Software Construction and User Interfaces (SE/ComS 319)

Ali Jannesari

Department of Computer Science

Iowa State University, Fall 2018

OBJECT MODELING

Outline

- Object modeling
- Object Constraint Language (OCL) – Self-study (not relevant for the exam)
  - Example

Recap

- Planning phase/game (analysis): functional model
  - What are the scenarios, use cases?
  - Documented in the specifications (in story cards)
- How do you do that?
  - Question customers, clients
  - Study work processes, forms, existing systems
  - Study scope, use your own world knowledge

Recap – Static model

- Then comes the **object modeling**
  - What are the classes?
  - Which associations exist? (including multiplicities/cardinalities)
  - Which attributes are needed?
  - Which methods (functions)?
  - **Last**: develop inheritance structures
    - Extract commonalities from classes and place them in super classes
    - Pay attention to the substitution principle.
- Object model is a **static model**

Recap – Dynamic model

- Dynamic model
  - Activity diagram
    - Describe parallel and sequential processes
  - Sequence diagram
    - Identify methods sent between objects of specific classes
    - Complete class diagram with new methods
    - Show call flows between several objects
  - State diagram
    - State transitions within a single object

Steps for object modeling

1. How do you find classes?
2. How to find associations?
3. How to find attributes?
4. Creating inheritance structure
5. Create a dynamic model
6. Determine the object life cycle
7. Define operations (methods)

1. How do you find classes?

- Can concrete objects be identified?
  - In technical systems, the real objects are potential objects for the system
  - In commercial systems, objects are often found in forms
- **Example**: identify object in the following requirement?
  - The elevator closes its door before moving to another floor
  - The **elevator** closes its **door** before moving to another **floor**

1. How do you find classes?

- Which abstractions exist in the legacy (Software) system?
  - Analyze existing forms (and contracts) or legacy systems
  - Take attributes and combine them into classes
    - This is a "bottom-up" approach

Example – Seminar organization (1)

Enrollment From for TEACHWARE seminars

a) Register as a participant to the following TEACHWARE seminars:

Title **first name** **Surname**

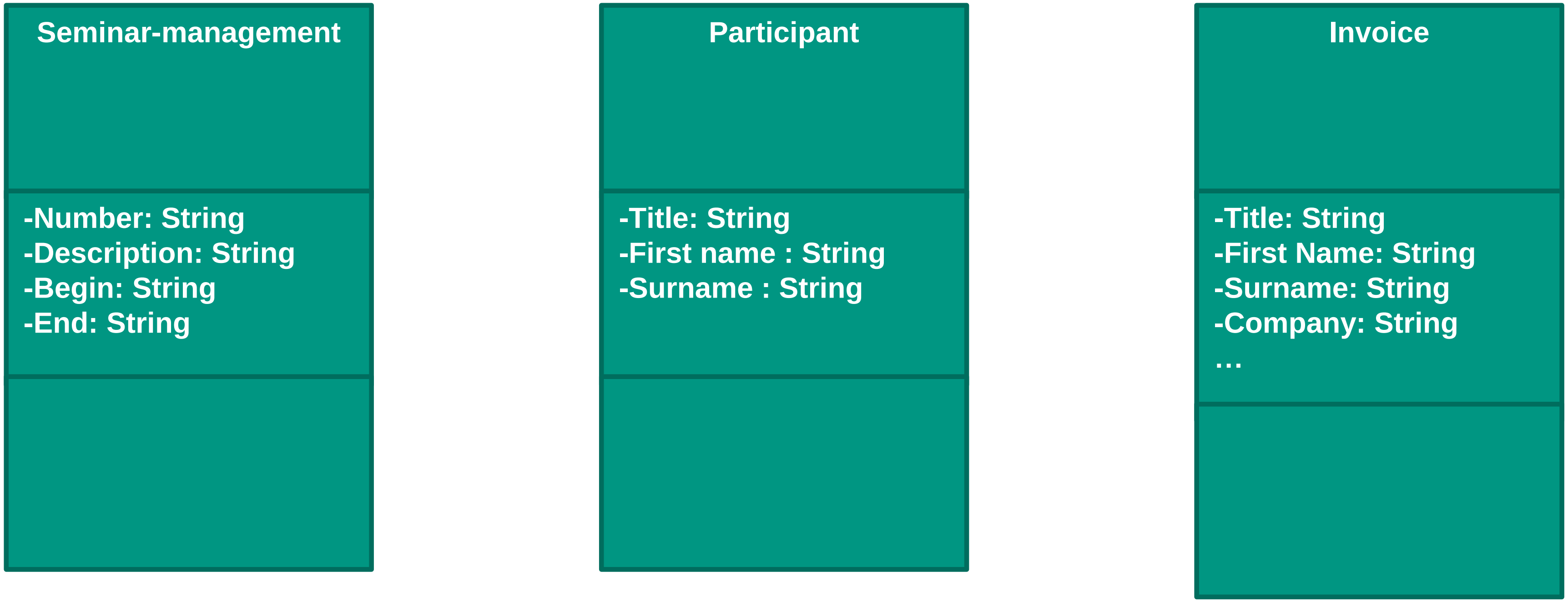
Event no. **seminar Description** **begin** **end**

b) Enrollment confirmation and invoice requested to:

Title **first name** **Surname**

company **Address** **phone**

Example – Seminar organization (2)



1. How do you find classes?

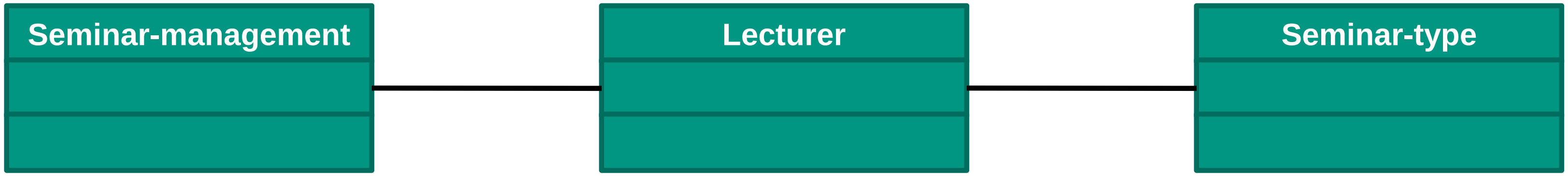
- **Document analysis**: Find the classes from scenarios and requirements (top-down approach)
  - **Syntactic** analysis
    - **Noun**-verb analysis
    - Example: The **elevator** closes its **door** before moving to another **floor**
  - **Content** analysis to find:
    - Attributes for potential classes
    - Actors which the system needs to remember something about them (are potential classes)

2. How to find associations?

- Are there permanent relationships between objects?
  - Do they exist for a longer period of time?
  - Are the associations relevant to the problem?
  - Does the relationship exist independently of all non-participating classes?
- Verbs in the problem description?
  - Spatial proximity (lives in)
  - Actions (drives)
  - Communication (talks to)
  - Possession (has)
  - General relationships (married to)

Example – Finding associations

- Example: Seminar organization requirement
  - Creating, modifying and deleting of lecturers as well assigning them to different seminars and seminar types /F180/
- Based on the given requirement, How many classes and associations you could find?



3. How to find attributes?

- Attribute candidates can also be identified by means of
  - Document analysis
  - Analysis of existing forms, or
  - Search for real properties ("technical data")
- Does the attribute belong to a class or association?
  - Does the attribute belong to every object of the class, even if the class in question is considered isolated from all other classes which it is associated with?
  - If yes, then the attribute belongs to a class
  - If not, then check whether it can be assigned to an association
  - If no assignment is possible, possibly a class or relationship is missing.

4. Creating inheritance structure

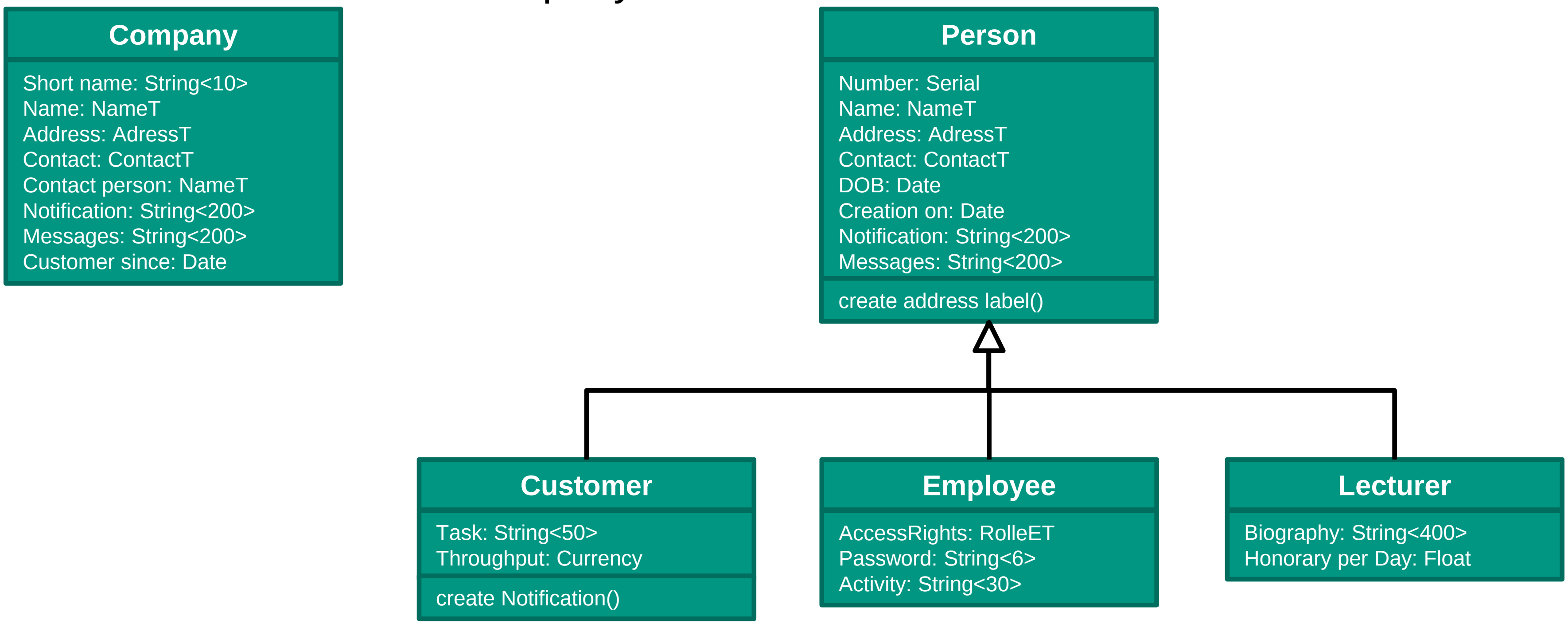
- The classes Sub1 and Sub2 are subclasses of the class Super, if:
  - It is necessary to distinguish the subclasses Sub1 and Sub2 in the system to be modeled
  - Although subclasses Sub1 and Sub2 have the attributes / operations of Super in common, they also have their own (**different**!) attributes/operations
  - There is an is-a relationship (substitution principle) between Sub1 or Sub2 and the superclass Super

4. Creating inheritance structure

- When is there no inheritance?
  - The classes Sub1 and Sub2 contain no (additional) attributes or operations, no operation is overwritten.
  - The class names of Sub1 and Sub2 indicate different types of the Super class.

Example – Creating inheritance structure

- A company and a person have many similarities but also major differences (e.g. a Company has no DOB).
  - For this reason a Company is not a subclass of a Person



5. Create a dynamic model

- **Input**: scenarios, use cases
- **Output**: sequence and activity diagrams
- Purpose:
  - Identify **operations** of the classes
  - Defining the flow of messages through the system
  - Check **completeness** and **correctness** of the static system
  - Create basis for system tests
- Determine several sequence diagrams from each business process
  - Are there variants in the business process, then different diagrams: one for each variant
  - Distinguish positive and negative cases

6. Determine the object life cycle

- **Output**: state diagram for classes, where necessary!
- Does the object have a "non-trivial" life cycle?
  - IBM * indicates that for typical information systems only 1-2% of classes have a nontrivial life cycle.
  - This is different for embedded systems, where control objects contain complex state diagrams.
- A trivial lifecycle occurs when there is only one state between initialization and destruction.
- In this case do not fall into "analysis paralysis"! (over-thinking, a situation so that a decision or action is never taken).

*Developing Object-Oriented Software – An Experience-Based Approach, Prentice Hall, 1997

7. Define operations (methods)

- Take over operations from sequence diagrams and object life cycles
- Enter operations as high as possible in the inheritance hierarchy
- Create a description

Analytical steps:

- Does the operation have a proper name?
  - Begins with a verb
  - Describes what is done
- Reasonable scope
  - especially not too extensive
- Functional binding
  - Each operation realizes one (1!) self-contained function

Subsystems – Modeling large systems

- Combine individual classes with common reference to a subsystem or package.
- Within a subsystem: "**strong cohesion**"
  - A strong cohesion exists when the subsystem
    - contains a topic that can be viewed and understood on its own
    - has a well-defined interface to the environment
    - has not just a set of classes, but the subsystems allow for a higher-level view of the system
- Between the subsystems: "**weak coupling**"
  - The interface between subsystems should contain as few associations as possible

Object modeling – References

- Meyer: Object-oriented software construction, 2. edition, Prentice Hall, 1997
- Bruegge, Dutoit: Object-Oriented Software Engineering, 3rd edition
- …

Self-Study (not relevant for the exam)

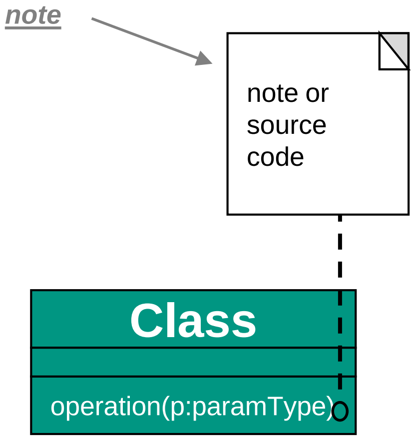
OBJECT CONSTRAINT LANGUAGE (OCL)

Object Constraint Language (OCL)

- **Constraint** (aka assertion): an expression that describes the permissible characteristics, contents or states of a model element and must always be evaluable to "true”
- Formulation of constraints at the model level with Object Constraint Language (OCL).

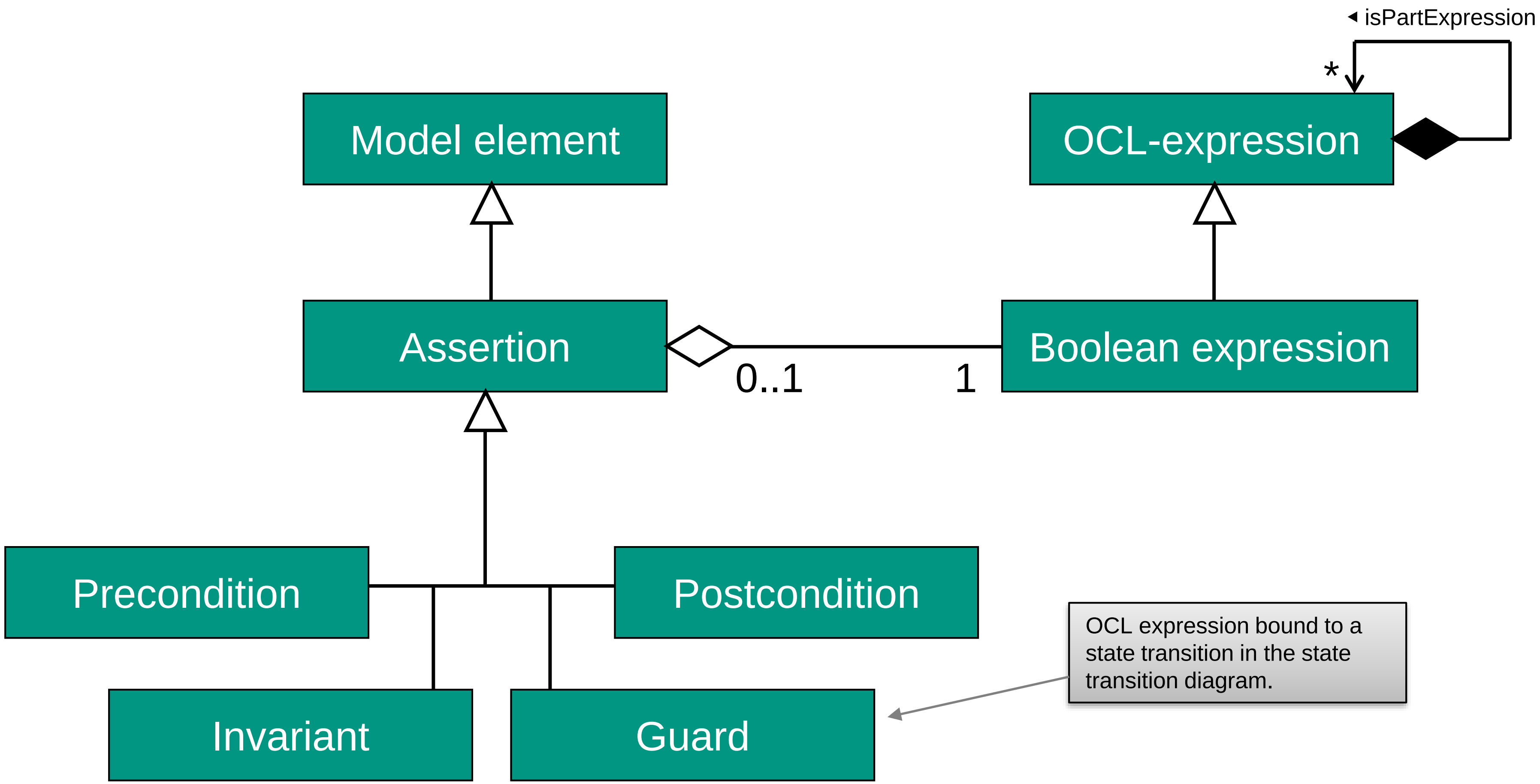
Assertions

- An assertion describes a **condition** that must be always valid
  - It can restrict the permissible **value** set of an attribute (domain restriction)
  - It can specify **precondition** and **postcondition** for messages or operations
  - Set a special **context** for messages or relationships
  - Assure **structural** properties
  - Define **orders**
  - Defining **temporal** relationships
  - Etc.



⇒ Assertions are ultimately an approach to formalizing the note.

Example: Meta-Model of Assertions in UML (Simplified)



Notation of assertions in OCL

- **Invariants** can be defined in curly brackets directly behind a model element:
  - Template:

    `{assertion}`

    `{Name: assertion}`

  - Example:
    - For a `radius` attribute, we specify:

      `radius: float {radius>= 0.0}`

Notation of assertions in OCL

- Separate with the keyword `context`:
  - Template:
    - ```
      context ResponsibleElement
              pre name0: asserion0
              inv name1: asserion1
              post name2: asserion2
      ```

      precondition *(optional)*

      Invariant *(optional)*

      postcondition *(optional)*

      Name of assertion *(optional)*
  - Example:
    - ```
      context Person
              inv majority: self.age >= 18
      ```
  - The predefined `self`-reference refers to each copy of the named class.

Notation of assertions in OCL

- **Assertions for operations**
  - Template:

Type and operation to which the assertaions refer

Type of return value

```
context Type1.operationX (param1: Type2): Type3
        pre Name0: assertion0
        inv Name1: assertion1
        post Name2: assertion2
```

- Here, `param1` has been explicitly defined as a parameter of the `operationX` operation.
- In the assertion expressions, all parameters and all attributes of the type can be used.

Assertion of preconditions and postconditions

- In the expression of the postcondition the predefined variable `result` may be used. It contains the result, i.e. the **return value** of the operation.
  - `context Rect.getArea (): Float`

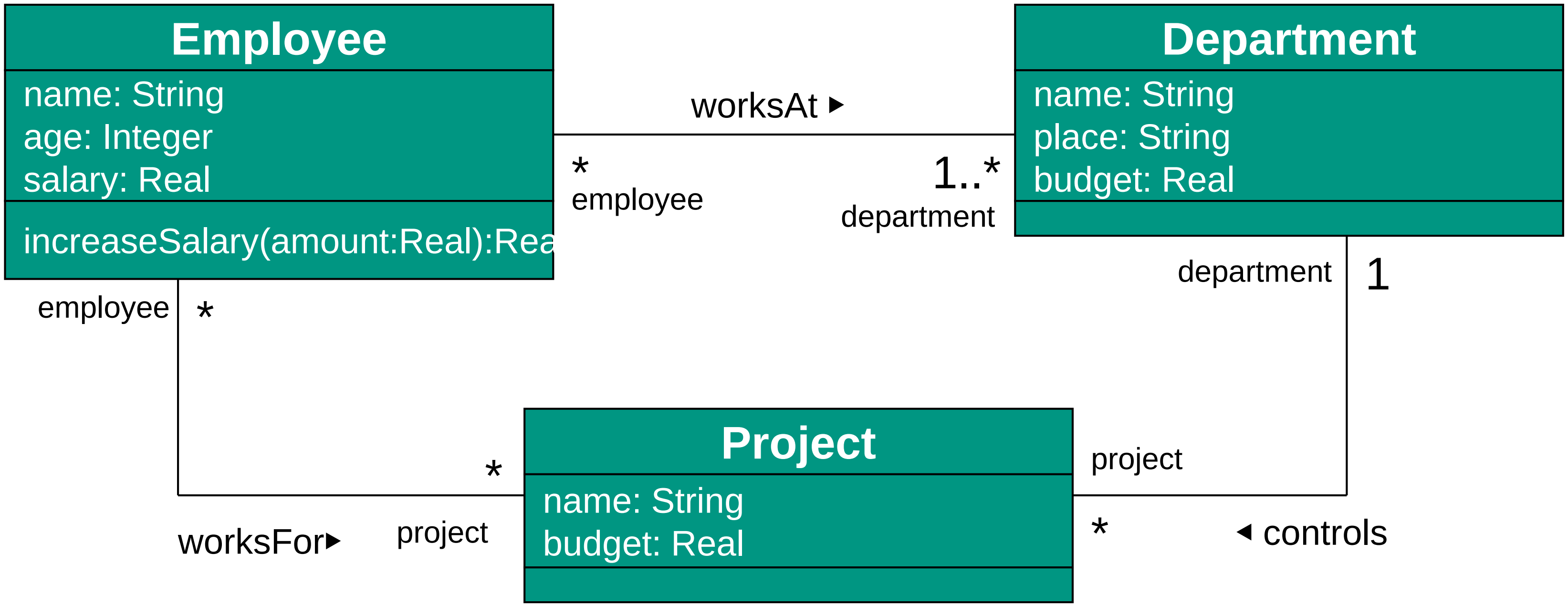
    `post: result = self.width * self.height`
- The value of an attribute or parameter **before the calculation** comes with the suffix "`@pre`”
  - `context Person.birthDayHappens ()`

    `post: age = age@pre + 1`

Predefined OCL types

- **Predefined OCL base types**
  - Boolean: true, false
  - Integer: -1, 2, 543523553
  - etc.
- **Predefined OCL base operations**
  - i1 = i2 Result: Boolean
  - i1 + r1 Result: Real
  - c->sum() integer or real (sum of all elements, if number)
  - c->includes(e) Boolean ($e \in c$?)
    c->isEmpty() Boolean ($c = \emptyset$?)
  - etc.

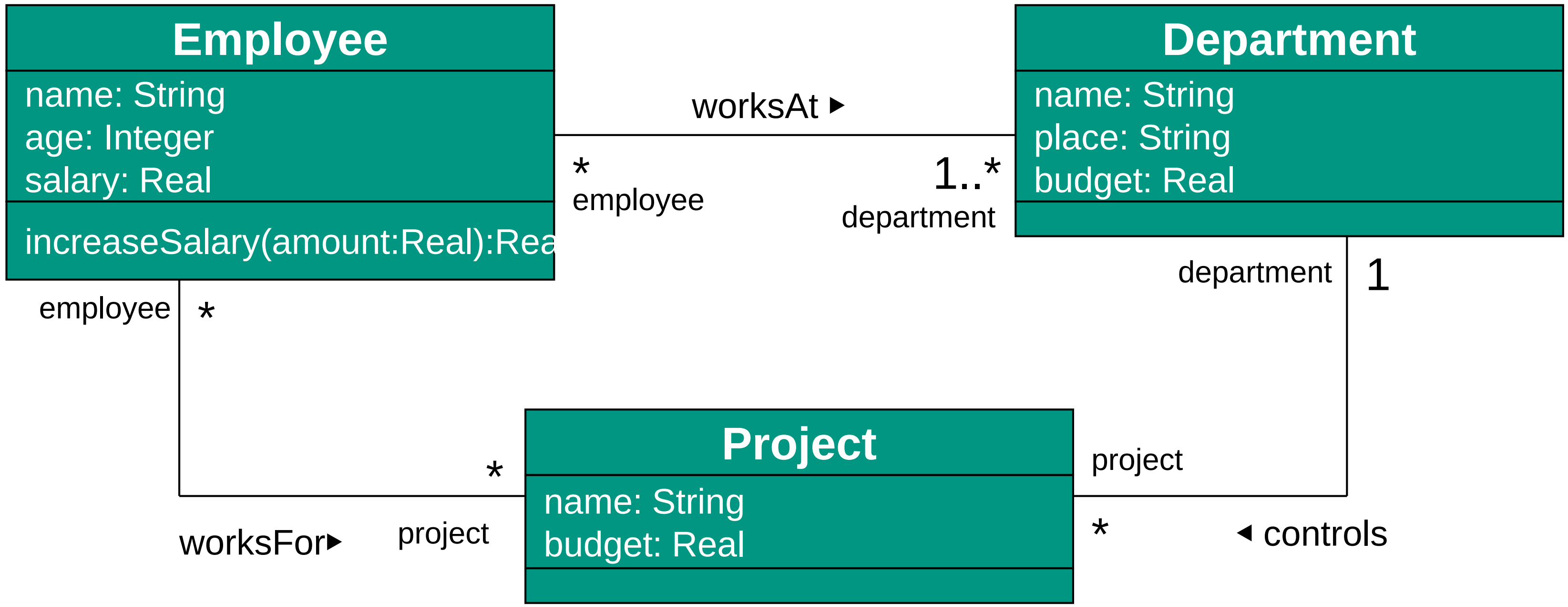
OCL – Example



- No department should ever have a negative budget:

```
context department inv:
        self.budget >= 0
```

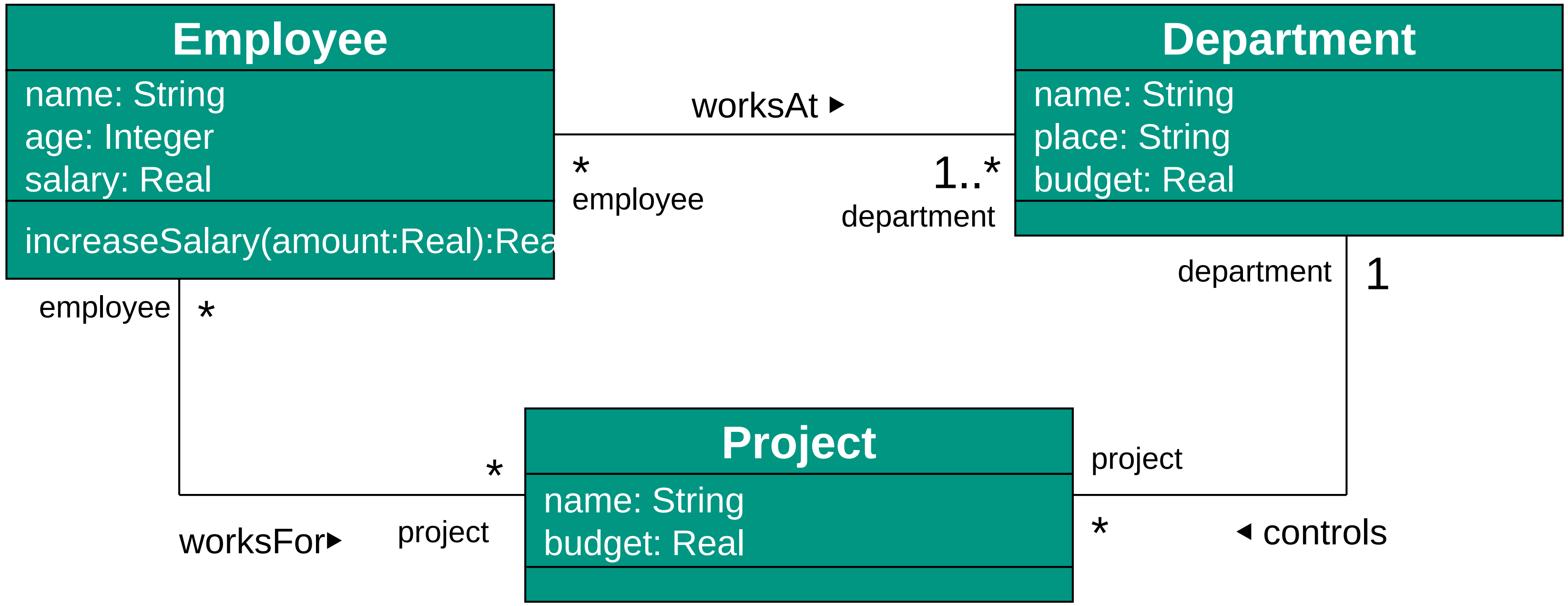
OCL – Example



- The code for salary increase must meet the following requirements:

```
context Employee.increaseSalalry(amount: Real): Real
    pre: amount>0
    post: self.salary = self.salary@pre + amount
    and result = self.salary
```

OCL – Example



- Employees who work on more projects receive a higher salary than other employees of the same department:

```
context department inv:
self.employee->forAll (a1, a2 | a1.project->size() >
a2.project->size() implies a1.content>a2.content)
```

OCL – References

More about OCL:

- Richters & Gogolla, „OCL: Syntax, Semantics, and Tools“ in A. Clark and J. Warmer (Eds.): Object Modeling with the OCL, LNCS 2263, pp. 42–68, 2002.
- „Object Constraint Language – Version 2.0“, OMG http://www.omg.org/cgi-bin/doc?formal/2006-05-01

Summary

- Object modeling
- OCL
  - Examples