4. UML Class Diagrams as a Formal Specification Language

4.1 Object-Oriented Specification with the Unified Modeling Language (UML)

4.1.1 Motivation

The algebraic specification formalism, which was studied in the preceding chapters, has several drawbacks regarding its practical applicability:

- The relationship to real programs is rather indirect. In particular, the restriction to completely state-less functions and data types opens a relatively large gap to concepts of modern programming languages, which are usually imperative and make extensive use of references.

- The relationship to real life in the early stages of system development is too weak. When applying algebraic specifications, people are expected to change completely the way they specify systems. It is rather difficult even for an expert to determine whether an algebraic specification actually mirrors the requirements for the system under discussion. A better integration with those semi-formal notations would be helpful which are in wide use already.

Most modern programming languages are object-oriented. For analysis and design of software, the object-oriented approach is clearly dominant nowadays. Therefore, we aim in the following at an approach that combines object-oriented concepts, in particular from analysis and design languages, with the semantic ideas of algebraic specifications. The result will be a language which is as precise as algebraic specifications (although more complex), but which allows a smooth and almost not recognisable transition from semi-formal to purely formal concepts. The material in this chapter consists of current research results, so some open points will be unavoidable.

4.1.2 Unified Modeling Language

It is very fortunate for the discipline of software engineering that in the last few years a standard language for semi-formal description of problems and systems has been defined, the Unified Modeling Language (UML). There are many graphical editing tools available for this language. So it is logical to concentrate on this language.
For the remainder of this chapter, some basic knowledge on object-oriented concepts is helpful. In the lecture, some more background information will be given. Moreover, there is a wealth of literature on the subject.  

**4.1.3 UML Class Diagrams**

The following figure is a simple example of a UML class diagram that contains some of the core features of UML: classes, attributes, operations, associations, aggregation, inheritance.

There is a relatively close relationship from an informal description of a problem to such a class diagram. One well-known technique to find classes is just to look for nouns in a text describing the problem to be solved. In a second step, we filter out those nouns that describe key concepts, where there exist several instances of the information described by the noun and where the instances have individual properties and/or responsibilities. In this example, we are dealing with teams and the arrangement of meetings for the team. The key concepts of TeamMember, Team and Meeting are very obvious. The concept of a Person is also an obvious generalisation of the concept of a TeamMember.

---

The classes are represented in a UML class diagram by rectangles with three compartments: class name, attributes, operations.

The attributes of a class define the individual properties of the instances of the class (the objects). Each instance (object) has its own value for such a property. The informal meaning of the attributes shown in this diagram is as follows:

- **Class Person, attribute name**: The name of the person (a string).
- **Class Person, attribute age**: The age of the person (a number).
- **Class TeamMember, attribute role**: The role of a team member in its team, like team leader, ordinary member, coordinator. Given as a string to achieve high flexibility in defining roles.
- **Class Meeting, attribute title**: The name of the meeting (a string).
- **Class Meeting, attribute day**: The day when the meeting takes place (a natural number, to avoid the complexities of real dates).
- **Class Meeting, attribute start**: The hour when the meeting starts (a natural number).
- **Class Meeting, attribute end**: The hour when the meeting ends (a natural number).
- **Class Meeting, attribute isConfirmed**: The status information whether a meeting is confirmed with its participants (i.e. there are no conflicts with other meetings of the participants).

Please note that the class diagram shows the more general type integer for attributes, which are natural numbers (or in fact even further restricted). This is just for compatibility with standard type systems of programming languages. Below we will discuss techniques to make precise any constraints on values.

The operations describe the responsibilities of objects of the class. An operation provides a functionality that is executed when the object receives a message calling the operation. Operations may have result values or not and may change the local state (e.g. attribute values).

The informal meaning of the operations shown in this diagram is as follows:

- **Class TeamMember, operation numMeetings()**: The current number of meetings a team member is concerned with.
• Class \textit{TeamMember}, operation \textit{numConfMeetings}(): The current number of confirmed meetings a team member is concerned with. A meeting is confirmed if the attribute \textit{isConfirmed} has the value \textit{true}.

• Class \textit{Meeting}, operation \textit{confirm}(): This operation compares the date of the meeting on which it is executed with all other confirmed meetings in which the participants of the current meeting are involved. If there is no time conflict, the status of the meeting changes to confirmed (i.e. \textit{isConfirmed} becomes true).

• Class \textit{Meeting}, operation \textit{cancel}(): This operation cancels a meeting.

In practice, varieties of interpretations for class diagrams appear:

• Some diagrams are meant as definition of a database schema or a similar purely static data structure. There is no additional hidden structure of the objects beyond the attributes; often there are only few or rather trivial operations. Implementations can be generated automatically from a UML diagram in some cases.

• Some class boxes in diagrams are meant as a definition of complex structures that are implemented manually, introducing internal attributes or data structures. Typically, such classes have many rather complex operations and only few, sometimes even zero attributes (see the \textit{Vector} example).

• Some diagrams show a mixture of both styles, like the example given above.

An \textit{association} expresses the fact that objects of two involved classes are linked, i.e. some objects of one class "know about" some objects of the other class. In the example, there is just one association \textit{participates} which is read from \textit{TeamMember} to \textit{Meeting} (as indicated by the little black triangle). It expresses which team member participates in which meeting.

A special class of an association is the \textit{aggregation}, indicated by a hollow rhombus at the association line. An aggregation denotes a whole-part relationship. Here, the only example is the aggregation between a \textit{Team} (the whole) and a \textit{TeamMember} (its parts).

The notion of \textit{inheritance} (arrow with hollow arrowhead) is used to denote that some class is a special case of another class, like between \textit{TeamMember} and \textit{Person}. Each object of class \textit{TeamMember} has all attributes, operations and associations of its superclass \textit{Person}.
UML class diagrams have a direct correspondence to object-oriented programs. So for instance, the class TeamMember can be translated easily into the following skeleton of a Java class declaration:

```java
class TeamMember {
    String role;

    int numMeetings() {...}
    int numConfMeetings() {...}
}
```

These few remarks already make clear that UML class diagrams are much better related to informal specifications and real programs than traditional formal specifications. Of course, they are usually considered to be non-formal or at most semi-formal. In the following sections, we will discuss how UML class diagrams can be "taken seriously" as a powerful formal specification language.