Chapter 9
Further Topics

In the preceding chapters, we have learned about the basic concepts of object-oriented modeling using numerous language elements of UML. We have learned how to apply these concepts to create UML diagrams. The diagrams offer different views of a complex system, providing abstraction mechanisms to make the complexity of the system manageable. With these mechanisms, UML offers a strong basis for many applications and we could fill many pages by looking at further topics. To give one example, the Systems Modeling Language (SysML) [37] was developed based on UML and extends a subset of UML with special concepts required for modeling complex physical systems.

However, introducing all further topics considering UML would go beyond the objectives of this book. As an outlook, we will briefly consider four further areas here: (i) structuring models, (ii) defining the language of UML, (iii) extension mechanisms in UML, and (iv) model-based software development. Without going into details, these convey a taster of just what is possible with modeling.

9.1 Structuring Models

If a diagram exceeds a certain size, there is a danger that it will become over-complicated. The multitude of model elements, regardless of whether they are classes, actions, states, and so on, very quickly overwhelms a human reader of a diagram. If the overall system consists of multiple subsystems whose elements are only minimally related to one another, then it is desirable to have a mechanism that groups the elements appropriately. For example, in most cases, it is confusing if the user interface elements are mixed with the elements for the database

M. Seidl et al., *UML @ Classroom*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-12742-2_9

access. In literature, different criteria for grouping elements have been identified [23]:

- *Functional cohesion*: elements with similar purpose are grouped.
- *Informational cohesion*: elements that are strongly related to one another but only weakly related to other elements are grouped.
- *Distribution structure*: when developing a distributed system, the elements are grouped according to their physical distribution—for example, elements on the client and elements on the server.
- *Structuring of the development*: the structuring reflects the division of the development tasks. This is particularly important if there is a team of developers involved in developing the system. Clearly defined responsibilities and interfaces avoid situations in which team members get in each other's way.

In programming languages, the concept of the "namespace" was introduced to enable structuring. In Java for example, this is realized in the form of *packages*. UML offers the *package diagram* for this purpose.

Package diagram

9.1.1 Packages

Package

A *package* allows you to group model elements, such as classes, data types, activities, and states, etc., but can also contain packages itself. The notation for a package is a rectangle with a smaller rectangle on top in the left corner—similar to an index card. The large rectangle contains the elements that the package groups (see Fig. 9.1(a)). The small rectangle contains the package name. If the package content is not relevant, the package name can also be positioned in the large rectangle (see Fig. 9.1(b)). Alternatively, the package content can be represented outside the large rectangle and connected to the package by lines that end in a circle containing a cross on the side of the package (see Fig. 9.1(c)).

Namespace

A model element may be included in a maximum of one package directly. This inclusion in a package defines the namespace in which an element is visible. The name of an element must be unique within a namespace. However, different elements may have the same name within different namespaces. Thus, if package `P1` contains a class `C`, it cannot be confused with class `C` in package `P2`. The package membership is thus a qualifying factor, allowing a clear differentiation between different elements with the same name. The unique name of an element is specified by prepending the package name followed by two colons. This gives us, for example, the two unique names `P1::C` and `P2::C`.

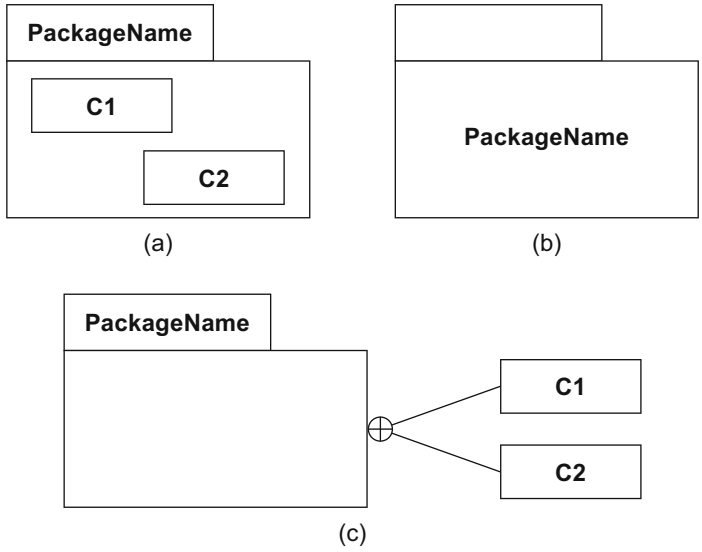

Figure 9.1
Notation alternatives for package diagrams

9.1.2 Importing Elements/Packages

Elements of a specific package can reference one another and communicate with one another without any further details provided this is not restricted by visibilities and navigation directions. For example, an element E that is located in a package P1 can also be used in a package P2 provided P2 does not contain an element with the same name E and P2 is included directly or indirectly in P1. Elements from other packages can either be imported or referenced using qualified names. All elements of the imported package with the corresponding visibility become visible in the importing package. These elements can thus be referenced directly. The name of an imported element is added to the namespace of the package and can then be used without qualification (that is, without namespace::).

Think back to the class diagram (Chapter 4)—there we defined visibilities of attributes, operations, and roles. In doing so, we also became familiar with the visibility package, notated by ~ (see Tab. 4.1 on page 59). This visibility means that the attributes, operations, and roles are only visible for elements within the same package.

Import relationships are denoted by a dashed arrow that points away from the importer and is labeled with «import». Of course, only elements that are visible externally can be imported, such as class C1 in package P3, which is imported by package P1 (Fig. 9.2).

Importing packages

A package can import entire packages in this way. For example, in Figure 9.2, package P1 imports package P2. This makes all visible elements of the imported package visible in the namespace of the importing package. This is handled and noted like every other import relationship.

For a more detailed examination of this topic, see [23].

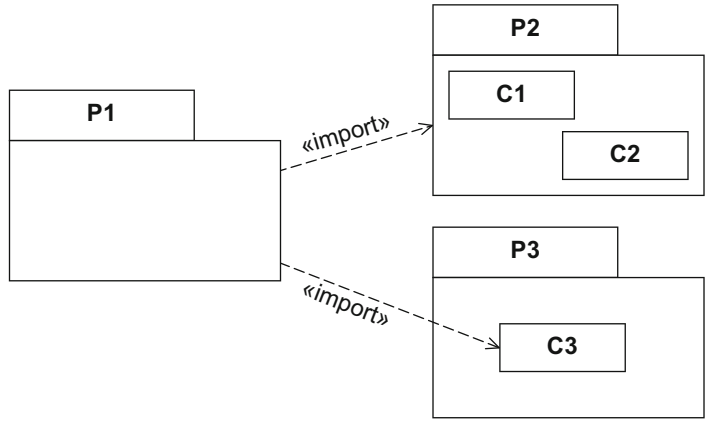

Figure 9.2
Import relationship

9.2 The UML Metamodel

Metamodel

Critical readers will ask themselves how we know how to apply the concepts of the different diagrams. This is described in the *metamodel*. A metamodel is a model that describes a modeling language. It thus states something *about* other models, a fact which is expressed by the Greek prefix "meta", which means "about". In the same way that elements of a model are referred to as model elements, the elements of a metamodel are referred to as metamodel elements.

Superstructure

The UML metamodel, the *superstructure* [35], specifies UML in the form of class diagrams. We use class diagrams to describe which elements a UML diagram may contain and how these elements are applied. However, this also means that a class diagram, which is part of UML, is itself specified by a class diagram. This is comparable with programming languages. Here it is possible and common to write a compiler for C in C, for example. With the concepts that we know from the class diagram, we can now specify modeling languages ourselves. The classes no longer represent persons, courses, and so on but rather language con-

cepts, such as classes, associations, and generalizations, etc. Figure 9.3 shows an example of a simple modeling language that is very similar to the UML class diagram and is itself represented in the form of a class diagram. This metamodel is similar to the metamodel of the real UML class diagram but it is heavily simplified. Almost all classes inherit from the class NamedElement. All direct and indirect instances of this class have a name that identifies them uniquely. Associations are described by a separate class. In contrast, a generalization is represented only as a relationship between classes. Of course, we could also model the generalization as a separate class. This would allow us to specify further properties for the generalization, as in "real" UML. For example, in UML, a generalization can be described as disjoint or overlapping (see 4.6.2), which is something we cannot do in our simplified metamodel.

Abstract syntax

The syntax of UML introduced here is also referred to as *abstract syntax*. If we were to draw an instance of the metamodel from Figure 9.3—that is, an object model—an association would be depicted as a separate element that connects other elements. An abstract class is then identified by the isAbstract flag. This notation is not particularly user-friendly. We have expressed associations simply with a direct connection between the classes that are in a relationship with one another and we have specifically identified abstract classes. This type of notation is more intuitive for human users. Therefore, in addition to the abstract syntax, UML defines the *concrete syntax*, which is a notation optimized for humans.

Concrete syntax

Infrastructure

Of course, there are many other important details about metamodels and metamodeling that we could discuss. For example, the obvious question is how the language used to create the metamodel is defined. For this purpose UML has the *infrastructure* [34] which introduces the required concepts. This is the metametamodel of UML. We could continue these definitions to infinity but the specification does not go beyond the metametalevel.

9.3 UML Extension Mechanisms

As a general purpose modeling language, UML provides a stable basis for a wide variety of requirements. It is not defined for specific application domains or for any specific technology. However, in some circumstances, UML is too general and using it involves a considerable amount of effort. In such cases, the use of a language optimized for the given domain and therefore offering special concepts is advantageous.

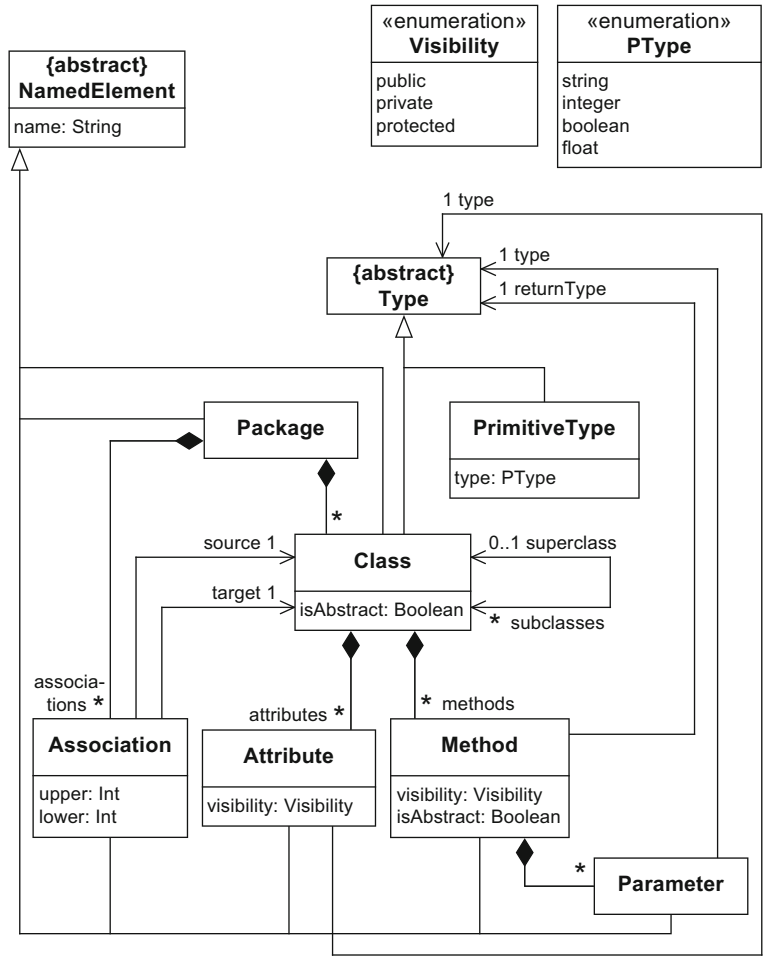

Figure 9.3
Heavily simplified metamodel of a class diagram

This type of language can be defined in one of the following three ways:

- Creation of a new metamodel
- Extension and modification of the UML metamodel
- Extension of the UML metamodel with language-inherent mechanisms

Creation of a new metamodel

If the description of the systems to be modeled requires language concepts that are very different to the language concepts of UML, it is probably more practical not to use the language definition of UML and to define your own modeling language by creating a *new metamodel*.

Extension and modification of the UML metamodel

As an alternative to creating a new metamodel, you can also *extend and modify* the UML metamodel in accordance with your requirements. In this case, you introduce new metaclasses and new associations between the metaclasses or overwrite existing properties. This type of extension is also referred to as a heavyweight extension. In many cases, it makes the interoperability of modeling tools more difficult, as of course not all of the tools support the extended metamodel.

Extension of the UML metamodel with language-inherent mechanisms

UML itself offers *language-inherent extension mechanisms*, that is, extension options that are provided in the language itself. These are already defined at the metametamodel level, that is, in the UML infrastructure. The extensions are thus implemented in a controlled way; existing language concepts can only be extended and made more specific and they must not be changed or generalized. This lightweight extension mechanism retains the interoperability between the different modeling tools. These lightweight extensions are based on *stereotypes* and *profiles*, which we will look at more closely below.

9.3.1 Stereotypes and Profiles

In Section 9.2, we saw that the UML metamodel itself is also a model. It describes the language elements of UML. The metamodel defines, for example, that a class diagram contains classes, associations, and generalizations, etc. The classes of the metamodel are referred to as metaclasses. A *stereotype* is a special metaclass in the UML metamodel. It allows you to extend any metaclass with additional meta-attributes (tag definitions) and to make it more specific using additional constraints. A metaclass for which a stereotype has been defined remains unchanged. In the simplest case, stereotypes are used to classify metaclasses without introducing additional meta-attributes and constraints.

Stereotype

A stereotype is denoted like a class, with the keyword «stereotype» above the name in the first compartment. The second compartment usually contains the meta-attributes. The constraints can be specified either after the meta-attributes or as a note. You can also specify a pictogram for a stereotype. This symbol is used later with the corresponding elements. Figure 9.4 shows an example of a stereotype. The stereotype Entity contains two meta-attributes, author and year, as well as two constraints. One constraint states that the values of the meta-attribute author can have a maximum of ten characters, with the other constraint stating that the value of the meta-attribute year must be smaller than 2006.

Extension relationship

A stereotype extends one or more metaclasses. This *extension relationship* is depicted as an arrow with a continuous line and filled arrowhead. The arrow points away from the stereotype to the metaclass. The

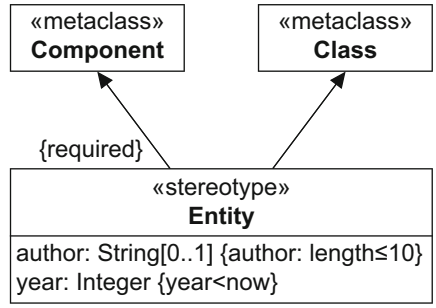

Figure 9.4
Specification of the stereotype "Entity"

extended metaclass is identified with the keyword «metaclass». Stereotypes defined in this way are optional, meaning that you do not have to use them in the modeling. You can also use model elements that are based on the original definition of the metaclass. To force the use of a stereotype, the extension relationship must be identified as mandatory through the use of the keyword {required}. In Figure 9.4, the stereotype Entity is thus optional for the metaclass Class but mandatory for the metaclass Component.

Stereotypes can be connected to one another via an inheritance relationship. In this context, they can be defined as abstract. A derived stereotype inherits all meta-attributes, constraints, and extension relationships of its higher level stereotypes. Figure 9.5 shows an example of inheritance for stereotypes. The stereotypes Entity and Session are derived from the abstract stereotype Bean. The keyword {abstract} in the name field identifies Bean as abstract. The stereotype Session introduces a new meta-attribute state. The extension relationship between Component and Bean is mandatory, meaning that when a component is modeled, the stereotype Bean must always be used with a specific value.

Profile

Profiles group stereotypes defined for a specific purpose. A profile is a special form of the package that we learned about at the beginning of this chapter. Therefore, a profile has the same notation as a package,

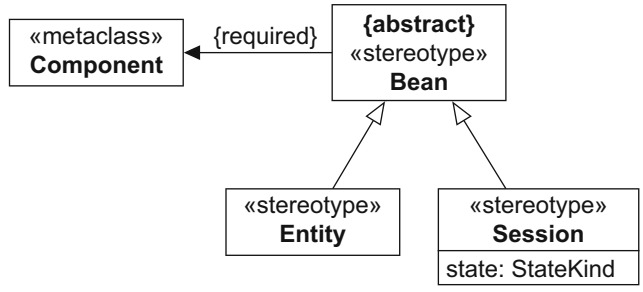

Figure 9.5
Example of inheritance with stereotypes [35]

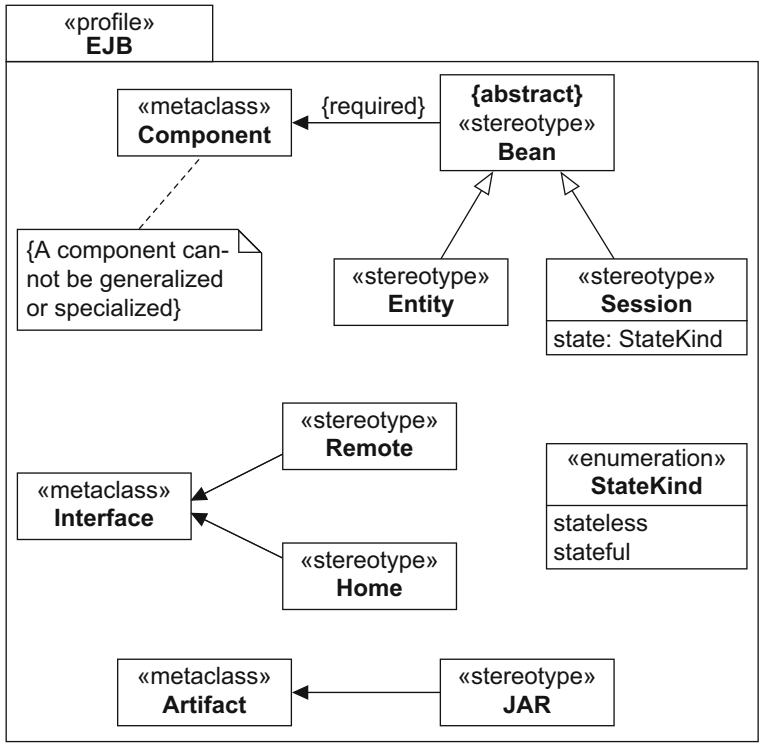

Figure 9.6
Example of a profile [35]

with the keyword «profile» prepended to the profile name. Figure 9.6 shows an example of a profile [35] for Enterprise JavaBeans (EJB).

9.3.2 Applying Stereotypes of a Profile

To use stereotypes in a specific application, you must first integrate the profile that contains the stereotypes. You do this with a dashed arrow with an open arrowhead pointing away from the package of the application towards the profile. This arrow is labeled with the keyword «apply». This imports the stereotypes defined in the profile into the namespace of the package. Figure 9.7 shows an example of the application of a stereotype. The package with the name UserModel contains a component Customer with the stereotype Session. The value for the meta-attribute state is specified in a note. The symbol that appears in the upper right corner of Customer designates components in UML notation.

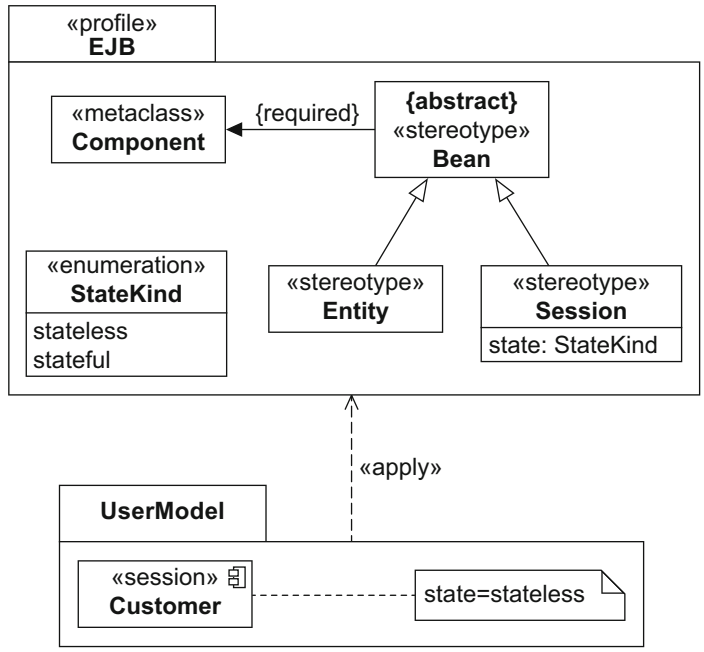

Figure 9.7
Example of the application of a stereotype

9.4 Model-Based Software Development

Models often serve as construction plans for software. For the developers, they stipulate which properties the end product should have. They specify the requirements that a software system must fulfill and describe which components occur in the system and how these components interact. Models are thus the basis for the development of executable code, which is traditionally created by human programmers. If the model and the code are to be kept up-to-date, any changes made must be implemented in both the model and the code. This involves additional effort.

The next step in simplifying the software development process is obvious—namely the automatic generation of the code from the models. This brings us to model-based software development. Models are significantly more than pretty pictures for documenting a software system and for use as sketches or blueprints. In model-based software development, the executable system is created from the models. Here, therefore, models have the same value as code.

Model-based software development

In *model-based software development (MBSD)*, source code is created partially or completely from models. Programming is thus replaced by modeling. The models must describe the function of the system to be developed as precisely as possible. The semantics of the modeling

languages used must be defined unambiguously so that the language elements can be transformed into code uniquely. General purpose modeling languages such as UML may be very flexible but they are often too general for specific applications, meaning that specific concepts are missing. For example, in the description of a web application, the "hyperlink" is a central concept that is not available in UML. Therefore, in model-based software development, smaller languages tailored to the respective domain are often used. These languages are also referred to as *domain-specific languages (DSL)*. Of course, they can be defined based on UML, as we saw in the previous section on stereotypes and profiles.

General purpose modeling language

Domain-specific language (DSL)

In MBSD, models are transformed into other models that, for example, contain specific details about the target platform of the system. Alternatively, executable code is generated in common programming languages. The expert knowledge that is required to create an executable program is therefore invested in these model transformations. It is thus retained in an infrastructure that can be reused again. In recent years, numerous special languages and frameworks have been introduced for creating these transformations.

The aim of MBSD is to significantly simplify the development of applications by replacing programming with modeling, without the developers or modelers having to have specific knowledge about the platforms used.

To summarize, model-based software development aims to provide the following advantages [49]:

- *Increase the speed of development:* code, which is often repeated, is created automatically and adapted to the current situation. There is a significant decrease in "copy and paste" activities.
- *Increase the quality of the software:* the automatic transformations reduce the risk of errors in the implementation. The code is also created independently of the abilities and experience of a developer. Thanks to the tool support, the implementation of the software architecture is better.
- *Central troubleshooting:* errors only have to be corrected in the model and they are then eliminated in the corresponding code parts when the transformation is executed again. The transformation rules may have to be adapted if they no longer match the requirements. However, this only has to be done once.
- *Increase the reuse:* modeling languages, in particular DSLs, only have to be created once. They aggregate expert knowledge that can then be used in different projects.

- *Handling complexity through abstraction:* the complexity of the implementation language remains hidden in the modeling languages; technical implementation details do not have to be considered.
- *Portability:* the code can be generated for different platforms from one model.

Models are used not only to generate the implementation of the system to be developed; they are also used for simulations, for analyzing system properties, for generating test cases, and for verifying and validating the system to be developed. Model-based software development can be implemented in different ways. For a detailed introduction to this topic, see [6, 49].