Class Diagram

Lecture A3 — Static Modeling: Structure of the System

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. 3 (Springer, 2015)

Topics covered

Objects and Classes

- Objects and Object Diagrams
- From object to class
- Attribute syntax (visibility, type, multiplicity, properties)
- Operation syntax
- Class variables and class operations

Relationships and Levels

- Binary Association (navigability, multiplicity, role)
- N-ary Association, Association Class
- Aggregation and Composition
- Generalization (inheritance)
- **Levels of detail: overview → implementation**
- Code generation

Objects and Classes

From instances to blueprints

Object

Definition

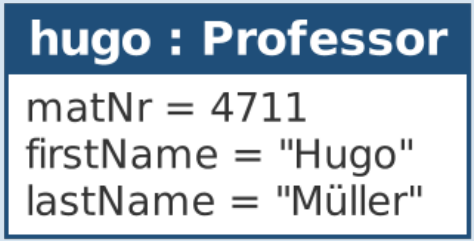
An object represents an **individual** of a system. It has a name, a class (type), and current values for each attribute.

Objects may be **anonymous** (no object name).

Object Diagram

An Object Diagram shows objects and their relationships (*links*) at a **specific moment in time**. It is a snapshot of the running system.

Notation



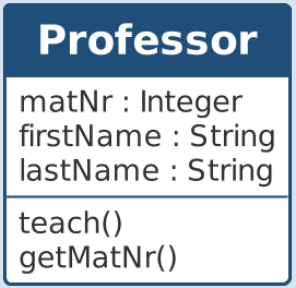
Object name underlined. Format: `name : ClassName`

From Object to Class

Key insight

Individuals of a system often have **identical characteristics and behavior**. A **class** is a construction plan for a set of similar objects. Objects are *instances* of classes.

Class notation



Three compartments

Class name	Identifies the class (top)
Attributes	Structural characteristics. Each instance has its *own* value (middle)
Operations	Behavior. Identical for all objects of the class (bottom)

Operations are *not* shown in Object Diagrams — only in Class Diagrams.

Attribute Syntax

Full syntax: `[visibility] name [: type] [multiplicity] [= default] [{properties}]`

Visibility

+	public	everybody
-	private	object itself only
#	protected	class and subclasses
~	package	same package

Multiplicity `[min..max]`

`[1]`	exactly one (default)
`[0..1]`	optional
`[*]`	zero or more
`[1..*]`	one or more

Type

Primitive: `Boolean`, `Integer`, `String`, `...`

User-defined class

Enumeration: «enumeration»

Composite data type: «datatype»

Properties

`{readOnly}` `{unique}` `{ordered}`

Derived attribute: /age (computed from dob)

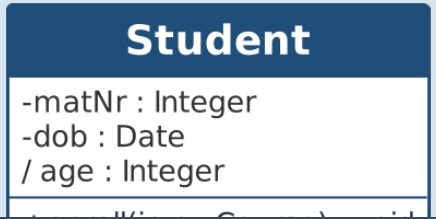
Operation Syntax

`[visibility] name ([params]) [: returnType] [{properties}]`

Parameter direction

`in`	input: value expected on call
`out`	output: new value after execution
`inout`	combined input/output

Example



Class variables and operations

Class variable (static): defined once per class, shared by all instances (e.g., counters, constants).

Class operation (static): can be called without an instance (e.g., constructors, factory methods, `Math.sin()`).

Notation: underline name

`+ getInstance() : Student`
`- count : Integer`

Levels of Detail

From conceptual overview to implementation blueprint

Specification of Classes: Levels of Detail

Key design decision

The same class can be specified at **different levels of granularity**. You choose based on the *purpose* of the model and its *audience*.

1 — Conceptual

Name only. No attributes, no operations.
Audience: stakeholders, domain experts.
Goal: shared vocabulary, scope.

2 — Specification

Interface visible: attributes and operations with names and types.
Audience: architect, developer.
Goal: API and component

3 — Implementation

Full detail: visibility, derived attrs, multiplicities, properties, static members.
Audience: developer, code generator.
Goal: direct blueprint.

Level 1 — Conceptual Overview

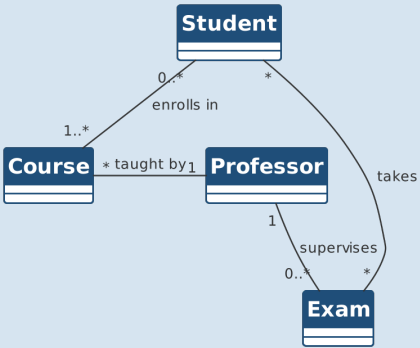
Name only — used in early requirements discussions and domain modelling

Characteristics

- Class name only — no compartments needed
- Focus: *what entities exist* and *how they relate*
- Associations shown with names and multiplicities
- Ideal for whiteboard discussions with stakeholders

At this level the diagram acts as a **domain model**: it communicates the vocabulary of the problem, not the solution.

Student Administration System — conceptual



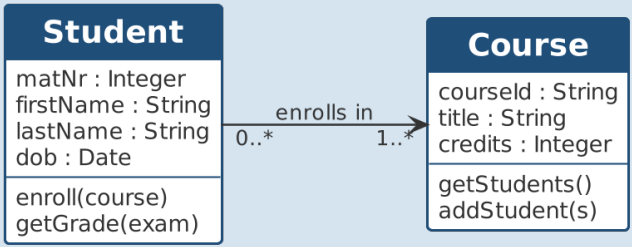
Level 2 — Specification Level

Attributes and operations visible — used for design reviews and API definition

Characteristics

- Attributes visible: name and type
- Operations visible: name, params, return type
- Visibility *may* be shown
- Focus: *what can be done* with the class (its interface/contract)
- Used in design reviews, sprint planning, and API documentation

Student Administration System — specification



Level 3 — Implementation Level

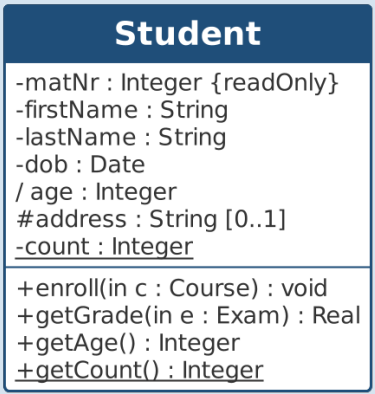
Full detail — used as blueprint for code generation

Characteristics

- Full visibility for all members
- Derived attributes (/age)
- Multiplicities on all attributes
- Property constraints (`{readOnly}`)
- Static members underlined
- Parameter directions explicit
- Almost directly translatable to code

At this level the model *is* the source of truth. Code generation tools (e.g., in

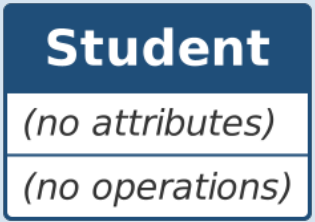
Student — implementation level



Levels of Detail — Side-by-Side Comparison

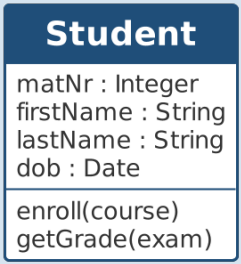
The same class `Student` at three levels

Level 1 — Conceptual



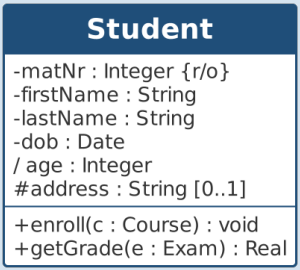
Early discussion, domain vocabulary.

Level 2 — Specification



API definition, design review.

Level 3 — Implementation



Blueprint for code generation.

Associations

Binary, n-ary, association class, navigability

Binary Association

Definition

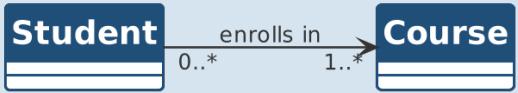
An association models possible **relationships between instances** of two classes. It connects objects at runtime.

Binary association elements

Name	describes the relationship
Role	describes how an object participates
Multiplicity	how many objects on each end
Navigability	direction of access
Visibility	visibility of role

Multiplicity notation

`1`	exactly one
`0..1`	zero or one
`*`	zero or more
`1..*`	one or more
`m..n`	between *m* and *n*



Binary Association — Navigability

Definition

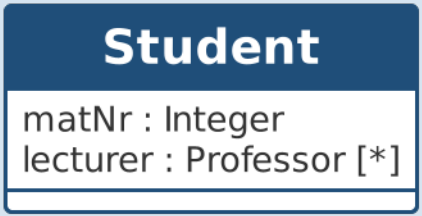
Navigability: an object knows its partner objects and can access their visible attributes and operations.

- Open arrowhead $\rightarrow$: navigable in that direction
- Cross $\times$: *not* navigable in that direction
- No arrow: navigability *unspecified* (bidirectional assumed)

UML standard vs. best practice

Standard: unspecified arrows by default.
Best practice: always specify navigability explicitly to avoid ambiguity.

Association as attribute

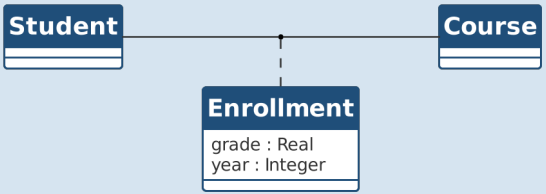


An association navigable toward `Professor` may be modeled as an attribute `lecturer` inside `Student`. Both notations are equivalent — the attribute form is often preferred in implementation.

Association Class and N-ary Association

Association Class

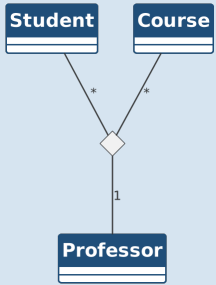
An association can itself have **attributes and operations**. This is modeled as an *association class*, connected to the association line by a dashed line.



N-ary Association

More than two partner objects are involved in the relationship. Represented by a **diamond** at the center.

No navigation directions in n-ary associations.



Aggregation, Composition, Generalization

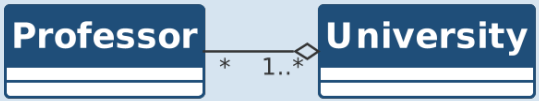
Structural relationships

Aggregation and Composition

Shared Aggregation ("has a")

A **parts-whole** relationship. Parts may exist independently and be shared by multiple wholes.

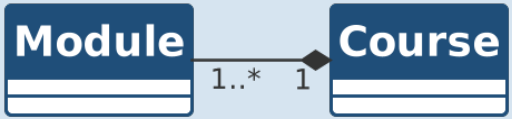
Notation: hollow diamond at the whole end.



Composition ("is part of")

An **existence-dependent** parts-whole relationship. Parts cannot exist without the whole; the whole *owns* the parts.

Notation: filled diamond at the whole end.



If the `Course` is deleted, all its `Modules` are deleted too.

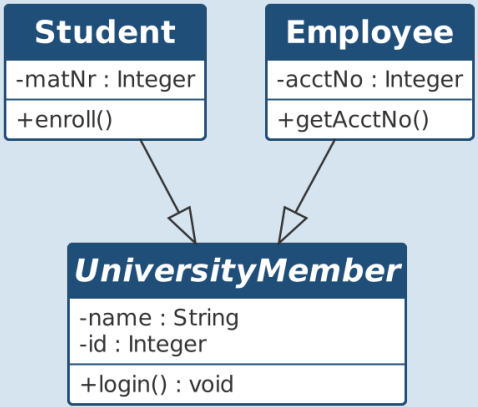
Generalization (Inheritance)

Definition

A **generalization** relationship:

- The subclass inherits all attributes, operations, and associations of the superclass
- The subclass may *extend* or *override* inherited behavior
- **Abstract classes**: cannot be instantiated, labeled `{abstract}` or in *italics*
- **Notation**: open (unfilled) arrowhead pointing to the superclass

Example



Aggregation vs. Composition vs. Association

	Association	Aggregation	Composition
Relationship	Generic	Part-of (weak)	Part-of (strong)
Part exists without whole?	Yes	**Yes**	**No**
Part shared?	N/A	**Yes**	**No**
Notation	Solid line	Hollow diamond	Filled diamond
Example	Student takes Exam	Univ. has Professors	Course has Modules

In practice, **composition** is the most important to get right: it implies lifecycle dependency. When the whole is destroyed, the parts are too.

Code Generation

From Class Diagram to implementation

Code Generation from Class Diagram

Level 3 diagrams are almost directly translatable to code

Mapping rules

Class	`class` declaration
Attribute	instance variable
Operation	method
Visibility	`public/private/protecte`
Generalization	`extends`
Interface	`implements` / `interface`
Composition	object owned, same lifecycle
Multiplicity $*$	array or collection

Generated Java skeleton

```
class Student extends UniversityMember {
  private int matNr;
  private String firstName;
  private String lastName;
  private Date dob;
  protected String address;

  public void enroll(Course c){...}
  public float getGrade(Exam e){...}
}
```

Association toward `Course[*]` becomes `Course[] courses` or a `List<Course>`.

Notation Elements

Name	Notation	Description
Class	3-part rectangle	Name, attributes, operations
Abstract class	Italics name	Cannot be instantiated
Association	Solid line	Relationship between classes
Navigability	Open arrowhead	Direction of access
Assoc. class	Dashed line	Association with attributes
Aggregation	Hollow diamond	Weak parts-whole
Composition	Filled diamond	Strong (lifecycle) parts-whole
Generalization	Open triangle	Inheritance
Object	Underlined name	Instance of a class

Key points

Class Diagram models static structure

It captures entities, their attributes, operations, and relationships. It does not model time or behavior.

Composition is the strongest relationship

Composition implies lifecycle dependency. Do not confuse it with aggregation (parts may exist independently).

Choose the right level of detail

Conceptual for stakeholders; Specification for design; Implementation for code generation. The choice is deliberate — not an accident.

Generalization vs. composition

Prefer composition over inheritance when in doubt (*favor composition*) — generalization creates tight coupling and is harder to refactor later.

What's next

Module B — Behavioral Modeling

Having modeled *what* the system is (Use Case + Class Diagram), we now model *how it behaves*.

Each behavioral diagram will be built on the same Student Administration System.

Upcoming lectures

B1 **Sequence Diagram**
Object interactions over time *(Seidl Ch. 4)*

B2 **Activity Diagram**
Workflows and processes *(Seidl Ch. 5)*

B3 **State Machine**
State-driven behavior *(Seidl Ch. 7)*

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