

In the previous lecture...

- Software life cycles
 - The importance of iterative development
- Requirement elicitation
 - An interdisciplinary effort
 - Functional vs. non-functional requirements
- In need of a formal language that are widely accepted

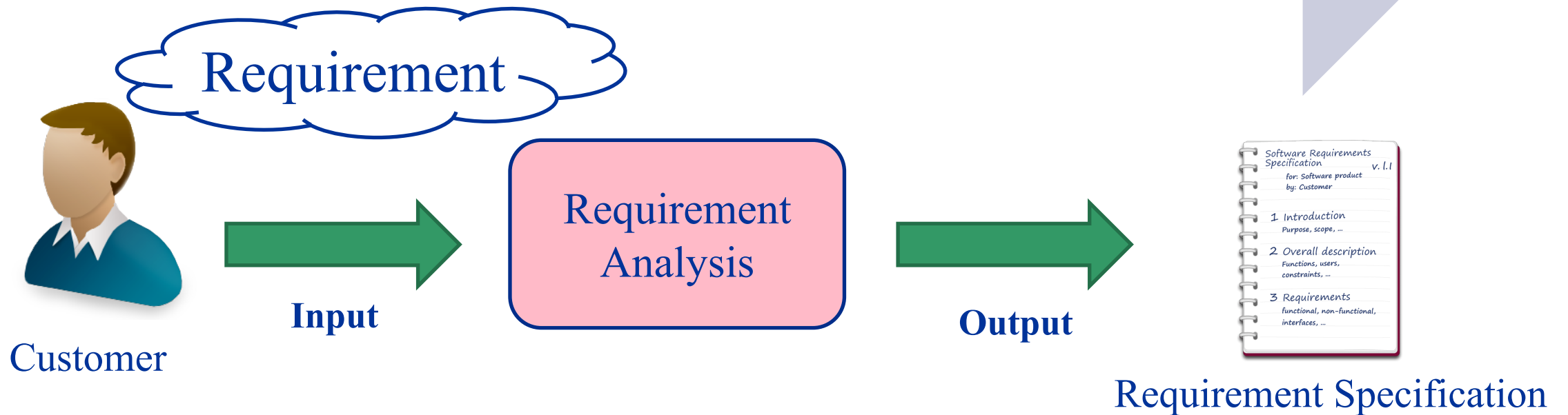
Lecture 8: OO Design and Use Case Diagram



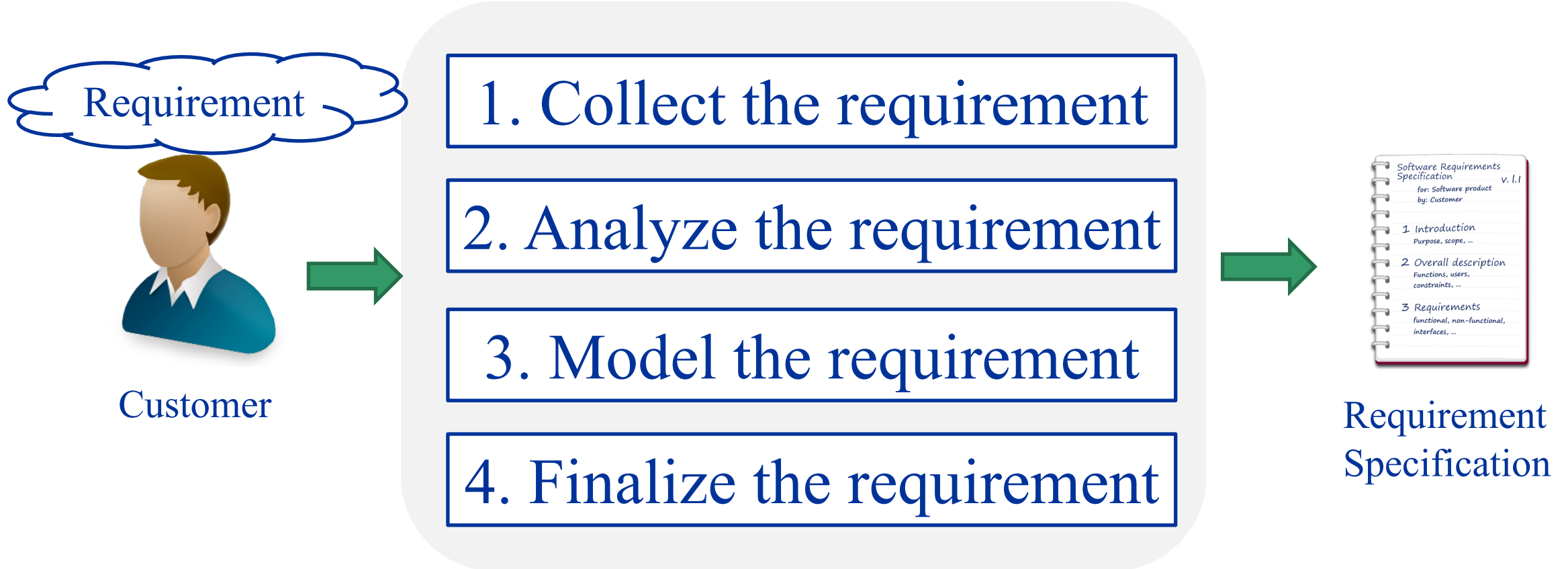
Outline

- Procedure-oriented Programming
- Object-oriented Programming
- Procedure-oriented Programming vs OO Programming
- UML
- Use-case Diagram

Software Development Life Cycle



Requirement Analysis



System Modeling

- System modeling is the process of **developing abstract models of a system**, with each model presenting a different view of perspective of that system.
 - Requirement Engineering
 - System Design
 - ...

How to model the system?

Procedure-Oriented Software Design

- Describe problems in terms of functions: $y=f(x)$
- Behaviors hard to describe as procedure



Procedure-Oriented Software Design

- Sensitive to requirement changes
- Nothing reusable
- Less intuitive (Communication problems)
- No information hiding

```
graduate()
{
    returnCafe();
    dropClass();
    returnBook();
}
```



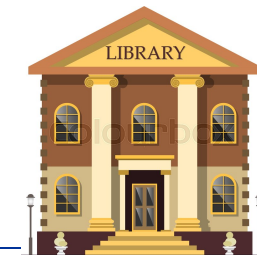
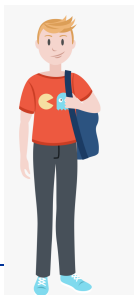
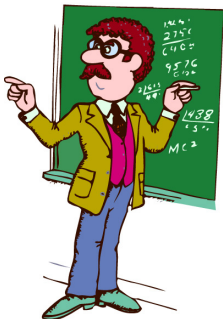
deposit()

deposit()

Joe	202001	\$100	Yes
Jane	202002	\$200	No

Joe	202001	CS132
Jane	202002	CS233

Joe	202001	Book 1
Jane	202002	Book 2



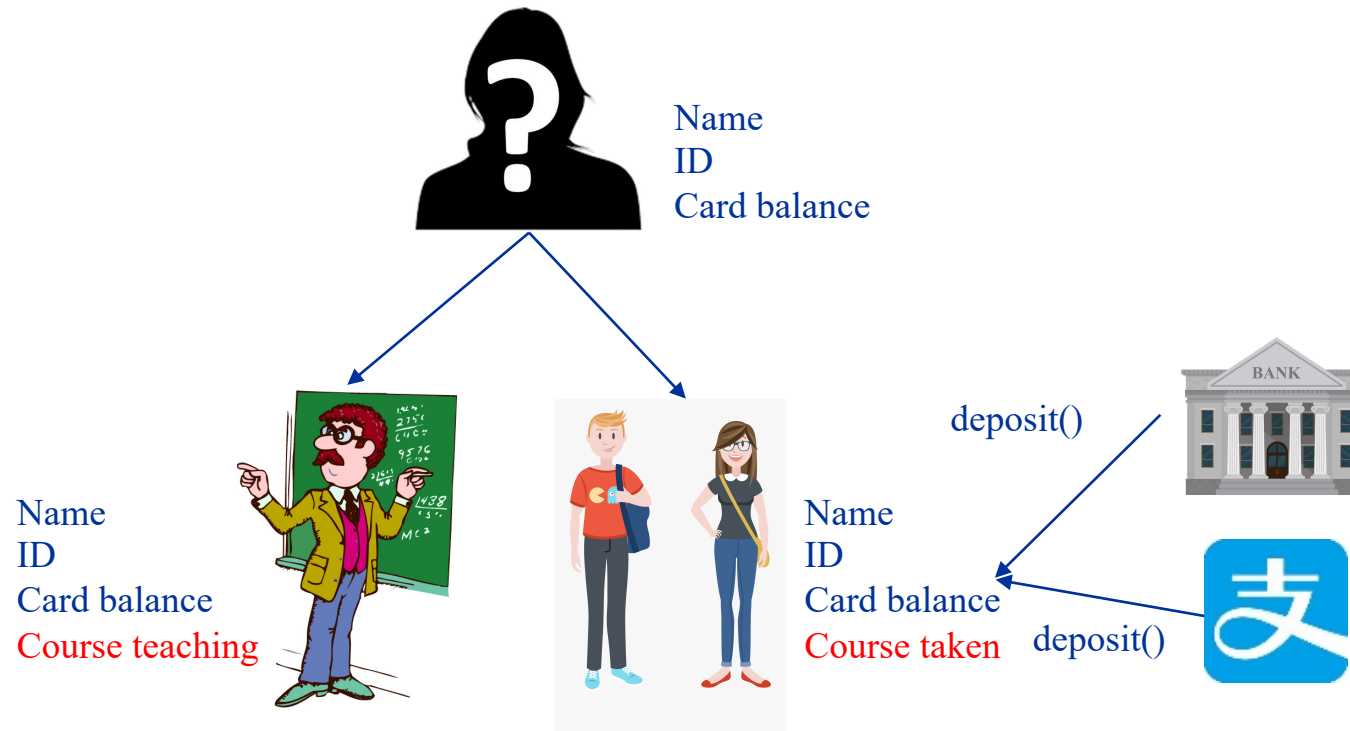
Engineering

DIY Community in Electrical Engineering

- Standardized “building blocks”
 - Easily accessible
- Standardized interface
 - Interchangeable components
- Can we define a software system as a collection of objects of various types that interact with each other through well-defined interfaces?

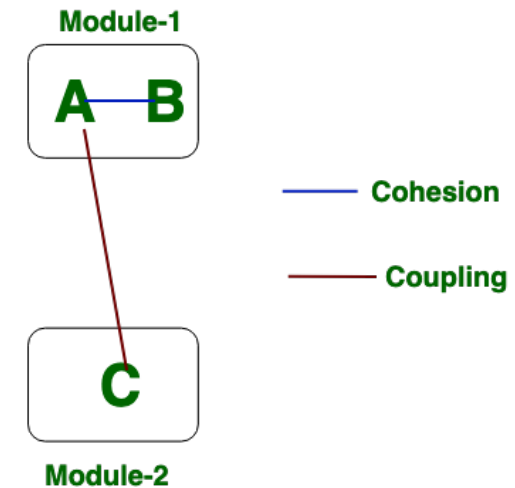
Object-Oriented Software Design

- Describe problems as objects and interactions between objects
- Much more intuitive



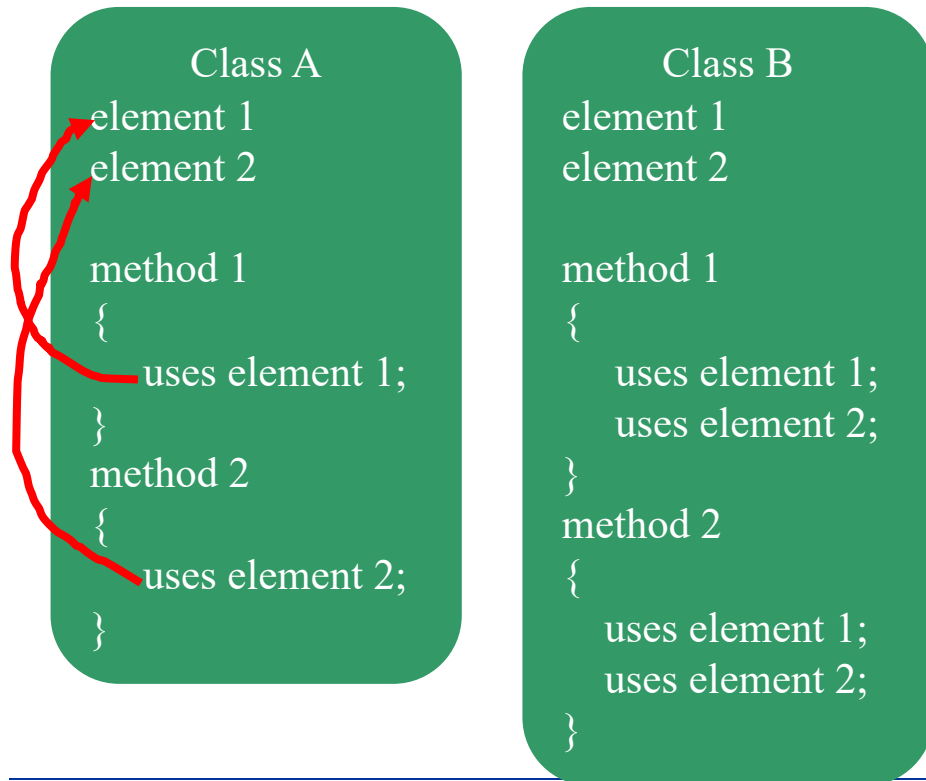
Benefits of OO

- Modularity: Decompose a system into a set of **cohesive** and **loosely coupled** modules
 - Reusability
 - Accidental vs. deliberate reuse
 - Encapsulation and information hiding
 - Interfaces
 - Access levels
 - Reduce coupling
- Inheritance: a relationship between different classes in which one class shares attributes of one or more different classes

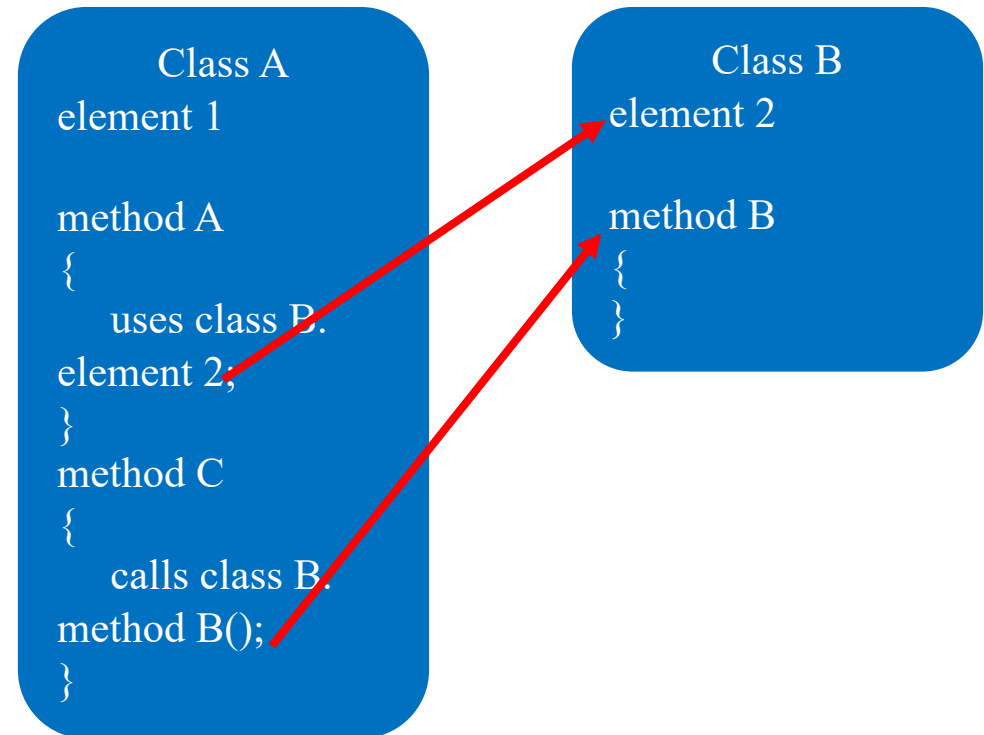


Cohesion vs. Coupling

- Low vs. high cohesion



- Tight Coupling (avoid)



Design Choices

- A method of an object may only call methods of:
 - The object itself.
 - An argument of the method.
 - Any object created within the method.
 - Any **direct** properties/fields of the object.
- **Don't talk to strangers!**
- When one wants a dog to walk, one does not command the dog's legs to walk directly; instead one commands the dog which then commands its own legs.

System Modeling

- System modeling is the process of **developing abstract models of a system**, with each model presenting a different view of perspective of that system.
 - Requirement Engineering
 - System Design
 - ...
- System modeling now usually means representing a system using some kind of graphical notation based on diagram types in the **Unified Modeling Language (UML)**.
- UML is for OO Design

History of UML

- Resulted from the convergence of notations from three leading object-oriented methods (The Three Amigos)
 - OMT (James Rumbaugh)
 - OOSE (Ivar Jacobson)
 - Booch (Grady Booch)
- 1995 Unified Method 0.8
- 1997 Unified Modeling Language 1.0
 - Object Management Group (OMG)
- Currently at UML 2.5.1+
- Has a very good eco-system and is still evolving

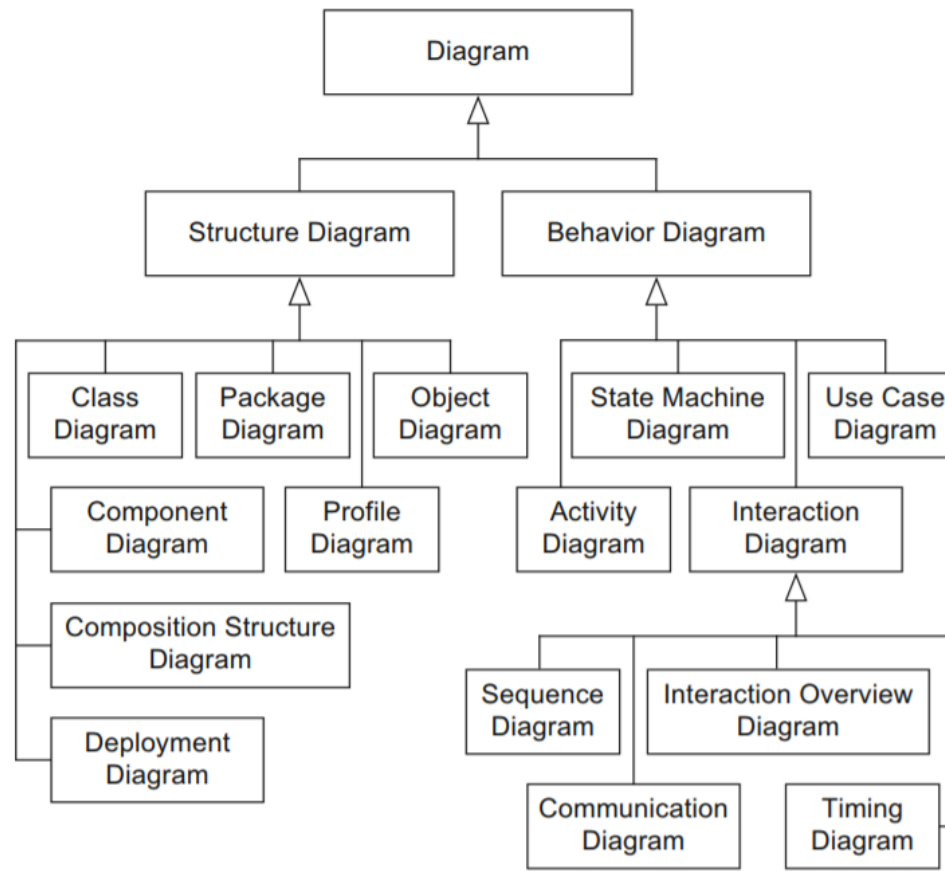
UML Partners

- IBM
 - NEC
 - Oracle
 - ...
-
- Very impressive list
 - Specialized to solve problems in different domains

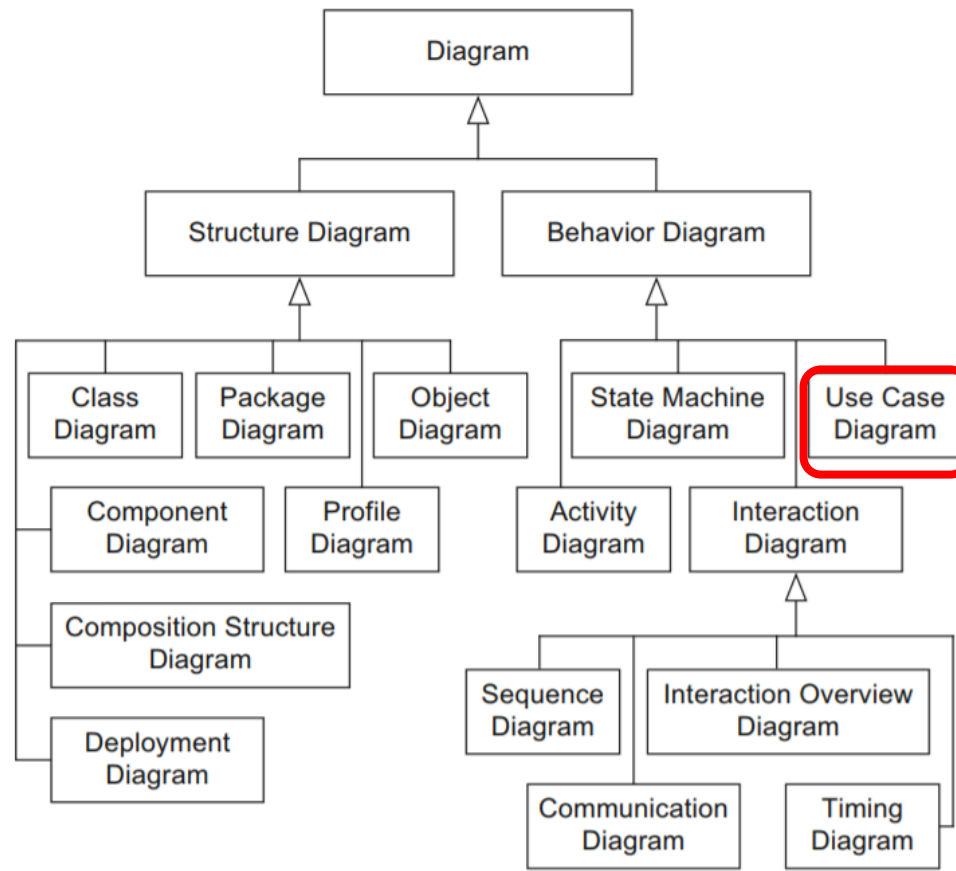
Unified Modeling Language (UML)

- Visualizing and documenting **analysis** and **design** effort.
- Unified because it ...
 - Combines main preceding OO methods (Booch by Grady Booch, OMT by Jim Rumbaugh and OOSE by Ivar Jacobson)
- Modelling because it is ...
 - Primarily used for visually modelling systems. Many system views are supported by appropriate models
- Language because ...
 - It offers a syntax through which to express modelled knowledge

UML Diagrams



Use Case Diagram

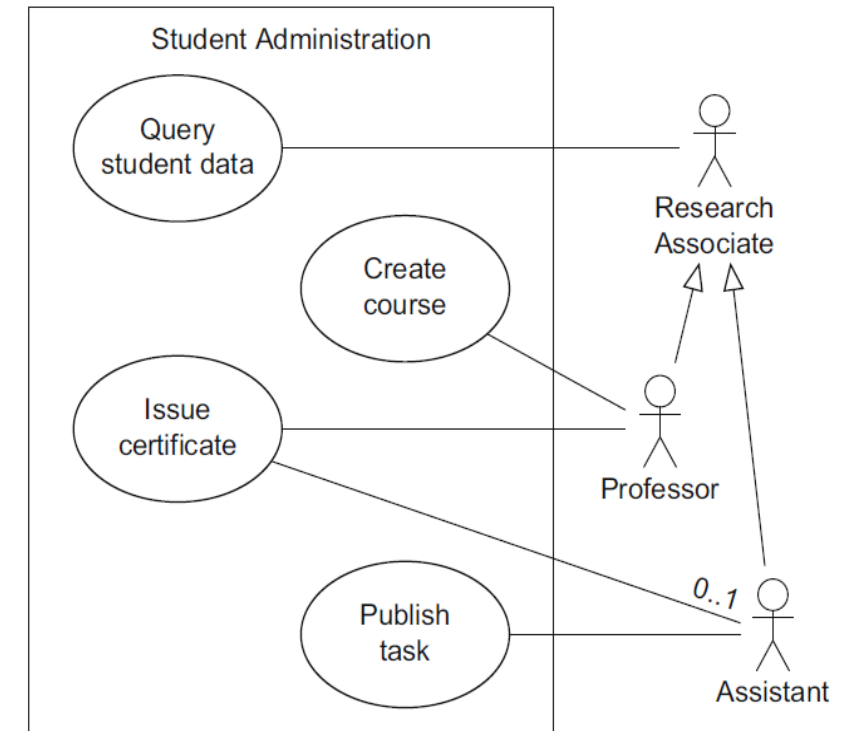


Use Case Diagram (2)

- Use case modeling was originally developed by Ivar Jacobsen in 1990s, and a UML diagram type to support use case modeling is part of the UML.
- A use case can be taken as a simple description of **what a user expects from a system in that interaction. (Functional)**
- Who's interacting with the system?
 - Most used for modelling interaction between the system and *external agents* (i.e. human user or other systems).
- How they are using it?

Why Use Case Diagram?

- A purpose of use case diagram is to **capture core functionalities of a system and visualize the interactions** of various things called as actors with the use case.
- Requirement (functions)
- Who's interacting with the system? (Actors)
- How they are using it? (Use cases)
- Define the boundary of the system



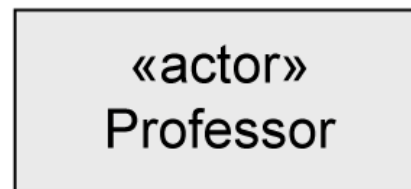
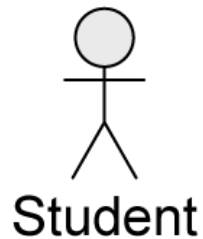
Use Cases

- A use case can be taken as a simple description of what a user expects from a system in that interaction. (Functional)
- Use case
 - Verb + Noun
 - i.e. Check deposit
- Use case needs to benefit the actor
 - i.e. Withdraw money is a use case, fill in the withdraw form is not



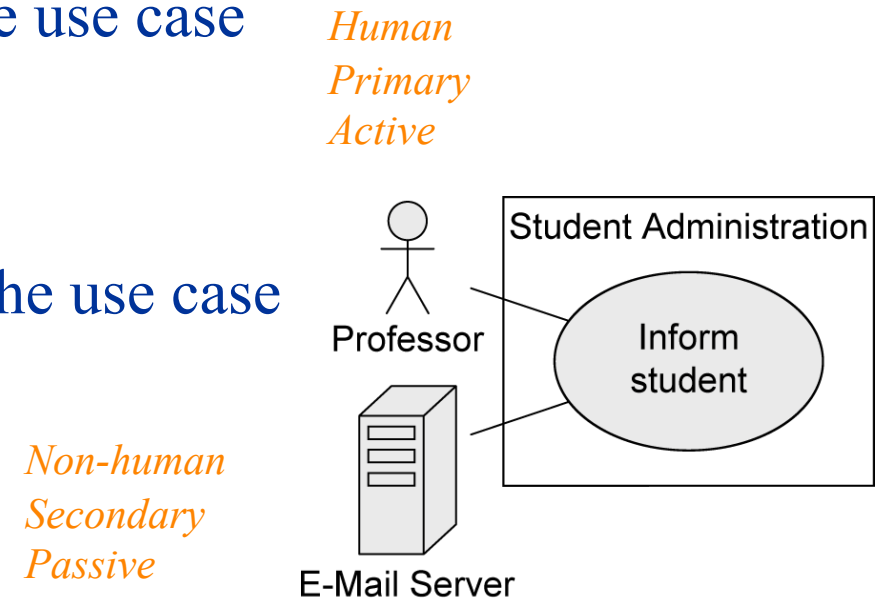
Actor

- Actor
 - Don't confuse with stakeholder
 - Not all stakeholders are actors in certain use cases
 - Not necessarily human. i.e. server, other systems
- Actors represent **roles** that users adopt.
 - Specific users can adopt and set aside multiple roles simultaneously.
- Actors **are not part of the system**, i.e., they are **outside** of the system boundaries.



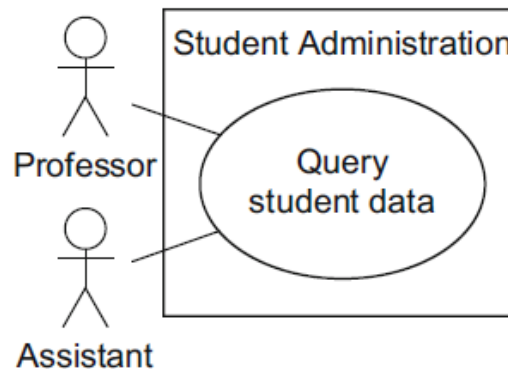
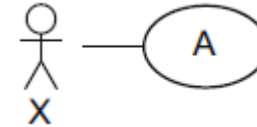
Actor (2)

- **Human**
 - E.g., **Student, Professor**
- **Non-human**
 - E.g., **E-Mail Server**
- **Primary**: has the main benefit of the execution of the use case
- **Secondary**: receives no direct benefit
- **Active**: initiates the execution of the use case
- **Passive**: provides functionality for the execution of the use case



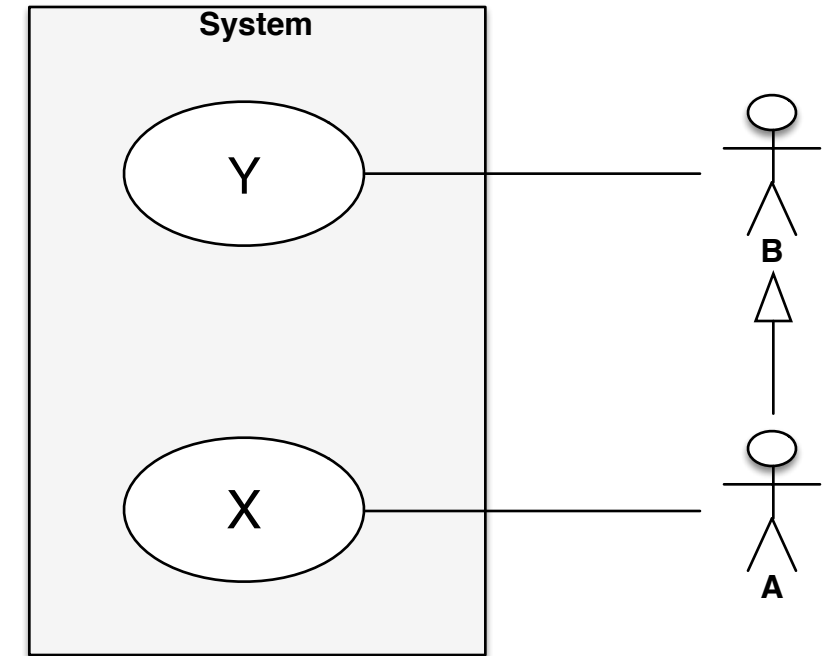
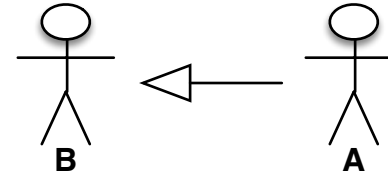
Associations

- Associate actors with use cases
- Every actor should interact with at least one use case
- Every use case should have at least one associated actor
- Actors are outside of system boundary
- Actors are connected with use cases via solid lines (*associations*).



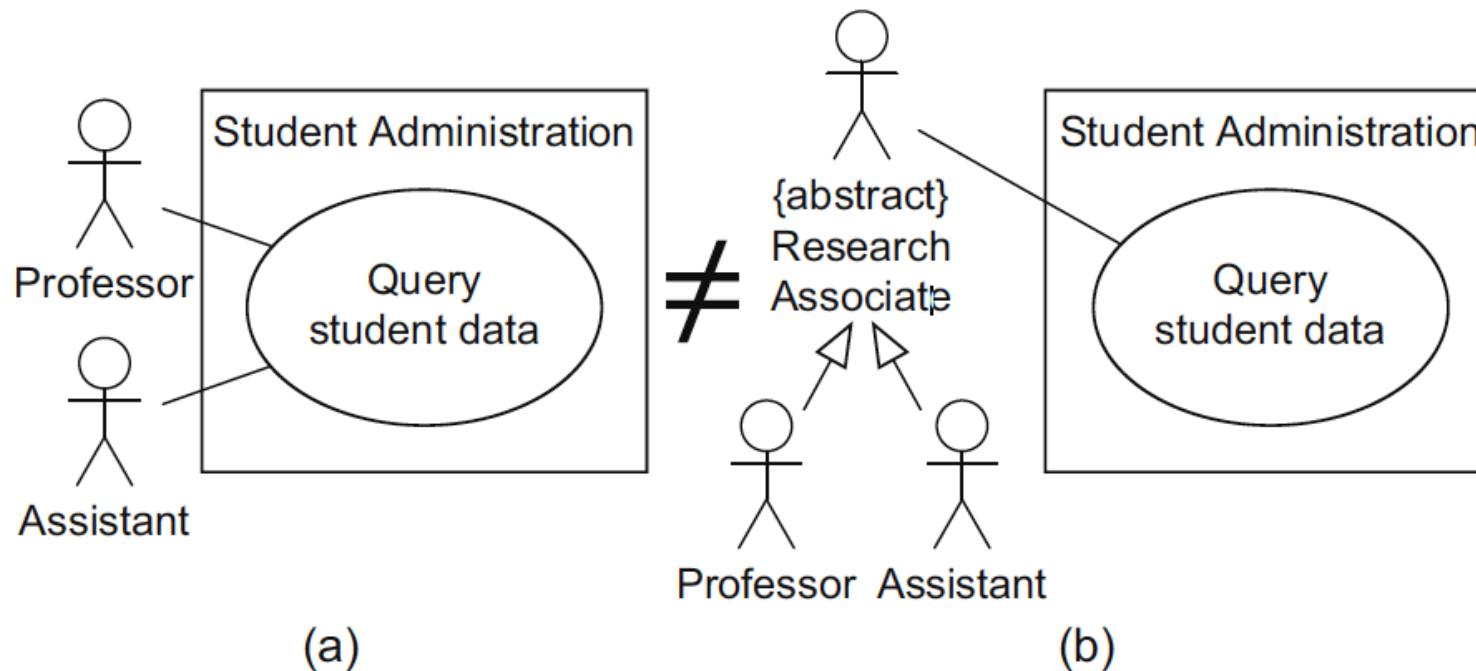
Relationships between Actors

- Generalization/inheritance
 - Actor **A** inherits from actor **B**.
 - **A** can communicate with **X** and **Y**.
 - **B** can only communicate with **Y**.
 - *Multiple inheritance* is permitted.
 - “is a” relation
(e.g., TA is a student)



Relationships between Actors (2)

- Abstract* actors are possible.

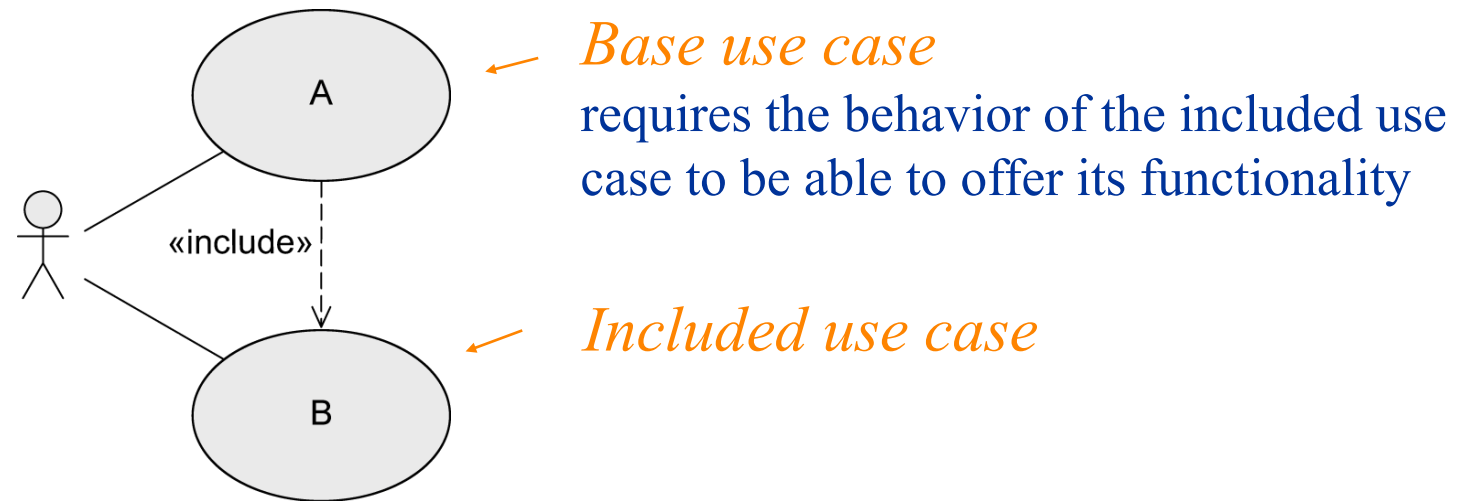
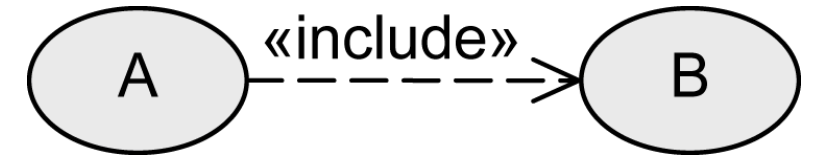


Professor AND Assistant needed
for executing **Query student data**

Professor OR Assistant needed
for executing **Query student data**

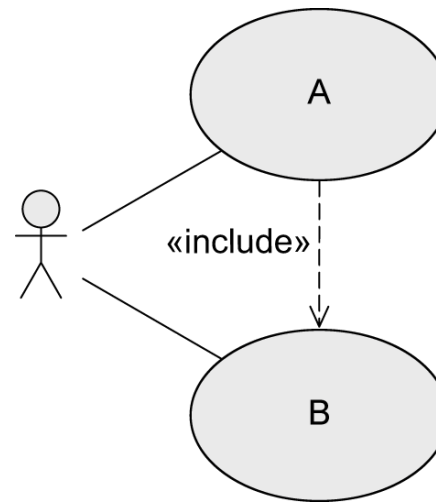
Relationships between Use Cases

- **Include (a.k.a Use)**
- The behavior of one use case (included use case) is *integrated* in the behavior of another use case (base use case)
 - The included use case is part of base use case
 - Like a subroutine



Relationships between Use Cases: Include (2)

- 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.



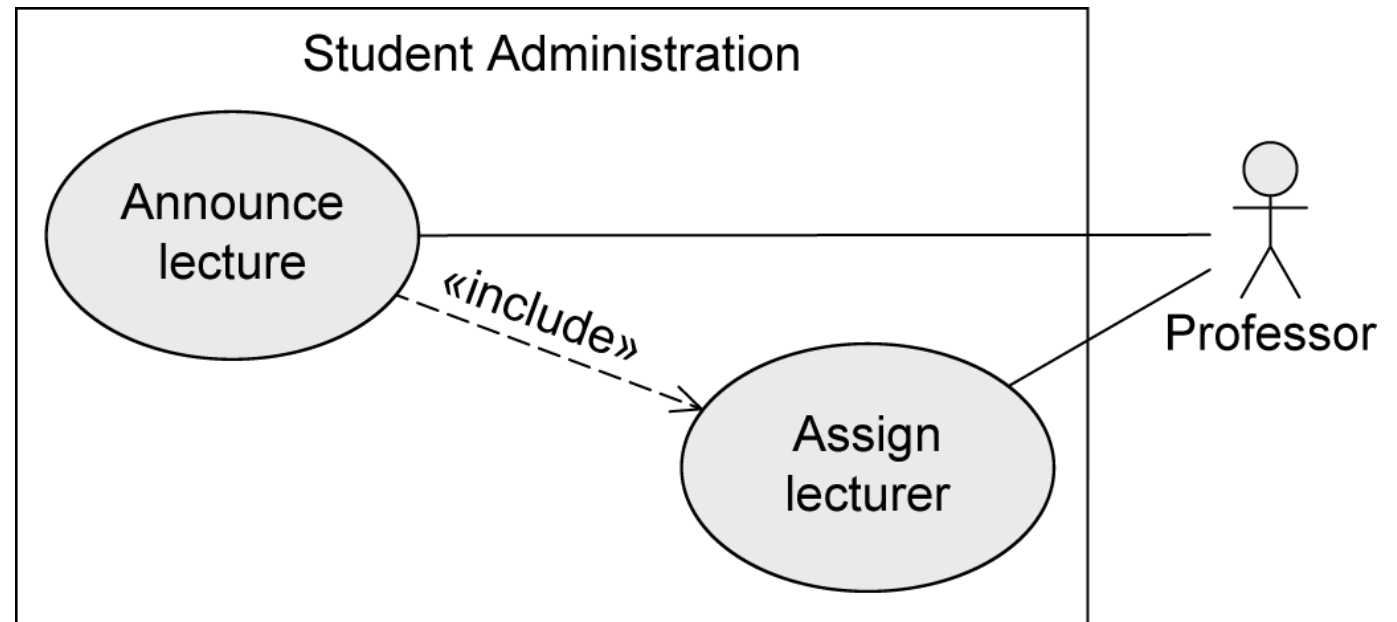
Base use case

requires the behavior of the included use case to be able to offer its functionality

Included use case

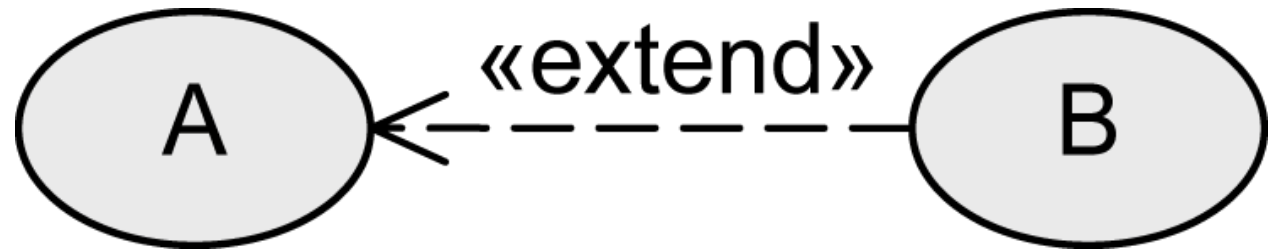
Relationships between Use Cases: Include (3)

- Announce lecture and Assign lecturer are in an «include» relationship, whereby Announce lecture is the base use case.
- whenever a new lecture is announced, the use case Assign lecturer must also be executed.



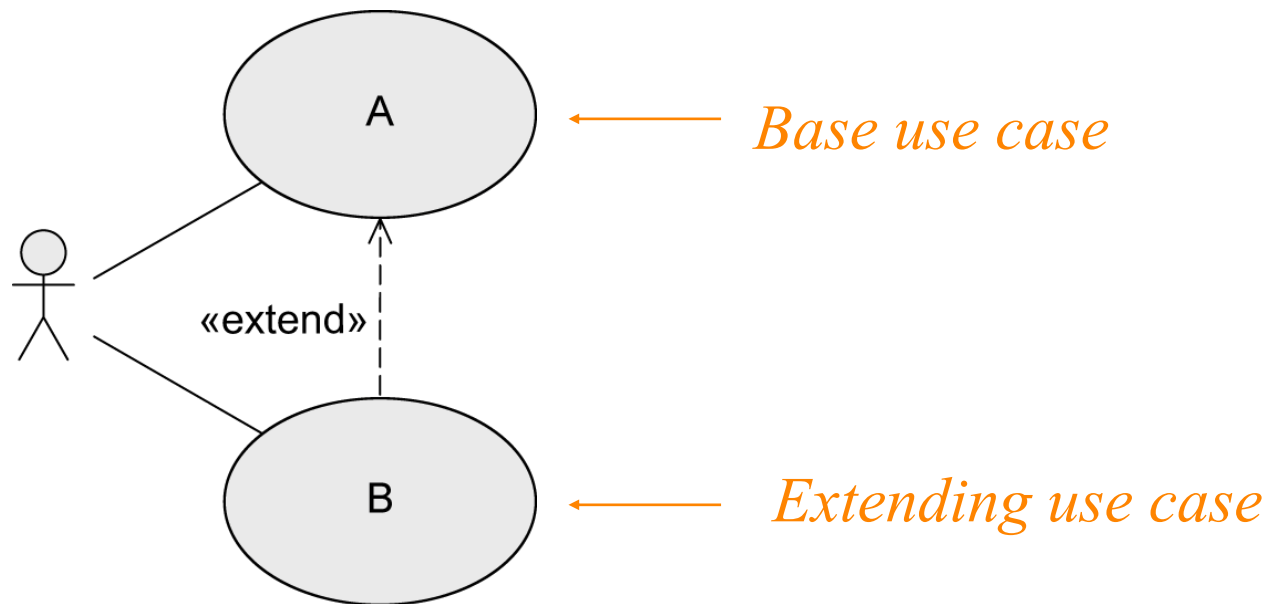
Relationships between Use Cases: Extend

- Extend
- 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**.
 - Optional
 - The extending use case and the base use case both can execute individually
 - The extending use case is triggered by a **condition** at an **extension point**



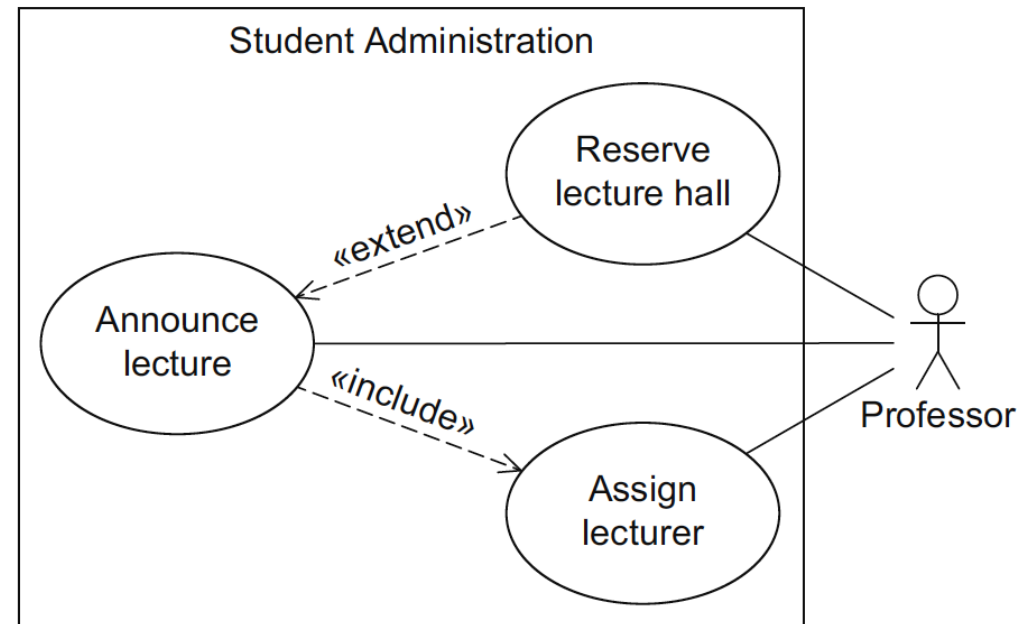
Relationships between Use Cases: Extend (2)

- Extension points define at which point the behavior is integrated.
- Conditions define under which circumstances the behavior is integrated.

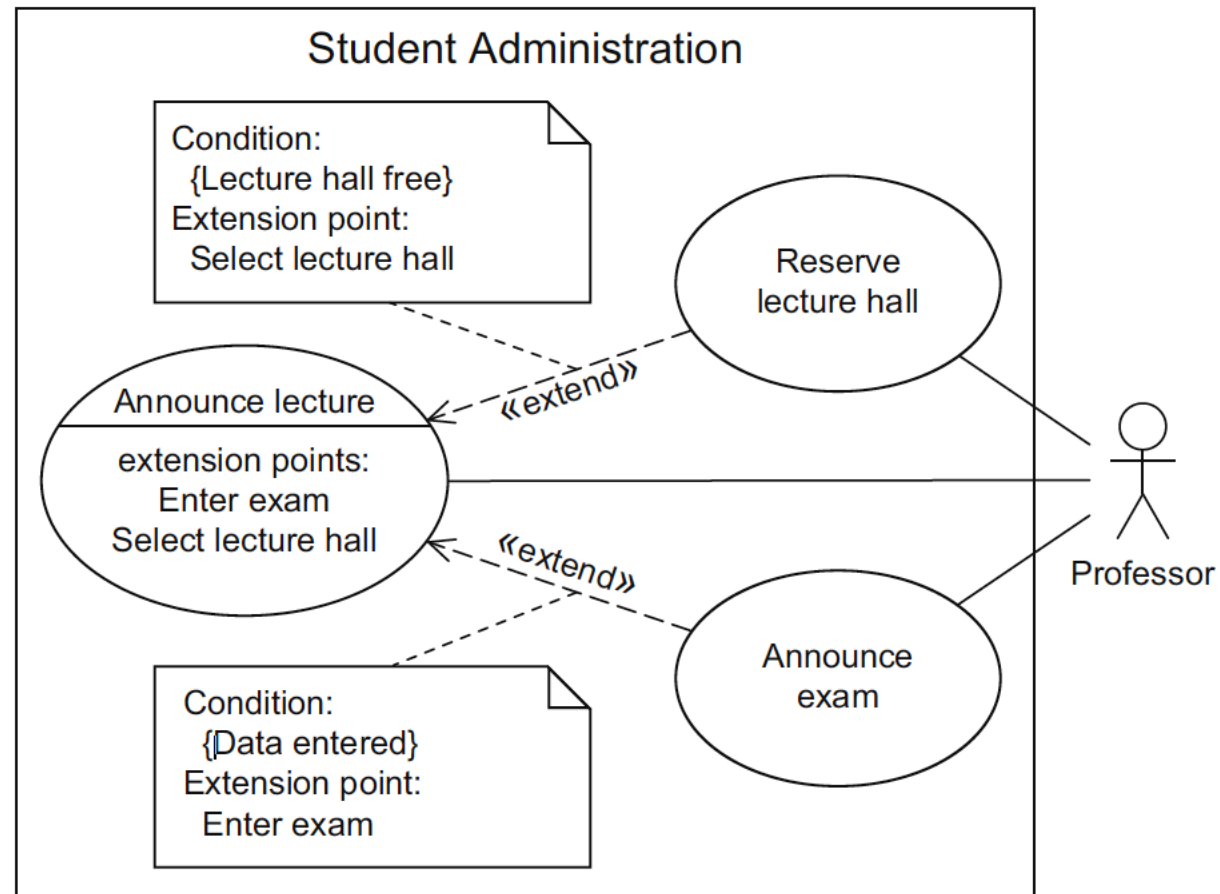


Relationships between Use Cases: Extend (3)

- Example: 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.

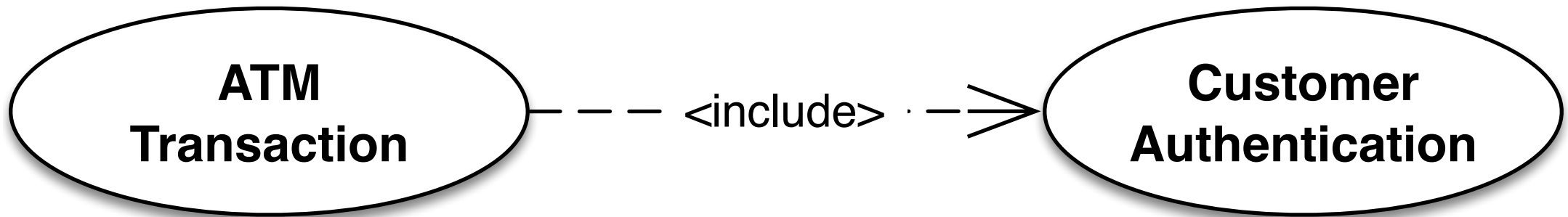


Relationships between Use Cases: Extend (4)



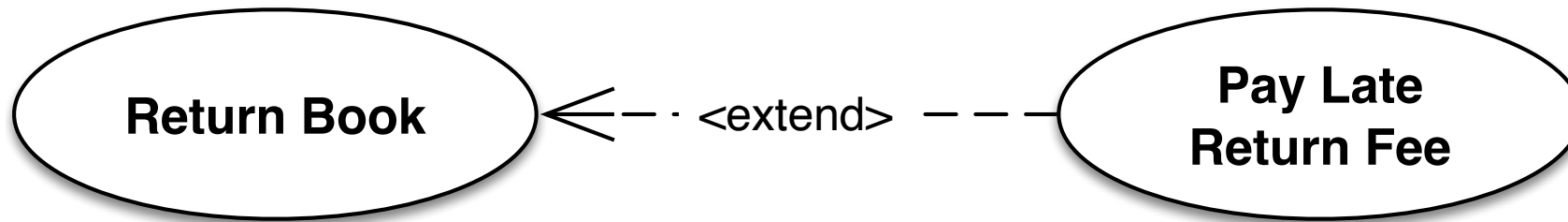
《include》 vs 《extend》

- Include:
 - The behavior of one use case (included use case) is *integrated* in the behavior of another use case (base use case);
 - The base use case **always requires the behavior of the included use case to be** able to offer its functionality.



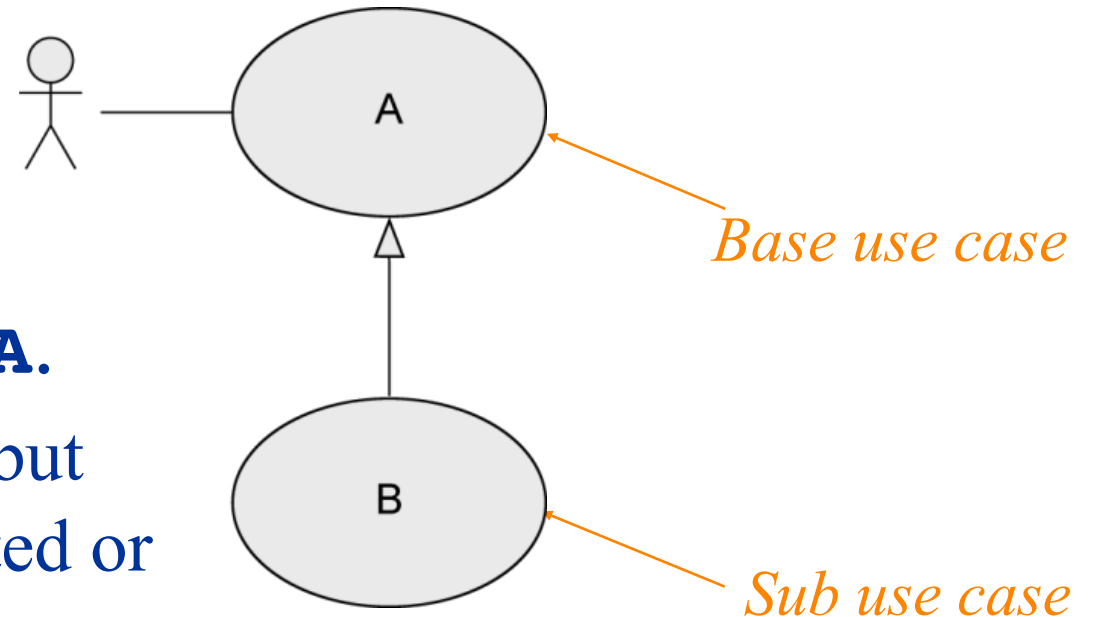
《include》 vs 《extend》 (2)

- Extend
- 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**.



Relationships between Use Cases: Generalization

- Use case **A** generalizes use case **B**.
- **B** inherits the behavior of **A** and may either extend or overwrite it.
- **B** also inherits all relationships from **A**.
- **B** adopts the basic functionality of **A** but decides itself what part of **A** is executed or changed



Example

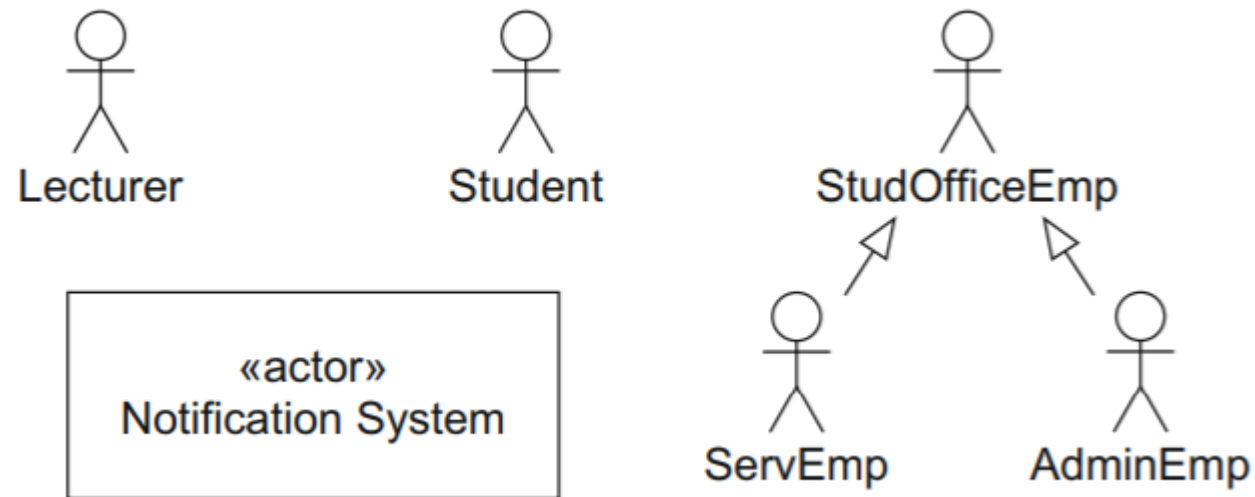
- Many important administrative activities of a university are processed by the student office. Students can register for studies (matriculation), enroll, and withdraw from studies here. Matriculation involves enrolling, that is, registering for studies.
- Students receive their certificates from the 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.

Example

- 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.

Example: Actors

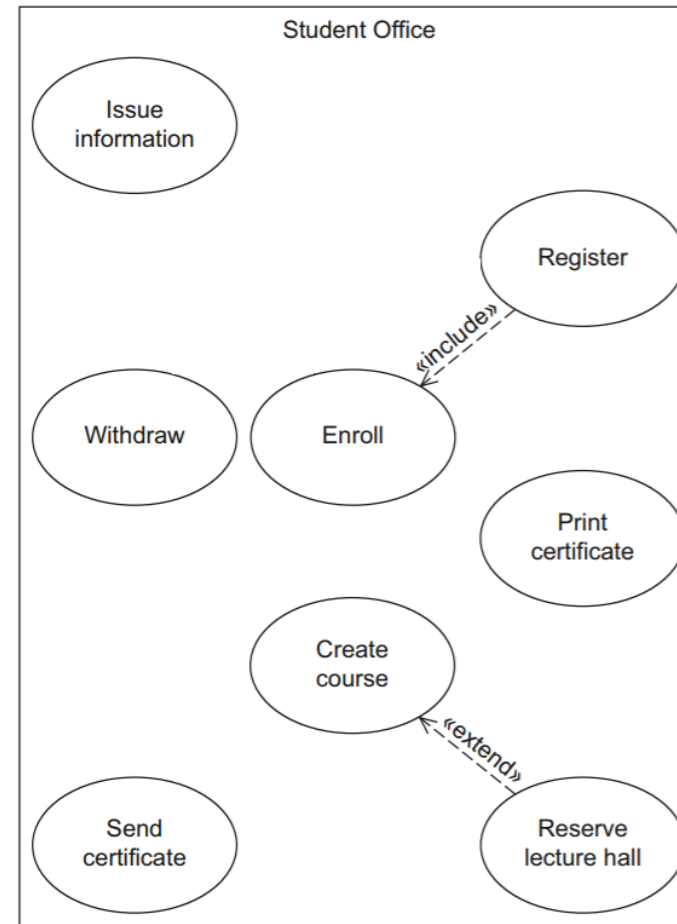
- Students and Notification system are not part of this use case



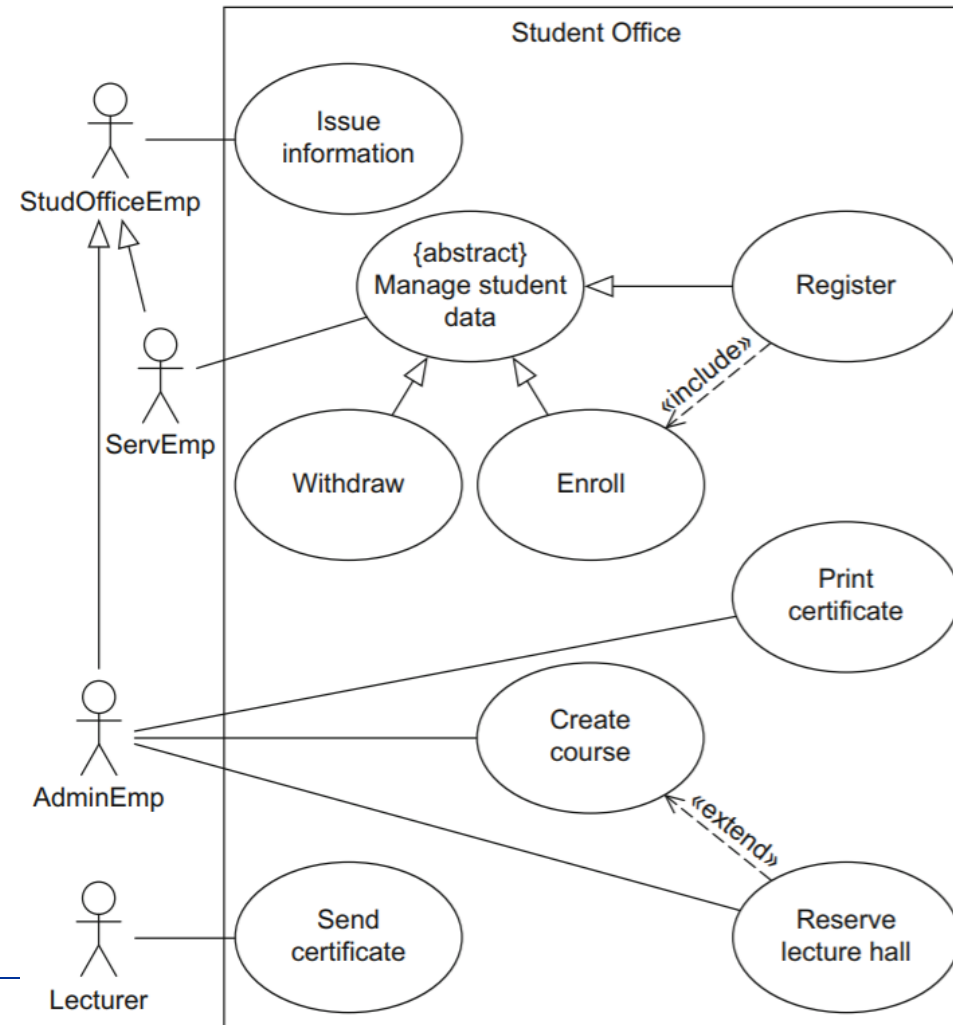
Example: Use Cases

Matriculation **involves** enrolling, that is, registering for studies.

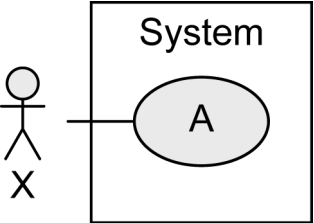


When creating courses, they can **reserve lecture halls**.



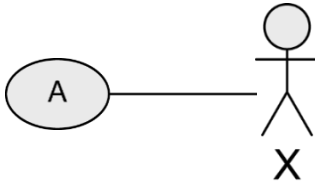
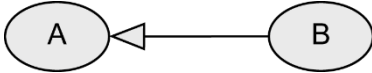
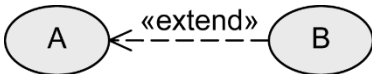
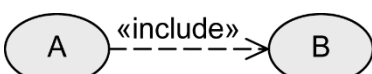
Example



Notation Element

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

Notation Element (2)

Name	Notation	Description
Association		Relationship between use cases and actors
Generalization		Inheritance relationship between actors or use cases
Extend relationship		B extends A: optional use of use case B by use case A
Include relationship		A includes B: required use of use case B by use case A

Best Practice – Identifying Actors

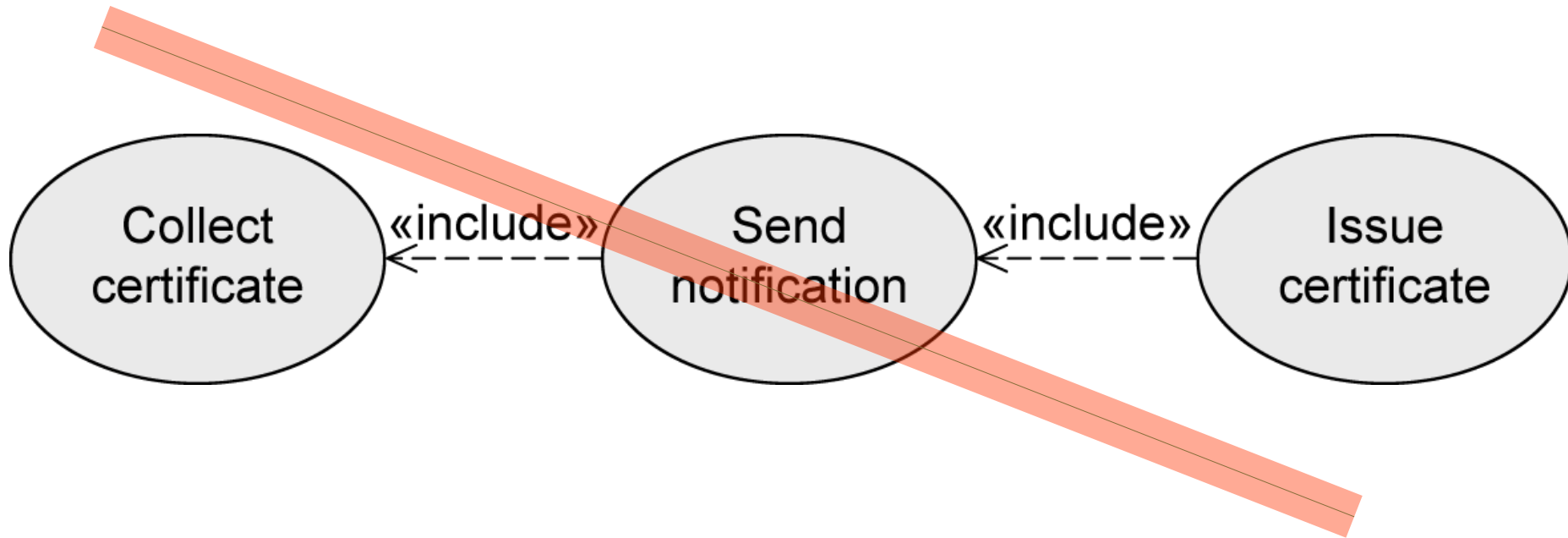
- Who uses the main use cases?
- Who needs support for their daily work?
- Who is responsible for system administration?
- What are the external devices/(software) systems with which the system must communicate?
- Who is interested in the results of the system?

Best Practice (2) – 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?

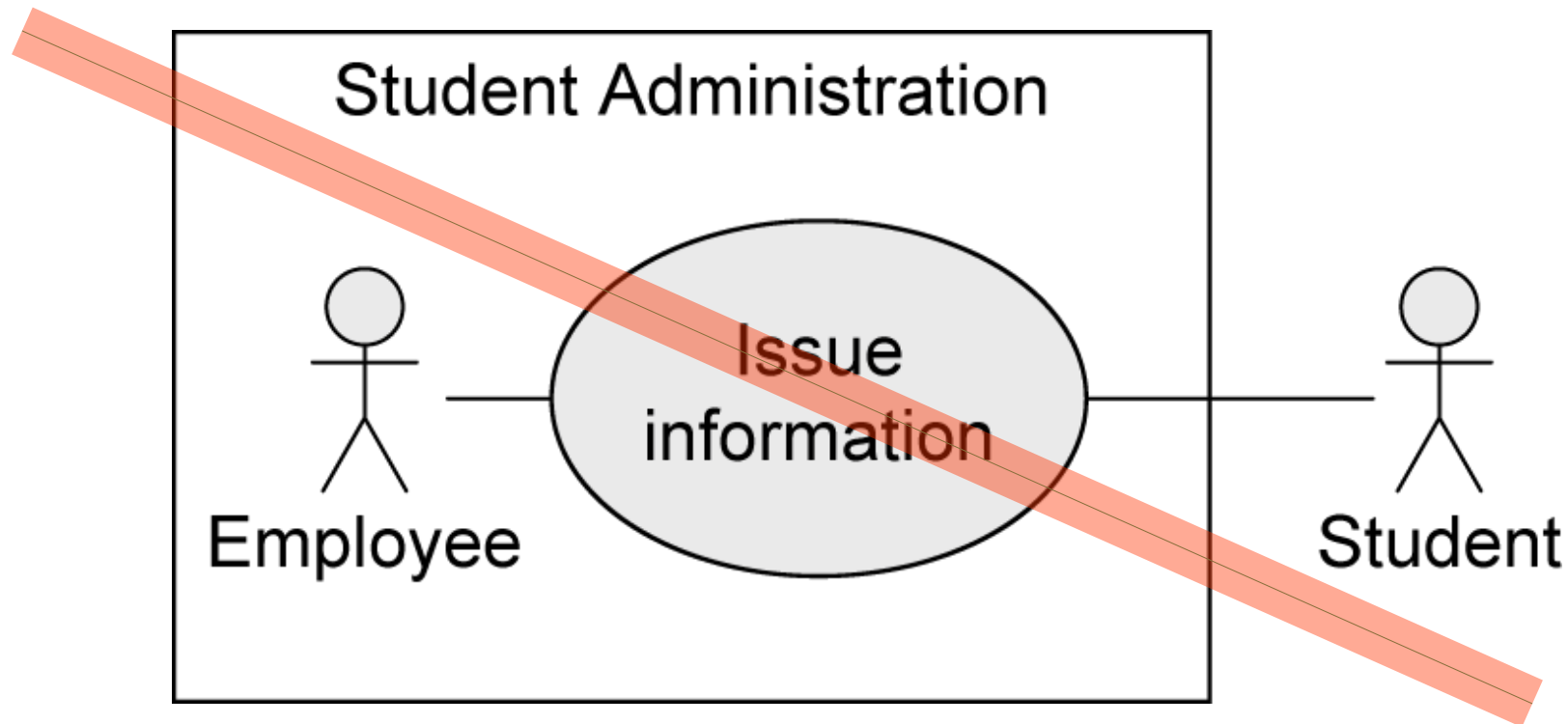
Typical Errors To Avoid

- Use case diagrams do not model processes/workflows!



Typical Errors To Avoid (2)

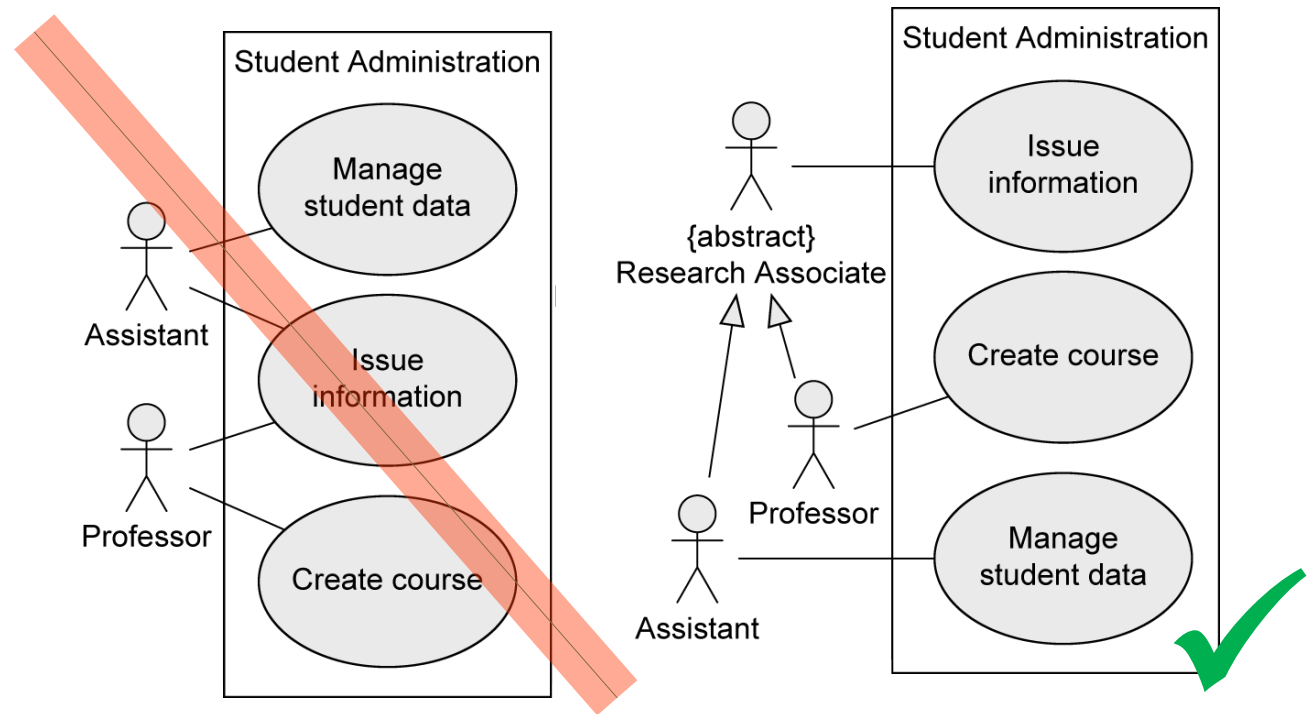
- Actors are not part of the system, hence, they are positioned outside the system boundaries!



Typical Errors To Avoid

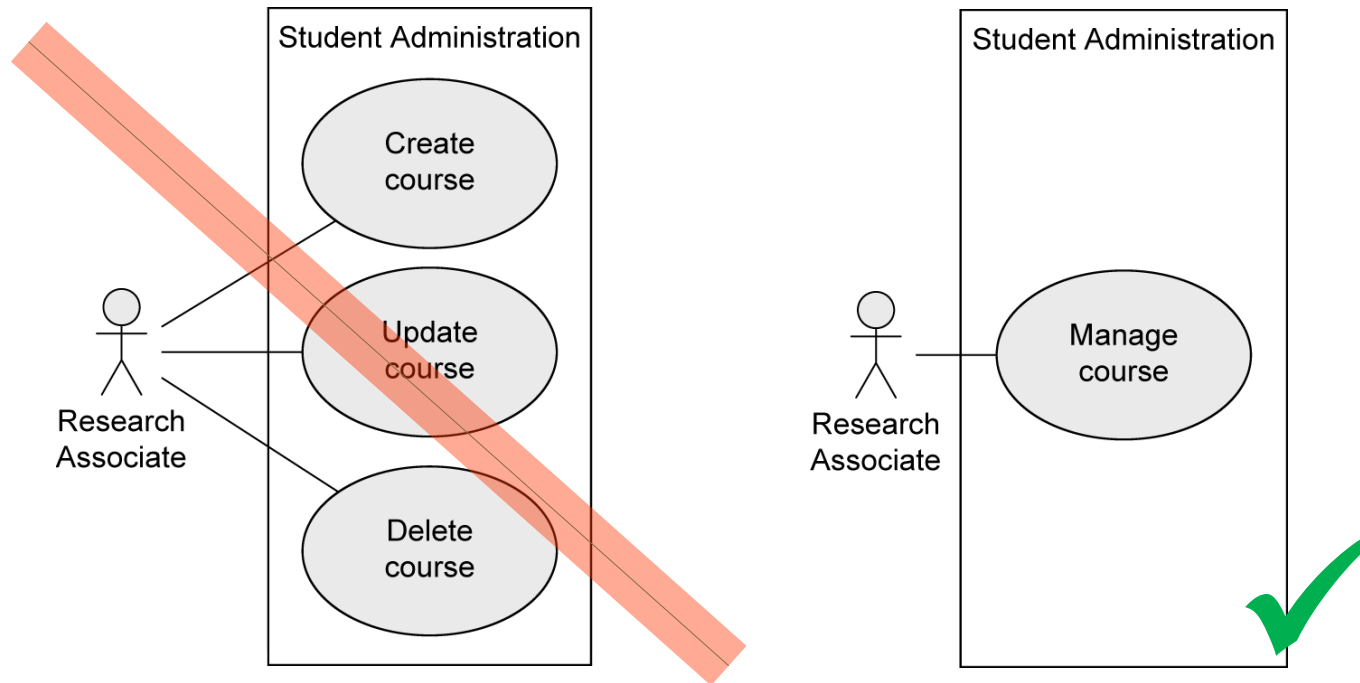
- Use case **Issue information** needs **EITHER** one actor **Assistant** **OR** one actor **Professor** for execution

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.



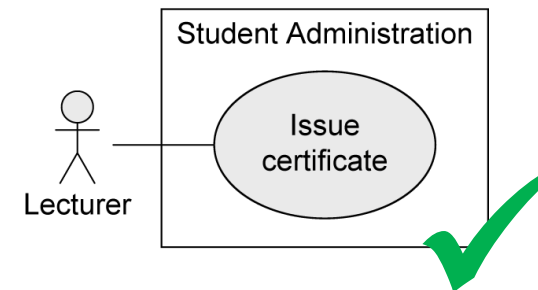
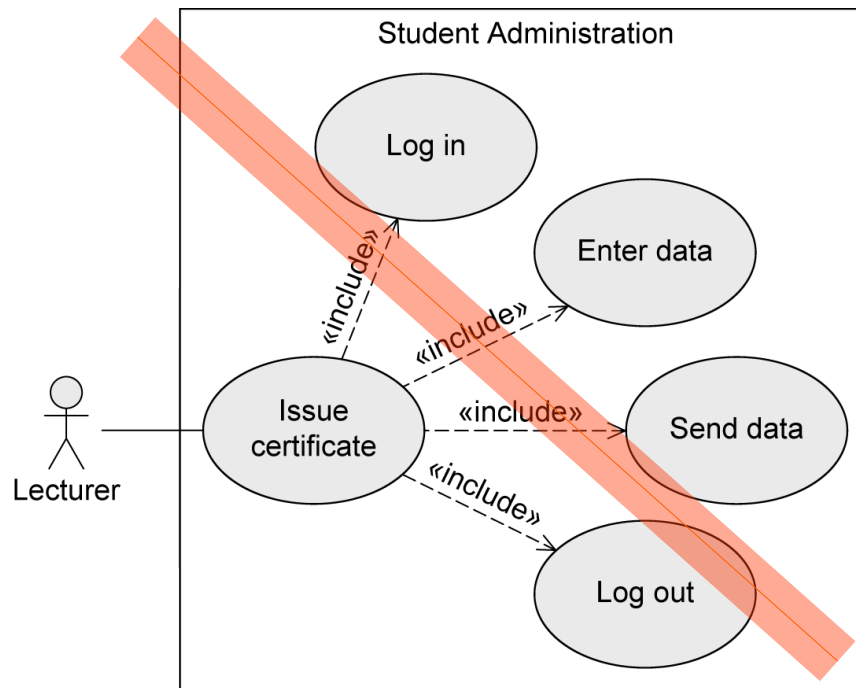
Typical Errors To Avoid

- Many small use cases that have the same objective may be grouped to form one use case

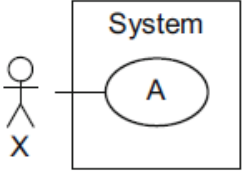
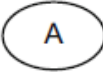
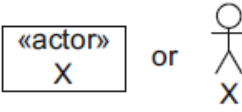

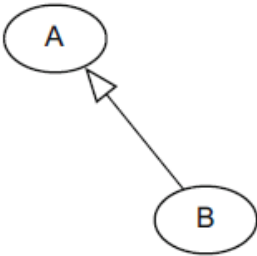


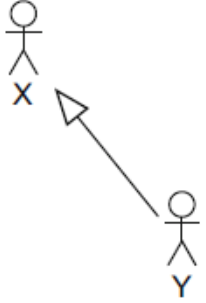
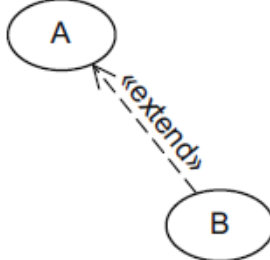
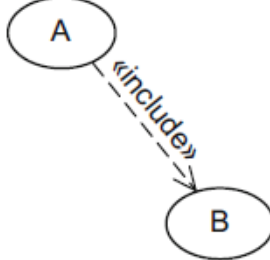
Typical Errors To Avoid

- The various steps are part of the use cases, not separate use cases themselves! -> NO functional decomposition



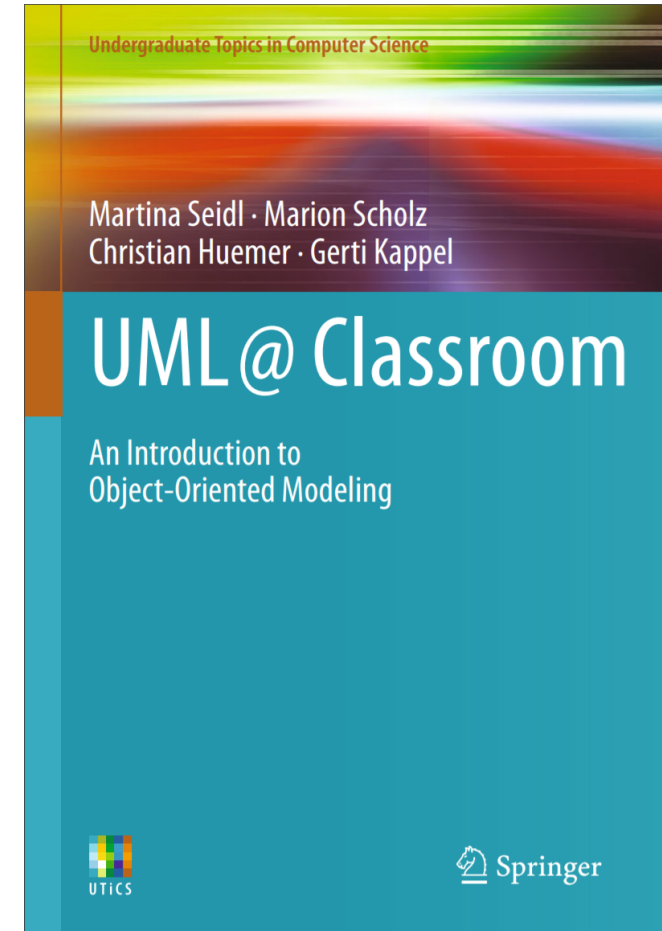
Summary: Use Case Diagram

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

Reference for UML

- Freely available online
- Search from our library website



UML Drawing Tools

- Microsoft Visio can draw basic UML diagrams
 - Available from the library
- Visual Paradigm (Community edition)
 - <https://www.visual-paradigm.com/download/community.jsp>
- IBM Rational Rose
 - Cracked version online (not recommended)