of associations:

- Only one side (at most) of an association can be an aggregate.
- If one side is a "by-value" aggregation, then the association is navigable to the other side.

## 2.5 Packages

In the version 0.8 documentation set, we used two different grouping mechanisms—categories for the logical model and subsystems for the code model—each with a distinctive icon. In the UML, we decided to collapse these two mechanisms into one all-purpose packaging construct, which can also be used for other groupings of modeling elements, such as use cases and processors. We call such a grouping a *package* and we draw it as a familiar desktop, namely, the "tabbed folder." We made these changes for three reasons:

- We needed a grouping mechanism elsewhere, and found that we kept adding new mechanisms that were all very similar.

- The semantics of categories and subsystems were similar, and use of stereotypes made it possible to introduce a more general concept yet retain some distinctions.

Figure 7 provides an example of several packages. In this figure we see four packages: `Clients`, `Business Model`, `Persistent Store,` and `Network`. In this sample diagram we show two classes inside the `Business Model` package, together with one nested package, `Bank`. A real model would have many more classes in each package. The contents might be shown if they are small, or they might be suppressed from a high-level diagram showing the system architecture. The entire system is a package.

This figure shows a fairly regular hierarchical structure, with packages dependent upon other packages. As in the 0.8 documentation set, we use the dependency relationship to show design and implementation dependencies. For example, we note that the package `Clients` depends upon the packages `Business Model` and `Network` directly, meaning that one or more elements within `Clients` depends on one or more elements within the other packages. Three of these outermost packages are shown in an elided form (and by convention we place their name in the body of the icon to save space), whereas the package `Business Model` is shown partially expanded. In this case, we see that the package `Business Model` owns the classes `Customer` and `Account` as well as the package `Bank.` Ownership may be shown by a graphical nesting of the icons or by the expansion of a package in a separate drawing (which might be more convenient in an on-line tool). Thus, it is possible to show relationships to the containing package as well as to nested elements.

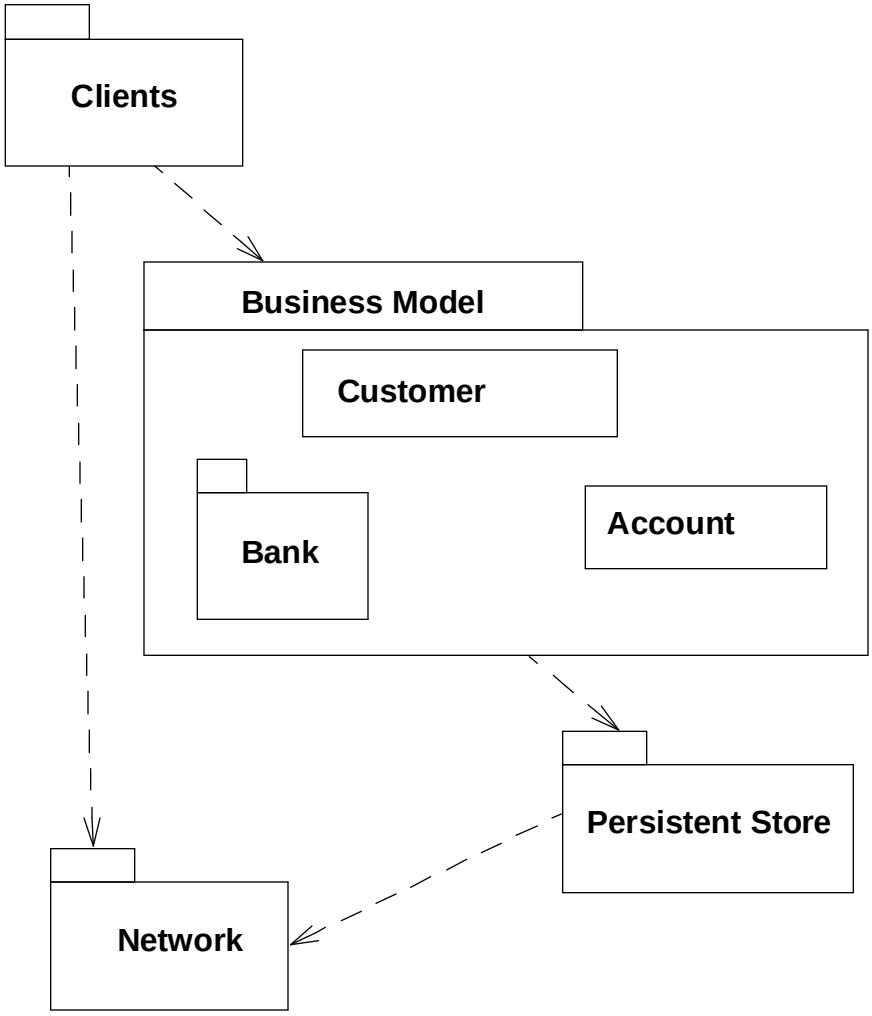


Figure 7: Packages with Dependencies

The semantics of packages do not change from the 0.8 documentation set: a package owns its contents and defines a nested name space for its contents. This means that every class (or other element) may belong to exactly one package. In other words, packages partition the elements in a model. A package may contain *references* to classes owned by other packages. In cases when the other packages are not shown, such references must be marked with fully qualified names, in the form `PackageName::ClassName`. A package can add associations to referenced classes but cannot modify their contents (attributes and operations). The navigability of associations within a package must be compatible with the visibility of the underlying classes.

Packages provide an organizational structure for the model, including grouping, naming, and configuration control units. Otherwise, packages do not add semantics beyond those of their contents. However, they may be used to understand models by summarizing semantics derived from their contents. Used in a top-down fashion for design, packages permit designers to constrain properties of their contents, such as dependencies on other elements, and therefore they can be used to specify semantics of groups of elements at a high level.

Packages turn out to be a wonderfully general mechanism for organizing models. They may be used to designate not only logical groupings and physical ones, but they may also be used to designate use case groups and processor groups. A use case group and a processor group is, as the names suggest, a packaging of use cases and processors, respectively. In each case, the usual semantics of package ownership apply. Also as usual, stereotypes may be used to distinguish one kind of package from another.

## 2.6 Nodes

The version 0.8 documentation set adopted the use of platform diagrams, which in 0.91 we now name deployment diagrams. These diagrams contained two different symbols, one designating a processor and the other designating a device. We distinguished these two semantically and iconically. In particular, processors had computational resources and thus could run processes and hold objects, whereas devices could not. Both processors and devices were rendered as a three-dimensional rectangular shape, but with processes having shaded sides. In the UML, we decided to collapse these two concepts into one. We made this change for two reasons:

- In real systems, it is rare to encounter any device that has no computational elements. What looks like a device to one system probably looks like a processor to another.
- Improvements in the semantics of stereotypes made it possible to introduce a more general concept.

In the UML, we call the elements that contain resources (including CPUs and memories) nodes. A node thus represents a resource in the real world upon which we can distribute and execute elements of our logical models. A node is rendered as a three-dimensional rectangular solid with no shadows. This icon is simple to draw and still conceptually different from all other elements, which are all drawn as two-dimensional shapes. It also conveys the idea of a "physical presence."

Figure 8 provides an example of a deployment diagram. In this figure we see six nodes. Two of these nodes (`Fax` and `Printer`) are designated with the `«device»` stereotype since, in the context of this system, they are not computationally interesting; it is of course not required to show a node's stereotype. Three of the remaining nodes are adorned with roles with respect to the server. For example, there are a set of `PC` nodes for `Order Entry`. Nodes are classes and thus can have the properties that classes have. In this case we have shown the multiplicity of each class within the entire system: many order entry PCs and many printers and one of each other node. (We could also draw an instance-level version of this diagram to show a particular deployment of nodes in an individual instance of a system.) Notice also that we have used the stereotype notation to distinguish different kinds of connections; in this case the system employs ISDN connections.

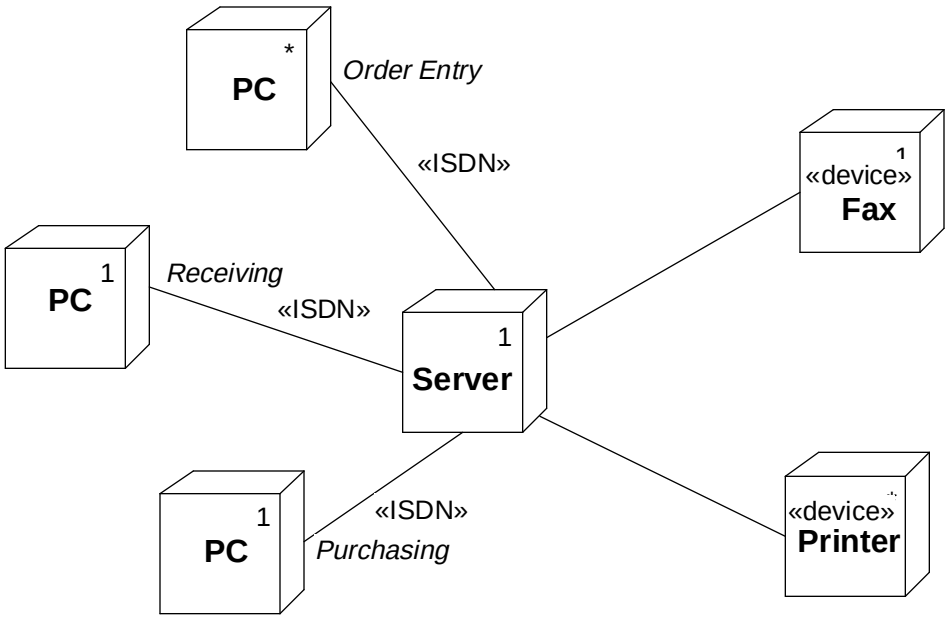


Figure 8: Nodes in a Deployment Diagram

## 2.7 Notation for Objects

Classes and objects are a pair of elements that have a type-instance distinction. This distinction is also possible for many other kinds of elements, such as use cases, operations, states, etc. Rather than trying to invent new symbols each time this distinction must be made, we have decided to adopt a uniform way of mapping between types and instances: For any kind of element, the same graphic symbol is use for the type construct and the instance construct. However, within the instance construct the designator string (the instance name, colon, and the type name) is underlined. For example, a class is drawn as a rectangle containing the name of the class; an object is drawn as a rectangle with an underline name of the object name and its class name (separated by a colon). See Figure 9 and other diagrams for an example of the notation for objects.

## 2.8 Conditionals in Interaction Diagrams

A scenario is a single execution history of an interaction. Because collaboration diagrams and sequence diagrams did not have conditional branches in version 0.8, they were restricted to showing scenarios. This meant that they could only show individual executions of a system and not the general interaction pattern. We have realized that there is no reason why the general interaction pattern should not be shown. Therefore, we have added a notation for conditionals to collaboration diagrams and sequence diagrams. Accordingly, we collectively call them *interaction diagrams* because they document general interactions.

In a collaboration diagram, a conditional is indicated with a guard condition in square brackets (Figure 9), the same notation used for synchronizing with another thread (which is also a kind of guard condition on execution), and the same notation used for guard conditions in state diagrams. In addition, each arm of the condition is labeled as a separate thread, the same as a concurrent

thread. A branch is simply a pair of parallel threads in which only one thread is chosen on any given pass; a concurrent fork is a pair of parallel threads in which both threads are chosen. Therefore a given named sequence in the messages indicates a single thread of control that is always executed sequentially.

In a sequence diagram, a conditional is indicated by splitting a message arrow into two parallel targets (Figure 10). Note that in the x=0 case in this example, neither branch is taken. As with finite state machines, at any given branch point, the conditional expressions in an interaction diagram must be unambiguous. A branch of control may require "splitting" the target object into two separate, parallel traces that represent alternative histories. This notation works if the subsequent control patterns are not too complicated. In cases of nested conditionals, it is better to "split" the entire sequence diagram by duplicating it or by separating the subsequences into separate subdiagrams.

We make the observation that a conditional is merely a fork of control into two (or more) parallel threads that may not both execute together. In other words, conditionality and concurrency are closely related.

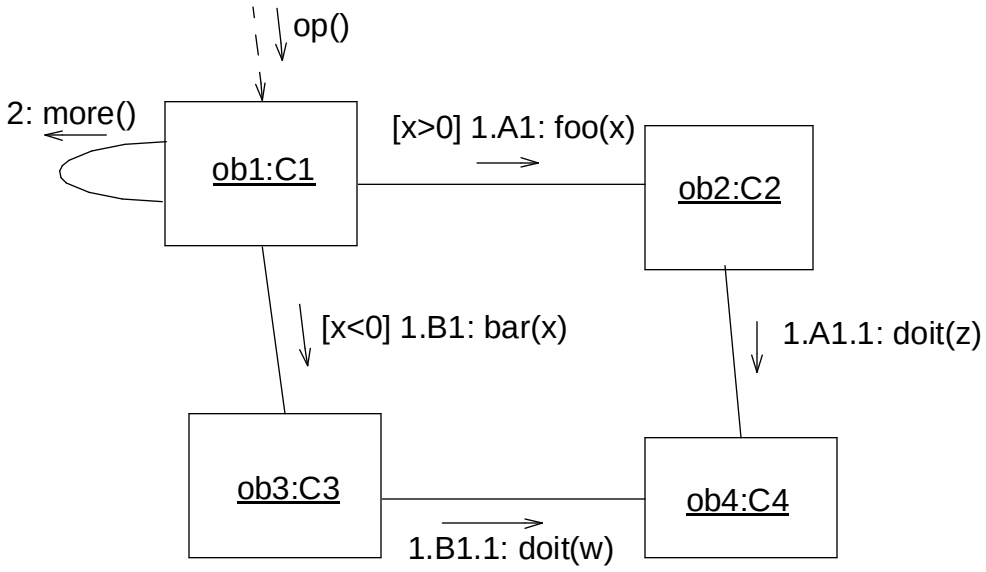


Figure 9: Collaboration Diagram with Conditional

Figure 10 also shows a recursive call (on the operation "`more`"). We have decided to use the intuitive notation from the Siemens pattern group's Object Message Sequence Charts[1] (OMSC) in which recursive calls are shown by stacking multiple activities on the object lifeline. Another aspect from (OMSC) is the notation for creation and destruction of objects in the sequence diagram: The object icon is drawn level with the arrow for the operation that creates it; a large "X" indicates the destruction of an object, usually by external command but in this case a self-destruction as the

1. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley, 1996, ISBN 0471958697.

last act of a method before returning. An arrow to a vertical dotted line indicates a call to a pre-existing object.

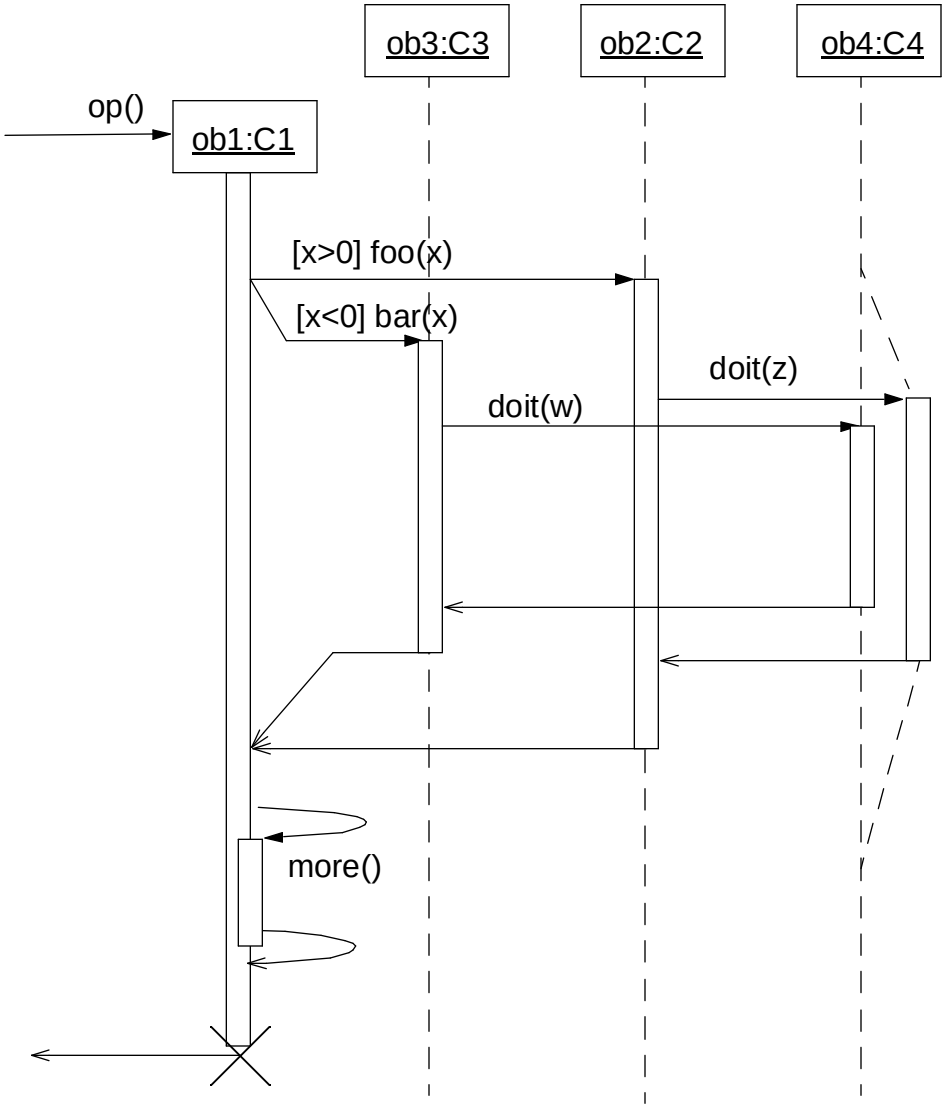


Figure 10: Sequence Diagram with Conditional, Recursion, Creation, and Destruction

## 3. What's New

### 3.1 Semantics of Stereotypes and Properties

In the version 0.8 documentation set, we introduced a general property mechanism for all UML elements. The purpose of this mechanism was to provide a simple yet expressive means to extend the UML in ways that we could not yet then imagine. Part of this property mechanism included the

concept of a stereotype, which was essentially the metaclassification of a UML element or, as we sometimes called it, the property that had no name.

Our initial work with stereotypes was motivated by a desire to distinguish various kinds of classes in a model (the word "stereotype" comes from Rebecca Wirfs-Brock). As we continued in our work on unification, we discovered that stereotypes were a far more powerful mechanism that we first realized. First and foremost, they helped us address the problem of metaclassification, such as distinguishing exception classes which should generally only be thrown or caught as well as distinguishing various kinds of analysis objects as found in OOSE (and which we further discuss in section 3.2). Second, they enabled us to collapse semantically similar concepts into one (such as nodes, as we discussed in sections 2.5 and 2.6), thereby simplifying the UML. Third, they allowed us to define core UML semantics very precisely, yet allowed end users some degrees of freedom in tailoring the language to their needs. For example, as we describe further in section 3.5, we already understand the most common forms of message synchronization (simple, synchronous, asynchronous, and various timed synchronization) but there is no way that we could ever specify all forms of synchronization, for not all of them yet exist. For example, the Java model of message synchronization is different than that of Ada's and it's likely that as experience with internet languages grow, other models will arise. In our approach, the UML can adapt to these new semantics without altering core UML semantics.

In the version 0.8 documentation set, we underspecified the semantics of stereotypes, largely because we didn't exactly understand their implications. Now, we can finally state their semantics more precisely. Specifically:

- A stereotype represents the metaclassification of an element. It represents a class within the UML metamodel itself, i.e., a type of modeling element. It is a way to allow users (methodologists, tool builders, and user modelers) to add new kinds of modeling types.
- Every element in the UML may have at most one stereotype; this stereotype is omitted when the normal predefined UML modeling semantics suffice.
- There is a separate list of stereotypes for each kind of UML element.
- We predefine some of these stereotypes, but we also allow users to define their own.
- Stereotypes have semantic implications which may be specified for every specific stereotype value. We are investigating ways to allow the semantics to be specified by users. Meanwhile the semantics can be stated in natural language or built into a particular editing tool.

A predefined stereotype is one whose value and semantics we define as part of core UML. We can think of at least two kinds of stereotypes: those with specific added semantics and those that provide convenient conceptual grouping but don't really add to the semantics. Our current list includes the following predefined stereotypes with semantics:

- Class and object stereotypes
  - Event — Designates a noteworthy occurrence that triggers transactions in finite state machine models (so that its structure may be shown)..
  - Exception — Designates an event that may be thrown or caught by an operation.
  - Interface — Designates the interface of a class or a package, consisting of a

set of message names and their signatures together with their dynamic semantics

- Metaclass — Designates the class of a class (as in Smalltalk).
- Utility — Designates a named collection of non-member functions.

- Task stereotypes
  - Process — Designates a heavy-weight task attached to its own address space.
  - Thread — Designates a light-weight task that executes with other threads in the same address space of an enclosing process.

The following lists are offered for convenience; they are not formally a part of the UML. They might be used in a tool to change the rendering, but they don't really have additional semantics beyond the nature of the things they contain:

- Package stereotypes
  - Category
  - Processor group
  - Module group
  - Subsystem
  - Service package
  - Use case group
- Node stereotypes
  - Device.
  - Processor
  - Memory
- Object type stereotypes (from OOSE; other methods have also defined similar lists)
  - Entity object type
  - Control object type
  - Interface object type

Relationships may be adorned with stereotypes as well. In fact, section 3.2 describes several other predefined stereotypes that map to concepts in OOSE, some of which are attached to class relationships.

Our list of predefined stereotypes is not yet complete. All of these lists will likely evolve before the final UML report is prepared.

Tagged values are extensible properties consisting of an arbitrary textual tag and a value. We are also compiling lists of recommended tags. Section 3.4 describes one new predefined tag, location, which denotes the node to which the item is attached. We are considering other tags, such as persistence, which would denote if an object's state is transient or sticky (or replicated or indeterminate; the list of possible values is open-ended).

There is one other important improvement to the notation for stereotypes that we have developed since the 0.8 documentation set. Specifically, we recommend that tools allow every stereotype to be further distinguished by style (color, line thickness, fill pattern, font) as well as by icon. The purpose of this improvement is to permit tool builders and end users to tailor the UML's graphical syntax in controlled ways, and to permit models to have special visual cues. For example, we might wish to have all exception classes stand out by rendering them in color. To do so, the user might designate the stereotype `exception` to be mapped to red, and then attach this stereotype to all

relevant classes. Because support for color varies among systems (and people!) use of color should be reserved to the individual user, rather than being predefined in UML.

On the other hand, attaching an icon to a stereotype is a bit more universal. In the UML, we permit users to attach an icon to a stereotype. (The details of specifying the image would be up to the editing tool.) In its canonical form, this icon may be displayed to the right of the stereotype value; Figure 11 provides such an example (middle). In its expanded form, this same item may be rendered simply with the icon and the item's name, as in Figure 11 (right side). In fact, the three items in Figure 11 are semantically equivalent; they are just rendered differently.

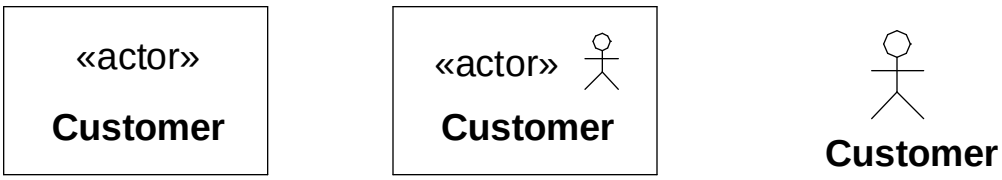


Figure 11: Stereotype Icons

Tools may permit and provide alternate views of UML symbols beyond those that we have provided. In fact, one of the major values of a tool is to provide ways for users to dynamically alter views of models to highlight various topics of interest. For example, the canonical form of a node is a three-dimensional rectangle; tools may render nodes in more domain-specific ways, such as by displaying the icon for a PC. Similarly, we have specified that a canonical class icon contains three compartments, one each for the name, the attributes, and the operations. For some purposes additional compartments are useful, for example, a list of business rules or a list of responsibilities of the class. It is permissible for tools to add additional compartments whose syntax must be specified by the toolmaker. We have already noted that compartments or selected contents of compartments should be suppressible within views. We support similar extensions in this spirit. However, with all extension mechanisms, including stereotypes, properties, view tailoring, special icons, etc., there is a danger of creating a private language that others cannot understand or use, so the benefits of special extensions must always be weighed against the loss of a common base.

### 3.2 Use Cases

Since the introduction of the 0.8 documentation set, we have worked to integrate the OOSE semantics of use cases fully into the UML. We had gotten the structural elements of use cases and scenarios fairly right, but we had not addressed the elements of OOSE's analysis and design models very well at all, nor had we considered the implications of robustness analysis. In doing this integration, we found that UML's stereotypes helped us add these features with little additional complexity.

We treat actors as classes with stereotypes for their semantics and notation. The stereotype icon for an actor is a stick figure, as in Figure 11. As with any stereotyped class, either the class icon or the stereotype icon can be used.

Generalization applies to actors and use cases. Concrete actors are specializations of abstract actors and concrete use cases are specializations of abstract use cases. Furthermore, actors may have (communication) associations with use cases, meaning that instances of the actor participate in instances of the use case. Use case classes may be related to other use cases classes by the *extends* and *uses* relationships; we regard these as stereotypes of the generalization relationship.

Interactions between actors and use cases can be described with both sequence diagrams and collaboration diagrams.Both actors and use cases can be described in words, of course. Indeed, it is particularly important to identify the purpose of a use case in words. There are also several other means of modeling use cases, for example, by listing their responsibilities, by listing their attributes and operations, and by defining their patterns of interaction using state machines.

Managing use cases at the user requirement level is fairly straightforward. The complexity of mapping use cases to design artifacts increases as a model progresses to the design stage. Some of these design artifacts include internal classes, packages, and interfaces. Interaction diagrams (sequence diagrams and collaborations diagrams) provide a way of doing this mapping, which can occur on several different levels. First, responsibilities of the use case are allocated to supporting classes, package, and interfaces in the design model. Attributes of the use case must be mapped onto attributes or associations of the supporting classes. Next each operation in the use case must be mapped onto operations on classes. Finally the flow of control among objects must be designed so that the proper operations are invoked at the proper times within the execution of the use case; this can be shown on interaction diagrams for the system as well as state diagrams for the constituent classes. The degree of formalism employed in the mapping depends on the development process followed. The UML does not prescribe a particular mapping between use cases and other entities. The mapping required by any particular process can be modeled using UML dependency relationships with appropriate stereotypes.

In the UML, integration of OOSE semantics also encompasses the following elements:

- The use of stereotypes to model interface, entity, and control objects
- The use of stereotypes to model the OOSE association stereotypes

Clarifying the relationship of use cases to scenarios and classes was largely a semantic issue, and had no notational implications. In 0.8, we had originally attached use cases to the class model directly, but we eventually realized this was wrong: use cases stand as peers to class models. The package scoping mechanism can accommodate various approaches to organizing system models, including independent use case models and class models as well as other possibilities. Finally, we may also attach state machines to use cases to definitively specifying their dynamic semantics.

In the 0.8 documentation set, we had rejected the stick figure icon for actors, and treated actors as objects. This was wrong: actors are classes. Actors may be modeled as a special stereotype of classes with their own stereotype icon. The UML is actually thus more general than OOSE, since a project could identify other stereotypes for classes that interact with use cases. For example, to model a system of interconnected systems, a project might introduce a stereotype denoting a system class.

Interface, entity, and control objects from OOSE can be handled in the same fashion. Basically, these are just stereotypes of classes, and so a project can predefine these three stereotypes when following an OOSE process. Other processes with different lists of predefined stereotypes can just as easily be handled.

OOSE's process has a rich set of association stereotypes; in practice, users of this process report that this set is necessary but sometimes insufficient. Again, using UML's stereotype mechanism, we can satisfy the OOSE association stereotypes but at the same time leave room for the introduction of other kinds of associations. The following table illustrates the mapping of OOSE associations into UML. Note that this mapping requires no new special notation:

| OOSE | UML | Stereotype |
|---|---|---|
| attribute | association | |
| consists of | association (aggregation) | |
| communicates | association | communication |
| subscribes to | association | subscribes to |
| acquaintance | association | |
| extends | generalization | extends |
| inheritance | generalization | |
| uses | generalization | uses |
| depends on | dependency | |

In OOSE, requirements are first class citizens, but there was no means of graphically rendering a requirement. In UML, we may use attach the stereotype `requirement` to a note, and thereby graphically reify this concept. Using existing UML semantics, we may thus show the dependencies of a requirement to other modeling elements.

### 3.3 Interfaces

In pushing the semantics the UML further, we found the need to model interfaces. An *interface* is a class stereotype that designates the external face of a class or a package, consisting of a set of operations and their signatures together with their dynamic semantics. Other classes may support or use an interface; we thus say that a class *conforms* to an interface in a particular *role.* The role describes the part played by each class within the interface; normal interfaces have *supplier* and *client* roles that complement each other. Such semantics are adequate to describe most libraries and frameworks, such as the interfaces found in Microsoft's COM as well as in Java's conception of interfaces. Whereas in COM and Java interfaces are largely only static named groups of operations, the UML permits attaching dynamic semantics, so that a partial ordering of operations legal for an interface may be specified. We call specification of legal activity sequences the *protocol* of the interaction. For real-time applications more complicated protocols are necessary to describe patterns of interaction, including multiple roles, two-way communication, and constraints on interaction sequences.

We examined a number of existing approaches to specifying interfaces and found a great deal of similarity among them. These include ROOM protocols, OOSE contracts, RDD contracts, Mi-

crosoft COM interfaces, and Helm, Holland & Gangopadhyay contracts. We have tried to incorporate the best of these ideas into a broadly-useful construct for describing interfaces.

An *interface* describes the legal patterns of interaction between two objects. There is a range of possible interaction complexity: at the simplest, an interface consists of a set of functions that can be called at any time in any order (a simple function library); a more complex interface is a set of functions with constraints on the order in which they can be called (a function library with setup functions, for example). An interface has *supplier* and *client roles*, each of which corresponds to a participant. (However, in many cases the client is uninteresting and only the supplier is really important.). The legal interaction sequences can be specified by a state machine in which the states correspond to activities by the participants and the transition triggers correspond to messages among the participants. (In the case of a simple class library the state machine can be omitted. Even in a more complicated case, frequently it is the participation in an interface that is of most interest, rather than seeing the state machine.) An interface may be drawn as a class with stereotype «`interface`» (Figure 12). The name of the class is the name of the interface. Dependency arrows from participant classes to the interface class are labeled with stereotype «`conforms`» and with the name of the role within the interface (such as “client” or “supplier”). Separate groupings of classes conforming to the interface may be shown on separate class icons; each appearance defines a set of classes that interact, i.e., an instantiation of the interface.

Normally we think of the interface as “belonging” to the supplier, but of course any interaction has two sides and most interfaces impose restrictions on what the clients can ask as well as what the supplier must do. People often think of an interface as a set of operations that the supplier provides, but this is insufficient in many practical cases: there are constraints in the order in which the operations can be called which means that the interface really requires a grammar or a state machine for its full specification.

We provide a stereotype icon for showing interfaces in a more compact form (Figure 13). The supplier of an interface (i.e., the class, package, or entire system conforming to the “supplier” role) shows a protruding solid “lollipop” labeled by the name of the interface. The client of an interface shows a dashed dependency arrow to the interface circle. This notation shows the matchup of suppliers and clients. Either the client or the supplier may also be drawn in isolation with the interface icon to show conformance to the interface.

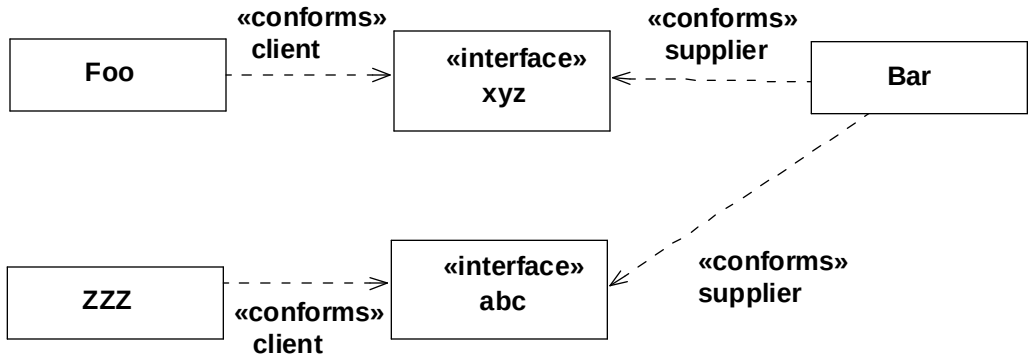


Figure 12: Reified Interface Notation

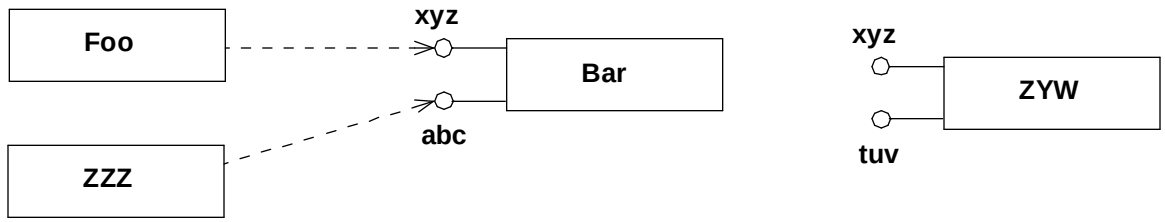


Figure 13: Symbolic Interface Notation

## 3.4 Distribution

In the version 0.8 documentation set, we indicated that our future work would address the problems of distribution and concurrency. In the UML, we have introduced a solution to these problems, largely by using existing UML features in creative way.

The problems of distribution and concurrency are not independent. Briefly, distribution involves at least the following three issues:

- The allocation and possible migration of objects (including processes and threads) relative to nodes in the deployment view
- The grouping of distributed objects; objects are rarely distributed in isolation, but rather, tend to be distributed in groups
- The specification of mechanisms for registering, identifying, moving, and communicating with remote objects

Similarly, concurrency involves at least the following four issues:

- The presence of the processes and threads that compose the system's process architecture
- The allocation of classes and objects to these processes and threads
- The patterns of communication among these processes and threads

- The allocation and possible migration of these processes and threads to memories and processors in the deployment view

The problems of distribution and concurrency bridge the logical and physical views of a system. To be clear, the essential modeling problems we are trying to solve include:

- How do I model the physical distribution and migration of objects in a system?
- How do I model the process architecture of a system?

This section addresses how the UML handles the first question, and section 3.5 addresses the second.

Modeling the physical distribution and migration of objects in the UML requires the introduction of one predefined property (`location`), plus the use of composite objects to denote distribution units. The `location` property denotes the name of the node to which the item is attached. Finally, a distribution unit (a stereotyped object) designates a physical packaging of objects which are distributed and may migrate as a unit across nodes. Since a distribution unit has a location and can be identified, it is a composite object that contains other objects.

The interface stereotype is an important element of modeling distributed systems. In the publish-and-subscribe mechanisms of CORBA ORBS and Microsoft's COM, one typically publishes an interface to which clients can subscribe and against which others implement, even if that implementation is distant from the intended clients. The semantics of interfaces were explained further in section Figure 3.3.

The location property allows one to model the physical partitioning of a system. To be precise, the location of an object manifests itself as a dependency in the UML metamodel, between the object and a node (or nodes). From the user's perspective, this dependency is rendered as a property. Thus, in a class diagram, certain classes might be designated to "live" on a particular node or nodes. For example, in a customer support system built upon a three-tier architecture, we might designate the class `Customer` to be attached to a server node, meaning that all of its instances reside there, while the class `Order Form` might be attached to a client, with an association between `Customer` and `Order Form` that essentially spans the two nodes. Instances of one class can also exist on multiple nodes (in which case the class must be visible on multiple nodes).

In a collaboration diagram, we can then model the migration of a `Customer` object from server to server. In the UML, the same object may be represented at multiple points during its lifetime with distinct object icons connected by the `«becomes»` dependency. Each appearance shows different values for its properties and attributes, such as the location property. Thus, at one point we might see `aCustomer` residing on `firstServer` and then at a later time residing on `secondServer`.

The `«becomes»` dependency shows two stages in the life of a single object (Figure 14). This diagram shows the migration of an object from one node to another as well as an example of dynamic classification in which an object changes its class. The dependency could be applied to the associations as well but this seems unnecessary in most practical cases. A related dependency relationship is `«copies»` which indicates that one object is as a copy of another object, presumably for performance or reliability reasons. The semantics of copying require further work.

The location of an object can also be shown by physical nesting of the icons. An object might be owned by a process that in turn is inside a node. This notation is intuitive but impractical for large

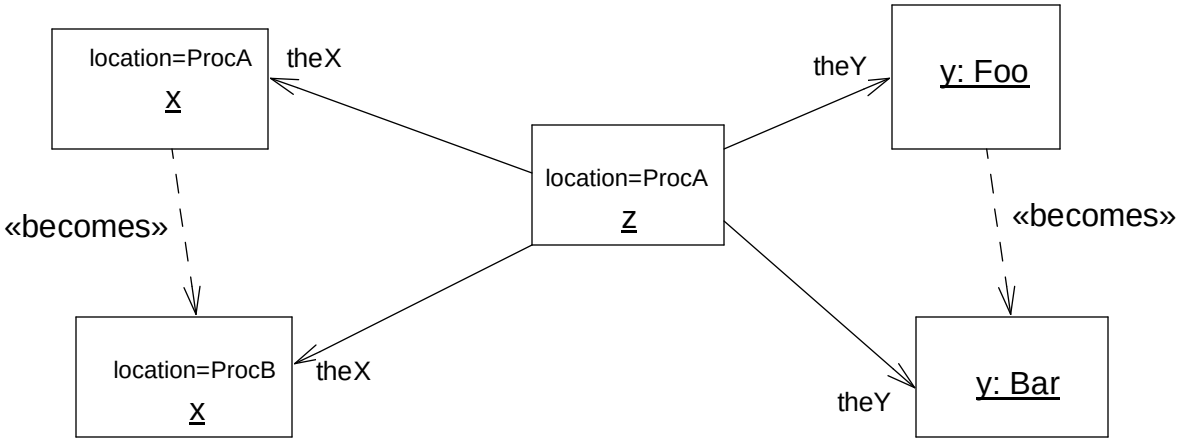


Figure 14: Reified Distribution Notation in a Collaboration Diagram

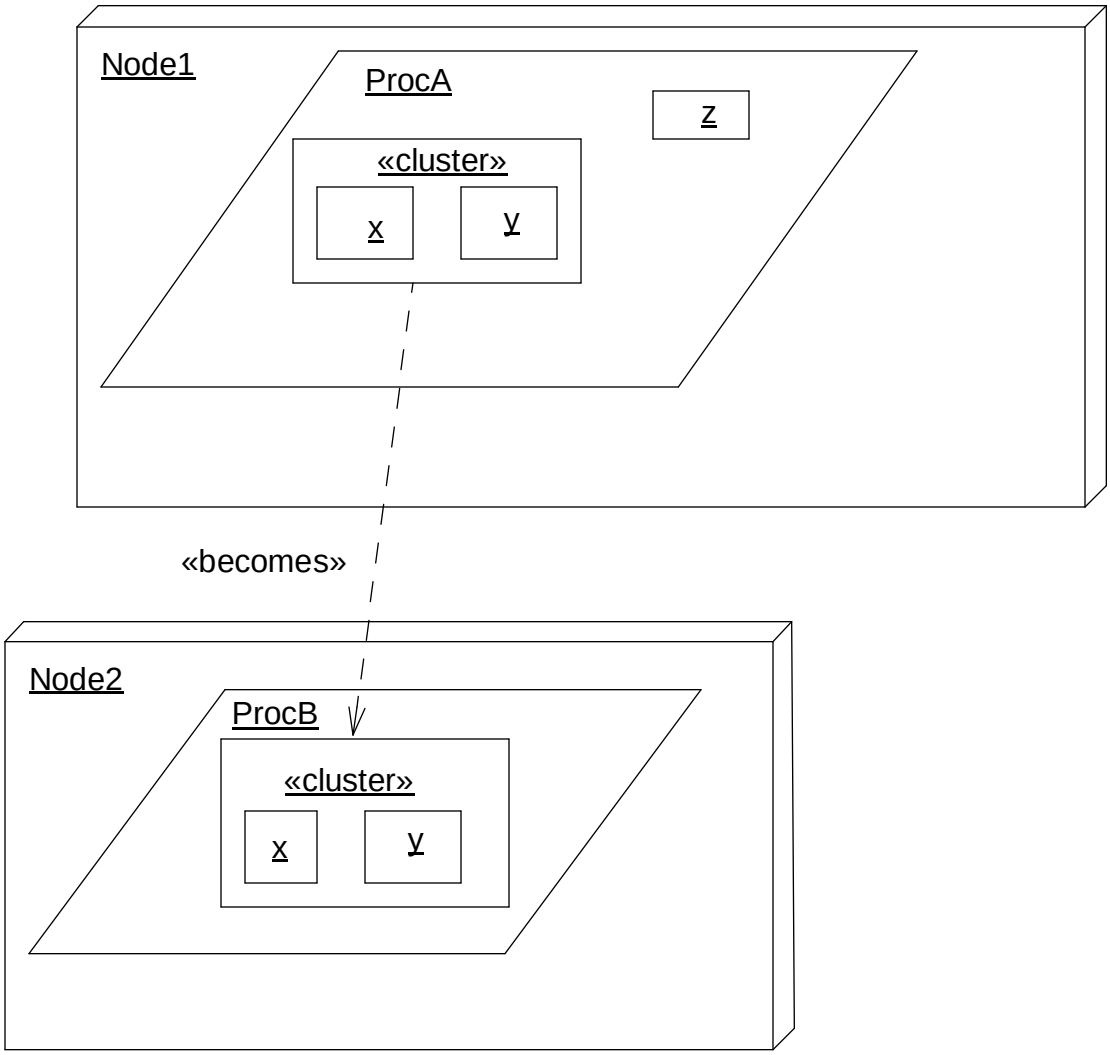


Figure 15: Explicit Distribution Notation

numbers of objects, but might be useful for showing collaboration diagrams that cross physical node boundaries (Figure 15).

Figure 14 shows the distribution notation to show objects that migrate from node to node. Figure 15 shows the migration of an object from one process to another on different nodes, using graphical containment to show the relative locations of nodes, processes, and objects. Figure 16 shows the distribution notation to show static dependencies among code. The understanding is that the code is present on the given nodes (either as an ordinary procedure, as some variety of DLL, or as an active object). Modules may also be marked to indicate whether the caller must be in the same node or can be remote. Classes may be drawn inside the modules that they are defined within, although a separate diagram may be necessary for large modules.

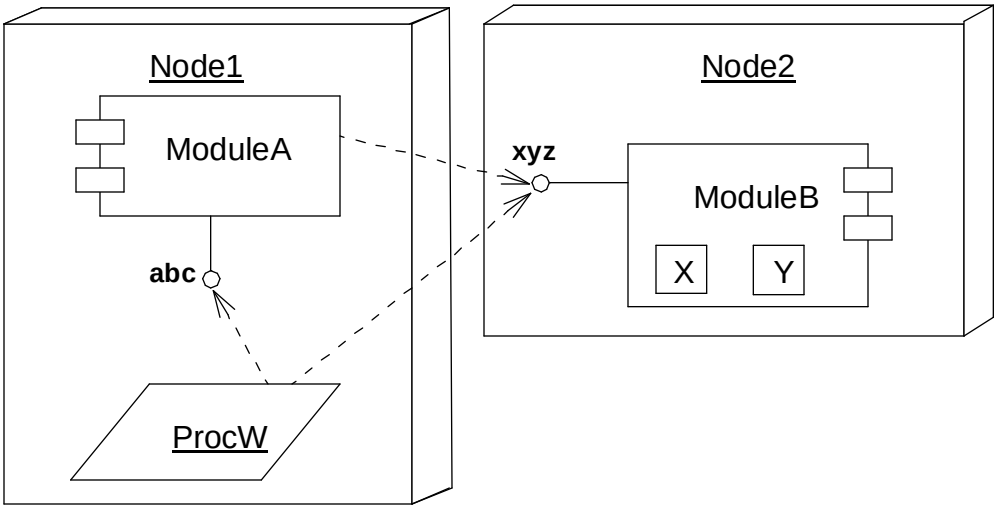


Figure 16: Distribution Notation with Modules

### 3.5 Real Time Semantics

In the version 0.8 documentation set, we introduced several elements that addressed some of the problems of modeling time- and space-critical process architectures. These elements were not clearly identified as such in the 0.8 documentation set, and so we summarize them here:

- Timing marks may be attached as constraints to sequence diagrams and collaboration diagrams in order to model timing (and space) budgets.
- Messages may have different kinds of synchronization semantics. We can represent these with stereotypes on the message; the three main kinds being *call* (send–wait, no process synchronization, no existing activity in the callee, caller blocks until nested activity terminates and returns), *asynchronous* (send–no wait, the caller does not wait for the callee, which has existing activity), and *rendezvous* (send–wait, the caller waits for the existing activity in the callee to reach a designated rendezvous state).

In 0.8, we underspecified the semantics of processes in threads. In the UML, we remedy this issue by introducing the following new features:

- Tasks are first class citizens. A task is the reification of a thread of control, distinct from the objects that it touches.

- A task is rooted in an *active object* (an object that maintains its own thread of control). A task may be implemented within a single process using a stack (the conventional implementation). However, we do not wish to preclude the distributed implementation of a task that crosses address spaces. The important thing is that a task is a sequential flow of control.
- A model may show classes of tasks as well as instance of tasks, and as such may exploit all the existing UML semantics of classes and objects.

A task object is an object with its own memory and thread of control. In class models, a task class may designate a whole set of active objects. In object models, a task object reifies a flow of control and as such sits at the head of a message trace. In class diagrams we render a task class as just a class, but with an appropriate stereotype. In interaction diagrams (including sequence diagrams and collaboration diagrams) we render a task object as an object. In both cases, the stereotypical icon is a rhombus tilted to the right. (The text stereotype can also be used but is less visual.)

### 3.6 Activity Diagrams

Sometimes it is useful to show the work involved in performing an operation by an object. In other words, we want to see the steps that occur in the execution of an implementation of an operation, including the sequential and concurrent sequencing of operations and branches in the flow of control. This is the *method* for implementing an operation (roughly equivalent words include *algorithm* and *procedure*). A method is attached to a particular class, so that several different methods might implement the same operation, depending on the class of the target object. Jim Odell has emphasized the usefulness of displaying method implementation for program design as well as work flow analysis. The implementation of a method can be expressed as a state machine (recall that a Turing machine and a computer program are state machines). An *activity diagram* is isomorphic to a special kind of state machine that describes the implementation of an operation in terms of its suboperations (Figure 17). (Note that UML state machines are much extended from classical state machines.) A Harel state diagram can show the implementation of an operation, but most of the events in the execution of an operation are of the form “completion of the previous operation.” Because the procedural implementation of operations is important, we provide special symbols to simplify the diagrams. These can be regarded as stereotypes of states; some people may prefer to think of them as an altogether different kind of thing. In a general state machine one can think of two kinds of states: an “activity” state represents the execution of an activity (such as an operation) by an object and the state terminates automatically when the activity completes; there is an implicit termination event that triggers the output transition. A “wait” state represents an object that is waiting for some external event to occur, beyond the control of the object owning the state. An activity represents a convenient shorthand for the first situation.

An activity diagram represents the state of the method execution, that is, the state of the object executing the method; the activities in the diagram represent activities of the object that performs the method. Its purpose is to understand the algorithm involved in performing a method. To complete an object-oriented design, the activities within the diagram must be assigned to objects and the control flows assigned to links in the object diagram. To show the assignment of an operation to a class, the syntax *ClassName::OperationName* can be used for the operation name. When these assignments have been made, a collaboration diagram shows the full object-oriented implementation of the method without drawing the individual operations (they are implicit in the message labels). The state of execution of the method is implicit on a collaboration diagram.

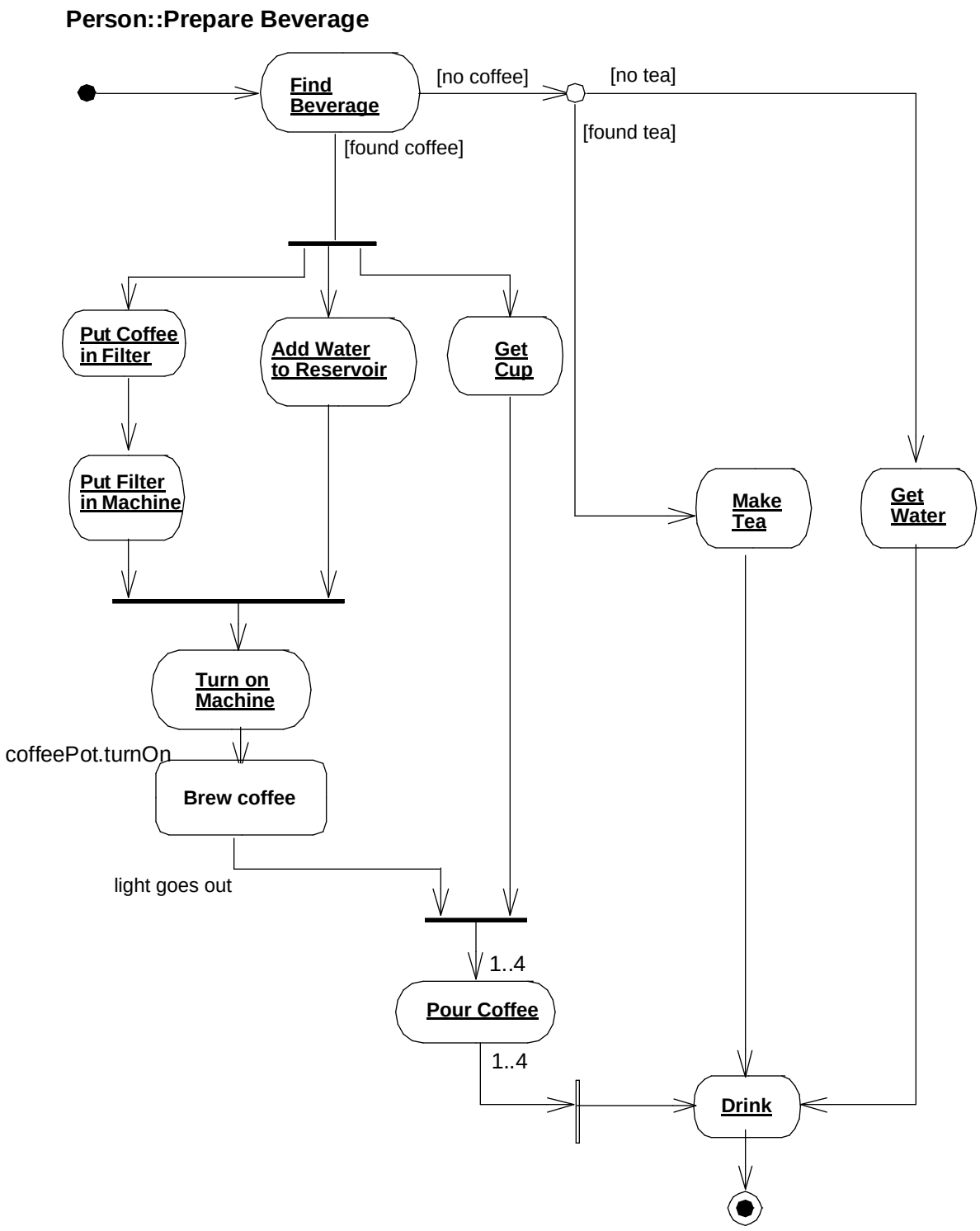


Figure 17: Activity Diagram

From the “outside,” an activity is a state of the method being described, a period of time during which one of the steps is being performed. When expanded, an *activity* is the invocation of an operation and its execution by the same object or another object. An activity is drawn as a “bulging

box" shape with straight top and bottom and rounded ends, containing the name of the operation. When an activity symbol appears within an activity diagram or other state diagram, it indicates the execution of the operation. Executing a particular step within the diagram represents a state within the execution of the overall method. The same operation name may appear more than once in a state diagram, indicating the invocation of the same operation in different places. An outgoing solid arrow attached to an activity symbol indicates a transition triggered by the completion of the activity. The name of this implicit event need not be written, but guard conditions that depend on the result of the activity or other values may be included. Multiple transitions with different guard conditions imply a branch of control; if the conditions are not disjoint then the branch is nondeterministic. A complex condition can be expressed as a "chain" of guard conditions by including a small round circle as a dummy intermediate node. Initiation of concurrent control is expressed by multiple arrows leaving a "synchronization bar" (a short heavy bar with incoming and outgoing arrows, similar to a Petri net transition symbol; this is part of the base state diagram notation). Merging of concurrent control is expressed by multiple arrows entering a synchronization bar. If the transition to the next operation does not occur immediately on completion of an operation, then an explicit event name can be placed on the transition. The execution of one of the activities can be expanded into its own method diagram. All of this notation follows from the basic state diagram notation with the provision for the implicit event on the termination of the activity.

Sometimes a number of copies of an operation are initiated concurrently. If the number of copies is determined at execution time, the individual occurrences of the operation cannot be shown on the diagram, because there is an arbitrary number of them. Instead a multiplicity symbol may be placed on the output arrow from a synchronization symbol. The meaning is that some number of copies of the operation, consistent with the multiplicity value, execute concurrently. Similarly, a multiplicity symbol may be placed on the input arrow to a synchronization symbol; this indicates that the transition occurs when all of the operations have completed. The details of each individual operation should be expanded as a separate diagram. In the example, "pour coffee" is a multiple operation, which are all completed before the single operation "drink".

Activity diagrams are used to show internal activity of an object, but external events may appear in them. An external event appears when the object is in a "wait state—a state during which there is no internal activity by the object and the object is waiting for some external event to occur as the result of an activity by another object (such as a user input or some other signal). There might be more than one possible event that will take the object out of the wait state; the first one that occurs triggers the transition. A wait state is a "normal" state, as found in a Harel state machine. An activity state is just a special case of a normal state, representing the execution of an operation with an implicit event on the termination of the operation execution that triggers the outgoing transition.

It is possible to combine the two symbols to show an internal activity that can be interrupted by an external event. The activity symbol is placed inside a state symbol. The normal procedural control flow arrows are connected to and from the activity symbol; under normal circumstance this path is followed. An arrow from the enclosing state symbol is labeled with the name of an external event; if the external event occurs before the internal activity is completed, then the normal control flow is interrupted and the control passes to the target state of the transition labeled by the external event. Note that this notation follows directly from the rules of the Harel state machines. Note that an activity symbol in a state diagram (including an activityan activity diagram) is a stereotype for a state with the internal activity "do: operationName" and outgoing transitions triggered by the implicit termination event of the operation. However, the activity symbol allows a clear distinction between

a “wait state” and an “internal activity” which is important to procedure design and also allows a clear visual distinction between the “normal” procedural flow of control and the “abnormal” interrupt flow of control, although in reality both are modeled using the exact same underlying model.

Sometimes it is desired to show operations whose execution is external to a particular method but which are necessary to its completion (i.e., the invocation of an external operation). These can be shown as normal wait states. The signal to the external object may be shown using the normal state machine “send” notation on a transition leading to a wait state, which is terminated by the event of receiving a response from the other object. For example, in Figure 17 the activity “turn on machine” results in sending a “turn on” signal to the “coffee pot” object, followed by a state in which the object waits for a response from the coffee pot. When the response is received in the form of the light going out, the exit transition on the “brew coffee” state is triggered and the proceeds to the next internal step of the method. This is all normal state diagram notation. As defined in the state machine notation, the external object and the signals can be shown explicitly or just represented as part of the transition syntax, whichever is more convenient.

Now having incorporated this specialized form of state diagram we could also construct other presentations of state diagrams as stereotypes of the basic states. For example, the ITU standard SDL graph, which is very well known in the field of telecommunication, is an activity diagram with stereotypes for activity states.

To summarize, an activity diagram is not really a new kind of diagram. It is simply a stereotyped state diagram in which all or most of the states are activity states, denoting procedural activity internal to the object owning the state diagram. However, activity states and wait states can be freely mixed in any state diagram.

### 3.7 Patterns and Interactions

As described in section 3.3, an interface describes the external face of a class or a package, consisting of a set of operations and their signatures together with their dynamic semantics. Interfaces express the externally-visible behavior of a single class. However, no class is an island, and so we needed to search for ways to model interactions across a collaboration of classes. This lead us to investigate the modeling of patterns as described by Gamma, et al, in their seminal work, *Design Patterns*. For example, the Gamma pattern `Chain of Responsibility` describes the collaboration between a `Client` object and a chain of `Handler` objects. This pattern effectively describes an interaction among several classes. We call these *interaction patterns*: the pattern includes a class structure as well as a dynamic statement of the legal collaboration. Note that UML interfaces are insufficient for modeling this kind of collaboration, since the pattern involves more than a binary client/supplier relationship.

Patterns such as this are first class modeling elements. We call them “first class” because they are in fact essential elements in the vocabulary of specifying a system’s architecture. We believe we have found a way to reify patterns in the UML without adding significant complexity. An interaction is essentially a design of a use case. A pattern is a template for a set of similar interactions.

For example, in Figure 18, we see the pattern `Chain of Responsibility` expressed as an interaction. We would draw such a diagram to state that we have imposed this particular pattern upon our architecture. If we zoom into the pattern, we would see the realization of the pattern as an interaction among a set of collaborating objects. Zoom out, as we see in this figure, and we can show how classes conform to this pattern, using roles to specify the role that each class plays. Thus,

the classes `KeyboardEventHandler`, `MIDIEventHandler`, and `GeneralEvent-Handler` conform to this pattern as `Handlers`. By *conform*, we mean that the concrete class satisfies the semantics of the class that participates in the pattern.

This approach scales up to not only include generative patterns as found in Gamma's book, but also to domain-specific interaction patterns which can be used within a particular domain, such as banking systems, as well as the ad-hoc design patterns found in every application. There are still some technical problems to be worked out. Patterns in the *Design Patterns* book usually have important instance-level regularities that do not always show up in the class diagram. We need to have a way to show both the class structure and the varieties of instance structure implied by a particular pattern. In this example, each handler class has a successor of a specific other class, but the general pattern permits successors to be of the same or a different class. The interesting aspect of many patterns is their dynamic behavior. We can certainly show this with interaction diagrams, but only in the context of a specific model; it is harder to keep the "patternness" of the pattern open. Finally patterns are templates, in a sense, in which classes, associations, attributes, and operations might all be mapped into different names which keeping the essence of the pattern; we need a way to clearly parameterize the patterns. Nevertheless we feel that we have enough facilities to capture patterns within the UML by expressing them in terms of interactions.

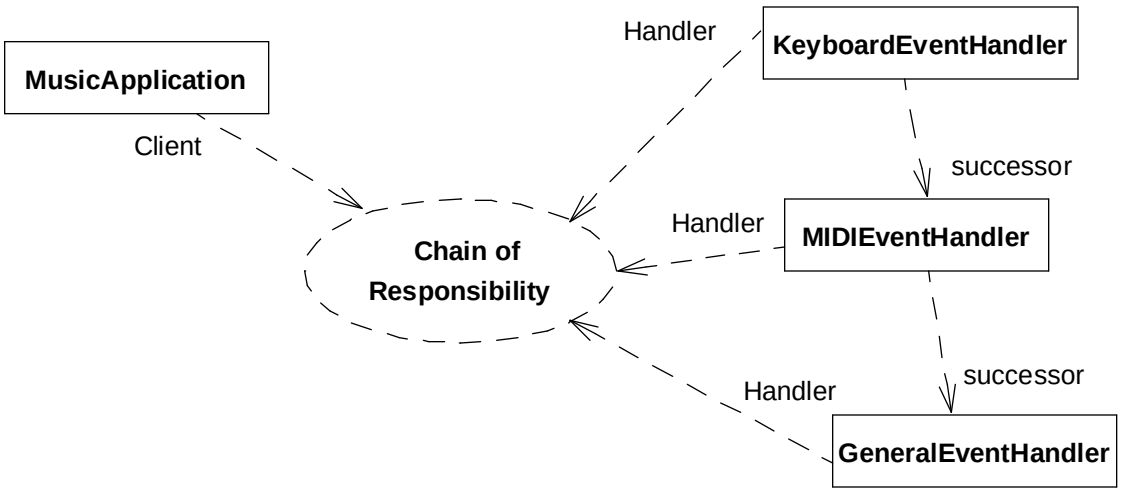


Figure 18: Conformance to a Pattern

## 4. Tentative Proposals

In this section we include some proposals that are not yet final. These proposals deal with important issues but still require further consideration. We ask for reader feedback to help settle these issues.

### 4.1 Traceability

The version 0.8 documentation set introduced the concept of the dependency relationship. This is sufficient for capturing dependencies within a model but was not intended to capture relationships

between elements in two distinct versions of a model or two different models. For such purposes we need to capture the derivation of elements from other elements across models.

For this reason, we have added a general traceability relationship to the UML. Its semantics include:

- Every element may have a trace to one or more elements in other models.
- Not all such traces are interesting; different development styles will designate specific traces as essential.
- A trace is purely structural; when we state that element A traces to element B, we simply mean that there is a connection between the two that may be followed.
- There is no graphical rendering of a trace defined by the UML.
- Tools may attach other semantics to traces; for example destroying element B might be prohibited if there are any traces to it.

As a typical example, we might want to trace a requirement from a use case model to a class in an analysis model and then possibly to a whole collaboration in a design model.

## 5. What's Next

### 5.1 Schedule

First, a little history. In October 1994, Grady and Jim joined forces to begin unifying the Booch and OMT methods. In October 1995, we released the 0.8 documentation set as the first fruits of our work. At the same time Ivar joined us, and we have been working since then to unify the Booch, OMT, and OOSE methods. This 0.91 document represents the results of that collaboration thus far.

Next, our future schedule. After the release of the 0.91 document, our work will focus on these major tasks:

- We will resolve the tentative proposals described in section 4.
- We will resolve the details of the other technical issues as described in section 5.2.
- We will write additional collateral about the UML as described in section 5.3.
- We will complete a formal submission to the Object Management Group (OMG) in response to their request for proposal.

It is this last task that is driving most of our work forward and creating for us a solid deadline for completing all the loose ends. The OMG submission is currently due around the end of 1996/early 1997. We will likely designate that release of the UML documentation set as version 1.0, reflecting the fact that this will be a major, stable, and complete release. Between now and that submission, we will incrementally be delivering some of the collateral as described in section 5.3 We will also be presenting two tutorials on the UML at OOPSLA'96. The first tutorial (to be offered twice) will focus on the basic syntax and semantics of the UML. The second tutorial will focus on the formal semantics of the UML. This first tutorial will be presented by Grady, Jim, and Ivar, and the second will be lead by Gunnar Overgaard, who is helping us develop the UML formal semantics.

Between now and the OMG submission, you will likely see courses and tools that support the UML; we have even encountered some large projects that are starting to use the UML. The UML is reasonably stable enough for this kind of early use, and we are in fact quite happy to see this level of support already. However, do realize that some details remain to be worked out, and these details won't be completely resolved until version 1.0, therefore changes must be expected in preliminary implementations of the UML.

## 5.2 Technical Issues

We have several technical issues to complete. For all of the technical issues, we have working proposals that we are considering but they were not yet ready for release as part of the 0.91 document. These technical issues include:

- The reification of tasks and operations
- The semantics of multiple models
- A number of low-level issues in the metamodel
- A formal specification of the UML semantics

In section 3.5, we discussed the fact that in the UML, tasks will become first class citizens. We understand the basic semantics of tasks, their common stereotypes (processes and threads), their graphical notation, and their connection to other parts of the UML, namely, interaction diagrams and the logical models of classes and objects. However, we have not yet completing integrating these concepts into our metamodel, and in the process we expect to come to a better understanding of task semantics. Hopefully, will find ways to make them even simpler. We also understand the basic interaction between the semantics of tasks and the semantics of distribution, but since this is relatively new ground we are proceeding carefully.

In section 4.1, we discussed the semantics of traceability. We introduced this concept in the UML to begin to address the multi-model problem. While we do understand the basic semantics of traceability, we have further work to do to make this simpler, and to express what kind of traceability relationships are most important.

We have a number of low-level issues in the metamodel left to resolve, virtually none of which will affect the typical UML user; they are primarily important only to tool builders and to the formal specification. Since the 0.8 documentation set, we have been reworking the metamodel, trying to make it self-consistent, precise, and simple. We intend to release a complete metamodel in version 1.0.

Creating a formal specification of the UML is hard work. We are pursuing a formal specification primarily because the very process of creating it forces us to uncover subtle issues that would otherwise have been glossed over. We are being assisted in this work by Gunnar Overgaard. It is our intent to deliver a preliminary formal model of the UML along with the 1.0 submission, and then continue that work to get a reasonable formal coverage of the entire UML. We expect to use a balance of mathematical notation and precise English to write the specification.

With regard to the resolution of all public comments on the UML, we have in fact retained all the comments we have received thus far. While we could not response to each submitter personally because of their sheer volume, will have and will continue to survey them to make certain that we've not let any critical issues drop through the cracks. Ed Eykholt has helped us manage the details of tracking the comments, and thus far, we have identified major trends in the comments sent to us. These trends have in fact impacted how we prioritized our work for this 0.91 document. Prior to the 1.0 submission, we will review all these comments plus the new ones we expect to get after release of the 0.91 document. We very much value this feedback, and have already learned a great deal about users want and like.

### 5.3 Other Collateral

In addition to the stream of documents consisting of the 0.8 release, this 0.91 release, and the forthcoming 1.0 release, there are a number of other pieces of collateral we have already written or are preparing for the UML. This collateral includes:

- White papers published in various public journals
- Web documents concerning the UML
- A series of books on the UML

Grady, Jim, and Ivar have and will continue to publish technical papers describing the UML. A number of these papers have already appeared in two SIGS journals, namely, JOOP (the Journal of Object-Oriented Programming) and ROAD (the Report on Object-oriented Analysis and Design).

Just prior to the release of this 0.91 document, we released a series of Frequently Asked Questions (FAQ) about the UML on our web site (www.rational.com). Over the coming months, we will add to these FAQ as means of addressing other common UML issues.

We are currently working on several books about the UML, including a Reference Manual, a User Guide, and a book on process. Grady, Jim, and Ivar are coauthors on all three of these works. These books will provide a complete and definitive statement of the semantics and use of the UML. After these books are substantially complete, we will likely turn our attention to updating some of our earlier works to the UML.

In addition to the collateral we are preparing, we are aware of a number of other individuals who have or will be writing books that either use or teach the use of the UML. There is already one book on Java that uses the UML, there are several public courses, and we know of several other books that are being written that address different aspects of the UML, all based on our previous preliminary publications. We hope for widespread tool support, training courses, and consultant usage in the future. We strongly encourage this kind of support, and we will work with authors, trainers, and consultants to ensure that their questions are addressed so that there will be wide-spread and consistent support for the UML.