# Software Engineering (OCL)

## Object Constraint Language

Lina YE


https://www.lri.fr/~linaye/GL.html

lina.ye@centralesupelec.fr

Sequence 3, 2017-2018

# Plan

# Motivation

## Why OCL

- Cannot represent all the relevant aspects of a specification (e.g., class diagram)
- Need to describe additional constraints without ambiguities
- Formal languages requires a strong mathematical background

## What is OCL

- Formal language to express constraints, that remains easy to read and write
- Developed by IBM and standardized by OMG
- Integrated into the UML standard

# Objet Constraint Language

## Where to use

- Specify invariants on classes and types in the class model
- Describe pre- and post-conditions on operations and methods
- Describe guards
- As a navigation language
- etc.

## Example

- How to represent the constraint that the age of an employee cannot be smaller than 18?

# Objet Constraint Language

- Pure specification language: do not have side effet
  - evaluation of an OCL expression returns a value
  - this evaluation do not alter the system state and is instantaneous

# Objet Constraint Language

- Pure specification language: do not have side effet
  - evaluation of an OCL expression returns a value
  - this evaluation do not alter the system state and is instantaneous
- Not a programming language
  - cannot write program logic in OCL
  - cannot invoke processes or activate non-query operations

# Objet Constraint Language

- Pure specification language: do not have side effet
  - evaluation of an OCL expression returns a value
  - this evaluation do not alter the system state and is instantaneous
- Not a programming language
  - cannot write program logic in OCL
  - cannot invoke processes or activate non-query operations
- Typed language: each expression has a type
  - expression must obey the type conformance rules of OCL
  - each classifier defined in a UML model represents a distinct OCL type
  - includes a set of supplementary predefined types

# Context

- Each constraint must be associated to one model element
- Such an element constitutes the context of the constraint
- Syntax: keyword context

## example

- class: nameClass
  context Person
- operation: nameClass::nameOperation(param1: Type1,...):TypeReturned
  context Account::getSolde(): Real
- attribute: nameClass::nameAtt: TypeAtt
  context Person::age : Integer

# Self

- In an OCL expression, reserved word self is used to refer to the contextual instance
- If the context is Person, then self refers to an instance of Person
- This keyword can be omitted when the context is clear

## example

- context Person
  self.name
- context Person
  name

# Invariant

- Determine a constraint that should be always true for all instances of a type
- Syntax

  inv: <logic expression>

## example

- Value of attribute nbEmployees in instances of Company must be less than or equal to 50

  context Company

  inv: self.nbEmployees≤50
- The stock price of each company is greater than 0 (stockPrice() is a operation defined in the class Company)

  context Company

  inv: self.stockPrice()>0

# Pre-condition

- Constraints associated with an operation or other behavioral feature
- Constraint assumed to be true before the execution of the operation
- Syntax
  pre: <logic expression>

## example

- The age of a person who has an income must be older than or equal to 18 (income() is an operation defined in the class Person)
  context Person:: income(): Integer
  pre: self.age≥18

# Post-condition

- Constraints associated with an operation or other behavioral feature
- Constraint satisfied after the execution of the operation
- Keyword result denotes the value returned by the operation, whose type is the returned type
- Keyword @pre denotes the attribute value before the operation
- Syntax

  post: <logic expression>

## example

- The age of a person who has an income cannot be smaller than 18, and the income must be less than 5000

  context Person:: income(): Integer

  pre: self.age≥18

  post: result<5000

# Init

- Reserved word init is used to represent the initial value
- Possibility to precise the initial value of an attribute or an association end when the object is created
- Syntax
  init: expression

## example

- Attribute isMarried in Person is initialized to false
  context Person:: isMarried: Boolean
  init: false

# Derive

- Reserved word derive is used to represent the derived value
- Precise how to obtain the derived value of an attribute based on the value of other attributes, such a constraint should always be respected
- Syntax
  derive: expression

## example

- The age of one person is obtained by subtracting their birth date from the current date
  context Person:: age: Integer
  derive: currentDate-dateOfBirth

# Access to characteristics of an object

- The access of attributes and operations of an object is specified by a dot followed by their name
- Syntax
  self.nameAttribute
  self.nameOperation($arg_1$, ..., $arg_n$)

**example**

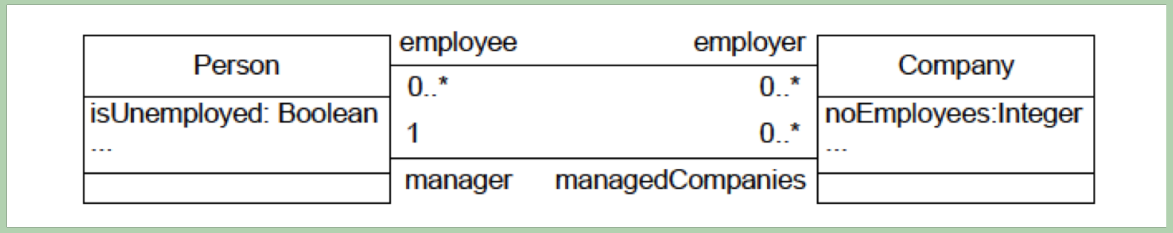
- context Person
  self.age
  self.income()

# Navigation

- From an object, an association is navigated by a dot followed by the opposite role name
- Value of expression depends on maximal multiplicity of the association end
  1: value is an object (".") or can also be used as a set containing a single object ("$\rightarrow$")
  $*$: value is a set of objects ("$\rightarrow$")
- For optional associations, it is useful to check whether there is an object or not when navigating the association

# Navigation: example

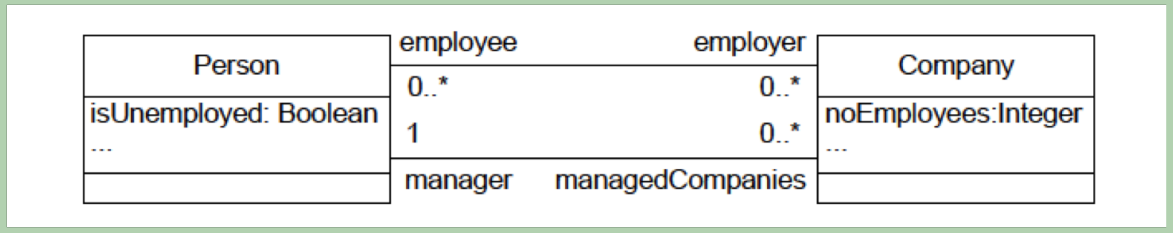
## Example




- context Company
  inv: self.manager.isUnemployed=false
  inv: self.employee→notEmpty()

# Navigation: example

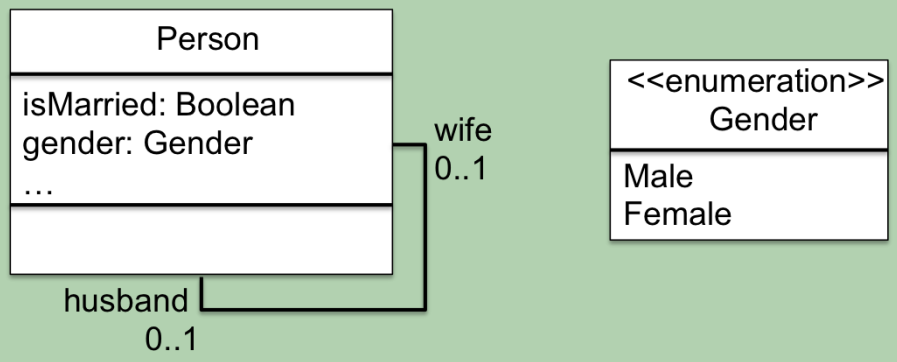
## Example




- context Company
  inv: self.manager.isUnemployed=false
  inv: self.employee→notEmpty()
- self.manager is an object of type Person
  context Company
  inv: self.manager.age> 40
- self.manager as a set
  context Company
  inv: self.manager→ size()=1

# Enumeration: example

## Example




- context Person
  inv: self.wife→notEmpty() implies
  self.gender=Gender::Male and
  self.husband→ notEmpty() implies
  self.gender=Gender::Female

# List of Types

## Predefined types with their operators

| Types | Operators |
|---|---|
| Boolean | and; or; xor; not; implies; if-then-else-endif;... |
| Integer | $*$; $+$; $-$; $/$; $abs()$; ... |
| Real | $*$; $+$; $-$; $/$; $abs()$; $floor()$; ... |
| String | concat(s: String); size(); substring(lower: Integer, upper: Integer);... |

## Implies

| P1 | P2 | **P1 implies P2** |
|---|---|---|
| True | True | **True** |
| True | False | **False** |
| False | True | **True** |
| False | False | **True** |

# Operations on Types

- One person who is married must be more than 18 years old
- One person is either male or female but cannot be both

# Operations on Types

- One person who is married must be more than 18 years old
- One person is either male or female but cannot be both

```
context Person
inv: self.isMarried implies self.age>18
```

# Operations on Types

- One person who is married must be more than 18 years old
- One person is either male or female but cannot be both

```
context Person
inv: self.isMarried implies self.age>18
```

```
context Person
inv: self.gender=Gender::male xor
self.gender=Gender::female
```

# Create a Variable

- If the same expression is used more than one time
- For the better readability of the constraint

## Syntax

- With keywords let...in

  let < *variable* >: TypeVar=< *request* > in

  < *expression* >
- With keyword def

  def: < *variable* >: TypeVar=< *request* >

  < *expression* >

# Create a Variable

## example

- An unemployed person has no job. Otherwise, he has at least one job.

- context Person
  inv: let numberJobs: Integer=self.job→size() in
  if isUnemployed then numberJobs=0
  else numberJobs > 0
  endif

# Create a Variable

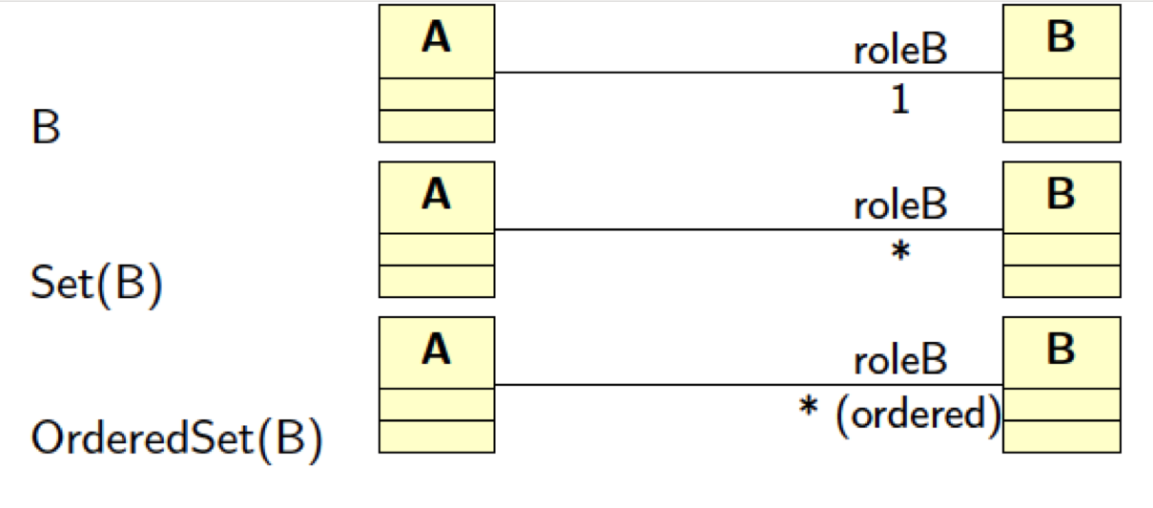
## example

- An unemployed person has no job. Otherwise, he has at least one job.
- The name of one person is the concatenation of first name and last name

- context Person
  inv: let numberJobs: Integer=self.job→size() in
  if isUnemployed then numberJobs=0
  else numberJobs > 0
  endif
- context Person
  def: name: String=self.firstName.concat(' ').
  concat(lastName)

# Collection

## Conversion of an association to a type within OCL



# Operation on Collection

## Syntax

- For a collection, $\rightarrow$ is used to apply some operation on it: nameCollection$\rightarrow$ operation()
- Recall: the dot is used for the access of a property of an object

## Some examples

- size(): Integer (return the number of elements in the collection)
- includes(object: T): Boolean (return true if object is included in the collection)
- excludes(object: T): Boolean (return true if object is not in the collection)

# Operation on Collection

## Some examples

- count(object: T): Integer (return the number of object)
- isEmpty(): Boolean (return true if the collection is empty)
- notEmpty(): Boolean (return true if the collection is not empty)
- includesAll(c: Collection(T)): Boolean (return true if the collection contains all elements of c)
- excludesAll(c: Collection(T)): Boolean (return true if the collection does not contains any element of c)
- sum(): T (return the sum of all elements in the collection)
- union(set: Set(T)): Set(T) (return the union of self with set)

# Operation on Collection

## Some examples

- any(exp: OclExpression): Type (return any element in self validating exp)
- =(set: Set(T)): Boolean (return true if self and set contain exactly the same elements)
- including(object: T): Set(T) (return a collection that contains all elements of self plus object)

# Operation on Collection

- One company has at least one employee

- context Company
  inv: self.employee $\rightarrow$ size() $>$ 0
  inv: self.employee $\rightarrow$ notEmpty()

# Operation on Collection

- One company has at least one employee
- The manager of a company is also an employee

- context Company
  inv: self.employee $\rightarrow$ size() $>$ 0
  inv: self.employee $\rightarrow$ notEmpty()
- context Company
  inv: self.employee $\rightarrow$ includes(self.manager)

# Operation on Elements

## Syntax

- collection $\rightarrow$ operation (expression)
- collection $\rightarrow$ operation (v | expression-with-v)
- collection $\rightarrow$ operation (v: Type | expression-with-v)

The expression is applied on each element of the collection

# Operation on Elements

- select: generate a sub-collection that contains only the elements satisfying expression

Each company must have at least one employee that is more than 50 years old

- context Company:
  inv: self.employee $\rightarrow$ select(p: Person | p.age$>$ 50) $\rightarrow$ notEmpty()

# Operation on Elements

- select: generate a sub-collection that contains only the elements satisfying expression
- reject: generate a sub-collection that contains only the elements that does not satisfy expression

Each company must have at least one employee that is more than 50 years old

- context Company:
  inv: self.employee $\rightarrow$ select(p: Person | p.age$>$ 50) $\rightarrow$ notEmpty()
- context Company:
  inv: self.employee $\rightarrow$ reject(p: Person | p.age$<=$ 50) $\rightarrow$ notEmpty()

# Operation on Elements

- forAll: return true if the expression is true for each element

Each company must have at least one employee that is more than 50 years old

- context Company:
  inv: self.employee $\rightarrow$ not (forAll(p: Person | p.age$<=$ 50))

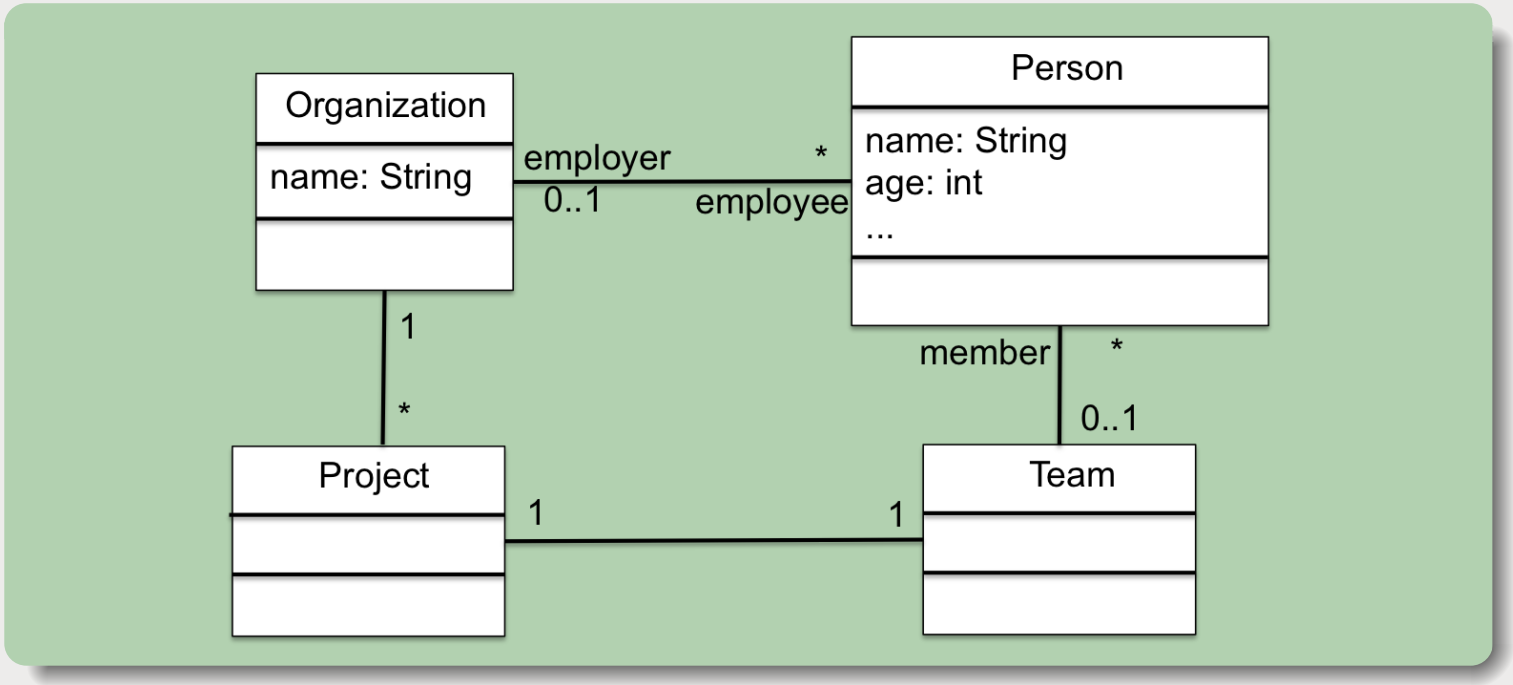
# Operation on Elements

- forAll: return true if the expression is true for each element
- exists: return true if the expression is true for at least one element

Each company must have at least one employee that is more than 50 years old

- context Company:
  inv: self.employee $\rightarrow$ not (forAll(p: Person | p.age$<=$ 50))
- context Company:
  inv: self.employee $\rightarrow$ exists(p: Person | p.age$>$ 50)

# Operation on Elements




The employer of an employee that participates a team project is the organisation that possesses this project

- context Person:
  inv: (self.employer→size()=1 and self.team→size()=1)
  implies self.employer=self.team.project.organisation

# Operation on Elements

- collect: create a new collection, for which each element is the result of the expression

It is required to obtain the set of birthday dates for all employees

- context Company
  self.employee $\rightarrow$ collect (p: Person | p.birthdayDate)

Shorthand for collect

- context Company
  self.employee.birthdayDate

# Iterate Operation

- iterate: calculate an accumulator whose value is built up during the iteration of the collection.
- Collection $\rightarrow$ iterate (e: Type; acc: Type=initial expression | expression with e and acc)

The sum of ages of all children for a person

- context Person
  self.children $\rightarrow$ iterate (p: Person; acc: Integer=0 | acc=acc+p.age)

# Re-typing or Casting

- With o.oclAsType(T2), one re-types an object o of type T1 into another type T2
- Let type Super be a super type of type Sub
- Allows one to use a property of an object defined on a subtype of the currently known type of the object
  context Super
  inv: self.oclAsType(Sub).p (accesses the p property defined in Sub, valid when actual type of self is Sub, otherwise, invalid)
- Can be used to access a property of a superclass
  context Sub
  inv: self.oclAsType(Super).p (accesses the p property defined in Super)

# Other functions

- oclIsTypeOf(t: Type): return true if the type of self and t are the same
- oclIsKindOf(t: Type): return true if t is a direct/indirect (supertype) type of self
- oclIsNew: used in a post-condition, return true if the object has been created during the operation

Example

context Person

inv: self.oclIsTypeOf(Person) –true

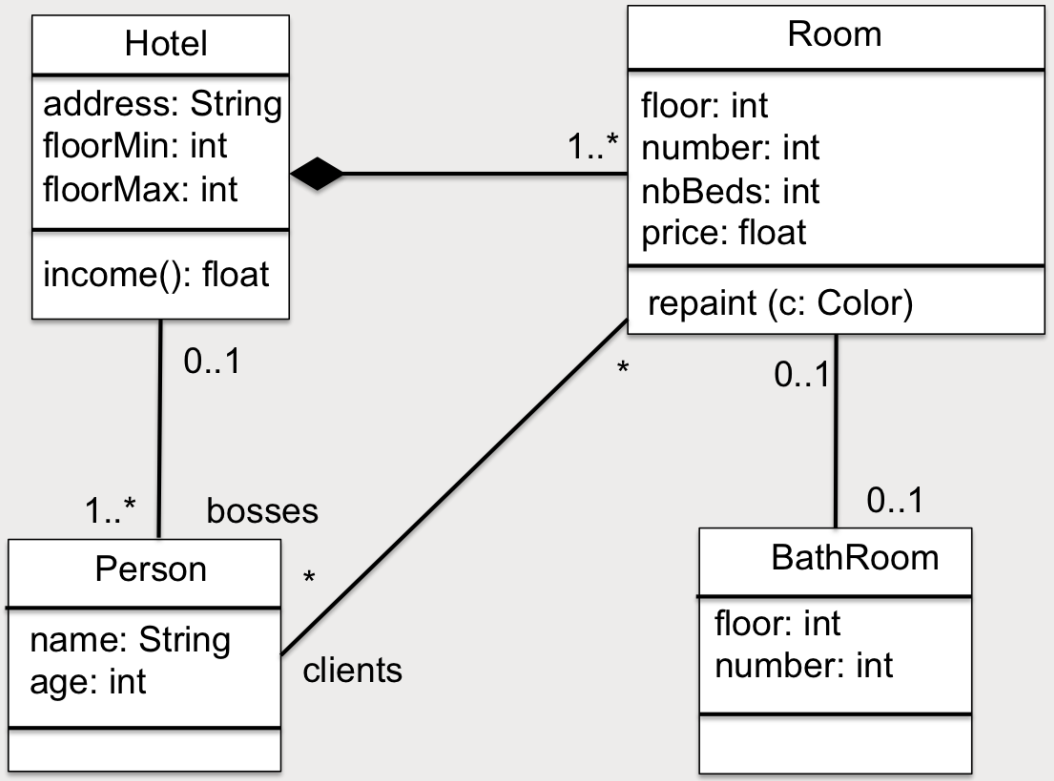
inv: self.oclIsTypeOf(Company) –false

# Class features

- Features of a class, not of its instances
- Either predefined or user-defined

- Predefined: allInstances holds on all types and returns the set of class instances
- There are at most 100 persons
  context Person
  inv: Person.allInstances() $\rightarrow$ size() $\leq$ 100
- A user-defined feature averageAge of class Person
  context Person
  Person.averageAge=(Person.allInstances() $\rightarrow$ collect (age) $\rightarrow$ sum()) / (Person.allInstances()$\rightarrow$ size())

# Example_1


Hotel
address: String
floorMin: int
floorMax: int
income(): float
Room
floor: int
number: int
nbBeds: int
price: float
repaint (c: Color)
1..*
0..1
1..*
bosses
*
*
0..1
0..1
Person
name: String
age: int
clients
BathRoom
floor: int
number: int

# OCL for example_1

A hotel never has a floor 13, because of superstition.

# OCL for example_1

A hotel never has a floor 13, because of superstition.

- context Room
  inv: self.floor <> 13
- context BathRoom
  inv: self.floor <> 13

# OCL for example_1

The number of clients for each room must be smaller or equal to the number of beds in the rented room. The children under 4 are not "taken into account" in this calculation rule (condition : maximum of one child under 4 per room).

# OCL for example_1

The number of clients for each room must be smaller or equal to the number of beds in the rented room. The children under 4 are not "taken into account" in this calculation rule (condition : maximum of one child under 4 per room).

- context Room
  inv: clients→size()≤nbBeds or
  (clients → size()=nbBeds+1 and clients→exists(p: Person| p.age<4))

# OCL for example_1

Each floor owns at least one room except for floor 13.

# OCL for example_1

Each floor owns at least one room except for floor 13.

- context Hotel
  inv: Sequence{floorMin, ..., floorMax}→forAll (f: Integer | f<>13 implies self.room→ select(r: Room|r.floor=f)→ notEmpty())

# OCL for example_1

Rooms are on the first to the last floor.

# OCL for example_1

Rooms are on the first to the last floor.

- context Hotel
  inv: self.room$\rightarrow$ forAll(r: Room | r.floor$\leq$ self.floorMax
  and r.floor$\geq$ self.floorMin)

# OCL for example_1

A room can be repainted when it is not occupied. Once repainted, the cost of a room is 10% more.

# OCL for example_1

A room can be repainted when it is not occupied. Once repainted, the cost of a room is 10% more.

- context Room::repaint(c: Color)
  pre: clients $\rightarrow$ isEmpty()
  post: price=price@pre*1.1

# OCL for example_1

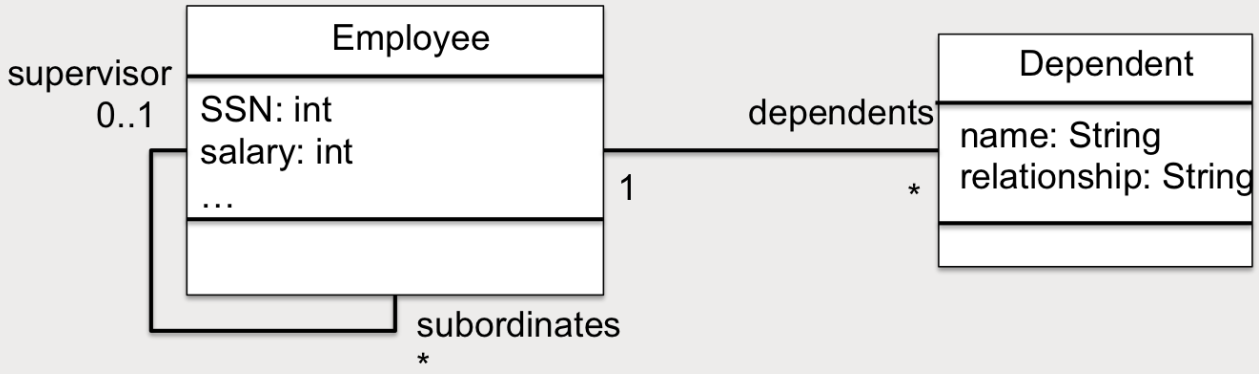
The hotel income is equal to the sum of prices for all rented rooms.

# OCL for example_1

The hotel income is equal to the sum of prices for all rented rooms.

- context Hotel::income():Real
  post: result=self.room $\rightarrow$ select(r: Room | r.clients $\rightarrow$ notEmpty())$\rightarrow$ collect(r:Room | r.price) $\rightarrow$ sum()

# Example_2


Employee
SSN: int
salary: int
…
supervisor
0..1
subordinates
*
dependents
1
*
Dependent
name: String
relationship: String

# OCL for example_2

- The salary of an employee cannot be greater than the salary of his/her superviser

  context Employee

  inv: self.supervisor$\rightarrow$ notEmpty() implies

  self.salary $<$ self.supervisor.salary

# OCL for example_2

- The salary of an employee cannot be greater than the salary of his/her superviser
  context Employee
  inv: self.supervisor$\rightarrow$ notEmpty() implies
  self.salary $<$ self.supervisor.salary
- The condition notEmpty must be tested since the multiplicity of the role is not mandatory

# OCL for example_2

- The SSN of employees is an identifier (or a key)
  context Employee
  inv: Employee.allInstances()→forAll(e1, e2 | e1 <> e2 implies e1.SSN <> e2.SSN)

# OCL for example_2

- The SSN of employees is an identifier (or a key)
  context Employee
  inv: Employee.allInstances()→forAll(e1, e2 | e1 <> e2 implies e1.SSN <> e2.SSN)
- The name and relationship of dependents is a partial identifier: they are unique among all dependents of an employee
  context Employee
  inv: self.dependents→ notEmpty() implies
  self.dependents → forAll(e1, e2 | e1 <> e2 implies e1.name <> e2.name or e1.relationship <> e2.relationship)

# OCL for example_2

- An employee cannot supervise him/herself

  context Employee

  inv: self.subordinates→excludes(self)