UML class diagrams

Tamás Ambrus, István Gansperger

Eötvös Loránd University
ambrus.thomas@gmail.com, gansperger@gmail.com

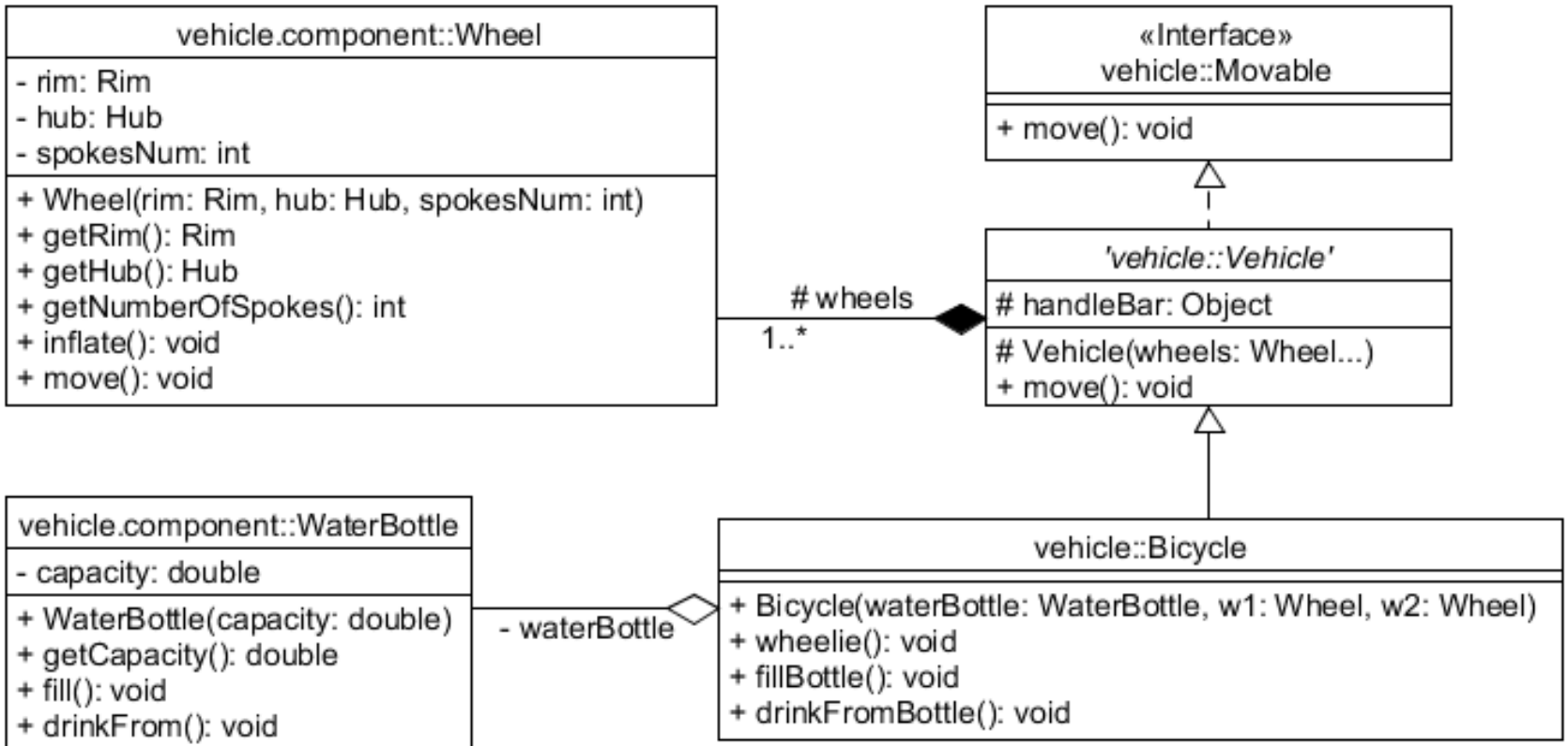
Which one do programmers prefer?

Textual representation:

Create a Bicycle type which is a Vehicle and a Movable. Bicycle should have 2 wheels, a handlebar and a water-bottle. As this is a Vehicle which is Movable, it inherits the move method. Implement the move and the wheelie method.

Which one do programmers prefer?

Part of UML representation:



Unified Modeling Language

Misunderstanding

What is wrong with the textual representation?

It is **always** incomplete. If you give a solution for the textual representation of a task, users will blame you, because you did not solve the problem as they thought. The model of the problem is different in their mind and different in yours.

UML

One of the purposes of Unified Modeling Language was to provide a stable and common design language that can be used to develop computer applications. To achieve this, UML provides many diagrams that can be categorized in two groups:

- structural diagrams: that represent the static aspect of the system,
- behavioral diagrams: that represent the dynamic aspect of the system.

Diagrams

Structural diagrams are:

- **class diagram**,
- object diagram,
- component diagram,
- ...

Behavioral diagrams are:

- **use case diagram**,
- sequence diagram,
- statechart diagram,
- ...

UML, well done!

Goal achieved

Using UML diagrams, we are really precise. Far less mistakes can be made. UML diagrams almost solve the problem, and additionally, they are easy-to-understand.

Using diagrams, the your mind is closer to the users' expectation, and that's the point.

UML class diagram (structural d.)

Aim

Class diagrams describe the structure of a system by showing the system's types, their attributes and methods.
They also specify the relationships between types and objects.

Definition

Class diagrams are directed, connected graphs, the vertices of which represent the types and the edges of which represent the relationships between types and objects.

Vertices

The vertices represent the types.
We use boxes for types divided to 3 parts:

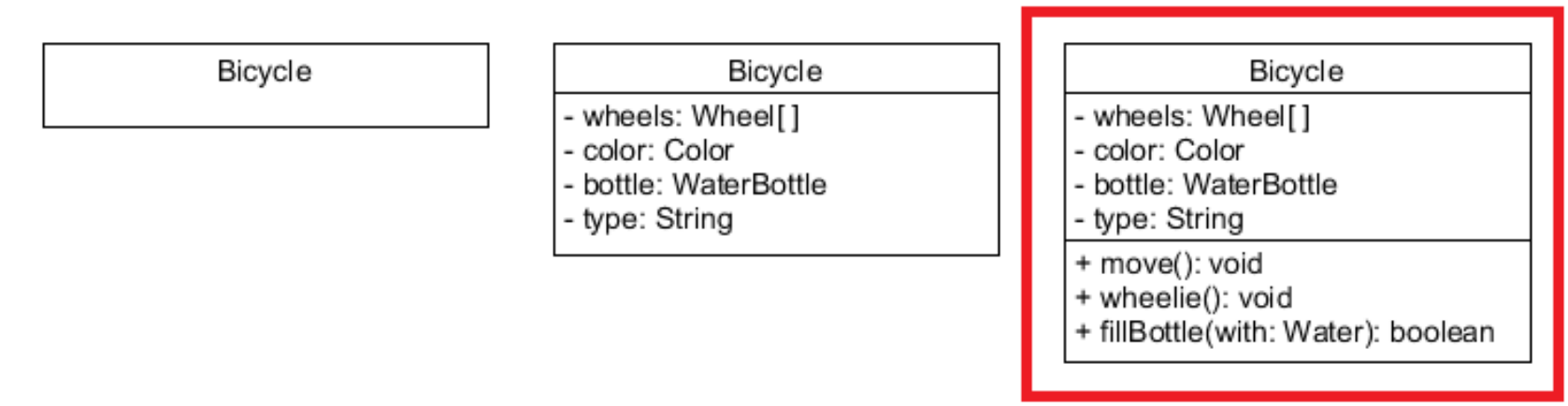
- top part: the name of the type (and additional information, like: interface, package name),
- middle part: in which attributes take place with their visibility, type and name,
- bottom part: for the methods with visibility, name, return type and parameters.

Note,

that the implementations of the methods are missing. Only a part of the signature is indicated.

Example type

Class from class diagram:



Class from Java code:

```
public class Bicycle {
    private Wheel[] wheels;
    private Color color;
    private WaterBottle bottle;
    private String type;

    public void move() { ... }
    public void repair() { ... }
    public boolean fillBottle(Water with) { ... }
}
```

Oh, teacher, just a few questions left

Notes

- how do we indicate whether a type is abstract?
- or interface?
- or a method is abstract?
- do you have any repr. of a static field?
- what is the representation of the visibilities?
- do we put constructors into a class diagram?
- but first: why do we need class diagrams if there are no implementations?

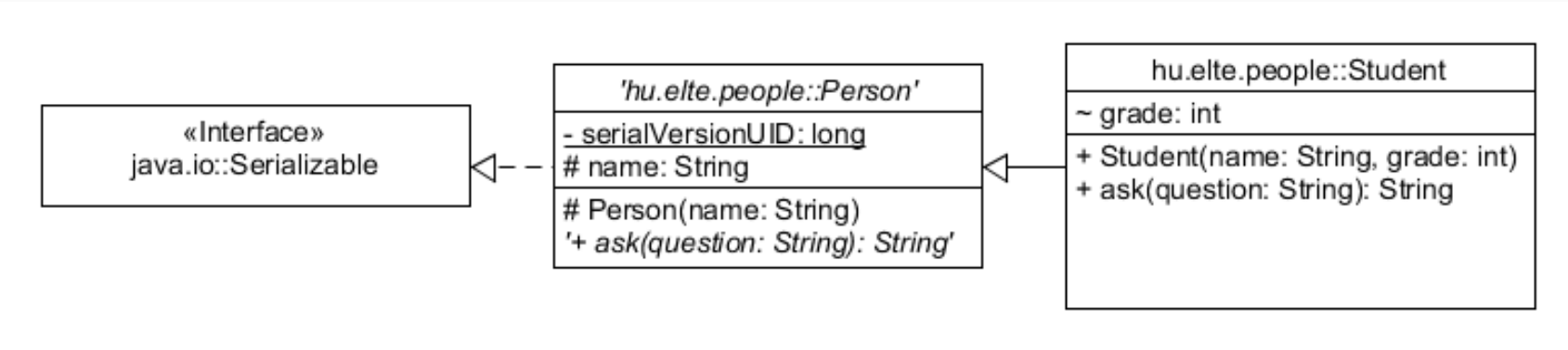
Answer for the most important question

Alright,

- class diagrams are worth at a planning session as a "plan" can be concretized
- if you have a new task and you couldn't draw an elementary class diagram, *you most likely are in trouble*
- as a senior programmer you often have to help juniors, e.g. by drawing a helper class diagram
- when getting a legacy code, diagrams speed up the understanding of the system
- it makes your model easier to understand

Sounds great!

What were the remaining questions?



Notation

- abstract type, abstract method: apostrophe (') + italic style
- interface: <<Interface>>
- static field, static method: underline
- public: +, private: -
- protected: #, package private: ~
- contructors must be shown in the class diagram as they are methods

Edges

The edges represent the relationships.

There are 11_2 types of relationships:

- dependency,
- association,
- generalization.

Dependency

Dependency is a lightweight semantic connection between two types.

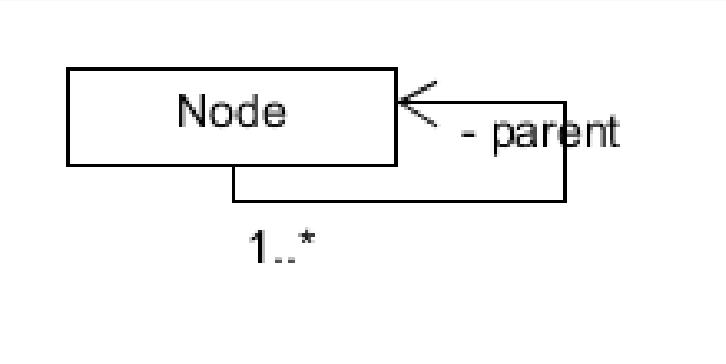
- we use this if a type uses another only for a parameter type or a return type
- changes to the definition of the contained type may cause changes in the container type
- uni-directional relationship


```
public class Journalist {
    public Article writeAbout(Topic t) { ... }
}
```

Association

Association is a bit stronger semantic connection between two types.

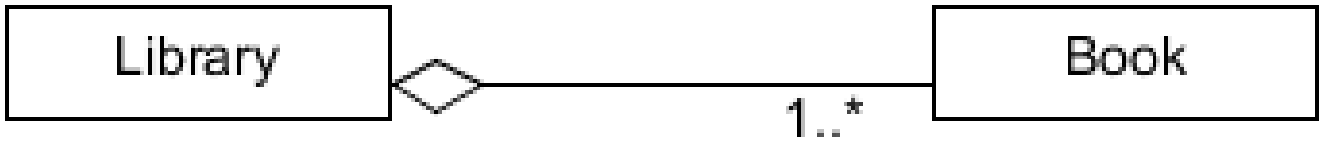
- we use this if a type stores another as class field; but only like as a helper object, it's not markedly containing
- can be uni- or bi-directional or even reflexive
- the ends of the association can be labeled with role names, multiplicity and visibility
- also may have other properties attached to it
- association has subtypes: **aggregation** and **composition**



Aggregation

Aggregation is "has a" relationship, it is more specific than ordinary association because "has a" means that a type **is part of** the container one.

- the contained type's lifecycle does not depend on the container type's
- if we destroy the container type's object, the contained type's one is allowed to stay alive

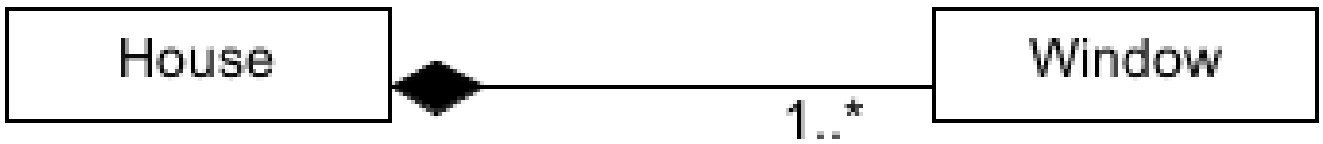



```
public class Library {
    private List<Book> books;
}
```

Composition

Composition is the strongest relationship between two types.

- the contained type's lifecycle depends on the container type's
- if we destroy the container type's object, the contained type's one is getting destroyed as well
- *nothing changes in code, this is only a logical difference*

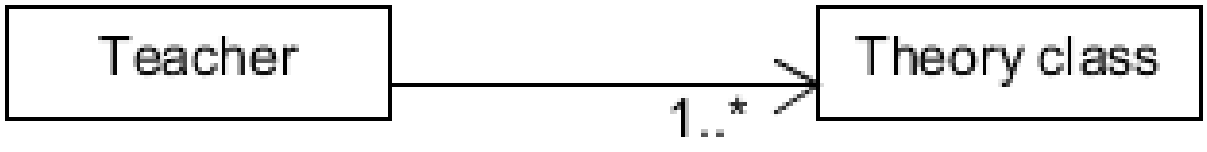


Multiplicity

Associations have multiplicities on each end:

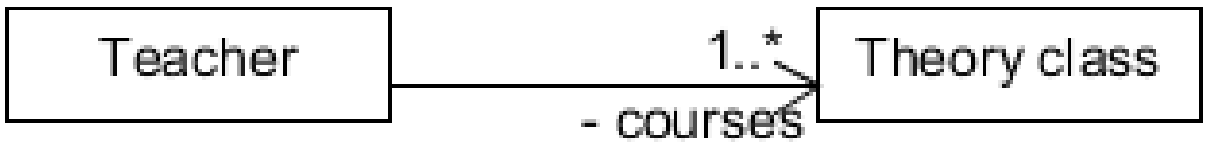
- 0 (zero)
- * (either zero or more)
- 1..* (at least one)
- 4..7 (it's between 4 and 7)

The default multiplicity for an end of the association is 1. For instance, here the teacher has at least one theory class, but only one teacher holds each.



Roles

We can specify the name of the contained objects with role names.

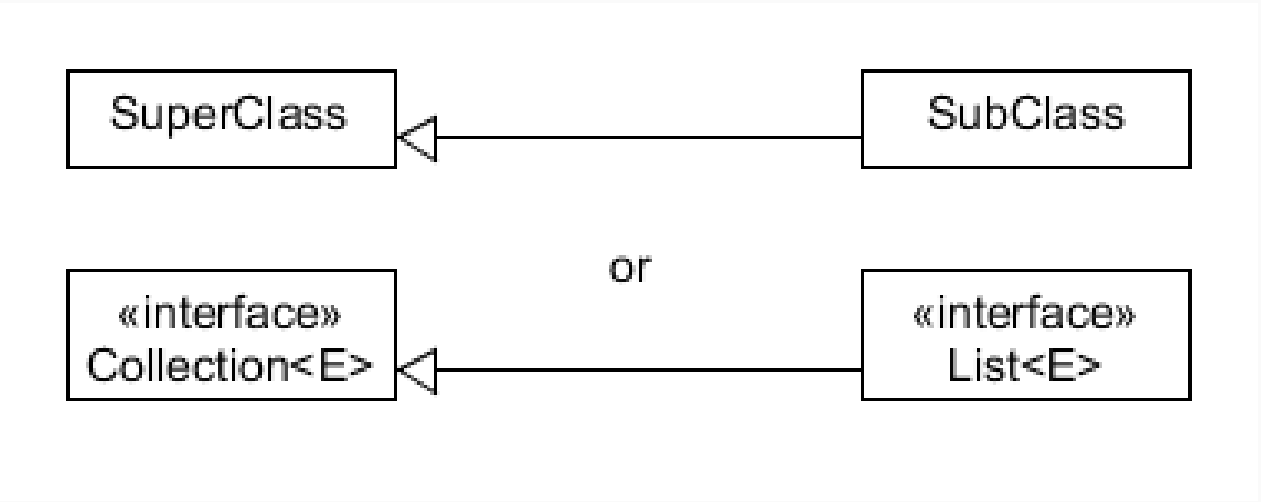



```
public class TheoryClass {}

public class Teacher {
    private Set<TheoryClass> courses;
}
```

Generalization

Generalization is the "is a" relationship between classes/interfaces. This is how you indicate "extends" on the class diagram.




```
public class SuperClass {}
public class SubClass extends SuperClass {}

public interface Collection<E> {}
public interface List<E> extends Collection<E> {}
```

Realization - special generalization

Realization is the special generalization, it occurs between a class and an interface only. This is how you indicate "implements" on the class diagram.


```
public interface List<E> {}
public class ArrayList<E> implements List<E> {}
```

Relationship - which one to draw

The goal is to draw always the most fitting relationship.

- you use a type in a method as a parameter or return value and nowhere else? $\rightarrow$ **use dependency**,
- you decided to put the variable to the class fields to make easier your work? $\rightarrow$ **use ordinary association**,
- you have a class field but if you destroyed the container object, the contained one wouldn't lapse? $\rightarrow$ **then it's an aggregation**,
- the contained object cannot be used for anything without the container object? $\rightarrow$ **definitely a composition**.

Notes

What cannot you read from a class diagram?

- whether the class, the variable or the method is final,
- what kind of exceptions a method may throw,
- how does the state of an object change,
- what is the method body,
- and so on.

If something is not clear, programmers should ask more the users or managers to clarify all the requirements.

References

- http://www.tutorialspoint.com/uml/uml_basic_notations.htm