Construction of User Interfaces (SE/ComS 319)

Ali Jannesari

Jinu Susan Kabala

Department of Computer Science

Iowa State University, Spring 2021

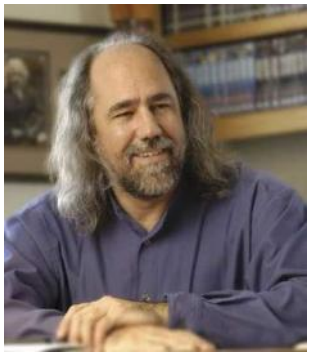
UML DIAGRAMS

Outline

- UML diagrams
- Class diagrams
- Interaction diagrams
  - Sequence diagrams
- State diagrams
- Package diagram

What is UML?

- UML: "Unified Modeling Language”
- UML is the union of three notations from object-oriented modeling:
  - Booch (Grady Booch)
  - OOSE: Object-oriented Software Engineering (Ivar Jacobson)
  - OMT: Object-Modeling Technique (James Rumbaugh)

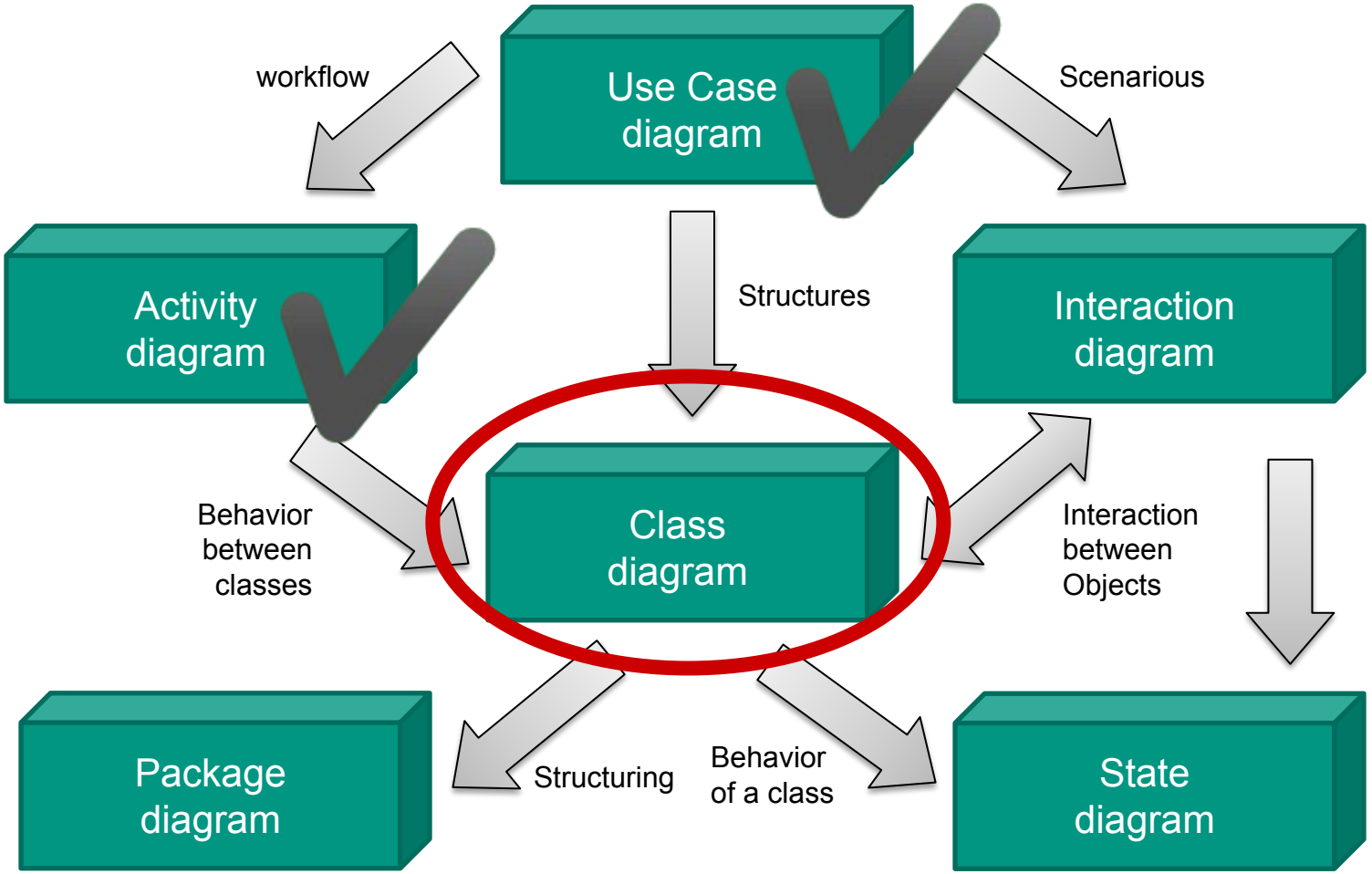


Grady Booch

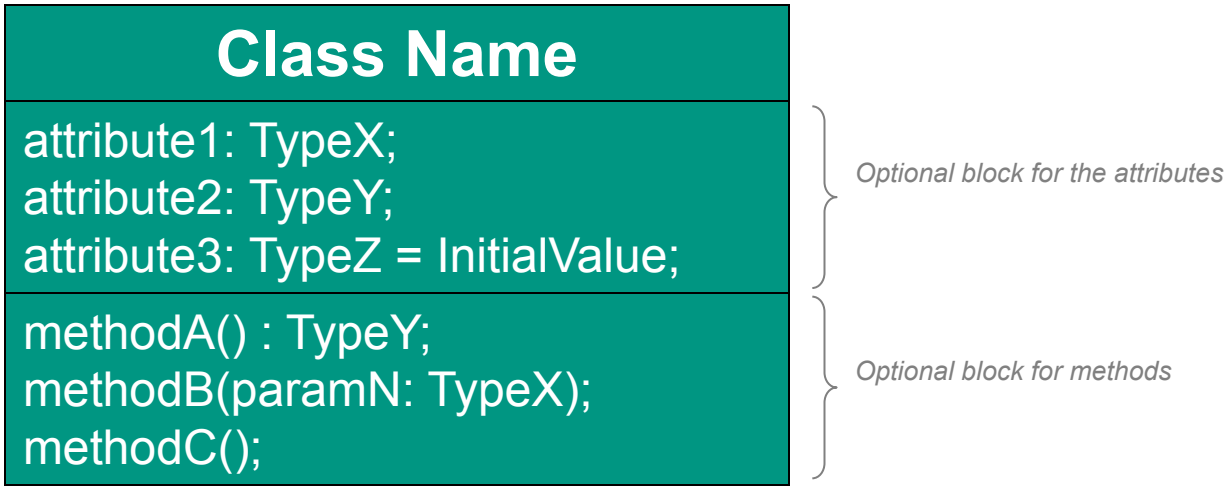
Ivar Jacobson

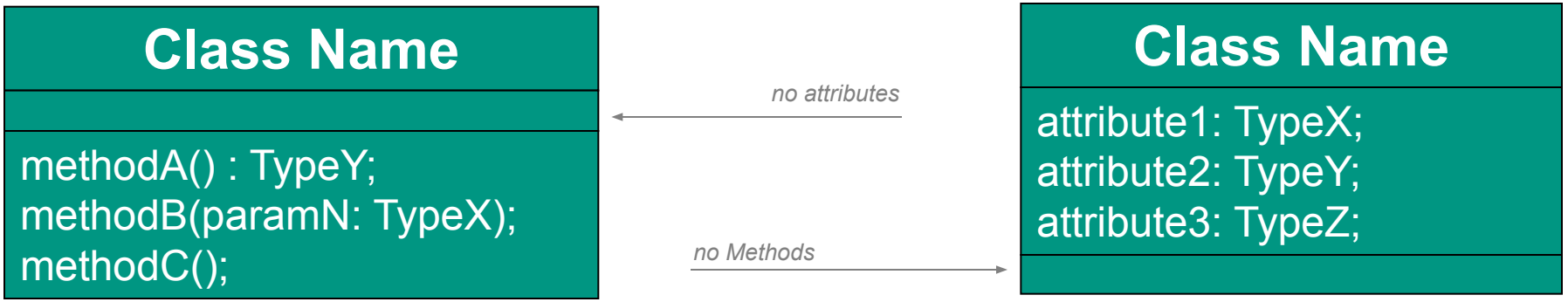
James Rumbaugh

UML diagrams



Classes in UML



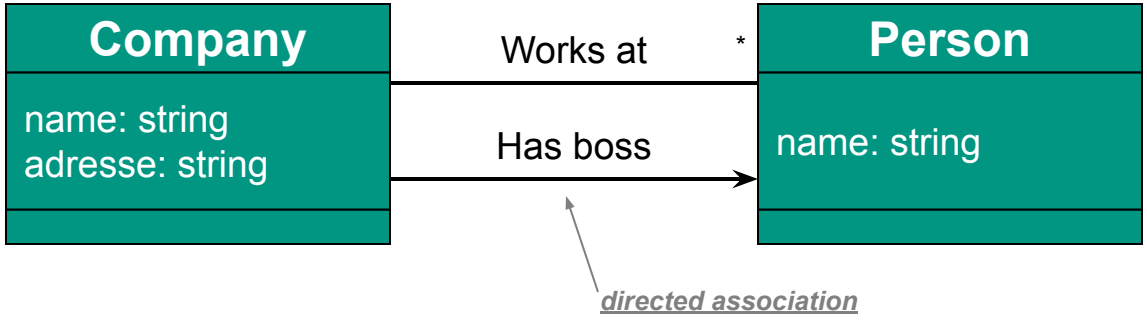


Association

- Association is specified between classes and describes possible relationships between instances
- Association defines properties of n-ary relations between sets:
  - The sets are given as classes
- **Multiplicity** allows to specify **cardinality,** i.e. number of elements or some collection of elements
  - Multiplicities indicate in how many tuples of the relation, elements of a given class could appear (for example 1: 1, 1: n, m: n).

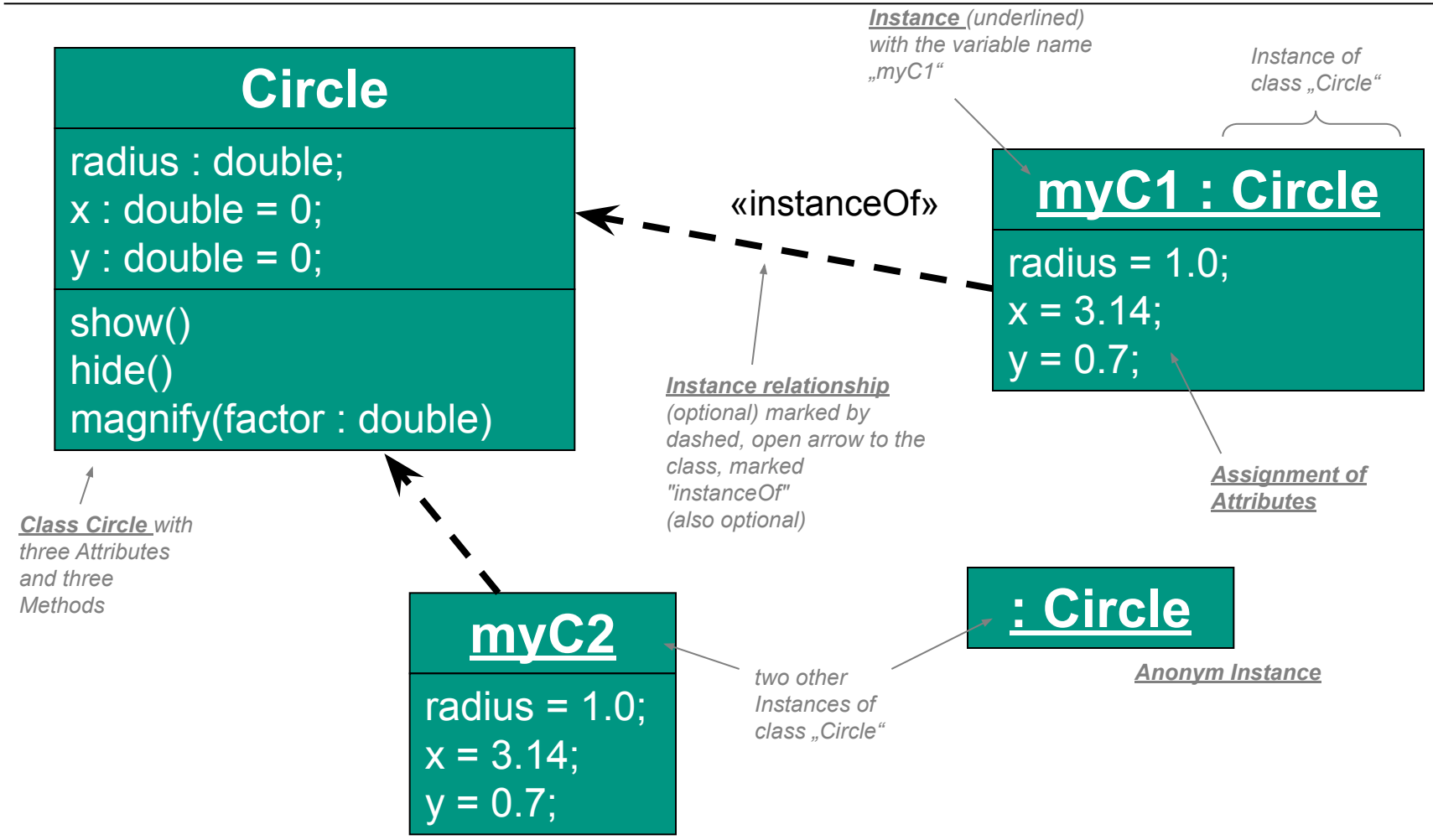
Class diagram – Example

- Describes the types of objects in the system
- Describes the static relationships among them



- Note: Class diagrams are multigraphs, i.e. several edges can consist between identical nodes.

Object / Instance diagram

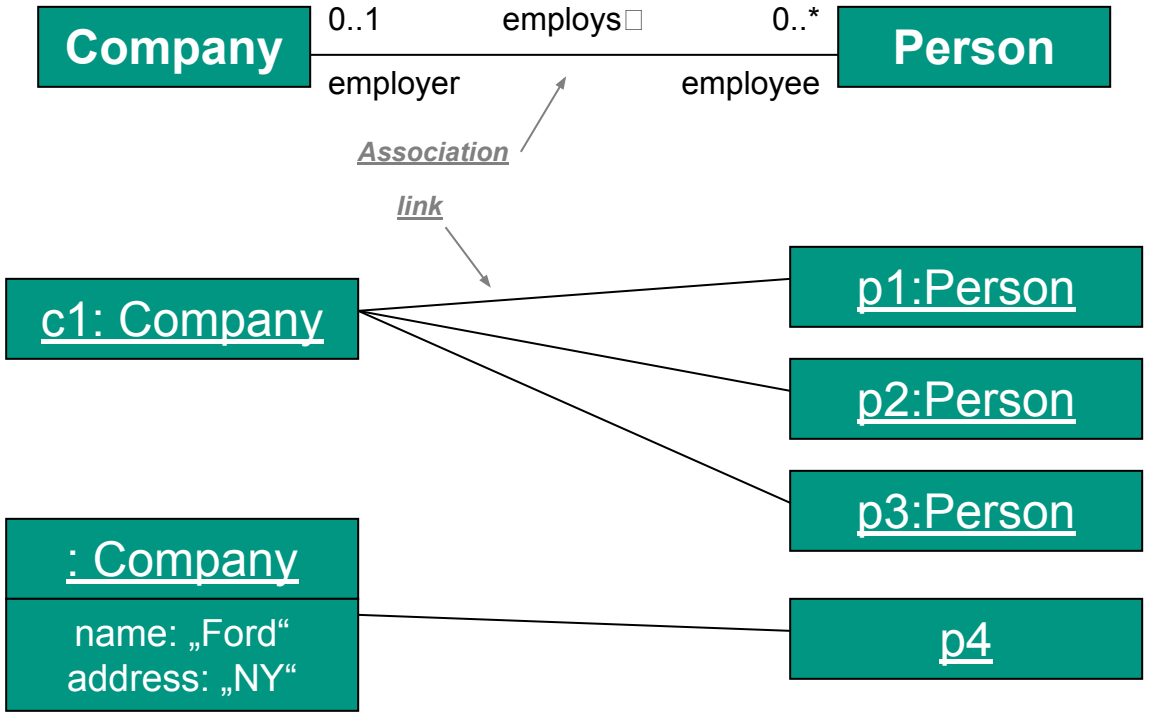


Components of class diagrams

- Class name
- Class properties
  - Attributes
  - Associations (could be bi-directional)
- Class operations
  - Visibility name (parameter list): return-type {property-string}
- Generalization
  - Inheritance (subclass, super class, interface, …)
- Dependency
- Constraints {}

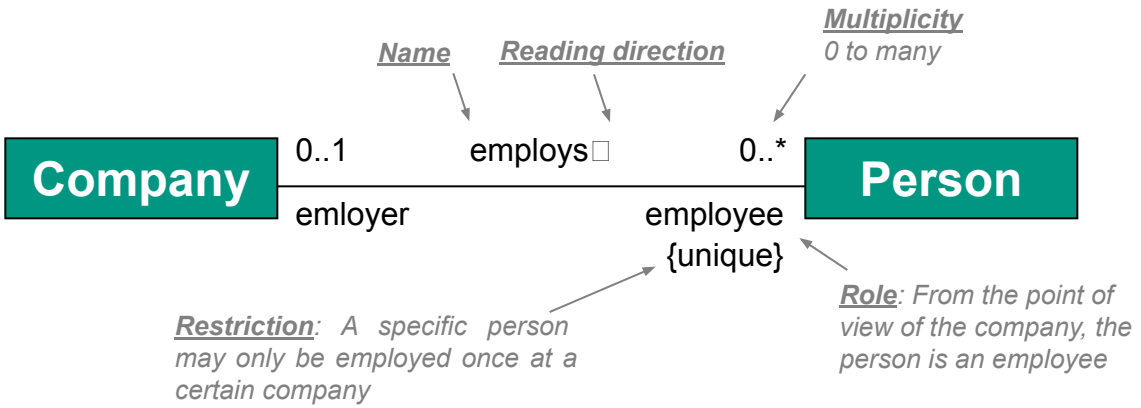
Association vs. links

- **Association** is specified between classes and describes **possible** relationships between instances (objects)
- **Link** is specified between instances, expressing an **actual** relationship between objects



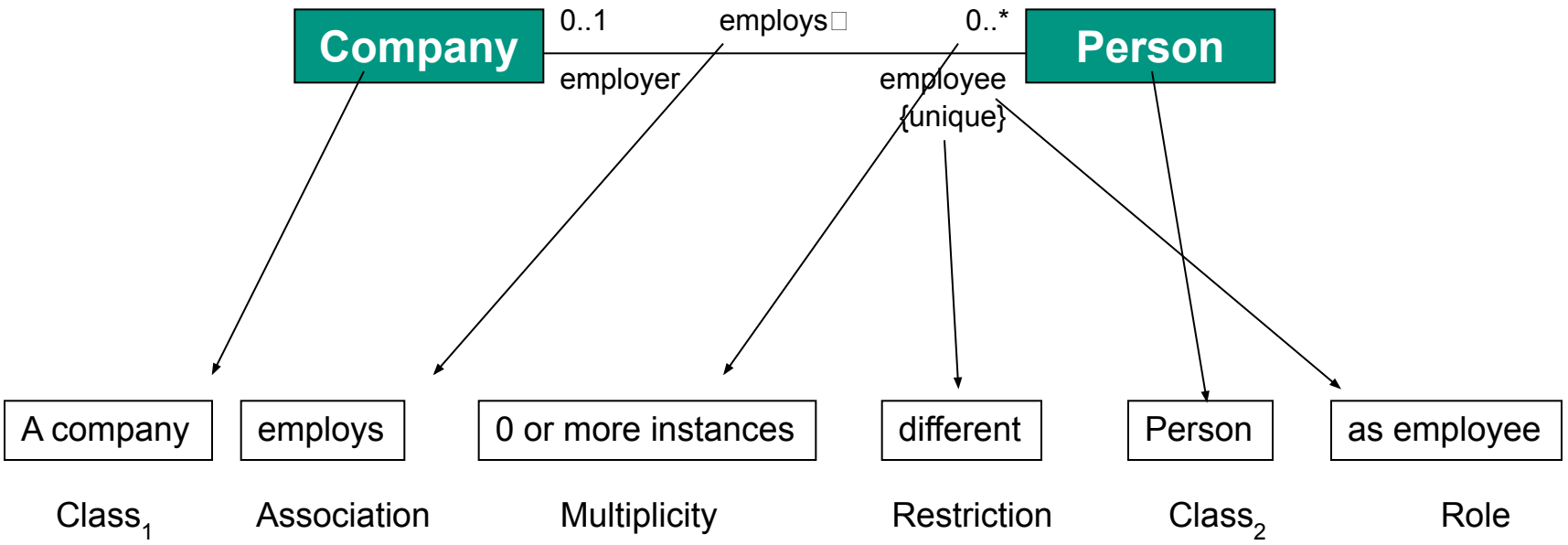
Default attributes of associations and association ends

- The relation characterized by an association can be described in more detail:

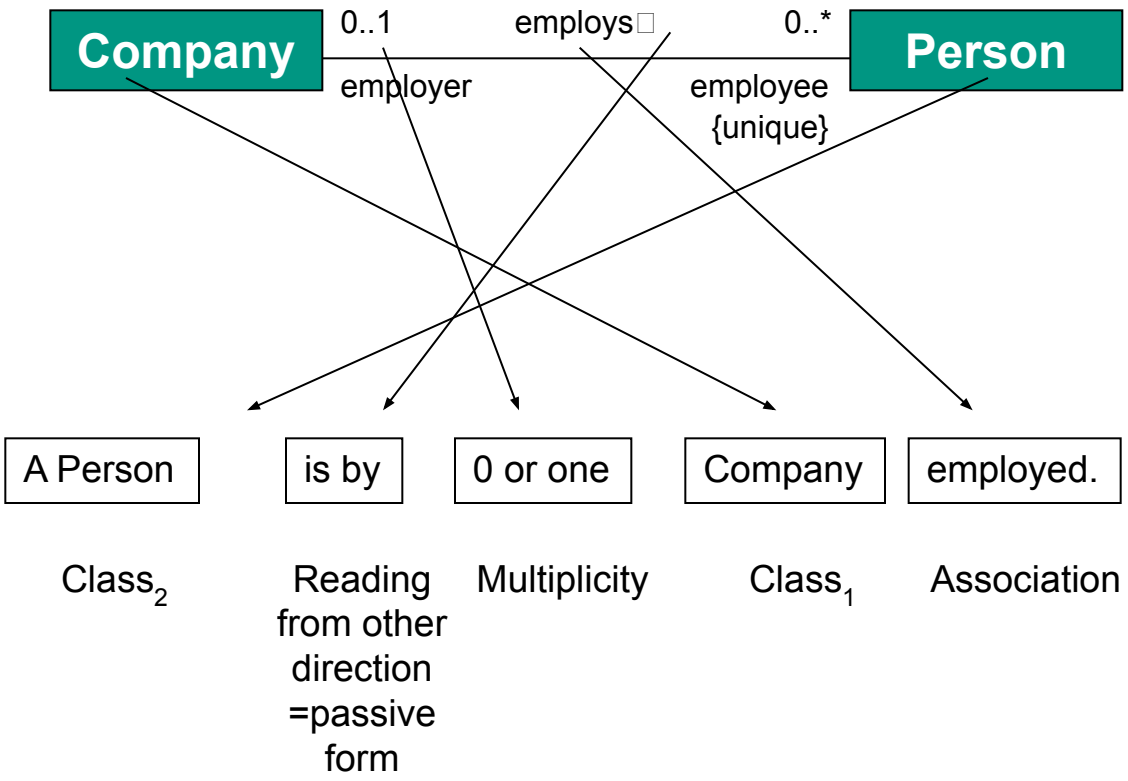


- Name, reading direction and role are "only" etiquette, which serve the interpretation. But they have no more precisely defined semantics.

Association – Example (1)

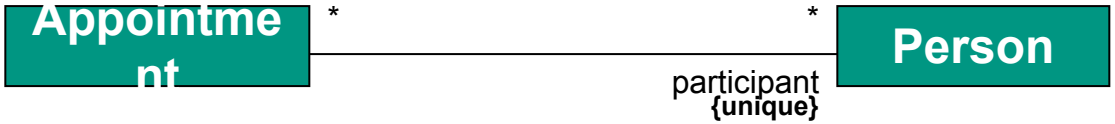


Association – Example (2)

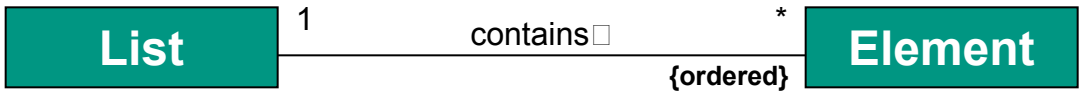


Restriction (Constraints) – Example

- A person can participate in any number of appointments, but **only once** at an appointment:

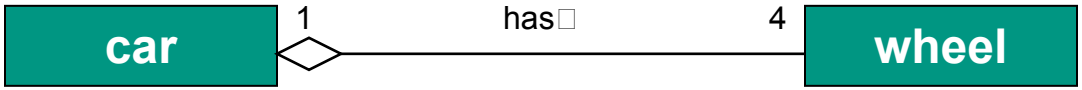


The elements of a list have a specific **order**:

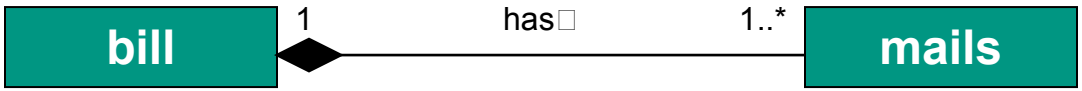


Special forms of associations

- There are special forms of associations. So those with special interpretation rules:
  - **Aggregation** (special form of association): Part-Whole-Relationship

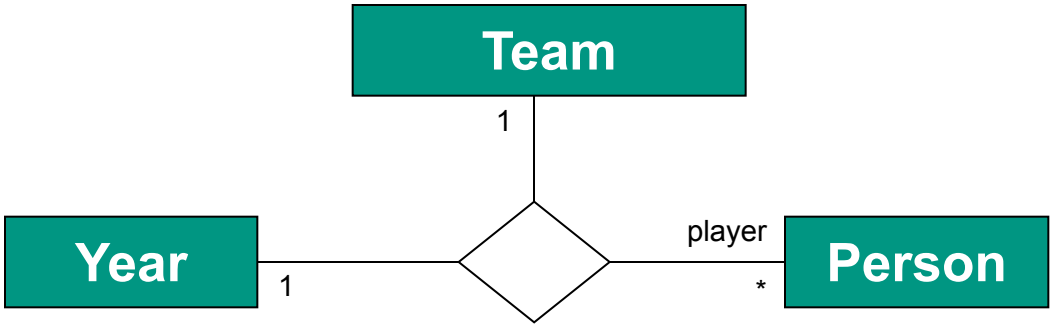


  - **Composition** (special form of aggregation): **strict**, parts have no right to exist without the whole (semantics important, e.g. delete operations)
    - Single owner, disappear with the owner



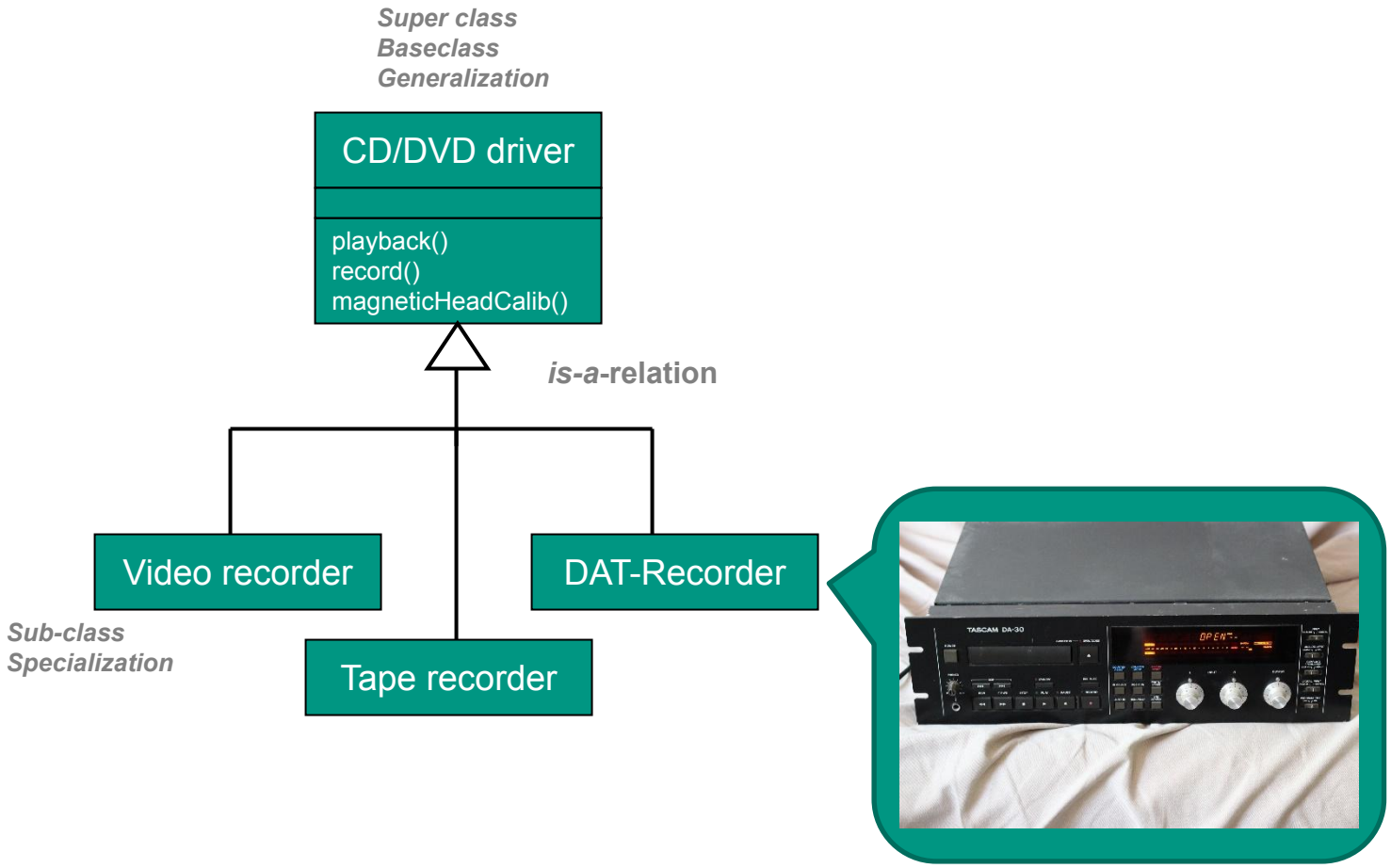
N-ary Association – Example

- **N-ary association** with more than two ends
- How is this to be interpreted?



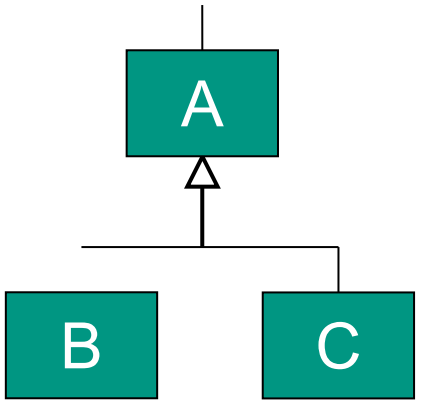
- This means that the persons X, Y and Z play in exactly one team in year DDDD (to be more precise: each person always has to play in exactly one team)

Inheritance



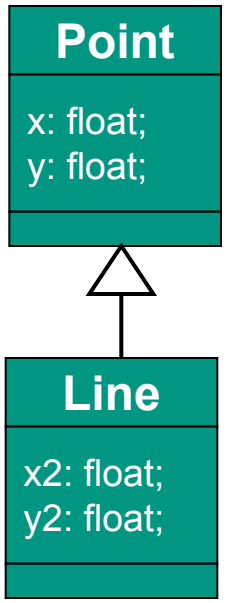
Inheritance

- Let A and B be classes, and ΩA and ΩB the set of objects that make up classes A and B.
  - Then B is a subclass / specialization of A (or A is a superclass / generalization of B) if: $\Omega B \subseteq \Omega A$.
- It is also said that B inherits from A.
- Since each instance of B is also an instance of A, the relationship between A and B is called the **"is-a" relationship**.
- If A has several subclasses, these subclasses should usually be disjoint.



Inheritance: is-a relationship

- To model a line, you could use the class point for that.
- Is inheritance a good idea here?



A line is not a special case of a point.
This modeling is bad!

Inheritance

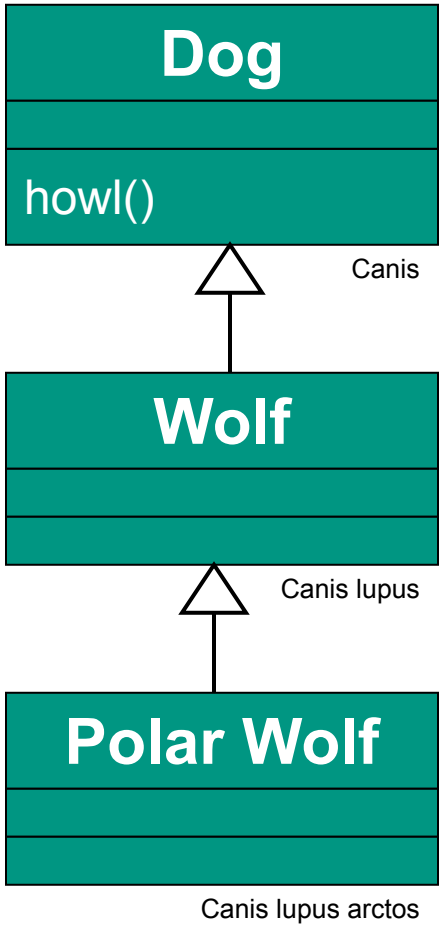
- In practice , the concept of "inheritance" offers advantages:
  - Can be used to identify common subsets of attributes, states, associations, and methods for different classes
    - These can then be summarized in a common superclass
  - Avoidance of design and implementation redundancy
  - Theoretically well-founded typing concept
- A mechanism for **code reuse** to allow independent extensions of the original software via public classes and interfaces.

Inheritance – Substitution principle

- **Substitution principle** (Liskov): In a program, where S is a subclass of C, each instance of class C can be replaced by an instance of S, and the program will continue to function correctly.
- All properties (attributes, associations, states, methods, etc.) of the superclass must exist in the subclass (as defined in the superclass).
- The subclass may still define additional properties that make it more special.
- The subclass can not omit superclass properties.
  - If necessary, this is not an inheritance relationship.
  - Substitution of the superclass by the subclass would not be possible.

Inheritance – Example

- The inheritance relationship is **transitive**.
  - A dog can howl().
  - A wolf is a dog.
  - The wolf inherits the method howling () from the dog!
    - The wolf can also howl().
  - A polar wolf is a wolf.
  - The polar wolf inherits all methods from the wolf incl. howl()
    - The polar wolf can also howl().



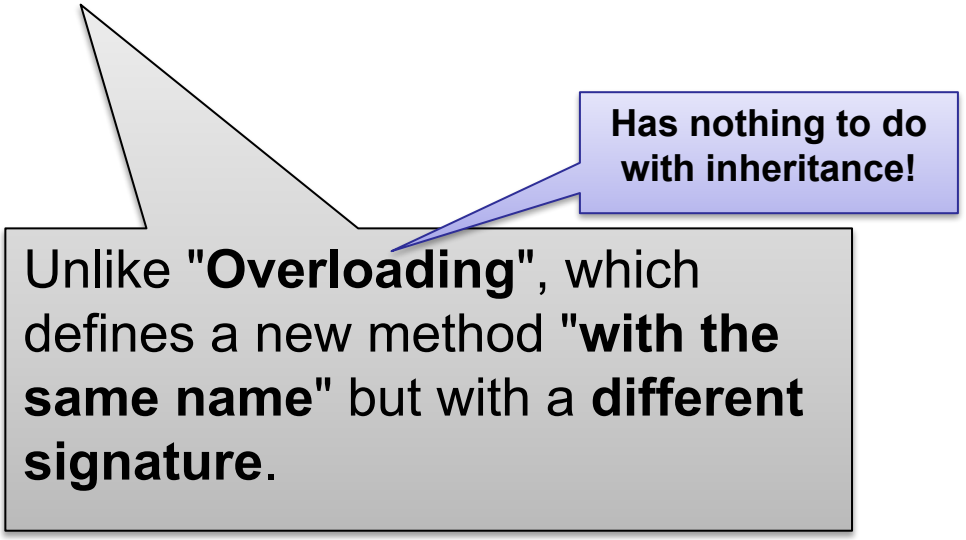
 You can send a message howl() to the polar wolf instance

Signature inheritance vs. Implementation inheritance

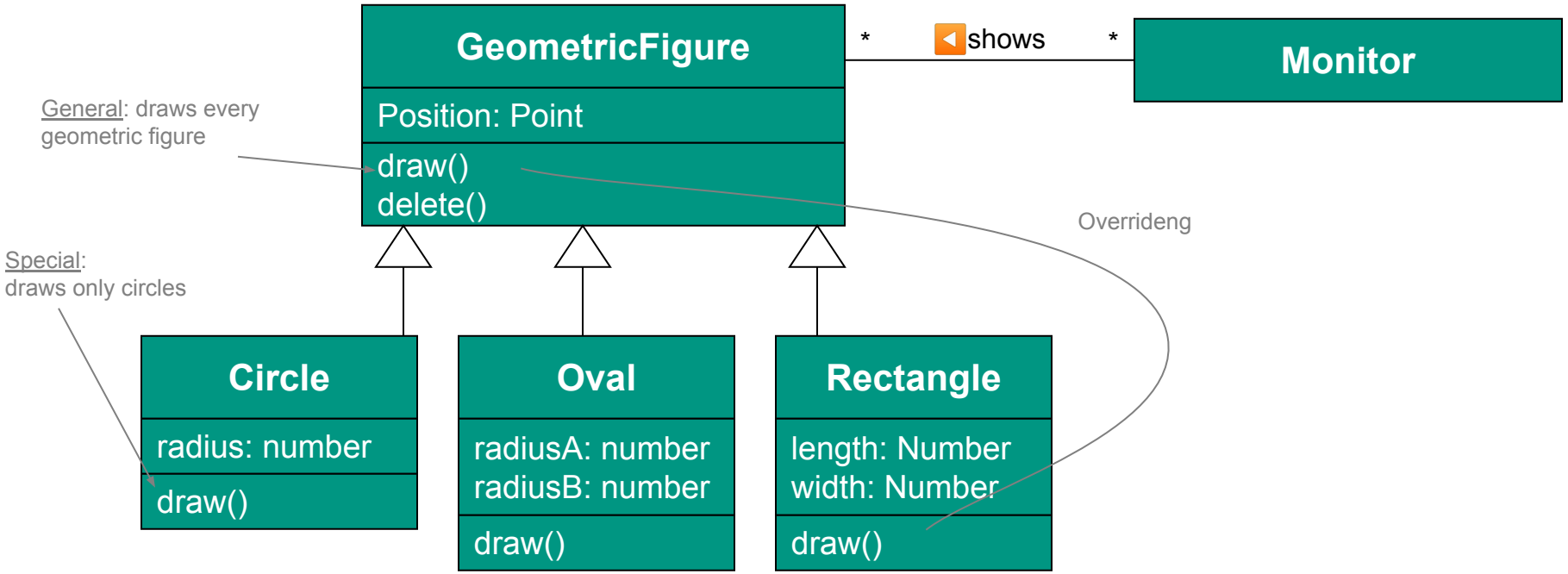
- **Signature inheritance**: A method defined and (possibly) implemented in the superclass **only** transfers its **signature** to the subclass.
- **Implementation inheritance**: A method defined and implemented in the superclass transfers its **signature** and its **implementation** to the subclass.
- Implementation inheritance does not work without signature inheritance, but the reverse is true.
- Signature inheritance is sufficient for 'everything' in the OOA / OOD / OOP, implementation inheritance is not necessary (but convenient).
  - For example, Java and C# offer both implementation inheritance and signature inheritance.

Method overriding

- Properties of the base class (superclass) can be adapted to the needs of the specialization
  - For example, methods can be changed.
- **Overriding**: A new implementation of an inherited method while **keeping the signature**

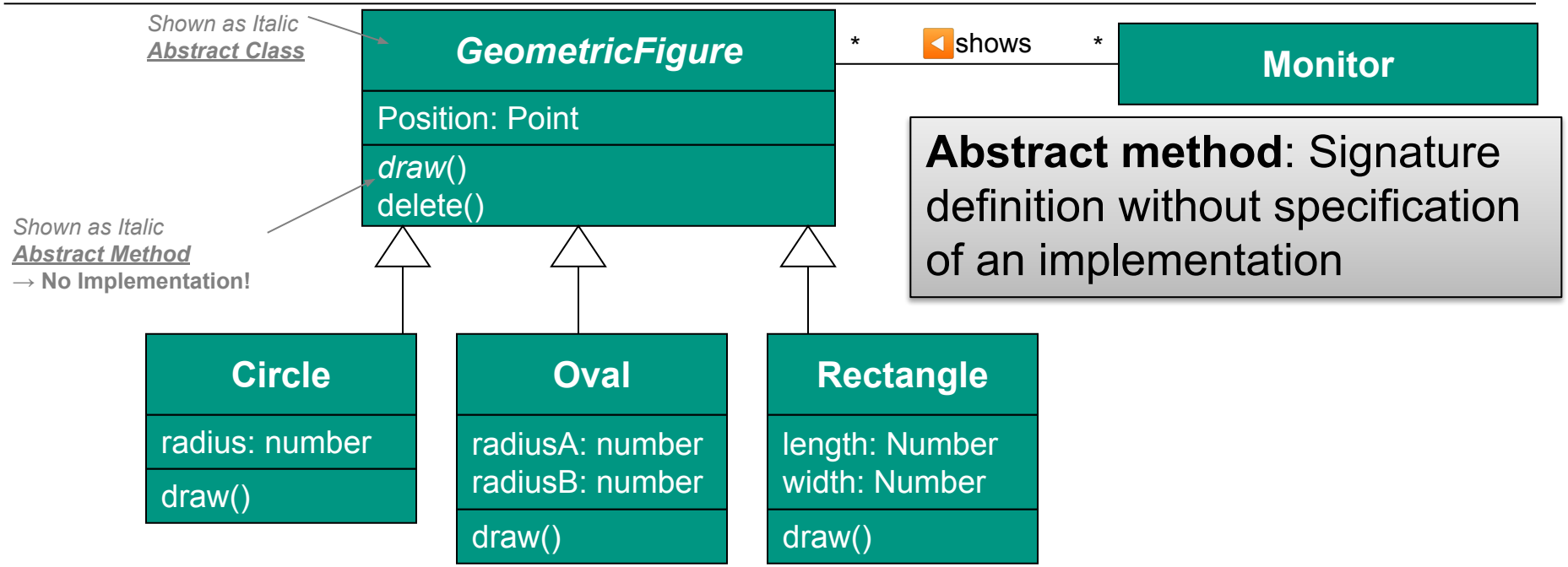


Overriding – Example



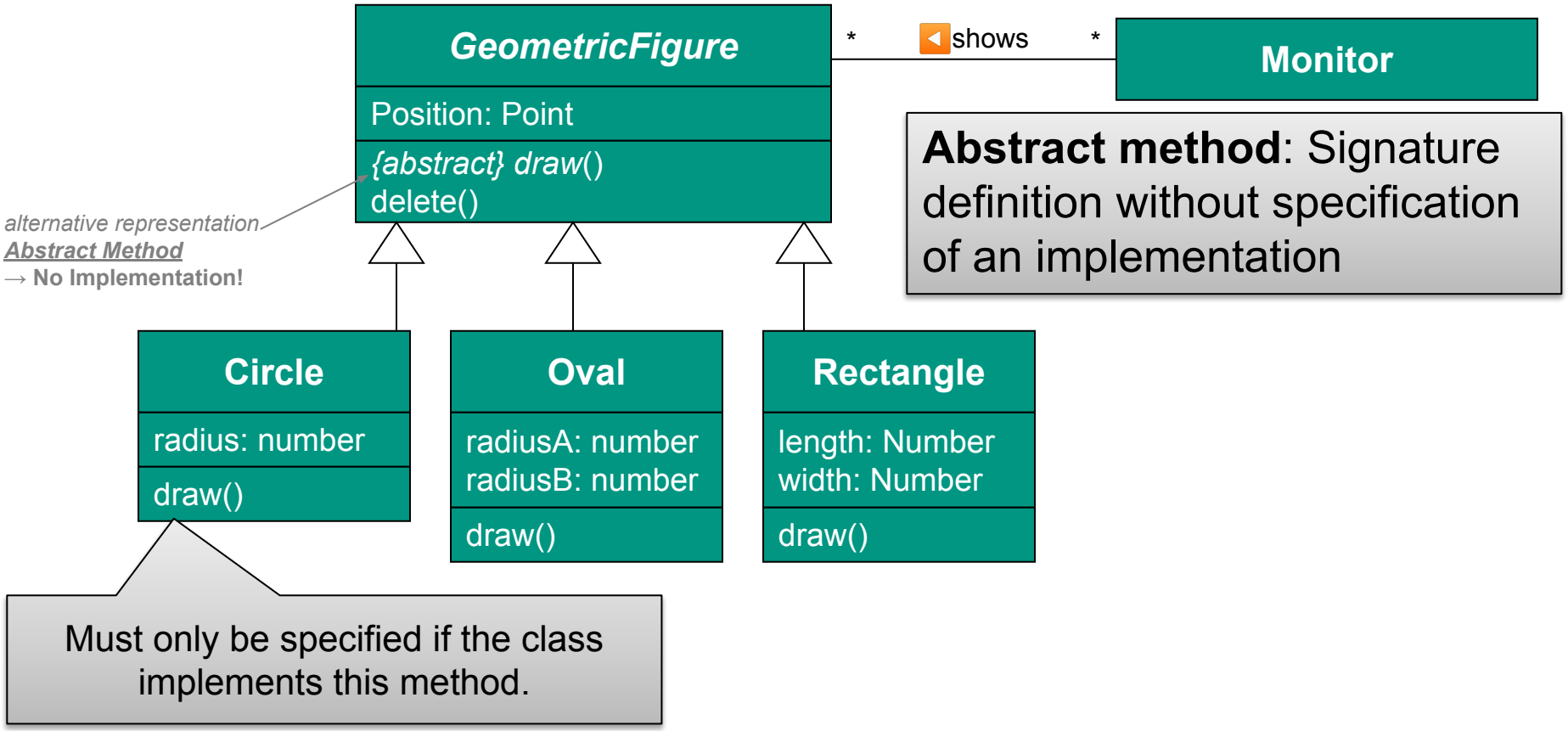
- Insight:
  - Each of the three specializations must implement their own drawing method
  - Can there ever be a meaningful implementation for drawing in GeometricFigure?

Solution: Abstract methods



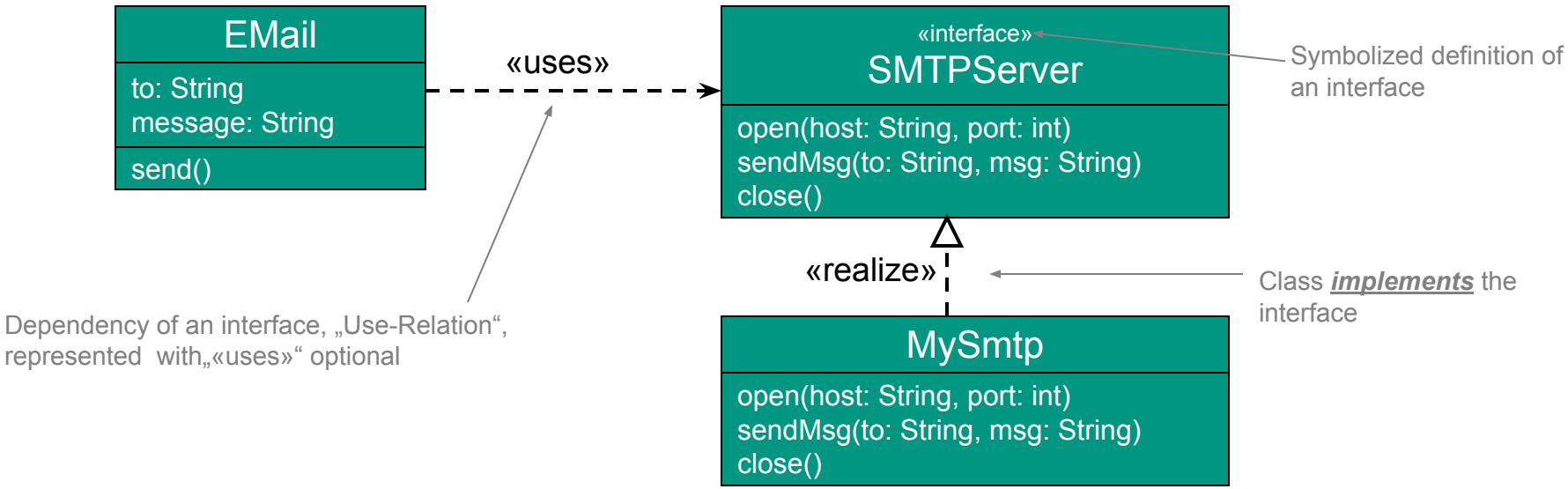
- If there is an abstract method then the class containing it is also abstract.
  - There are no instances of abstract classes themselves. Why?
  - Abstract classes ”pass on the obligation" to implement a method. Why?

Abstract methods (alternative representation)



Interfaces

- **Interface**: Defining a set of **abstract methods** that must be offered by the classes that implement them.
  - Interface can not be instantiated directly
  - Using class may instantiate arbitrary implementing classes.



Use of interfaces

- Interfaces transfer the **obligation** to implement certain methods.
  - However, interfaces also **guarantee** that certain methods exist.
  - Instance of a class that implements a particular interface can be used as if they were an ”instance of the interface" (is-a-semantics)
- Idea: with an interface you indicate **how** an object is to be used, **not what** it represents.

Interface – Example

- You get a "black" box.?
- You see,
  - play ()
  - stop()
  - ff()
  - RWD()
  - Skip ()
- You can send the messages to the device.
- Although you do not know what kind of device it is, you can use it because you know the interface.

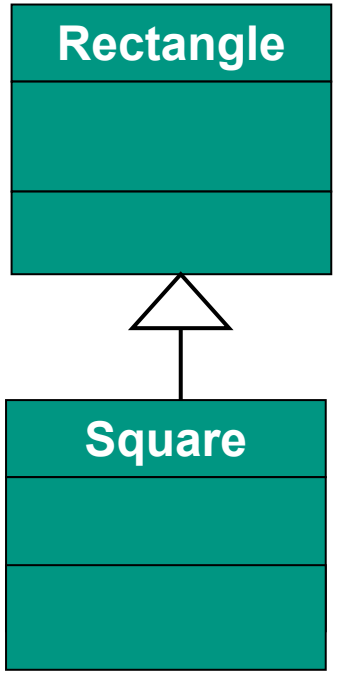
"Inheritance" with interfaces

- With interfaces there is no inheritance!
  - If an interface B extends an interface A, then the set of abstract methods of A is a subset of the set of abstract methods of B ($A \subseteq B$).
  - If a class X implements an interface A, then the set of abstract methods of A is a subset of the method definitions of X, where X may additionally specify one implementation at a time.
- Since it is only a matter of set-inclusion, **multiple inheritance** of interfaces is not a problem.
- What problems can occur with multiple inheritance of classes?

Is this modeling ok? (1)

Example

- Is this modeling ok?
- Obviously, the set of squares is a subset of the rectangles.



Is this modeling ok? (2)

- Structurally, the modeling is correct because squares are actually a subset of the set of rectangles.
- But in behavior the two classes might not be consistent.
- Only when superclass objects are completely substitutable by subclass objects the subclass objects can be used without harm in the context of the superclass.

Polymorphism

- Polymorphism means **diversity**
- To process objects differently depending on their data type or class
- More specifically, it is the ability to redefine methods for **derived classes**
- The provision of a **single interface** to entities of different types

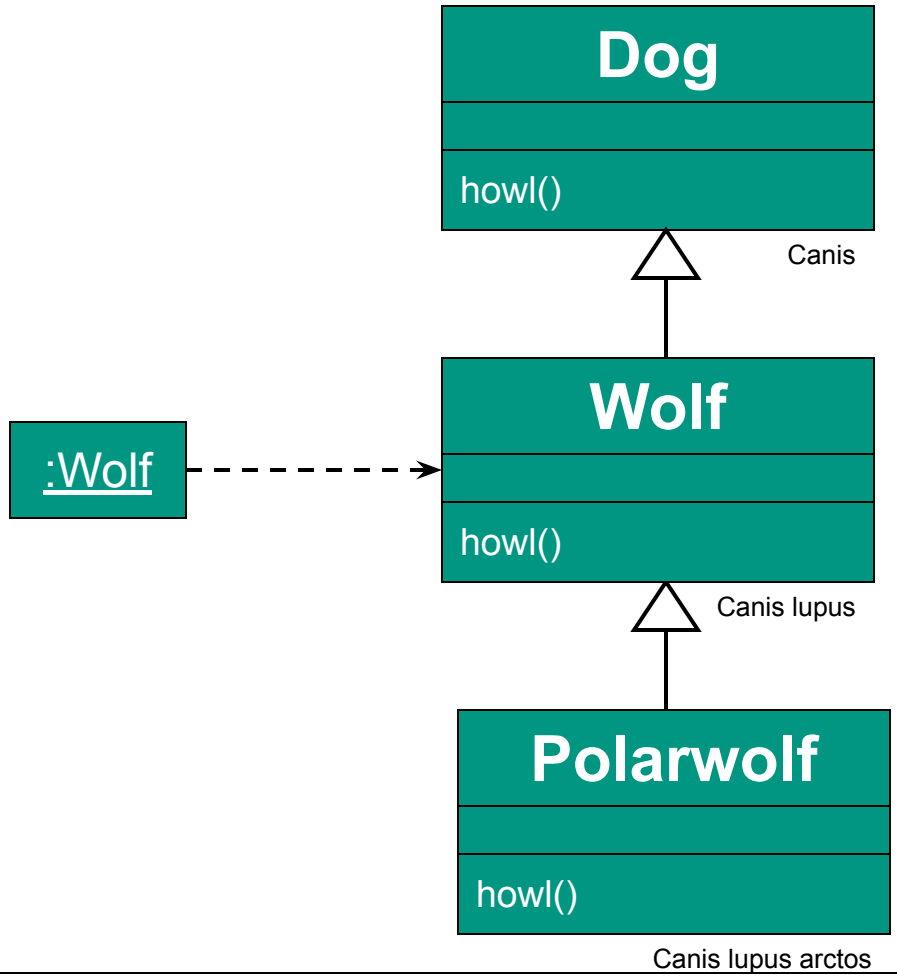
Polymorphism

- "**Static**" (**overloaded**)
  - There can be several methods with the same name (but the signature must be different so that the compiler knows which one to use)
- "**Dynamic**" ("**use of inheritance**" – overriding)
  - It calls the method with the specified signature, which is the most specific in the inheritance hierarchy (as viewed from the class of the current instance)

Has nothing to do with OO or inheritance!

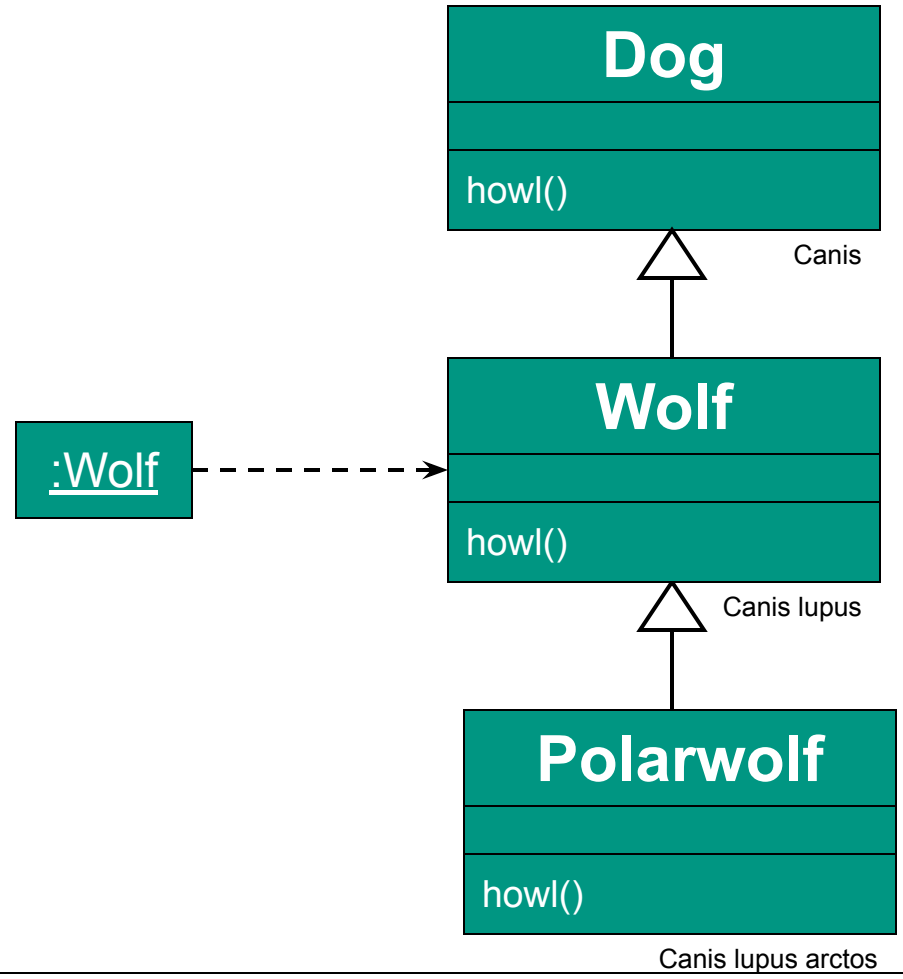
Dynamic polymorphism – Example

```
Dog d;
Wolf w = new Wolf();

d = w;
d.howl();
```

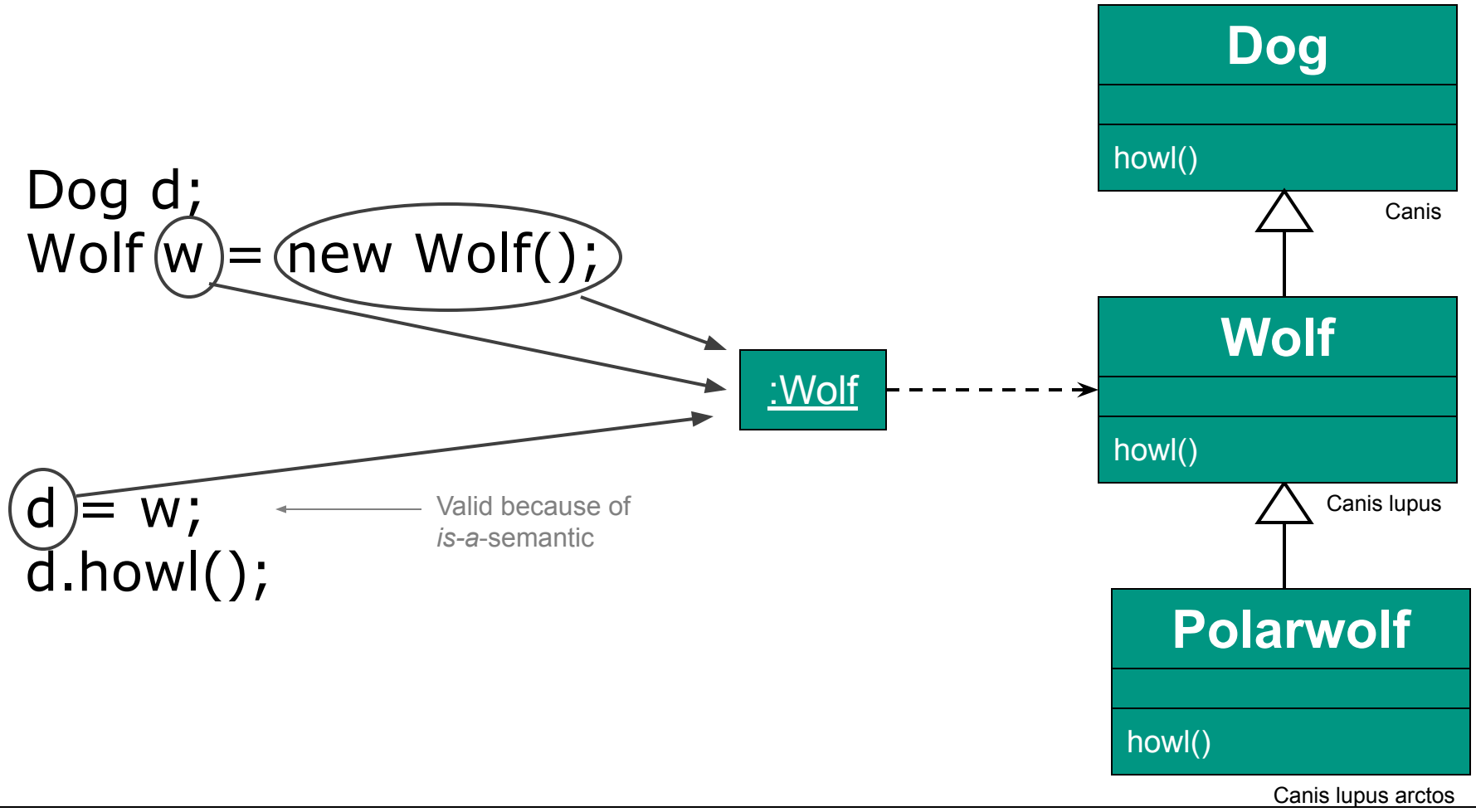


Dynamic polymorphism – Example

```
Dog d;
Wolf w = new Wolf();

d = w;
d.howl();
```

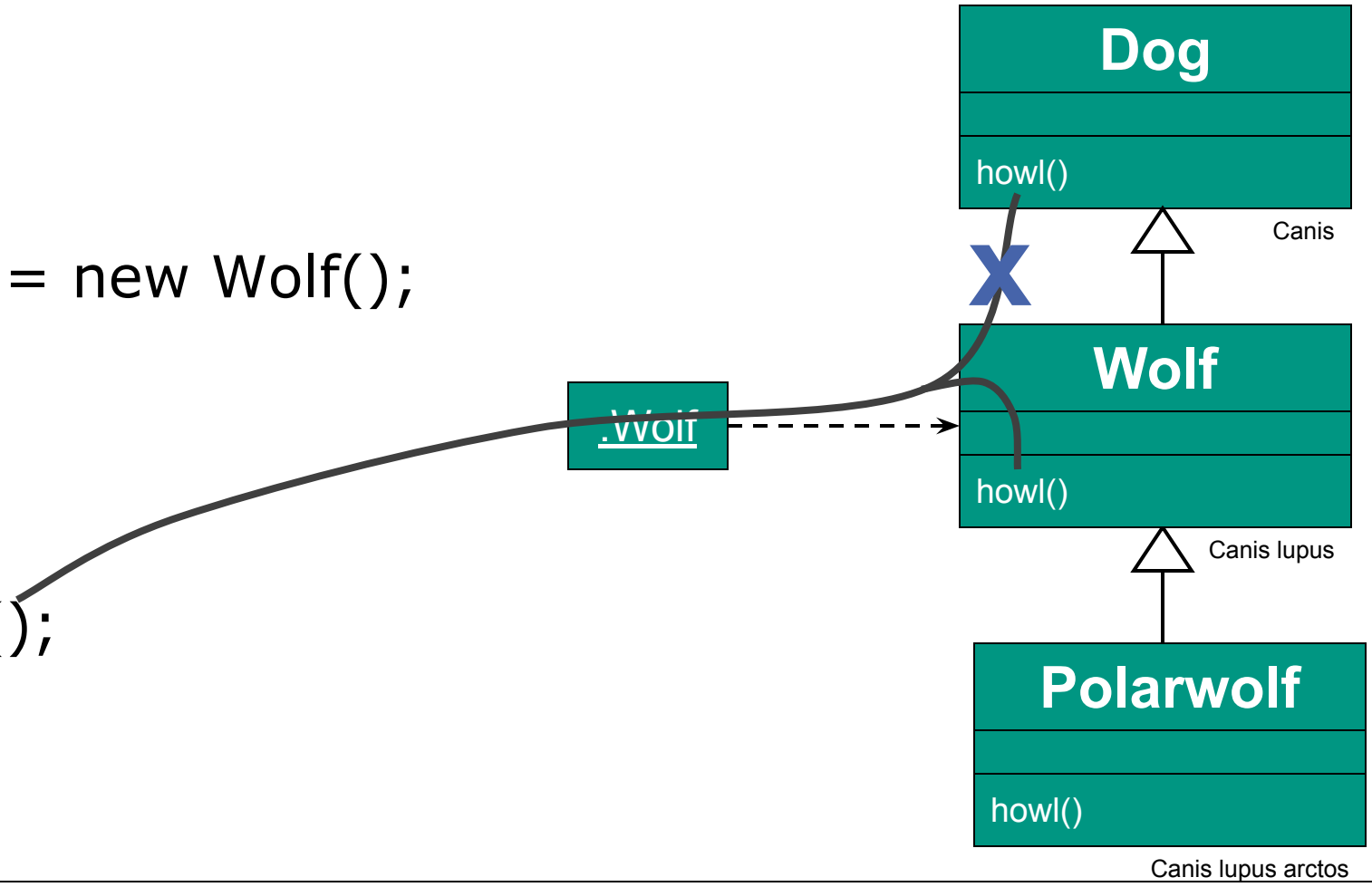


Dynamic polymorphism – Example



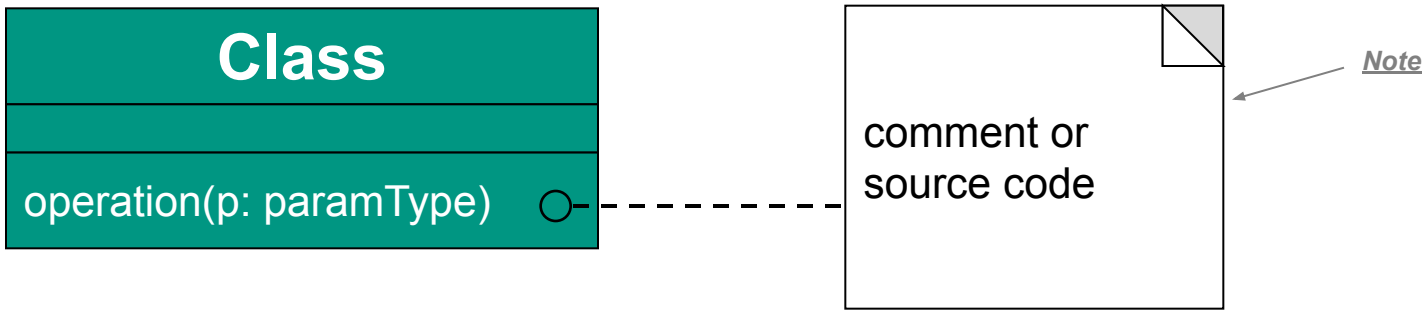
Dynamic polymorphism – Example

```
Dog d;
Wolf w = new Wolf();


d = w;
d.howl();
```

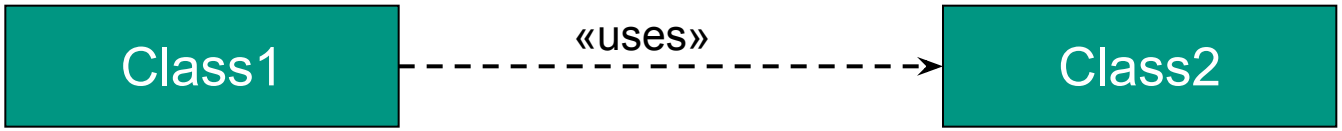


What's left in the class diagram ...

- Notes / Comments



- Dependency (Class1 depends on Class2, for example because it uses Class2 as parameter, local variable, or return value)



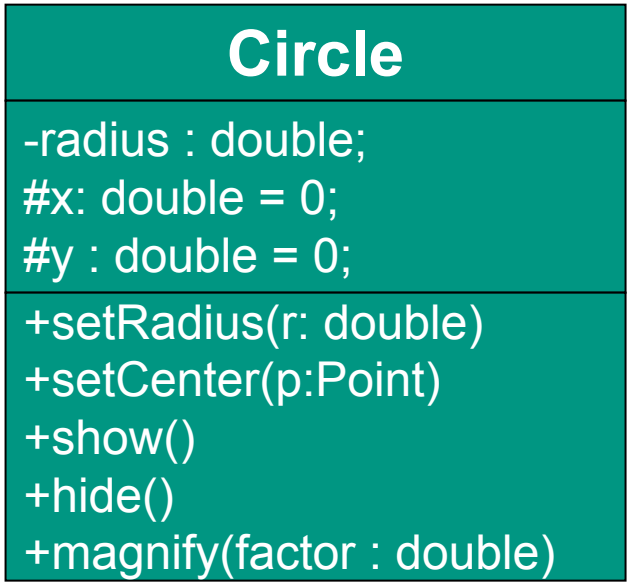
Visibility – Access protection

- The attributes and methods of an object can be protected from access by other objects. Access allowed with:
  - **private** ("-"): only (but all!) instances of the same class
  - **protected** ("#"): instances of the same class and all derived classes, as well as instances from the same package (subsystem / library)
  - **public** ("+"): each instance
- The corresponding symbol is simply **prefixed** to the attribute/method name.
- **Note**: "private" does not mean "just the copy itself"!

Visibility – Access protection

- The ”access protection" is limited to the time of compilation
  - All access rights can be changed via code manipulation (for example with BCEL for Java), so do not use against deliberate attacks.
    - BCEL: Byte Code Engineering Library
  - They are "only" intended for a clean design, which wants to avoid excessive distribution of information, and thus is friendly to change.

Visibility – Example

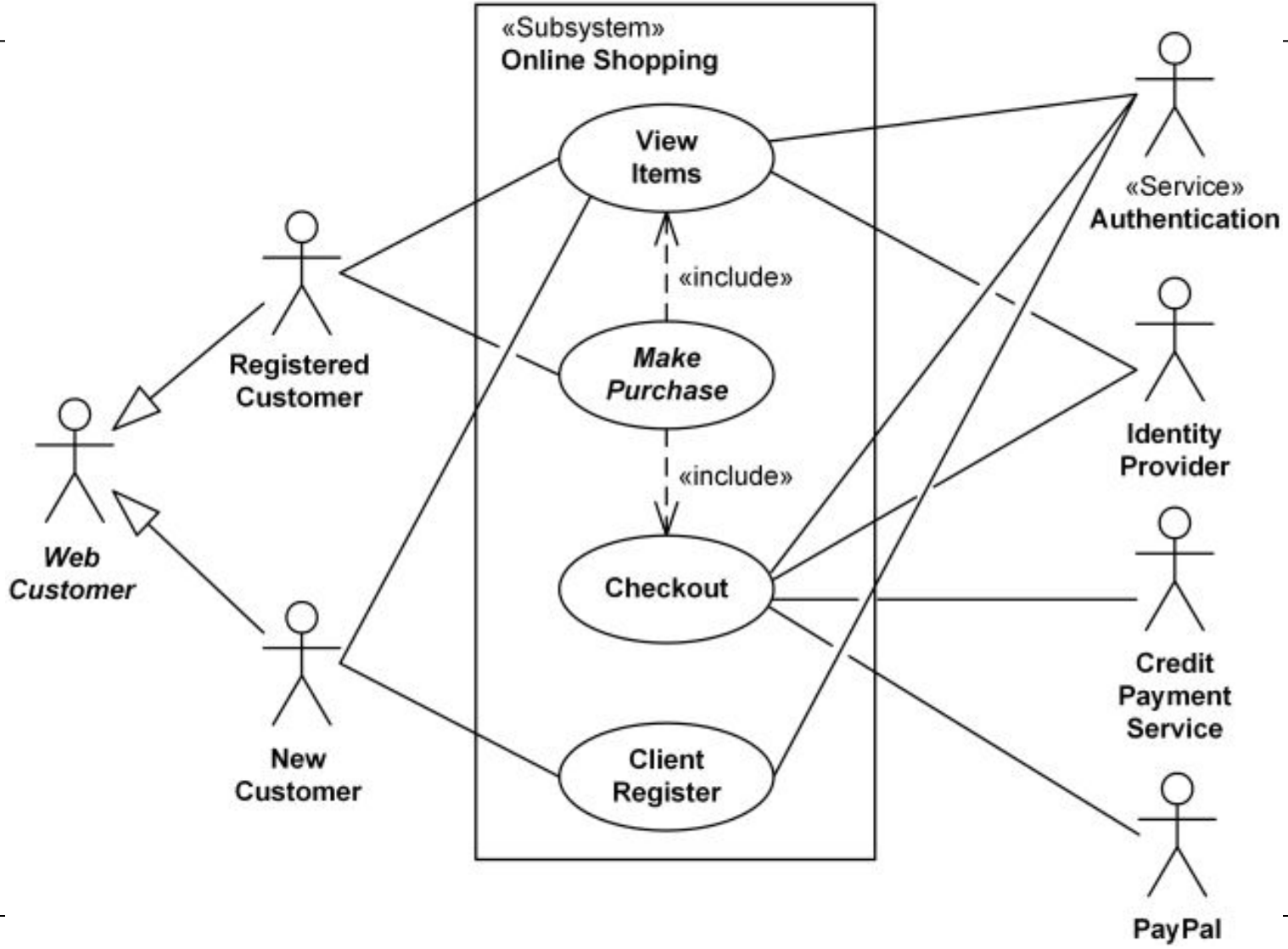


- No indication of visibility would mean: public

Example: Online shopping model

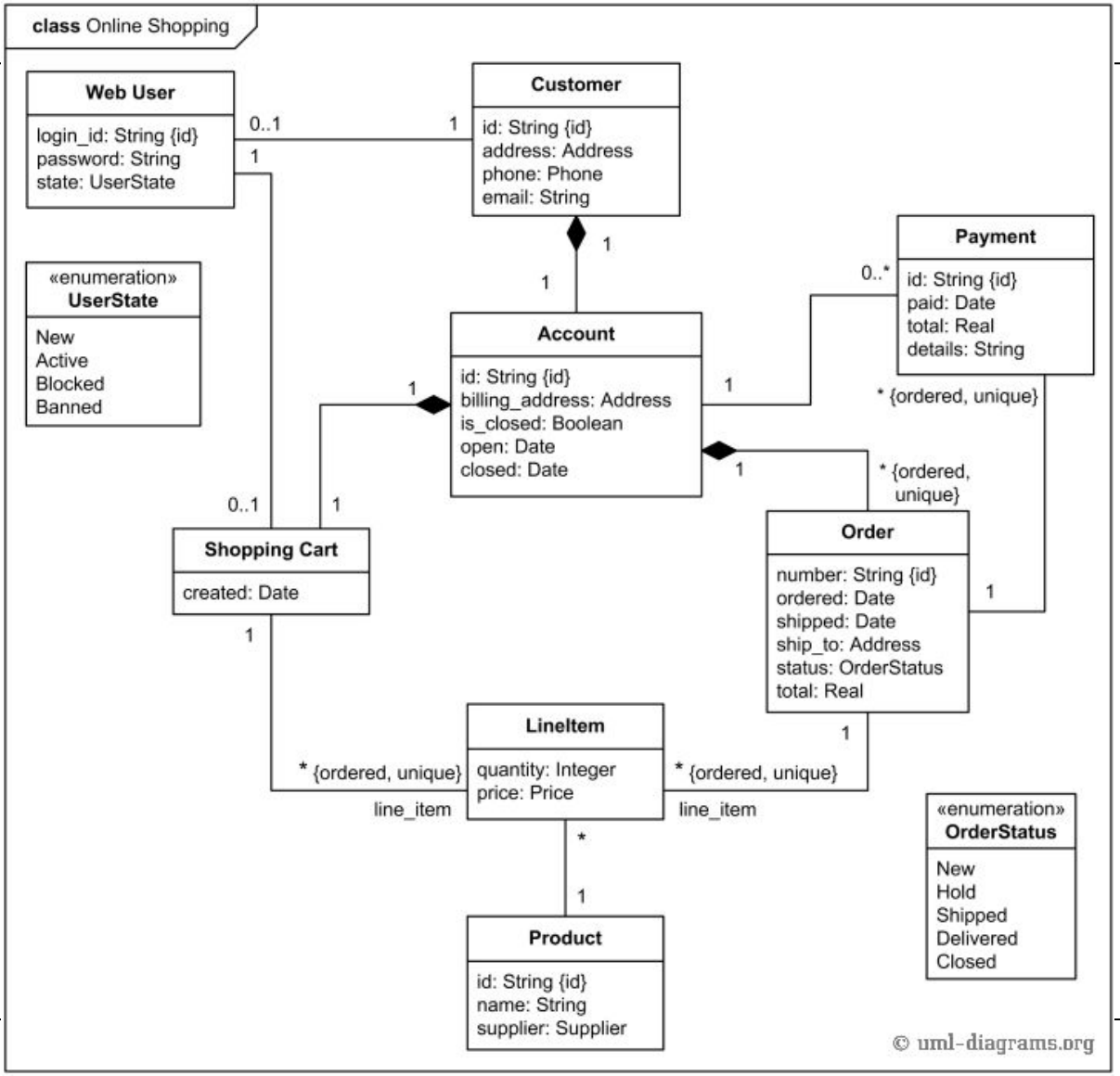
- What are the actors and use cases?
- What are the scenarios and steps for “check-out” use case?
- What is the activity diagram?
- What is the **class diagram**?
  - See: https://www.uml-diagrams.org/class-diagrams-examples.html

Example: Online shopping model – Use case diagram



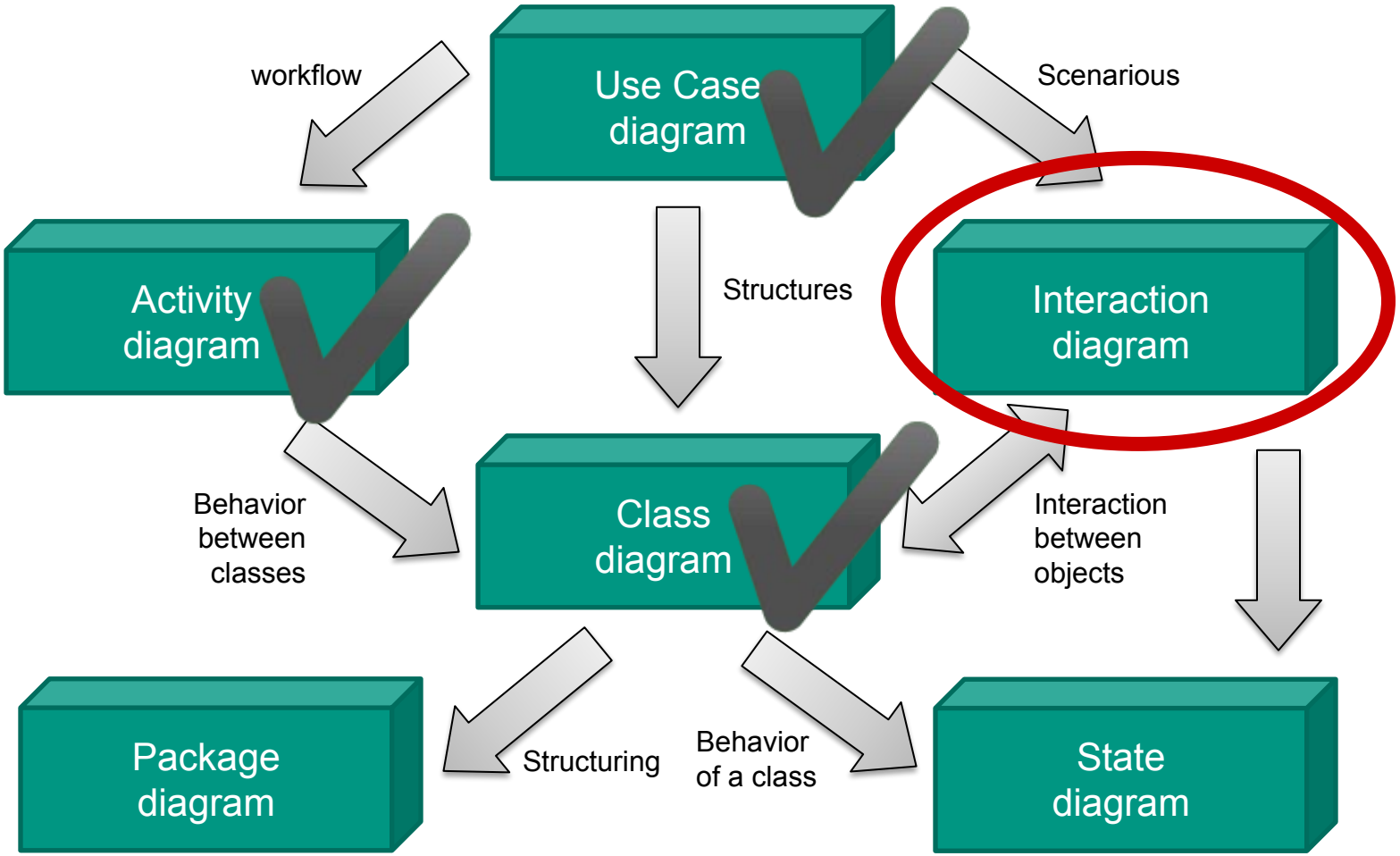
Source: https://www.uml-diagrams.org/class-diagrams-examples.html

Example: Online shopping model – Class diagram



Source: https://www.uml-diagrams.org/class-diagrams-examples.html

UML diagrams



Interaction diagrams

- Show the interactions necessary for a certain purpose between objects
- The class diagram is the basis of the interaction diagrams
- **Four types of interaction diagrams:**

1. Collaboration diagram / communication diagram (not covered)
   - Focus: **Structure** of the interaction partners
2. Time chart/diagram (not covered)
   - Focus: **Temporal** coordination
3. Interaction overview (not covered)
   - Activity diagram to **illustrate** complex sequence diagrams
4. **Sequence diagram**
   - Focus: **Messaging**

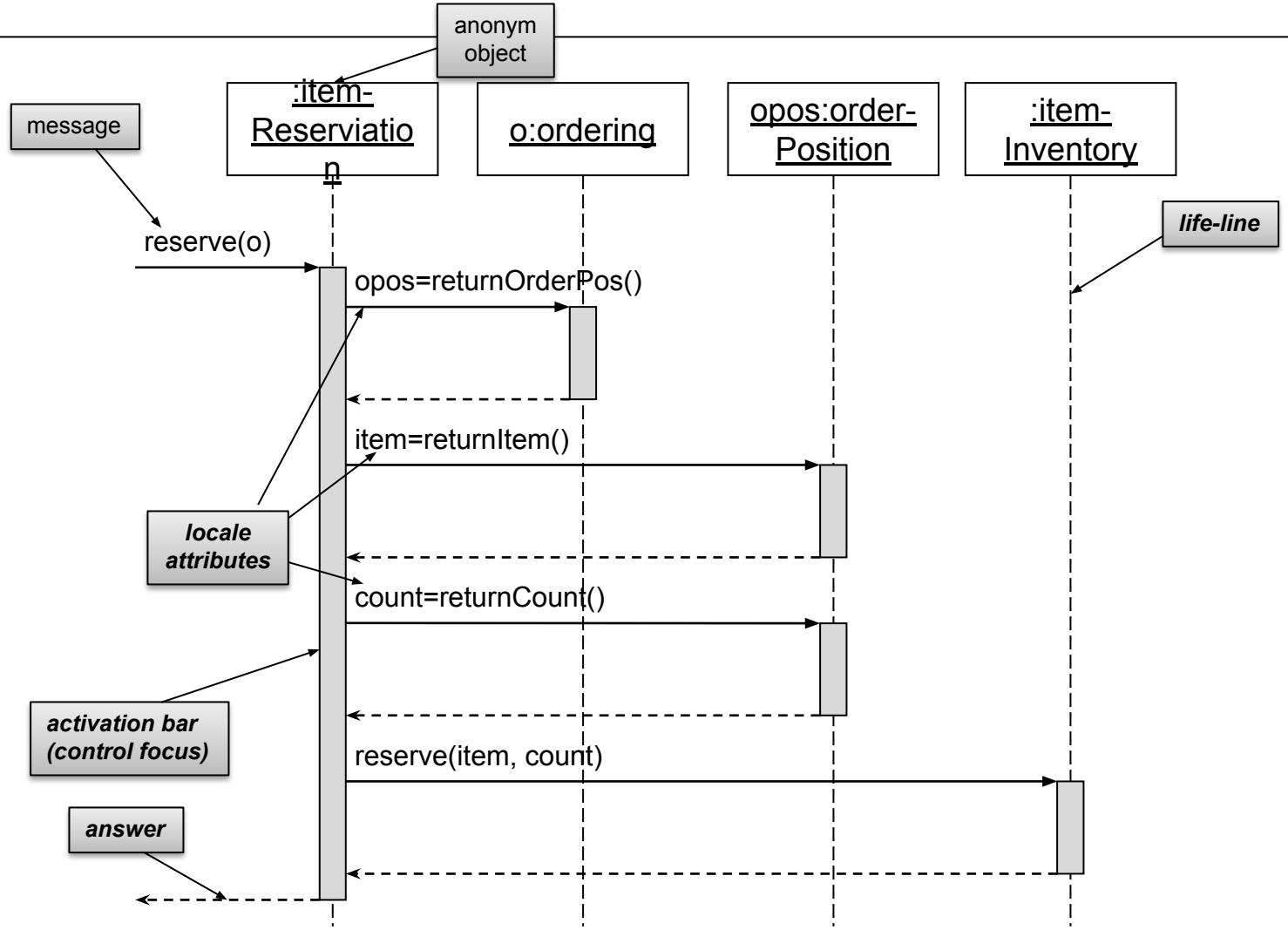
Sequence diagram

- Originally for the modeling of telecommunication services
- Exemplary representation of a possible procedure of a use case
  - Representation of variants are possible (”alt", "loop"), but this option should rarely be used
- **Focus** on the **temporal progress** (**chronological workflow**) of the messages
  - Time runs from top to bottom
  - Participants (roles) are represented by vertical dashed lines
  - Messages are drawn by horizontal arrows between participant-lines
  - Intuitive but needs a lot of space

Sequence diagram

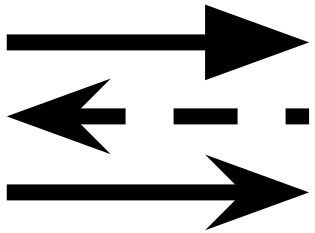
- Describes how objects collaborate/interact with each other in one scenario
- Components of sequence diagram
  - Participants
  - Life-line
  - Activation bar
  - Message
  - Regular calls, self calls
  - Creating and destroying object
  - Loops and conditional loops, alt, opt

Example


anonym object
message
:item-Reservation
o:ordering
opos:order-Position
:item-Inventory
life-line
reserve(o)
opos=returnOrderPos()
item=returnItem()
locale attributes
count=returnCount()
activation bar (control focus)
reserve(item, count)
answer

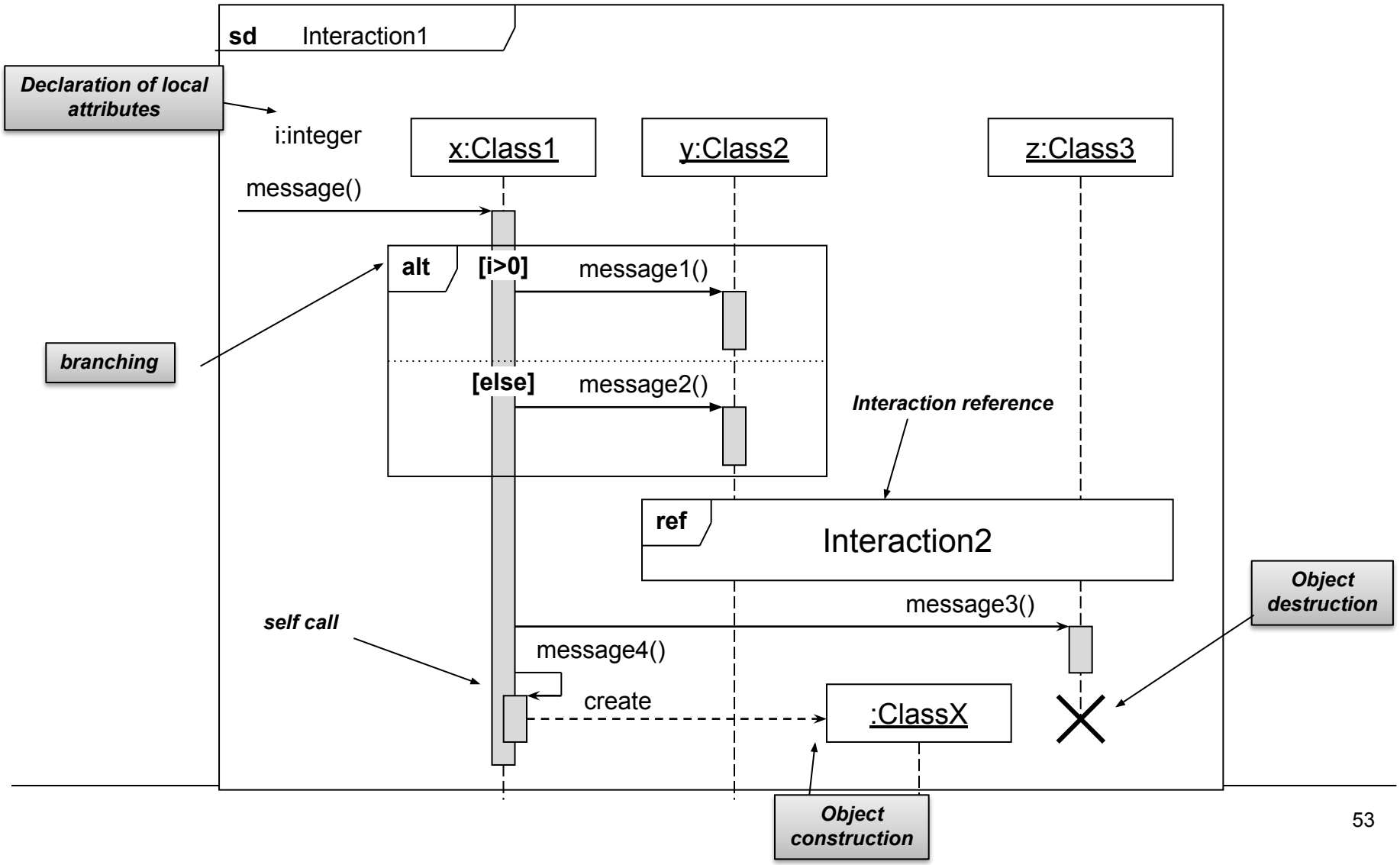
Components of sequence diagram

- Message types
  - Synchronous messages
  - Answers (optional)
  - Asynchronous messages



  - Activation bar (control focus)
    - Overlies the dashed lifelines by vertical bars
    - Indicates the role of program control
    - optional

Other components and notations



Operators

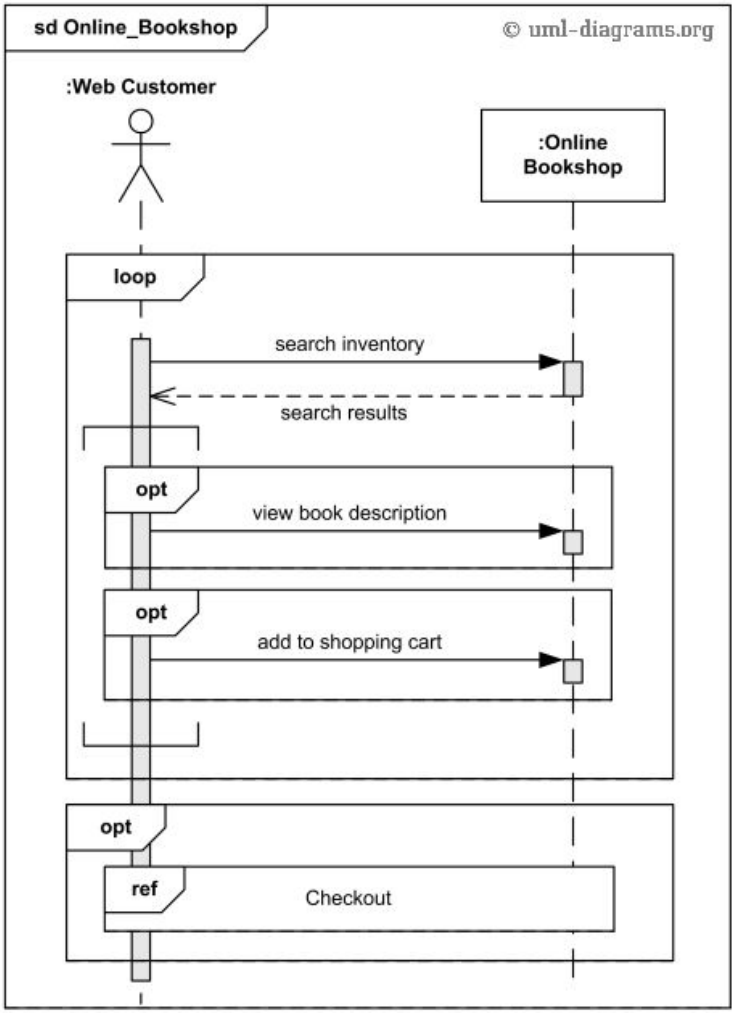
- Operators serve to express alternative processes and branches
  - Use with caution, as this will quickly become confusing
  - In a variety of ways an activity diagram is more suitable

We focus more on regular sequence diagrams for group projects!

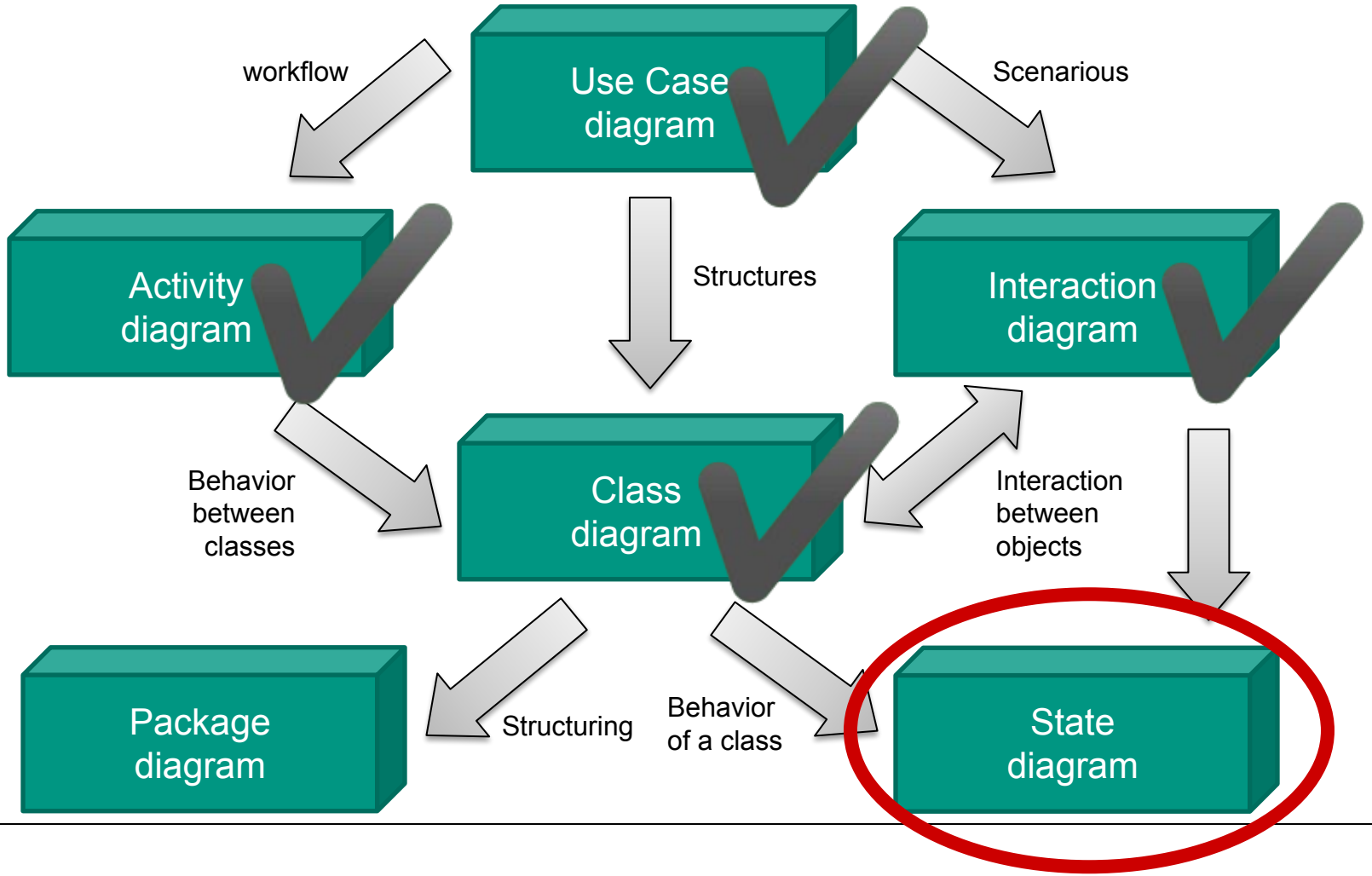
operator	cond./parameter	meaning
alt	[cond.1], [cond.1], … [else]	Only one of the alternatives is executed.
break	[condition]	If the condition is true, then only the block is executed and then the scenario ends.
opt	[condition]	Optional sequence. The subsequence is executed only if the condition is true.
par		Included subsequences are executed in parallel.

Sequence diagram – Example: Online bookshop

Source: https://www.uml-diagrams.org/class-diagrams-examples.html



UML diagrams

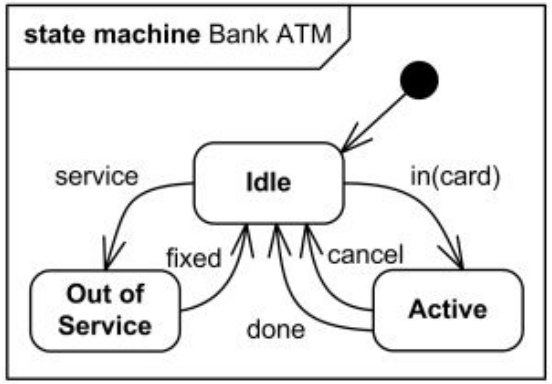


State diagram

- Describes possible states of an object as well as possible state transitions (finite automaton)
- Based on the interaction diagrams for classes with "interesting behavior", e.g.:
  - Real things that are called "automatic" (cash machine, garage opener, washing machine)
  - Communication protocols
  - Interactive devices
- Validity
  - For the entire life cycle
  - For the execution of an operation

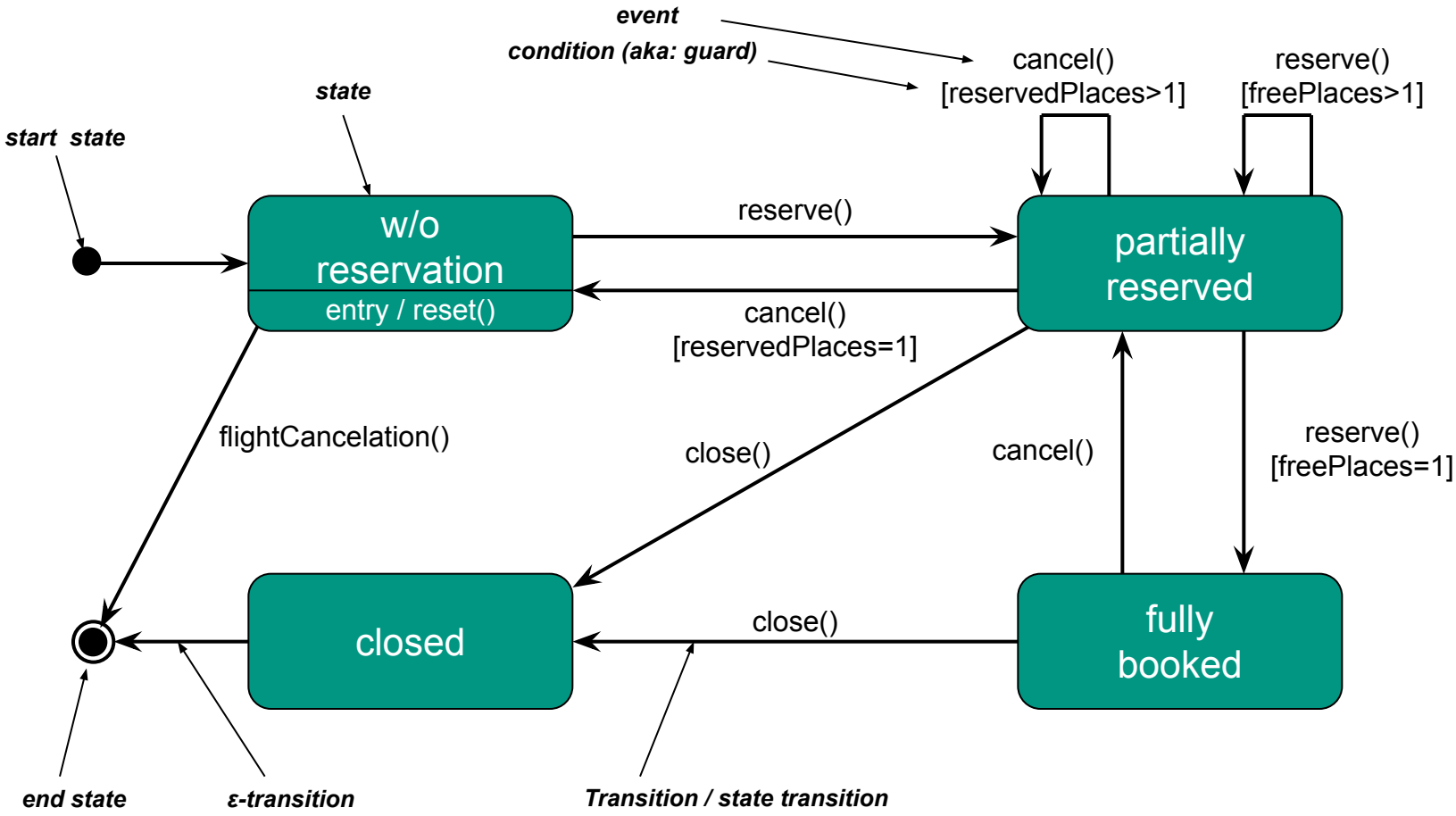
State diagram – Example

- High level behavioral state machine for a bank ATM



Source: https://www.uml-diagrams.org

Example: Flight reservation



State diagram

- A state transition is triggered by an event
  - The transition descriptions are entered in the following form on the arrows:

    ```
    event(arguments)
    [condition]
    /Operation(arguments)
    ```
- A state transition only takes place if, at the time the event occurs, the corresponding condition is also valid (guarded transition).
- Special case**: ε-transition** (also “spontaneous transition”), needs no event, can be done at any time, if:
  - The system is in the state and
  - The condition is met

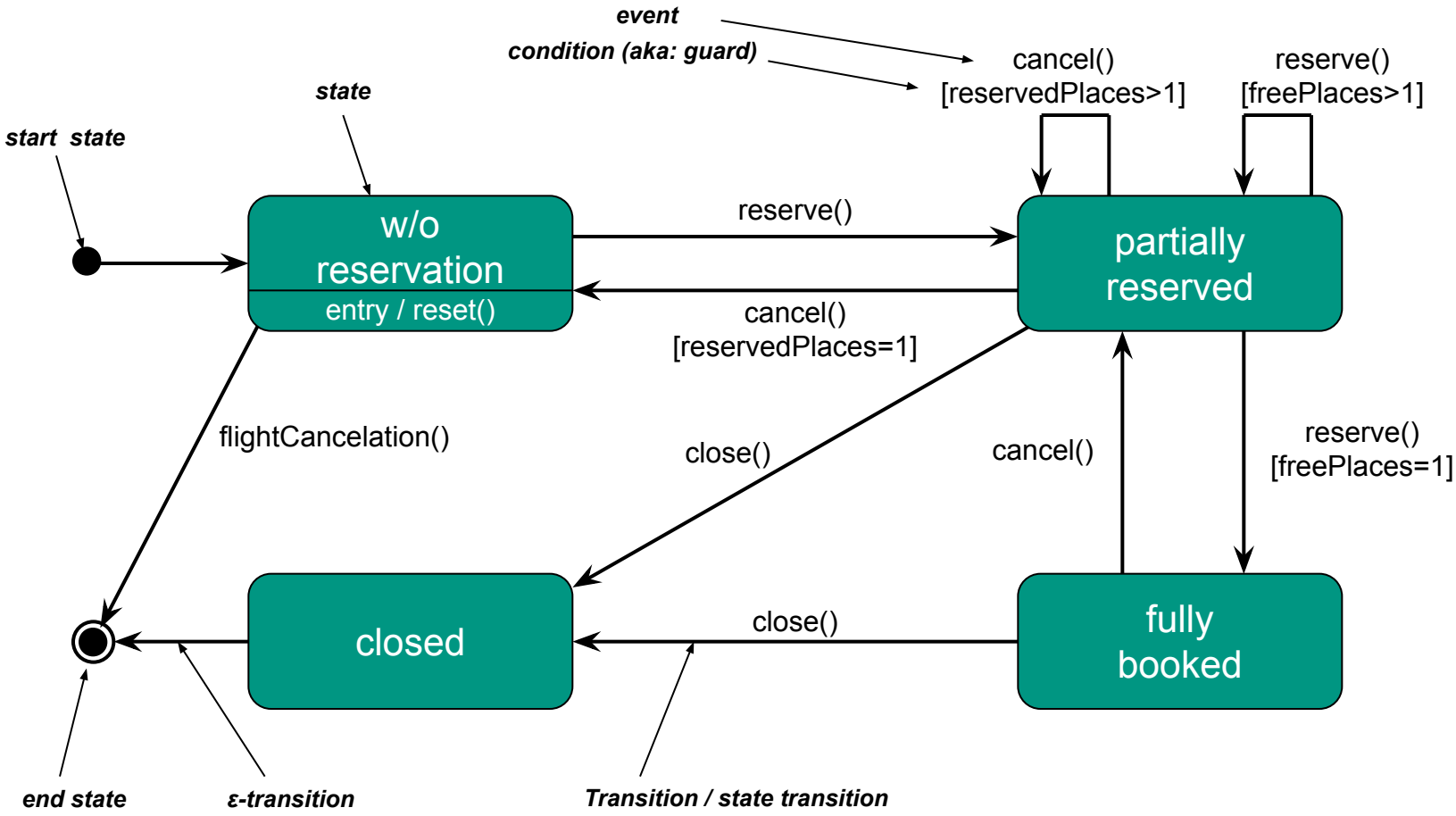
State diagram

- Special events
  - **at** (expression): The expression describes **an exact absolute time**. Once the timing is reached, the transition happens.
  - **after** (expression): Here the expression must describe a **relative time**.

State diagram – Action

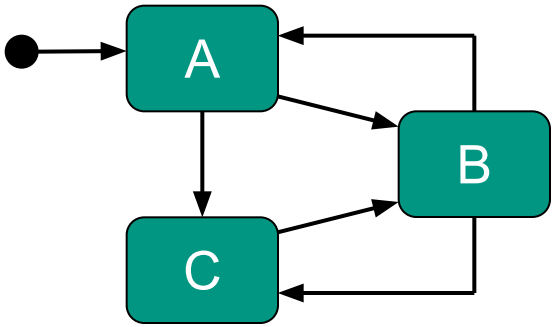
- Actions
  - An action can be associated with a state transition.
  - **Entry action**: will be executed when transitioning to the state.
    - `entry / action()`
  - **Exit action**: will be executed when transitioning from one state to another
    - `exit / action()`
- An action is executed immediately at the corresponding transition and does not require any (or negligible) time

Example: Flight reservation

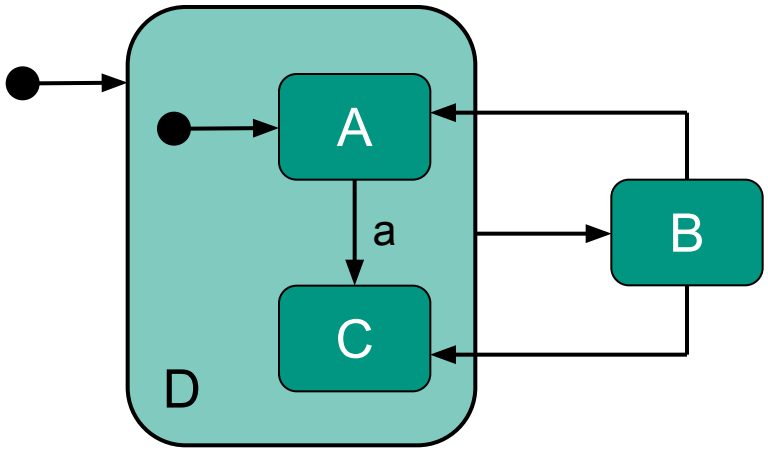


State diagram – Hierarchical state machine

- Hierarchical state machine



$\widehat{=}$

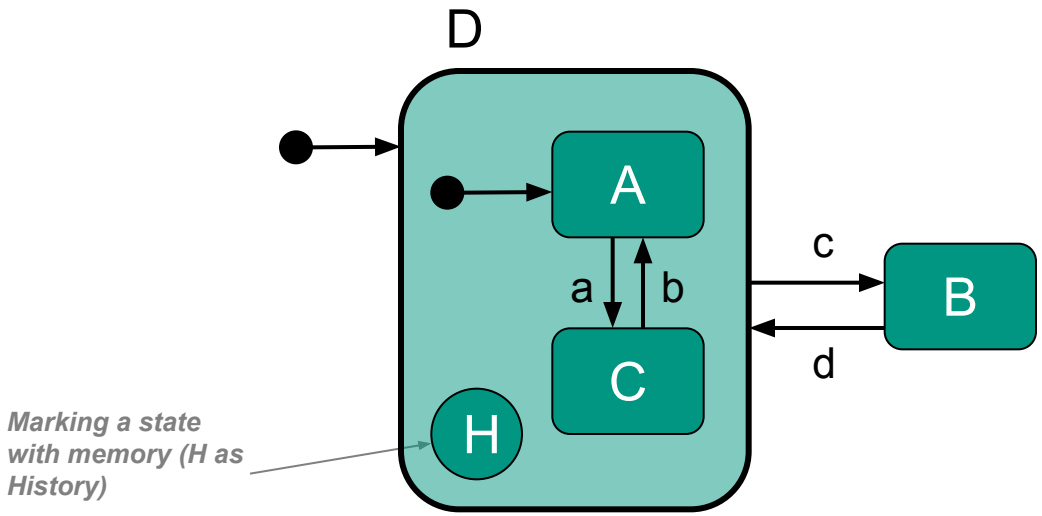


w/o hierarchiy

with hierarchy

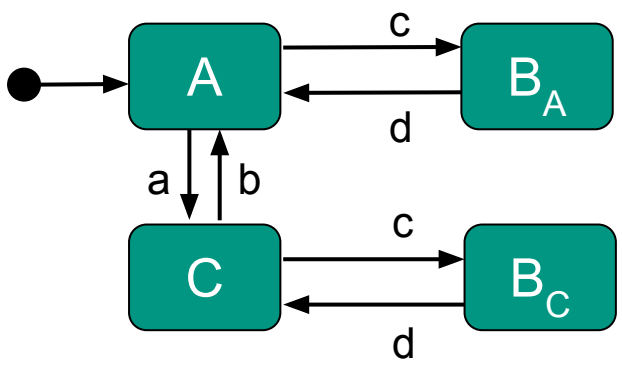
State diagram – States with memory

- States with memory
- When transitioning to a state with sub-states, the system returns to the state last assumed



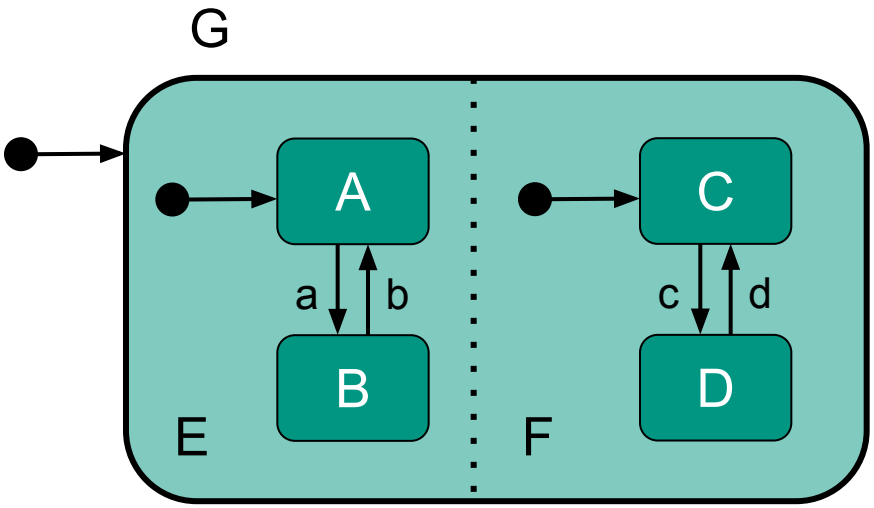
State diagram – States with memory

- Is an automaton (state diagram) with memory more powerful than a "normal" finite automaton without memory?
  - No, because you can simulate memory with eventually many additional states.
  - Simulation of the previous example without memory:



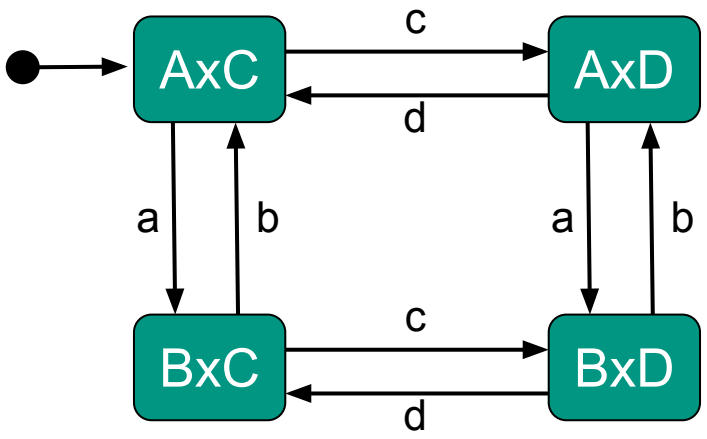
State diagram – Concurrency

- Concurrency
- While System lingers in state G, it can accept all state combinations of E × F

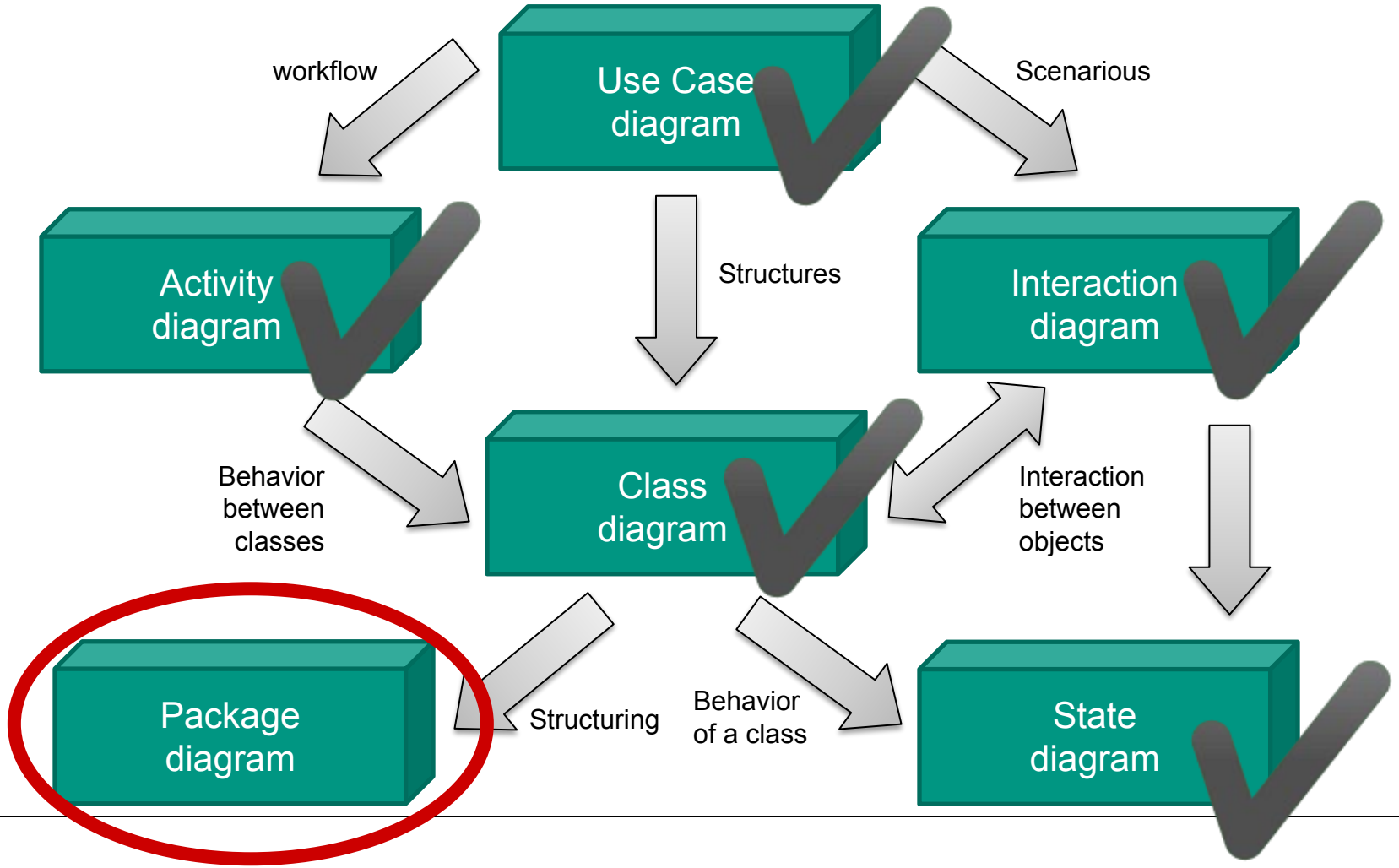


State diagram – Concurrency

- Is a concurrent automaton (state diagram) more powerful than a "normal" finite automaton?
  - No, because you can simulate the parallelism with eventually many additional states.
  - Simulation of the previous example without parallelism:



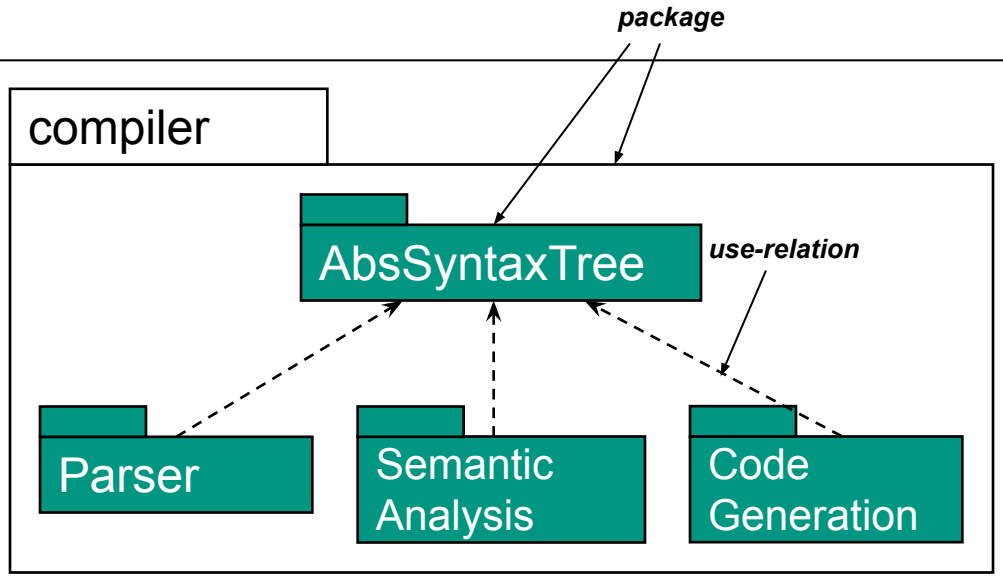
UML diagrams



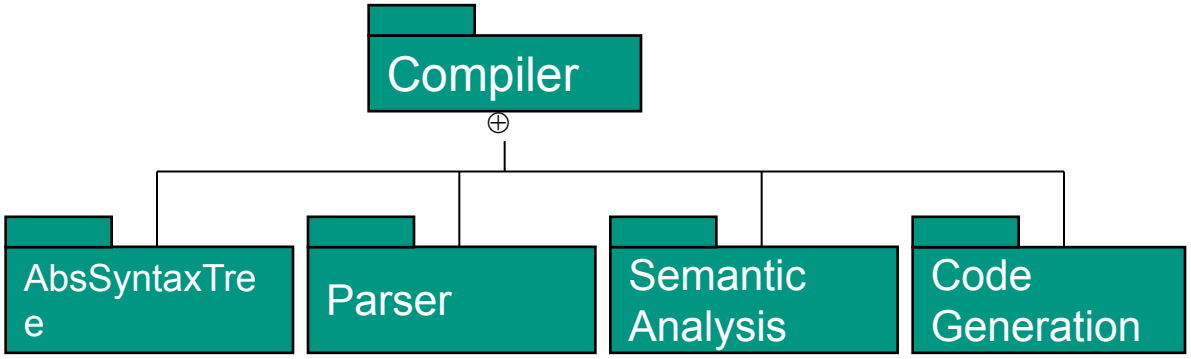
Package diagram

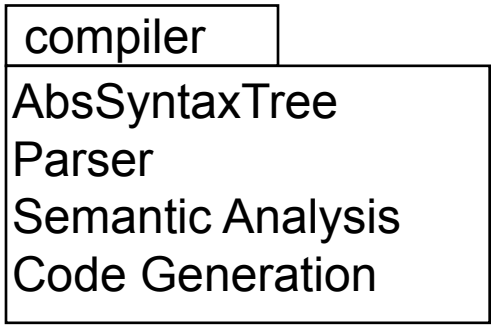
- Packages are collections of model elements (ME) of any type (e.g., use cases, classes, ...)
- The package diagram serves to **structure** the overall model into manageable and clear units
- An ME has a unique name within the package
- An ME can be provided with visibility
  - `Standard visibility is public`
- An ME can be cited in other packages by its qualified name:
  - `Package-name::ME-Name`
- In a package diagram dependencies between packages are shown with a dashed arrow.

Example



Alternative: (without use-relation)





Summary

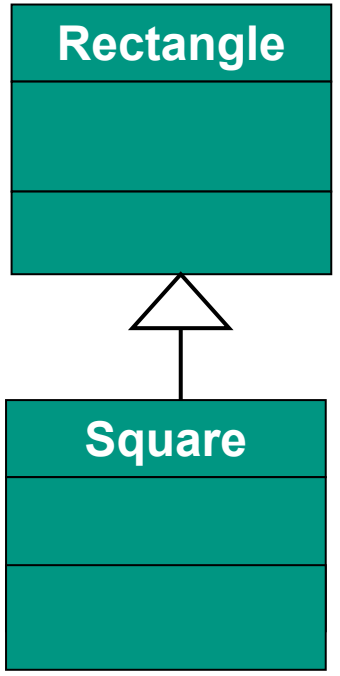
- UML diagrams
  - Class diagrams
  - Interaction diagrams
    - Sequence diagrams
  - State diagrams
  - Package diagrams

Literature – UML

- More:
  - http://www.uml-diagrams.org/
  - UML specification: http://www.omg.org/spec/UML/2.4.1/

What are the constraints to set? (1)

- Assertion
  - Pre-condition
  - Post-condition
  - Invariant
- Example
  - Is this modeling ok?
  - Obviously, the set of squares is a subset of the rectangles.



What are the constraints to set? (2)

- Structurally, the modeling is correct because squares are actually a subset of the set of rectangles.
- But in behavior the two classes are not consistent.
- We need to check that the pre-conditions and post-conditions ”match" before we replace an object with a subclass.
- Only when superclass objects are completely substitutable by subclass objects the subclass objects can be used without harm in the context of the superclass.

Signature adaptation

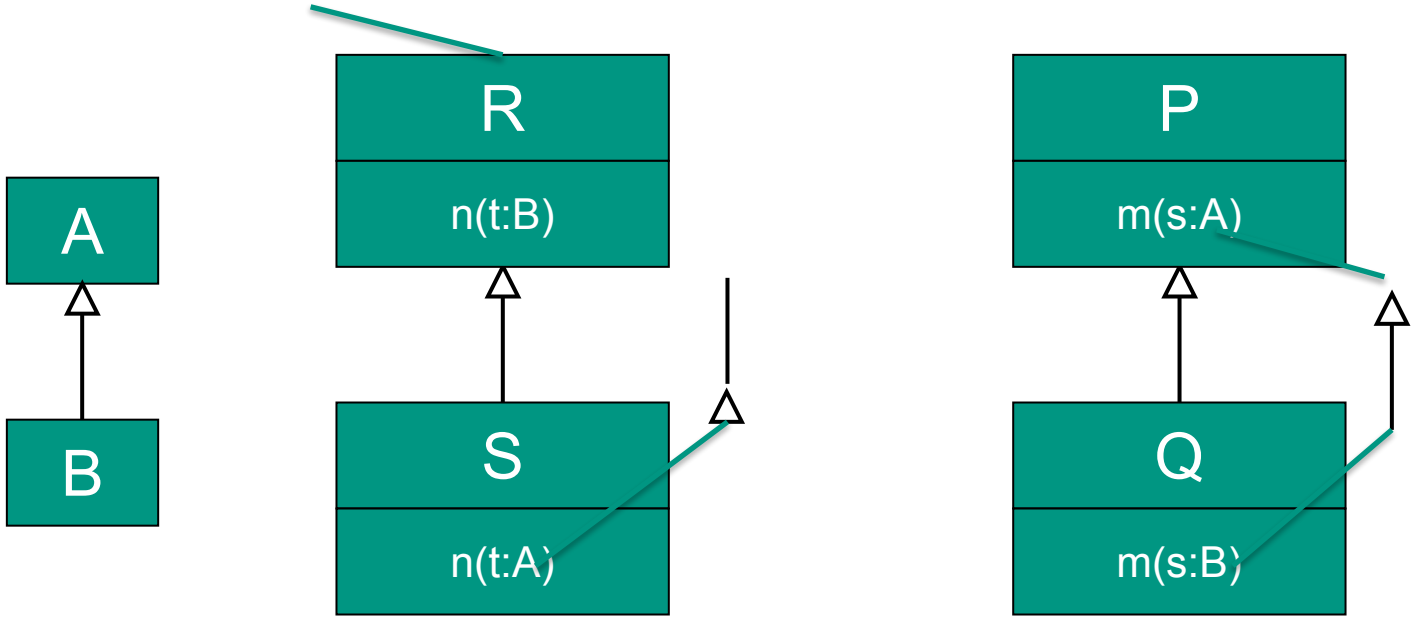
- The substitution principle only requires that you can use an instance of the subclass as if it were an instance of the superclass.
  - It is not required that the signature remains the same!

 Changes to the signature should be possible! - but how?

Parameter variance – Signature adaptation

- **Variance**: Modification of the types of parameters of an overridden method
- **Covariance**: Use of a specialization of the parameter type in the overriding method
- **Contravariance**: Use of a generalization of the parameter type in the overriding method
- **Invariance**: no modification of the type

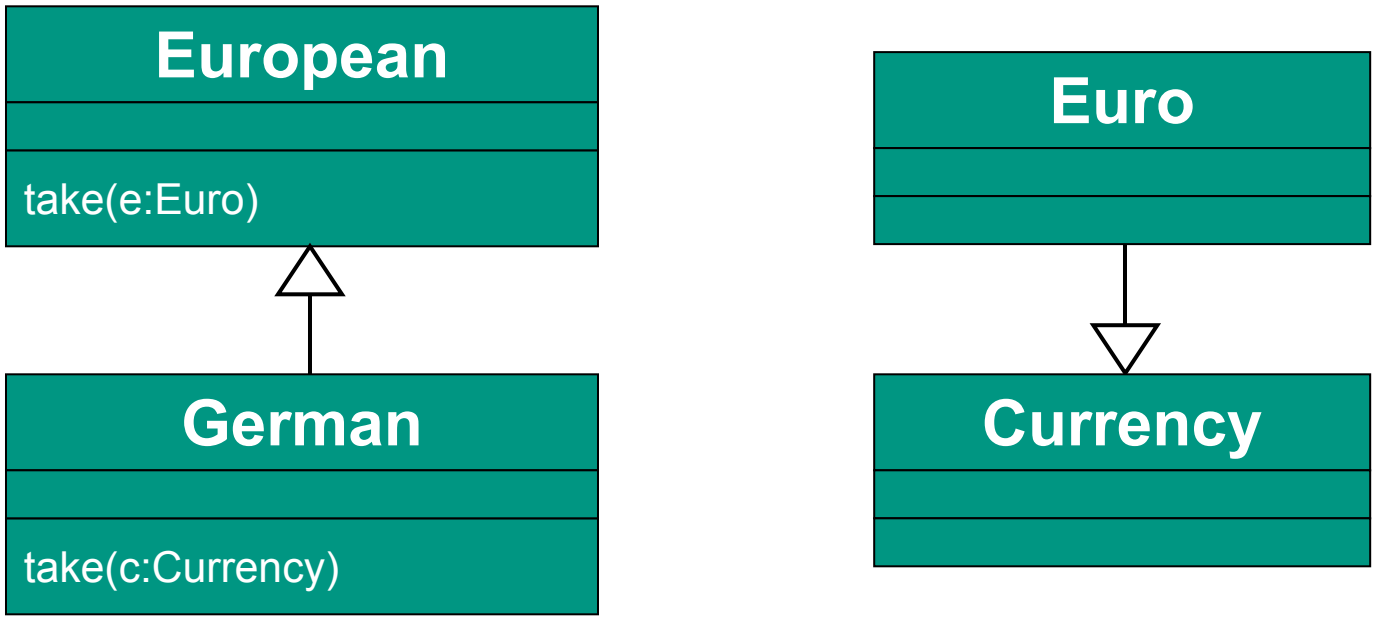
Contravariance/Covariance – Example



contravariance
of parameter t

covariance
of parameter s

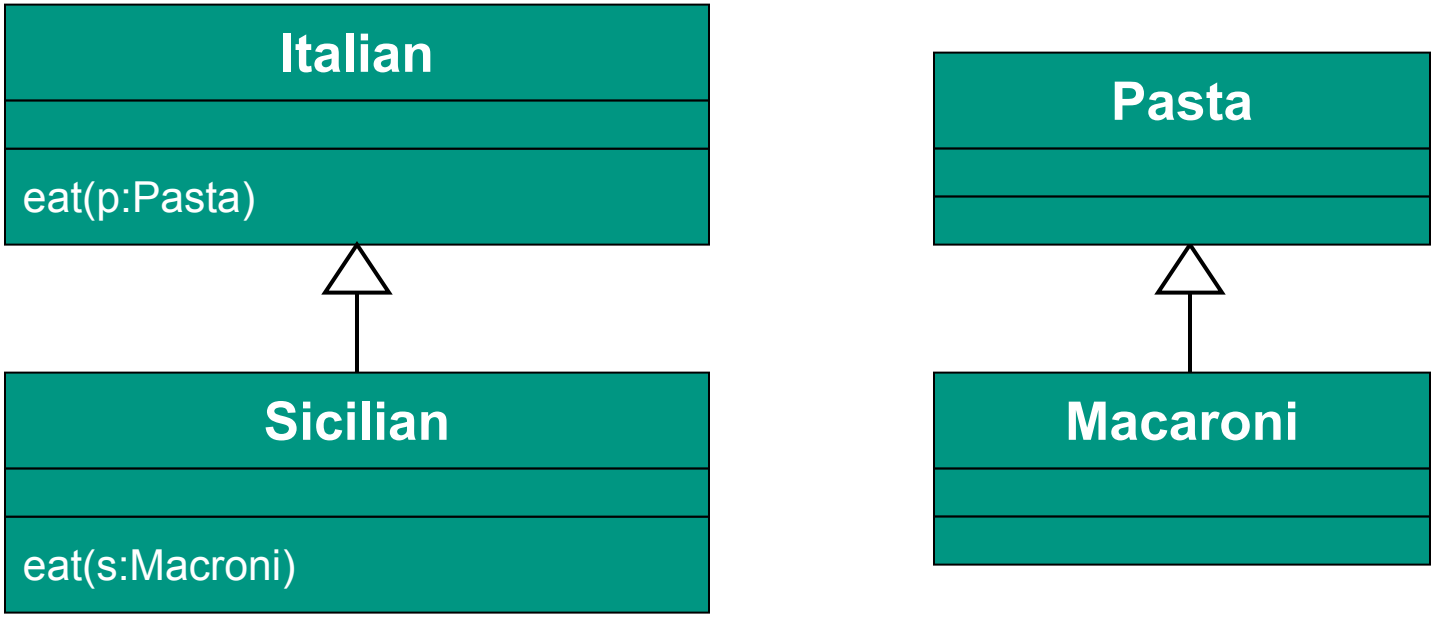
Contravariance and substitution principle – Example



Contravariance and substitution principle – Example

```
European e;
German g;

Currency w;
Euro eu;

e.take(eu); ✔

e = g;
e.take(eu); ??
```

Covariance and substitution principle – Example



Covariance and substitution principle – Example

```
Italian it;
sicilian s;

Pasta p;
Macaroni ma;

it.eat(p); ✔

it = s;
it.eat(p); ??
```

Question …

What is allowed for input parameter?

- Variance (in general)
- Covariance
- Contravariance
- Invariance

Permitted variance – Substitution principle

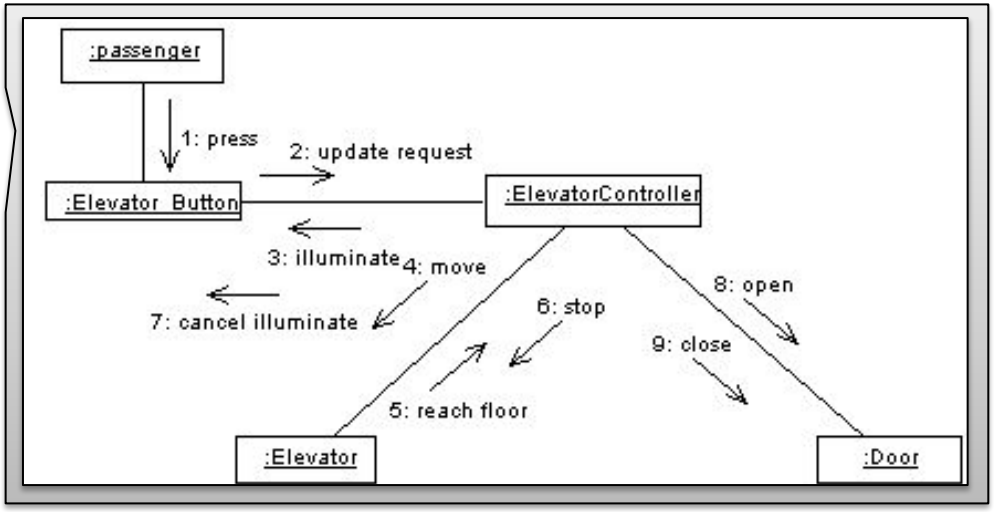
- In order to fulfill the substitution principle, the following modifications of the parameter types are possible for overriding methods:

Input parameter	contravariance
Output parameter (also return values and exeptions)	covariance
Parameters that are both input and output parameters	invariance

- Note: Not all variations are allowed in Java or C#!
- In Java:
  - Input parameters of a method: only Invariance
  - Return (output) types: Invariance and covariance

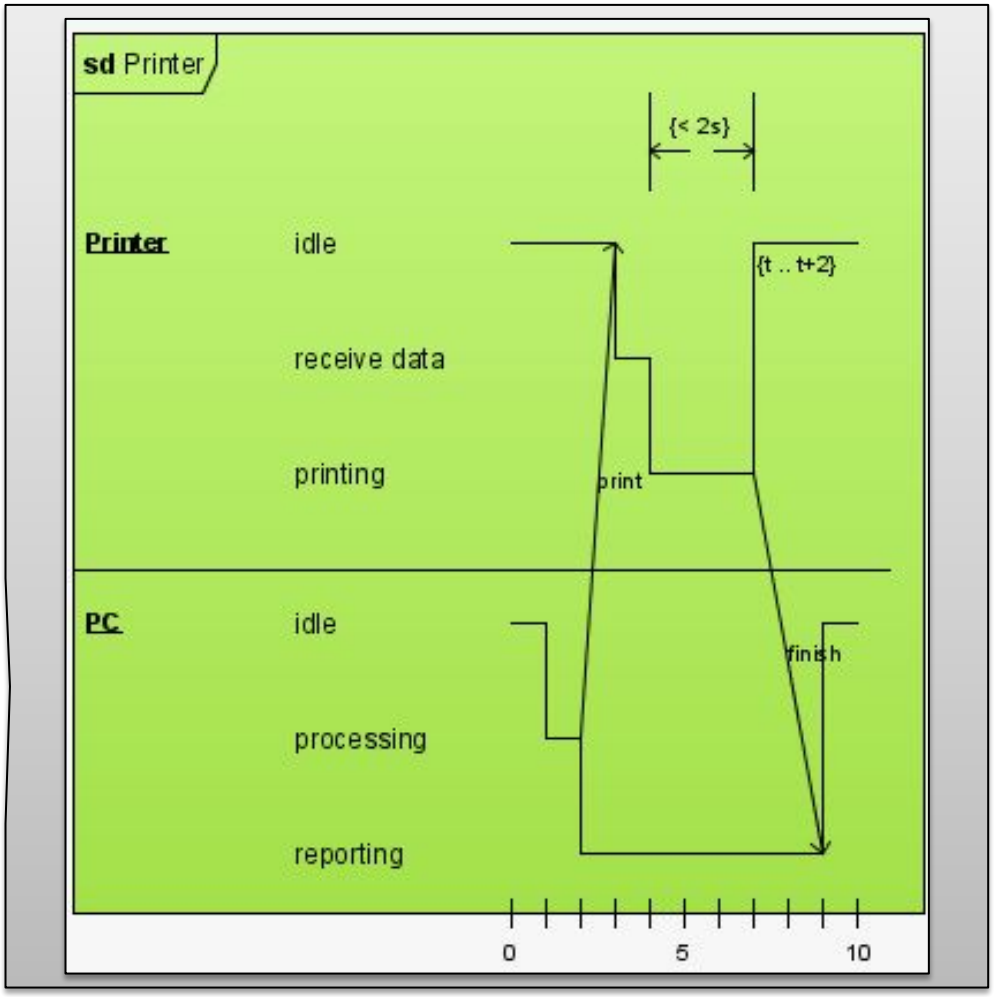
Interaction diagrams – Collaboration diagram

1. **Collaboration diagram** / communication diagram
   - Focus: **Structure** of the interaction partners
   - Example: Elevator (Illuminate button)



Interaction diagrams – Time chart

2. **Time chart**/diagram
   - Focus: **Temporal** coordination
   - Example: Printing process



Interaction diagrams – Overview diagram

3. Interaction overview (overview diagram)

- Activity diagram to **illustrate** complex sequence diagrams
- Example: ACSystem