Use Case Diagram

Lecture A2 — Static Modeling: System Functionality

Prof. Ing. Lelio Campanile

Dipartimento di Matematica e Fisica
Università degli Studi della Campania Luigi Vanvitelli
Corso di Laurea Magistrale in Data Science

A.A. 2025/2026

Based on: Seidl et al. — *UML @ Classroom*, Ch. 6 (Springer, 2015)

Where we are in the process

We have collected and documented **functional requirements** (user stories, use cases in prose, system boundary). The next step is to represent them *visually* and *precisely* using a UML Use Case Diagram.

User story (requirements)

*"As a **Professor**, I want to **announce an exam** so that students can register for it."*

⇒

Use Case Diagram element

Actor: Professor
Use case: Announce exam
Association:
Professor – Announce exam

⇒

Topics covered

Concepts

- Introduction and motivation
- Use cases
- Actors
- Relationships: UC $\leftrightarrow$ Actor
- Relationships between use cases
- Relationships between actors

Practice

- Description of use cases (template)
- Best practices: identifying actors and UCs
- Typical errors to avoid
- Notation reference

Running example throughout: **Student Administration System** (Prof, Student, Admin Staff, E-Mail Server)

Introduction

Use Case Diagrams: what, why, and when

Role of the Use Case Diagram

The use case is a fundamental concept of many object-oriented development methods. Use case diagrams express the **expectations of the customers/stakeholders** and are essential for a detailed design. The diagram is used during the **entire analysis and design process**.

Three questions a UC diagram answers

Q1 What is being described? → The **system**

Q2 Who interacts with the system? → The **actors**

Q3 What can the actors do? → The **use cases**

Student Administration System — first look

System: Student administration system

Actors: Professor

Use cases: Query student data Issue certificate Announce exam

Use Cases and Actors

The building blocks of the diagram

Use Case

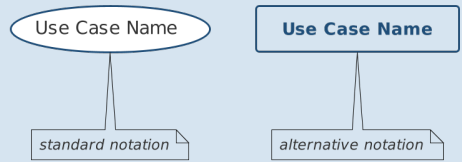
Definition

A **use case** describes functionality expected from the system under development. It:

- provides tangible benefit for one or more actors
- is derived from collected customer wishes
- documents the functionality that the system offers

The set of all use cases describes the complete functionality that a system shall provide.

Notations



Actor (1/3) — Role and boundary

Definition

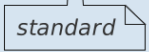
Actors interact with the system:

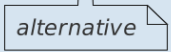
- **by using use cases** — actors initiate the execution
- **by being used** — actors provide functionality for the execution of use cases

Actors represent *roles* that users adopt. A specific user can adopt multiple roles simultaneously.

Key constraint: Actors are *not* part of the system — they are outside the system boundary.

Notations





Actor (2/3) — Actor vs. class

Important distinction

User data is *also* administered within the system. An actor and a class of the same name model **different things**:

Actor: `Assistant`

Represents the *role* of a person who interacts with the system from outside.

The actor Assistant interacts with the system "Laboratory Assignment" by using it.

Class: `Assistant`

Describes objects *inside* the system that store data about the user (e.g., name, ssNr, ...).

The class Assistant is part of the structural model — it appears in the Class Diagram.

Same name, different model, different purpose. The actor is *outside* the system boundary; the class is *inside*.

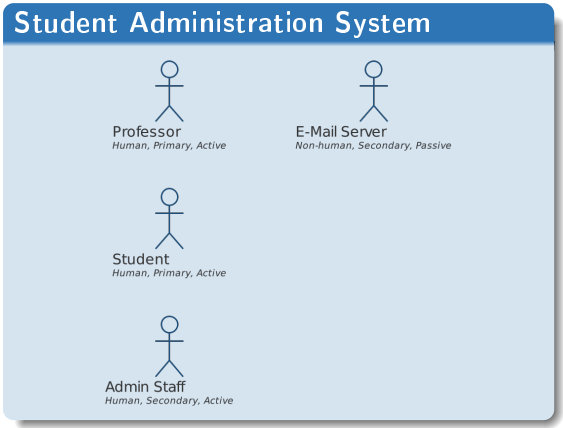
Actor (3/3) — Taxonomy

By nature

Human	e.g., Student, Professor
Non-human	e.g., E-Mail Server, external database

By role in the use case

Primary	has the main benefit
Secondary	provides support, no direct benefit
Active	initiates the execution
Passive	provides functionality for execution



Relationships

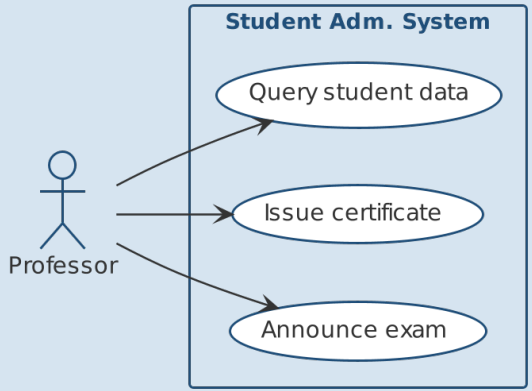
Associations, include, extend, generalization

Relationships between Use Cases and Actors

Rules

- Actors are connected to use cases via **solid lines** (associations)
- Every actor must communicate with **at least one** use case
- An association is always **binary**
- Multiplicities may be specified

Student Administration System



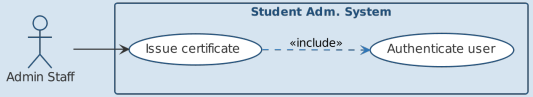
Relationships between Use Cases — «include»

Semantics

The behavior of one use case (*included UC*) is **always integrated** in the behavior of another (*base UC*).

Base UC	*requires* the included UC to offer its own functionality
Included UC	*may* be executed on its own

Example



Best practice

Use «include» to factor out **shared sub-behaviour** that multiple use cases need (e.g., "Authenticate user" included by several UCs).

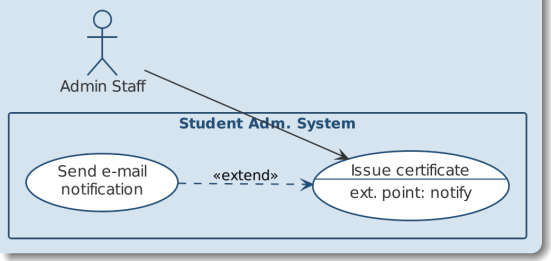
Relationships between Use Cases — «extend»

Semantics

The behavior of one use case (*extending UC*) **may** be integrated in another (*base UC*), but does not have to.

- Both UCs may also be executed independently
- **Extension points** define where behavior is injected
- **Conditions** define when it is injected

Example with extension point



Best practice

Use «extend» for **optional behavior** that adds to a base UC only under specific conditions.

«include» vs. «extend» — comparison

	«include»	«extend»
Direction	Base → Included	Extending → Base
Execution	**Always** executed	**Conditionally** executed
Dependency	Base *requires* included	Base is *independent*
Use when...	shared sub-behaviour	optional / exceptional behaviour

Arrow direction: both use **dashed arrow with stereotype label**. The arrow points *from* the dependent UC *toward* the independent one.

Generalization of Use Cases

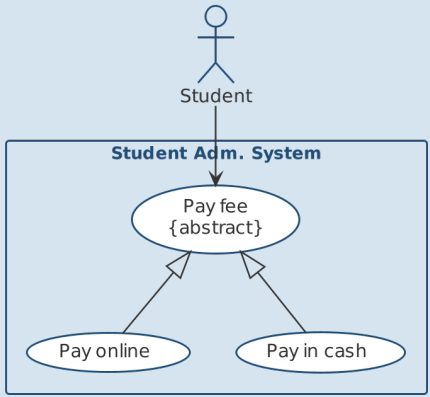
Semantics

Use case A generalizes use case B:

- B inherits the behavior of A and may extend or overwrite it
- B inherits all relationships from A
- A may be labeled `{abstract}` — not directly executable, only B is executable

Similar to class inheritance: the sub-use-case adopts the basic functionality and specialises it.

Example



Generalization of Actors

Semantics

Actor A inherits from actor B (super-actor):

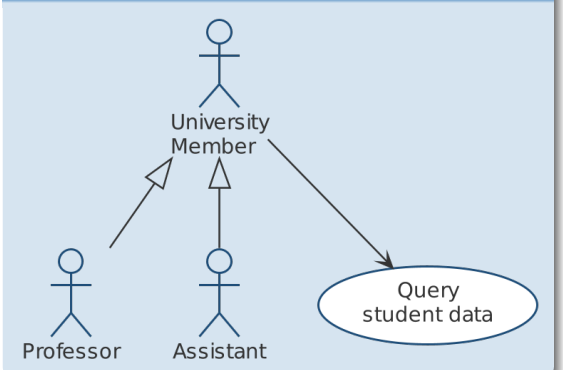
- A can communicate with all UCs that B can, plus more
- Multiple inheritance is permitted
- Abstract actors are possible

AND vs. OR semantics

AND Professor *and* Assistant *both* needed to execute UC

OR Professor *or* Assistant can execute UC independently

Example



Describing Use Cases

From diagram to textual specification

Description of Use Cases — Template

Why a textual description?

The diagram captures the *structure*; the textual description captures the *flow*. Both are needed for a complete specification. *[Cockburn, Writing Effective Use Cases, 2000]*

Required fields

- **Name**
- **Short description**
- **Precondition**: prerequisite for successful execution
- **Postcondition**: system state after success
- **Error situations** + system state on error
- **Actors** that communicate with the UC

Flow fields

- **Trigger**: events that initiate/start the UC
- **Standard process**: the main success scenario, step by step
- **Alternative processes**: deviations from the standard process (numbered with prime: $4'$, $5'$, ...)

Description of Use Cases — Example

UC: Reserve lecture hall — Student Administration System

Header

Name	Reserve lecture hall
Short desc.	An employee reserves a hall for an event.
Precondition	Employee is authorized.
Postcondition	A lecture hall is reserved.
Error	No free lecture hall available.
On error	No reservation made.
Actor	Employee
Trigger	Employee requires a lecture hall.

Standard process

1. Employee logs in to the system.
2. Employee selects the lecture hall.
3. Employee selects the date.
4. System confirms the hall is free.
5. Employee confirms the reservation.

Alternative process

(4′) Hall is *not* free.
(5′) System proposes an alternative.
(6′) Employee selects alternative and confirms.

Best Practices & Typical Errors

Best Practices — Identifying actors and use cases

Identifying Actors — ask yourself:

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

Identifying Use Cases — ask yourself:

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

Typical Errors to Avoid

Error 1 — UC diagrams are not process models

Use case diagrams describe *what* the system does, not *how* or *in what sequence*. Do not model workflows or control flow — use an **Activity Diagram** for that.

Error 2 — Actors inside the boundary

Actors are *always* outside the system boundary rectangle. Placing actors inside the boundary is a common mistake.

Error 3 — Ambiguous actor assignment

Error 4 — Over-granularity

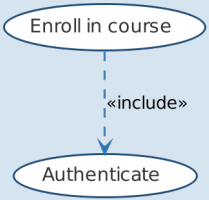
Many small use cases with the same objective should be **grouped** into one use case. Avoid one UC per button click.

Error 5 — Functional decomposition

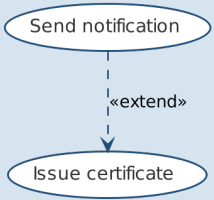
Steps of a use case are **not** separate use cases. “Select date”, “confirm reservation”, “log in” are steps inside the UC, not independent UCs. No functional decomposition!

Best Practices — «include» and «extend»

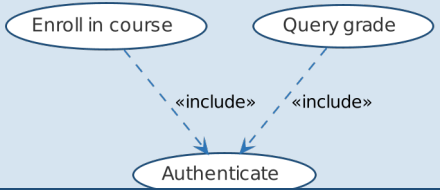
«include» — UML standard



«extend» — UML standard



«include» — best practice



The extending UC points *toward* the base UC.
Remember: direction is opposite to «include».

Notation Elements

Name	Notation	Description
System boundary	Rectangle	Separates system from external environment
Use case	Ellipse	Unit of functionality offered by the system
Actor	Stick figure	Role interacting with the system
Association	Solid line	Relationship between actor and use case
Generalization	Open-head arrow	Inheritance between actors or use cases
«extend»	Dashed arrow	Optional integration of extending UC into base
«include»	Dashed arrow	Mandatory integration of included UC into base

Key points

Use Case Diagrams are static

They describe *what* the system does from an external perspective. They do not model sequence, flow, or internal behavior.

Actors are outside the boundary

An actor is a role, not a person. Actors are always external to the system; do not place them inside the system rectangle.

«include» vs. «extend»

`«include»`: the base UC *always* needs the included UC.
`«extend»`: the extending UC is *optional*.
Arrow direction is **opposite** for the two stereotypes.

Textual description is essential

The diagram provides structure; the textual description (Cockburn template) provides the flow. Both are needed for a complete specification.

What's next

Lecture A3 — Class Diagram

Modeling the *static structure* of the system.

From the same Student Administration System we now ask:
What entities exist? How are they organized? What attributes and operations do they have?

Seidl et al., Ch. 3

Module B — Behavioral Modeling

Once structure is modeled, we model *behavior*:

- **Sequence Diagram**: object interactions over time
- **Activity Diagram**: workflows and processes
- **State Machine**: state-driven behavior

Each behavioral diagram corresponds to a UC scenario.

Thank you

Prof. Ing. Lelio Campanile

lelio.campanile@unicampania.it

Dipartimento di Matematica e Fisica
Università degli Studi della Campania Luigi Vanvitelli

Elements of Software Engineering and Information Systems

Corso di Laurea Magistrale in Data Science