Chapter 4

Class diagrams

Class diagrams support the specification of the concept of classes known from object-oriented programming. The objective of class diagrams is to specify classes and their operations as well as the dependencies between them.

4.1 Learning objectives of this chapter

The learning objectives of this chapter are the followings. At the end of this chapter, you shall

- know what is a UML class diagram;
- be capable of creating detailed UML class diagrams.

4.2 Basics of Object Oriented Programming: Classes

4.2.1 basic notions

Classes are units of encapsulation which decompose the responsibilities of a system and its global system state. Each class represents an abstract notion of state and behaviour. Classes can be instantiated into *objects*. Apart from the state and behaviour specified by the class definition, each object has a unique identity.

The definition of a class comprises three main parts:

1. A *name* - keep it short and meaningful
2. Zero or more *attributes*. Attributes are parts of state with a given name and a type representing the domain of its values. Attributes may be given an initial value from their domain.
3. Zero or more *operations*. Operations specify the behaviour of the class. Each operation has a given name and a list of parameters. Each parameter has a name and a type. Operations may also have a return value of a given type.

Figure 4.1 shows an example of a simple class. The class is named 'IWLayerInfo-Event' and has a number of attributes and one operation.

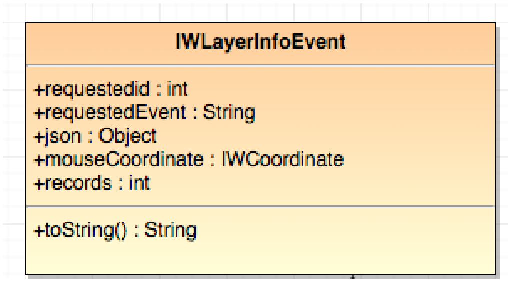

Figure 4.1: A simple class.

4.2.2 Multiplicity

Attributes and associations (see later) can have a *multiplicity*. A multiplicity is a range of the form $a..b$ where $a \in \mathbb{N}$ and $b \in \mathbb{N} \cup \{*\}$. Symbol $*$ represents any arbitrary natural number greater than or equal to a. Multiplicity $a..a$ is usually denoted by a and $0..*$ – any number – is denoted by $*$.

4.2.3 Visibility

Information hiding and encapsulation are key concepts in managing the complexity of software specifications. They imply that the details of the functionality and the state should be hidden from the outside world, and interfaces should provide a stable access to the functionality that a class provides. This is partially achieved by defining visibility ranges for attributes or operations of a class.

Private An operation or attribute is called *private* when it cannot be accessed by any method beyond the class boundaries, that is, by operations of other classes or global statements of the program. Private visibility is denoted by a *minus sign* before the name of the operation of attribute.

Public An operation or attribute is called *public* when the operation or attribute is public, that is, it can be accessed outside the class boundaries. Public visibility is denoted by a *plus sign* before the name of the operation or attribute. It is against information hiding to define public attributes. Instead, to access attributes, one may define public *set* and *get* methods, which check the validity of values before assigning them to private attributes. We omit the set and set methods from class diagrams. We assume their existence, though. If exceptionally set and get methods contain particular functionality beyond checks and copies, we will mention them.

Protected An operation or attribute is called *protected* when it is visible only to the members of the class and those of the class inheriting from it.

4.3 Relationships

There are three types of relationships between classes: dependency, association, and generalisation.

4.3.1 Dependency

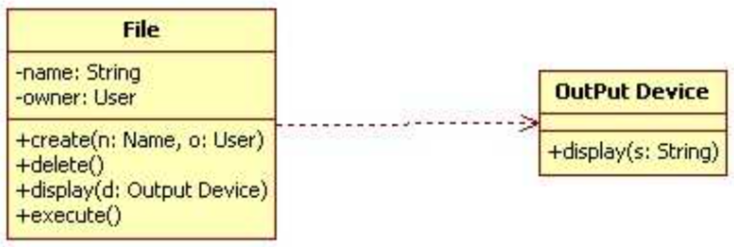

Figure 4.2: A dependency relation between File and Output Device.

A dependency relationship expresses the fact that a class needs another class to realise some of its behaviour. A dependency is denoted by a dotted-line arrow. It is a directed arrow with a dotted arrow-head. Figure 4.2 shows a dependency relation.

4.3.2 Association

An *association* relationship means that objects of one class have a reference to objects of another class. Thus, an association can be seen as an attribute of the source class, of which the type is the target class. Associations are intrinsically directed, showing which class can *navigate* – that is, has a reference – to the other class. Associations with the same source and target class are also allowed. Such associations represent references to the objects of the same type, for instance, a linked list. Associations are represented by solid directed lines. If there is an association in both directions, then the two-way association is represented by a solid line with either arrow-heads at both ends or no arrow-heads at all. The latter is the preferred notation. The name of the reference is put above the line at the target side; this name is also called the *role*. To denote the number of objects participating in each association, one can put a multiplicity below the line at each side.

For some associations, extra information need to be stored, for instance, for a person who is also a member of a library, a member-identifier needs to be stored. Such information and its corresponding operations are gathered in an *association class* and the class is attached to the corresponding association relation.

There are two special kinds of associations:

1. aggregation: an aggregation represents the whole/part relationship. In an aggregation relation, the part – the target of the association – may continue to exist even of the whole – the source of the association – is destroyed. It is represented like an association with an empty diamond at the source – whole – side.

2. composition: a composition relation is a stronger type of association in which the whole creates the parts and upon its destruction, destroys them. Composition is represented just like an aggregation in which the diamond at the source – whole – is filled.

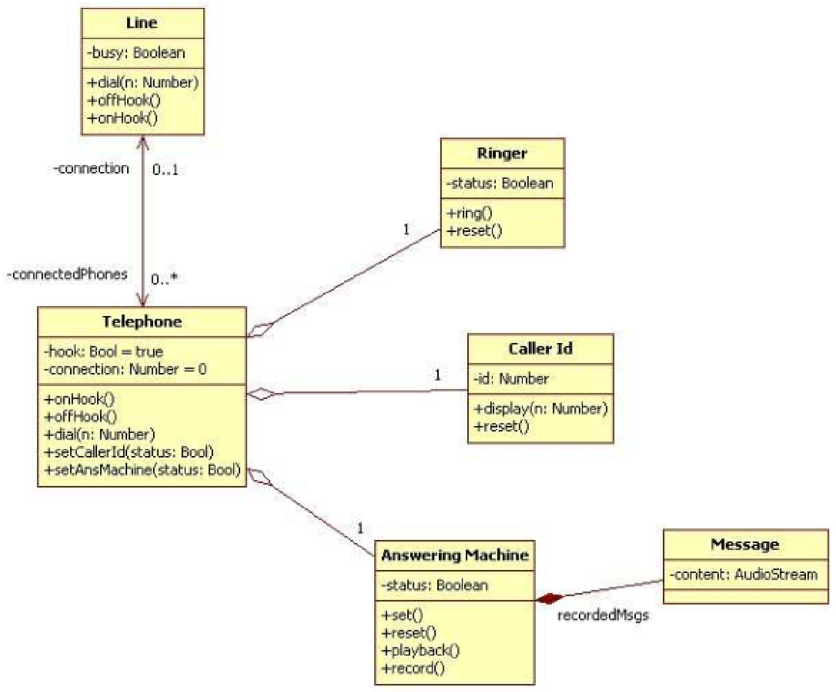

Figure 4.3: Association relations in the telephone system.

Consider the telephone system described in Section 3.2. The telephone set has a number of parts: ringer, answering machine, and caller ID display. The answering machine, in turn, may contain a number of recorded messages. The relationship among the telephone set and its parts is an *aggregation* relation, since the parts are made and can be destroyed independently of the telephone set. The relation between the answering machine and its recorded messages is a *composition* relation, since the messages are created and destroyed by the machine. The messages cannot exist without the machine. The telephone set is connected to the line, by which it can dial other lines. The line can also signal the telephone set in order to notify an incoming call. The relation between the line and telephone set is thus a two-way association. An excerpt of the class diagram of this system in depicted in Figure 4.3.

4.3.3 Generalisation

Generalisation is used when the *source* of the relation inherits all the state and behavioural specification of the*target*. The source of the generalisation – called *sub-class* – can replace the target – called *super-class* – in all its relationships, but not vice versa. Generalisation is also called *inheritance*. Generalisation is denoted by a directed line, of which the arrow-head is a triangle. Cyclic, hence self-loop, generalisation relations are not allowed.

A class may be generalised by more than one class, that is, it can be the source of more than one generalisation relation. This phenomenon is called *multiple inheritance*. It is a rather involved concept and one can also represent a similar design by using two generalisations and one aggregation, thus including the two specialised classes in one class.

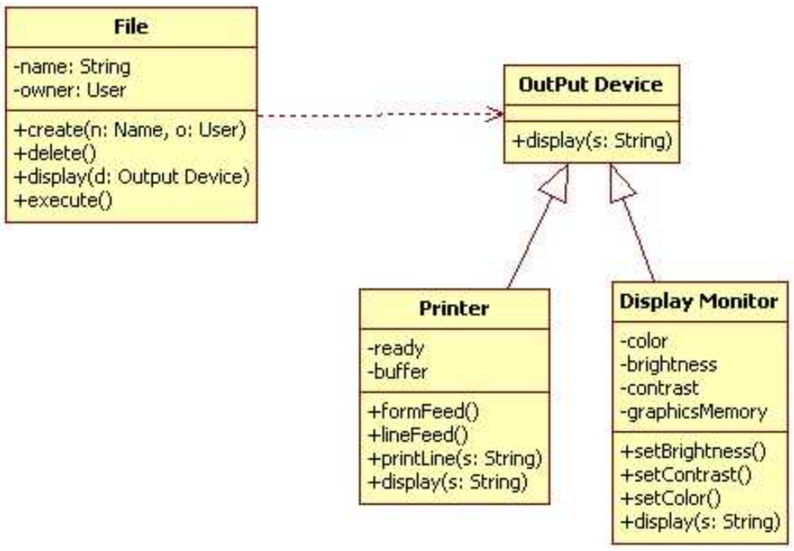

Figure 4.4: Generalisation relations in the File System.

Consider the file system example. In this example, there is a number of output devices on which the display command should be able to write. This includes a display monitor and a printer. This information is captured by the diagram depicted in Figure 4.4.

4.4 Conclusion

Class diagrams support the specification of the structure of a system in terms of classes and their relationships. The next chapter defines sequence diagrams. These diagrams allow to graphically specify interactions between users, classes and objects. It somehow gives a graphical representation of the interactions sketched in use cases.

4.5 Exercises

Exercise 4.5.1. Consider the specification of the UML-Bookshop in Exercise 3.7.1. Identify classes, attributes and operations in each use-case and draw the class diagram for this system.

Exercise 4.5.2. Consider the specification of the transport company in Exercise 3.7.2. Identify classes, attributes and operations in each use-case and draw the class diagram for this system.

Exercise 4.5.3. Consider the specification of the embedded system of the railway controller in Exercise 3.7.3. Identify classes, attributes and operations in each use-case and draw the class diagram for this system.

Exercise 4.5.4. Consider a building comprising a number of floors in which an elevator system is working. The elevators move among different floors in the building. There are two types of buttons in the system: buttons inside the elevator and a button outside at each floor. A button may be turned on by pushing it or turned off, when the door of the elevator on the corresponding floor is opened. The door of the elevator is closed before it moves from one floor to the other.

1. Identify classes, their attributes and operations in the above system.

2. Draw the class diagram for this system.