Chapter 3
The Use Case Diagram

Use case diagram

The *use case diagram* allows us to describe the possible usage scenarios (use cases) that a system is developed for. It expresses what a system should do but does not address any realization details such as data structures, algorithms, etc. These details are covered by other diagrams such as the class diagram (see Chapter 4) or the interaction diagrams (see Chapter 6). The use case diagram also models which user of the system uses which functionality, i.e., it expresses who will actually work with the system to be built.

The *use case* is a fundamental concept of many object-oriented development methods. It is applied during the entire analysis and design process. Use cases represent what the customer wants the system to do, that is, the customer's requirements of the system. At a very high abstraction level, the use cases show what the future system is for. A use case diagram can also be used to document the functionality of an existing system and to record retrospectively which users are permitted to use which functionality.

Specifically, we can employ a use case diagram to answer the following questions:

1. What is being described? (*The system.*)
2. Who interacts with the system? (*The actors.*)
3. What can the actors do? (*The use cases.*)

The use case diagram provides only a few language elements. At first glance, this diagram seems to be extremely simple to learn and use. In practice, however, the use case diagram is an extremely underestimated diagram. The content of a use case diagram express the expectations that the customer has of the system to be developed. The diagram documents the requirements the system should fulfill. This is essential for a detailed technical design. If use cases are forgotten or specified imprecisely or

M. Seidl et al., *UML @ Classroom*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-12742-2_3

incorrectly, in some circumstances the consequences can be extremely serious: the development and maintenance costs increase, the users are dissatisfied, etc. As a consequence, the system is used less successfully and the investments made in the development of the system do not bring the expected returns. Even though software engineering and methods of requirements analysis are not the subject of this book, we briefly explain why it is essential to create use cases very carefully. Furthermore, we discuss where errors are often made and how these can be avoided with a systematic approach. For a detailed introduction to these topics, see for example [3, 45].

3.1 Use Cases

Use case

A *use case* describes functionality expected from the system to be developed. It encompasses a number of functions that are executed when using this system. A use case provides a tangible benefit for one or more actors that communicate with this use case. The use case diagram does not cover the internal structure and the actual implementation of a use case. In general, a use case is triggered either by invocation of an actor or by a *trigger event*, in short, a *trigger*. An example of a trigger is that the semester has ended and hence the use case Issue certificate must be executed.

Trigger

Use cases are determined by collecting customer wishes and analyzing problems specified in natural language when these are the basis for the requirements analysis. However, use cases can also be used to document the functionality that a system offers. A use case is usually represented as an ellipse. The name of the use case is specified directly in or directly beneath the ellipse. Alternatively, a use case can be represented by a rectangle that contains the name of the use case in the center and a small ellipse in the top right-hand corner. The different notation alternatives for the use case Query student data are illustrated in Figure 3.1. The alternatives are all equally valid, but the first alternative, the ellipse that contains the name of the use case, is the one most commonly used.

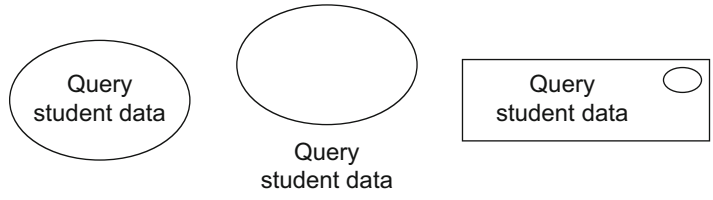


Figure 3.1
Notation alternatives for use cases

The set of all use cases together describes the functionality that a software system provides. The use cases are generally grouped within a rectangle. This rectangle symbolizes the boundaries of the *system* to be described. The example in Figure 3.2 shows the Student Administration system, which offers three use cases: (1) Query student data, (2) Issue certificate, and (3) Announce exam. These use cases may be triggered by the actor Professor.

System

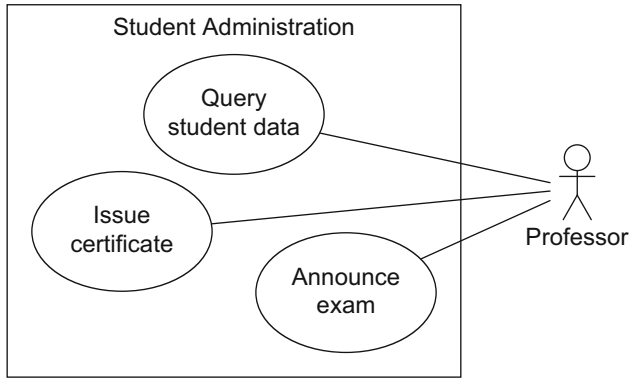


Figure 3.2
Representation of system boundaries

3.2 Actors

To describe a system completely, it is essential to document not only what the system can do but also who actually works and interacts with the system. In the use case diagram, *actors* always interact with the system in the context of their use cases, that is, the use cases with which they are associated. The example in Figure 3.2 contains only the actor Professor, who can query student data, announce exams, and issue certificates. Actors are represented by stick figures, rectangles (containing the additional information «actor»), or by a freely definable symbol. The notation alternatives are shown in Figure 3.3. These three notation alternatives are all equally valid. As we can see from this example, actors can be *human* (e.g., student or professor) or *non-human* (e.g., e-mail server). The symbols used to represent the actors in a specific use case diagram depend on the person creating the model or the tool used. Note in particular that non-human actors can also be portrayed as stick figures, even if this seems counterintuitive.

Actor

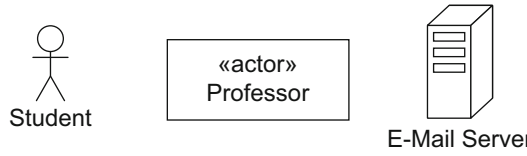


Figure 3.3
Notation alternatives for actors

Types of actors:

- *Human/non-human*
- *Active/passive*
- *Primary/ secondary*

An actor interacts with the system by using the system as an *active* actor, meaning that the actor initiates the execution of use cases; alternatively, the interaction involves the actor being used by the system, meaning that the actor is a *passive* actor providing functionality for the execution of use cases. In example (a) in Figure 3.4, the actor Professor is an active actor, whereas the actor E-Mail Server is passive. However, both are required for the execution of the use case Inform student. Furthermore, use case diagrams can also contain both *primary* and *secondary* actors, also shown in this example. A primary actor takes an actual benefit from the execution of the use case (in our example this is the Professor), whereas the the secondary actor E-Mail Server receives no direct benefit from the execution of the use case. As we can see in example (b) in Figure 3.4, the secondary actor does not necessarily have to be passive. Both the Professor and the Student are actively involved in the execution of the use case Exam, whereby the main beneficiary is the Student. In contrast, the Professor has a lower benefit from the exam but is necessary for the execution of the use case. Graphically, there is no differentiation between primary and secondary actors, between active and passive actors, and between human and non-human actors.

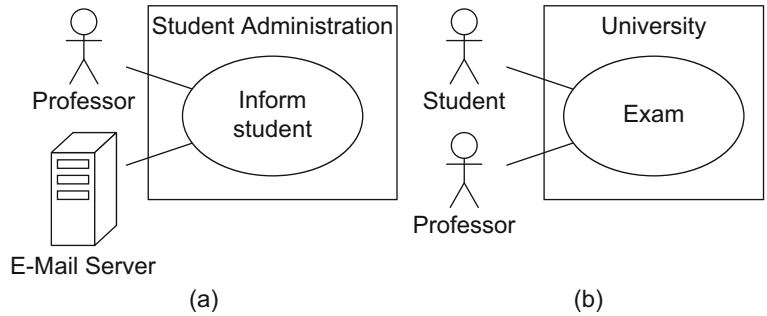


Figure 3.4
Examples of actors

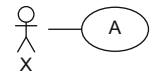
An actor is always clearly outside the system, i.e., a user is never part of the system and is therefore never implemented. Data about the user, however, can be available within the system and can be represented, for example, by a class in a class diagram (see Chapter 4). Sometimes it is difficult to decide whether an element is part of the system to be imple-

mented or serves as an actor. In example (a) in Figure 3.4, the E-Mail Server is an actor—it is not part of the system but it is necessary for the execution of the use case Inform student. However, if no external server is required to execute this use case because the student administration system implements the e-mail functionality itself or has its own server, the E-Mail Server is no longer an actor. In that case, only the Professor is required to inform students about various news items.

3.3 Associations

In the examples in Figure 3.4, we connected the actors with use cases via solid lines without explaining this in more detail. An actor is connected with the use cases via *associations* which express that the actor communicates with the system and uses a certain functionality. Every actor must communicate with at least one use case. Otherwise, we would have an actor that does not interact with the system. In the same way, every use case must be in a relationship with at least one actor. If this were not the case, we would have modeled a functionality that is not used by anyone and is therefore irrelevant.

Association



An association is always binary, meaning that it is always specified between one use case and one actor. Multiplicities may be specified for the association ends. If a multiplicity greater than 1 is specified for the actor's association end, this means that more than one instance of an actor is involved in the execution of the use case. If we look at the example in Figure 3.5, one to three students and precisely one assistant is involved in the execution of the use case Conduct oral exam. If no multiplicity is specified for the actor's association end, 1 is assumed as the default value. The multiplicity at the use case's association end is mostly unrestricted and is therefore only rarely specified explicitly.

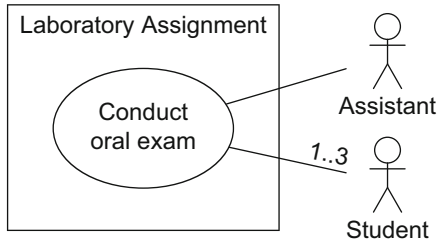


Figure 3.5
Multiplicities in associations

Role

Actors do not represent a specific user—they represent *roles* that users adopt. If a user has adopted the respective role, this user is authorized to execute the use cases associated with this role. Specific users can adopt and set aside multiple roles simultaneously. For example, a person can be involved in the submission of a certain assignment as an assistant and in another assignment as a student. The role concept is also used in other types of UML diagrams, such as the class diagram (see Chapter 4), the sequence diagram (see Chapter 6), and the activity diagram (see Chapter 7).

3.4 Relationships between Actors

Synonyms:

- *Generalization*
- *Inheritance*

Generalization for actors

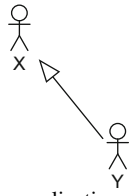
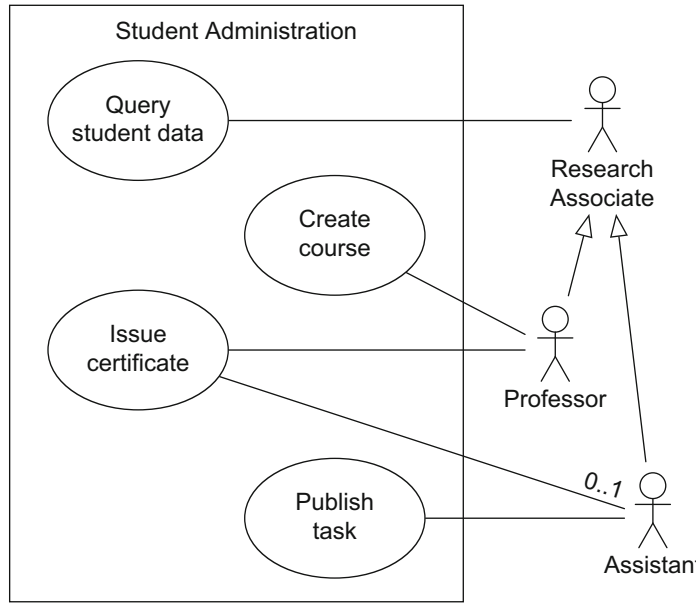


Figure 3.6
Example of generalization for actors

Actors often have common properties and some use cases can be used by various actors. For example, it is possible that not only professors but also assistants (i.e., the entire research personnel) are permitted to view student data. To express this, actors may be depicted in an *inheritance relationship* (generalization) with one another. When an actor Y (sub-actor) inherits from an actor X (super-actor), Y is involved with all use cases with which X is involved. In simple terms, generalization expresses an "is a" relationship. It is represented with a line from the sub-



actor to the super-actor with a large triangular arrowhead at the super-actor end. In the example in Figure 3.6, the actors Professor and Assistant inherit from the actor Research Associate, which means that every professor and every assistant is a research associate. Every research associate can execute the use case Query student data. Only professors can create a new course (use case Create course); in contrast, tasks can only be published by assistants (use case Publish task). To execute the use case Issue certificate in Figure 3.6, an actor Professor is required; in addition, an actor Assistant can be involved optionally, which is expressed by the multiplicity 0..1.

There is a great difference between two actors participating in a use case themselves and two actors having a common super-actor that participates in the use case. In the first case, both actors must participate in the use case (see Fig. 3.7(a)); in the second case, each of them inherits the association. Then each actor participates in the use case individually (see Fig. 3.7(b)).

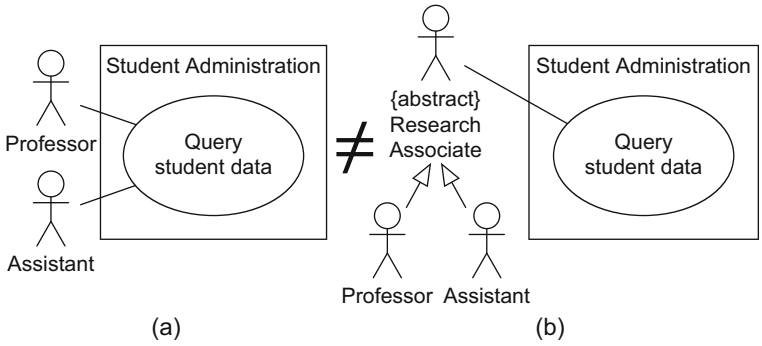


Figure 3.7
Example with and without generalization

Abstract actor

If there is no instance of an actor, this actor can be labeled with the keyword {abstract}. Alternatively, the names of abstract actors can be represented in italic font. The actor Research Associate in Figure 3.7(b) is an example of an abstract actor. It is required to express that either a Professor or an Assistant is involved in the use case Query student data. The use of abstract actors only makes sense in the context of an inheritance relationship: the common properties of the sub-actors are grouped and described at one point, namely with the common, abstract super-actor.

Generalization is a fundamental concept of object orientation and can be applied to many different language elements of UML. For a more detailed introduction to generalization, see Chapter 4.

3.5 Relationships between Use Cases

Up to this point, we have learned only about relationships between use cases and actors (associations) and between actors themselves (generalization of actors). Use cases can also be in a relationship with other use cases. Here we differentiate between «include» relationships, «extend» relationships, and generalizations of use cases.

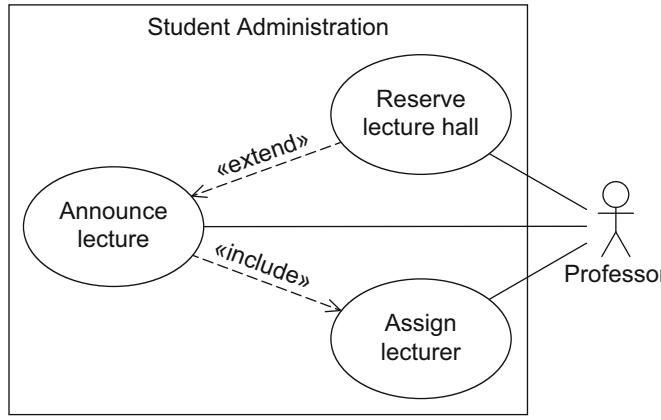
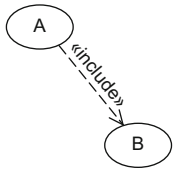


Figure 3.8
Example of «include» and «extend»

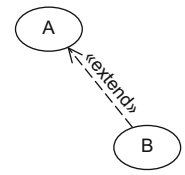


If a use case A includes a use case B, represented as a dashed arrow from A to B labeled with the keyword «include», the behavior of B is integrated into the behavior of A. Here, A is referred to as the *base use case* and B as the *included use case*. The base use case always requires the behavior of the included use case to be able to offer its functionality. In contrast, the included use case can be executed on its own. The use of «include» is analogous to calling a subroutine in a procedural programming language. In the use case diagram in Figure 3.8, the use cases Announce lecture and Assign lecturer are in an «include» relationship, whereby Announce lecture is the base use case. Therefore, whenever a new lecture is announced, the use case Assign lecturer must also be executed. The actor Professor is involved in the execution of both use cases. Further lecturers can also be assigned to an existing lecture as the included use case can be executed independently of the base use case. One use case may include multiple other use cases. One use case may also be included by multiple different use cases. In such situations, it is important to ensure that no cycle arises.

If a use case B is in an «extend» relationship with a use case A, then A can use the behavior of B but does not have to. Use case B can therefore be activated by A in order to insert the behavior of B in A. Here,

A is again referred to as the *base use case* and B as the *extending use case*. An «extend» relationship is shown with a dashed arrow from the extending use case B to the base use case A. Both use cases can also be executed independently of one another. If we look at the example in Figure 3.8, the two use cases Announce lecture and Reserve lecture hall are in an «extend» relationship. When a new lecture is announced, it is possible (but not mandatory) to reserve a lecture hall. A use case can act as an extending use case several times or can itself be extended by several use cases. Again, no cycles may arise. Note that the arrow indicating an «extend» relationship points towards the base use case, whereas the arrow indicating an «include» relationship originates from the base use case and points towards the included use case.

«extend»



Condition

A *condition* that must be fulfilled for the base use case to insert the behavior of the extending use case can be specified for every «extend» relationship. The condition is specified, within curly brackets, in a note that is connected with the corresponding «extend» relationship. A condition is indicated by the preceding keyword Condition followed by a colon. Two examples are shown in Figure 3.9. Within the context of the use case Announce lecture, a lecture hall can only be reserved if it is free. Furthermore, an exam can only be created if the required data has been entered.

Extension point

By using *extension points*, you can define the point at which the behavior of the extending use cases must be inserted in the base use case. The extension points are written directly within the use case, as illustrated in the use case Announce lecture in the example in Figure 3.9. Within the use case symbol, the extension points have a separate sec-

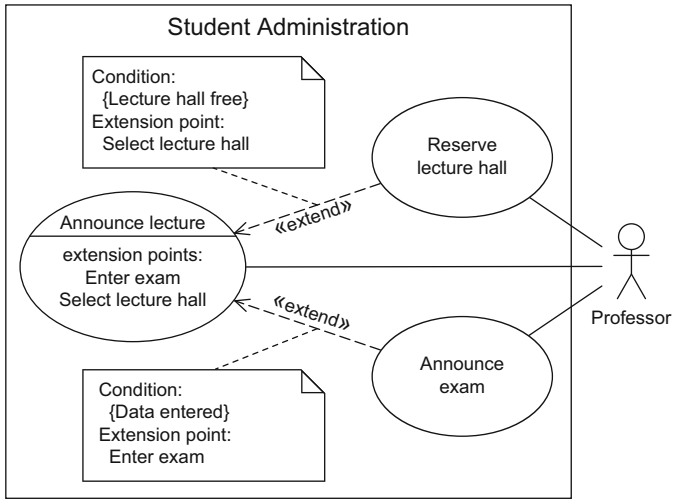
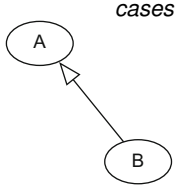


Figure 3.9
Example of extension points and conditions

tion that is identified by the keyword extension points. If a use case has multiple extension points, these can be assigned to the corresponding «extend» relationship via specification in a note similarly to a condition.

Generalization for use cases



In the same way as for actors, *generalization* is also possible between use cases. Thus, common properties and common behavior of different use cases can be grouped in a parent use case. If a use case A generalizes a use case B, B inherits the behavior of A, which B can either extend or overwrite. Then, B also inherits all relationships from A. Therefore, B adopts the basic functionality of A but decides itself what part of A is executed or changed. If a use case is labeled {abstract}, it cannot be executed directly; only the specific use cases that inherit from the abstract use case are executable.

The use case diagram in Figure 3.10 shows an example of the generalization of use cases. The abstract use case Announce event passes on its properties and behavior to the use cases Announce lecture and Announce talk. As a result of an «include» relationship, both use cases must execute the behavior of the use case Assign lecturer. When a lecture is announced, an exam can also be announced at the same time. Both use cases inherit the relationship from the use case Announce event to the actor Professor. Thus, all use cases are connected to at least one actor, the prerequisite previously stipulated for correct use case diagrams.

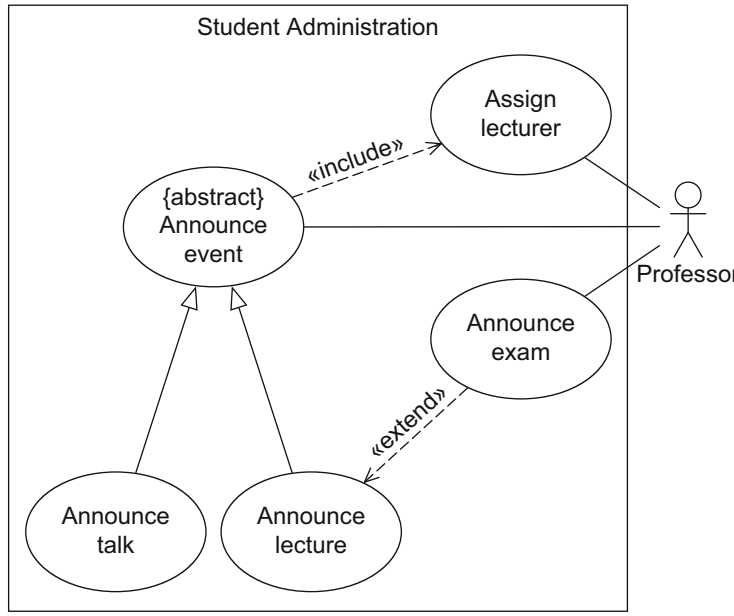


Figure 3.10
Example of generalization of use cases

Generalization allows us to group the common features of the two use cases Announce lecture and Announce talk. This means that we do not have to model both the «include» relationship and the association with the professor twice.

3.6 Examples of Relationships

To explain again explicitly how the different relationship types in a use case diagram interact with one another, let us take a look at the use case diagram from Figure 3.11 and discuss some interesting cases that occur here.

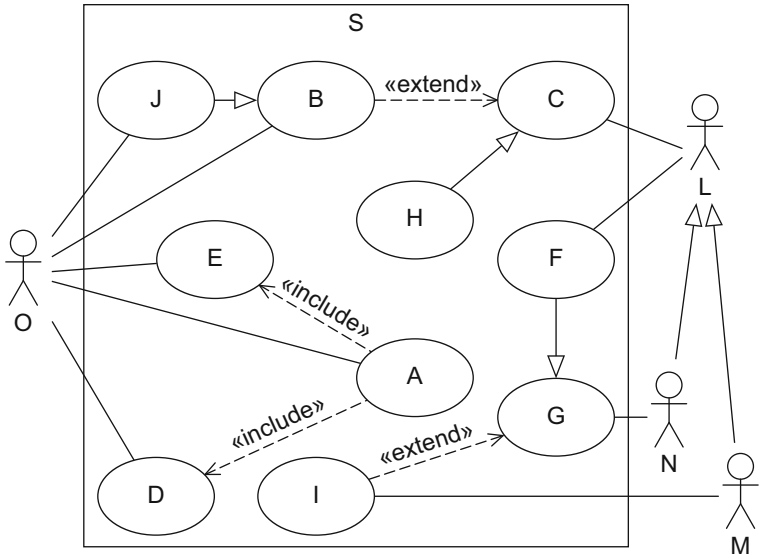


Figure 3.11
Examples of relationships in a use case diagram

- *The use case A includes the use cases E and D.* An actor O is involved in all three use cases. There is no specification of whether this is the same user or different users, that is, different instances of O.
- *The use case H inherits from the use case C.* As use case C is executed by the actor L, an actor L must also be involved in the execution of H. The actors N and M inherit from L. Therefore, both use cases C and H can also be executed by an actor M or N.

- *The use case J inherits from the use case B.* As a result of the inheritance relationship, an actor O is involved in the execution of use case J. However, an association with O is also modeled for J directly. The consequence of this is that two actors in the role O are involved in the execution of J. Note that these two actors can coincide.
- *The use case F inherits from the use case G.* As a result of the inheritance relationship, an actor N is involved in the execution of use case F. For F, an association with the actor L is also modeled directly. Therefore, an actor N and, due to the inheritance relationship of the actors L, N, and M, either an actor L or an actor M or an additional actor N is involved in the execution of F. If two actors N are involved, they may coincide.
- *The use case I extends the use case F.* As use case F inherits from use case G and as I extends use case G, this relationship is passed on to F. If G and I were in an «include» relationship, this relationship would also be passed on to F in the same way.
- *The use case J extends the use case H.* This is as a result of the inheritance relationships from B to J and from C to H.

3.7 Creating a Use Case Diagram

So, how do you create a use case diagram? First you must identify actors and use cases and then place them in relationships with one another. You then describe the use cases in detail. At first glance, this diagram seems to be simple due to the low number of concepts involved. But in fact, use case diagrams are often created incorrectly with a lot of errors. Therefore, here we take a brief look at the principles of creating use cases. For details, see the extensive literature on requirements engineering, for example [16, 30, 40]. We then explain some typical pitfalls to be avoided when modeling use case diagrams.

3.7.1 Identifying Actors and Use Cases

According to [30], there are two ways to identify use cases for prospective system design:

1. Analysis of requirements documents
2. Analysis of the expectations of future users

Requirements documents are generally natural language specifications that explain what the customer expects from a system. They should doc-

ument relatively precisely who will use the system and how they will use it. If you follow the second approach for finding use cases, you must first identify the future users—that is, the actors. To identify the actors that appear in a use case diagram, you must answer the following questions:

Questions for identifying actors

- Who uses the main use cases?
- Who needs support for their daily work?
- Who is responsible for system administration?
- What are the external devices/(software) systems with which the system must communicate?
- Who has an interest in the results of the system?

Once you know the actors, you can derive the use cases by asking the following questions about the actors [27]:

Questions for identifying use cases

- What are the main tasks that an actor must perform?
- Does an actor want to query or even modify information contained in the system?
- Does an actor want to inform the system about changes in other systems?
- Should an actor be informed about unexpected events within the system?

Iterative and incremental determination of use cases

In many cases, you model use cases iteratively and incrementally. In doing so, you often start with the "top level" requirements that reflect the business objectives to be pursued with the software. You then continue to refine them until, at a technical level, you have specified what the system should be able to do. For example, a "top level" requirement for a university administration system could be that the system can be used for student administration. If we refine this requirement, we define that new students should be able to register at the university and enroll for studies, that the students' grades for different courses should be stored, etc.

3.7.2 Describing Use Cases

To ensure that even large use case diagrams remain clear, it is extremely important to select short, concise names for the use cases. When situations arise in which the intention behind the use case and its interpretation are not clear, you must also describe the use cases. Again, it is important to ensure that you describe the use cases clearly and concisely, as otherwise there is a risk that readers will only skim over the document.

Structured approach to describing use cases

A generally recognized guideline for the length of use case descriptions is approx. 1–2 pages per use case. In [15], Alistair Cockburn presents a structured approach for the description of use cases that contains the following information:

- Name
- Short description
- Precondition: prerequisite for successful execution
- Postcondition: system state after successful execution
- Error situations: errors relevant to the problem domain
- System state on the occurrence of an error
- Actors that communicate with the use case
- Trigger: events which initiate/start the use case
- Standard process: individual steps to be taken
- Alternative processes: deviations from the standard process

Table 3.1
Use case description for Reserve lecture hall

Name:	Reserve lecture hall
Short description:	An employee reserves a lecture hall at the university for an event.
Precondition:	The employee is authorized to reserve lecture halls. Employee is logged in to the system.
Postcondition:	A lecture hall is reserved.
Error situations:	There is no free lecture hall.
System state in the event of an error:	The employee has not reserved a lecture hall.
Actors:	Employee
Trigger:	Employee requires a lecture hall.
Standard process:	(1) Employee selects the lecture hall. (2) Employee selects the date. (3) System confirms that the lecture hall is free. (4) Employee confirms the reservation.
Alternative processes:	(3') Lecture hall is not free. (4') System proposes an alternative lecture hall. (5') Employee selects the alternative lecture hall and confirms the reservation.

Table 3.1 contains the description of the use case Reserve lecture hall in a student administration system. The description is extremely simplified but fully sufficient for our purposes. The standard process and the alternative process could be refined further or other error situations and alternative processes could be considered. For example, it could be possible to reserve a lecture hall where an event is already taking place—this makes sense if the event is an exam that could be held in the lecture hall along with another exam, meaning that fewer exam supervisors are required. In a real project, the details would come from the requirements and wishes of the customers.

3.7.3 Pitfalls

Unfortunately, errors are often made when creating use case diagrams. Six examples of typical types of errors are discussed below. For a more detailed treatment of this topic, see [39].

Error 1: Modeling processes

Even if it is often very tempting to model entire (business) processes or workflows in a use case diagram, this is an incorrect use of the diagram. Let us assume we are modeling the system Student Office (see the final example of this chapter on page 42). If a student uses the function Collect certificate, the student must first be notified that the certificate is ready for collection in the student office. Naturally, the lecturer must have sent the certificate to the student office, i.e., the certificate has been issued. The use cases Collect certificate, Send notification, and Issue certificate may be connected chronologically but this should not be represented in a use case diagram. It is therefore incorrect to relate these use cases to one another using «include» or «extend» relationships as shown in Figure 3.12. The functionality that one of these use cases offers is not part of the functionality that another use case offers, hence the use cases must be used independently of one another.

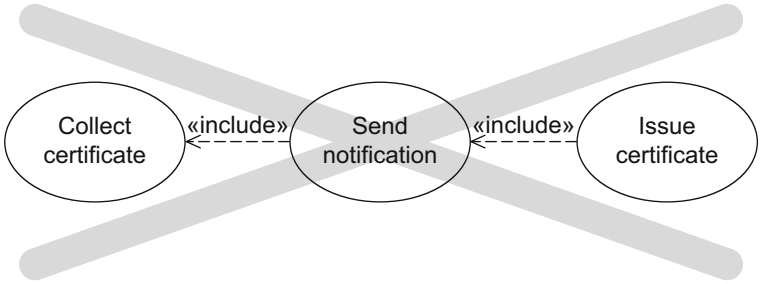


Figure 3.12
Incorrect excerpt of a use case diagram: modeling processes

Error 2: Setting system boundaries incorrectly

When modeling a use case diagram, you must consider very carefully where to draw the boundaries of the diagram. As already mentioned, this is often not clear. Actors are always outside the system boundaries: if they are to be located within the system, they are part of the system and therefore they *must not* be modeled as actors. In Figure 3.13, the Employee is depicted within the boundaries of the system Student Administration. Of course the student administration system includes employees. However, as we want to create a use case diagram of this system,

we must consider whether we want to view these employees as actors or as part of the student administration system. If they are a part of the system, they must not be modeled as actors. In that case, some other entity outside the system should be an actor. If they are not part of the system but are necessary for the execution of the use cases, they must be represented as actors—outside the system boundaries.

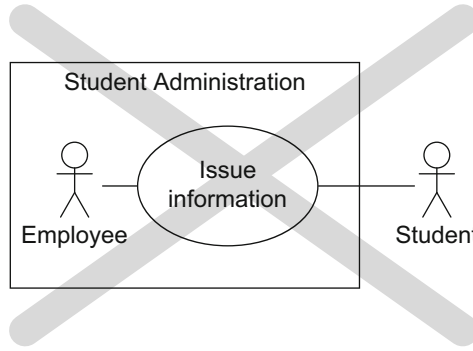


Figure 3.13
Incorrect excerpt of a use case diagram: incorrect system boundaries

Error 3: Mixing abstraction levels

When identifying use cases, you must always ensure that they are located on the same abstraction level. Avoid representing "top level" use cases with technically oriented use cases in the same diagram, as is the case in Figure 3.14. In this example, the management of student data and the selection of a printer, which is a technical feature of the system, are shown together. To avoid this type of error, you should therefore proceed iteratively. First create a use case diagram with use cases that are based on the business objectives (in our example, management of student data). Then refine these use cases down to the technical requirements (selecting a printer).

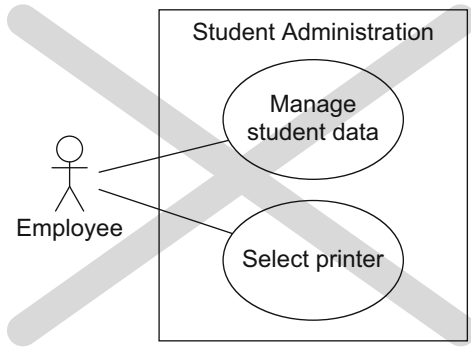


Figure 3.14
Incorrect excerpt of a use case diagram: mixing abstraction levels

Error 4: Functional decomposition

Use cases—even included or extending use cases—can always be executed independently. If they can only be executed within the scope of another use case and not independently, they are not use cases and must not be depicted as such. Their functionality must then be covered in the description of the use case that uses them. In Figure 3.15(a), the use case Issue certificate is broken down into the individual subfunctions necessary to execute the use case. These subfunctions are modeled as use cases even though sometimes they are not meaningful independent use cases, such as Enter data.

The use case Log in is also not a functionality that is part of Issue certificate. In fact, it is a precondition that the user must be logged in with sufficient authorizations for being able to execute this use case. Therefore, a reduced use case diagram, as shown in Figure 3.15(b), is sufficient. The other information specified in Figure 3.15(a) must be specified in the use case description.

Error 5: Incorrect associations

If a use case is associated with two actors, this does not mean that either one or the other actor is involved in the execution of the use case: it means that both are necessary for its execution. In the use case diagram in Figure 3.16(a), the actors Assistant and Professor are involved in the execution of the use case Issue information, which is not the intention. To resolve this problem, we can introduce a new, abstract actor Research Associate from which the two actors Assistant and Professor inherit. The actor Employee is now connected with the use case Issue information (see Fig. 3.16(b)).

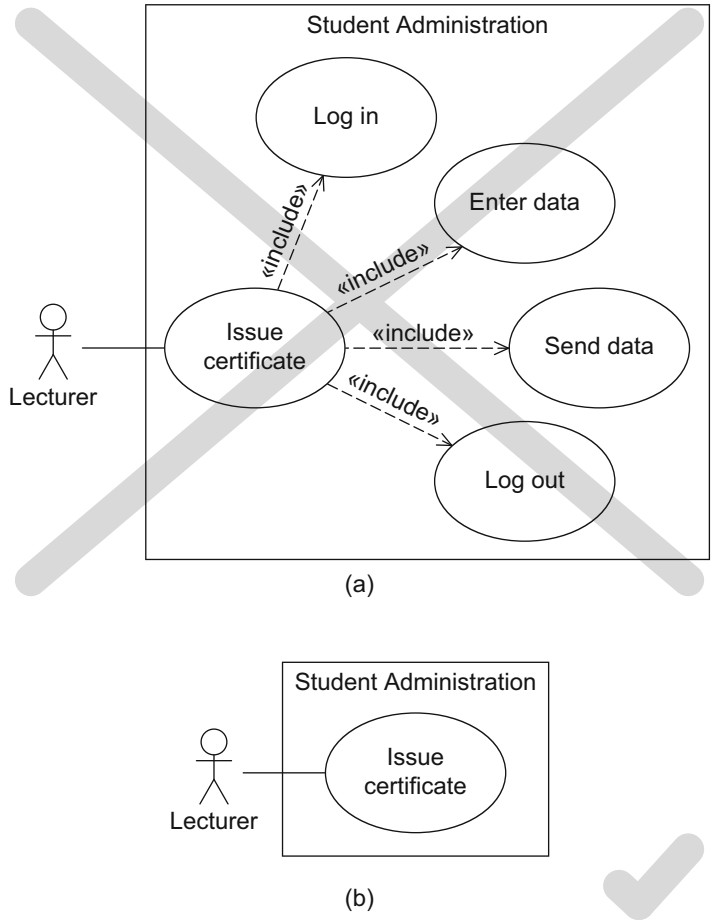


Figure 3.15
Incorrect excerpt of a use case diagram: functional decomposition

Error 6: Modeling redundant use cases

When modeling use cases, it is very tempting to create a separate use case for each possible situation. For example, in the use case diagram in Figure 3.17(a), we have modeled separate use cases for creating, updating, and deleting courses. This shows the different options available for editing a course in the system. In such a small use case diagram as that shown in Figure 3.17(a), it is not a problem to show the differentiations at such a detailed level.

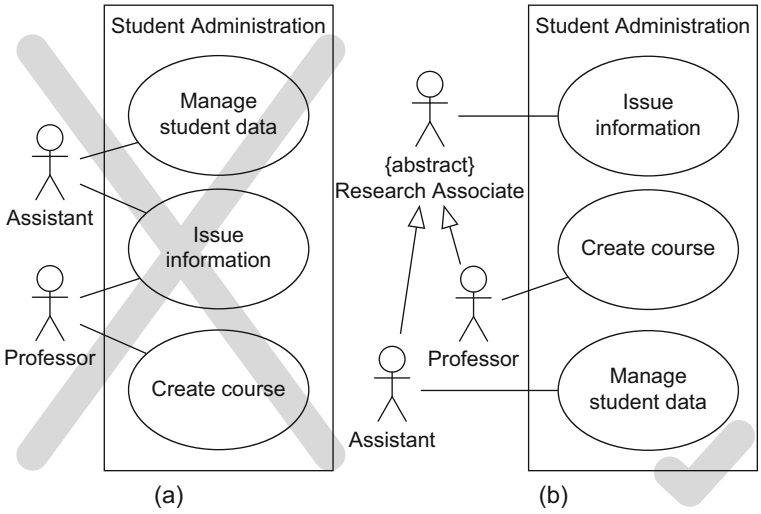


Figure 3.16
Incorrect excerpt of a use case diagram: incorrect associations

However, when modeling a real application, the diagram would very quickly become unmanageable. To counteract this, it might make sense to group use cases that have the same objective, namely the management of a course. This is reflected in Figure 3.17(b). The individual steps are then specified in the description of the standard process.

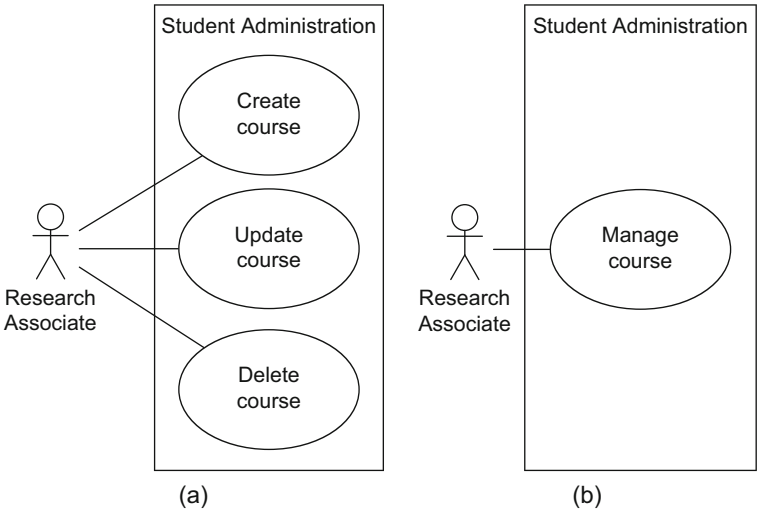


Figure 3.17
Modeling redundant use cases

3.7.4 A Final Example

Information system of the student office of a university

To conclude this chapter, we create a use case diagram that describes the functionality of the information system of a student office in accordance with the following specification:

- Many important administrative activities of a university are processed by the student office. Students can register for studies (matriculation), enroll, and withdraw from studies here. Matriculation involves enrolling, that is, registering for studies.
- Students receive their certificates from the student office. The certificates are printed out by an employee. Lecturers send grading information to the student office. The notification system then informs the students automatically that a certificate has been issued.
- There is a differentiation between two types of employees in the student office: a) those that are exclusively occupied with the administration of student data (service employee, or ServEmp), and b) those that fulfill the remaining tasks (administration employee, or AdminEmp), whereas all employees (ServEmp and AdminEmp) can issue information.
- Administration employees issue certificates when the students come to collect them. Administration employees also create courses. When creating courses, they can reserve lecture halls.

To create a use case diagram from this simplified specification, we first identify the actors and their relationships to one another. We then determine the use cases and their relationships to one another. Finally, we associate the actors with their use cases.

1. *Identifying actors*

   If we look at the textual specification, we can identify five potential actors: Lecturer, Student, employees of the types ServEmp and AdminEmp, as well as the Notification System. As both types of employees demonstrate common behavior, namely issuing information, it makes sense to introduce a common super-actor StudOfficeEmp from which ServEmp and AdminEmp inherit. We assume that the Notification System is not part of the student office, hence we include it in the list of actors. Figure 3.18 summarizes the actors in our example.

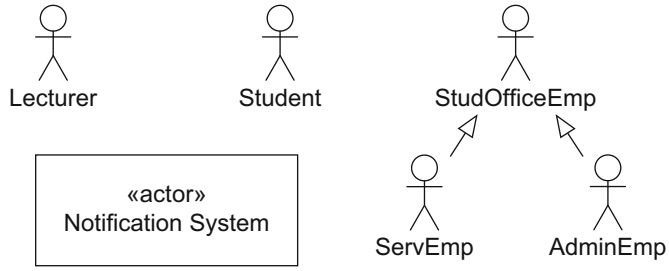


Figure 3.18
Identified actors

2. *Identifying use cases*

In the next step, we identify the use cases (see Fig. 3.19). In doing so, we determine which functionalities the student office must fulfill.

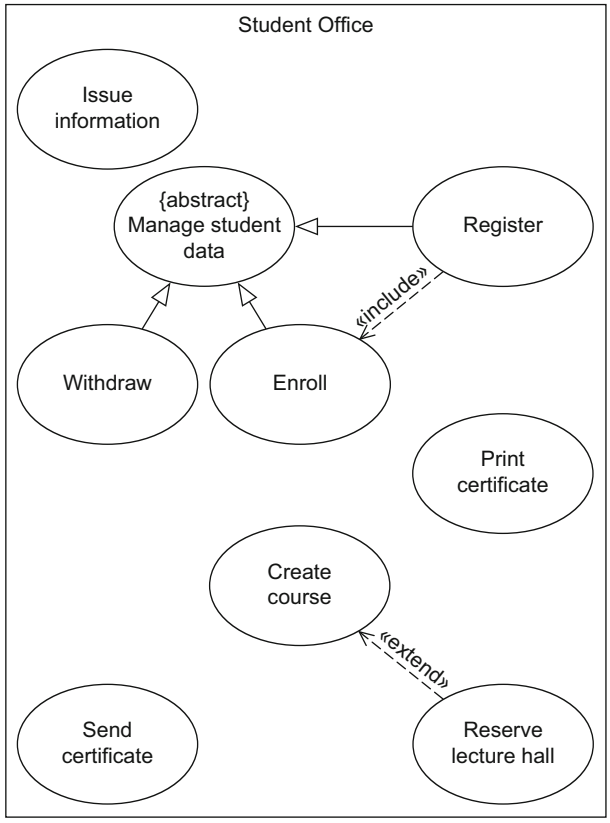


Figure 3.19
Identified use cases

The specification is very short. However, we know that the objective is to model the information system supporting the employees of a student office rather than modeling the functionalities the student office provides for the students. If we were to model the latter, we would need a use case Collect certificate, for example, in which a student would be involved. This use case is not included in the information system as it is not related to the collection of the certificates. The use case Print certificate is, however. To print, naturally we need a printer. Should we add this to our list of actors? We do not do this as we consider the printer to be an integral part of the system to be modeled.
We also have the functions Register, Enroll, and Withdraw. We could group these in one use case Manage student data as they are all performed by an actor ServEmp. In doing so, however, we would lose the information that matriculation includes enrollment for studies. Therefore, we do not reduce the three use cases to one use case. We express the relationship between Register and Enroll with an «include» relationship. As the three use cases have the association to ServEmp in common, we still introduce the use case Manage student data and model that the use cases Register, Enroll, and Withdraw inherit from this use case. To express that this use case cannot be instantiated, we define it as an abstract use case.
Lecturers can execute the use case Send certificate. If a certificate is sent to the student office, the student affected is notified. However, we do not model a separate use case Notify student as, according to the specification above, students are only notified in the context of the use case Send certificate. If Notify student cannot be executed independently, this activity is not a use case of the information system. Furthermore, we have the use cases Issue information, Reserve lecture hall, and Create course, where Reserve lecture hall extends the use case Create course. Figure 3.19 shows the resulting use cases.

3. *Identifying associations*

Now we have to associate our actors and the use cases (see Fig. 3.20). Note that we now have two fewer actors than potential candidates identified (see Fig. 3.18). There are no longer any students—students may not use the information system in the form that we have modeled it. And there is no longer a notification system as this is considered to be part of the student office.
Finally, we need a meaningful description of the use cases.

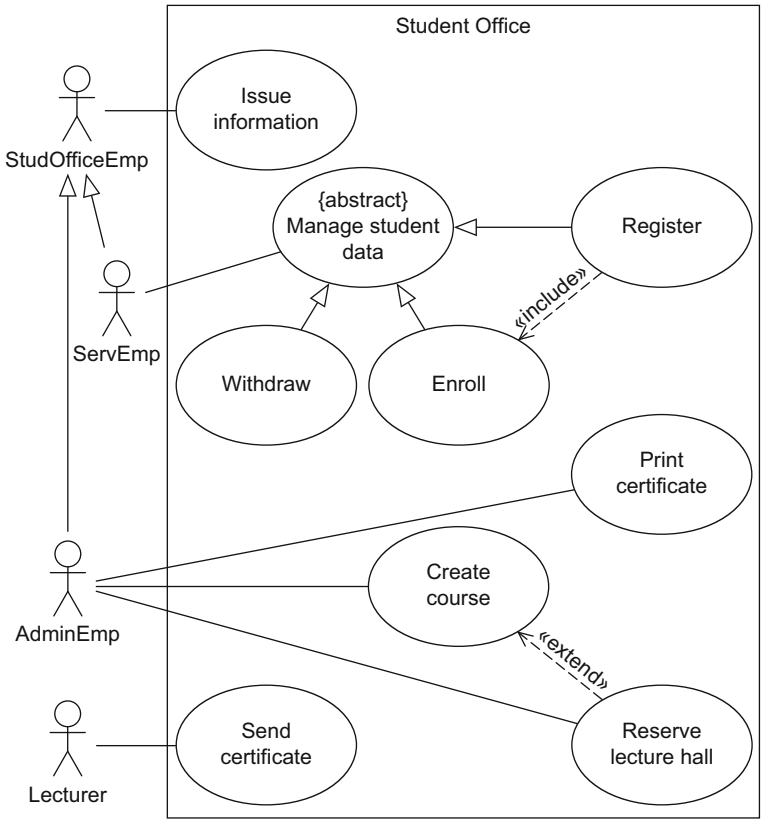


Figure 3.20
Use case diagram of the information system of the student office of a university

4. *Describing the use cases*

   Table 3.2 shows the description of the use case Print certificate as an example.

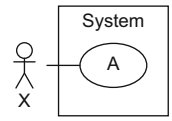
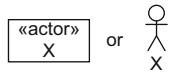
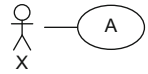
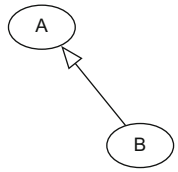
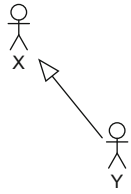
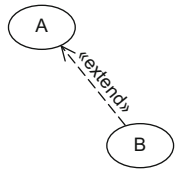
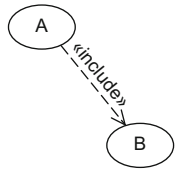
Table 3.2
Use case description for Print certificate

Name:	Print certificate
Short description:	On request from a student, an employee prints the student's certificate for a course on paper.
Precondition:	All data relevant for the certificate has been sent and the student has been graded.
Postcondition:	Certificate is available to the student in printed form.
Error situations:	Printer is not working.
System state in the event of an error:	Certificate is not printed.
Actors:	AdminEmp
Trigger:	Student requests printed certificate.
Standard process:	(1) Student enters the student office and requests a certificate. (2) AdminEmp enters the student's matriculation number. (3) AdminEmp selects the certificate. (4) AdminEmp enters the print command. (5) System confirms that the certificate was printed. (6) Certificate is handed over to the student.
Alternative processes:	(1') Student requests certificate via e-mail. (2-5) As above (6') Certificate is sent by post.

3.8 Summary

The use case diagram describes the behavior of a system from the view of the user. This means that this diagram presents the functionalities that the system offers but does not address the internal implementation details. The boundaries of the system—what can the system do and what can it not do?—are clearly defined. The users (actors) are always outside the system and use the functionalities of the system, which are depicted in the form of use cases. The relationship between a use case and an actor is referred to as an association. To keep use case diagrams as compact as possible, generalization is supported for both actors and use cases, which allows the extraction of common properties. Use cases can also access the functionality provided by other use cases by means of «include» and «extend» relationships. The most important notation elements are summarized in Table 3.3.

Table 3.3
Notation elements for the use case diagram

Name	*Notation*	*Description*
System		Boundaries between the system and the users of the system
Use case		Unit of functionality of the system
Actor		Role of the users of the system
Association		X participates in the execution of A
Generalization (use case)		B inherits all properties and the entire behavior of A
Generalization (actor)		Y inherits from X; Y participates in all use cases in which X participates
Extend relationship		B extends A: optional incorporation of use case B into use case A
Include relationship		A includes B: required incorporation of use case B into use case A