

Chapter 8

All Together Now

In the preceding chapters, we have looked in detail at five UML diagrams that enable us to describe different aspects of a system. In the examples given, we have seen that the diagrams each realize different views of a system. Therefore, the diagrams must be interpreted together as a whole, taking into account how they interact with one another, rather than each one being considered in isolation. They supplement each other by illustrating the system to be developed from different perspectives. In this chapter, we model three concrete examples from different application areas that show the interaction between the different diagrams.

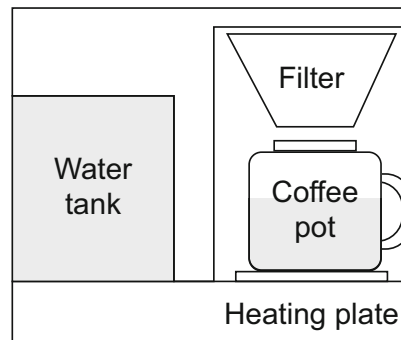
8.1 Example 1: Coffee Machine

An important device encountered time and again in a university is the coffee machine. Let us look at a filter coffee machine as shown in [Figure 8.1](#). The coffee machine consists of a water tank, a heating plate, a coffee pot, and a water pipe that leads from the water container to the filter. When there is water in the tank and the coffee machine is switched on, the water is heated. The pressure pushes the water upwards through the pipe into the filter which contains the ground coffee. Finally, the brewed coffee flows out of the filter into the coffee pot. The coffee machine is available in two different versions, one with a “keep warm” function (model A) and one without (model B). If the water tank is empty and the coffee machine is switched on, in model A the “keep warm” function is activated. In the same situation, model B simply switches off.

The use case diagrams in [Figure 8.2](#) describe the functionality of-

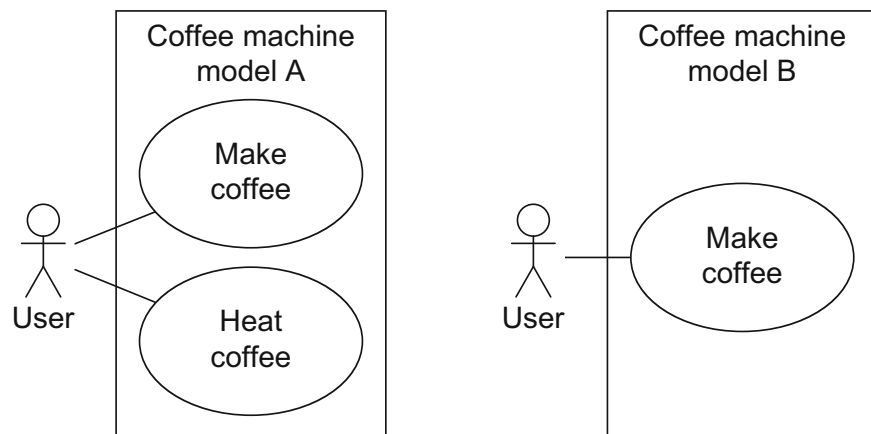
Use case diagram

Figure 8.1
Coffee machine



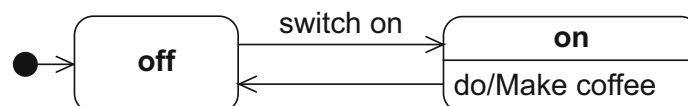
ferred by model A and model B. While model A offers the two functions Heat Coffee and Make Coffee, model B can only be used to make coffee. In both cases, we assume very simple coffee machines that are limited to the “core competencies” of coffee machines.

Figure 8.2
Use case diagrams for a
coffee machine



Naturally, users can switch the coffee machine on and off. Maintenance activities such as filling the machine up with coffee or cleaning the filter must also be possible. We have intentionally not modeled these as separate use cases as they are preparatory tasks required to achieve the actual objective—the brewed or warmed up coffee.

Figure 8.3
State machine diagram for
coffee machine model B



From the description, we can see that both coffee machine models can take the states on and off. Model B exits the state on when the coffee has been made (see Fig. 8.3); in model A, the event switch off must occur for this state change. In model A, the state on can be refined into the states ready (where the heating function is available) and in use (the coffee is being made). The machine can only switch to the state in use when the water tank of the coffee machine is filled (see Fig. 8.4).

State machine diagram

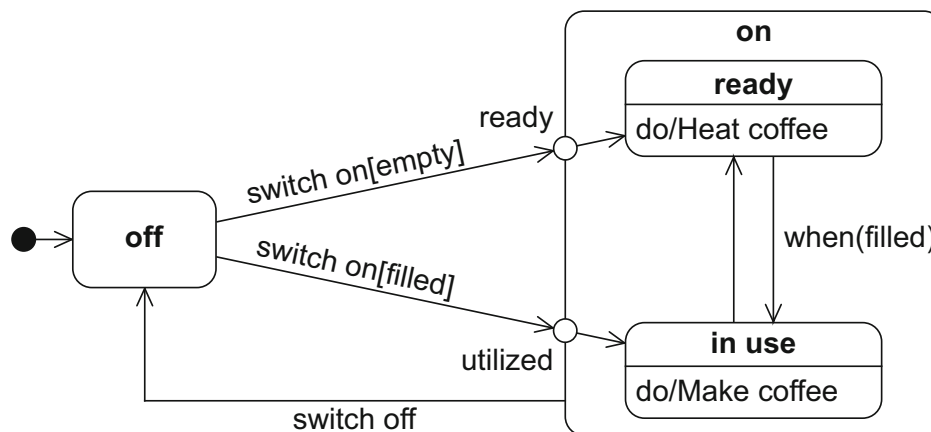


Figure 8.4
State machine diagram for
coffee machine model A

The activity diagram in Figure 8.5 describes how to use coffee machine model B. First, the coffee machine is prepared for making the coffee. This involves cleaning the filter, filling the machine with ground coffee, filling the machine with water, and switching the machine on. Note that the filter is always cleaned before the ground coffee is added and that the water is added before the coffee machine is switched on, otherwise the machine switches itself off immediately.

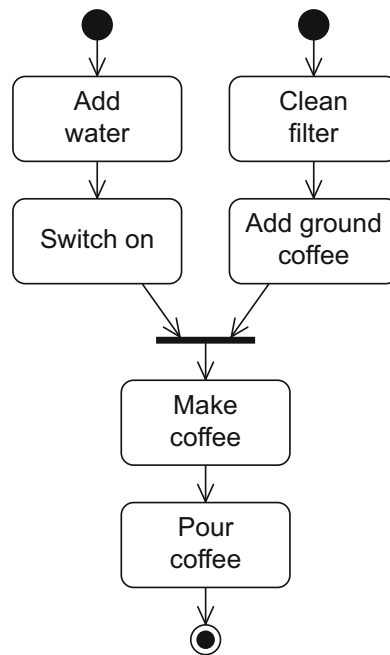
Activity diagram

Apart from these restrictions, the actions can be performed in any order. This is represented in the activity diagram by two concurrent sub-paths, each with a separate initial node. The coffee is not made—that is, the water is not poured through the filter—until the two incoming edges of the synchronization node both obtain a token. Our diagram does not cover a situation in which the coffee machine is used without cleaning the filter and without adding ground coffee.

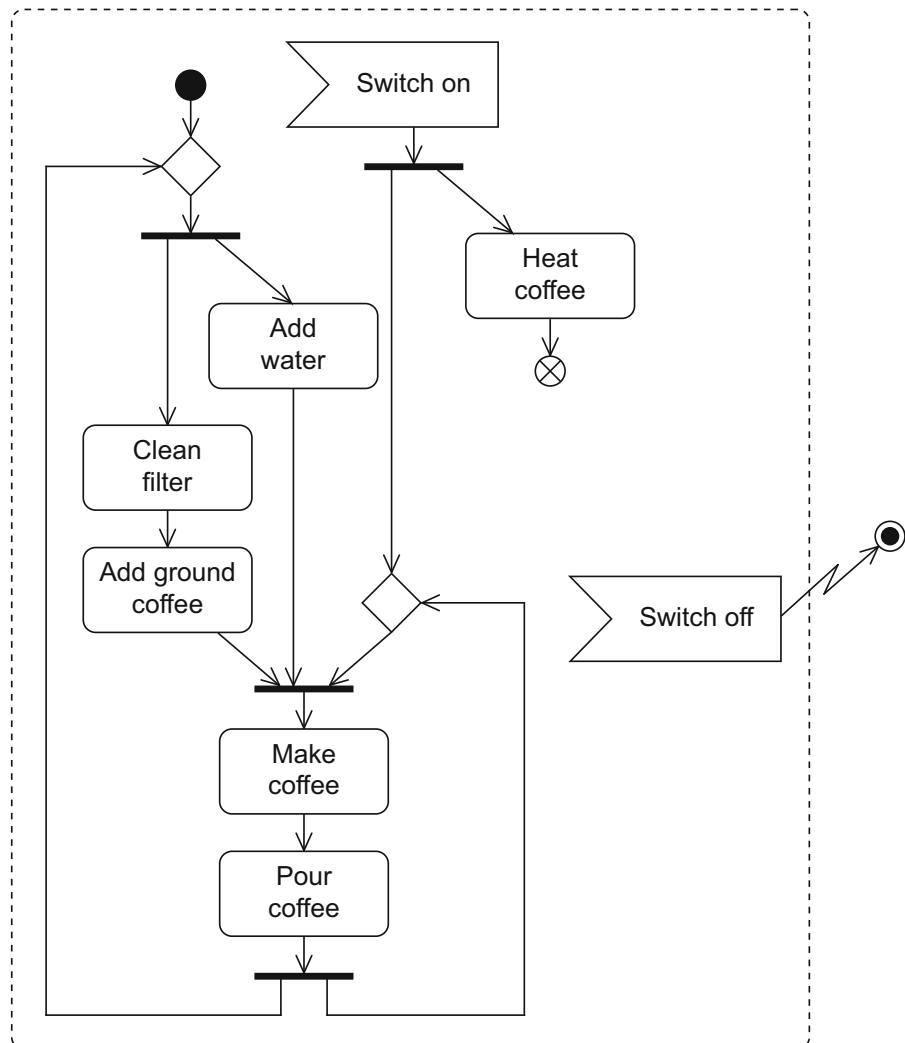
The activity diagram in Figure 8.6 describes the use of model A. As soon as the coffee machine is switched on, it executes the action Heat coffee. When the coffee machine has been fully prepared for making coffee, that is, when ground coffee and water have been added, the keep warm function is switched off and coffee is made. We model this with a synchronization node. The signal Switch off ends the entire process.

Figure 8.5

Activity diagram for coffee machine model B

**Figure 8.6**

Activity diagram for coffee machine model A



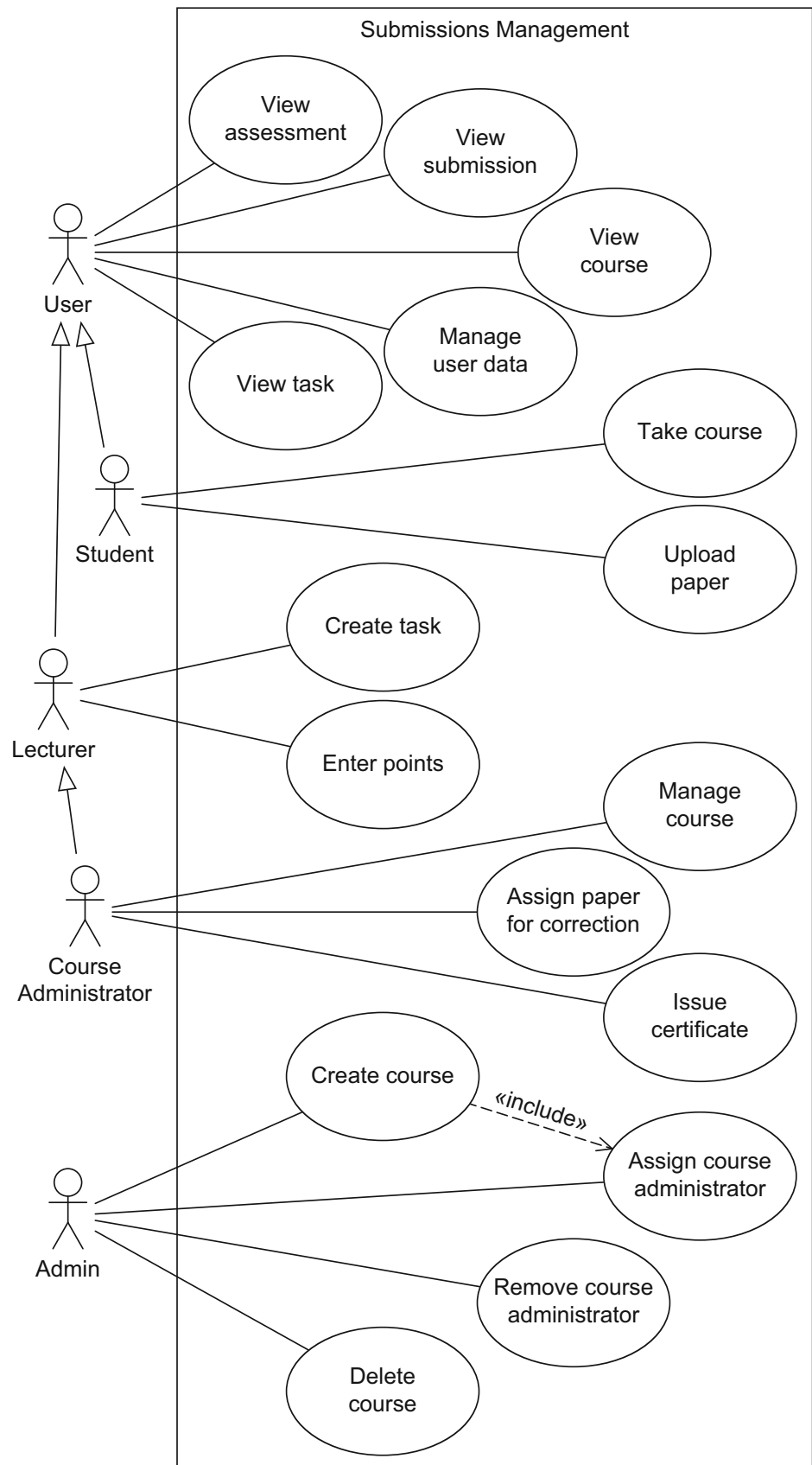
8.2 Example 2: Submission System

UML diagrams are generally used to describe software systems, such as the student administration system of a university, at which we have looked in this book from various aspects. Creating a continuous, detailed model of the entire system which could actually be implemented in executable code would go beyond the scope of this book. However, we will again extract a part of the system to illustrate the interaction of the different diagrams. To this end, we will look at a submission system that is to be used to manage submissions, that is, the students' papers for assignment tasks. The requirements for this system are as follows:

- Every course in the system has lecturers assigned to it. This is done by one of the course administrators, who is also a lecturer. As part of a course, lecturers may create tasks and assess papers submitted by students. Therefore, the lecturers award points and give feedback.
- The course administrator defines which lecturer assesses which papers. At the end of the course, the course administrator also arranges for certificates to be issued. A student's grade is calculated based on the total number of points achieved for the submissions handed in.
- Students can take courses and upload papers.
- All users—students and lecturers—can manage their user data, view the courses and the tasks set for the courses (provided the respective user is involved in the course), and view submitted papers as well as grade points. However, students can only view their own papers and the related grades. Lecturers can only view the papers assigned to them and the grades they have given. The course administrator has access rights for all data.
- A course is created and deleted by an administrator.
- When a course is created, at least one administrator must be assigned to it. Further course administrators can be assigned at a later point in time or assignments to courses can be deleted. The administrator can also delete whole courses.
- Information about users and administrators is automatically transferred from another system. Therefore, functions that allow the creation of user data are not necessary.
- All of the system functions can only be used by persons who are logged in.

The actors and use cases for the specification above are summarized in the use case diagram in [Figure 8.7](#). With regard to the actors, we differentiate between the administrators and all other users, who are in turn subdivided into lecturers and students. With regard to the lecturers, we further differentiate course administrators.

Figure 8.7
Use case diagram for a
submission system



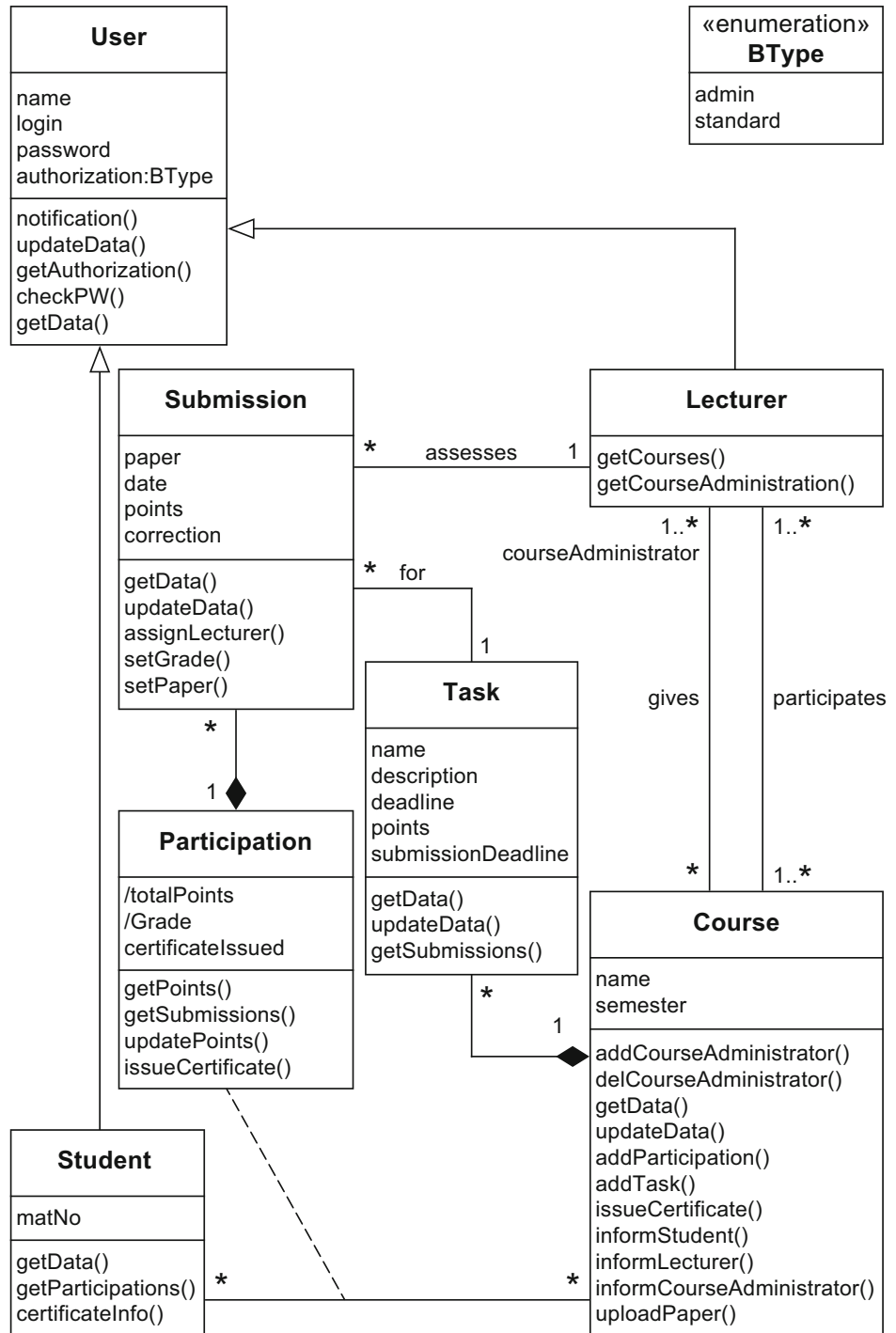
We can derive the use cases directly from the specification above. The use cases are generally not detailed enough for the actual implementation of the system. We now have to highlight exactly which requirements the system must satisfy. However, we do not include this information in the use case diagram as it would make the diagram overloaded and difficult to read. Instead, we document it in the descriptions of the use cases which we learned about in Chapter 3.

The log-in and log-out processes are not represented in the use case diagram as they are not functions desired by the actors but instead contribute to the secure use of the system. Logging in can be considered as a precondition for using the system.

Now that the requirements for the system have been specified, we can zoom into the system and model its structure and behavior. We represent the internal structure of the submission system with the class diagram shown in [Figure 8.8](#). You will notice that all of the actors that appear in the use case diagram are also modeled in the class diagram, even though we stated that they are not part of the system. It is important to understand that in the class diagram, it is the data of the actors that is represented and not the actors themselves. This data is necessary to implement authorizations, the assignment of submissions to students, etc. Information about the users is stored in the class `User`. We use the attribute `authorization` to differentiate between administrators (value `admin`) and all other users (value `standard`). Administrators can be direct instances of the class `User` and lecturers and students are modeled by further classes that have an inheritance relationship to `User`. In principle, this means that it is possible for a lecturer or even a student to be an administrator as well. At first glance, this contradicts our use case diagram, in which the actor `Admin` is in an inheritance relationship to `User`. However, if we consider that the actors in the use case diagram represent roles, our class diagram is correct. One person can of course take multiple roles. In the use case diagram, we have only excluded, for example, that an administrator can automatically view information from the courses. If we really wanted to model a strict differentiation here, we would have to introduce a separate class `Admin` or formulate a constraint that forbids the attribute `authorization` from taking the value `admin` for lecturers or students.

In our class diagram, a lecturer becomes a course administrator by being in a `gives` relationship to a course. Tasks are always assigned to a course, in the same way that submissions are assigned to a task. We model a student's participation in a course using an association class that contains information about the student's total number of points and the grade. Both values are calculated automatically and the attributes are therefore labeled as derived attributes.

Figure 8.8
Class diagram for a sub-
mission system



The class diagram in [Figure 8.8](#) does not ensure that a lecturer can only assess tasks of a course in which this lecturer is involved. We have to specify these and other restrictions that are important for the consistency of the overall system. Therefore, additional constraints are required. These can be specified in languages such as the Object Constraint Language (OCL) [36] which is beyond the scope of this book. Then the check that determines whether the instances of a model comply with the specified restrictions can often be performed automatically.

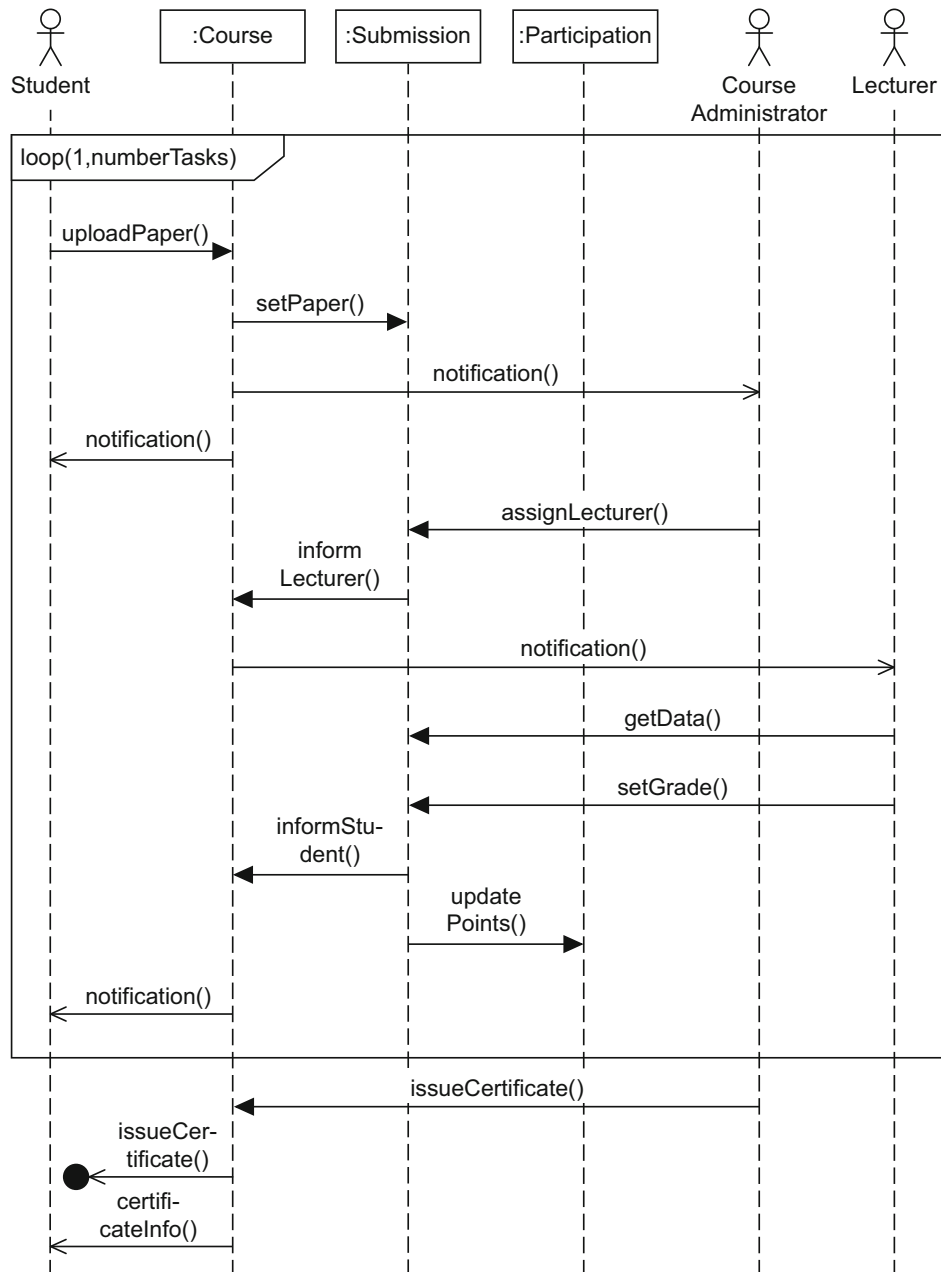
In its current form, the class diagram does not contain any platform-specific information, which means that we have not defined what the implementation should look like. For the general specification of the system to be developed, we want to remain at this abstract level in the subsequent steps.

In the next step, we want to model a typical usage scenario of the submission system, that is, how the actors, the system, and the functions that we have specified in the use case diagram interact in a specific situation. We could create an activity diagram to do this. However, as we want to focus on the communication aspect, we use a sequence diagram instead. [Figure 8.9](#) shows the following usage scenario: A student uploads the solved assignment paper to the submission system. The system informs the course administrator that a new assignment paper has been submitted and confirms to the student that the paper has been successfully received. In the sequence diagram, we do not show the action necessary for saving the paper submitted as it is not relevant for the representation of this specific communication process. Via the submission system, the course administrator assigns a lecturer to the paper. Once the system has informed the lecturer that a paper has been assigned, the lecturer assesses the paper. To do this, the lecturer downloads the paper from the submission system and enters the grade in the system. Then the student is informed that the uploaded paper has been graded. The described communication flow takes place not just once but for every task that has to be completed for a course. Therefore, in the sequence diagram in [Figure 8.9](#), the messages described above are enclosed by a loop fragment.

Once all of the tasks have been processed, the course administrator can arrange for the certificates to be issued. The submission system also informs the student of the final grade.

The sequence diagram described in [Figure 8.9](#) reflects the use of the submission system at a very high abstraction level. Although this highlights the general function of the system, many details are still not specified. Therefore, we have to zoom into the system further. To illustrate this, let us look at the activity Issue certificate shown in [Figure 8.10](#). Here we have decided to use an activity diagram to illustrate the detailed process.

Figure 8.9
Communication flows



We assume that on the user interface, the course administrator sees an overview of the assigned courses. Firstly, the course administrator selects the course to issue certificates for. The students who have taken the course are displayed. The course administrator can then select whether to issue certificates for all or only for certain students. In the latter case, the administrator must also specify the students who shall obtain certificates. The grades are calculated, sent to the student office, and each student is informed of the grade. Note that for the practical implementation, it is extremely important to model all possible flows in as much

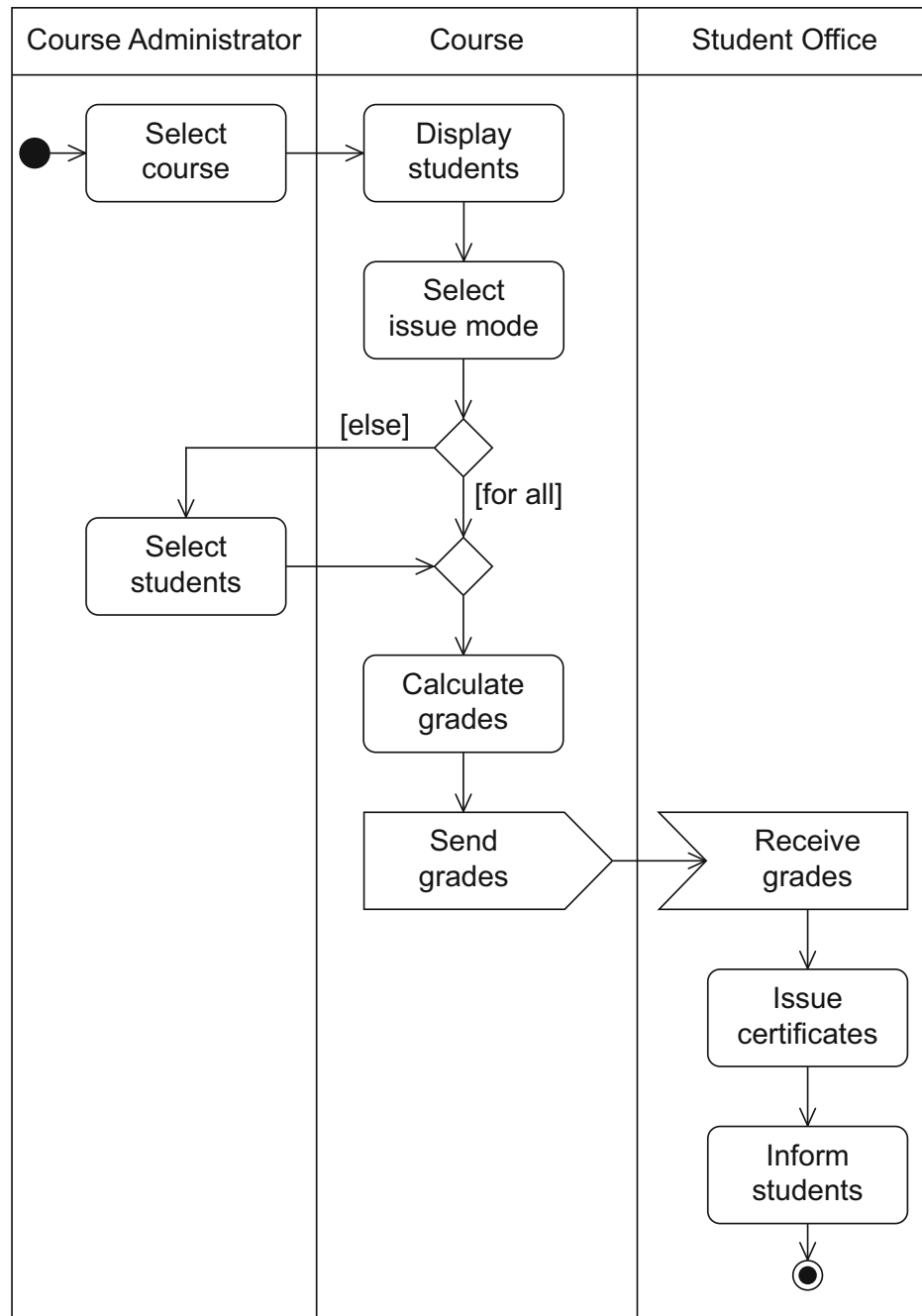
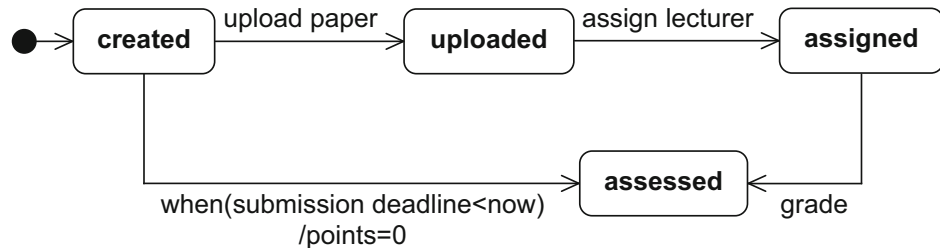


Figure 8.10
Activity diagram for “Issue certificate”

detail as possible and as close to reality as possible. In our activity diagram we have not considered the error situations. It is also not possible in our diagram to manually correct a certificate that has already been issued. All of these cases would also have to be considered. Modeling the processes incorrectly can lead to the system not being accepted by its users. As a result, the increased efficiency expected with the introduction of the system will not be achieved.

To describe the system precisely, it is important also to show the states that the individual components of the system and the system itself can take. As an example, let us look at the state machine diagrams for the submission of a paper (see Fig. 8.11) and for the participation in a course (see Fig. 8.12).

Figure 8.11
State machine diagram for
class Submission



In our system, an instance of class Submission refers to the upload of a paper by a student that then can be assessed by the lecturer. The submission therefore concerns the administration of the file that the student hands over to the lecturer for the assessment, rather than the file itself. When a task is created and released for the students, every student can submit a paper in the system. The submission initially has the state created. It exits this state when the student uploads a paper or when the submission deadline expires—in the latter case the task is assessed with zero points for the student. The submission then changes to the state assessed. If the student submits the solved paper in the system, the submission changes to the state uploaded. It exits this state when the submission is assigned to a lecturer for assessment. Once the assessment has taken place, the submission takes the final state assessed. As the information about a submission is stored for documentation purposes, the submission remains in the state assessed “forever” and no final state is modeled.

The states that a specific participant of a specific course can take are described in a similar way (see Fig. 8.12). In our example, the participation refers to the course participation of a certain student in a certain course. It is documented whether a student has completed a course and, if this is the case, the grade that the student received for this course is saved. When a student has registered for a course, the state of the student’s participation in this specific course is initially not assessed. This clearly shows why we describe the states of course participation and not the states of a student. If a student had the state not assessed, it would not be possible to differentiate between different courses that the student has taken. However, what we want to show is the state of a student with reference to a specific course. In the class diagram, this information is taken into account with the association class Participation (see Fig. 8.8).

For a specific course, a student is initially in the state not assessed, then in the state partially assessed, and finally in the state certificate issued, unless the student is not assessed at all. This can happen, for example, if the student was registered but has never actually attended the course and never completed any activities. The state certificate issued has two substates—positive and negative. Guards specify which of these states occurs. These substates can change if a certificate is corrected.

The diagrams in Figures 8.7 to 8.12 illustrate how the different aspects of a submission system and the interaction between these aspects can be modeled. However, they do not specify any technical details for the implementation, representing instead a sketch that describes how the system should look. Once all of the requirements have been documented in the model, the actual implementation can begin. Different approaches are feasible. One option would be to start the implementation in executable code immediately. However, as no interfaces are specified, sooner or later the different components will not fit together anymore, which makes the maintenance of the system more complicated than with carefully designed interfaces.

Therefore, we recommend refining the model further until you have an exact specification of the system to be developed.

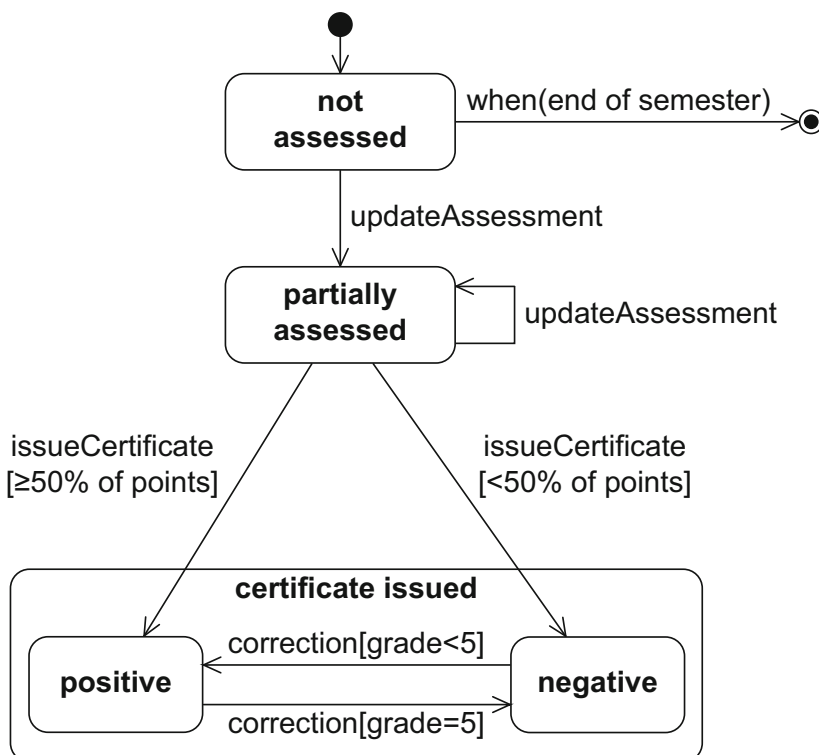


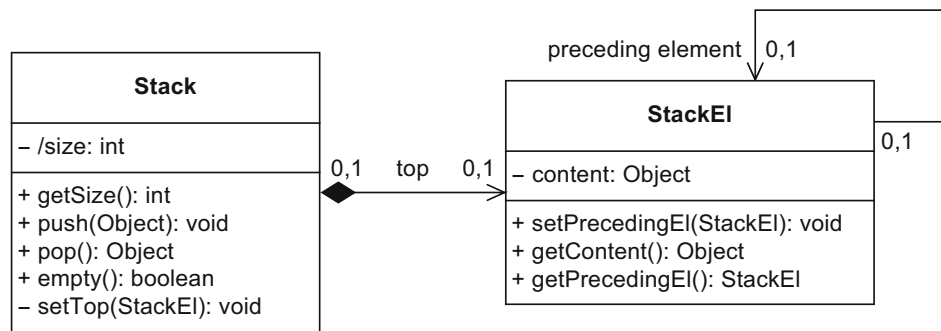
Figure 8.12

State machine diagram for participation of a student in a course

8.3 Example 3: Data Type Stack

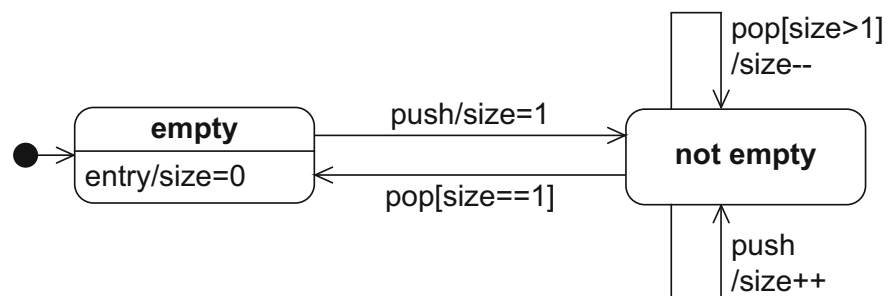
The final example in this chapter is the modeling of the data structure *Stack*. Elements can be placed on the stack using the push function and removed from the stack using the pop function. The order in which elements are removed follows the LIFO principle (Last In, First Out), which means that pop always delivers the element that was last placed on the stack with push and removes it from the stack. Further functions that the class *Stack* should support are the determination of the actual size of the stack, that is, the number of elements on the stack, and the query about whether an element is on the stack at all. As we want to realize a stack with no size restriction, we realize it using a recursive data structure as shown in the class diagram in [Figure 8.13](#). The class *Stack* only knows the uppermost element on the stack. Each element in the stack refers to its direct predecessor. The actual content of a stack element is saved via the private variable *content*.

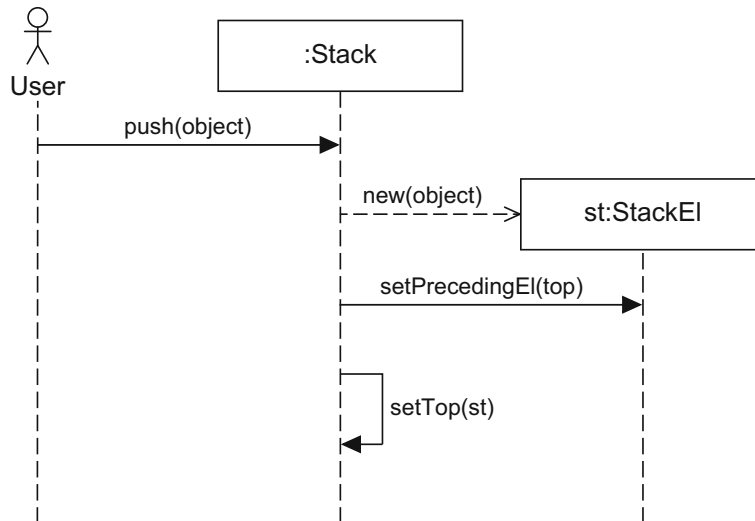
Figure 8.13
Class diagram for a stack



The states that a stack can take are shown in [Figure 8.14](#). Initially the stack is in the state *empty*. If an element is placed on the stack, the stack changes to the state *not empty*. Every time push is called, the size of the stack increases by one. Every time pop is called, the size reduces by one. If there is only one element on the stack and pop is called, then the stack changes to the state *empty*.

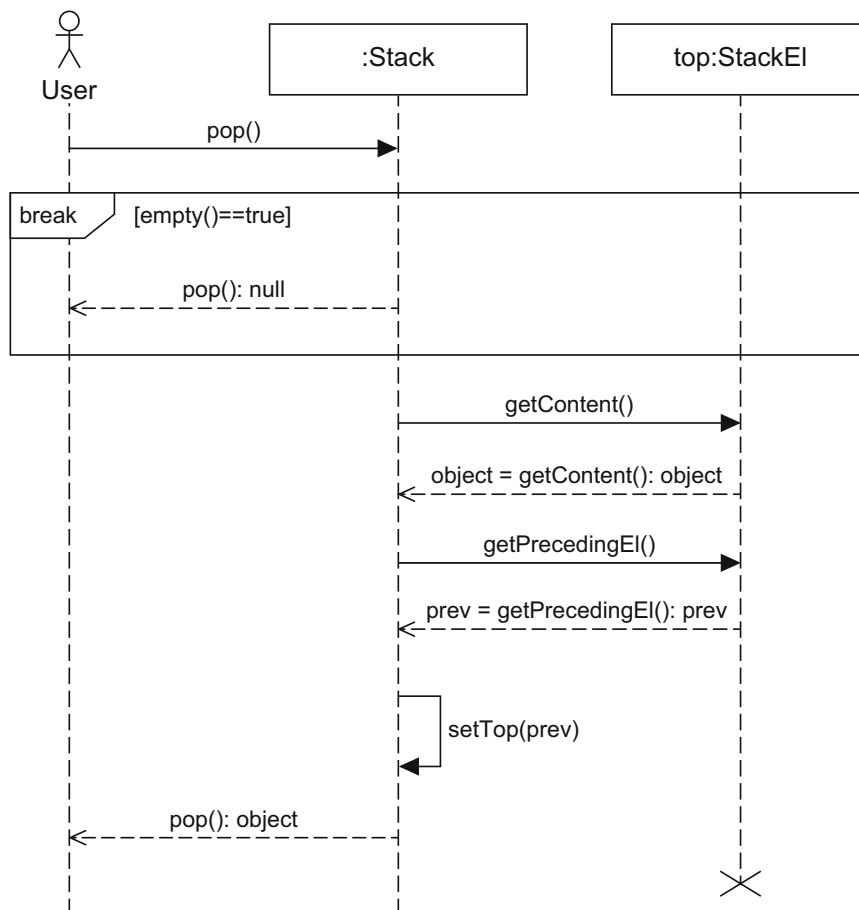
Figure 8.14
State machine diagram for a stack



**Figure 8.15**

Sequence diagram for the addition of an element to the stack

The realization of push and pop is shown in the sequence diagrams 8.15 and 8.16. These diagrams are very close to implementations and reflect how the variables are set.

**Figure 8.16**

Sequence diagram for the removal of an element from the stack

To place an object object on the stack, a new instance of StackEl must be created where the attribute content is set to object. The current top element of the stack becomes the predecessor of the new stack, which now becomes the top element on the stack.

The pop operation reverses the effect of a push operation. The content of the current top element is returned and its predecessor becomes the new uppermost element. If there is no element on the stack, the value null is returned.

8.4 Summary

In the three examples discussed in this chapter, we have not only repeated the most important concepts of UML but have also shown how the different diagrams interact. This interaction allows us to describe a system completely without a developer being supplied with all information at once and being overwhelmed by this flood of information. It enables us to focus on specific questions. The information that is shown in the different diagrams redundantly contributes to making the model more consistent overall, as it allows errors to be found at an earlier development stage and more easily.