Classes

Jana Kostičová

Class

= a template for creating objects

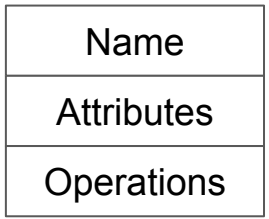
- It defines the **properties (attributes)** and **behaviors (methods, operations)** that objects (instances) of that class will have
- Classes are fundamental building blocks in software design, enabling modular and organized development. They help manage complexity and promote reusability across different parts of a system

UML Class diagram:

= a structure diagram (it shows the static structure of the objects in a system)

Class = a classifier whose features are attributes and operations

- Only class name is mandatory
- Abstract class - the name is in italics or the annotation {abstract} is used after or below its name



UML Class diagram: modeling perspectives

**Three modeling perspectives according to Daniels[2002]*

Conceptual / domain model

- Represents the concepts in the domain
- Software is not yet modeled

Classes, possibly with attributes and high-level operations
Associations, multiplicities

"Specification" (logical application) model

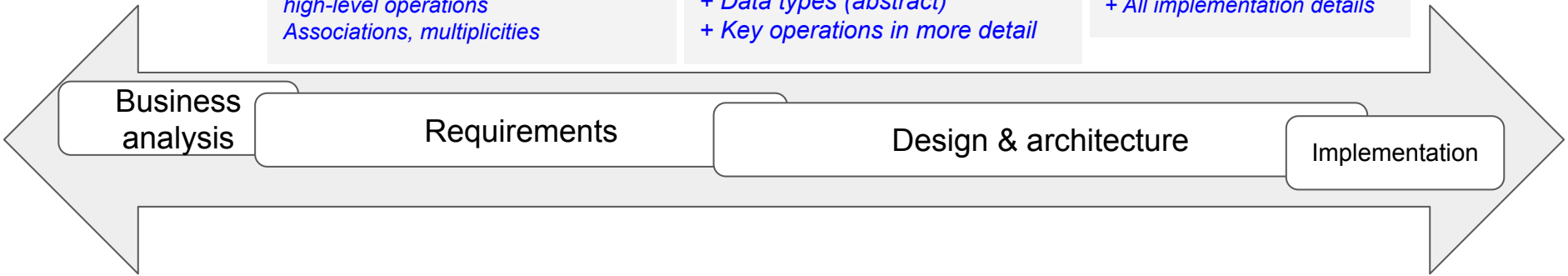
- Focused on software but still independent from implementation
- Captures data types (abstract)

+ Data types (abstract)
+ Key operations in more detail

Implementation model

- Represents specific implementation

+ All implementation details



- A perspective can be anything between the simplest form of conceptual model and the most detailed form of implementation model
- It is first necessary to determine which perspective will be used and to follow it consistently when modeling

UML models vs Code

UML is language-agnostic

Common tasks:

- Generating code from model (forward engineering)
- Generating model from code (reverse engineering)

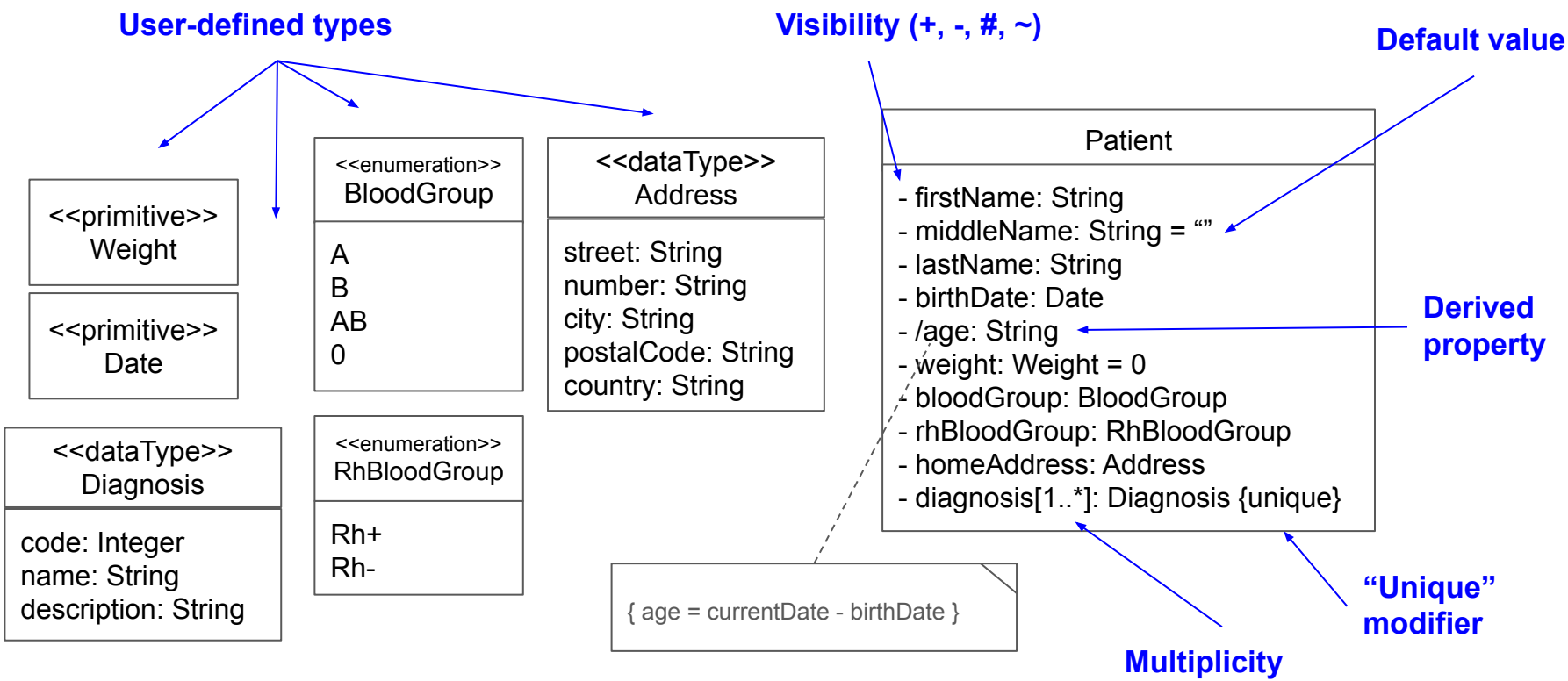
Mappings to/from target language need to be defined.

Tools: Enterprise Architect, Visual Paradigm, IBM Rational Software Architect

Model-driven development (MDD):

- Popular in the 2000s, but not so much anymore
- Related trends: E.g., low code / no code platforms

Data types - examples



Data types

UML built-in primitive types: Boolean, Integer, UnlimitedNatural, String, Real

- Their purpose is primarily to define the UML itself (they are used in metamodels)

User-defined data types (`<<dataType>>` stereotype)
= a classifier (similar to class) whose instances are identified only by their value

- A data type can have internal structure (attributes)
- A data type can have operations

Rule of thumb: If you need to add operations to a data type, it may indicate you need a class instead.

So how are structured data types different from classes?

There cannot exist two instances of the data type with the same structure and the same values (they are considered equal).

Specializations:

1. Primitive type **(`<<primitive>>` stereotype)**
   - Without any substructure.
   - It may have an algebra and operations defined outside of UML, for example, mathematically.
2. Enumeration **(`<<enumeration>>` stereotype)**
   - List of values

Commonly development teams have a convention that they can use standard data types from a particular programming language as data types in UML, especially in implementation models.

Class attributes in UML (1)

[visibility] [‘/’] name [‘:’ type] [‘[‘ multiplicity ‘]’] [‘=’default] [‘{‘ prop-modifier [‘,’ prop-modifier]* ’}’]

Visibility

- **‘+’** public, **‘-’** private, **‘#’** protected, **‘~’** package

‘/’ - derived property

- = Property which can be computed from other properties

Type

- UML built-in primitive types or user-defined data types

Multiplicity

- Positive number (0, 1, 2, …),
- Interval: lower-bound ‘..’ upper-bound; ‘*’ for infinite upper bound

Default

- An expression for the default value(s) of the property

In UML, class attributes are represented by properties.

Class attributes in UML (2)

[visibility] [‘/’] name [‘:’ type] [‘[‘ multiplicity ‘]’] [‘=’default] [‘{‘ prop-modifier [‘,’ prop-modifier] ‘}’]*

Property modifier

‘readOnly’: Read-only property

‘ordered’: Ordered property

‘unique’ / ‘nonunique’: Duplicates not allowed / allowed in a multi-valued property.

‘redefines’ property-name

- Property redefines an inherited property identified by property-name

‘subsets’ property-name

- Property is a proper subset of the property identified by property-name

‘union’: Property is a derived union of its subsets

prop-constraint

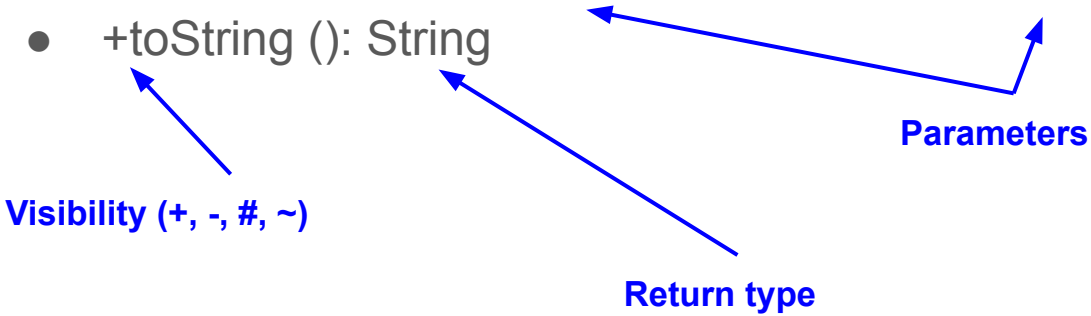
- An expression that specifies a constraint that applies to the property.

Language-specific constraints can be used

Static attributes are underlined

Class operations - examples

- #display ()
- -hide ()
- +createWindow (location: Coordinates, container: Container [0..1]): Window
- +toString (): String



UML visibility vs OO languages

	public	private	protected	package
Java	`public` (exact match)	`private` (exact match)	`protected` (similar, protected + package)	***no keyword*** (default - exact match)
C++	`public` (exact match)	`private` (exact match)	`protected` (exact match)	**N/A**
C#	`public` (exact match)	`private` (exact match)	`protected` (similar)	`internal` (similar, single assembly visibility)
Python	`name` (convention only)	`__name` (name mangling - can be accessed as `_Class__name`)	`_name` (convention only, similar)	**N/A**
JavaScript	`name`	`#name`	**N/A**	**N/A**

Class operations in UML (1)

[visibility] name '(' [parameter-list] ')' *[':' return-spec]*

parameter-list ::= parameter['**,**' parameter]*

parameter ::= [direction] parm-name ':' type-expression['['multiplicity']'] ['='** default] **['{'** parm-property['**,**' parm-property]* **'}']

- Visibility
  - As for attributes:
  - **'+'** public, **'-'** private, **'#'** protected, **'~'** package
- Parameters
  - Direction: 'in', 'out', 'inout' (default = 'in')
  - Type-expression: specifies the data type of the parameter
  - Multiplicity, default, parm-property: as for attributes

Class operations in UML (2)

[visibility] name '(' [parameter-list] ')' [':' return-spec]

return-spec ::= [return-type] ['[' multiplicity ']'] ['{' oper-property [',' oper-property]* '}']

- Return type
  - UML built-in primitive type or user-defined DataType / Class / Interface
- Operation properties (modifiers):

  'query' The operation does not change the state of the system

  'ordered' The values of the return parameter are ordered

  'unique' The values returned by parameters have no duplicates

  'redefines' oper-name:
  - The operation redefines an inherited operation identified by oper-name

  oper-constraint:
  - A constraint that applies to the operation.

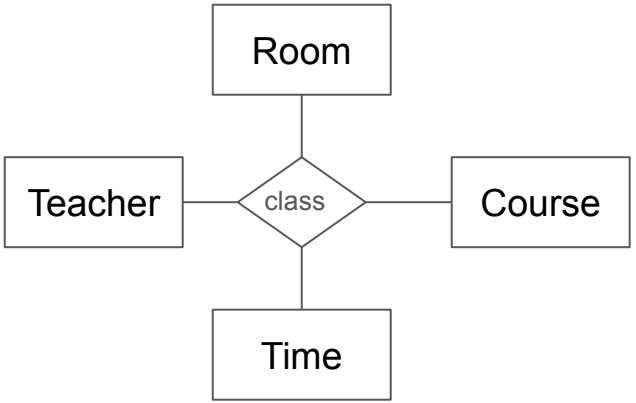
Language-specific constraints can be used

- Static operations are <u>underlined</u>

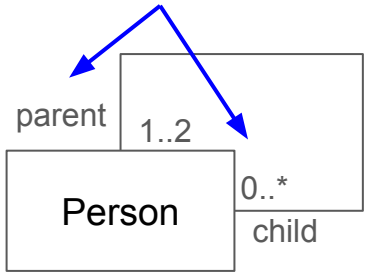
UML association - examples

Association name with “►” symbol indicating the order of association ends and reading

4-ary association



Association ends



Self-association

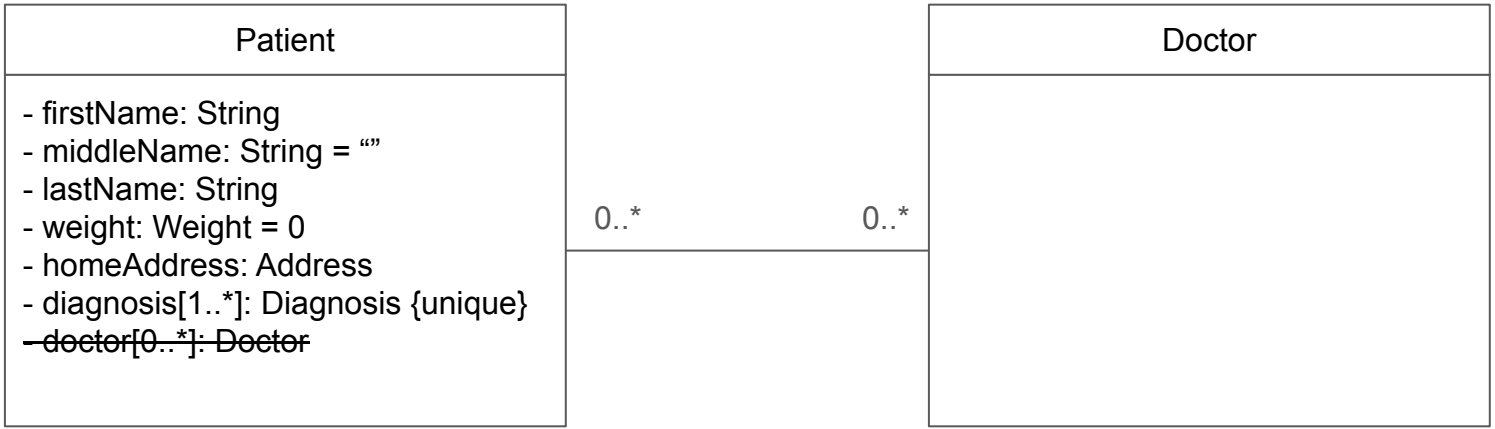
UML association

= a relationship between classes that indicates that there can be links between objects of given classes

- Association name (optional)
  - A solid triangle next to the association name indicates the order of association ends as well as the order of reading (used especially in conceptual modeling)
- Association ends
  - Association has at least two ends
  - Self-associations are allowed
  - Each association end is represented by a property - i.e., properties represent both class attributes and association ends
- Association and its ends may be derived; marked by ‘/’ before their names.

Attribute vs association

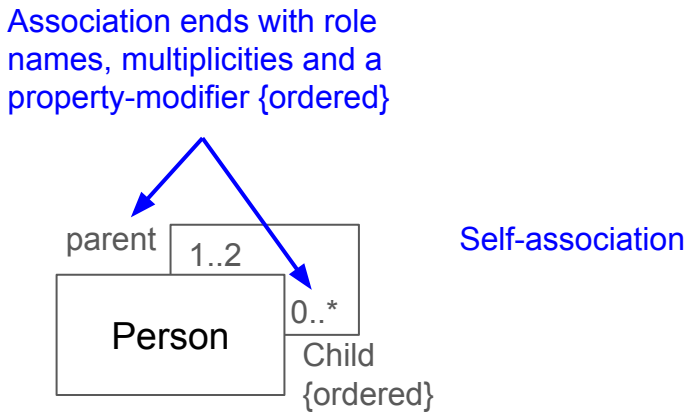
“A useful convention for general modeling scenarios is that a Property whose type is a kind of Class is an Association end, while a property whose type is a kind of DataType is not. This convention is not enforced by UML.” (*UML Specification 2.5.1)*



- String, Weight, Address and Diagnosis are DataTypes → attributes
- Doctor is a class → association

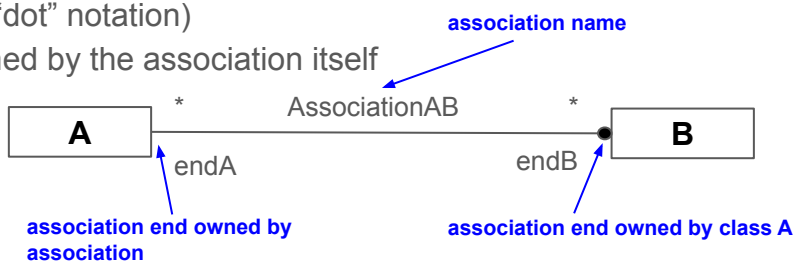
Association end

- Association role name
  - It indicates what role the object of the given class plays within the relationship
- Multiplicity, visibility
  - As for attributes
- Aggregation kind
  - Aggregation vs composition, see later
- Property string (in curly braces)
  - As *property-modifier* for attributes



Association end - ownership and navigability

- **Ownership**
  - *Association end endB is owned by class A*
  - Association end can be owned by one of the connected classes or by the association itself
  - Indicated by a small circle (“dot” notation)
  - If not shown, the end is owned by the association itself



- **Navigability**

  - *At runtime, objects of class B can be accessed efficiently from objects of class A*
  - Association ends owned by classes are always navigable
  - Association ends owned by associations may be navigable or not

UML aggregation and composition

- Special types of associations
- They have an “aggregation kind” set:
  - Shared → aggregation
  - Composite → composition

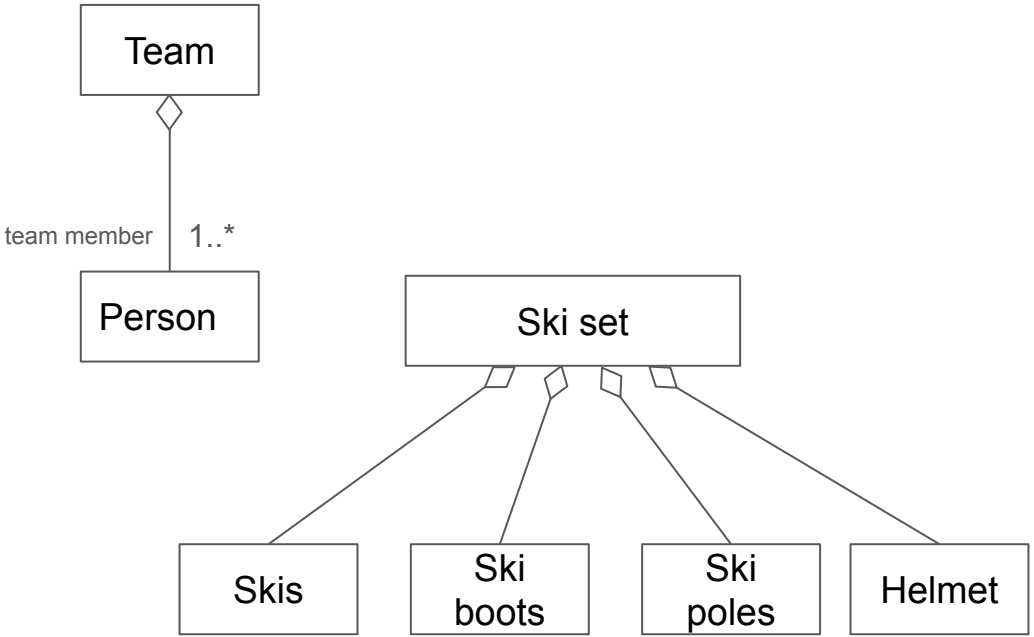
Aggregation

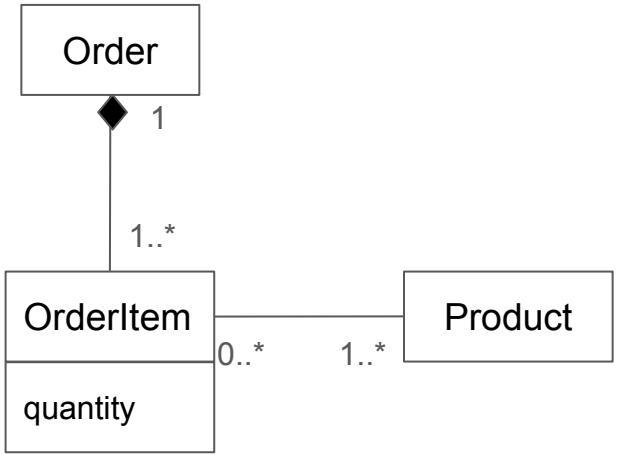
- Weak relationship between the whole and its parts - part can exist without the whole

Composition

- Strong relationship between the whole and its parts - part cannot exist without the whole
- If a composite is deleted, its parts are typically deleted as well

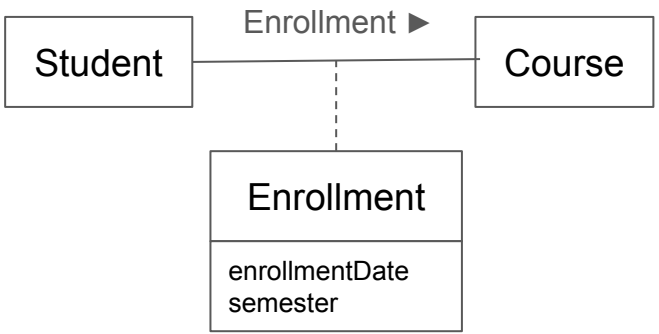
Examples


Team
team member
1..*
Person
Ski set
Skis
Ski boots
Ski poles
Helmet


Order
1
1..*
OrderItem
quantity
0..*
1..*
Product

Association class

- A class representing an association, allows adding attributes and operations to the association
- It has the same name as the association

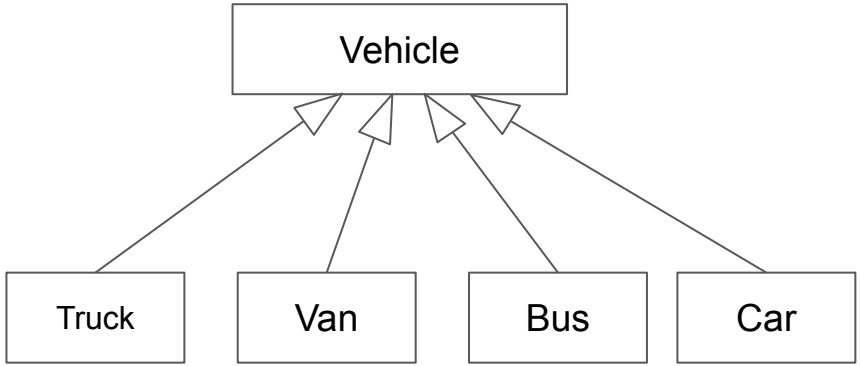


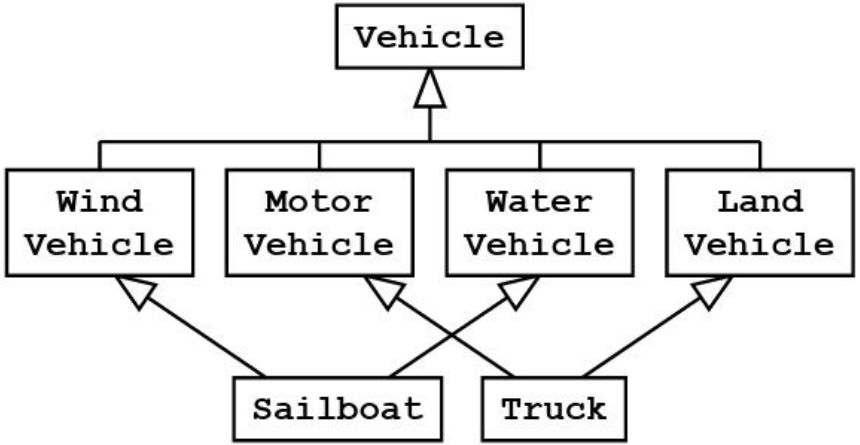
UML generalization

= the taxonomic relationship between a more general class and a more specific class

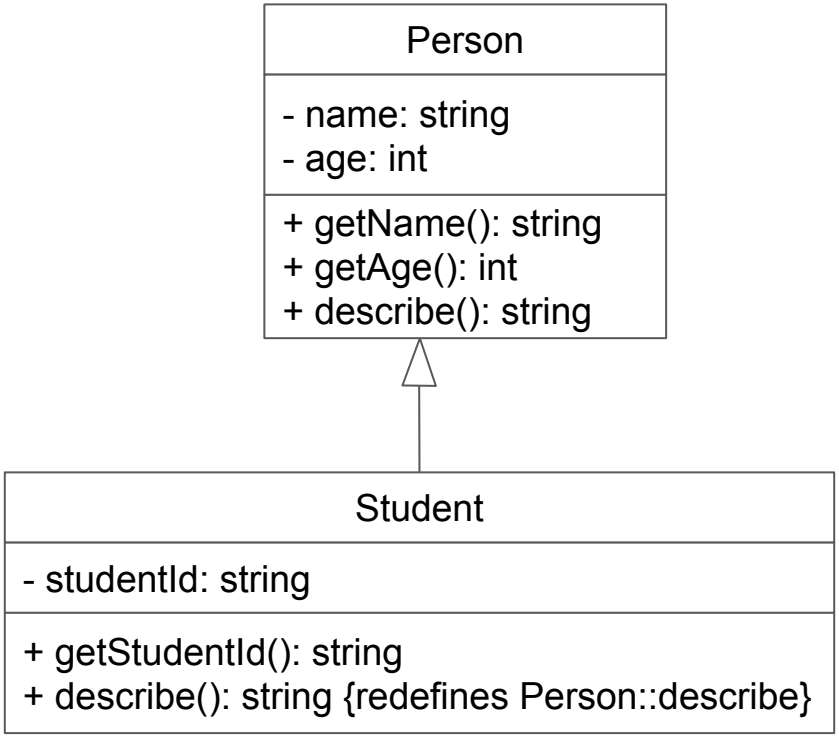
- The specific class inherits the features of the more general class
- A class can inherit from multiple general classes
  - Not allowed in some OO languages (Java, C#). During code generation, interfaces are typically used for second, third, …, n-th superclasses.

Examples:





Another example

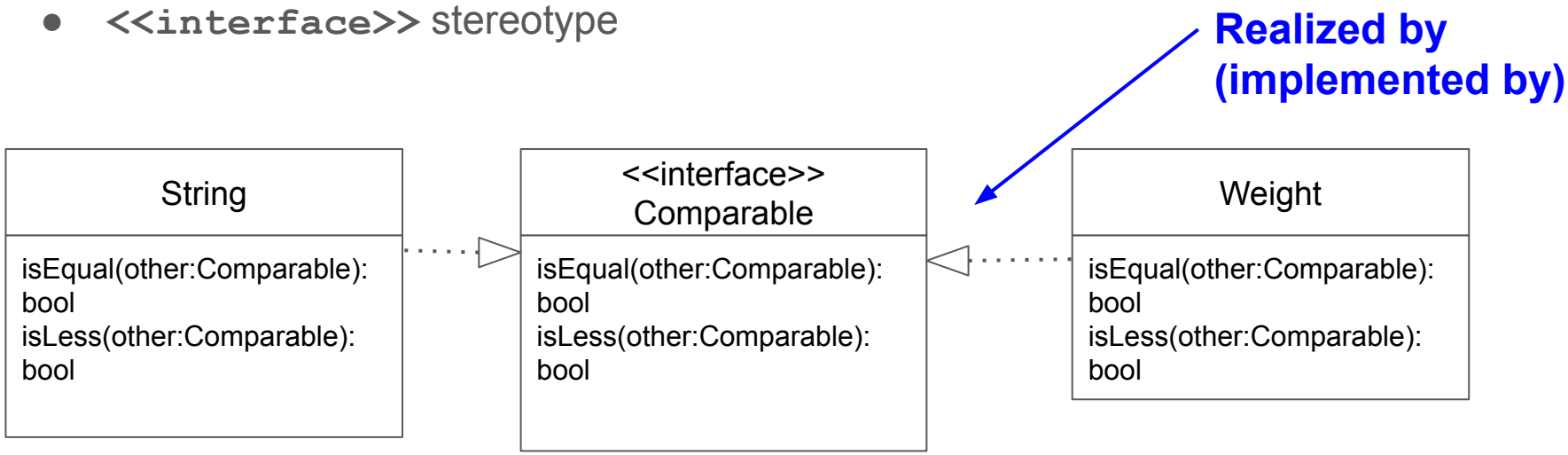


- Student inherits name, age, getName(), getAge(), describe()
- Student overrides describe()

UML interface

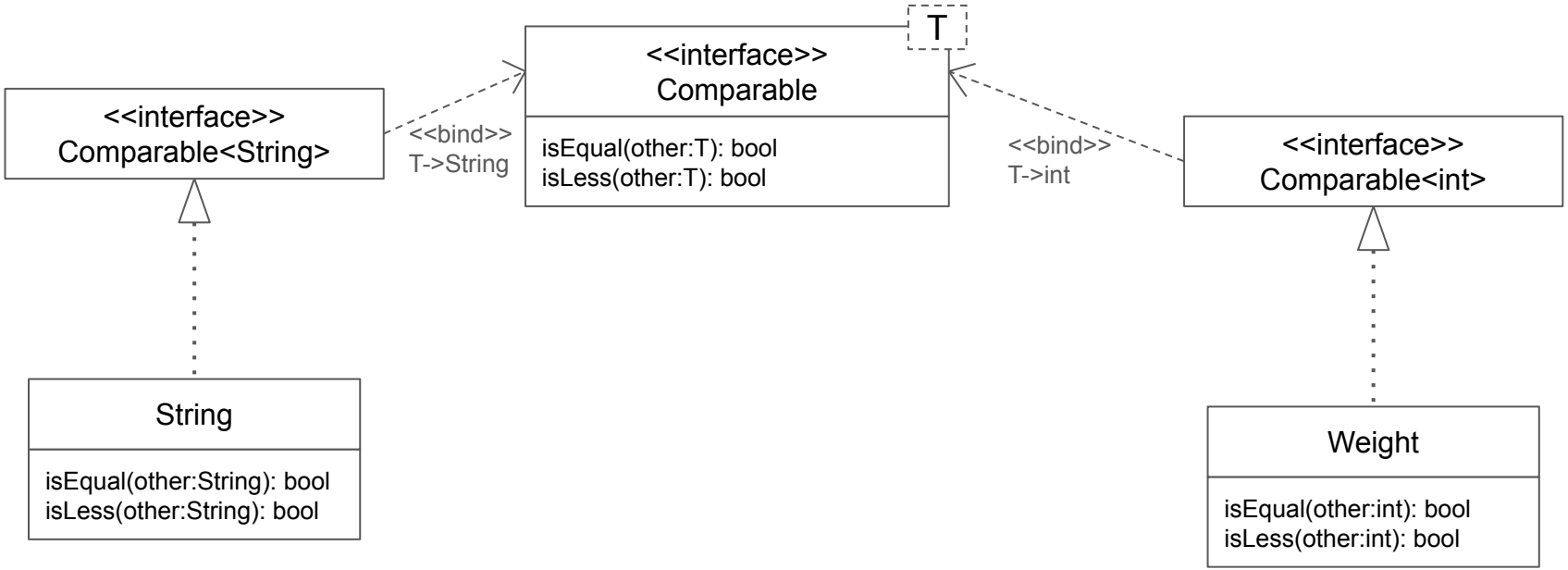
= a declaration of a coherent service (features and obligations)

- Not instantiable
- Can be implemented by an instantiable classifier
- **<<interface>>** stereotype



UML Templates

- Allow classifiers to be parameterized with template parameters
- Limitations - see example



Dependency relationship

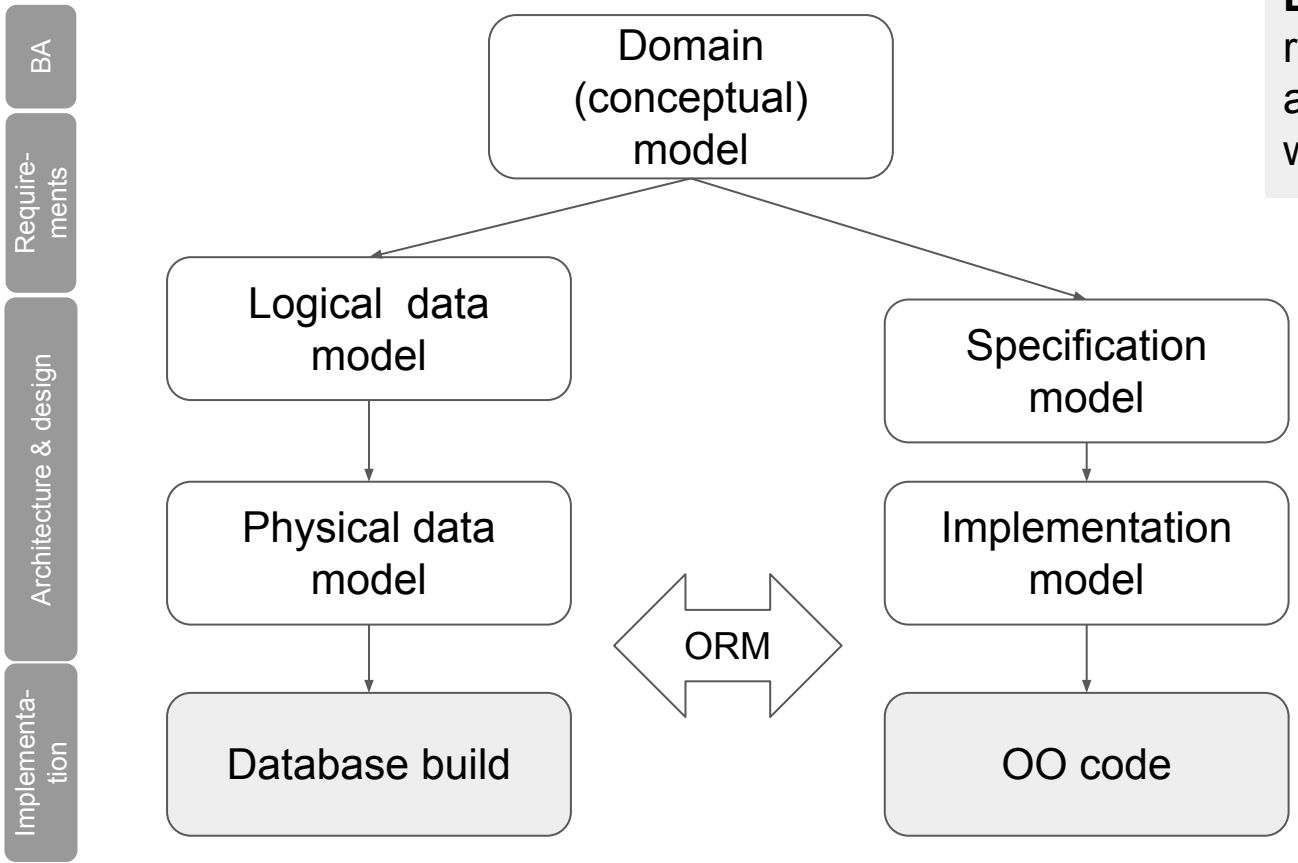
= a relationship where one class **(the client)** relies on another class **(the supplier)** because it uses its services or functionality

- Can be used in multiple UML diagram (class, component, deployment, use-case)
- Notation:

- Examples
  - Client class uses a supplier class that has global scope
  - Client class uses a supplier class as a parameter for some of its operations
  - Client class uses a supplier class as a local variable for some of its operations
  - …

Data modeling vs OO modeling

Data model = visual representation of data and its relationships within a database



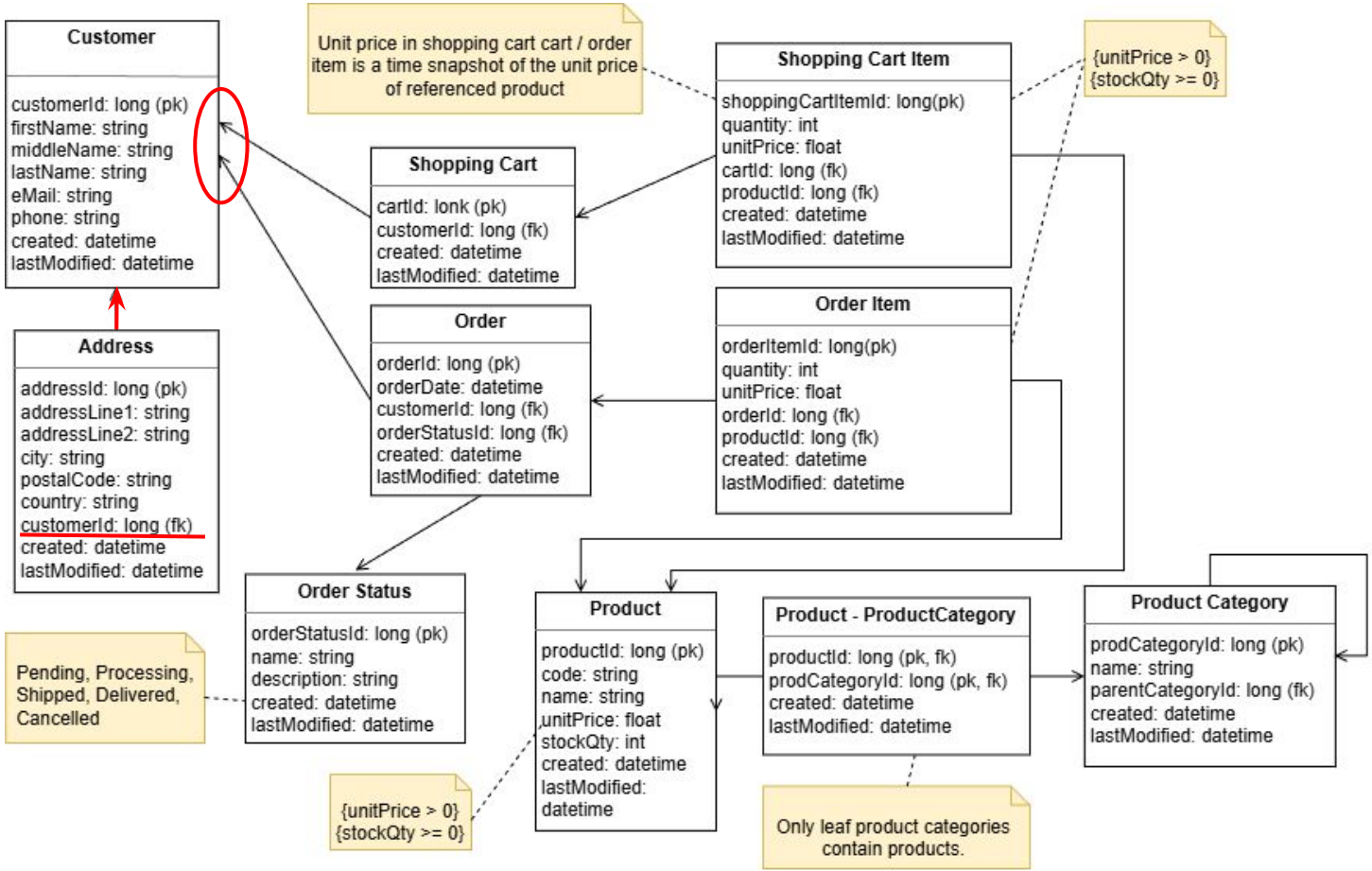
UML Class diagram in data modeling

- Not originally intended for this purpose
- .. but often used for this purpose
  - Class ≈ entity/table
  - Attributes ≈ columns
  - Associations ≈ foreign keys
  - Operations usually omitted

Logical relational data model

In practice, it is often accepted to combine two notations for references (attribute + association)

Navigability is commonly used to denote the direction of a reference



Further reading

- R. Lukoťka: Domain analysis, modeling (PTS1)
- S. Ambler: UML Class Diagrams: An Agile Introduction
- M. Fowler: UML Distilled: A Brief Guide to the Standard Object Modeling Language 3rd Edition, 2003

References

- OMG. OMG Unified Modeling Language. Version 2.5.1, December 2017
- K. Fakhroutdinov: UML Class and Object Diagrams Overview
- R. Červenka, UML Classes
- J. Daniels: Modeling with a Sense of Purpose, 2002
- B. Selic: Getting It Right on the Dot, 2013