Unified Modelling Language (UML)

Table of contents

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.

Today's outline

- Recap: inheritance vs. composition
- UML class diagrams: attributes, methods, 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

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`

Unified Modelling Language (UML)

A **UML diagram** is the standard way of illustrating relationships among classes — e.g. for our `Animal` class:

```
        Animal
  /       |       \
 Cat      Dog     Fox
```

UML class diagram

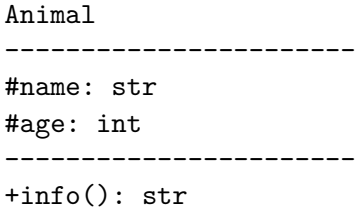
A **class diagram** is a type of UML diagram that shows:

- Classes and their attributes/methods.
- Relationships (e.g. inheritance, composition, association).
- It's great for object-oriented design.

A class box has 3 parts, plus a visibility marker for each attribute/method:

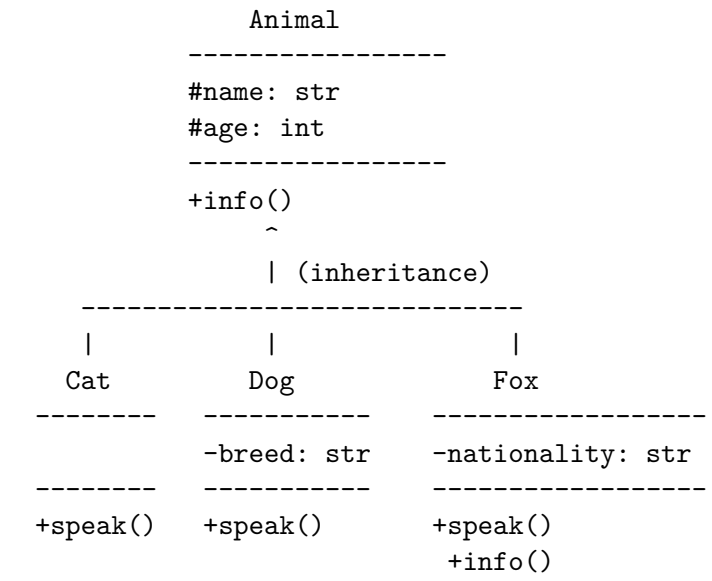
- **Top section**: class name (e.g. `Animal`).
- **Middle section**: attributes (e.g. `name: str`).
- **Bottom section**: methods (e.g. `speak(): str`).
- **Visibility**: `-` (private), `+` (public), `#` (protected).

```
ClassName
-----------------------
-privateAttribute
+publicAttribute
#protectedAttribute
-----------------------
+method()
-privateMethod()
```



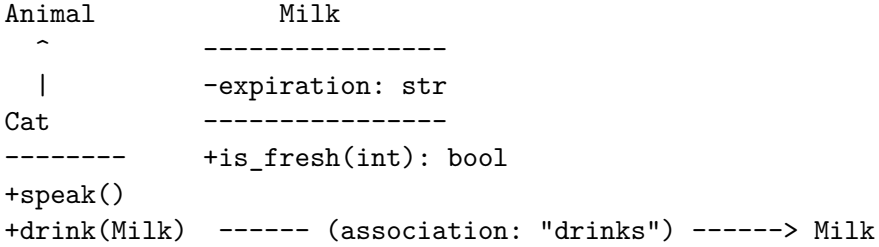
UML inheritance diagram

Each subclass inherits from `Animal` and overrides `speak()`. Inheritance is drawn as a solid line with a **hollow arrowhead** pointing from the subclass to the superclass:



UML association diagram

Association — drawn as a plain solid line — represents one class *using* or *knowing about* another, without owning it. For example, a `Cat` can `drink(Milk)`:



UML multiplicity

Multiplicity shows how many objects are involved in a relationship:

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

For example, an `Animal` can be associated with `1..*` `Vet`s, and a `Vet` is associated with `1..*` `Animal`s:

```
Animal  1..* ------ Associated ------ 1..*  Vet
```

UML composition diagram

Composition ("has a" relationship, filled diamond): here the "part" *cannot* exist independently of the "whole". For example, an `Animal` has exactly one `Heart`:

```
Animal    1     1   Heart
```

UML aggregation diagram

Aggregation (hollow diamond) is a weaker form of composition: here the "part" *can* exist independently of the "whole". For example, an `Owner` has zero or more `Animals` (pets), but an `Animal` can exist without an `Owner`:

```
Owner      0..*   Animal
```

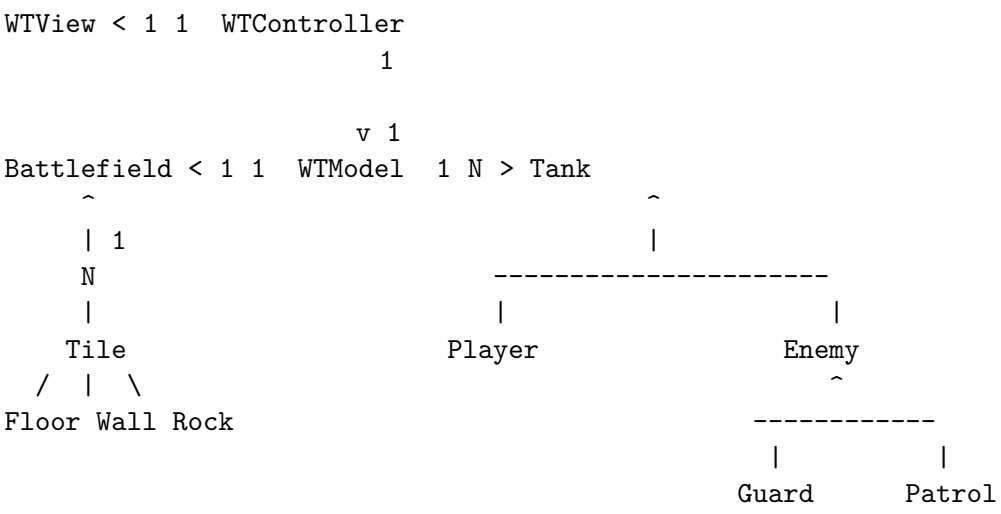
Exercise

Try modelling a UML diagram for the `DroneFlight` class (from 2025-09-23-composition[3]) and its child `DeliveryDrone` class (from 2025-09-23-inheritance[4]'s exercise) — left unsolved in the source.

[3] courses/csse1001/2025-09-23-composition.pdf
[4] courses/csse1001/2025-09-23-inheritance.pdf

A larger example

Real systems combine all of these relationships. A simplified tank-battle game architecture might look like:



The dashed arrows here represent *dependencies* between the controller, model, and view — this is the classic **Model-View-Controller (MVC)** pattern, which we'll cover properly next week.

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[5] (Lecture 10B).

[5] courses/csse1001/2025-10-07-advanced-inheritance.pdf