Unified Modelling Language (UML)

Table of contents

Outline 1
Recap: inheritance vs. composition 1
What UML is for 2
UML class diagram 2
UML inheritance diagram 3
Association, multiplicity, composition and aggregation 3
Exercise 7
A larger example 7
Summary 8

See csse1001[1] for course logistics — this note covers Lecture 10A's technical content. See uml[2] for the full reference on UML diagram notation, and csse1001-diagram-notation[3] for the drawing conventions this course actually uses.

Outline

- Recap: inheritance vs. composition
- What UML is for
- UML class diagrams: the three-part class box, and visibility
- Relationship diagrams: inheritance, association, multiplicity, composition, aggregation
- A larger, real-world example

Recap: inheritance vs. composition

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/uml.pdf
[3] courses/csse1001/csse1001-diagram-notation.pdf

Feature	Inheritance	Composition
Relationship	"Is-a"	"Has-a"
Flexibility	Less flexible (changes in parent affect children)	More flexible (components can be changed)
Reusability	Extends base class functionality	Contains objects that provide functionality
Example	`Dog` **is a** `Animal`	`Drone` **has a** `Camera`

See 2025-09-23-inheritance[4] and 2025-09-23-composition[5] for where these two came from.

What UML is for

Each person on a team asks a different question about the same design: *"What classes?"*, *"Who calls what?"*, *"Where to pass this?"*, *"When you pass a drone object to…"*. A diagram answers these questions cheaply. Source code answers them expensively.

A **unified modelling language** diagram is the standard way of illustrating relationships among classes. For our `Animal` class, the simplest possible version is four name-only boxes: `Animal` on top, with `Cat`, `Dog` and `Fox` beneath it, joined by one arrow pointing up into the parent.

UML class diagram

A **class diagram** is a type of UML diagram that shows:

- Classes and their attributes/methods.
- Relationships (e.g. inheritance, composition, association).

It is great for object-oriented design.

The class box

A class box has 3 parts, plus a **visibility** marker on every attribute and method:

- **Top section**: class name (e.g. `Animal`).
- **Middle section**: attributes (e.g. `name: str`).
- **Bottom section**: methods (e.g. `speak(): str`).
- **Visibility**: `-` (private), `+` (public), `#` (protected).

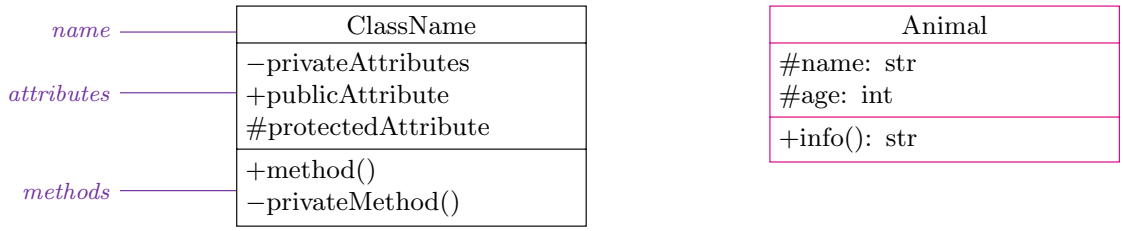

Figure 1: The three-part class box: a generic template on the left and the concrete `Animal` box on the right, outlined in magenta. The purple italic labels name the three parts of the box: name, attributes and methods.

Two details of the notation:

- **An empty compartment is kept, not collapsed.** `Cat` has no attributes of its own, and its box still has a (blank) attributes band. It does not drop to a two-part box.
- **Return types are optional.** The same method can be written `+info(): str` or bare `+info()`. The difference means nothing.

UML inheritance diagram

Each subclass inherits from `Animal` and overrides `speak()`. Inheritance is drawn as a **solid line with a filled triangular arrowhead pointing up into the parent**, with orthogonal (right-angle) routing: one stem up from a horizontal bus, and a vertical drop to each child.

> ⚠ The arrowhead is filled, not hollow
>
> Textbook UML draws generalisation with a **hollow** triangle. CSSE1001 draws it **solid filled**, on every inheritance arrow. A diagram that matches the textbook does not match the course. Follow the course. See csse1001-diagram-notation[6].

Association, multiplicity, composition and aggregation

Four more relationships can be added to the same `Animal`/`Cat`/`Dog`/`Fox` tree, **one** at a time. Figure 3 shows the four additions, each drawn in magenta.

[4] courses/csse1001/2025-09-23-inheritance.pdf
[5] courses/csse1001/2025-09-23-composition.pdf
[6] courses/csse1001/csse1001-diagram-notation.pdf

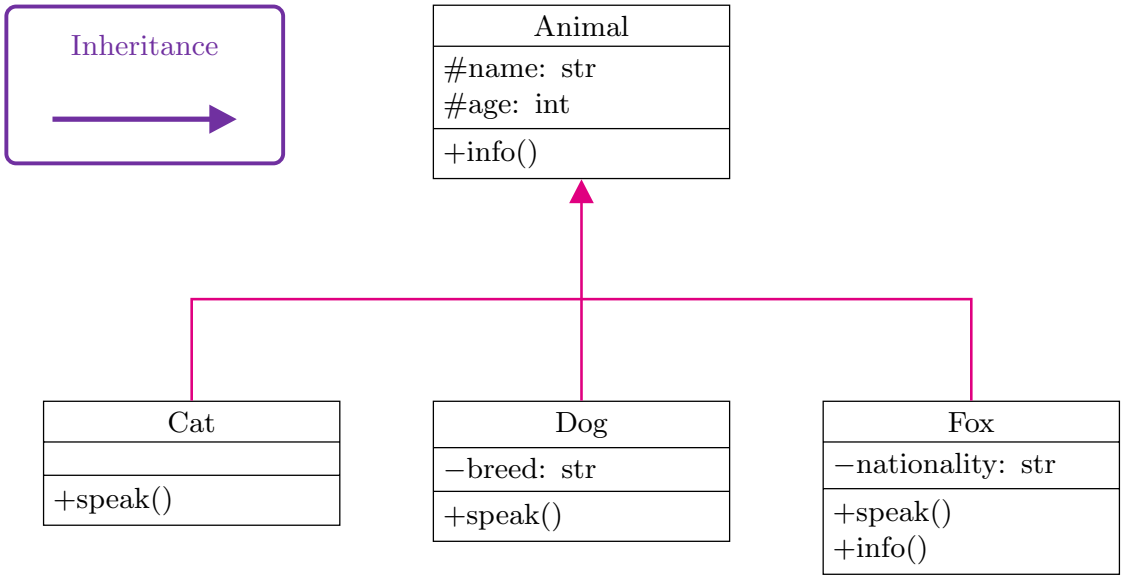

Figure 2: The inheritance diagram for Animal and its three subclasses. Cat, Dog and Fox each override speak(); Fox also overrides info(). Magenta marks the inheritance relationship, and the purple box in the corner is the legend for its arrow. The arrowhead is filled, the lines run through an orthogonal bus, and Cat's attribute compartment is kept although it is empty.

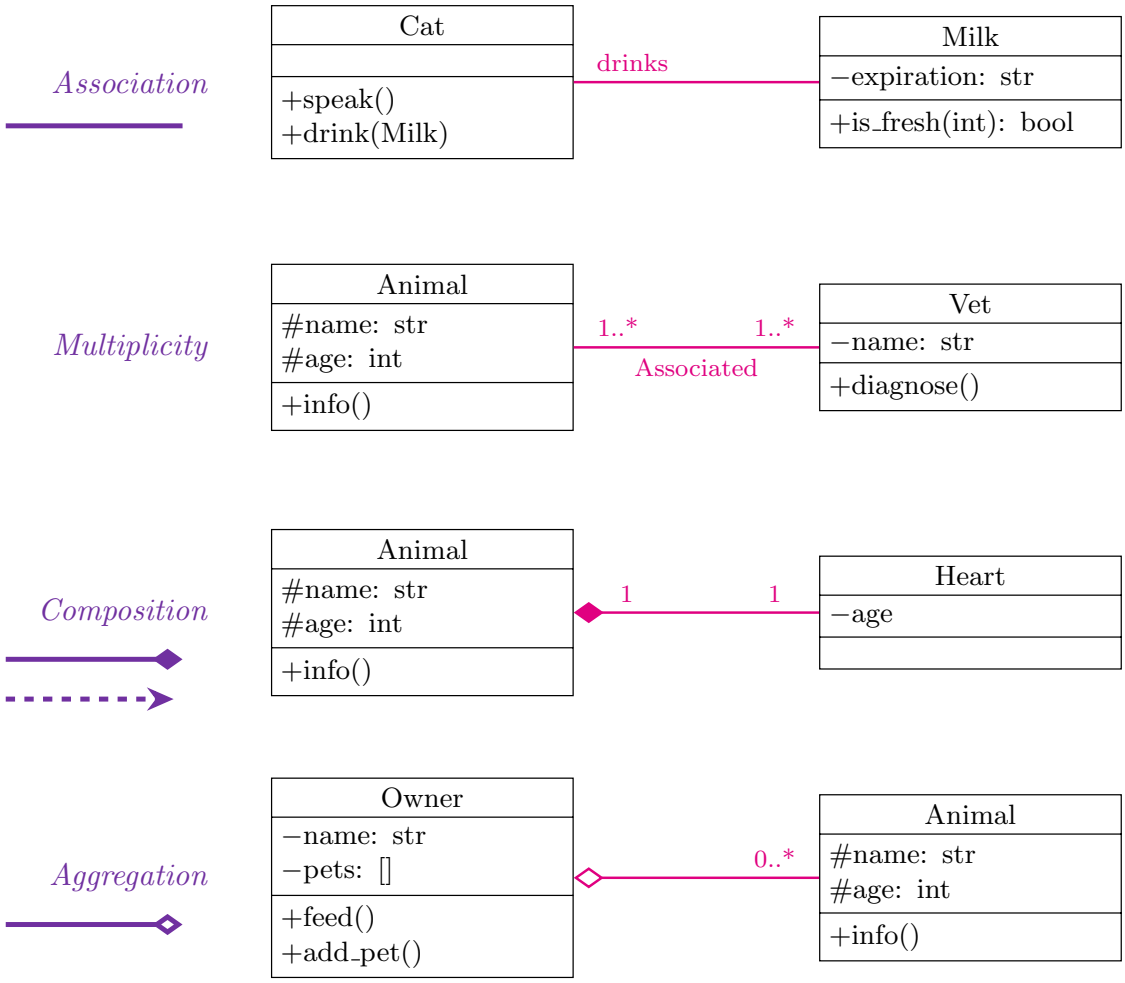

Figure 3: The four non-inheritance relationships, each with its name and symbol in purple on the left; multiplicity has a name and no symbol. Association - Cat drinks Milk. Multiplicity - Animal and Vet, 1..* at both ends, labelled Associated. Composition - Animal has exactly one Heart, filled diamond at the Animal end; the dashed filled-head arrow beneath the diamond symbol is a second symbol with no name. Aggregation - an Owner has 0..* Animals, hollow diamond at the Owner end. Each fragment attaches to the Animal/Cat/Dog/Fox inheritance tree of the previous figure, which is unchanged and so is not repeated.

Association

Association — a plain solid line, no arrowhead — represents one class *using* or *knowing about* another, without owning it. The example is a `Cat` that can `drink(Milk)`: `Cat` gains a `+drink(Milk)` method, `Milk` is a class of its own with `-expiration: str` and `+is_fresh(int): bool`, and the line between them carries the verb label `drinks` at one end.

Multiplicity

Multiplicity shows how many objects are involved in a relationship. The vocabulary:

Multiplicity	Meaning
`0..1`	Zero to one
`n`	Specific number
`0..*`	Zero to many
`1..*`	One to many
`m..n`	Specific number range

The example is `Animal 1..*` —— `Associated` —— `1..* Vet`: an animal can see one or more vets, and a vet treats one or more animals. The counts sit **above the line, next to the end they qualify**; the verb sits below it.

Composition

Composition ("has a", **filled** diamond at the *whole* end): the "part" *cannot* exist independently of the "whole". An `Animal` has exactly one `Heart` — `1` at both ends — and the `Heart` class carries a single attribute `-age`.

Aggregation

Aggregation (**hollow** diamond at the *whole* end) is a weaker form of composition: the "part" *can* exist independently of the "whole". An `Owner` (`-name: str`, `-pets: []`, `+feed()`, `+add_pet()`) has `0..*` `Animal`s, but an `Animal` exists perfectly well with no owner.

The mnemonic that falls out of the two diagrams: **the diamond always sits at the end that owns**, and it is filled when destroying the whole destroys the part.

Exercise

Try modelling a UML diagram for the `DroneFlight` class (from 2025-09-23-composition[7]) and its child `DeliveryDrone` class (from 2025-09-23-inheritance[8]'s exercise).

A larger example

Real systems combine all of these. The diagram below has thirteen classes, and it is the only whole-system class diagram in the course. Going by the class names (`Battlefield`, `Tank`, `Player`, `Enemy`, `Guard`, `Patrol`), the system is a simplified tank-battle game.

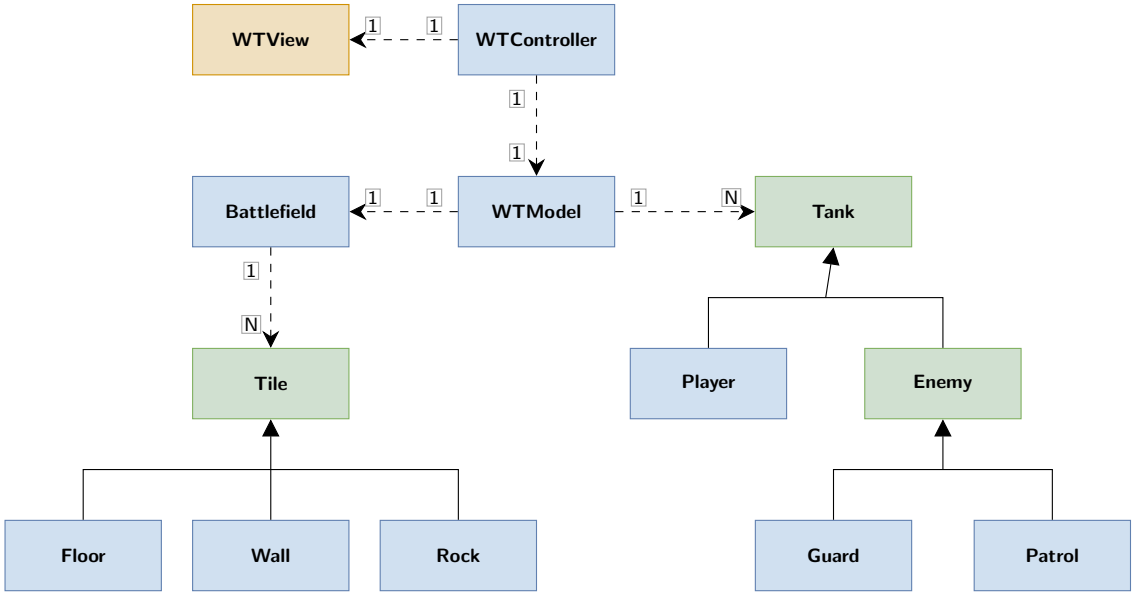

Figure 4: The tank-battle system diagram, with its multiplicities. Dashed arrows are associations/dependencies (WTController to WTView and WTModel; WTModel to Battlefield and Tank; Battlefield to Tile); solid arrows are inheritance. Both heads are solid: the dependency head is a filled concave-backed arrow, the inheritance head the course's filled triangle. csse1001-diagram-notation[9] describes the dependency head as open. The boxes are single-compartment with pastel fills, not the three-part black boxes above. The fill colours have no stated meaning.

Reading it off the diagram:

- One `WTController` is associated with one `WTView` and one `WTModel`.

[7] courses/csse1001/2025-09-23-composition.pdf
[8] courses/csse1001/2025-09-23-inheritance.pdf
[9] courses/csse1001/csse1001-diagram-notation.pdf

- One `WTModel` is associated with one `Battlefield` and with `N` `Tank`s.
- One `Battlefield` is associated with `N` `Tile`s, and `Tile` is specialised into `Floor`, `Wall` and `Rock`.
- `Tank` is specialised into `Player` and `Enemy`; `Enemy` is further specialised into `Guard` and `Patrol` — a three-level chain, which is the multilevel inheritance[10] pattern taught in the very next lecture.

The `WTModel`/`WTView`/`WTController` split is the **Model-View-Controller** pattern covered in 2025-10-14-model-view-controller[11]. The naming and the direction of the arrows show it: the controller depends on both, and the model depends on neither.

Summary

UML is a visual language used to design and describe software systems. In object-oriented programming, class diagrams are one of the most-used UML tools. They help represent the structure of a system by showing classes, their attributes and methods, and the relationships between them — such as inheritance, association, aggregation, and composition. Using UML before writing code makes it easier to plan, communicate, and maintain software designs effectively.

Next: 2025-10-07-advanced-inheritance[12] (Lecture 10B).

[10] courses/csse1001/2025-10-07-advanced-inheritance.pdf
[11] courses/csse1001/2025-10-14-model-view-controller.pdf
[12] courses/csse1001/2025-10-07-advanced-inheritance.pdf