Software Engineering Courses

Level: 3 Licence (bachelor) Computer Science

Semestre1

Outlines

Section 1. : Object Oriented Paradigm – Reminder.

1. **Introduction / Definition**
2. **Object**
3. **Class: attribute, method, constructor, modifiers, inheritance, overriding, overloading, …**
4. **Abstract class, interface**
5. **Template (Parametrized class)**
6. **Relationships between classes: association, dependency, template binding, …**

Section 2.: Unified Modeling Language (UML).

1. **What is UML?**
2. **Aims of UML**
3. **UML Model**
4. **UML Diagrams: Use case, state charts, activity, Class, …**
5. **Limits of UML**
6. **UML profile (Extension of UML)**
7. **OCL**

Section 3.: Software Engineering.

1. **Introduction – Basic concepts**

Where we find a software?
Positive impacts of a software
Negative impacts of a software
What is a Software?
Software and Hardware
Components of software
Comparison with industrial products
Software types

Section 4. Software development

How does software development starts

Characteristics of good software (customer point of view)

Characteristics of good software (supplier point of view)

What we should know about software development

What we need to develop a software

2. Software Engineering

Definition

Aims

Components

Section 5. Software development methodologies

Introduction

Tasks, Tools and Professions

Development methodologies:

- **Traditional methods**
- **Agile methods**
- **Unified Process (UP)**

Section I

Object Oriented Paradigm Concepts – Reminder.

1.1. Introduction

Object-Oriented software can be seen as a set of objects that cooperate and interact with each other in order to achieve the software desired task. In real Word, each autonomous entity can be seen as an object. Objects communicate through message sending mechanism. The objects that share the same proprieties belongs to the same Class. In order to facilitate software architecture and design, the OOP introduces several functionalities and relationships between software components.

1.2. Object

Did you write programs with C, Pascal, … ???

What is the difference between Java/C++/C# and C/Pascal ???

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
float x1, x2, delta, a, b, c;
printf("Entrez les valeurs des: a, b, c \n");
scanf("%f %f %f",&a, &b, &c);

delta = (b*b)-(4*a*c);

 if (delta == 0.0)
  {
   x1 = -b /(2*a);       printf("la solution unique est xs = %.2f \n",x1);
  }

 if (delta > 0.0)
 {
  x1 = (-b - sqrt(delta))/(2*a);        x2 = (-b + sqrt(delta))/(2*a);
  printf("les deux racines sont : x1 = %.2f et x2 = %.2f \n",x1, x2);
 }

 if (delta < 0.0)    printf("l'equation n admet pas de solution");

return 0;
}
```

Objects are components that reflect real world entities. In a classroom, the following entities are objects: teacher, student, table, chair, room, etc. The set of objects that share the same proprieties are produced (instantiated) from the same **class**.

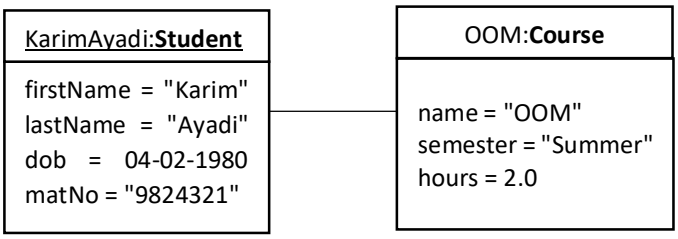


Fig 1.1: Example of an Object Diagram

1.3. Class

The class is the blueprint that generates objects (**instances**). It is a structure that groups together objects properties (**attributes**) and operations (**methods**) to manipulate these properties.

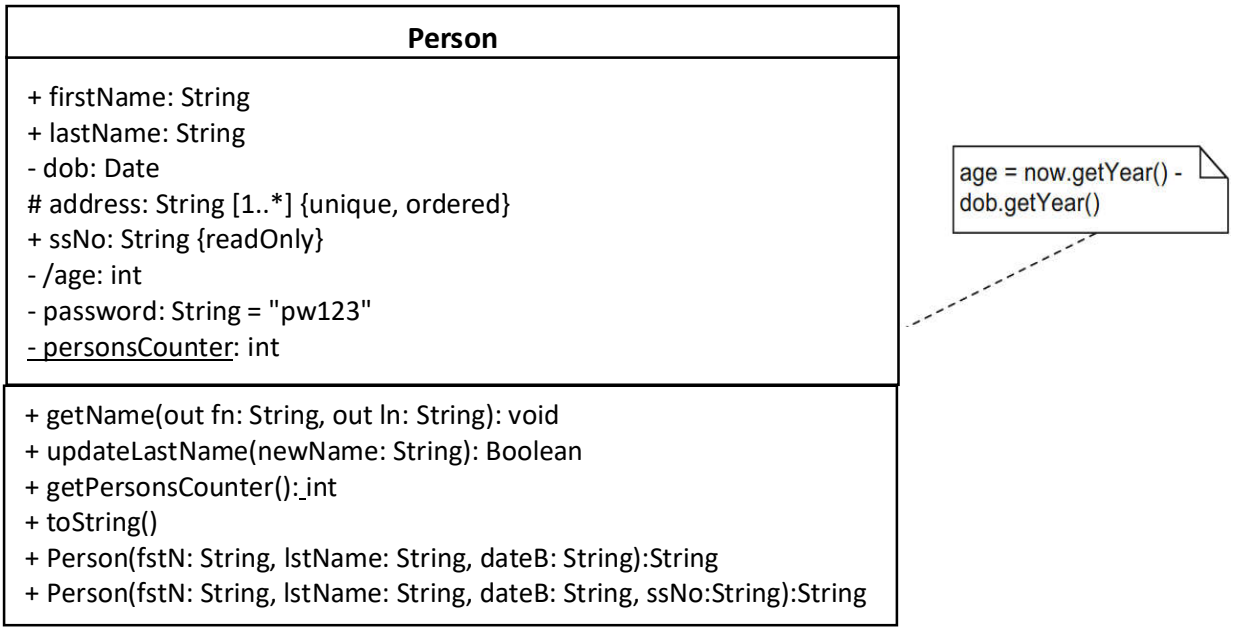


Fig 1.2: Example of a Class with attributes and methods

An object (class instance) is a **memory state** (representation) that contains values of each of the class' attributes. Any modification on the object corresponds to a modification on the memory sate (memory area) representing the object.

Attributes (instance variables / properties) are named entities with their datatypes (see Fig **1.2**). An attribute should be specified with one of the following visibility levels (modifiers): Public, Private, Protected and Package:

Access Right	public (+)	private (-)	protected (#)	Package (~)
Members of the same class	Yes	yes	yes	Yes
Members of derived (inherited) classes	Yes	no	yes	Yes
Members of any other class	Yes	no	no	in same package

Static members (attributes) of a class can be declared using the storage class modifier **static**. These data **members** are shared by all instances of this class and are stored in one place (one copy for the class). Contrary to non-**static** data **members** which are created for each class object **variable**. Static attributes are underlined, just like static operations (see the attribute **personCounter** of the class Person, Fig 1.2).

Data Encapsulation means that class data (instance variables) are protected against any non-authorized access or modification. We only could access and modify class data through class **methods**.
For example, let P an object of the class Person (Fig 1.2). Outside the class, we could get directly the last name of P: P.lastName. But, P.dob is incorrect, since the attribute dob is private.

Methods are computational bodies having **parameters** (**arguments**). Method signature (type) specify the datatypes of method (in/out) parameters. Methods have the same visibility levels as attributes. There is a particular method called **class constructor.** It has the same name as the class. It is used to create the instances of the class.

Polymorphism is the ability given to the same operation to behave (to act) differently according to the list of its arguments and to the context of the class in which it is found. We distinguish between two essential forms of polymorphism: **overriding** and **overloading.**

Methods overriding allows a subclass (through inheritance feature) to provide a specific implementation of a method that is already provided by one of its super-classes. The method toString() of the class Person should be overridden in the class Student (that inherits Person) in order to print the matNo (the registration number) of the student, in addition to person attributes.

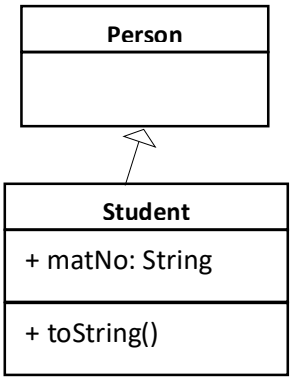
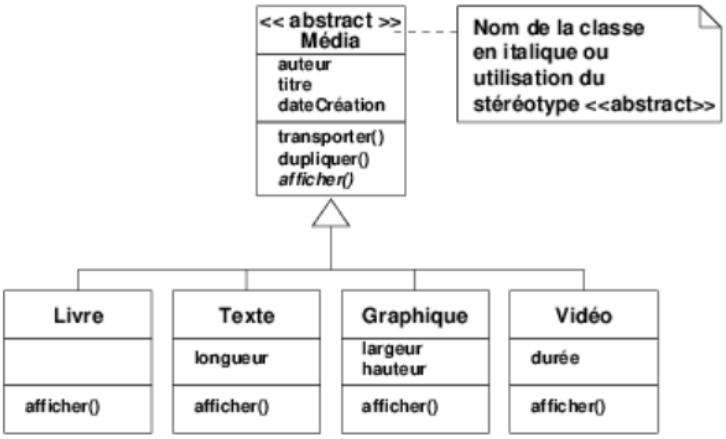


Fig 1.3: Example of redefined method through inheritance ' overriding'

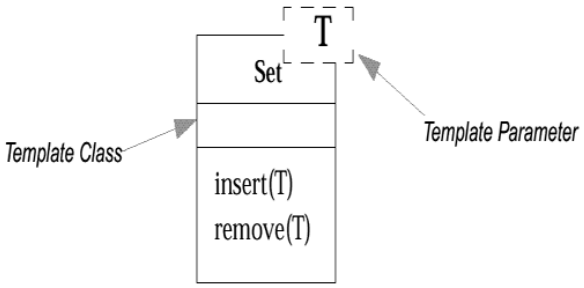
Methods overloading allows a class to have more than one method having the same name, but with a different argument lists, see the example of the class Person constructor (fig 1.2).

Interface of a class: It is a structure composed of methods types (signatures). One class is operated (exploited) through its interface, like the dashboard of a car.

Abstract class is a **class** that contains at least one **abstract** method. An **abstract** method is a method that is provided without its computational body, only the signature of the method is provided. It is not possible to create instances from abstract classes. An abstract class could be seen as a general template which may be specialized to generate concrete classes through the inheritance feature.



Template classes are parameterized classes specified with a list of formal parameters. It is used to model a generic class which could be specialized by substituting its formal parameters with actual ones. The aim of templates is to make one template useful to generate different specializations.



1.4. Relationships between classes:

Binary Association between two classes is either a strong relation (**association**) or a week relation (**dependence**) between two classes.

Association is a structural relation between classes' instances. That is one class uses the second class as type of its attributes.

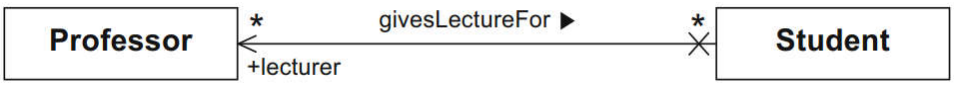


Fig 1.3: Binary association

A binary association is characterized by:

1. Reading direction: ▶
2. Navigability: > / ><
3. Multiplicity *, 0..*, 1..*, …
4. Role: +lecture

Figure 1.4 shows a comparison of binary association between UML and Java. Be careful, you should distinguish between **Conceptual** and **implementation** levels.

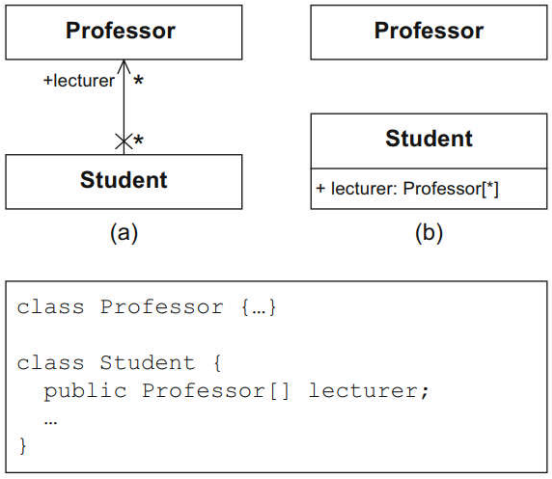


Fig 1.4: Associations in UML and Java

Figure 1.5 presents other examples of binary association with multiplicities:

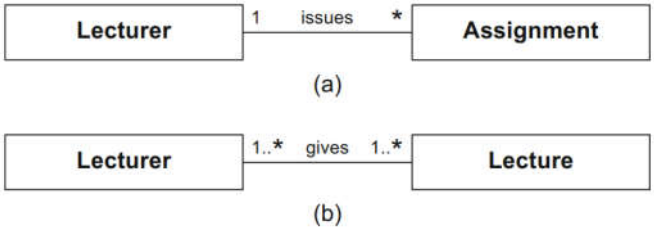


Fig 1.5: Associations with different multiplicities

Association Classes

In UML diagrams, an association class is a class that is part of an association relationship between two other classes. You can attach an association class to an association relationship to provide additional information about the relationship. An association class is identical to other classes and can contain operations, attributes, as well as other associations.

The next figure shows examples of an association classes.

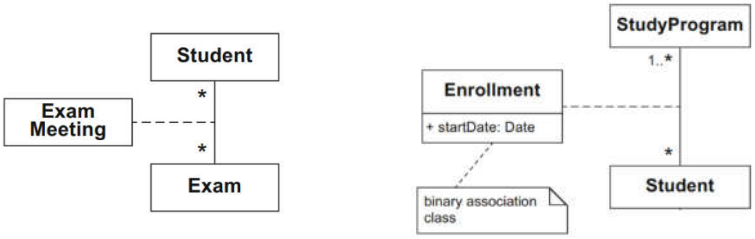


Fig 1.6: Association classes

Aggregation:

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.

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 (see example in Fig 1.7).

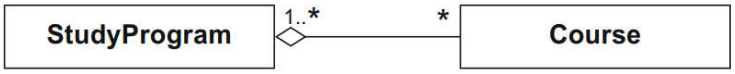


Fig 1.7: Example of a shared aggregation

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 (see Fig 1.8).

Fig 1.8: Example of a composition

Dependency expresses that the method of one class uses another class as type of its parameters. The next figure show an example of a dependency; the class **Cercle_origine** uses the class **Point** to type the parameter p of the method *appartient*.

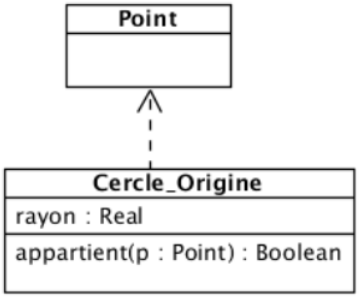


Fig 1.8.1: Example of Dependency relationship

Inheritance is the mechanism that enables a new subclass to acquire all the attributes, methods, and properties of its super-classes. In addition, the subclass can define its own attributes and methods, as it can redefine attributes and methods already defined in super-classes. Note that **multiple inheritance** is considered as an extension of the simple inheritance model, where a class is allowed to have several super-classes.

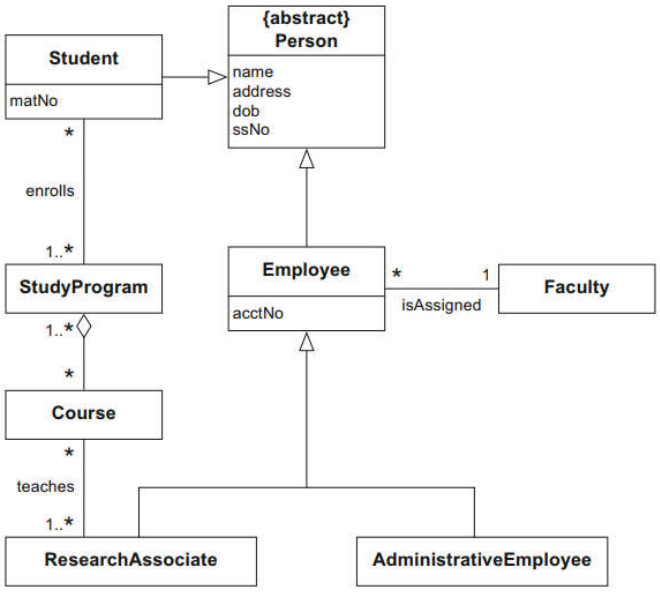


Fig 1.8: Example Inheritance relatioship

Template binding: Template **(**parameterized class) cannot be used in the same way as a non-parameterized class. It is only used to create new classes by substitution (or partial substitution) of its formal parameters by actual ones (**template binding**) or to be inherited by a new template of the model. A parameterized class cannot be used to create instances, or type an attribute or type an operation parameter of another class

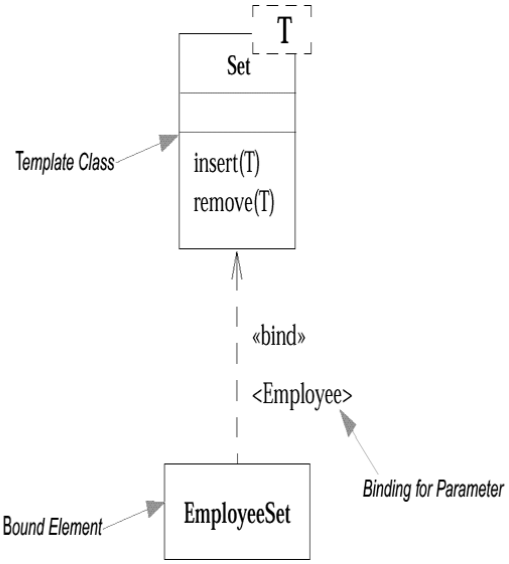


Fig 1.9: Example UML template and template binding

2. Unified Modeling Language (UML).

2.1 Introduction

The development of computer applications began with the invention of the Von Neumann machine in 1945. Along with technological developments, several programming languages have appeared. We find procedural programming languages (imperative paradigm) like FORTRAN (since 1954), Pascal (since 1970) and C (since 1972). In order to avoid side e effects when running programs and to give more rigor to software products, languages of functional paradigm such as Lisp (since 1960) and ML (since 1980) have also been developed.

Then (in the 1980s), the need to facilitate the design, maintenance and reliability of programs led to the invention of the object-oriented paradigm. The latter considers the programs in terms of object classes grouping together data structures and methods. Object paradigm languages include languages such as C ++ (since 1983) and Java (since 1995). Currently, most programming languages are influenced by the object paradigm.

With industrial developments, computer applications have become very large (millions of lines of code), which makes checking and maintaining codes very difficult. The classic method of code writing (used in the 1970s and 1980s) has become insufficient.

Indeed, building a small house is not like building a huge building. The latter needs a detailed architecture and design before starting its realization. Thus, several methods of planning and design of computer applications are appeared in the early 90s. The three most common methods are: Grady Booch method [Boo91], the OOSE method [JCC94] of Ivar Jacobson, and the OMT method [RBL + 90] of James Rumbaugh. In 1995, these three methods were unified into a single standard called UML (Unified Modeling Language).

UML allows the modeling and the design of a systems using a set of graphical diagrams. Some UML diagrams enables to model the static aspect and others are used to model the dynamic aspect of systems. As a successor of the aforementioned methods, UML quickly became a standard for OMG (Object Management Group). It enjoys great popularity, both in industrial software and in academic circles. This popularity is enhanced by a significant number of tools, based on UML notations, in software engineering workshops.

2.2 UML Model and UML Diagrams

A UML model is the design description of a real system using UML notations. UML allows the description of a model through different views. Each view is represented by one or more **diagrams**. A UML diagram is represented by a connected graph where summits are the elements and the arcs are relations.

A single diagram generally represents only one aspect of the designed system. The current version of UML (UML 2.5) uses up to 23 different diagram types ([OMG15a]) to describe the structural aspect (describing the components of a system and their relation without taking the time factor into account) and the behavioral aspect (shows the behavior of the system, the interactions of objects and their evolution over time) of a system. The most used diagrams (those of the UML2.4 version) are summarized in Fig 1.

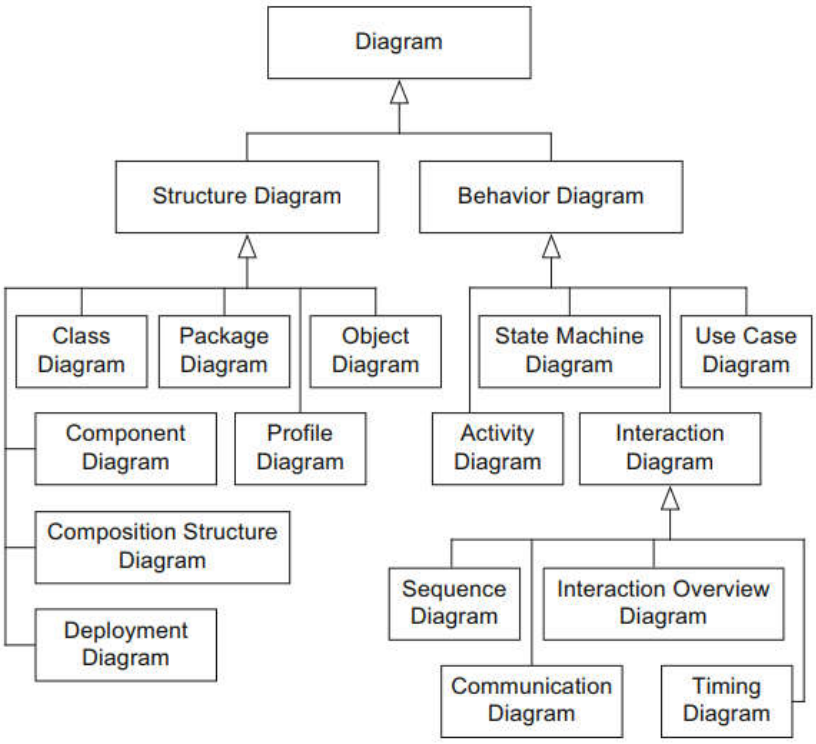


Fig 2.1: UML Diagrams

This course restricts itself to the five most important and most widespread types of UML diagrams, namely the use case diagram, class diagram (including the object diagram), state machine diagram, sequence diagram, and activity diagram. We present these diagrams in the order in which they would generally be used in software development projects. We begin with the use case diagram, which specifies the basic functionality of a software system. The class diagram then defines which objects or which classes are involved in the realization of this functionality. The state machine diagram then defines the intra-object behavior, while the sequence diagram specifies the inter-object behavior. Finally, the activity diagram allows us to define those processes that “implement” the use cases from the use case.

2.3 The 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. 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.

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.)

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.

Use cases are determined by collecting customer wishes and analyzing specified. 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 2.2. 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

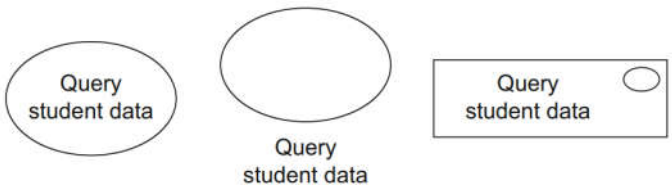


Fig 2.2: Alternative Notations for use cases

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. 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 2.3. These three notation alternatives are all equally valid. As we can see from this figure, 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.

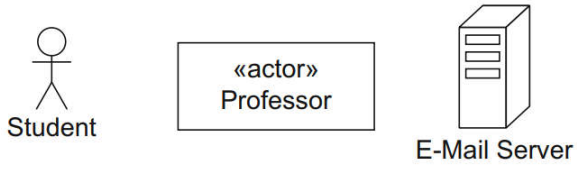


Fig 2.3: Notation alternatives for actors

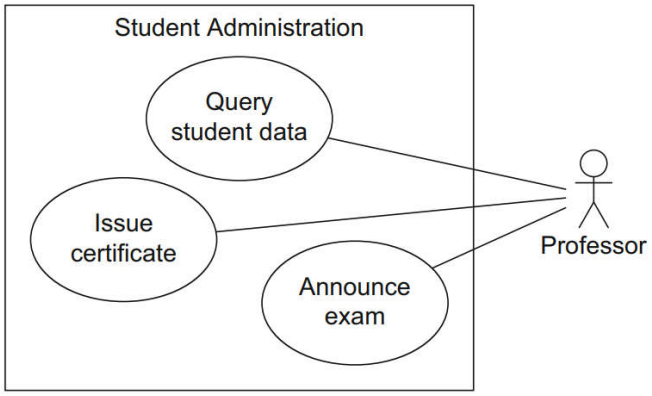


Fig. 2.4 Representation of system boundaries

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 2.4 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.

Types of actors: active/passive and 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 2.5, 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 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 2.5, 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

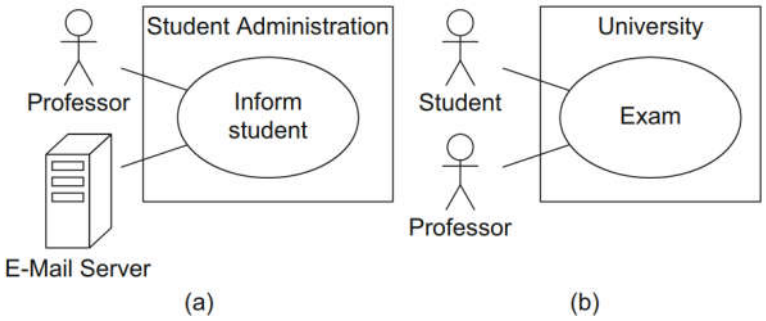


Fig. 2.5 Examples of actors

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. Sometimes it is difficult to decide whether an element is part of the system to be implemented or serves as an actor. In example (a) in Figure 2.5, 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.

Associations of Actors

An actor is connected with the use cases via associations which express that the actor communicates with the system and uses a certain functionality.

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.

Note that a 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.

Generalization (Inheritance) for actors

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 2.6, an actor Professor is required; in addition, an actor Assistant can be involved optionally, which is expressed by the multiplicity 0..1.

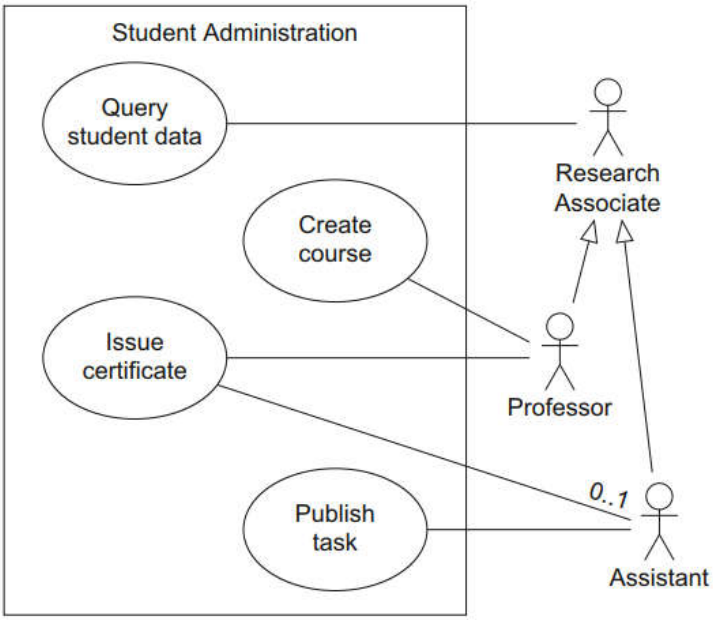


Fig. 2.6 Example of actors Inheritance

Abstract actors

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 2.7 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 regrouped and described at one point, the abstract super-actor.

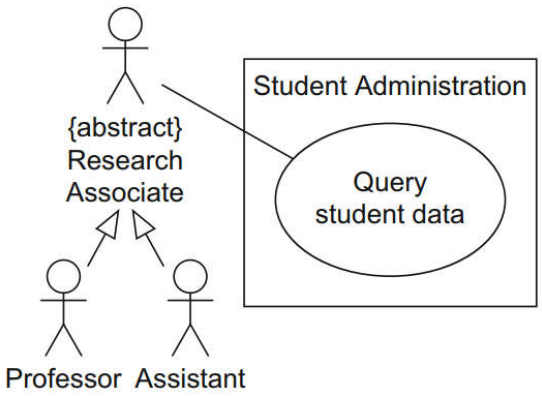


Fig. 2.7 Abstract Actors

Relationships between Use Cases

Use cases can also be in a relationship with other use cases. Here we distinguish between «include» relationships, «extend» relationships, and generalizations of use cases.

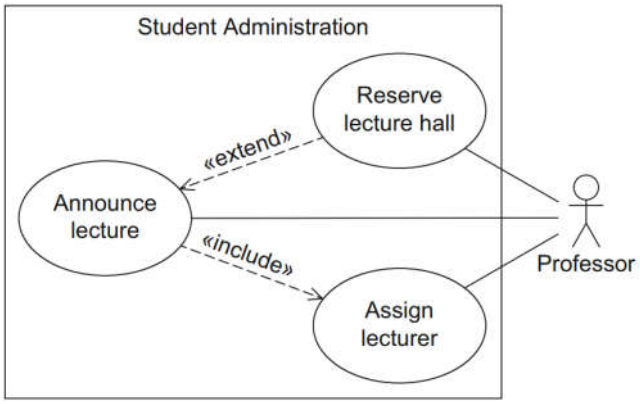


Fig. 2.8 Example of «include» and «extend»

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. In the use case diagram in Figure 2.8, 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 2.8, 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.

Condition and Extension Point

A condition that must be fulfilled for the base use case to insert 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 2.9. 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.

We 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 section 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.

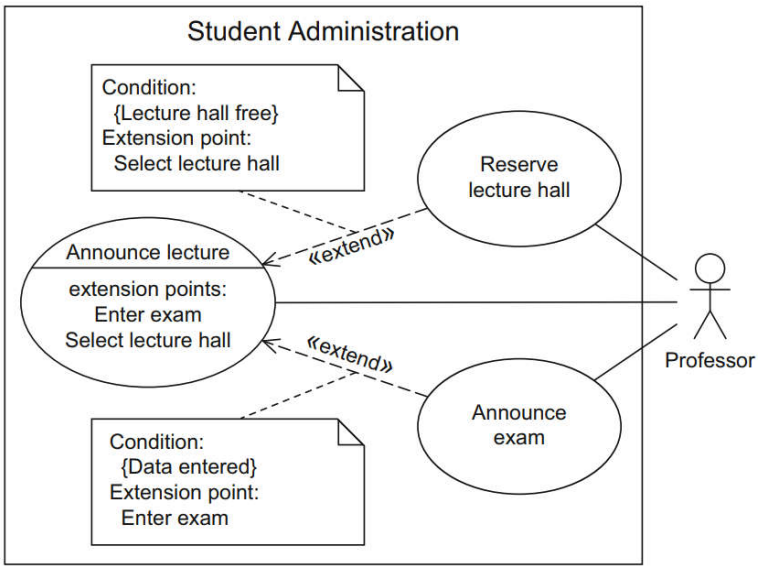


Fig. 2.9 Example of Extension Point and Condition

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.

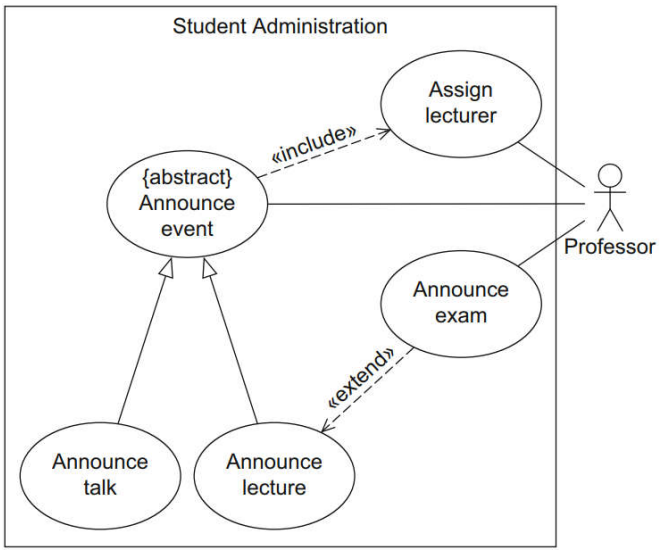


Fig. 2.10 Example of generalization of use cases

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.

Exercise 01: Describe (read correctly) the use case diagram presented in Fig 2.10.

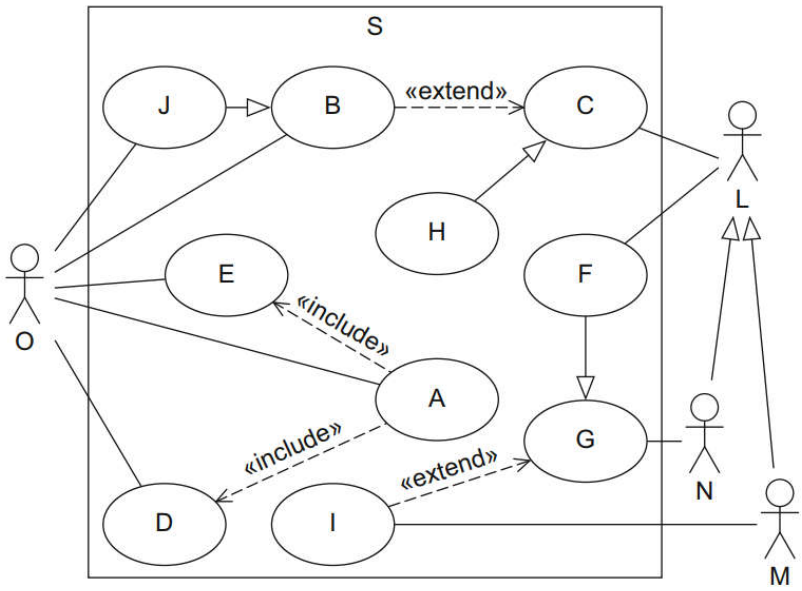


Fig. 2.10 Example of a use case diagram with relationships

Describing Use Cases

Alistair Cockburn[1] presents a structured approach for the description of use cases that contains the following information:

- Name
- Short description
- Pre-condition: prerequisite for successful execution
- Post-condition: 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 2.1 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.

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 2.1 description of the use case Reserve lecture hall

[1] A. Cockburn. Goals and Use Cases. *Journal of Object-Oriented Programming*, 10(5):35–40, 1997.