

Chapter 2

A Short Tour of UML

Before introducing the most important concepts of UML in the following chapters, we first explain the background of this modeling language. We look at how UML came into being and what the “U” for “Unified” actually means. We then answer the question of how UML itself is defined, that is, where do the rules come from that dictate what a valid model should look like in UML? Furthermore, we outline what UML is used for. Finally, we give a short overview of all 14 UML diagrams in the current version 2.4.1 of the UML standard specification. These diagrams can be used for modeling both structure and behavior.

2.1 The History of UML

The introduction of object-oriented concepts in information technology originates from the work of the early 1960s [12]. The first ideas were implemented in systems such as Sketchpad, which offered a new, graphical communication approach between man and computer [28, 51].

Origins of object orientation

Today, the programming language SIMULA [24] is regarded as the first object-oriented programming language. SIMULA was primarily used to develop simulation software and was not particularly widely used. It already included concepts such as classes, objects, inheritance, and dynamic binding [2].

SIMULA

The introduction of these concepts was the start of a revolution in software development. In the subsequent decades, there followed a multitude of programming languages based on the object-oriented paradigm [21]. These included languages such as C++ [50], Eiffel [31], and Smalltalk [28]. They already contained many of the important concepts of modern programming languages and are still used today.

Object-oriented programming languages

The emergence and introduction of object orientation as a method in software engineering is closely connected to the appearance of object-oriented programming languages. Today, object orientation is a proven and well-established approach for dealing with the complexity of software systems. It is applied not only in programming languages but also in other areas, such as in databases or the description of user interfaces.

As we have already discussed in Section 1.2, where we introduced the notion of a model, software systems are abstractions aimed at solving problems of the real world with the support of machines. Procedural programming languages are not necessarily the most appropriate tools for describing the real world: the differences in concept between a natural description of a problem and the practical implementation as a program are huge. Object-oriented programming was an attempt to develop better programs that, above all, are easier to maintain [12].

Over the years, object orientation has become the most important software development paradigm. This is reflected in object-oriented programming languages such as Java [4] or C# [32] and object-oriented modeling languages such as UML. However, the road to the current state-of-the-art of software development was long and winding.

Ada

In the 1980s, the programming language Ada, funded by the United States Department of Defense, was extremely popular due to its powerful concepts and efficient compilers [25]. Even back then, Ada supported abstract data types in the form of *packages* and concurrency in the form of *tasks*. Packages allowed the separation of specification and implementation and the usage of objects and classes of objects. Ada thus distinguished itself fundamentally from other popular languages of that time, such as Fortran and Cobol. As a consequence, there followed a great demand for object-oriented analysis and design methods to make the development of Ada programs easier. Due to the wide distribution of Ada and the pressure from the United States Department of Defense, these modeling methods were based specifically on the characteristics of Ada. Grady Booch was one of the first researchers to publish work on the object-oriented design of Ada programs [5].

Booch method

Over time, a number of further object-oriented modeling methods arose (see [12] for an overview). In general, the modeling methods had either a strong reference to programming languages, such as the Booch method, or a strong reference to data modeling, such as the *Object Modeling Technique* (OMT) approach developed by James Rumbaugh et al. [42]. OMT supported the development of complex objects in the sense of an object-oriented extension of the entity-relationship model [14] which had been introduced for describing databases.

*OMT approach by
Rumbaugh et al.*

*OOSE approach by
Jacobson et al.*

Independently of this, Ivar Jacobson et al. introduced the *Object-Oriented Software Engineering* (OOSE) approach [27]. This approach was originally developed to describe telecommunication systems.

In the 1980s and early 1990s, the modeling world was flooded with a multitude of different modeling languages. Considerable effort was required to deal with the resulting compatibility problems. The models of different project partners were often not compatible and it was not always possible to reuse models in different projects. The result was exhausting discussions about different notations, which detracted from the actual modeling problems. As new modeling languages were appearing all the time, there was no clarity about which were worthy of investment and which were just a short-lived trend. If a language did not become accepted, all investments that had been made to establish it within a project or a company were generally lost. Looking back, this time of numerous approaches, often with the difference being only in the detail, is referred to as the *method war*.

Method war

To put an end to this unsatisfactory situation, in 1996 the *Object Management Group* (OMG) [33], the most important standardization body for object-oriented software development, called for the specification of a uniform modeling standard.

Object Management Group (OMG)

In the previous year, 1995, Grady Booch, Ivar Jacobson, and James Rumbaugh had combined their ideas and approaches at the OOPSLA conference (OOPSLA stands for Object-Oriented Programming, Systems, Languages, and Applications). Since then, Booch, Jacobson, and Rumbaugh have often been called the “three amigos”. They set themselves the following objectives [1]:

Three amigos

- Use of object-oriented concepts to represent complete systems rather than just one part of the software
- Establishment of an explicit relationship between modeling concepts and executable program code
- Consideration of scaling factors that are inherent in complex and critical systems
- Creation of a modeling language that can be processed by machines but can also be read by human beings

The result of their efforts was the *Unified Modeling Language* (UML) which was submitted in version 1.0 in 1997 in response to the OMG call. A number of former competitors were involved in the creation of version 1.1 that subsequently appeared in 1998. One of the main objectives was a consistent specification of the language core of UML which is documented in the *metamodel* (see Chapter 9). The metamodel defines which model elements the language UML provides and how to use them correctly. For formulating constraints which the model elements have to fulfill, the *Object Constraint Language* (OCL) [36], based on predicate logic, was introduced. In subsequent versions, along with the revision of certain language concepts, mechanisms for the interchangeability of models in the form of the *XML Metadata Interchange format*

Unified Modeling Language (UML)

Metamodel

Object Constraint Language (OCL)

*XML Metadata
Interchange format
(XMI)*

(XMI) [38] were added. In addition to these rather small changes, in 2000 the OMG initiated a modernization process for UML. This finally led to the adoption of the language standard UML 2.0 in 2005. With the exception of small changes which, through interim versions, resulted in the current version 2.4.1, this is the language description of UML that we will get to know and use in this book.

Today, UML is one of the most widespread graphical object-oriented modeling languages. Despite the numerous revisions, its roots (Booch method, OMT, OOSE) are still clearly recognizable. UML is suitable for modeling both complex object relationships and processes with concurrency. UML is a general purpose modeling language, meaning that its use is not restricted to a specific application area. It provides language and modeling concepts and an intuitive graphical notation for modeling various application areas, enabling a software system to be specified, designed, visualized, and documented [43]. The result of modeling with UML is a graphical model that offers different views of a system in the form of various diagrams.

2.2 Usage

UML is not tied to a specific development tool, specific programming language, or specific target platform on which the system to be developed must be used. Neither does UML offer a software development process. UML in fact separates the modeling language and modeling method. The latter can be defined on a project-specific or company-specific basis. However, the language concepts of UML do favor an iterative and incremental process [43].

*Use in the software
development process*

UML can be used consistently across the entire software development process. At all stages of development, the same language concepts can be used in the same notation. Thus, a model can be refined in stages. There is no need for a model to be translated into another modeling language. This enables an iterative and incremental software development process. UML is well-suited for various application areas with different requirements regarding complexity, data volume, real time, etc.

*Generic language
concepts*

The UML model elements and their correct use are specified in the UML *metamodel* [35]. The language concepts are defined so generically that a wide and flexible applicability is achieved. To avoid restricting the use of UML, the standard is (intentionally) vague at various points, permitting different interpretations in the form of semantic variation points. However, this is a two-edged sword; it also leads to different implementations of the language standard by modeling tools, which in turn, unfortunately makes it difficult to exchange models.

Semantic variation point

2.3 Diagrams

In UML, a model is represented graphically in the form of *diagrams*. A diagram provides a view of that part of reality described by the model. There are diagrams that express which users use which functionality and diagrams that show the structure of the system but without specifying a concrete implementation. There are also diagrams that represent supported and forbidden processes. In the current version 2.4.1, UML offers 14 diagrams that describe either the structure or the behavior of a system.

Diagram

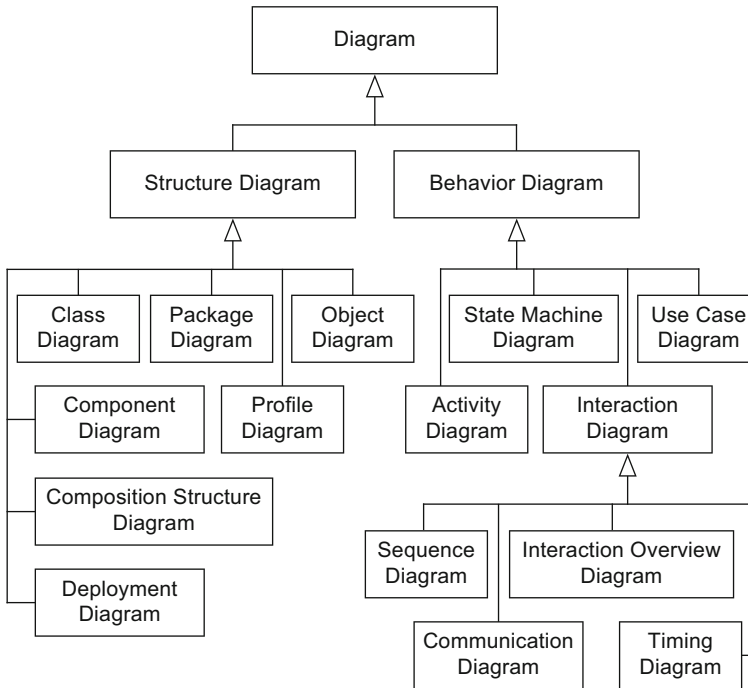


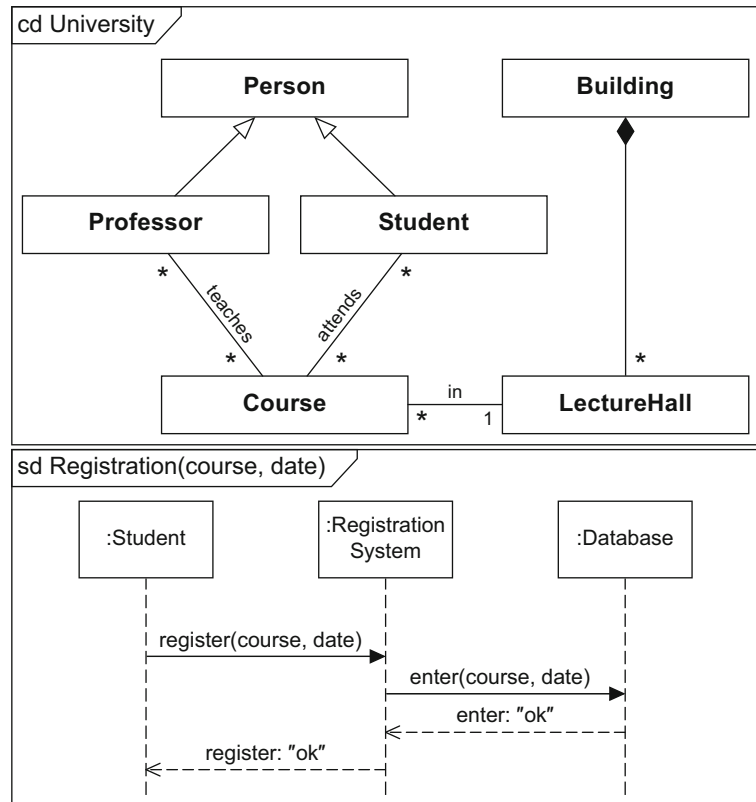
Figure 2.1
UML diagrams

Figure 2.1 shows a taxonomy of the 14 UML diagrams [35], giving a very rough categorization. As the figure shows, we differentiate between *structure diagrams* and *behavior diagrams*. The behavior diagrams include the interaction diagrams, which in turn consist of four diagrams (see Chapter 6).

A diagram is usually enclosed by a rectangle with a pentagon in the top left-hand corner. This pentagon contains the diagram type and the name of the diagram. Optionally, parameters may be specified following the name which then can be used within the diagram. Figure 2.2 con-

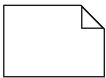
*Notation for diagram
frame*

Figure 2.2
Examples of UML diagram
frames



tains two examples of diagram frames. In particular, it shows a class diagram (cd) with the name University and a sequence diagram (sd) called Registration with the parameters course and date.

Note



A concept that may occur in all diagrams is the *note*. A note can contain any form of expression that specifies the diagram and its elements more precisely—for example, in natural language or in the Object Constraint Language (OCL). Notes may be attached to all other model elements. Figure 2.3 shows an example of the use of a note which specifies in natural language that persons are not permitted to grade themselves. The class Person and the association grades represent concepts of the class diagram that will be introduced in Chapter 4.

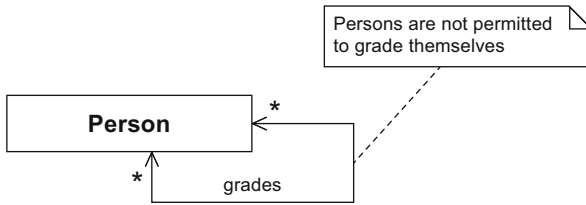


Figure 2.3
Example of a note

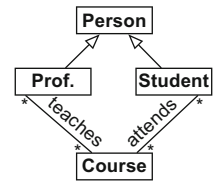
2.3.1 Structure Diagrams

UML offers seven types of diagrams for modeling the structure of a system from different perspectives. The dynamic behavior of the elements in question (i.e., their changes over time) is not considered in these diagrams.

The Class Diagram

Just like the concepts of the object diagram (see next paragraph), the concepts of the *class diagram* originate from conceptual data modeling and object-oriented software development. These concepts are used to specify the data structures and object structures of a system. The class diagram is based primarily on the concepts of *class*, *generalization*, and *association*. For example, in a class diagram, you can model that the classes *Course*, *Student*, and *Professor* occur in a system. Professors teach courses and students attend courses. Students and professors have common properties as they are both members of the class *Person*. This is expressed by a generalization relationship.

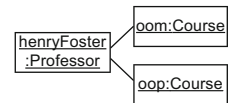
Class diagram
(see Chapter 4)



The Object Diagram

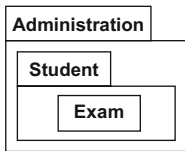
Based on the definitions of the related class diagram, an *object diagram* shows a concrete snapshot of the system state at a specific execution time. For example, an object diagram could show that a professor Henry Foster (henryFoster) teaches the courses Object-Oriented Modeling (oom) and Object-Oriented Programming (oop).

Object diagram
(see Chapter 4)



The Package Diagram

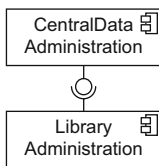
Package diagram



The *package diagram* groups diagrams or model elements according to common properties, such as functional cohesion. For example, in a university administration system, you could introduce packages that contain information about the teaching, the research, and the administrative aspects. Packages are often integrated in other diagrams rather than being shown in separate diagrams.

The Component Diagram

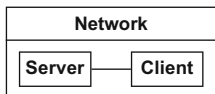
Component diagram



UML pays homage to component-oriented software development by offering *component diagrams*. A component is an independent, executable unit that provides other components with services or uses the services of other components. UML does not prescribe any strict separation between object-oriented and component-oriented concepts. Indeed, these concepts may be combined in any way required. When specifying a component, you can model two views explicitly: the external view (black box view), which represents the specification of the component, and the internal view (white box view), which defines the implementation of the component.

The Composition Structure Diagram

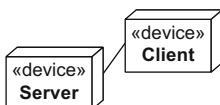
Composition structure diagram



The *composition structure diagram* allows a hierarchical decomposition of the parts of the system. You can therefore use a composition structure diagram to describe the internal structure of classes or components in detail. This enables you to achieve a higher level of detail than, for example, in a class diagram because the modeling is context-specific. You can specify details of the internal structure that are valid precisely for the context under consideration.

The Deployment Diagram

Deployment diagram

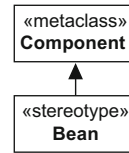


The hardware topology used and the runtime system assigned can be represented by the *deployment diagram*. The hardware encompasses processing units in the form of nodes as well as communication relationships between the nodes. A runtime system contains artifacts that are deployed to the nodes.

The Profile Diagram

Using *profiles*, you can extend UML to introduce domain-specific concepts. The actual core of the language definition of UML, the meta-model, remains unchanged. You can thus reuse modeling tools without having to make adjustments. For example, you can use profiles to introduce the concept of Java Enterprise Beans.

Profile diagram



2.3.2 Behavior Diagrams

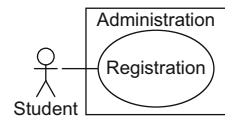
With the *behavior diagrams*, UML offers the infrastructure that enables you to define behavior in detail.

Behavior refers to the direct consequences of an action of at least one object. It affects how the states of objects change over time. Behavior can either be specified through the actions of a single object or result from interactions between multiple objects.

The Use Case Diagram

UML offers the *use case diagram* to enable you to define the requirements that a system must fulfill. This diagram describes which users use which functionalities of the system but does not address specific details of the implementation. The units of functionality that the system provides for its users are called *use cases*. In a university administration system, for example, the functionality Registration would be a use case used by students.

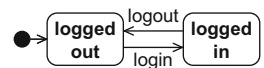
Use case diagram
(see Chapter 3)



The State Machine Diagram

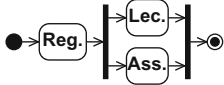
Within their life cycle, objects go through different states. For example, a person is in the state logged out when first visiting a website. The state changes to logged in after the person successfully entered username and password (event login). As soon as the person logs out (event logout), the person returns to the state logged out. This behavior can be represented in UML using the *state machine diagram*. This diagram describes the permissible behavior of an object in the form of possible states and state transitions triggered by various events.

State machine diagram
(see Chapter 5)



The Activity Diagram

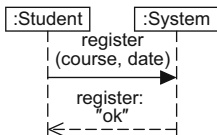
Activity diagram
(see Chapter 7)



You can model processes of any kind using *activity diagrams*: both business processes and software processes. For example, an activity diagram can show which actions are necessary for a student to participate in a lecture and an assignment. Activity diagrams offer control flow mechanisms as well as data flow mechanisms that coordinate the actions that make up an activity, that is, a process.

The Sequence Diagram

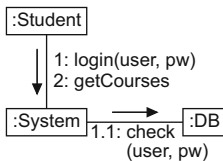
Sequence diagram
(see Chapter 6)



The *sequence diagram* describes the interactions between objects to fulfill a specific task, for example, registration for an exam in a university administration system. The focus is on the chronological order of the messages exchanged between the interaction partners. Various constructs for controlling the chronological order of the messages as well as concepts for modularization allow you to model complex interactions.

The Communication Diagram

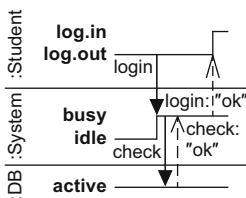
Communication diagram
(see Chapter 6)



Similarly to the sequence diagram, the *communication diagram* describes the communication between different objects. Here, the focus is on the communication relationships between the interaction partners rather than on the chronological order of the message exchange. Complex control structures are not available. This diagram clearly shows who interacts with whom.

The Timing Diagram

Timing diagram
(see Chapter 6)

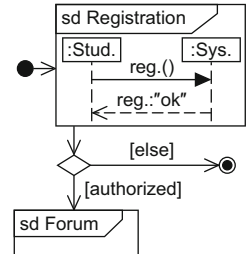


The *timing diagram* explicitly shows the state changes of the interaction partners that can occur due to time events or as a result of the exchange of messages. For example, a person is in the state logged in as soon as the message is received from the university administration system that the password sent is valid.

The Interaction Overview Diagram

The *interaction overview diagram* models the connection between different interaction processes by setting individual interaction diagrams (i.e., sequence diagram, communication diagram, timing diagram, and other interaction overview diagrams) in a time-based and causal sequence. It also specifies conditions under which interaction processes are permitted to take place. To model the control flow, concepts from the activity diagram are used. For example, a user of the university administration system must first log in (which already represents a separate interaction with the system) before being allowed to use further functionalities.

Interaction overview diagram
(see Chapter 6)



2.4 Diagrams Presented in this Book

As already explained in Chapter 1, this book restricts itself to the five most important and most widespread types of UML diagrams, namely the use case diagram, class diagram (including the object diagram), state machine diagram, sequence diagram, and activity diagram. In this book, we present these diagrams in the order in which they would generally be used in software development projects. We begin with the use case diagram, which specifies the basic functionality of a software system. The class diagram then defines which objects or which classes are involved in the realization of this functionality. The state machine diagram then defines the intra-object behavior, while the sequence diagram specifies the inter-object behavior. Finally, the activity diagram allows us to define those processes that “implement” the use cases from the use case diagram.



<http://www.springer.com/978-3-319-12741-5>

UML @ Classroom

An Introduction to Object-Oriented Modeling

Seidl, M.; Scholz, M.; Huemer, C.; Kappel, G.

2015, XII, 206 p., Softcover

ISBN: 978-3-319-12741-5