Chapter 1
Introduction

Unified Modeling Language (UML)

The *Unified Modeling Language* (UML) is a consolidation of the best practices that have been established over the years in the use of modeling languages. UML enables us to present the widely varying aspects of a software system (e.g., requirements, data structures, data flows, and information flows) within a single framework using object-oriented concepts. Before we venture too deeply into UML, however, in this chapter we first explain why modeling is an indispensable part of software development. To do this, we look at what models are and what we need them for. We briefly recapitulate the basic concepts of object orientation before continuing with an overview of the structure of the book.

1.1 Motivation

Imagine that you want to develop a software system that a customer has ordered from you. One of the first challenges you are confronted with is clarifying what the customer actually wants and whether you have understood the customer's exact requirements for the prospective system. This first step is already critical for the success or failure of your project. The question is, how do you communicate with your customer? Natural language is not necessarily a good choice as it is imprecise and ambiguous. Misunderstandings can easily arise and there is a very great risk that people with different backgrounds (e.g., a computer scientist and a business manager) will talk at cross-purposes, which can have serious consequences.

What you need is to be able to create a model for your software. This model highlights the important aspects of the software in a clear form

M. Seidl et al., *UML @ Classroom*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-12742-2_1

of notation that is as simple as possible but abstracts from irrelevant details, just like models in architecture, e.g., construction plans. A construction plan for a building contains information such as the floor plan. Construction materials to be used are not specified at this point in time; they are irrelevant and would make the plan more complicated than necessary. The plan also does not contain any information about how the electrical cables are to be laid. A separate plan is created for this aspect to avoid presenting too much information at once. Just like in architecture, it is important in information technology that people with different backgrounds (e.g., architect and builder) can read, interpret, and implement the model.

Modeling language

Modeling languages were developed precisely for such scenarios and demonstrate clearly defined rules for a structured description of a system. These languages can be *textual* (e.g., a programming language such as Java) or *visual* (e.g., a language that provides symbols for transistors, diodes, etc. that can be combined with one another). Modeling languages can be designed for a specific domain, for example, for describing web applications. On the one hand, these *domain-specific modeling languages* provide tools and guidelines for solving problems in a specific field efficiently; on the other hand, they can also be restrictive. Alternatively, modeling languages can be designed for general purpose use. The language UML, which is the subject of this book, is a general purpose modeling language. We will use UML to get to know the most important concepts of object-oriented modeling.

Object-oriented modeling

Object-oriented modeling is a form of modeling that obeys the object-oriented paradigm. In the following two subsections, we will look briefly at the notion of a model and the main concepts of object orientation. This will provide us with a good basis for our subsequent examination of object-oriented modeling with UML.

1.2 Models

System

Models allow us to describe systems efficiently and elegantly. A *system* is an integrated whole made up of components that are related to one another and influence each other in such a way that they can be perceived as a single, task-based or purpose-based unit. In this regard, they separate themselves from the surrounding environment [52]. Examples of systems are material things, such as cars or airplanes, ecological environments, such as lakes and forests, but also organizational units such as a university or a company. In information technology, we are interested in particular in software systems and thus in models that describe software systems.

Software system

Abstraction

Software systems themselves are based on *abstractions* that represent machine-processible facts of reality. In this context, abstraction means generalization—setting aside specific and individual features. Abstract is the opposite of concrete. Abstracting therefore means moving away from specifics, distinguishing the substance from the incidental, recognizing common characteristics [29].

Selecting means of abstraction

When creating software systems, it is extremely important to select suitable means of abstraction: on the one hand for the implementation, but on the other hand also for the subsequent use of the software systems. Choosing the correct means of abstraction makes programming easier. The individual parts then have simple and small interfaces. New functionality can be introduced without the need for extensive reorganization. Choosing the wrong means of abstraction might result in a number of nasty surprises during implementation: the interfaces will be complicated and it will be difficult to implement changes. You can only manage the ever-increasing complexity of modern software systems with suitable means of abstraction [26]. This is where modeling can provide valuable services.

To develop a better understanding of modeling concepts, below we present widespread and generally recognized definitions of the notion of a model as well as the properties that a good model should possess.

Model

The notion of a *model* is important not only in information technology but also in many other scientific disciplines (mathematics, philosophy, psychology, economics, etc.). Derived from the Latin "modulus", which designates a scale in architecture, during the Renaissance the word "modello" was used in Italy for an illustrative object intended to present the form and design of a planned building to a client and to clarify design and architectural questions. Over the subsequent centuries, the notion of a "model" has been used in various branches of science for a simplified description of complex facts from reality.

Definition by Herbert Stachowiak

In 1973, Herbert Stachowiak proposed a model theory that is distinguished by three characteristics [48]:

1. *Mapping:* a model is always an image (mapping) of something, a representation of natural or artificial originals that can be models themselves.
2. *Reduction:* a model does not capture all attributes of the original, rather only those that seem relevant to the modeler or user of the model.
3. *Pragmatism:* pragmatism means orientation toward usefulness. A model is assigned to an original based on the following questions: *For whom? Why? What for?* A model is used by the modeler or user instead of the original within a specific time frame and for a specific purpose.

Models support a representation of a system that is reduced to the essentials in order to minimize the complexity of the system to manageable aspects. A system is usually described not by one single view but by a number of views that together produce a unified overall picture. Thus, one view might describe the objects involved and their relationship to one another; another view might describe the behavior of a group of objects or present the interactions between different objects.

Properties of models

Models must be created with great care and due consideration. According to Bran Selic [47], five characteristics determine the quality of a model:

- *Abstraction:* a model is always a reduced representation of the system that it represents. Because the details that are irrelevant in a specific context are hidden or removed, it is easier for the user to understand the essence of the whole.
- *Understandability:* simply omitting irrelevant details is not enough to make a model understandable. It is important to present the remaining elements as intuitively as possible—for example, in a graphical notation. The understandability results directly from the expressiveness of the modeling language. Expressiveness can be defined as the ability to present complex content with as few concepts as possible. In this way, a good model reduces the intellectual effort required to understand the content depicted. For example, typical programming languages are not particularly expressive for a human reader as a lot of effort is required to understand the content of the program.
- *Accuracy:* a model must highlight the relevant properties of the real system, reflecting reality as closely as possible.
- *Predictiveness:* a model must enable prediction of interesting but not obvious properties of the system being modeled. This can be done via simulation or analysis of formal properties.
- *Cost-effectiveness:* in the long-run, it must be cheaper to create the model than to create the system being modeled.

Descriptive model

Prescriptive model

Models can be used for various purposes. Thus we distinguish between *descriptive* and *prescriptive* models [17]. *Descriptive models* show a part of the reality to make a specific aspect easier to understand. For example, a city map describes a city in such a way as to help a non-local person to find routes within the city. In contrast, *prescriptive models* are used to offer a construction manual for the system to be developed.

Executable code as model

In this book, we look at how the different aspects of a software system can be modeled using a modeling language—the Unified Modeling Language—such that executable code can be derived either manually or (semi)automatically, or easily understandable documentation can be

created. Incidentally, the executable code, developed in any programming language, such as Java, is also a model. This model represents the problem to be solved and is optimized for execution on computers.

To summarize, there are three applications for models [19]:

- Models as a sketch
- Models as a blueprint
- Models as executable programs

Models as a sketch

Models are used as a *sketch* to communicate certain aspects of a system in a simple way. Here, the model is not a complete mapping of the system. Sketches are actually distinguished by their selectivity, as they are reduced to the essential aspects for solving a problem. Sketches often make alternative solutions visible. These are then discussed in the development team. Thus, models are also used as a basis for discussion.

Models as a blueprint

In contrast to the use of models as sketches, completeness is very important when models are used as a *blueprint*. These models must contain sufficient detail to enable developers to create ready-to-run systems without having to make design decisions. Models used as blueprints often do not specify the whole system, only certain parts. For example, the interface definitions between subsystems are defined in the model, whereby the developers are free to decide on the internal implementation details. If the models are behavioral descriptions, the behavior can also be simulated and tested to identify faults in advance.

Forward and backward engineering

Models as sketches and blueprints can be used for both *forward engineering* and *backward engineering*. In forward engineering, the model is the basis for creating code, while in backward engineering, the model is generated from the code to document the code in a clear and easily understandable way.

Models as executable programs

Finally, models can be used as *executable programs*. This means that models can be specified so precisely that code can be generated from them automatically. In the context of UML, model-based software development has become extremely popular in recent years; it offers a process for using UML as a programming language. We will address this briefly in Chapter 9 of this book, after we have discussed the basics of UML. In some application areas, such as the development of embedded systems, models are already being used instead of traditional programming languages. In other areas, active research is taking place to raise the development of software systems to a new and more easily maintainable and less error-prone abstraction level.

1.3 Object Orientation

Object orientation

If we want to model in an object-oriented style, we must first clarify what *object orientation* means. The introduction of object orientation dates back to the 1960s when the simulation language SIMULA [24] was presented, building on a paradigm that was as natural to humans as possible to describe the world. The object-oriented approach corresponds to the way we look at the real world; we see it as a society of autonomous individuals, referred to as objects, which take a fixed place in this society and must thereby fulfill predefined obligations.

There is not only one single definition for object orientation. However, there is a general consensus about the properties that characterize object orientation. Naturally, objects play a central role in object-oriented approaches. Viewed simply, objects are elements in a system whose data and operations are described. Objects interact and communicate with one another. In general, we expect the concepts described below from an object-oriented approach.

1.3.1 Classes

Class

In many object-oriented approaches, it is possible to define *classes* that describe the attributes and the behavior of a set of objects (the instances of a class) abstractly and thus group common features of objects. For example, people have a name, an address, and a social security number. Courses have a unique identifier, a title, and a description. Lecture halls have a name as well as a location, etc. A class also defines a set of permitted operations that can be applied to the instances of the class. For example, you can reserve a lecture hall for a certain date, a student can register for an exam, etc. In this way, classes describe the behavior of objects.

1.3.2 Objects

Object

The instances of a class are referred to as its *objects*. For example, `lh1`, the Lecture Hall 1 of the Vienna University of Technology, is a concrete instance of the class `LectureHall`. In particular, an object is distinguished by the fact that it has its own identity, that is, different instances of a class can be uniquely identified. For example, the beamer in Lecture Hall 1 is a different object to the beamer in Lecture Hall 2, even

if the devices are of the same type. Here we refer to *identical* devices but not the *same* device. The situation for concrete values of data types is different: the number `1`, which is a concrete value of the data type `Integer`, does not have a distinguishable identity.

An object always has a certain state. A state is expressed by the values of its attributes. For example, a lecture hall can have the state `occupied` or `free`. An object also displays behavior. The behavior of an object is described by the set of its operations. Operations are triggered by sending a message.

1.3.3 Encapsulation

Encapsulation

Encapsulation is the protection against unauthorized access to the internal state of an object via a uniquely defined interface. Different levels of visibility of the interfaces help to define different access authorizations. Java, for example, has the explicit visibility markers `public`, `private`, and `protected`, which respectively permit access for all, only within the object, and only for members of the same class, its subclasses, and of the same package.

1.3.4 Messages

Message

Objects communicate with one another through *messages*. A message to an object represents a request to execute an operation. The object itself decides whether and how to execute this operation. The operation is only executed if the sender is authorized to call the operation—this can be regulated in the form of visibilities (see the previous paragraph)—and a suitable implementation is available. In many object-oriented programming and modeling languages the concept of *overloading* is supported. This enables an operation to be defined differently for different types of parameters. For example, the operator + realizes different behavior depending on whether it is used to add up integers (e.g., `1 + 1 = 2`) or to concatenate character strings (e.g., `"a" + "b" = "ab"`).

Overloading

1.3.5 Inheritance

Inheritance

The concept of *inheritance* is a mechanism for deriving new classes from existing classes. A subclass derived from an existing class (= su-

perclass) inherits all visible attributes and operations (specification and implementation) of the superclass. A subclass can:

- Define new attributes and/or operations
- Overwrite the implementation of inherited operations
- Add its own code to inherited operations

Class hierarchy

Inheritance enables extensible classes and as a consequence, the creation of *class hierarchies* as the basis for object-oriented system development. A class hierarchy consists of classes with similar properties, for example, `Person` ← `Employee` ← `Professor` ← `...` where `A` ← `B` means that `B` is a subclass of `A`.

When used correctly, inheritance offers many advantages: reuse of program or model parts (thus avoiding redundancy and errors), consistent definition of interfaces, use as a modeling aid through a natural categorization of the occurring elements, and support for incremental development, i.e., a step-by-step refinement of general concepts to specific concepts.

1.3.6 Polymorphism

Polymorphism

In general terms, *polymorphism* is the ability to adopt different forms. During the execution of a program, a polymorphic attribute can have references to objects from different classes. When this attribute is declared, a type (e.g., class `Person`) is assigned statically at compile time. At runtime, this attribute can also be bound dynamically to a subtype (e.g., subclass `Employee` or subclass `Student`).

A polymorphic operation can be executed on objects from different classes and have different semantics in each case. This scenario can be implemented in many ways: (i) via *parametric polymorphism*, better known as genericity—here, type parameters are used. In Java for example, the concrete classes are transferred to the operations as arguments; (ii) via *inclusion polymorphism*—operations can be applied to classes and to their direct and indirect subclasses; (iii) via *overloading of operations*; and (iv) via *coercion*, that is, the conversion of types. The first two methods above are known as *universal polymorphism*; the other two methods are referred to as *ad hoc polymorphism* [13].

1.4 The Structure of the Book

In Chapter 2 we give a short overview of UML by recapitulating the history of its creation and taking a brief look at its 14 different diagrams. Then, in Chapter 3, we introduce the concepts of the use case diagram. This diagram enables us to describe the requirements that a system to be developed should satisfy. In Chapter 4 we present the class diagram. This diagram allows us to describe the structure of a system. To enable us to model the behavior of a system, in Chapter 5 we introduce the state machine diagram, in Chapter 6 the sequence diagram, and in Chapter 7 the activity diagram. We explain the interaction of the different types of diagrams in Chapter 8 with three examples. In Chapter 9, we briefly examine advanced concepts that are of significant importance for the practical use of UML.

The concepts are all explained using examples, all of which are based on the typical Austrian university environment. In most cases they represent heavily simplified scenarios. It is not our intention in this book to model one single, continuous system, as there is a high risk that in doing so we would become lost in a multitude of technical details. We have therefore selected examples according to their didactic benefit and their illustrative strength of expression. In many cases, we have therefore made assumptions that, for didactic reasons, are based on simplified presentations of reality.

UML is based entirely on object-oriented concepts. This is particularly noticeable in the class diagram, which can easily be translated into an object-oriented programming language. We will get to know the class diagram and possible translations to program code in Chapter 4. However, UML has not been designed for one specific object-oriented language. For the sake for readability, we use a notion of object-orientation as found in modern programming languages like Java or C#.