A Critical Look upon UML 1.0

K. Bergner, A. Rausch, M. Sihling

Lehrstuhl IV, Institut für Informatik,
http://www4.informatik.tu-muenchen.de
Technische Universität München,
D-80290 München, Germany

Abstract: The Unified Modeling Language consists of a set of mostly graphical description techniques for the specification, modeling, and documentation of object-oriented systems. Based on the experience gained in using UML 1.0 for the development of a medium-sized, distributed Java program, we comment on its strengths and weaknesses. Furthermore, some proposals for extensions and changes are made.

1 Introduction

The Unified Modeling Language has been proposed by Grady Booch, Ivar Jacobson, and James Rumbaugh as a standard notation for object-oriented analysis and design [BRJ97]. UML version 1.0 incorporates and improves the successful methods OOA/OOD [Boo94], OMT [RBP+91], and OOSE [Jac92] developed by the same authors, and adds some new contributions. This evaluation of UML 1.0 is based on a case study in which we have developed a medium-sized, distributed Java system for planning break supervision schedules in large schools. The application example was provided originally in [RSLML96] for the evaluation of programming paradigms and tools by the DACH group [DAC]. The system can be roughly categorized as a graphical, distributed editor; it offers simple edit functions and requires neither specialized algorithms nor complex concurrency management.

One of the goals of the case study was to examine the individual description techniques as well as their interrelationships and use as a whole in the context of a complete and self-contained example. To do that, we tried to apply each technique as recommended in [BRJ97], showing all its possible application areas. An example are activity diagrams, which were used for modeling business processes during analysis and also for describing the control flow of operations during design.

Most of the comments and proposals in the following sections are directly motivated by concrete problems during the development of the break planner application. We did not try to cover the aspects and possible problems of other application areas systematically. Due to the restricted space, we do not treat the software engineering process, but refer to the detailed case study report for that purpose [BRS97]. As this report includes the complete UML development documents and provides methodical guidelines and hints, it can be used as a tutorial for UML 1.0 and object-oriented development in general.
The rest of this paper mainly consists of sections covering the individual description techniques of UML, assuming that the reader is familiar with their presentation in [BRJ97]. At the end, a short conclusion is given.

2 Use Case Diagrams

Use case diagrams model the users and their interactions with the system at a very high level of abstraction. Hence, use case diagrams are very useful for early requirements analysis because they enforce the identification of the different users and uses of a system and can be easily understood by customers.

In the so-called “semantics definition” of [BRJ97], it is “the responsibility” of a use case to “specify a set of use case instances, where a use case instance represents a sequence of actions a system performs that yields an observable result of value to a particular actor”.

Seen as an isolated description technique, use case diagrams are not a very powerful formalism. They contain hardly any information about the functionality of a system, but are useful mainly as a structuring aid for other kinds of diagrams that describe the action sequences of a system in more detail, like sequence diagrams, collaboration diagrams, and activity diagrams.

In the meta-model provided by the UML authors the concept of a use case is derived from the “Type” concept. A somewhat strange effect of this derivation is that use cases inherit some properties that we cannot make sense of. As expressed in Section 7 of the semantics document [BRJ97], they have attributes and operations, but it is not explained what these attributes and operations should be. In our eyes their presence conflicts with the semantical definition of a use case as a set of action sequences.

Another unclear point concerns the semantics of the <uses> and <extends> relations. Apart from informal descriptions where both relations are equally described as a sort of “inclusion” or “extension”, the UML 1.0 semantic definition documents provide no useful information on this issue.

A possible interpretation of A <uses> B is given by the following conditions, according to Coleman [Col97]:

- A incorporates B as a sub-flow of events. It must be specified where B is inserted.
- The details of use case B are hidden from A.
- B is a fully fledged use case and may involve some or all of A’s connections.

With respect to the corresponding sequences, this can be interpreted as: One or more sequences of A’s sequence set contain sequences of B’s sequence set as contiguous subsequences at certain locations in time. Note that although this situation resembles a procedure call in programming languages,
one cannot assume the presence of a “runtime connection” or a procedure call between two use cases because use cases are only conceptual modeling constructs usually not directly implemented in a system.

In contrast to this, the situation $B \text{ <extends> } A$ can be defined by other conditions, again following [Col97]:

- $B$ is a variation of $A$. It contains additional events (e.g. for a failure or to deal with an extra complexity) for certain conditions.
- It has to be specified where and how $B$ enhances $A$.
- $B$ is not a fully fledged use case.

With respect to the corresponding sequences, this can be interpreted as: $B$ contains all action sequences of $A$ and furthermore adds own sequences that contain sequences of $A$ as (possibly non-contiguous) subsequences. Other questions left open by [BRJ97] are:

- Is it possible to have a use case without a connection to an actor?
  Such a use case would in some way contradict with the purpose of a use case as a modeling concept for the usage of a system by external actors, but could be handy for modeling internal system tasks that can be handled automatically and do not need external interaction.

- If a use case $A$ is extended by a use case $B$, does $B$ have to be connected with all actors of $A$?
  We think the answer is negative, considering the example of a hypothetical use case Edit Data extended by Edit Confidential Data, which contains additional functionality for authorization. In this example, there may well be (classes of) actors associated exclusively with only one of these use cases.

- If a use case $A$ is extended by a use case $B$, must $B$ have <uses>-connections to the same use cases as $A$?
  According to the sequence interpretation of use cases given above, the answer must be yes. However, it is questionable whether such obligatory and, therefore, redundant connections should be represented in a use case diagram. Our recommendation is to draw them only if the extended use case introduces own functionality that <uses> the concerning use case, too, and to leave them out otherwise.

UML 1.0 does not make a proposal for the structure and the contents of a textual use case dictionary. Besides informal textual descriptions, we added items for the estimated number, the experience level, and the location of each actor group, and the frequency and the required security level of each use case, respectively. Later on, information like this can be used for various purposes, for example, to design an adequate GUI and a help system, or to estimate the performance of a design.
To strengthen the connection to class diagrams, we also included entries listing the classes concerned by a use case. By comparing the actions of a use case’s action sequences with the attributes and operations of the assigned classes, one can check whether the classes contain all functionality needed (and not more than needed) and get hints about what data is shared among which actors. A better way to visualize the correlation between classes and use cases could be provided by a tool that highlights the classes belonging to a certain use case in the class diagram.

In total, we think that UML use cases are a valuable tool for requirements analysis. A clear definition of their syntax and semantics seems to be possible, but is missing in [BRJ97].

3 Class and Instance Diagrams

UML’s so-called static structure diagrams mainly model the data aspect of an object-oriented system, but contain also information about the operations of the data items. Static structure diagrams exist in two variants: Class diagrams show the classes of the program code, their attributes and operations, and the relationships and dependencies between them. Object diagrams or instance diagrams show graphs of object instances that may arise during runtime of a system. Class diagrams may be seen as a special kind of E/R-diagrams [Che76] and are very common in object-oriented development methods [SM88, RBP+91, Boo94, CAB+94]. They are used for data modeling in the early development phases and are later refined and enriched with additional attributes and operations. Finally, they can be translated into class skeletons.

Semantics of Associations Although class diagrams are a well-known formalism in many object-oriented development methods, their semantics is not totally clear with respect to associations. Some of the possible implementations for associations are:

- Reference attributes or—in the case of 1-to-n or m-to-n-associations—containers with references can be embedded into the associated class. If the association must be bidirectional, both concerned classes have to be adapted, and the consistency of the two directions has to be ensured.

- Associations can also be implemented by a dedicated association class containing references to the associated objects. The main advantage of this implementation scheme is the possibility to store the association instances between all objects in a container and to enumerate them quickly. Furthermore, one can add additional attributes easily.

- In Java, associations to a so-called “singleton class” with exactly one instance can be implemented in a special way by an “implicit class
reference”. To allow for associations to the singleton class $S$, for example, $S$ can be given a static variable $S\text{.theInstance}$ that is instantiated once during the initialization of class $S$. This would make it possible to access this variable’s object with the Java idiom $S\text{.theInstance}$ in the code of other classes.

However, one could principally also think of implementing an association via a network socket or via explicit object names. An unambiguous, abstract characterization of the semantics of associations together with some common implementation schemes could help to automate the generation of association implementations by means of tools.

**Boundary Concept**  Class diagrams as used in UML lack the concept of a system scope, making it hard to distinguish entities that must be implemented from entities in the environment of the system. This is in general true for all description techniques of UML except for use case diagrams where this boundary is represented by the rectangle enclosing the use case ellipses. The addition of a system boundary concept, as, for example, in Fusion [CAB+94], would fill this gap.

**Instance Diagrams**  In general, class diagrams do not constrain the possible object graphs that may occur during the runtime of a program. For the description of object graphs, UML additionally offers instance diagrams. They show a snapshot of the runtime objects and their interconnections in a certain situation.

Instance diagrams are not suitable for the description of large object configurations, as they do not scale well and get too large and too complicated even for small examples. Because they can only describe snapshots, they can also not be used to specify dynamic object graphs with changing associations and objects.

In order to deal with complex, dynamic object graphs, more powerful description and specification formalisms than instance diagrams are needed, like, for example, component diagrams as introduced in [Ber96].

**Refinement**  During the development of a system, one usually constructs a series of class diagrams from the analysis class diagram, until a detailed, refined implementation class diagram is reached. Although UML contains notations to represent refinement steps, we could not use them in our case study because the proposed notation for “refinement within a given model” would have led to a huge, incomprehensible model containing all the classes of the different development phases together with their refinement relations (see [BRJ97], Notation Guide, Section 4.26, and left side of Figure 1). In our opinion, this variant should be restricted to the special case in which one wants to demonstrate the refinement of a single entity explicitly.
The notation for “refinement between models” is based on “an invisible hyperlink supported by a dynamic tool” and is, therefore, not a suitable notation for a paper-based presentation. We think that a simple and general notation for the representation of refinement relations between models should be introduced that is usable also for a paper-based presentation. We propose a special symbol for this purpose, reminiscent of a back-reference as well as of a generalization arrowhead (see right side of Figure 1).

![Diagram](Image)

Figure 1: Representation of Refinement Within and Between Models

The semantics of refinement relations is left entirely open in UML—there are no rules clarifying which refinement steps exist and when they can be applied. A more formal treatment would allow the implementation of tools that support or maybe even automate the execution and validation of refinement steps. Note that although we have restricted our discussion to class diagrams, it applies also to other kinds of diagrams, like, for example, sequence diagrams.

**Responsibilities** UML lacks a notational construct for grouping operations and attributes of classes together to so-called “responsibilities”, a concept introduced by Wirfs-Brock [WBWW90].

We propose to use a simple, tree-like notation, where responsibilities are represented in bold font above the indented names of their contained elements (see Figure 2). A tool could then be used for switching between folded and unfolded views.

Even in the context of relatively small class diagrams like the ones of the break planner, responsibilities would allow to comprehend the structure of classes with many features much faster. The use of equally named responsibilities in different classes could furthermore help in understanding mechanisms implemented by collaborating classes. An example would be the use of MVC responsibilities, grouping together the collaborating operations in various model, view and controller classes.

Responsibilities can first be introduced during requirements analysis to specify the data and functionality of a class informally, leaving open the final names and signatures of its operations and attributes. During design and implementation, responsibilities can then be refined stepwise by adding attributes, operations, and also other responsibilities. Specialized documentation tools like javadoc [SUN97] could also use responsibilities to present class features in a structured way.

Another possibility is the use of standardized responsibility schemes that could be unfolded automatically with the help of a tool. This would for example be useful for the handling of associations or data attributes, where the same patterns of attributes and access operations appear over and over
Figure 2: Grouping of Class Features with Responsibilities

again in a detailed class diagram. An example for a responsibility schema is a scalar attribute, like job in Figure 2. It is usually implemented by the attribute itself and two access methods getJob and setJob for reading/writing the attribute's value. Another example are managed 1-to-n associations that are usually implemented by a special pattern of operations (see also the paragraph above on associations).

Interfaces According to UML, an interface should be represented as a little, named circle next to the class implementing the interface. The advantage of this notation is that it needs less diagram space and fewer lines in comparison to a class-like representation. The disadvantage is that one cannot see the operations of an interface in the class diagram. There is also no notation for representing subtype relations between interfaces.

The notation should therefore only be used for standard interfaces with known functionality that are not subtyped. A good example in the context of Java is the standard interface Serializable.

If the operations of an interface are important or an interface is subtyped, we recommend to use a class-like notation where interfaces are marked with the special stereotype «Interface».

4 Sequence Diagrams

Sequence diagrams, a variant of the well-known message sequence charts [IT93, LRH97] and extended event traces [SH96, BHKS97], show example
communication histories between users or objects. The UML variant is extended with constructs for the creation and deletion of objects as well as for synchronous and asynchronous communication.

It is out of question that sequence diagrams are a useful description technique. In our experience, they are—together with class diagrams—one of the predominant description techniques in design meetings. Furthermore, sequence diagrams can be given a precise semantic, as, for example, shown in [BHKS97].

UML enhances sequence diagrams with a notation for modeling the message flow between “entire sets of objects” instead only between single objects. It is, however, not clear what the semantics of this construct is—is a message related to all or only some of the objects in the set? If the latter is true, how is the subset specified? If there is an arrow from a set of objects to a single object, how many messages does the single object receive?

Sequence diagrams can be used as test cases for an existing implementation of a system. For this purpose, additional information like preconditions, input data, and test instructions should be provided for sequence diagrams, and there should be methodical guidelines on the usage of sequence diagrams for testing.

In addition to “exemplary” sequence diagrams, the UML variant contains features like conditional subsequences that make sequence diagrams useful also for the specification of behavior. However, there exists no notation to distinguish sequence diagrams meant as comprehensive specifications from sequence diagrams showing just exemplary interactions. It is also not clear for which object configurations a sequence diagram specification is valid—does the sequence diagram imply that only certain configurations appear during the runtime of a system or does it apply only in certain situations?

5 Collaboration Diagrams

Collaboration diagrams resemble sequence diagrams in most respects. However, they emphasize the relationships between objects and show the flow of time only implicitly using sequence numbers. An automatic translation between sequence diagrams and collaboration diagrams is, therefore, possible with one exception: Connections not used for communication can only be represented in collaboration diagrams. Apart from this rather unimportant issue, it seems just a matter of personal style which of both techniques one wants to use.

The authors of [BRJ97] also propose the usage of “before-after conditions” for declarative specifications of the behavior of a type’s instances ([BRJ97], Section 7.3). However, it is left unclear how this formalism is related to the other description techniques for the specification of a system’s dynamics, what formalisms are admissible in before-after conditions, and how the “context” of a type should be defined.

Another collaboration diagram notation is used for design patterns which
would otherwise not be visible in a class diagram. This notation has proved valuable to describe the presence of the observer pattern in our case study [BRS97]. However, it is doubtful whether more complex design patterns can be integrated into a class diagram at all—the different components of the microkernel pattern in [BMR+96] represent, for example, rather subsystems than classes, so that mapping them to simple classes makes no sense.

6 Class State Diagrams

Class state diagrams can be used to model the data state and its changes during the lifecycle of the objects of a certain, single class. The data state of an object consists of the actual attribute values of the object, its references to other objects, and possibly also the data states of referenced objects. A special notation is provided for state transitions that trigger the sending of messages to other objects.

State transition diagrams are a universal and well-known formalism for specifying the state space and the state transition relation of entities. However, most questions about their methodical use and about their semantics in the context of object-oriented modeling are left open in the UML specification. Consistency criteria and methodical guidelines for the simultaneous use and the transition between activity diagrams, state diagrams, sequence diagrams, and collaboration diagrams are urgently needed.

A similar problem concerns the consistency between different, but related documents of a single dynamic description technique. Classes related by inheritance should inherit not only attributes and operation signatures, but also dynamic behavior as specified by the class state diagrams of their base classes [Rum96], and refined versions of classes should have a suitably refined behavior. Also needed is the possibility to assign a state diagram to a compound component and to break it down into subordinate state diagrams.

7 Activity Diagrams

Activity diagrams are a special kind of state transition diagrams used to specify control state. They can be used on different abstraction levels for business process modeling of user interactions as well as for modeling the control flow of single operations.

Although the use of activity diagrams for business process modeling during the analysis phase seems like a simple and natural concept, their implementation and translation to source code is not trivial. In principle, the following possibilities exist:

**Explicit Control:** Activity diagrams serve as an explicit, operational specification for a specialized workflow engine controlling the functionality of components of a system. Using a workflow engine seems to become a
common architecture for the integration of legacy components (which may be also whole programs) into a larger system.

**Implicit Control:** Activity diagrams serve as a specification for the interaction of components. This approach is common with user interfaces, where the control flow will likely not be implemented by a centralized workflow component, but will be integrated in the callbacks and operations of the GUI classes. When used to specify the interactions of GUI elements, activity diagrams resemble the so-called “interaction diagrams” of Denert [Den91].

To facilitate the translation to source code or to even allow an automated implementation via tools, the semantics of activity diagrams must be clearly defined. This would also help the user to understand the connection between action states and actual programs or GUI prototypes.

Another critical point is the lack of methodical guidance for the transition from exemplary sequence diagrams to prescriptive activity diagrams. In general, it is not trivial to identify the common characteristics of a set of possible sequences and to build suitable activity diagrams that allow all these sequences. This is particularly difficult because a single action sequence can in principle be the result of the interleaved execution of more than one use case.

UML states that an activity diagram is “a special form of a state diagram”. As such, it should use the syntactical constructs introduced in the section on state diagrams. However, that seems not to be the case with so-called “complex transitions”, especially when the “swimlane” notation is used.

First, there is a minor inconsistency in Figure 50 of [BRJ97]: The surrounding complex transitions are missing and concurrency is represented there by short heavy horizontal bars instead of short heavy vertical bars, as specified in section 8.6.2 of [BRJ97].

Second, the diagram in Figure 53 of [BRJ97] contains action states like Pay in swimline Customer and Take order in swimline Sales that are meant to run concurrently, but it does not have any surrounding complex transition. The correct representation should have additional complex transitions after Request service and before Deliver order.

Although activity diagrams are intended to specify the dynamic behavior of a system, they cannot deal with changing object graphs properly. This can be seen from diagram 3: It contains two swimlanes, namely Break and Teacher. While the action Break::hasConflict belongs to a special, single object, the action Break::overlaps is executed for all instances of Break that are connected to its supervising Teacher object. However, there is no notation to discern actions concerning a single object from actions concerning a whole set of objects in the same swimlane.

During design, activity diagrams can be used to specify the control flow of methods. When used for this purpose, they are very similar to well-known techniques like Nassi-Shneidermann charts or flowcharts. However,
most operations in object-oriented programs are very simple and creating an activity diagram for them would not introduce additional clarity.

8 Implementation Diagrams

Implementation diagrams exist in two variants. Component diagrams show the structure of the source code and its partitioning into components, and deployment diagrams show the run-time implementation structure and the distribution of objects and components on physical computing nodes. However, UML seems to neglect the difference between component and deployment diagrams and allows combined diagrams containing both aspects. This is shown in Figure 4 which contains Figures 56 and 57 of [BRJ97]: The component diagram on the right contains the program component Scheduler that accesses the component instance meetingsDB. In a semantically correct diagram, the Scheduler component should be marked as instance with underlined text, similar to the component instance meetingsDB.
The notation provided seems also overly simple. It does, for example, not contain concepts for inheritance and recursive containment of component instances, and it allows only the representation of static configurations consisting of a fixed number of hard-wired component instances. To overcome these limitations, more powerful formalisms like the one proposed in [Ber96] are needed.

Finally, UML says nothing about the relation of implementation diagrams with the other description techniques. This would be necessary especially for instance diagrams, sequence diagrams, and collaboration diagrams because these diagrams also refer to object and component runtime configurations.

9 Conclusion

All in all our experiences with UML in the case study were not totally negative—we could overcome all problems and were able to model most aspects of the break planner system. However, this does not imply that UML is a mature modeling language that can be used for real projects without problems. Our main criticisms are:

- UML provides a wealth of description techniques, but defines neither their syntax nor their semantics precisely and unambiguously [BHH+97]. This makes modeling sometimes deceptively easy because one is allowed to draw all kinds of diagrams that have no useful meaning for the subsequent implementation.

- The missing semantic foundation is also problematic with respect to the relationships between the various development documents, especially when it comes to describing the dynamic behavior of a system. Because there exist no consistency criteria between description techniques it is hard to check whether all of their requirements can be
combined and fulfilled by the implementation. Consistency criteria and methodical guidelines could also make the production of development documents easier because they restrict the possibilities of the developers and force them to consider only meaningful diagrams.

- The same considerations apply to UML’s concept of refinement—it is not defined when a development document is a refined version of another document and what development steps are admissible for refining documents.

- UML’s description techniques cannot deal sufficiently with complex, changing object graphs and hierarchical composite objects. The existing notations for so-called “multi-objects” in sequence and collaboration diagrams seem very ad-hoc and leave many questions open, as well as the whole description technique of component diagrams.

- UML lacks abstraction techniques for large class diagrams with many attributes and operations.

- Some of the techniques of UML aim at the implementation of a CASE tool and cannot be presented on paper. An example are most relationships between development documents which shall be represented by “invisible hyperlinks” according to UML’s definition.

- The UML Notation Guide and the UML Semantics Document [BRJ97] do not contain sufficient examples and are not very readable, if not to say confusing. This is especially true for the Semantics Document: Large parts of it seem to be machine-generated English, the index is nearly unusable because it contains too many references for each entry, and the definitions contained are informal and unclear.

References


[DAIJ97] DACH Group. Universität Hamburg, FB Informatik, AB Softwaretechnik; Johannes-Kepler-Universität, Linz, Austria, Institut für Wirtschaftsinformatik, Doppler-Labor für Software Engineering; GMD Bonn, Schloß Birlinghoven, St. Augustin; UBS Information Technology Laboratory, Zürich, Switzerland.


