Chapter 4
The Class Diagram

Class diagram

We use the *class diagram* to model the static structure of a system, thus describing the elements of the system and the relationships between them. These elements and the relationships between them do not change over time. For example, students have a name and a matriculation number and attend various courses. This sentence covers a small part of the university structure and does not lose any validity even over years. It is only the specific students and courses that change.

The class diagram is without doubt the most widely used UML diagram. It is applied in various phases of the software development process. The level of detail or abstraction of the class diagram is different in each phase. In the early project phases, a class diagram allows you to create a conceptual view of the system and to define the vocabulary to be used. You can then refine this vocabulary into a programming language up to the point of implementation. In the context of object-oriented programming, the class diagram visualizes the classes a software system consists of and the relationships between these classes. Due to its simplicity and its popularity, the class diagram is ideally suited for quick sketches. However, you can also use it to generate program code automatically. In practice, the class diagram is also often used for documentation purposes.

Object diagram

Before we introduce the concepts of the class diagram, let us first take a look at *objects*, which are modeled in *object diagrams*. Object diagrams allow you to depict concrete objects that appear in a system at a specific point in time. Classes provide schemas for characterizing objects and objects are instances of classes. The object diagram visualizes instances of classes that are modeled in a class diagram.

M. Seidl et al., *UML @ Classroom*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-12742-2_4

4.1 Objects

A system contains numerous different individuals. Individuals might be not only persons but also animals, plants, inanimate objects, artifacts, etc. that can be identified uniquely. For example, as part of her *IT Studies* program, *Helen Lewis* attends the lecture *Object-Oriented Modeling (OOM)* at the university. *Helen Lewis*, *IT Studies*, and *Object-Oriented Modeling* are individuals (concrete objects) in a university administration system and are in a relationship with one another.

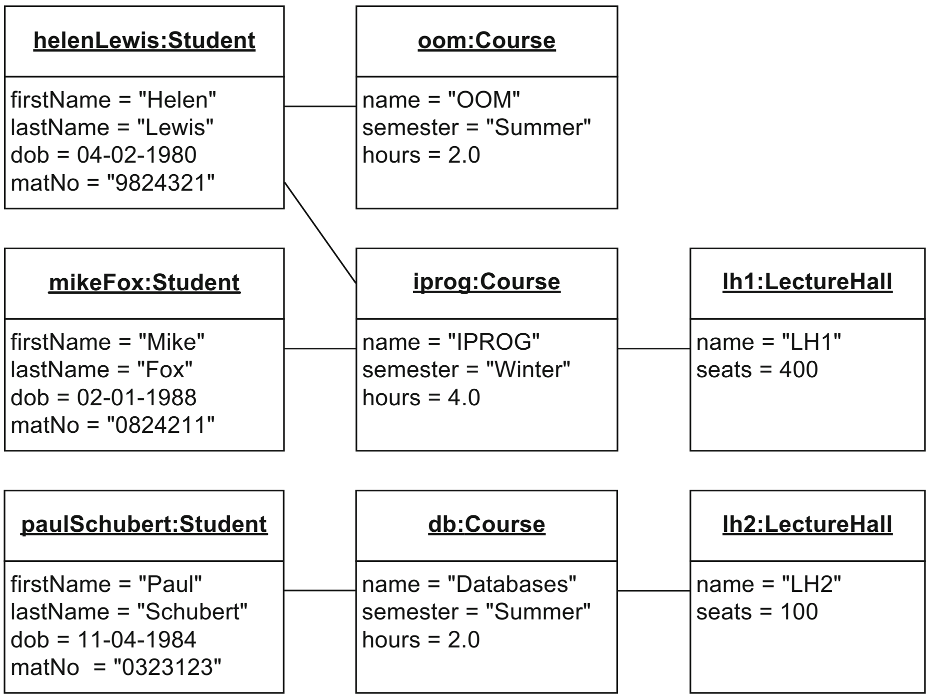


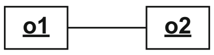
Figure 4.1
Example of an object diagram

Object diagram

In UML, we depict concrete *objects* of a system and their relationships (*links*) using *object diagrams*. Figure 4.1 shows a small object diagram. It contains three student objects: helenLewis, mikeFox, and paulSchubert. The first name and the last name of the object helenLewis are *Helen* and *Lewis* respectively. We also know the date of birth and matriculation number for each of these objects. The system contains the three courses oom (Object-Oriented Modeling), iprog (Introduction to Programming), and db (Databases). The course iprog takes place in lecture hall lh1 and the course db takes place in lecture hall lh2. There is no corresponding information for oom. Student helenLewis attends the two courses oom and iprog. Student mikeFox also attends iprog; course db is attended only by student paulSchubert (at least, among these three students).

Object

Link



An object has a unique identity and a number of characteristics that describe it in more detail. It rarely appears in isolation in a system; instead, it usually interacts and communicates with other objects. The relationships between the objects are referred to as *links*. The characteristics of an object include its *structural characteristics* (attributes) and its *behavior* (in the form of operations). Whilst concrete values are assigned to the attributes in the object diagram, operations are generally not depicted. Operations are identical for all objects of a class and are therefore usually described exclusively for the class.

In the object diagram, an object is shown as a rectangle which can be subdivided into multiple compartments. The first compartment always contains information in the form objectName:Class. This information is centered and underlined. In Figure 4.1 for example, helenLewis and oom are object names and Student and Course are classes. The object name or the specification of the class may be omitted. If only a class name is given, it must be preceded by a colon. If the class name is omitted, the colon is also omitted. If the object name is omitted, this object is referred to as an *anonymous object*. Examples of different notation alternatives are shown in Figure 4.2.

Anonymous object

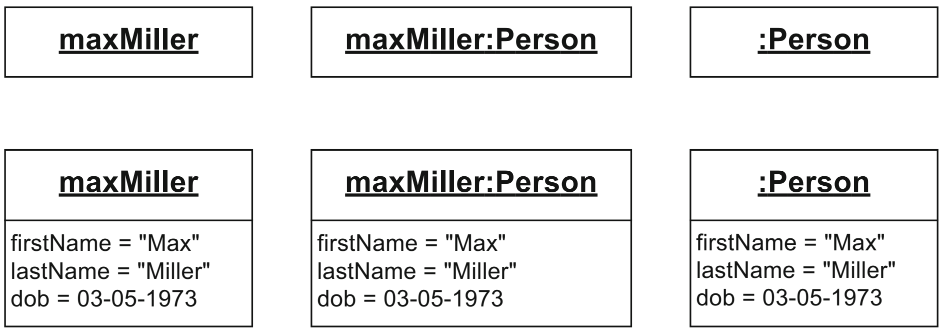


Figure 4.2
Notation alternatives for objects

If the rectangle has a second compartment, this compartment contains the attributes of the object and the current values of these attributes (see Fig. 4.1 and Fig. 4.2). A link is represented as a continuous line connecting the objects that are in a relationship with one another. Although the name of an object must be unique, different objects can have attributes with identical values. If, in our system, there were two people with the first name *Max* and the last name *Miller*, and both were born on the same day, we would have to represent them using different objects with different object names (e.g., maxMiller1 and maxMiller2). However, their attribute values would be identical.

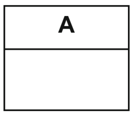
The values of the attributes generally change over time. For example, if the person *Max Miller* changes his last name, the individual as a whole does not change, only the value of the attribute lastName. The object diagram therefore always represents only a snapshot of objects at a specific moment in time and the objects can develop further and change as time passes. If specific objects are not represented in the object diagram, this does not mean that they do not exist; it merely expresses that the unrecorded objects are not important for the moment.

From object to class

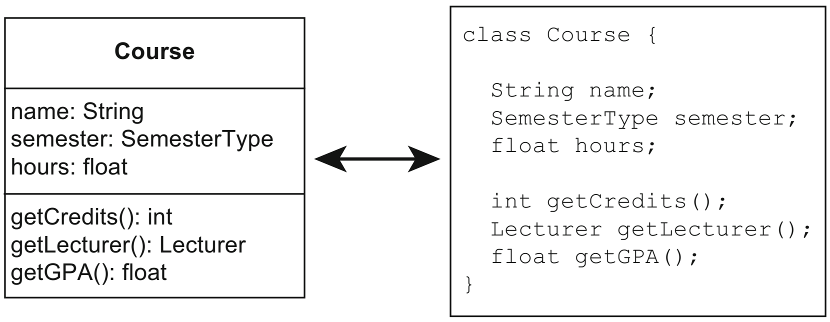
Many individuals that appear in a system have identical characteristics and behavior. For example, persons always have a first name, a last name, and a date of birth. Students also have a matriculation number. Courses always have a name and a number of hours, as well as a semester in which they take place. Information about the lecture halls includes the number of seats available. If every person, every course, and every lecture hall of the system were to be modeled individually, the model would soon become over-complicated and impossible to maintain. Using classes enables you to describe similar objects without having to detail each and every object individually.

4.2 Classes

Class



A *class* is the construction plan for a set of similar *objects* that appear in the system to be specified. Classes can characterize, for example, persons (e.g., students), things (e.g., buildings), events (e.g., courses or exams), or even abstract concepts such as groups. In object-oriented programming languages like Java [4], programs are created based on classes. Figure 4.3 compares a class definition from a UML class diagram with a class definition in Java.



```
class Course {

  String name;
  SemesterType semester;
  float hours;

  int getCredits();
  Lecturer getLecturer();
  float getGPA();
}
```

Figure 4.3
Definition of a class in UML and Java

Instance
Characteristics of classes ...

Objects represent the concrete forms of classes and are referred to as their *instances*. The relevant characteristics of the instances of a class are described through the definition of structural characteristics (*attributes*) and behavior (*operations*). Operations enable objects to communicate with one another and to act and react.

An attribute allows you to store information that is known for all instances but that generally has different specific values for each instance. Operations specify how specific behavior can be triggered on individual objects. For example, the class Course from Figure 4.3 has the attributes name and hours. Figure 4.1 shows concrete forms of these attributes. Possible operations of this class are getGPA() and getLecturer(), which return the grade point average or lecturer for a course respectively.

... are attributes and operations

To ensure that a model remains clear and understandable, we generally do not model all of the details of the content: we only include the information that is relevant for the moment and for the system to be implemented. This means that we *abstract* from reality to make the model less complex and to avoid an unnecessary flood of information. In the model, we restrict ourselves to the essentials. For example, in a university administration system, it is important to be able to manage the names and matriculation numbers of the students; in contrast, their shoe size is irrelevant and is therefore not included.

Level of detail

Abstraction

4.2.1 Notation

In a class diagram, a class is represented by a rectangle that can be subdivided into multiple compartments. The first compartment must contain the name of the class, which generally begins with a capital letter and is positioned centered in bold font (e.g., Course in Figure 4.4).

According to common naming conventions, class names are singular nouns. The class name should describe the class using vocabulary typical for the application domain. The second compartment of the rectangle contains the *attributes* of the class, and the third compartment the

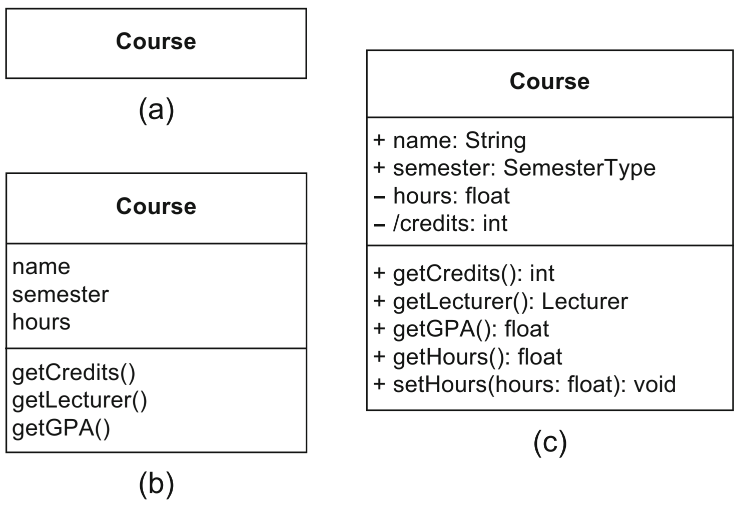


Figure 4.4
Representation of a class and its characteristics

operations of the class. The contents of these compartments are positioned left-justified and are optional. In general, the level of detail in these compartments reflects the respective phase of the software development process in which the class is being examined. While the class diagram excerpt in Figure 4.4(a) does not contain any details of the class Course, the diagram in Figure 4.4(b) is the result of a more detailed analysis of the characteristic features of courses, showing specifically that the class Course contains three attributes and three operations. The diagram in Figure 4.4(c) presents even more detail (such as the type information and visibilities), including information that is relevant for implementation or for automatic code generation. If specific information is not included in the diagram, this does not mean that it does not exist; it simply means that this information is not relevant at this moment in time or is not included for practical reasons, for example, to prevent the diagram from becoming over-complicated. Attributes and operations are usually accessed via their names, which, according to naming conventions, begin with a lower case letter.

4.2.2 *Attributes*

Attribute

Type

Figure 4.5 shows the syntax of attributes. An attribute has at least a name. The type of the attribute may be specified after the name using : Type. Possible attribute types include primitive data types, such as integer and string, composite data types, for example a date, an enumeration, or user-defined classes (see Section 4.8). By specifying name: String, for example, we define the attribute name with type String. Figure 4.6 shows further examples of attribute types. We will look at the subsequent, optional multiplicity specification in more detail in the next section.

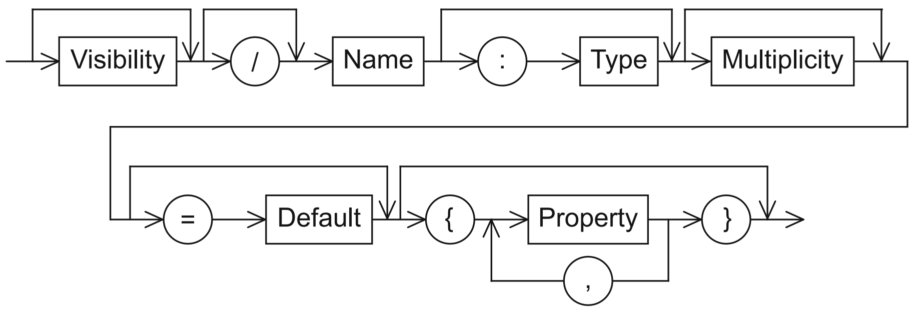


Figure 4.5
Syntax of the attribute specification

To define a *default value* for an attribute, you specify = *Default*, where *Default* is a user-defined value or expression (see Fig. 4.6). The system uses the default value if the value of the attribute is not set explicitly by the user. Thus it is impossible that at some point in time, an attribute has no value. For example, if in our system, a person must always have a password, a default password *pw123* is set when a new person is entered in the system. This password is valid until it is reseted.

Default value

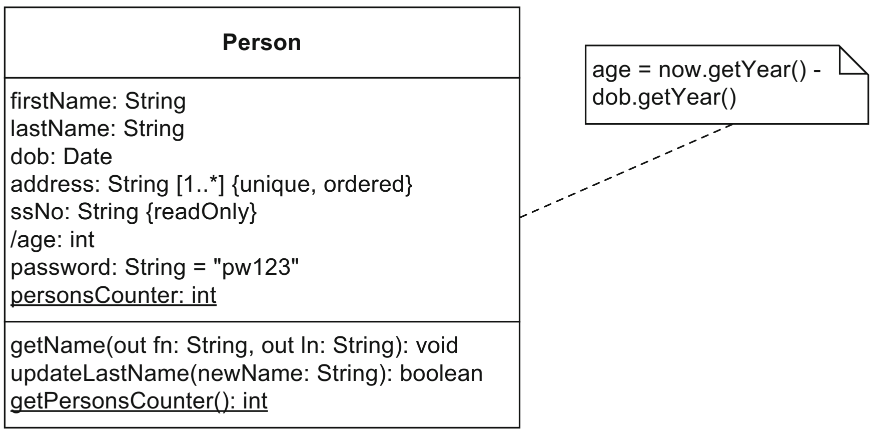

Person
firstName: String
lastName: String
dob: Date
address: String [1..*] {unique, ordered}
ssNo: String {readOnly}
/age: int
password: String = "pw123"
personsCounter: int
getName(out fn: String, out ln: String): void
updateLastName(newName: String): boolean
getPersonsCounter(): int
age = now.getYear() - dob.getYear()

Figure 4.6
Properties of attributes

You can specify additional *properties* of the attribute within curly brackets. For example, the property {readOnly} means that the attribute value cannot be changed once it has been initialized. In the example in Figure 4.6, the social security number ssNo is an attribute that must not be changed. Further properties will be introduced in the next section within the description of multiplicity specifications.

Properties of attributes

The specification of a forward slash before an attribute name indicates that the value of this attribute is derived from other attributes. An example of a *derived attribute* is a person's age, which can be calculated from the date of birth. In Figure 4.6, a note contains a calculation rule for determining a person's age. Depending on the development tool used, such notes are formulated in natural language, in a programming language, or in pseudocode. The optional visibility marker (+, –, #, or ∼) in front of an attribute name or operation name as shown in Figure 4.4(c) is discussed in detail on page 58.

4.2.3 Multiplicities

The *multiplicity* of an attribute indicates how many values an attribute can contain. This enables you to define arrays, just like in programming languages. The multiplicity is shown as an interval enclosed by square

Multiplicity

brackets in the form [minimum .. maximum], whereby minimum and maximum are natural numbers indicating the lower and upper limits of the interval. The value of minimum must be smaller than or equal to the value of maximum. If there is no upper limit for the interval, this is expressed with an asterisk ∗. The class Person in Figure 4.6 contains an attribute address: String [1..∗]. This denotes that a person has at least one and possibly multiple addresses. If minimum and maximum are identical, you do not have to specify the minimum and the two dots. For example, [5] means that an attribute adopts exactly *five* values. The expression [∗] is equivalent to [0..∗]. If you do not specify a multiplicity for an attribute, the value *1* is assumed as default, which specifies a single-valued attribute. The valid notation for multiplicities is summarized in Figure 4.7.

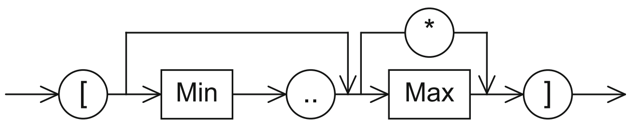


Figure 4.7
Syntax of the multiplicity specification

If an attribute can adopt multiple values, it makes sense to specify whether the attribute is:

- A set (no fixed order of elements, no duplicates)
- A multi-set (no fixed order of elements, duplicates possible)
- An ordered set (fixed order, no duplicates)
- A list (fixed order, duplicates possible)

Unique, non-unique, ordered, unordered

You can make this specification by combining the properties {non-unique} and {unique}, which define whether duplicates are permitted or not permitted, and {ordered} and {unordered}, which force or cancel a fixed order of the attribute values. For example, the attribute address: String [1..∗] {unique, ordered} contains all the addresses for a person (see Fig. 4.6). As each address should only be contained once, the attribute is labeled {unique}. By specifying {ordered}, we express that the order of the addresses is important. For example, the first address could be interpreted as the main residence.

4.2.4 Operations

Operation

Operations are characterized by their name, their parameters, and the type of their return value (see Fig. 4.8). When an operation is called in a program, the behavior assigned to this operation is executed. In programming languages, an operation corresponds to a method declaration

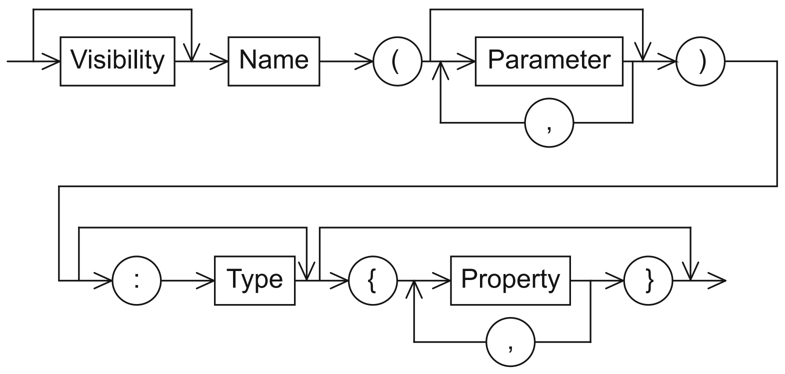


Figure 4.8
Syntax of the operation specification

or function declaration which is defined but not implemented. The class diagram is not suitable for describing the behavior of objects in detail as it only models signatures of the operations that the objects provide; it does not model how these operations are actually implemented. UML offers special behavior diagrams for depicting the implementation of operations, for example the activity diagram (see Chapter 7).

Parameters

In a class diagram, the operation name is followed by a list of *parameters* in parentheses. The list itself may be empty. A parameter is depicted similarly to an attribute. The only obligatory information is the name of the parameter. The addition of a type, a multiplicity, a default value, and further properties, such as ordered, unique, or their negated counterparts is optional (see Fig. 4.9).

Return value

The optional return value of an operation is specified with the type of the return value. In Figure 4.6, the class Person has an operation updateLastName(newName: String): boolean. The only parameter, newName, has the type String and specifies the new name for a person. The return value has the type boolean. If *true* is returned, the renaming was successful, otherwise *false* is returned.

Input and output parameters

If required, you can also prepend a direction to the parameter name. This direction can have one of the following values: in, out, or inout (see Fig. 4.9). The value indicates whether the parameter is an *input parameter*, an *output parameter*, or both. If a parameter has the direction in, this indicates that when the operation is used, a value is expected from this parameter. The specification of the direction out expresses that after the execution of the operation, the parameter has adopted a new value. If an operation should have multiple return values rather than just one, you can express this using multiple parameters with the direction out. The specification of inout indicates a combined input/output parameter. If no direction is specified, in is the default value. In Figure 4.6, the operation getName(out fn: String, out ln: String) has two parameters with the direction value out. For example, if we use the operation getName in a program by calling `getName(firstName, lastName)`, whereby `firstName` and `lastName` are variables in the sense of an imperative programming language, successful execution of the operation produces the following results: the variable `firstName` contains the first name and the variable `lastName` contains the last name of the object of type Person on which the operation `getName` was called.

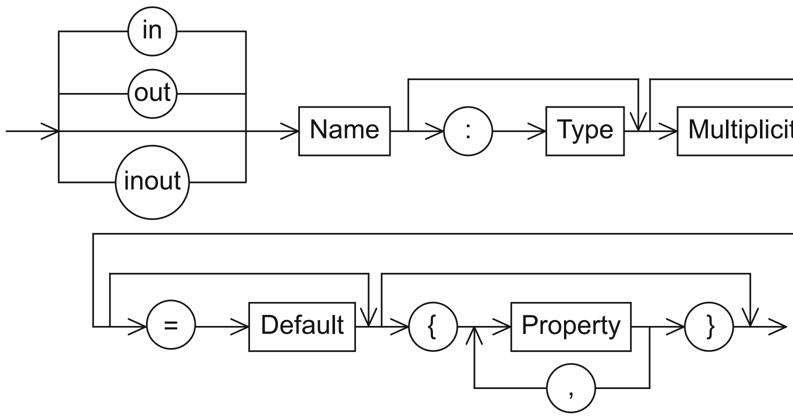


Figure 4.9
Syntax of the parameter specification

4.2.5 Visibility Markers

Visibility

The *visibility* of attributes and operations specifies who is and who is not permitted to access them. If an attribute or operation does not have a visibility specified, no default visibility is assumed. Table 4.1 lists the types of visibilities and their meaning in UML. Only an object itself knows the values of attributes that are marked as private. In contrast, anyone can view attributes marked as public. Access to protected attributes is reserved for the class itself and its subclasses. If a class has a package attribute, only classes that are in the same package as this class may access this attribute. Accordingly, the visibility of an operation specifies who is permitted to use the functionality of the operation. Examples are given in Figure 4.4(c) on page 53. Note that the meaning of visibilities can vary in different programming and modeling languages even if they have the same name in the different languages.

Information hiding

Visibilities are used to realize *information hiding*, an important concept in computing. Marking the attributes that represent the state of an object as private protects this state against unauthorized access. Access is therefore only possible via a clearly defined interface, such as via operations that are declared public.

In some cases, class diagrams contain only those attributes and operations that are visible externally. Attributes and operations of classes that are marked as private are often omitted, as they are important for the realization, that is, the implementation of a class, but not for its use. Therefore, whether or not attributes and operations marked as private are specified depends on the intention behind and the time of creation of the class diagram.

Table 4.1
Visibilities

Name	Symbol	Description
public	+	Access by objects of any classes permitted
private	−	Access only within the object itself permitted
protected	#	Access by objects of the same class and its subclasses permitted
package	∼	Access by objects whose classes are in the same package permitted

4.2.6 Class Variables and Class Operations

Attributes are usually defined at instance level. If, for example, a class is realized in a programming language, memory is reserved for every attribute of an object when it is created. Such attributes are also referred to as *instance variables* or *instance attributes*. In Figure 4.10 for example, lastName and dob are instance variables. If, in an object-oriented program generated from this class of diagram, `person1` is an instance of the class `Person`, for example, `person1.lastName` can be used to refer to the last name of the person. Access to this person's date of birth is not possible as the visibility of the attribute `dob` is private. To find out the date of birth of `person1`, the function `person1.getDob()` must be called. An operation such as getDob() can only be executed if a corresponding instance that offers this operation was created beforehand. In our case, this is the instance `person1`. An operation may use all visible instance variables.

Synonyms:

- *Instance variable*
- *Instance attribute*

In contrast to instance variables, *class variables* are created only once for a class rather than separately for every instance of this class. These variables are also referred to as *static attributes* or *class attributes*. Counters for the number of instances of a class (see Fig. 4.10) or constants such as π are often realized as static attributes. In the class diagram, static attributes are underlined, just like *static operations*. Static operations, also called *class operations*, can be used if no instance of the corresponding class was created. Examples of static operations are mathematical functions such as `sin(x)` or constructors. Constructors are special functions called to create a new instance of a class. The method invocation `Person.getPCounter()` uses the static opera-

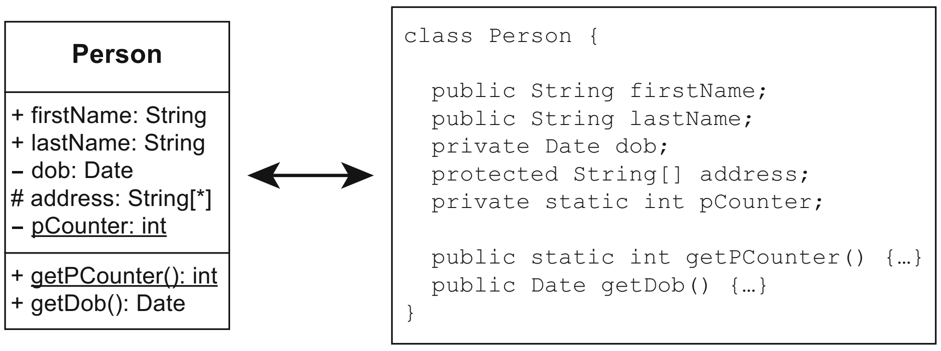
Synonyms:

- *Class variable*
- *Class attribute*
- *Static attribute*

Synonyms:

- *Class operation*
- *Static operation*

tion `getPCounter()` defined in Figure 4.10; the operation is called directly via the class and not via an instance. Unless stated otherwise, attributes and operations denote instance attributes and instance operations in most object-oriented languages. We also follow this convention in this book.



```
class Person {

  public String firstName;
  public String lastName;
  private Date dob;
  protected String[] address;
  private static int pCounter;

  public static int getPCounter() {…}
  public Date getDob() {…}
}
```

Figure 4.10
Translation of a class from UML to Java

4.3 Associations

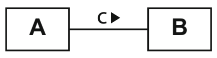
Association

Associations between classes model possible relationships, known as *links*, between instances of the classes. They describe which classes are potential communication partners. If their attributes and operations have the corresponding visibilities, the communication partners can access each other's attributes and operations. A class diagram can be viewed as a graph in which the classes represent the nodes and the associations represent the edges. Figure 4.11 depicts a class diagram and a valid object diagram. The class diagram shows that the classes Professor and Student are related via the association givesLectureFor. In the role as a lecturer, a professor has zero or more students and one student has zero or more professors in the role of lecturer. The object diagram models a concrete scenario.

4.3.1 Binary Associations

Binary association

Reading direction



A *binary association* allows us to associate the instances of two classes with one another. The relationships are shown as edges (solid line) between the partner classes involved. The edge can be labeled with the *name of the association* optionally followed by the *reading direction*, a small, black triangle. The reading direction is directed towards one end

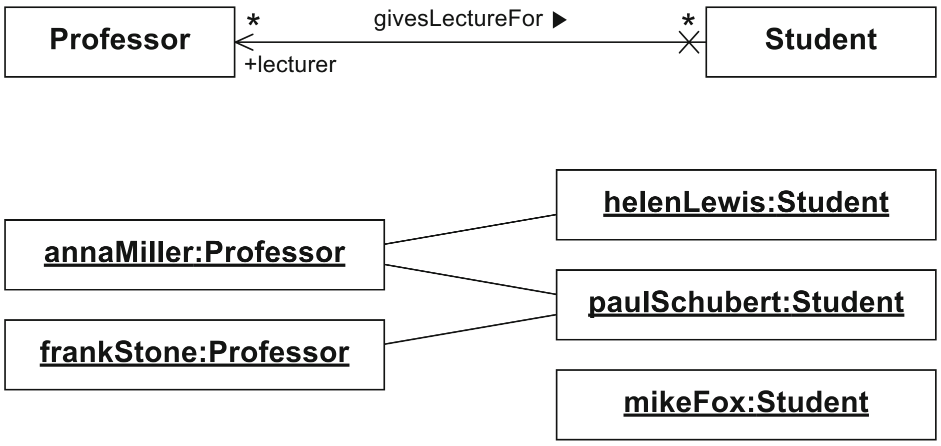
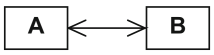


Figure 4.11
Example of a binary association in a class diagram and a valid object diagram

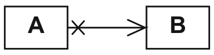
of the association and merely indicates in which direction the reader of the diagram should "read" the association name. We have already seen a binary association with reading direction in Figure 4.11. In this diagram, the reading direction indicates that professors give lectures for students and not the other way around.

Navigability

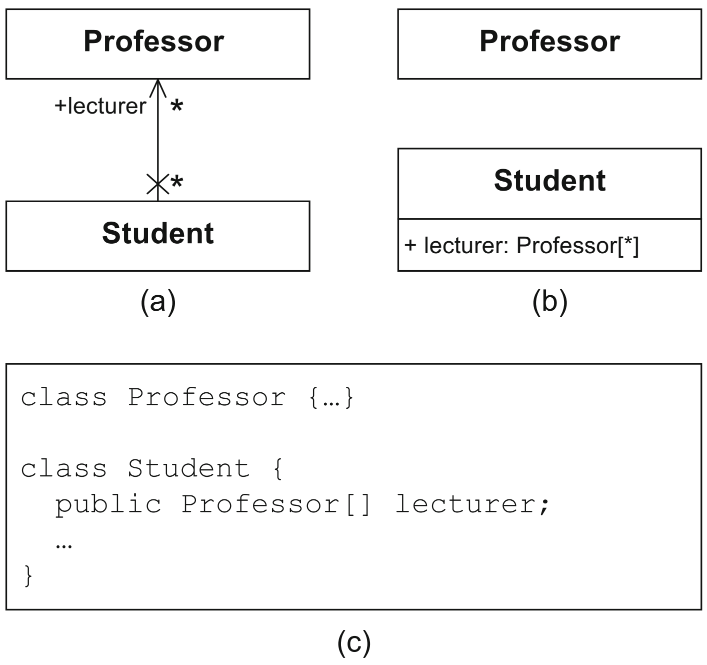


If the edge is directed, that is, at least one of the two ends has an open arrowhead, navigation from an object to its partner object is possible. In simple terms, *navigability* indicates that an object knows its partner objects and can therefore access their visible attributes and operations. The navigation direction has nothing to do with the reading direction, as the example in Figure 4.11 shows. The reading direction indicates that professors give lectures for students. However, the navigability specified indicates that students can access the visible characteristics of professors whose lectures they attend. In contrast, a professor cannot access the visible characteristics of the students who attend the professor's lecture because the professor does not know them.

Non-navigability



A non-navigable association end is indicated by the explicit specification of an X at the association end concerned. For example, if such an X appears at the association end of A for an association between the classes A and B, this means that B cannot access the attributes and operations of A—not even the public ones. Bidirectional edges without arrowheads or X at their ends do not provide any information about the navigation direction but in practice, bidirectional navigability is usually assumed. The navigation direction represents a hint for the subsequent implementation because in object-oriented programming languages, associations are realized as references to the associated objects. An association can also be represented in this way in the class diagram, that is, as an attribute with the appropriate multiplicity, whereby the type of the attribute is the class of the corresponding partner objects. This representation has the same semantics as a navigable association end. Figure 4.12



```
class Professor {…}

class Student {
  public Professor[] lecturer;
  …
}
```

Figure 4.12
Associations in UML and Java

shows (a) a class diagram in which the student-professor relationship is modeled explicitly as an association, (b) a class diagram in which the relationship is represented by an attribute in the class Student, and (c) the translation into Java. The class diagram in Figure 4.12(a) is preferable, as here the relationship between the classes is visualized explicitly and it is visible immediately, while in the alternative in Figure 4.12(b), the association between Student and Professor can only be recognized by reading the type information of the attribute lecturer.

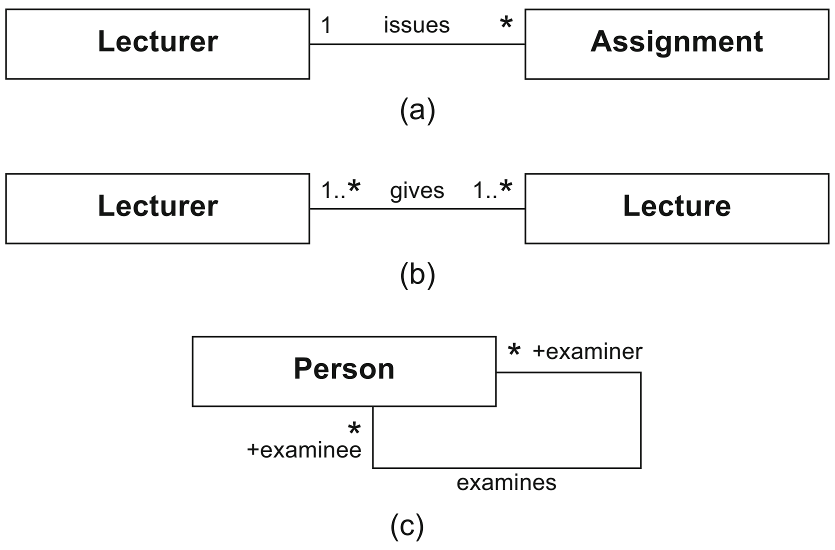


Figure 4.13
Examples of multiplicity specifications in binary associations

In the same way that multiplicities of attributes and parameters are specified, *multiplicities* of associations are given as an interval in the form minimum..maximum. They specify the number of objects that may be associated with exactly one object of the opposite side. The values that the minimum and maximum may adopt are natural numbers and an asterisk $*$, which expresses that there is no restriction. If minimum and maximum are identical, one value and the dots can be omitted. Again, 0..$*$ means the same as $*$. Figure 4.13 shows examples of multiplicity specifications for binary associations. Figure 4.13(a) shows that a lecturer may issue no, one, or multiple assignments and that an assignment is issued by exactly one lecturer. No assignment may exist without an association to a lecturer. Figure 4.13(b) shows that a lecturer gives at least one lecture and a lecture is given by at least one lecturer. Finally, Figure 4.13(c) shows that a person in the role of examiner can examine any number (≥ 0) of persons and a person in the role of examinee can be examined by any number of examiners. In the example in Figure 4.13(c), the model does not exclude the case that persons may examine themselves. If this should be prohibited, additional constraints must be specified.

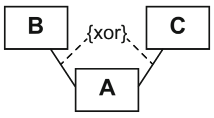
Multiplicity

You may also label the association ends with role names. A *role* describes the way in which an object is involved in an association relationship, that is, what role it plays in the relationship. In the association in Figure 4.13(c), the Person adopts the role of examiner or examinee.

Role

To express that an object of class A is to be associated with an object of class B or an object of class C but not with both, you can specify an *xor constraint* (exclusive or). To indicate that two associations from the same class are mutually exclusive, they can be connected by a dashed line labeled {xor}. For example, an exam can take place either in an office or in a lecture hall but not in both (see Fig. 4.14).

xor constraint



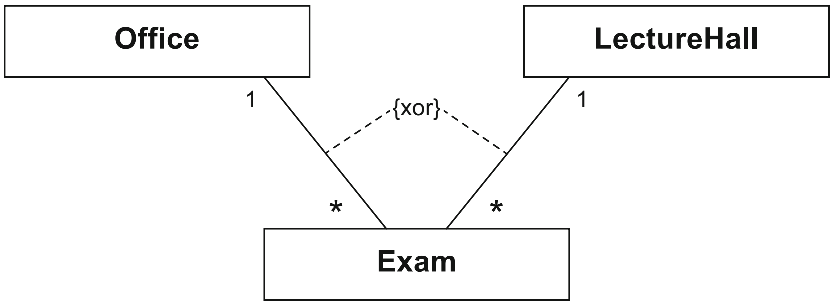
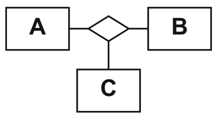


Figure 4.14
Examples of associations with xor constraints

4.3.2 N-Ary Associations

N-ary association

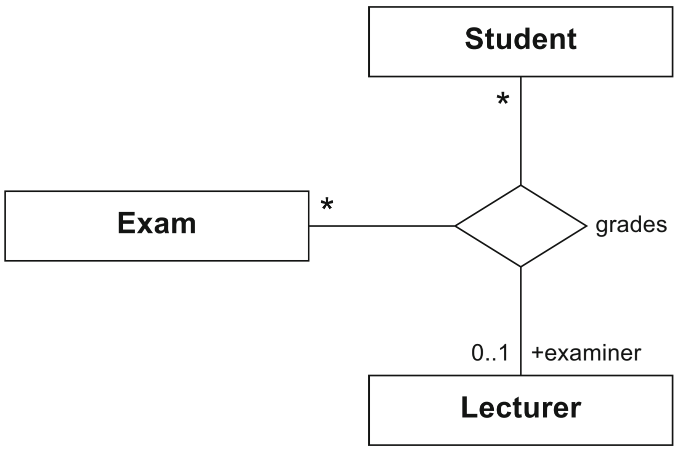


Multiplicities for n-ary associations

If more than two partner objects are involved in a relationship, you can model this using an *n-ary association*. An n-ary association is represented with a hollow diamond in the center. The diamond is connected with all partners of the relationship by means of an undirected edge. The name of the association is specified next to the diamond. There are no navigation directions for n-ary associations; however, multiplicities and role names are possible. Multiplicities define how many objects of a role/class may be assigned to a fixed $(n-1)$-tuple of objects of the other roles/classes.

Figure 4.15 models the relationship grades between the instances of the classes Lecturer, Student, and Exam. The multiplicities are defined as follows: one specific student takes one specific exam with no lecturer (i.e., does not take this exam at all) or with precisely one lecturer. This explains the multiplicity 0..1 for the class Lecturer. One specific exam with one specific lecturer can of course be taken by any number of students and one specific student can be graded by one specific lecturer for any number of exams. In both cases, this is expressed by the multiplicity $*$. In this model, it is not possible that two or more lecturers grade one student for the same exam.

Figure 4.15
Example of n-ary (here ternary) association ...



If you tried to express this ternary association with two binary associations, you would have a model with a different meaning. In the representation shown in Figure 4.16, an exam can be graded by multiple lecturers. The ternary association in Figure 4.15 clearly shows which lecturer a student passed a specific exam with—this is not the case with the diagram shown in Figure 4.16.

For example, with the model shown in Figure 4.15, it is possible to express that student s1 took the exam e1 with lecturer l1 and that student

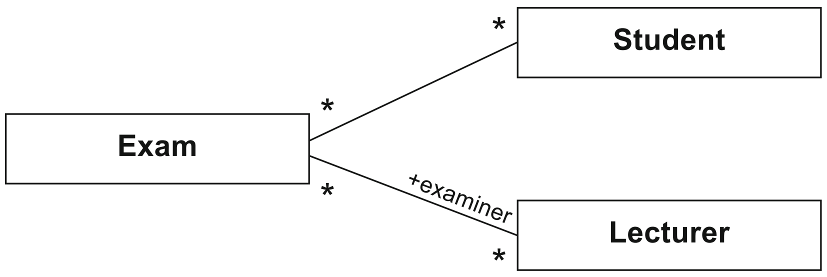


Figure 4.16
... versus example with two binary associations ...

s2 took the same exam e1 with lecturer l2. With the model shown in Figure 4.16, it is only possible to express that the students s1 and s2 took the exam e1 and that exam e1 has two examiners l1 and l2. With this model, you cannot express which lecturer grades which student.

As an alternative to the ternary association in Figure 4.15, an additional class can be introduced which is connected to the original classes via binary associations (see Fig. 4.17). However, in this model it is possible that one student is graded multiple times for one and the same exam what is not possible with the model of Figure 4.15.

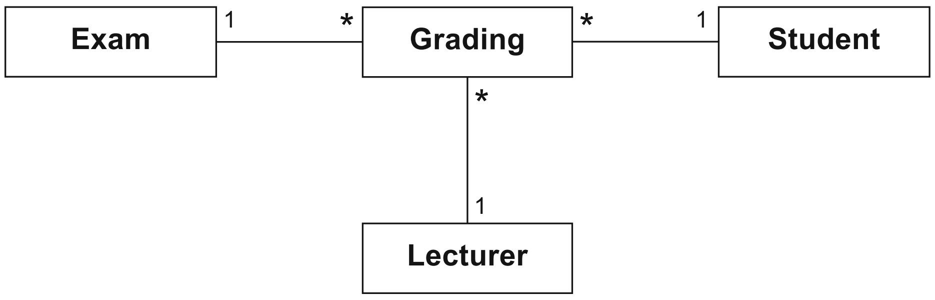


Figure 4.17
... versus example with additional class

4.4 Association Classes

If you want to assign attributes or operations to the relationship between one or more classes rather than to a class itself, you can do this using an *association class*. An association class is represented by a class and an association that are connected by a dashed line. The association can be binary or n-ary. Although the representation includes multiple components, an association class is *one* language construct that has both the properties of a class and the properties of an association. Therefore, in a diagram, the class and association of an association class must have the same name, although you do not have to name both (see the association classes Enrollment and Grade in Fig. 4.18). An association class

Association class

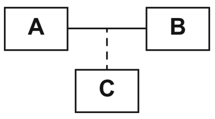
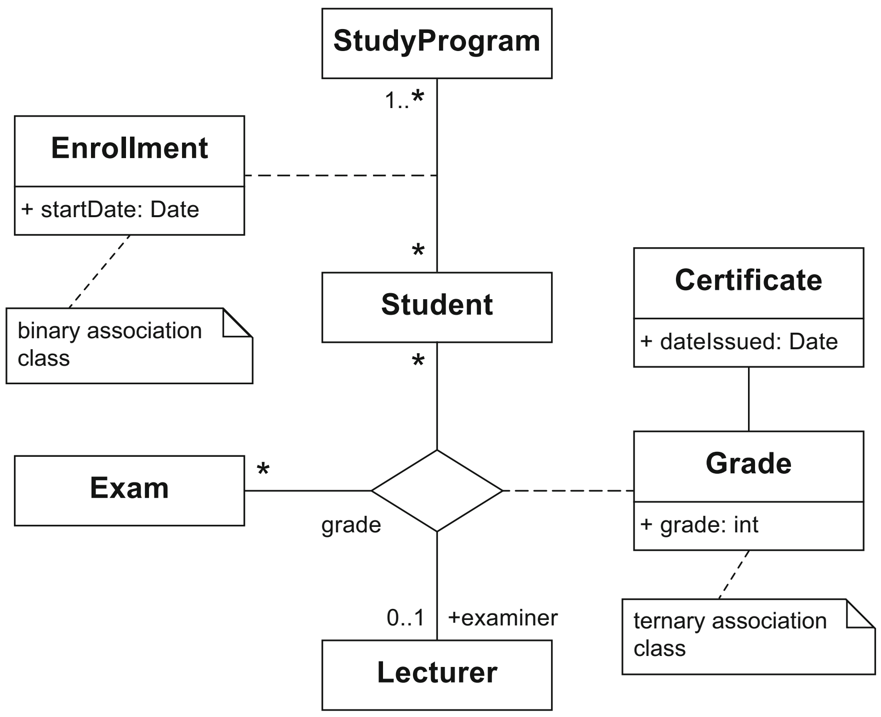


Figure 4.18
Examples of association classes

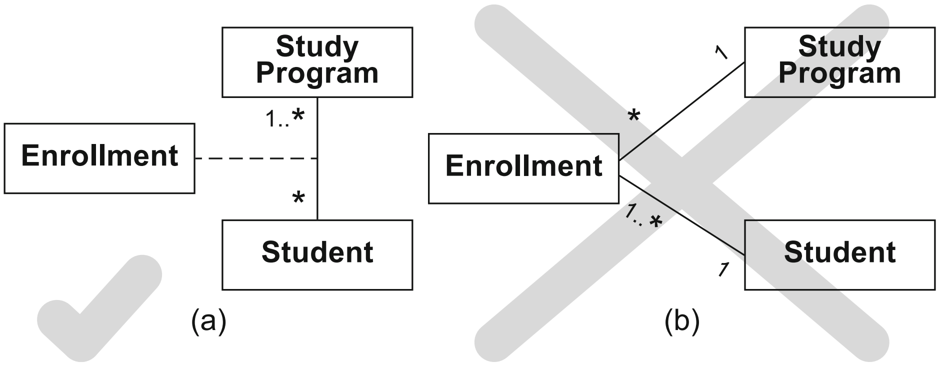


can also have associations with other classes. In Figure 4.18, the association class Grade, which contains information about a student's grade for a specific exam, is associated with the class Certificate.

In general, you cannot replace an association class with a "normal" class which is itself associated with the original two associated classes, as shown by the following example. Let us assume that we want to model that a student enrolls for at least one study program and has precisely one enrollment for each chosen study program. In turn, any number (≥ 0) of students can enroll for one specific study program. This situation is shown in Figure 4.19(a).

Figure 4.19(b) shows the attempt to model this situation with only "normal" classes. An enrollment is assigned to precisely one student

Figure 4.19
Attempt to model an association class with a "normal" class and corresponding relationships



and precisely one study program, while one study program is related to any number of enrollment objects. A student has at least one enrollment. So far the requirements are met. However, if we examine the diagram more closely, we see that in Figure 4.19(b), a student can have multiple enrollments for one and the same study program, which is not the intention. In contrast, in Figure 4.19(a), a student can enroll for a specific study program only once.

If duplicates are explicitly required for an association class, at least one association end must be identified as {non-unique}. If this property is not specified explicitly, the default value {unique} is assumed. In Figure 4.20(a), a student can only be granted an exam meeting to discuss the result of the student's written exam once. Figure 4.20(b) shows a more student-friendly model. There, the use of {non-unique} allows a student to have more than one exam meeting.

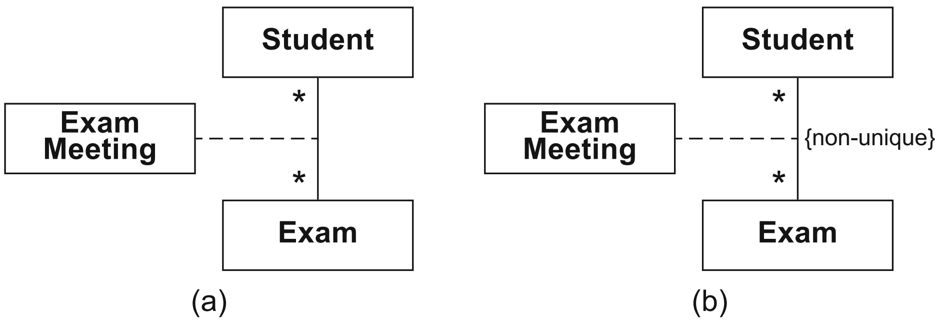


Figure 4.20
Example of {unique} and {non-unique} association ends

4.5 Aggregations

Aggregation

Parts-whole relationship

An aggregation is a special form of association that is used to express that instances of one class are parts of an instance of another class. UML differentiates between two types: *shared aggregation* and *composition*. Both are represented by a diamond at the association end of the class that stands for the "whole". The differentiation between composition and shared aggregation is indicated by a solid diamond for a composition and a hollow diamond for a shared aggregation. Both are transitive and asymmetric associations. In this case, transitivity means that if B is part of A and C is part of B, C is also part of A. Asymmetry expresses that it is not possible for A to be part of B and B to be part of A simultaneously.

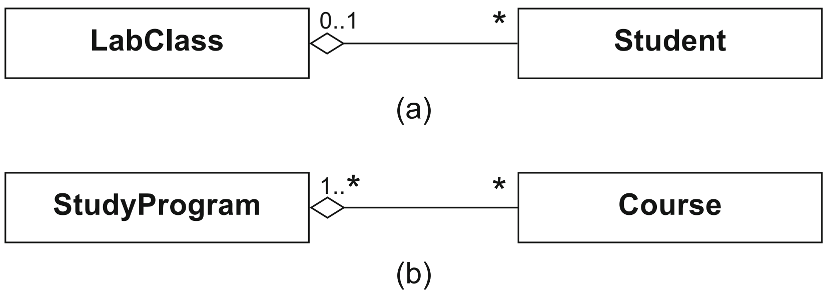
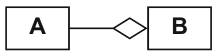


Figure 4.21
Examples of shared aggregations

4.5.1 Shared Aggregations

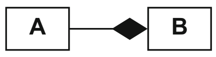
Shared aggregation



In the UML standard, a *shared aggregation* has intentionally informal semantics. In principle, a shared aggregation expresses a weak belonging of the parts to a whole, meaning that parts also exist independently of the whole. The multiplicity at the aggregating end may be greater than 1, meaning that an element can be part of multiple other elements simultaneously. Shared aggregations can therefore span a directed acyclic graph. Figure 4.21 shows two examples of the use of a shared aggregation. In Figure 4.21(a), a lab class consists of any number of students. However, a student can participate in a maximum of one lab class. In Figure 4.21(b), a study program is made up of any (≥ 0) number of courses. A course is assigned to at least one (≥ 1) study program.

4.5.2 Compositions

Composition



The use of a *composition* expresses that a specific part can only be contained in at most one composite object at one specific point in time. This results in a maximum multiplicity of 1 at the aggregating end. The composite objects therefore form a forest of trees, indicating an existence dependency between the composite object and its parts; if the composite object is deleted, its parts are also deleted. Figure 4.22 shows examples of compositions. A lecture hall is part of a building. Due to the multiplicity 1, there is an existence dependency between elements of these two classes. The lecture hall cannot exist without the building. If the building no longer exists, the lecture hall also does not exist anymore. The situation is different for a beamer which is also associated with a lecture hall by a composition. However, the multiplicity 0..1 is specified at the aggregating end. This means that the beamer can exist without the lecture hall, that is, it can be removed from the lecture hall. If the beamer is located in the lecture hall and the lecture hall ceases

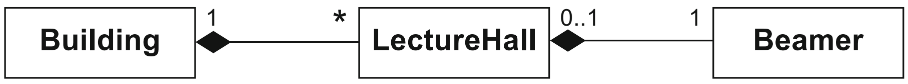


Figure 4.22
Examples of compositions

to exist—for example, because the building is torn down—the beamer also ceases to exist. However, if it was removed from the lecture hall beforehand, it continues to exist.

A shared aggregation is differentiated from an association only by the fact that it explicitly visualizes a "part of" relationship. In a composition, the existence dependency signifies a far stronger bond between the composite object and its parts, which means that a composition and an association are not interchangeable. A composition is usually used if the parts are physically embedded in the composite object or are only visible for the composite object. If the parts are referenced externally, this can indicate that a shared aggregation is sufficient. Furthermore, if the composite object is deleted or copied, its parts are also deleted or copied when a composition is used.

Existence dependency of a composite object's parts

4.6 Generalizations

Different classes often have common characteristics. For example, in Figure 4.23, the classes Student, ResearchAssociate, and AdministrativeEmployee all have the attributes name, address, dob, and ssNo. Students and employees of both types are distinguished by further characteristics specific to the respective class: a student has a matriculation

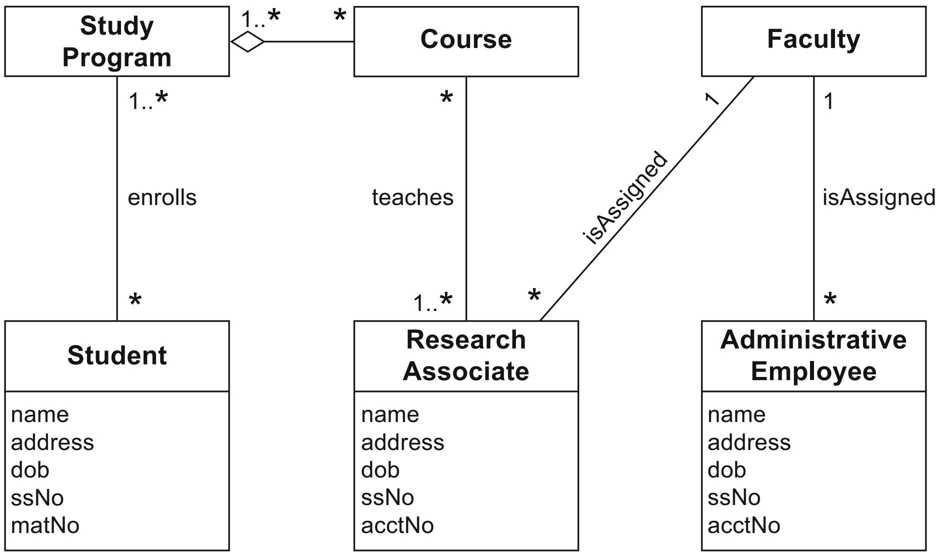


Figure 4.23
Class diagram without generalization

number and has enrolled for at least one study program; employees have a checking account and are assigned to a faculty. Instances of the class ResearchAssociate are in a teaches relationship with any number of instances of the class Course.

We can use a generalization relationship to highlight commonalities between classes, meaning that we no longer have to define these common characteristics multiple times. Conversely, we can use the generalization to derive more specific classes from existing classes. If we want to add a class Professor, which is a subclass of ResearchAssociate, in Figure 4.23, we use the generalization to avoid having to copy the characteristics of the class ResearchAssociate to the class Professor.

4.6.1 Inheritance

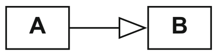
Inheritance from superclass to subclass

Synonyms:

- *Inheritance*
- *Generalization*
- *"Is a" relationship*

The *generalization relationship* expresses that the characteristics (attributes and operations) and associations that are specified for a general class (*superclass*) are passed on to its *subclasses*. Therefore, the generalization relationship is also referred to as *inheritance*. This means that every instance of a subclass is simultaneously an indirect instance of the superclass. The subclass "possesses" all instance attributes and class attributes and all instance operations and class operations of the superclass provided these have not been marked with the visibility private. The subclass may also have further attributes and operations or enter into other relationships independently of its superclass. Accordingly, operations that originate from the subclass or the superclass can be executed directly on the instance of a subclass.

Generalization notation



A generalization relationship is represented by an arrow with a hollow, triangular arrowhead from the subclass to the superclass, for example from Student to Person in Fig. 4.24. The name of a superclass must be selected such that it represents an umbrella term for the names of its subclasses. To ensure that there are no direct instances of the class Person, we label this class with the keyword {abstract}. The class Person therefore becomes an *abstract class* and only its non-abstract subclasses can be instantiated. We will look at details of abstract classes in Section 4.7 on page 72.

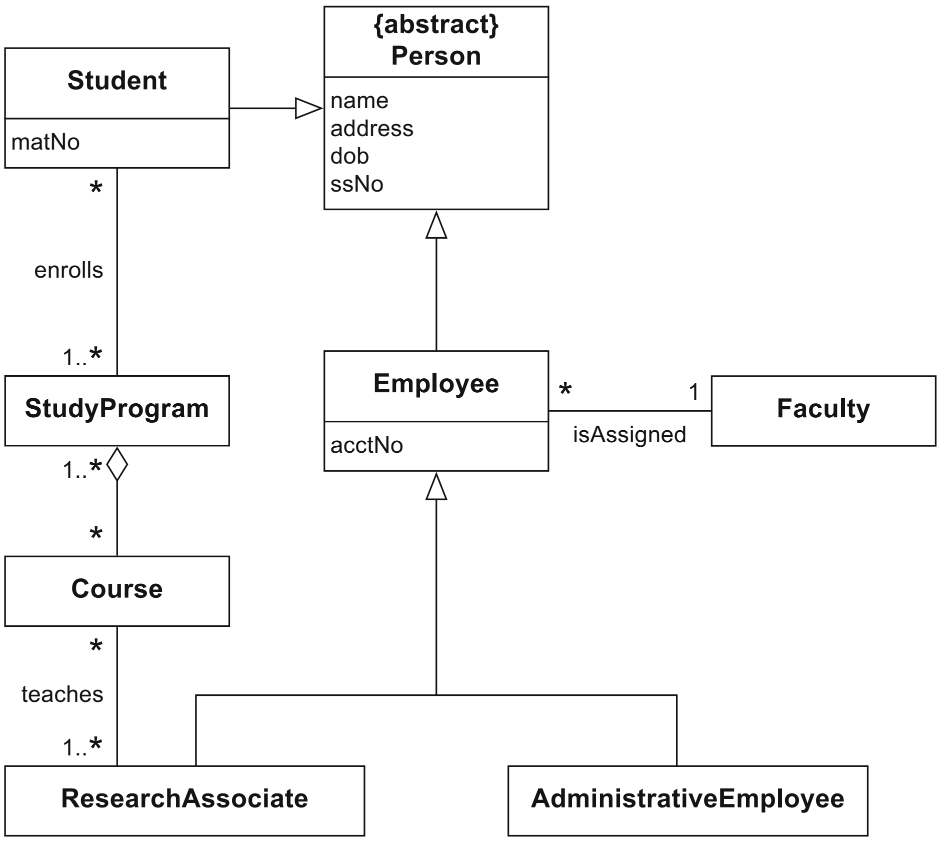


Figure 4.24
Class diagram with generalization

Transitivity of the generalization relationship

The generalization relationship is also referred to as an "is a" relationship. For example, every student is a person (see Fig. 4.24). Every research associate and every administrative employee is an employee and, due to the transitivity of the generalization relationship, every administrative employee is also a person. If, as in object-oriented programming languages, we consider a class to be a type, subclasses and superclasses are equivalent to subtypes and supertypes.

Subtype and supertype equivalent to subclass and superclass

Multiple inheritance

UML allows *multiple inheritance*, meaning that a class may have multiple superclasses. For example, a tutor is both an employee of the university and a student (see Fig. 4.25). Due to the transitivity of inheritance, single inheritance creates an inheritance hierarchy, whereas multiple inheritance creates a (directed acyclic) inheritance graph.

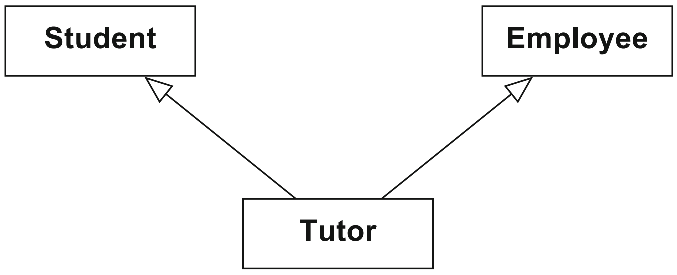


Figure 4.25
Example of multiple inheritance

4.6.2 *Classification*

Multiple classification

Classification refers to the “instanceOf” relationship between an object and its class. In many object-oriented programming languages, an object can usually only be the direct instance of precisely one class. In contrast, UML allows *multiple classification*. With multiple classification, an object can be an instance of multiple classes without these classes having to be associated with one another in an inheritance relationship. In contrast to multiple inheritance, no new class inheriting the characteristics of the superclasses involved is introduced.

Generalization set

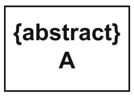
For example, instances of Employee can be differentiated according to their job, that is, whether they are researchers or administrators, and whether they are financed directly via the university or via a project. Multiple classification means that an object can be an instance of multiple classes whose characteristics the object then has. In Figure 4.26, we have divided the generalization relationships into two groups. The sets Job and Financing form *generalization sets* which group subclasses according to multiple independent criteria. Generalization sets can be described more precisely by the following constraints:

- *Overlapping* or *disjoint*: in an overlapping generalization set, an object may be an instance of multiple subclasses simultaneously. In a disjoint generalization set, an object may be an instance of a maximum of one subclass.
- *Complete* or *incomplete*: in a complete generalization set, each instance of the superclass must be an instance of at least one of the subclasses. In incomplete generalization sets, this is not necessary.

This results in four combinations: {complete, overlapping}, {incomplete, overlapping}, {complete, disjoint}, and {incomplete, disjoint}. If none of these constraints are specified explicitly, {incomplete, disjoint} is the default value. Examples are shown in Figure 4.26: an employee must belong to either the research or administrative personnel but not both. The employee can be financed directly via the university, via a project, via both, or in another, unspecified way, for example via a scholarship.

4.7 Abstract Classes vs. Interfaces

Abstract class



Classes that cannot be instantiated themselves are modeled as *abstract classes*. These are classes for which there are no objects—only their subclasses can be instantiated. Abstract classes are used exclusively to highlight common characteristics of their subclasses and are therefore

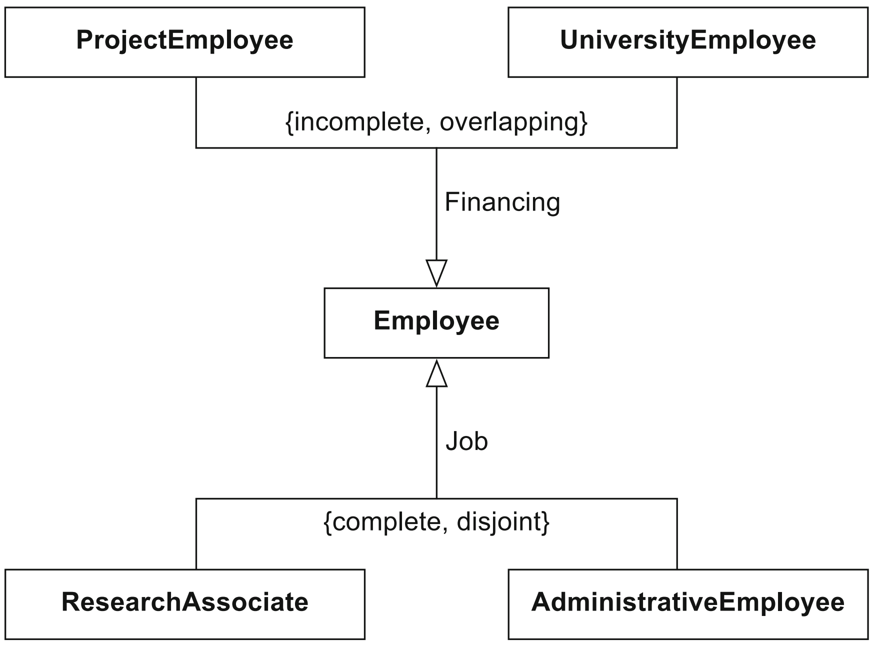


Figure 4.26
Example of multiple classification

only useful in the context of generalization relationships. Operations of abstract classes can also be labeled as abstract. An *abstract operation* does not offer any implementation itself. However, it requires an implementation in the concrete subclasses. Operations that are not abstract pass on their behavior to all subclasses.

Abstract operation

Abstract classes and abstract operations are either written in italic font or indicated by the specification of the keyword {abstract} before their name (see Fig. 4.27). In manually produced class diagrams in particular, the use of the second notation alternative is recommended, as italic handwriting is difficult to recognize.

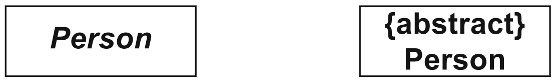


Figure 4.27
Notation for abstract classes

In the example in Figure 4.28, the class Person is abstract. Hence, there cannot be any instances of Person itself but there can be instances of the specific subclasses Employee and Student.

Similarly to the abstract class, an *interface* also does not have an implementation or any direct instances. An interface represents a contract. The classes that enter into this contract, that is, the classes that implement the interface, obligate themselves to provide the behavior specified by the interface. In contrast to the relationship between an abstract class and its subclasses, an “is a” relationship between an interface and the classes that implement it is not necessary. Operations of interfaces never have an implementation.

Interface

An interface is denoted like a class but with the additional keyword «interface» before the name. A dashed inheritance arrow with a hollow, triangular arrowhead from a class to an interface signifies that this class implements the interface. A dashed arrow with an open head with the keyword «use» expresses that a class uses an interface. Let us look at the example from Figure 4.28. The classes Person and Course implement the interface Printable. The classes that implement Printable must provide an operation print(). This operation is different for every class. For a course, the name and the number of hours are printed; for a Person, the name and address are printed. In the class Student, the operation print() is specified again. This expresses that the Student extends the behavior of the operation print() inherited from Person. The method print() is overwritten, meaning that the matriculation number is also printed. For Employee this is not necessary, assuming that the behavior specified for print() in Person is sufficient. The class Printer can now process each class that implements the interface Printable. Thus, a specific print() can be realized for each class and the class Printer remains unchanged.

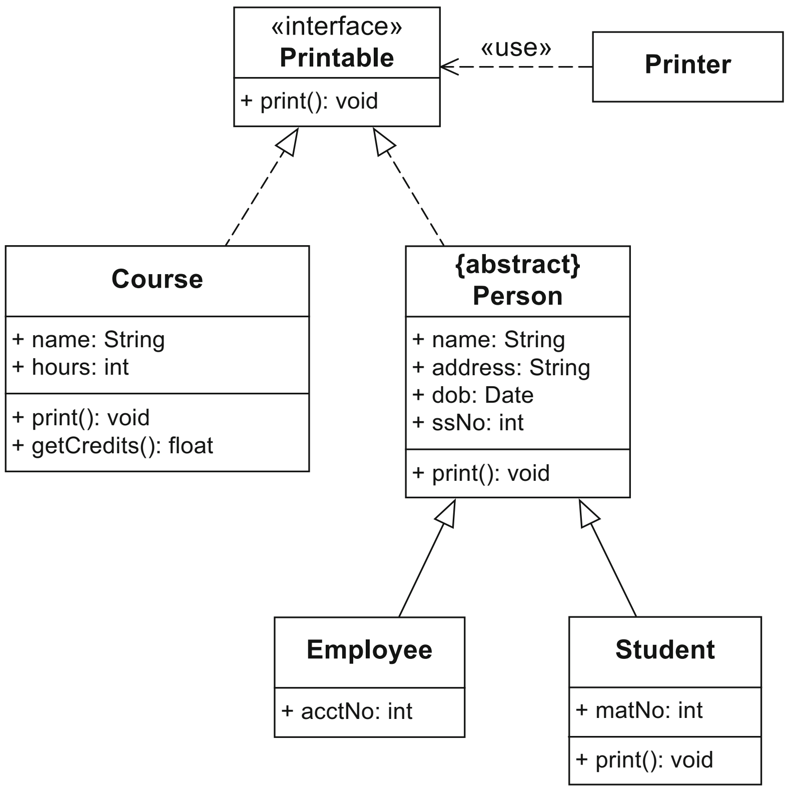


Figure 4.28
Example of an interface

4.8 Data Types

Class vs. data type

Attributes, parameters, and return values of operations have a type that specifies which concrete forms they may take. For example, the name of a person has the type String. A type can be either a class or a *data type*. Instances of data types are also referred to as their *values*. In contrast to instances of classes (objects), values do not have their own identity. If two values are identical, they cannot be differentiated. For example, let us look at the class Book, whose instances are different copies of the book *UML@Classroom*. These copies can be uniquely identified and differentiated even though their attributes have the same content. However, different occurrences of a value, for example the number 2, cannot be differentiated. This differentiation becomes evident in the application of the comparison operation ==, as provided by Java for example. If we compare two variables of the type `int` (integer data type) and both variables have the same value, the result of the comparison operation is true. If we compare two different objects with ==, the result is false in general even if all attributes have the same values.

Data type

In UML, a data type is visualized in the same way as a class, with the difference that the name of the data type is annotated with the additional keyword «datatype» (see Fig. 4.29(b)). As the example in Figure 4.29(b) shows, data types can have an internal structure in the form of attributes. In UML, there are also two special forms of data types, namely primitive data types and enumerations.

Primitive data type

Primitive data types do not have any internal structure. In UML there are four pre-defined primitive data types: Boolean, Integer, UnlimitedNatural, and String. User-defined primitive data types are identified by the specification of the keyword «primitive». Primitive data types may have operations (see Fig. 4.29(a)) that are executed on their values.

Enumeration

Enumerations are data types whose values are defined in a list. The notation is the same as for a class with the specific identification «enumeration». In Figure 4.29(c), the enumeration AcademicDegree is defined. This enumeration lists all academic degrees that are known in our system. Therefore, attributes of the type AcademicDegree may take the values bachelor, master, and phd. These values are called *literals*.

Literal

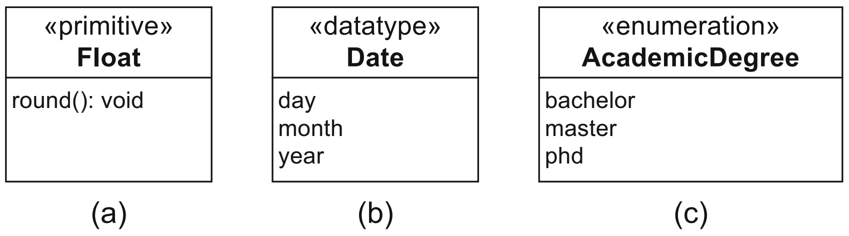


Figure 4.29
Examples of data types

User-defined types are used as specified in the syntax description of attributes and operations in Figure 4.5 (page 54) and in Figure 4.8 (page 57). Let us look at the type definitions from Figure 4.29 again. These could be used in the following attribute definitions: weight: Float, dob: Date, and title: AcademicDegree [∗].

4.9 Creating a Class Diagram

UML describes the syntax and semantics of classes and their relationships but not how the classes and relationships are constructed. Unfortunately, it is not possible in principle to completely extract classes and their characteristics from a natural language text automatically. However, there are guidelines for creating a class diagram. Nouns such as person, employee, course, etc. often indicate classes. In contrast, names of values such as *Paul* or *object-oriented modeling* and expressions that indicate the relationships between potential classes are rarely classes. Values of attributes are often expressed by adjectives or also by nouns and operations often result from verbs. The following three aspects are important: which operations can an object of a class execute? Which events, to which the object must be able to react, can theoretically occur? And finally, which other events occur as a result? If the values of an attribute can be derived from another attribute, for example, if the age of a person can be calculated from their date of birth, it should be identified as a derived attribute. Further, it is essential to consider not only the current requirements but also the extensibility of the system.

As we now know how to derive a class diagram from a textual specification, we will do so for the following requirement specification:

Information system of a university

- A university consists of multiple faculties which are composed of various institutes. Each faculty and each institute has a name. An address is known for each institute.
- Each faculty is led by a dean, who is an employee of the university.
- The total number of employees is known. Employees have a social security number, a name, and an e-mail address. There is a distinction between research and administrative personnel.
- Research associates are assigned to at least one institute. The field of study of each research associate is known. Furthermore, research associates can be involved in projects for a certain number of hours, and the name, starting date, and end date of the projects are known. Some research associates teach courses. They are called lecturers.
- Courses have a unique number (ID), a name, and a weekly duration in hours.

1. *Identifying the classes*

   First, we must identify the elements that occur in the system University that identify the classes. These are shown in Figure 4.30.

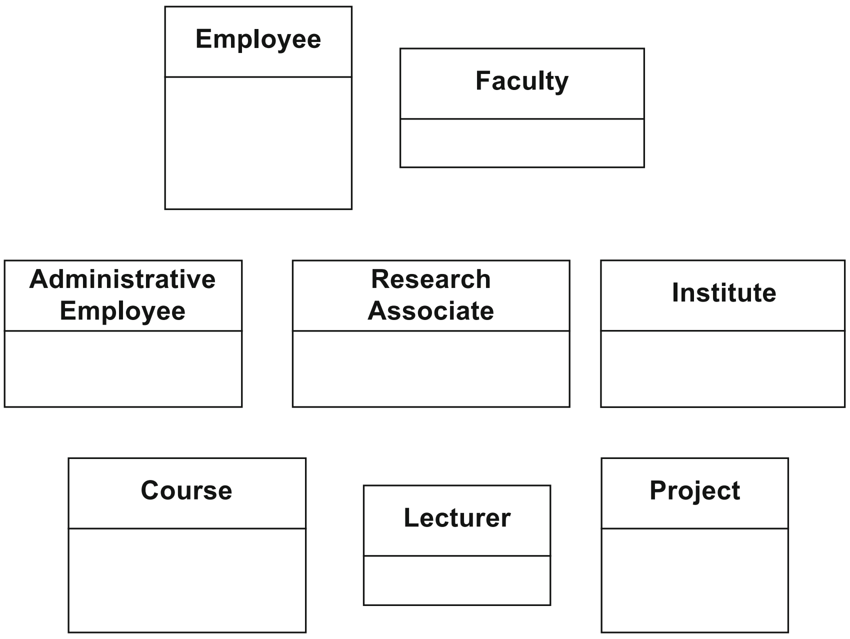


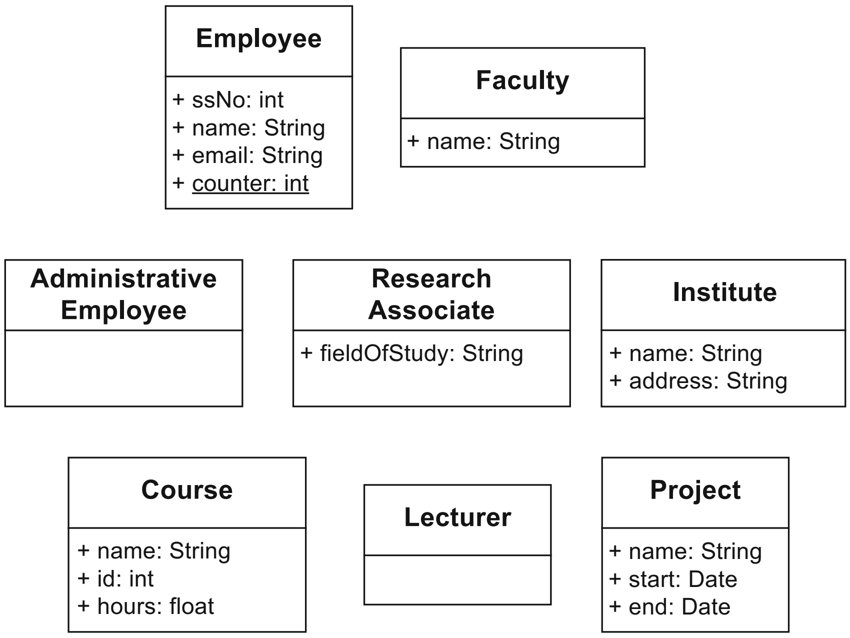
Figure 4.30
Identified classes

   As we can see, University is not a separate class. We have not forgotten it—we have intentionally not included it. We are using the class diagram to describe the system University, hence the instance of our model contains those objects that occur within a university, for example, the Vienna University of Technology. If we included a class University which itself consists of other classes from Figure 4.30, we could model multiple university information systems simultaneously. Our model would then also describe, for example, the Johannes Kepler University Linz.

2. *Identifying the attributes*

   We can now describe our classes in more detail using attributes. The classes and their attributes are shown in Figure 4.31.
   We have defined meaningful data types for our attributes even though these are not included in the specification. We also set the visibility of all attributes to public so that in this phase, we do not have to think about which attributes are visible from the outside and which are not. The attribute counter of the class Employee is defined as a class attribute as its values do not belong to an instance. This attribute is increased when an instance of the class Employee is created.

Figure 4.31
Classes and their attributes



3. *Identifying the relationships between classes*

   Classes can be linked with one another in three ways. They can be in a sub-/superclass relationship (generalization), be related by means of an aggregation, or linked via associations.

4.9.1 Generalizations

The following sentences strongly indicate a generalization relationship: "There is a distinction between research and administrative personnel." and "Some research associates teach courses. Then they are called lecturers." We model these generalization relationships as shown in Figure 4.32. As every employee of a university belongs to either the research or administrative personnel, we can set the class Employee to abstract.

4.9.2 Associations and Aggregations

To complete the class diagram, we need to add the associations and aggregations and their corresponding multiplicities. The classes Lecturer and Course are linked by means of the association teaches. An employee leads the faculty. Here the employee takes the role of a dean. A faculty

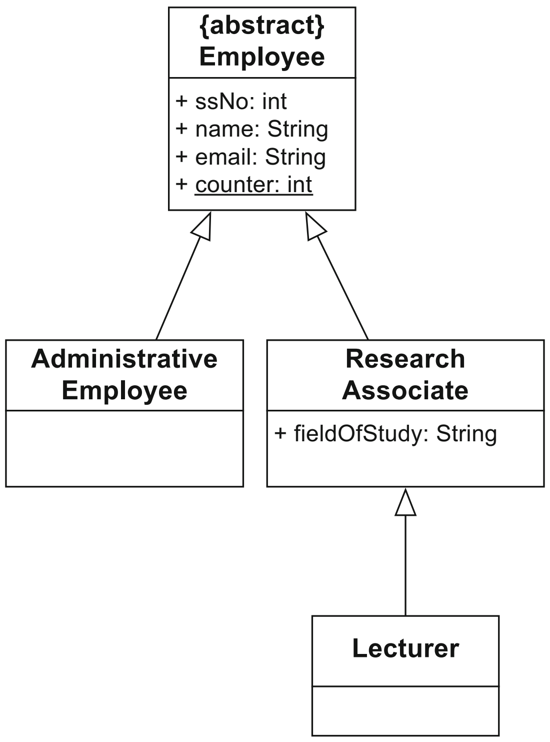


Figure 4.32
Identified generalization relationships

consists of multiple institutes. We assume that there is an existence dependency which we model with a composition. Research associates are assigned to an institute, meaning they are part of an institute. Using a composition here would be incorrect as there is no existence dependency between instances of Employee and Institute. However, a shared aggregation is possible in order to represent the parts-whole relationship explicitly. Finally, we have the involvement of research associates in projects, whereby we know the number of hours of participation. For this we need the association class Participation. This association class further details the relationship between the project and the research associate with the number of hours. Figure 4.33 shows the complete class diagram for the given task.

Note that the resulting model is not unique even for such small examples; it depends on the one hand on the intended application, and on the other hand on the style of the modeler. For example, if we had created the model with the intention of generating code from it, we would perhaps have designed the interfaces more carefully and specified more differentiated visibilities. It is a matter of taste that Lecturer is a separate class but dean is a role. We could also have specified Lecturer as a role at the end of the association teaches which would have been defined between the classes ResearchAssociate and Course.

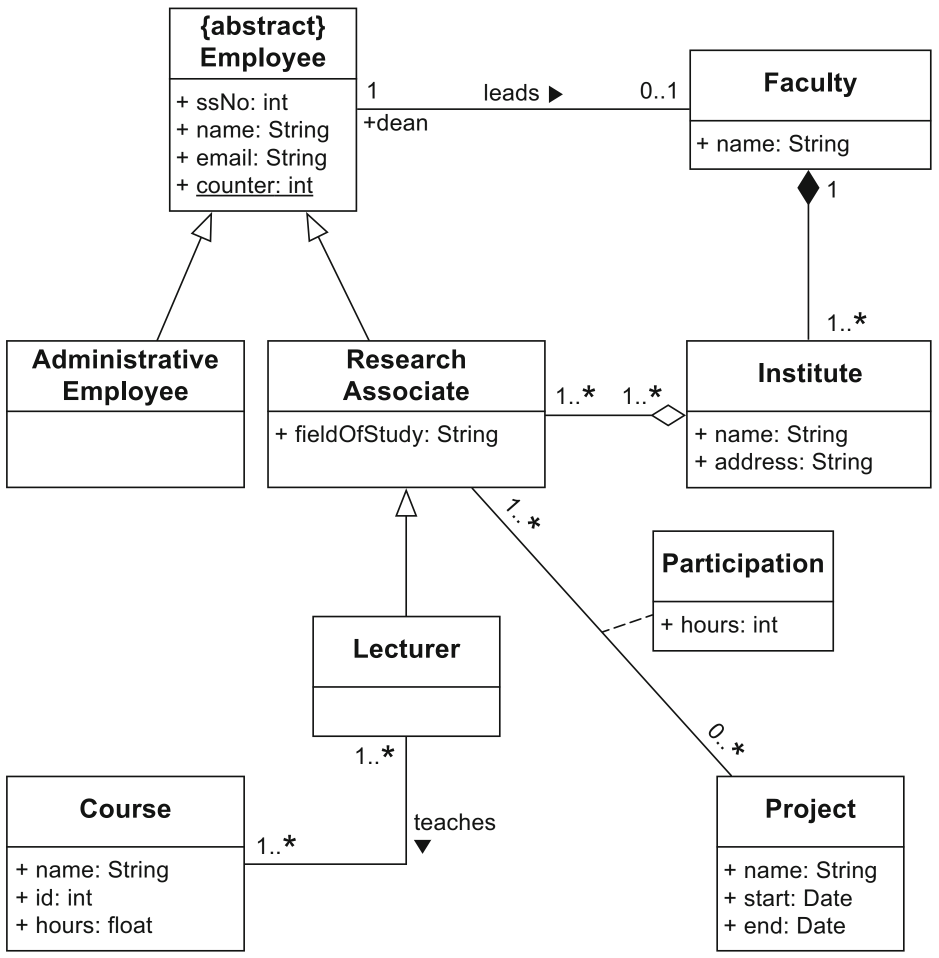


Figure 4.33
Class diagram of the information system of a university

4.10 Code Generation

Forward engineering

Class diagrams are often created with the intention of implementing the modeled elements in an object-oriented programming language. As many of the concepts of the class diagram are available in identical or similar form in object-oriented programming languages such as Java, C#, or C++, in many cases a translation can take place automatically and requires only minimal manual intervention. The class diagram is also suitable for documenting existing program code, with the advantage that the relationships between classes are represented graphically. There are a number of tools for reverse engineering program code into class diagrams automatically.

Reverse engineering

Data modeling also involves similar concepts to those of the class diagram. For example, here the entity-relationship diagram (ER diagram) [14] is used—with the exception of different notation, it is very similar to the class diagram. Both diagrams show the elements (classes or entities) of a system and the relationships (associations or relations) between them. In both cases, these elements are characterized by their attributes. Considerable differences are visible if we compare the focus of the two types of diagrams. While the ER diagram describes the elements of a database, the class diagram shows how to implement the modeled system in an object-oriented programming language. Thus, in the ER diagram, we can define key attributes that are required to identify entries in a table. This is not possible directly in a class diagram but it is also not necessary, as each object is identified by a unique object ID. In contrast, the specification of behavior, which is possible in the class diagram through operations, is not supported in the ER diagram. Therefore, the recommendation is to use the diagram type that is best for the problem in question. The following example again illustrates the connection between a class diagram (see Fig. 4.34) and the Java code generated from it (see Fig. 4.35).

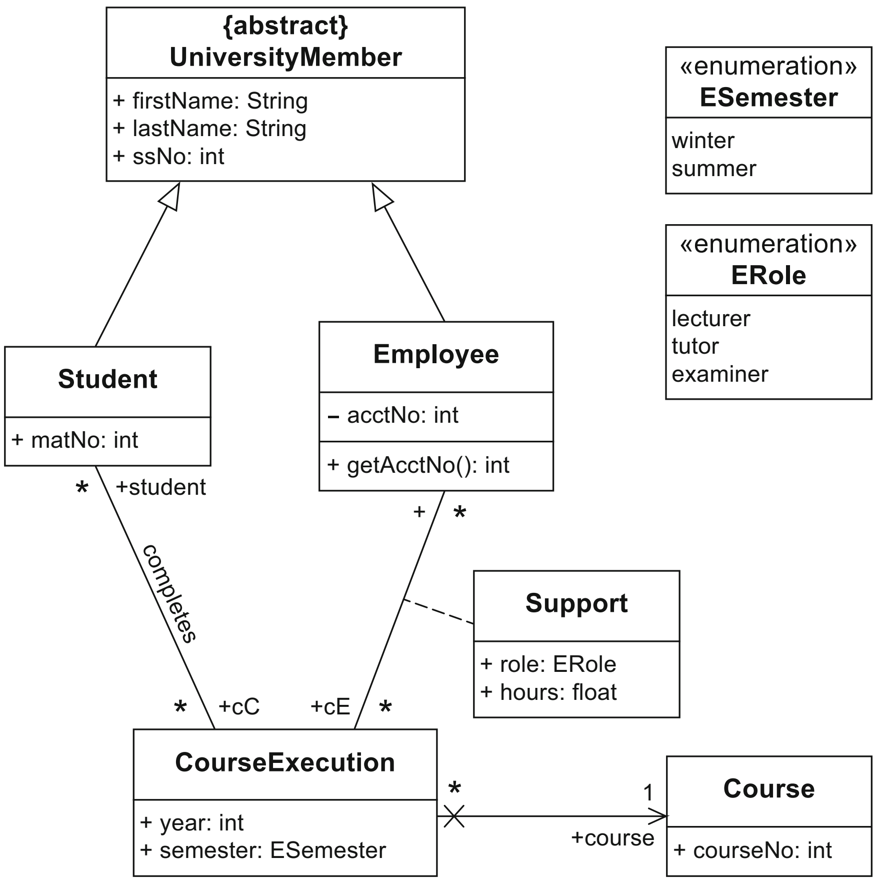


Figure 4.34
Class diagram from which code is to be generated

Many elements can be translated 1:1. Both abstract and concrete classes are adopted directly in the code with their attributes and operations. In the code, associations are represented as attributes. Note that the multiplicity of an association end is reflected in the type of the attribute. If the multiplicity is greater than one, we can use, for example, an array, as we did for the courses. Instead of arrays we could also use generic data types, for example the Java data type `Collection`; in contrast to arrays, with generic data types we do not have to know the size at initialization [4].

We have to make sure that we implement the navigation directions correctly. The navigation information provided in terms of arrowheads at the association ends tells us which class has to know about which other class—and this is realized via the attributes that model the association ends.

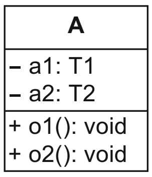
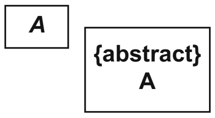
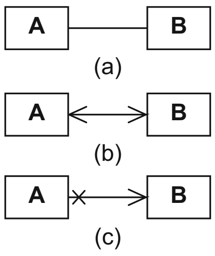
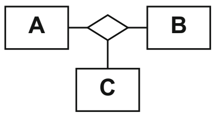
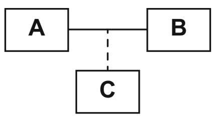
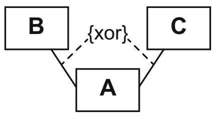
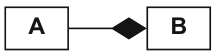
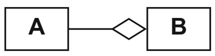
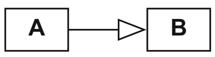
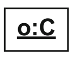
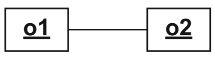
Some concepts, such as association classes or n-ary associations, do not exist directly in common programming languages such as Java. We thus have to consider how to simulate these concepts. Our example contains the association class Support. In the code this is implemented as a hash table. A hash table is a data structure that contains elements in the form (key, data). If the key (which must be unique) is known, the related data can be found efficiently.

Up to this point we have been able to describe the structure of elements and their relationships. We were not able to express behavior. In the above example we had only one operation, getAcctNo(), which returns the account number of the employee. The content of the method body was generated automatically as it is a getter method that encapsulates the access to a variable of the same name. For other operations, for example, operations that were intended to calculate something, the implementation cannot be derived automatically. UML offers other diagrams for modeling behavior and we will introduce these in the following chapters. To complete this chapter, Table 4.2 summarizes the most important concepts of the class and object diagrams.

Figure 4.35
Java code that can be generated automatically from Fig. 4.34

```
abstract class UniversityMember {
        public String firstName;
        public String lastName;
        public int ssNo;
}

class Student extends UniversityMember {
        public int matNo;
        public CourseExecution [] cC; // completed c.
}

class Employee extends UniversityMember {
        private int acctNo;
        public CourseExecution [] cE; // supported c.
        public int getAcctNo { return acctNo; }
}
class CourseExecution {
        public int year;
        public ESemester semester;
        public Student [] student;
        public Course course;
        public Hashtable support;
                // Key: employee
                // Value: (role, hours)
}

class Course {
        public int courseNo;
}

Enumeration ESemester {
        winter;
        summer;
}

Enumeration ERole {
        lecturer;
        tutor;
        examiner;
}
```

Table 4.2
Notation elements of the class and object diagrams

Name	*Notation*	*Description*
Class		Description of the structure and behavior of a set of objects
Abstract class		Class that cannot be instantiated
Association		Relationship between classes: navigability unspecified (a), navigable in both directions (b), not navigable in one direction (c)
N-ary association		Relationship between *N* (in this case 3) classes
Association class		More detailed description of an association
xor relationship		An object of A is in a relationship with an object of B or with an object of C but not with both
Strong aggregation = composition		Existence-dependent parts-whole relationship (A is part of B; if B is deleted, related instances of A are also deleted)
Shared aggregation		Parts-whole relationship (A is part of B; if B is deleted, related instances of A need not be deleted)
Generalization		Inheritance relationship (A inherits from B)
Object		Instance of a class
Link		Relationship between objects