Lecture 12
OCL

Rogardt Heldal

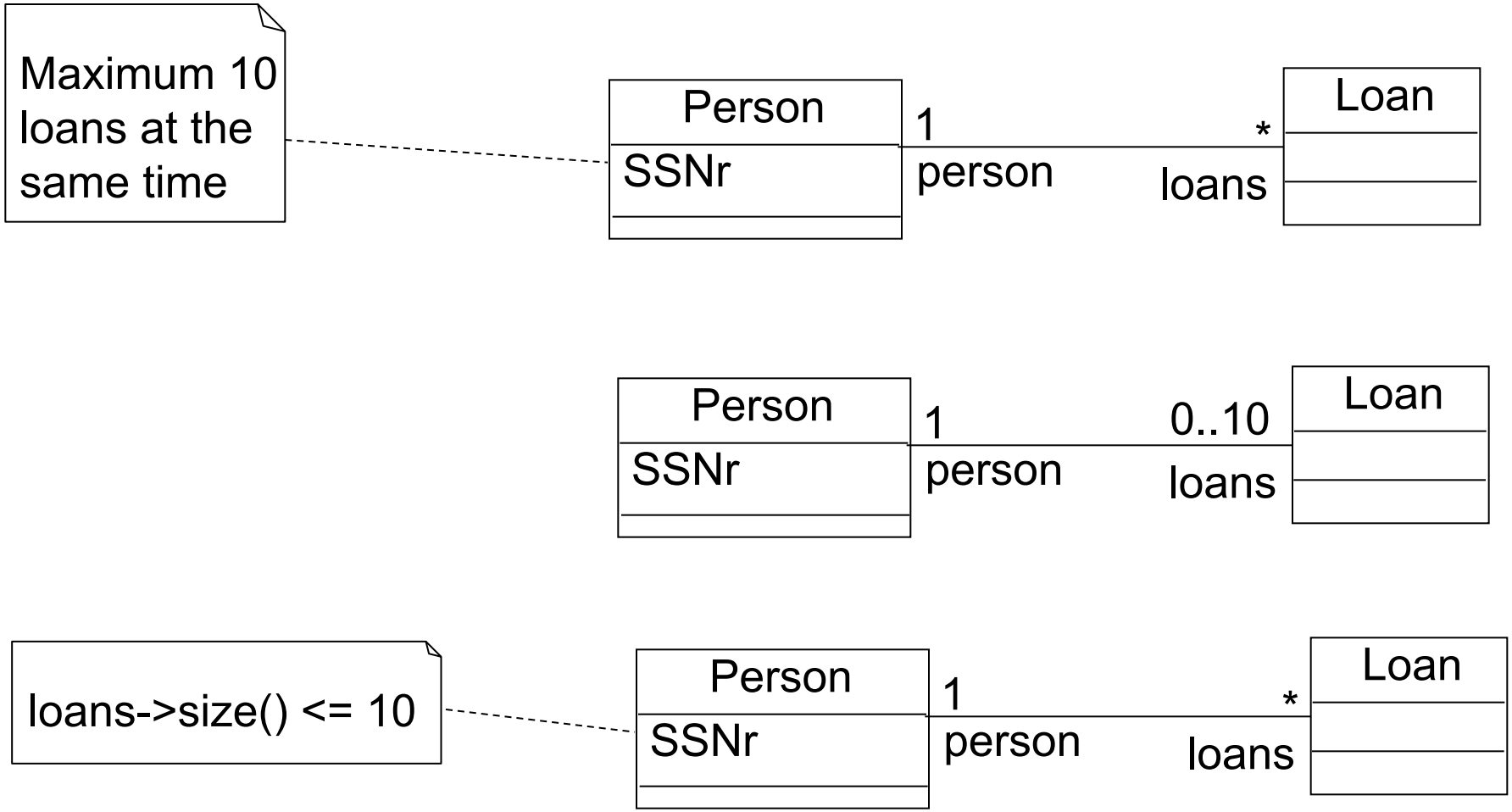
Constraints

- Invariant
- Pre- and post condition
- Guards

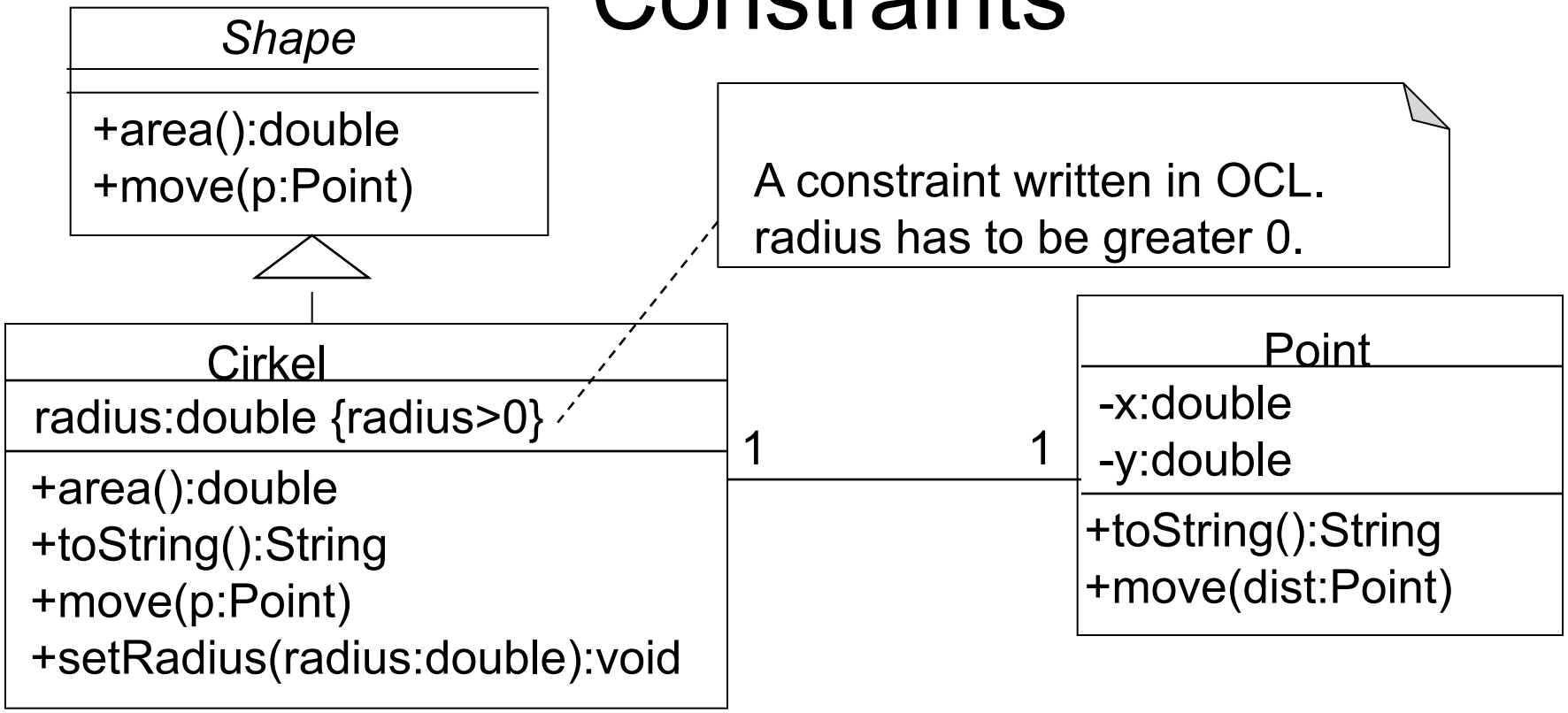
Common terms

- **Comment:** comment to an element, e.g., specification in natural language or a constraint
- **Constraint**: restriction of the usage of a UML element. Here, we consider constraints written in the formal language OCL

Motivation: Constraints



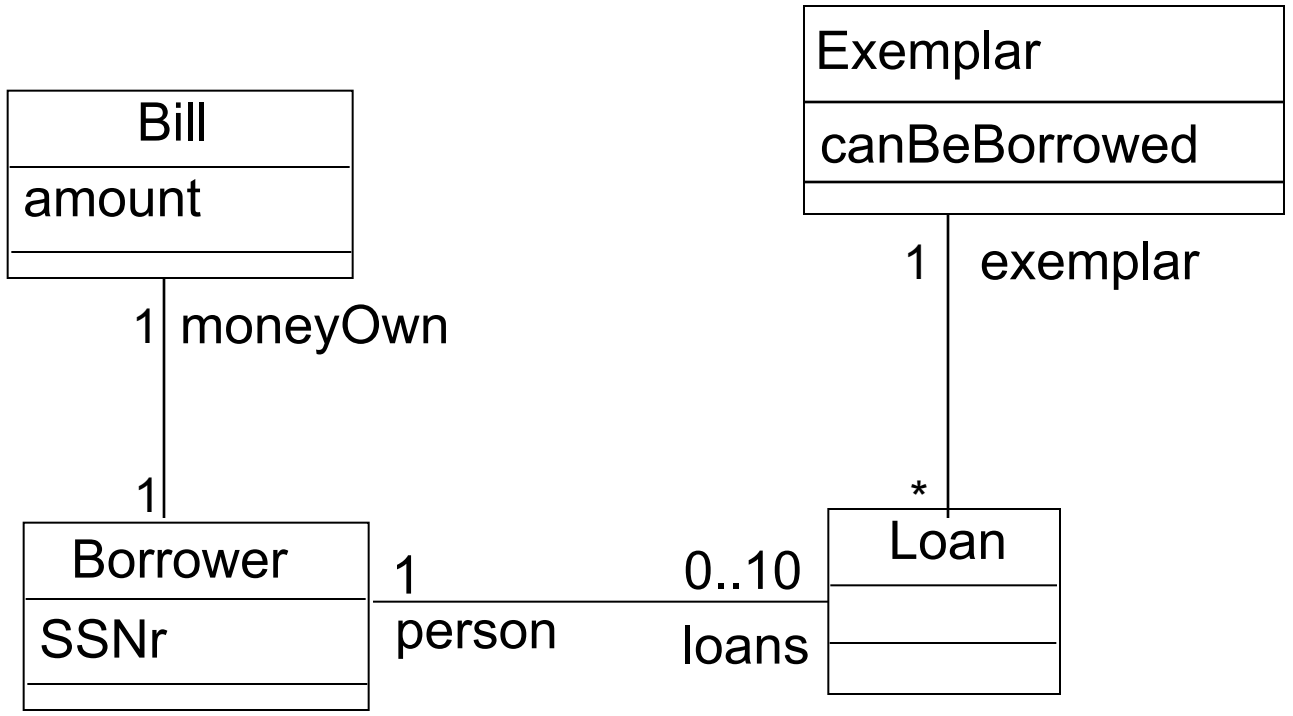
Constraints



Constraints can be written in natural language or using a formal language like OCL. Advantage of a formal language is that there are no ambiguities.

Motivation: Constraints

- The domain model cannot express that borrower can only have exemplars which can be borrowed, “canBeBorrowed”. Furthermore, that amount in class Bill always has to be >=0.



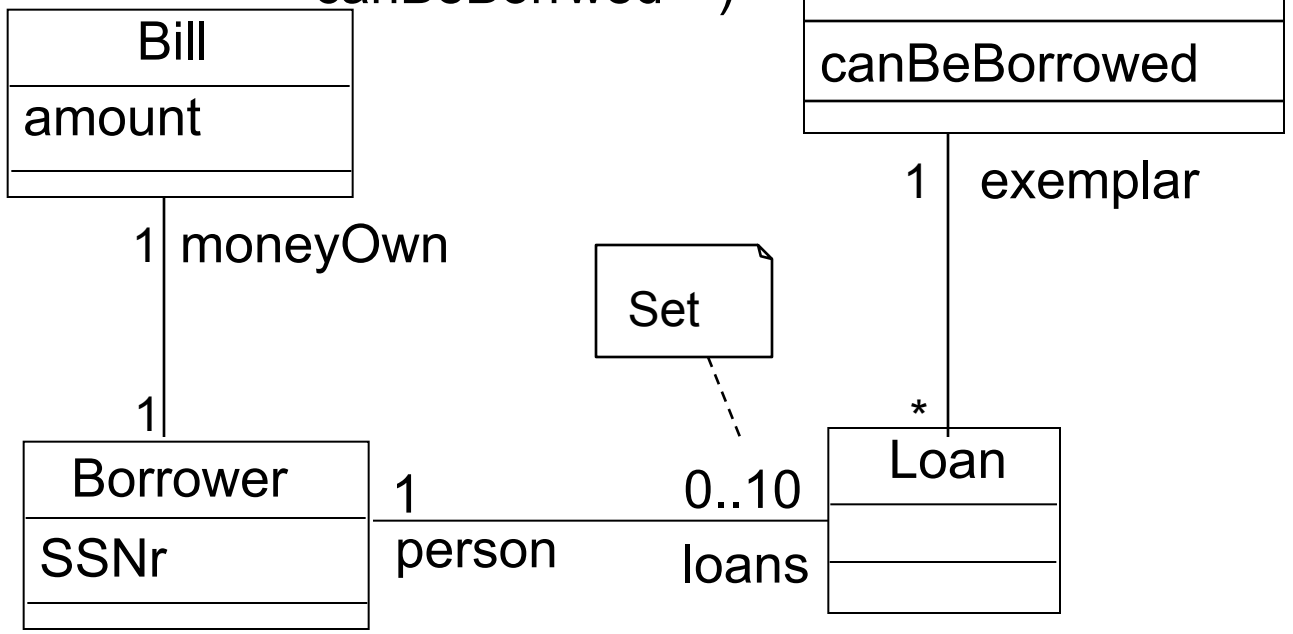
Motivation: Constraints

- Context Borrower inv:

```
loans
    -> forAll (
        exemplar
            .
        canBeBorrwed    )
```

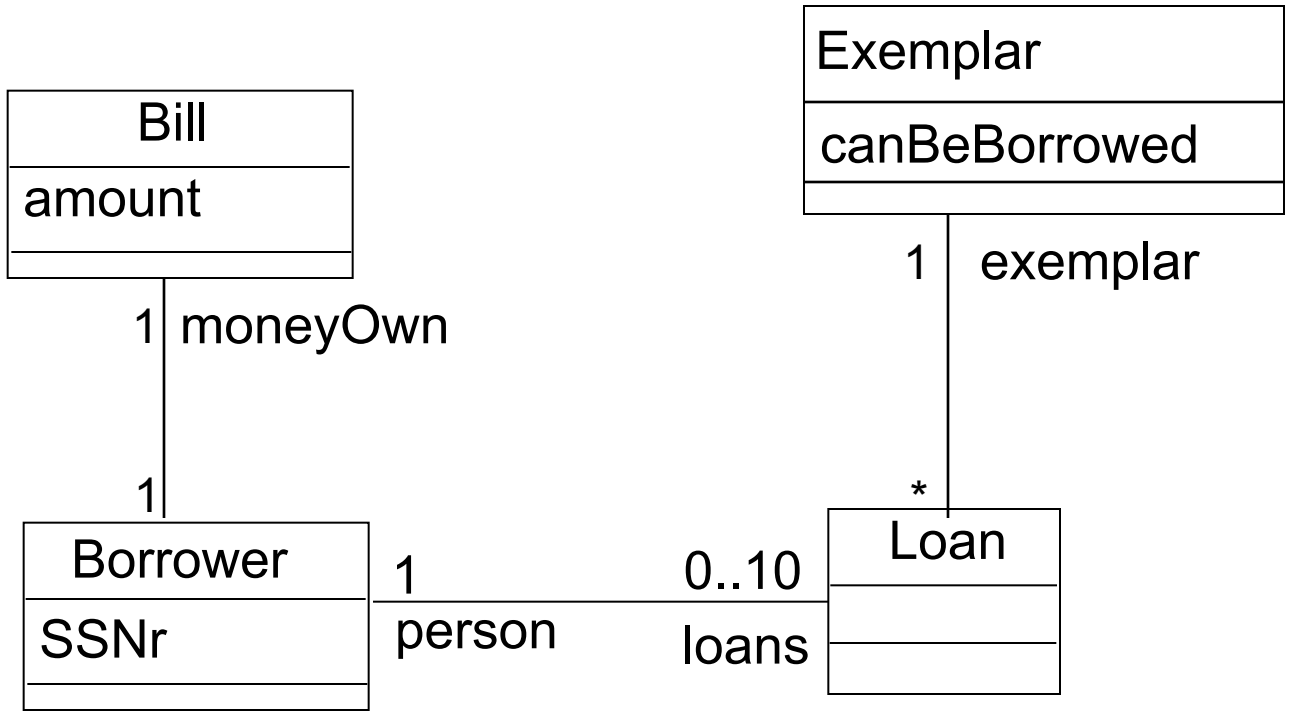
Used with collection operations

Used when referring to one thing



Motivation: Constraints

- context Bill inv:
  amount >=0

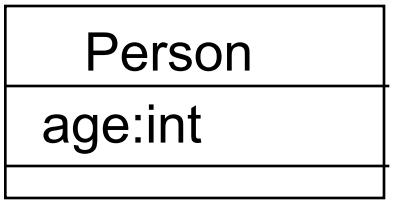


Constraints

- context Borrower inv:
  loans->forAll(exemplar.canBeBorrwed)

- context Bill inv:
  amount >= 0

Invariants



- A property that has to hold for all instances of a class/ interface/concept. For example:

- **context** Person **inv:** -- invariant of class Person
  age > 16

- **context** Person **inv:**
  self.age > 16 -- Variable **self** always points to the
  -- instance of Person itself.

Association Ends and Navigation

Navigation from one class to another, along an association, works mostly like accessing attributes. The role name of the association end is used for identifying the target.

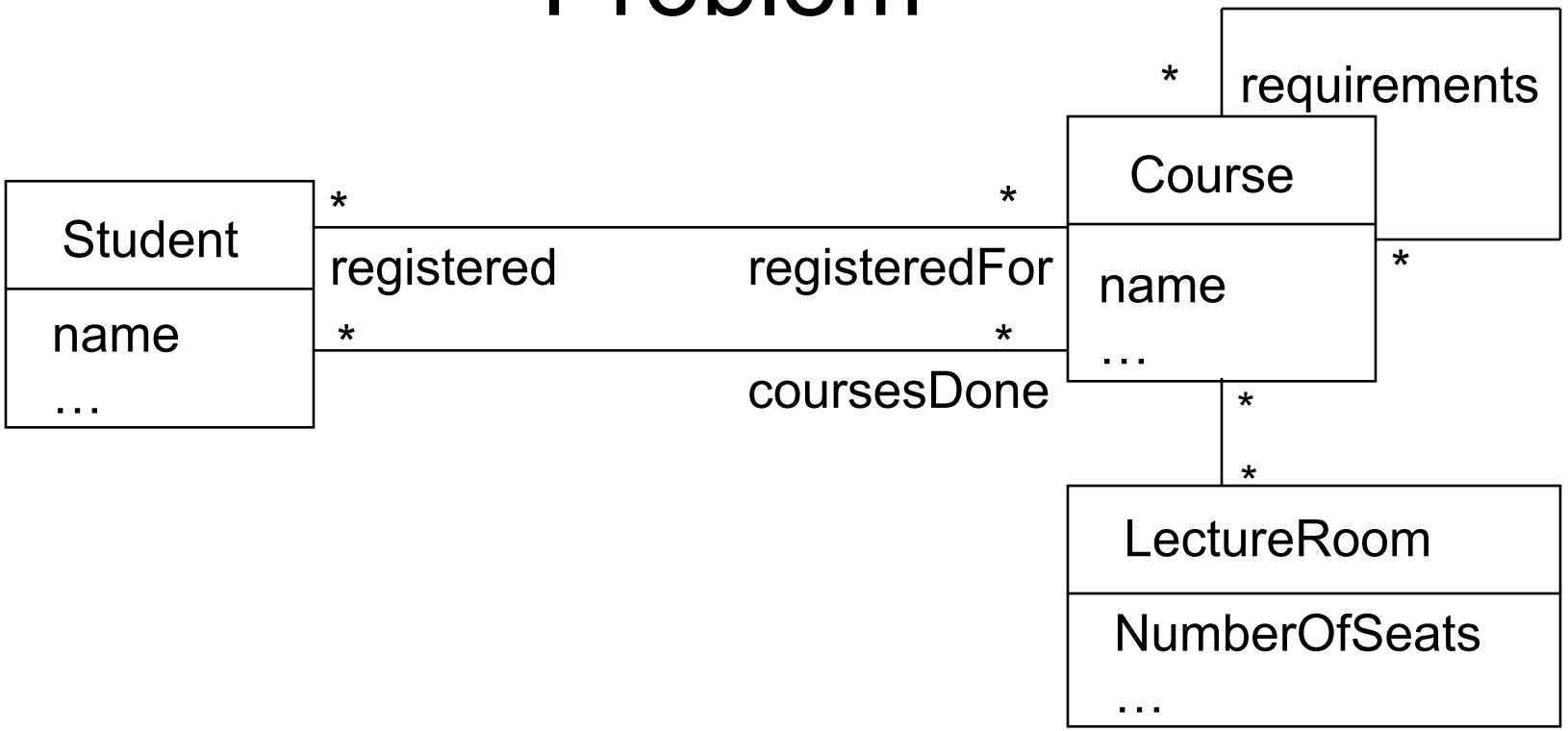
context Company **inv:**

employees->forAll(age > 16)

Choice of Context

- An invariant ”age > 16” in class **Person** ensures that there is no person younger than 17
- An invariant ” employees->forAll(age > 16)” in class **Company** ensures that no employee of a company is younger than 17. Other persons can be young …

Problem



Problem:

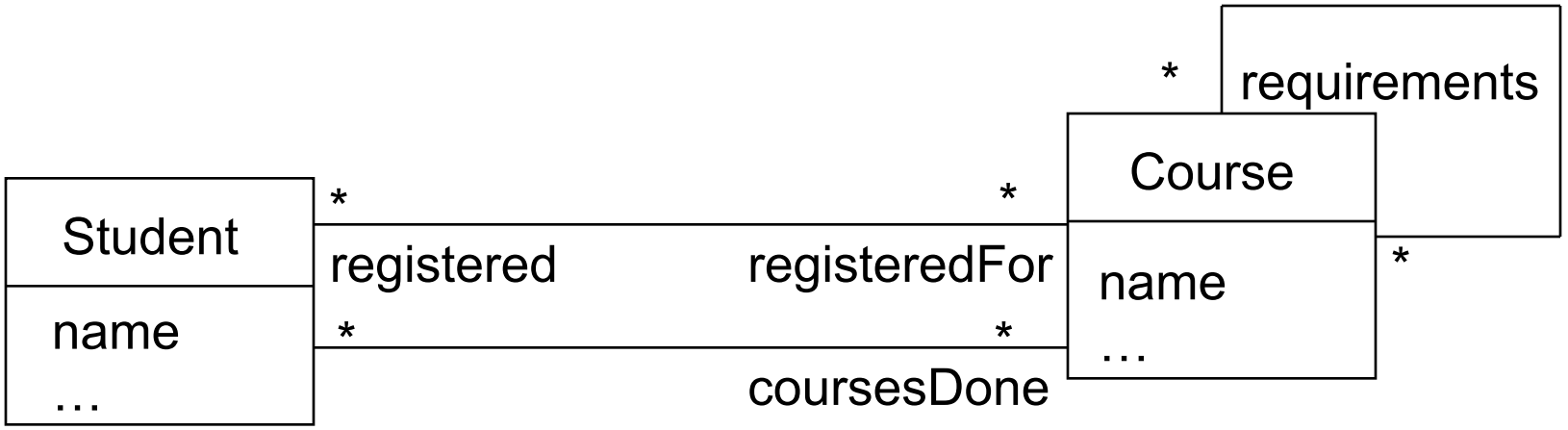
What invariants are natural to include here?

Solutions

- Problems:
  - A course can be requirement for itself.
  - The model does not constrain a student to not read a course without the correct prerequisites.
    - In reality this might happen. To let the system not permit this situation might be too strict.
  - Number of seats in a lecture room is not constrained in any way.
  - The model permits more students to register than there are seats in a lecture room.

- Comments: these are business rules which is hard or impossible to state using only domain models.

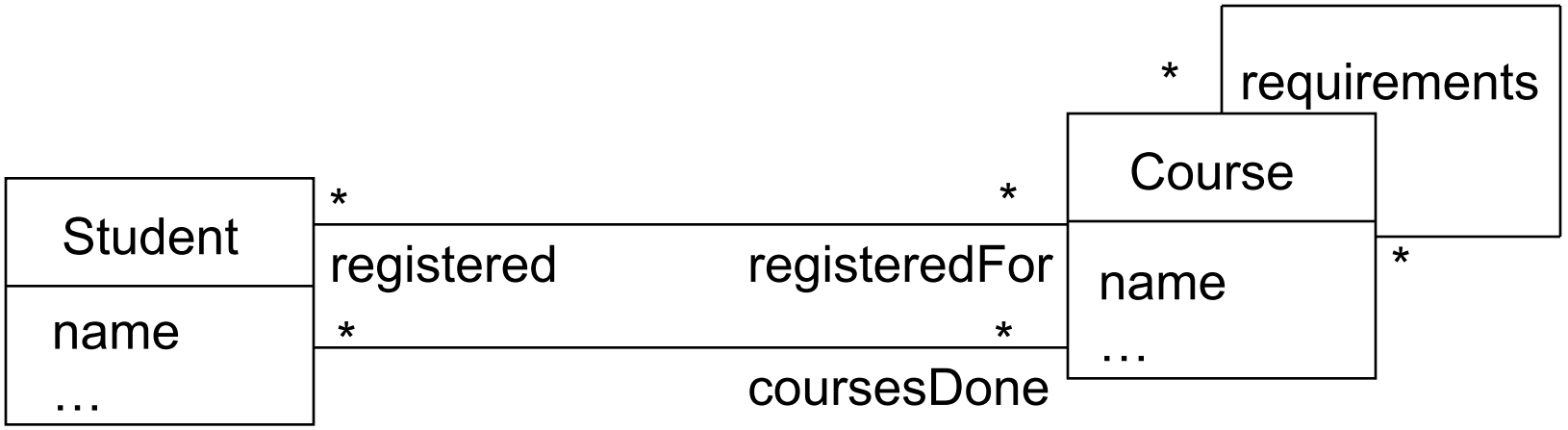
Example: Constraints



A course can be requirement for itself:

```
context Course inv:
  not requirements->includes(self)
```

Example: Constraints

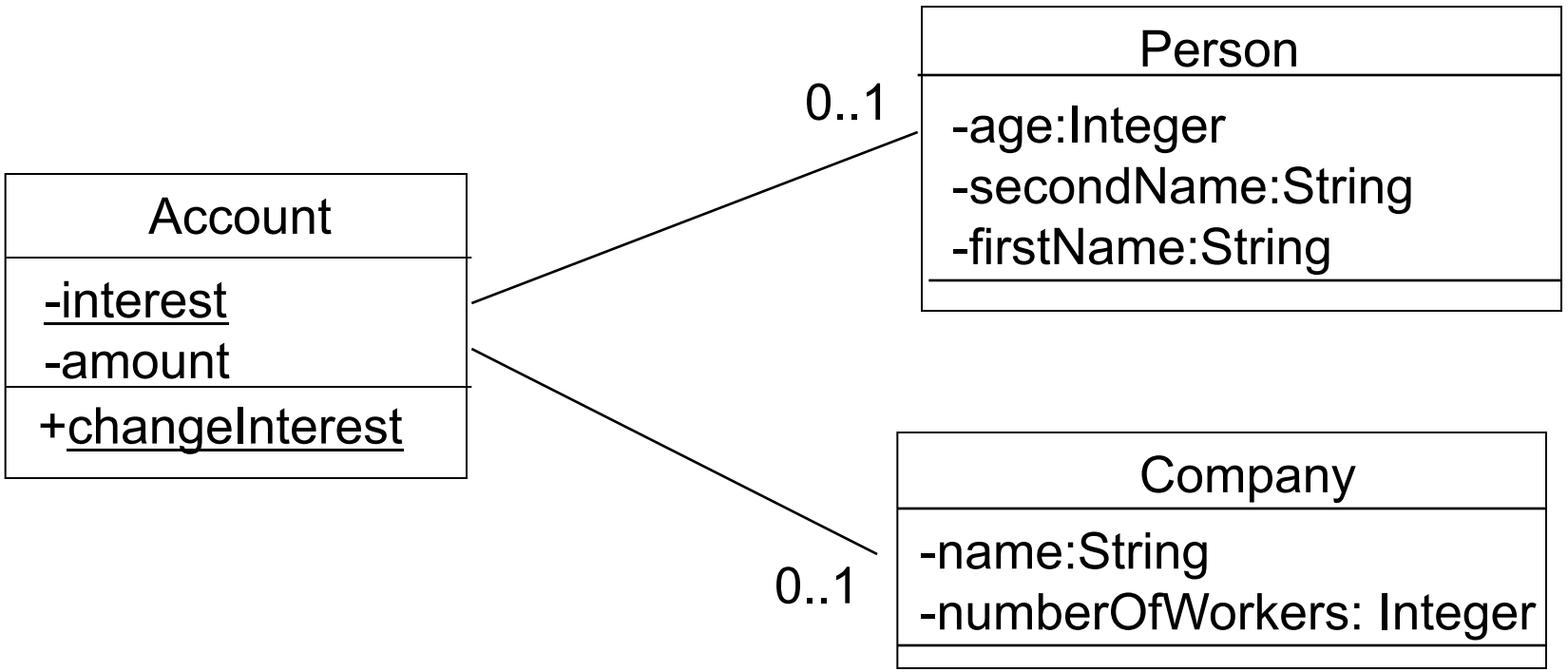


The model does not constrain a student to not read a course without the correct prerequisites:

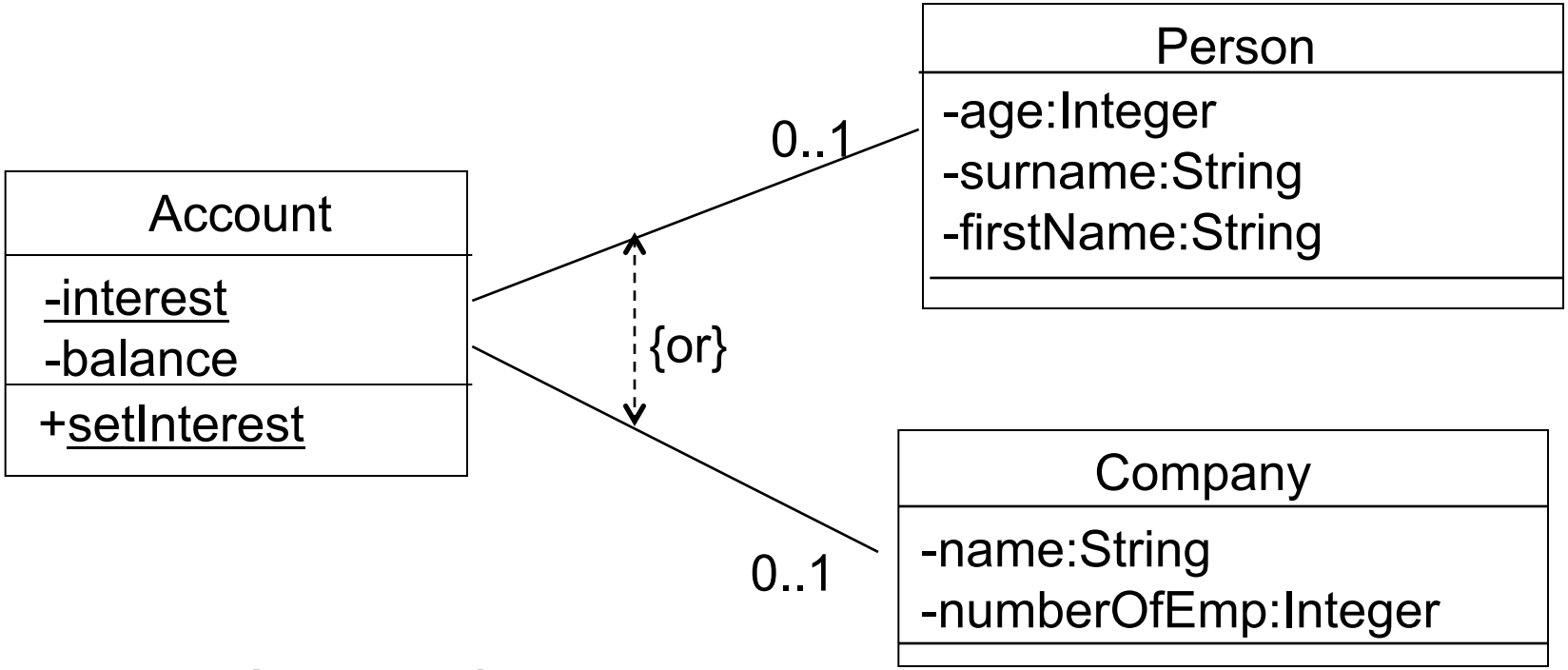
```
context Student inv:
  coursesDone -> includesAll(self.registeredFor
                             ->collect(requirements)->flatten())
```

Problem: UML

An **Account** can be associated with a **Person** or a **Company** but not with both. What is the problem with the diagram below?



Solution {or}

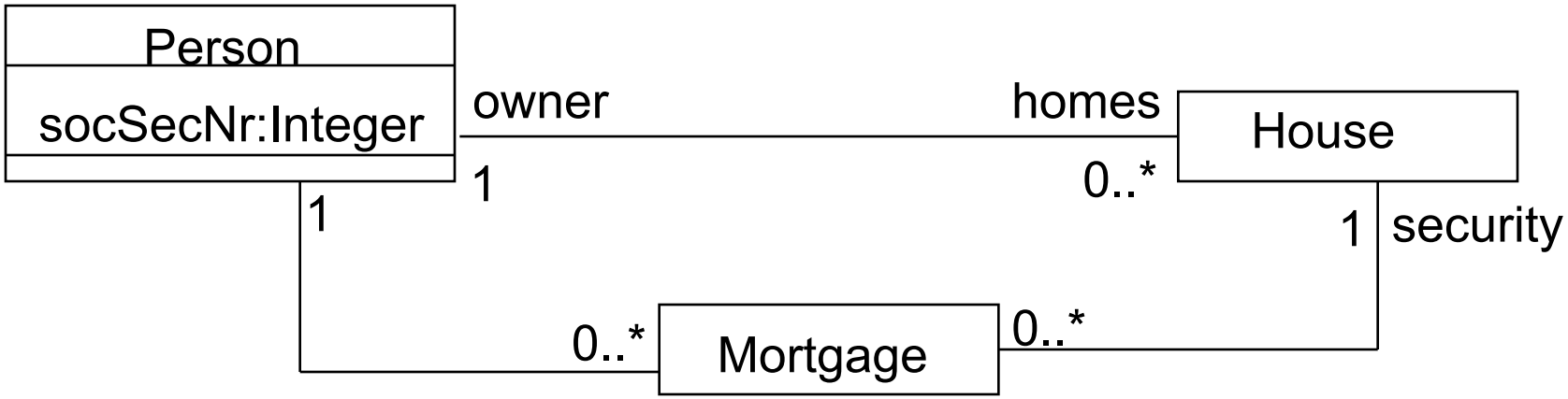


context Account inv:
person->intersection(company)->isEmpty

context Account inv:
self.person->isEmpty or self.company->isEmpty

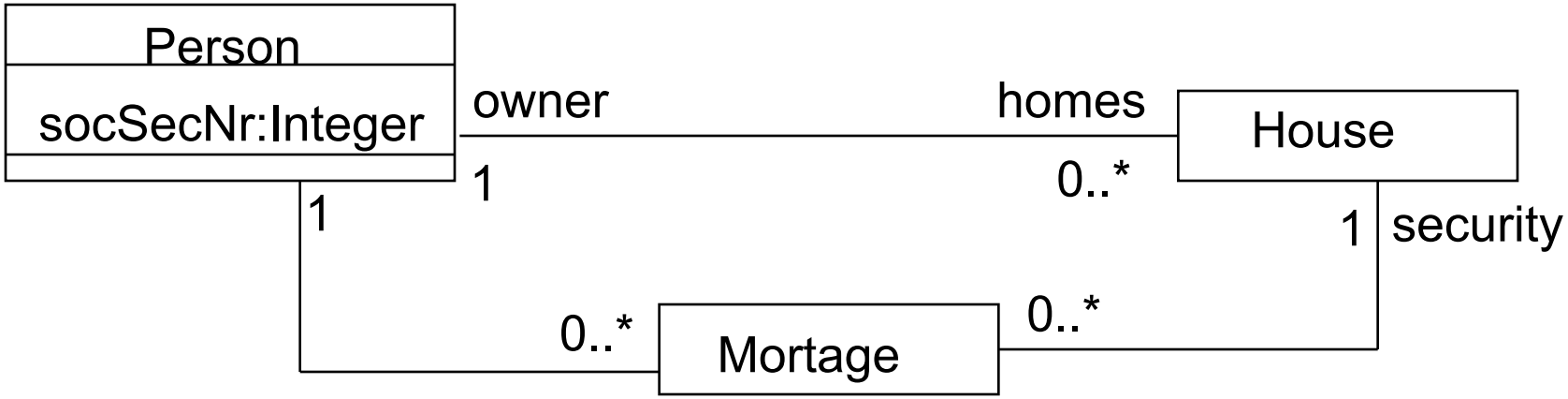
Problem

- When buying a house, one can take a mortgage with the security being another house one owns. What is the problem with the diagram below:

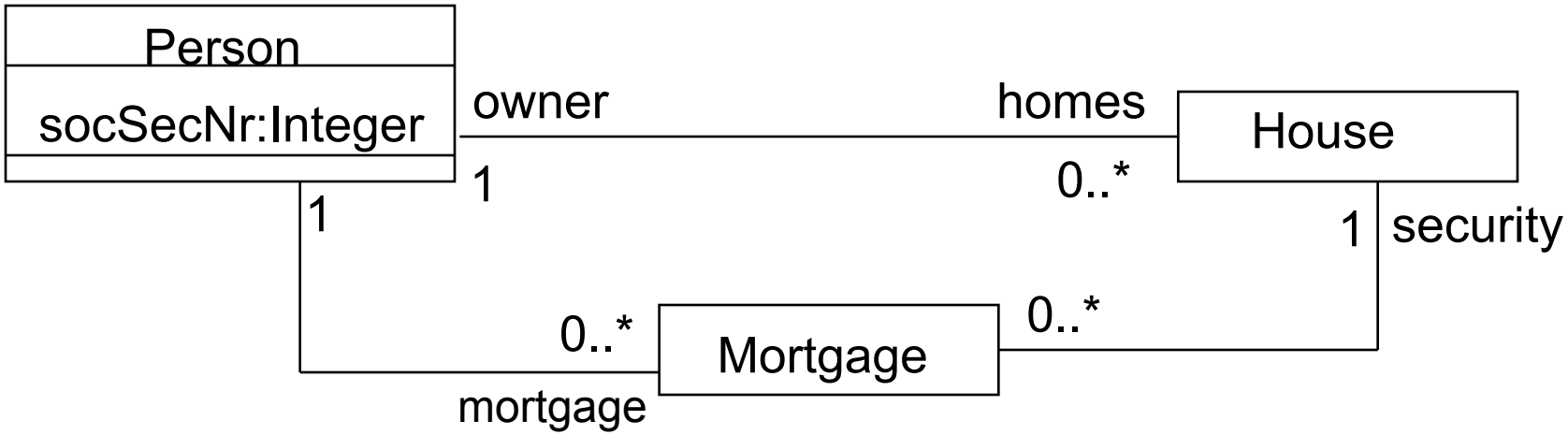


Solution

- If a house is used as security, then one has to own the house. This cannot be expressed in UML alone.
- It cannot be expressed that socSecNr is unique.



Solution

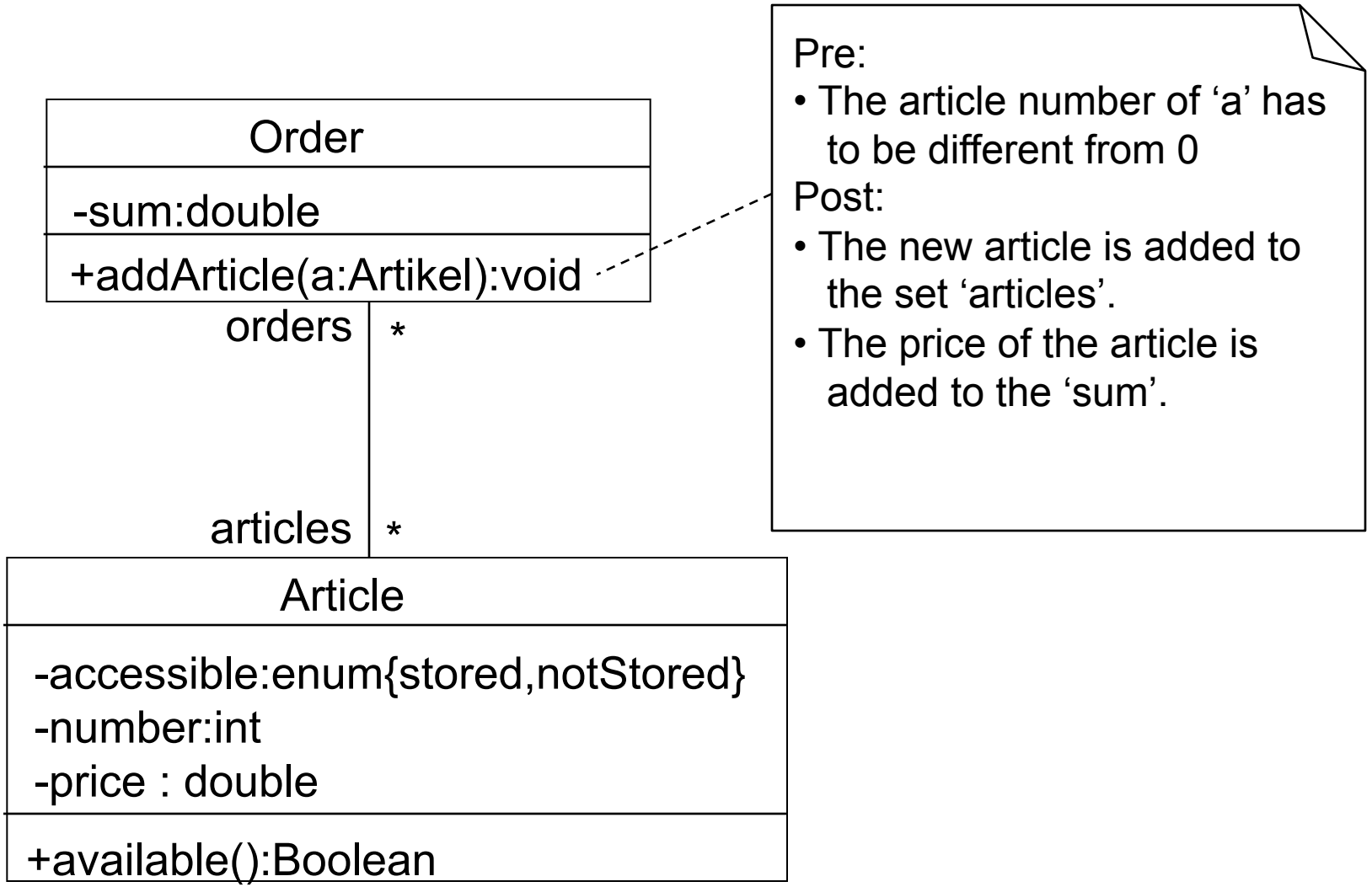



```
context Person inv:
   mortgage.security.owner = self
```

Class Diagrams

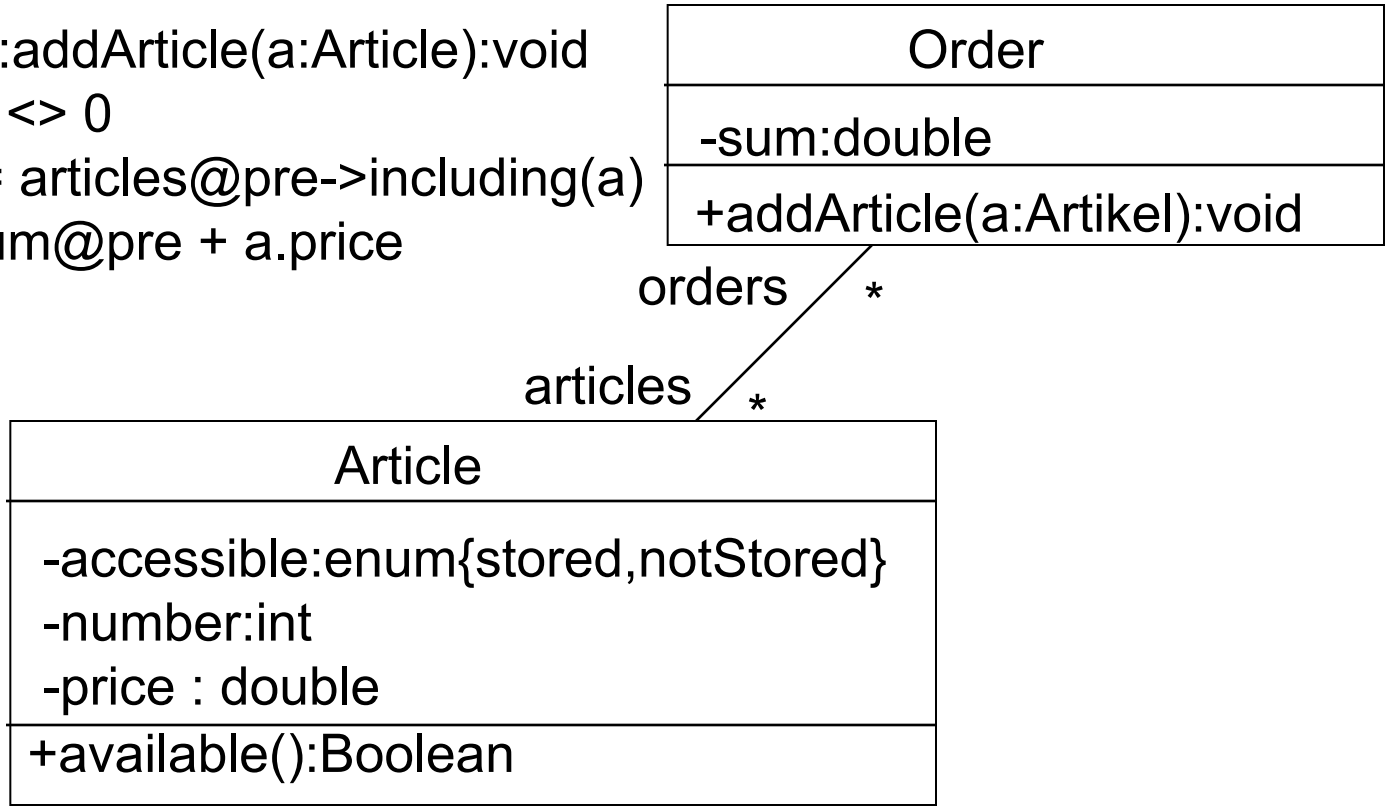
- As we have seen, it make sense to add formal OCL invariants to domain models, things which always have to hold in the model.
- In class diagrams one should also include OCL invariants in a similar way.
- Furthermore, in class diagram one can also include formal OCL pre- and post-conditions on operations, example following …

Pre- and Postconditions



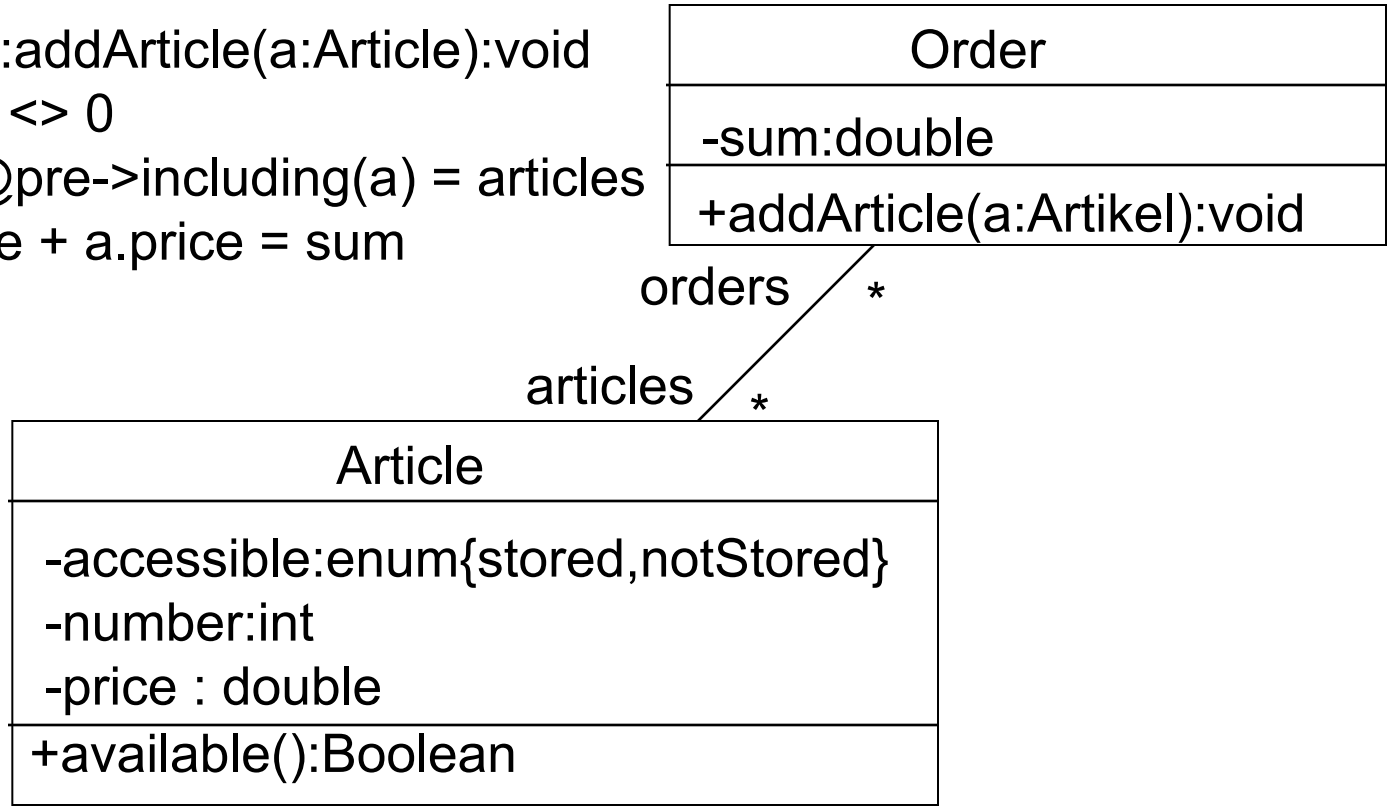
Pre- and Postconditions

```
context Order::addArticle(a:Article):void
pre: a.number <> 0
post: articles = articles@pre->including(a)
post: sum = sum@pre + a.price
```

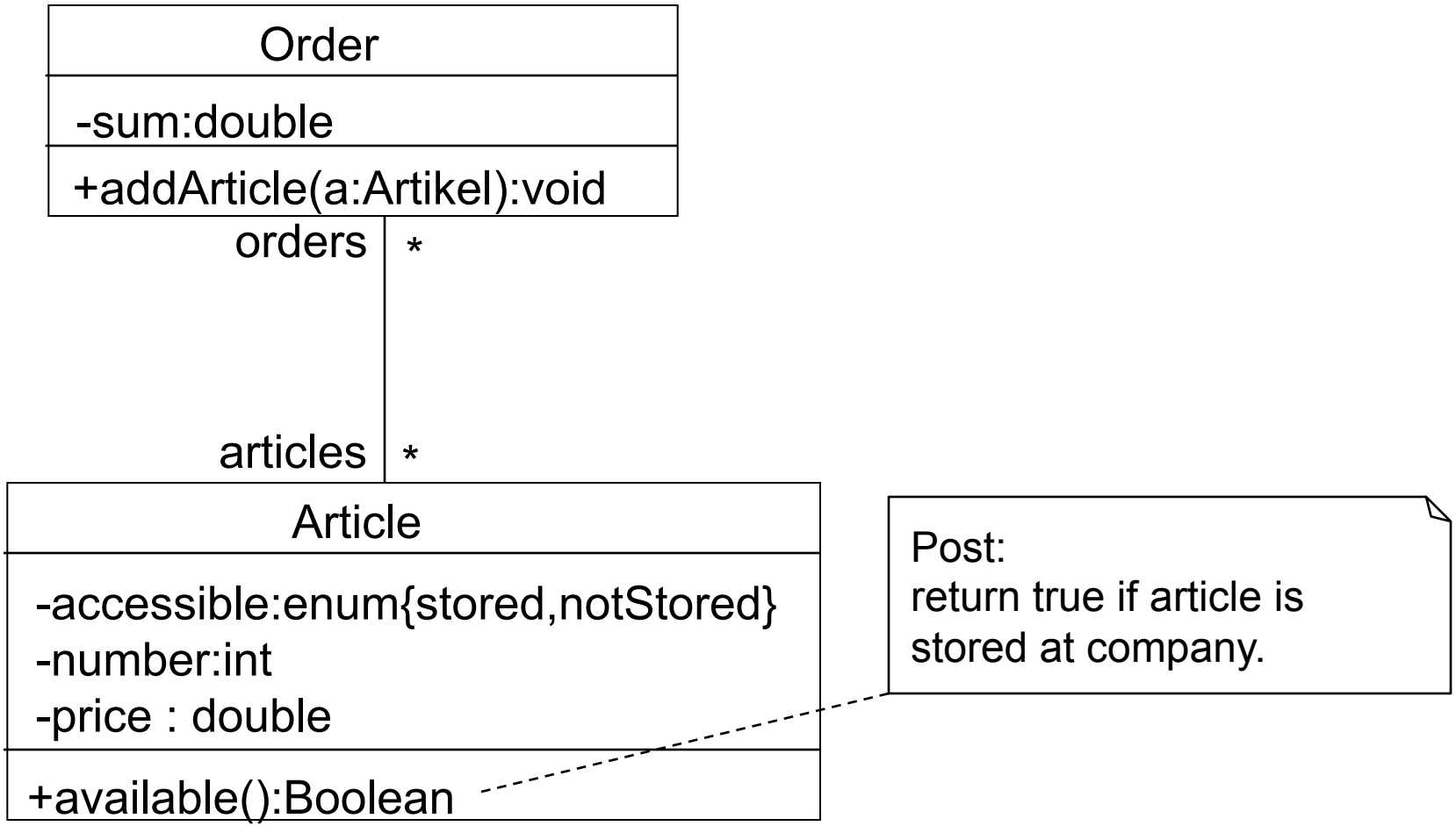


Pre- and Postconditions

```
context Order::addArticle(a:Article):void
pre: a.number <> 0
post: articles@pre->including(a) = articles
post: sum@pre + a.price = sum
```

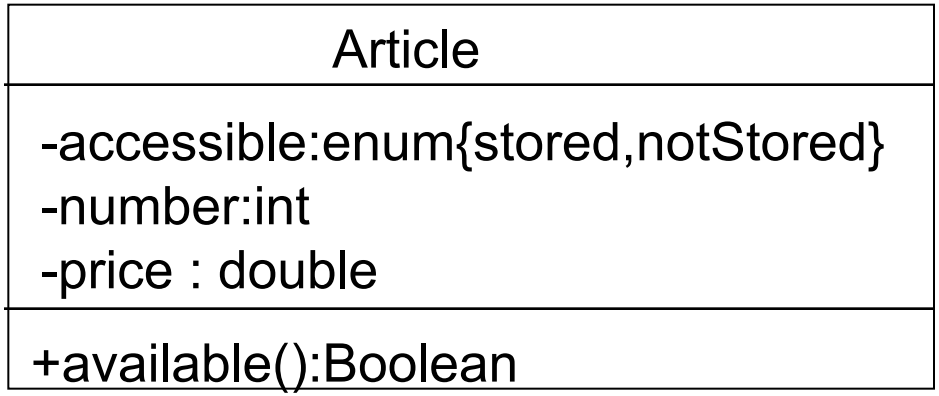


Pre- and Postconditions



Pre- and Postconditions

context Article::available():Boolean
post: result = (accessible = #stored)



Lot of small examples to show the power of OCL

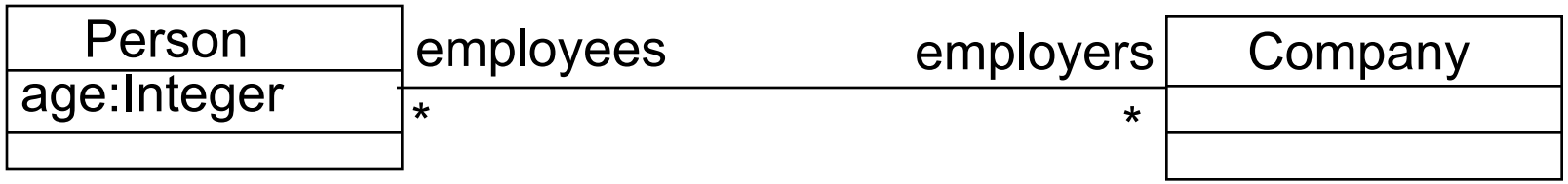
Set Operations

- Operations on collections (sets, bags, sequences) are always invoked with an arrow '->', e.g.

context Company **inv**:
numberOfEmp = employees -> size()

Example: select

```
context Company inv:
  self.employees->select(age > 45)->notEmpty
```



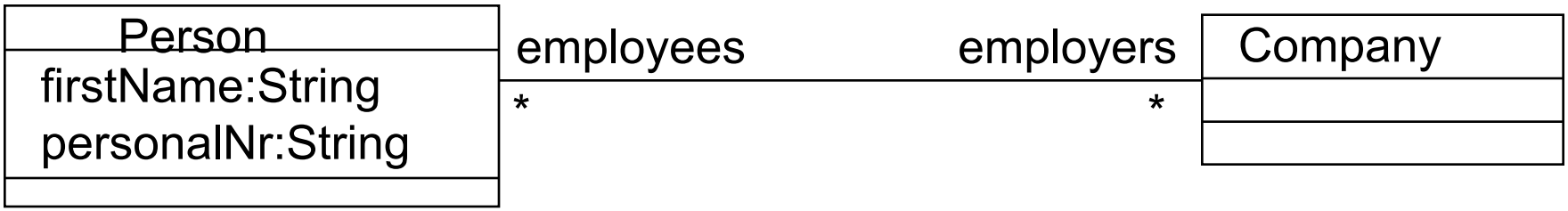
Example: collect

```
context Company:
   self.employees->collect(birthDate)              -- Bag(Date)

self.employees->collect(birthDate)->asSet
```

Example: ForAll

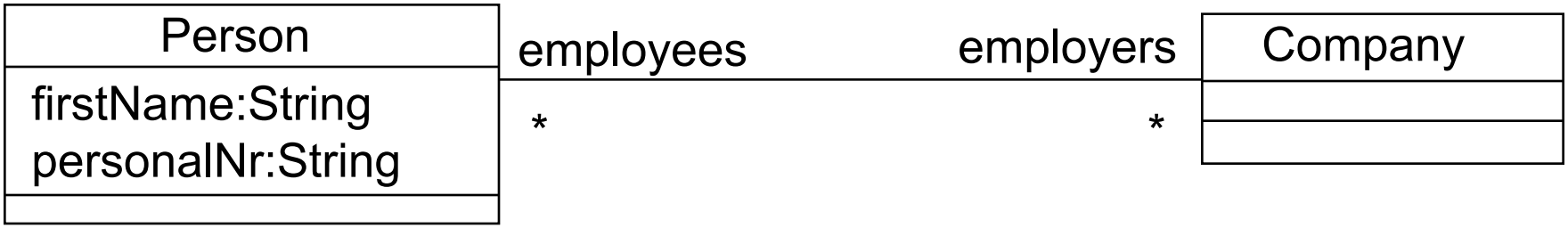
context Company inv:
    self.employees->forall(firstName = 'Jack')

context Company inv:
    self.employees->forall(e_1,e_2:Person |
        e_1 <> e_2 implies e_1.personalNr <> e_2.personalNr)



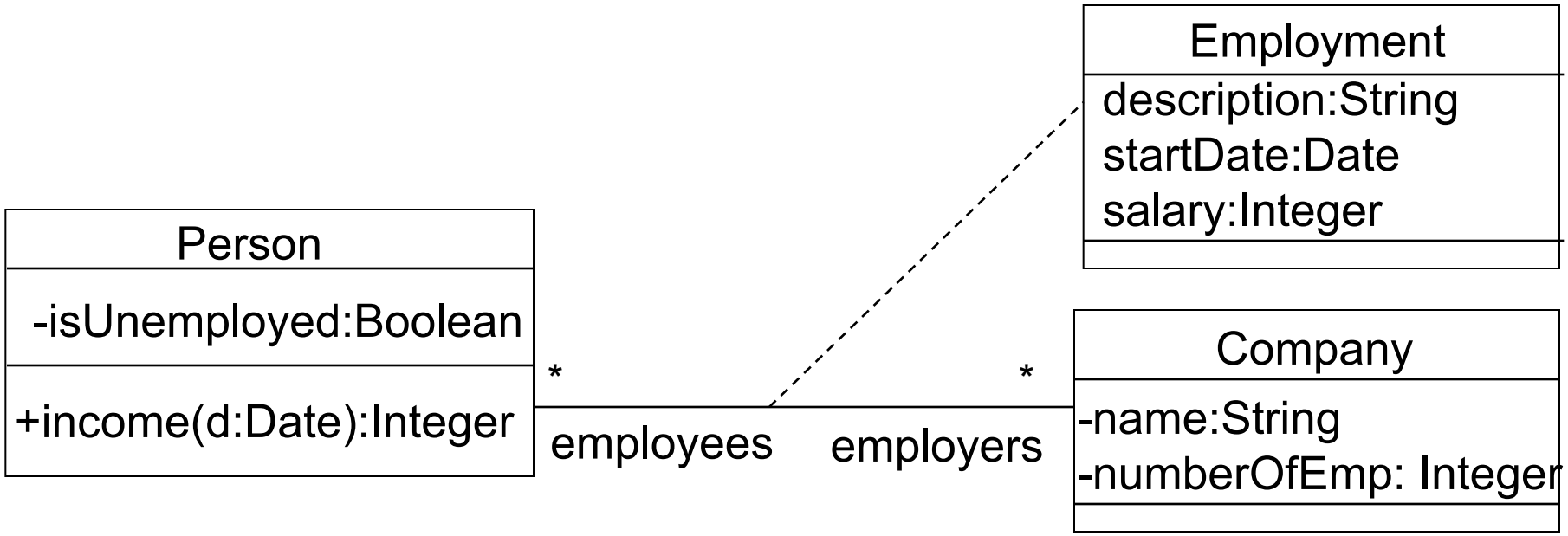
Example: Exists

```
context Company inv:
    self.employee->exists(firstName = 'Jack')
```

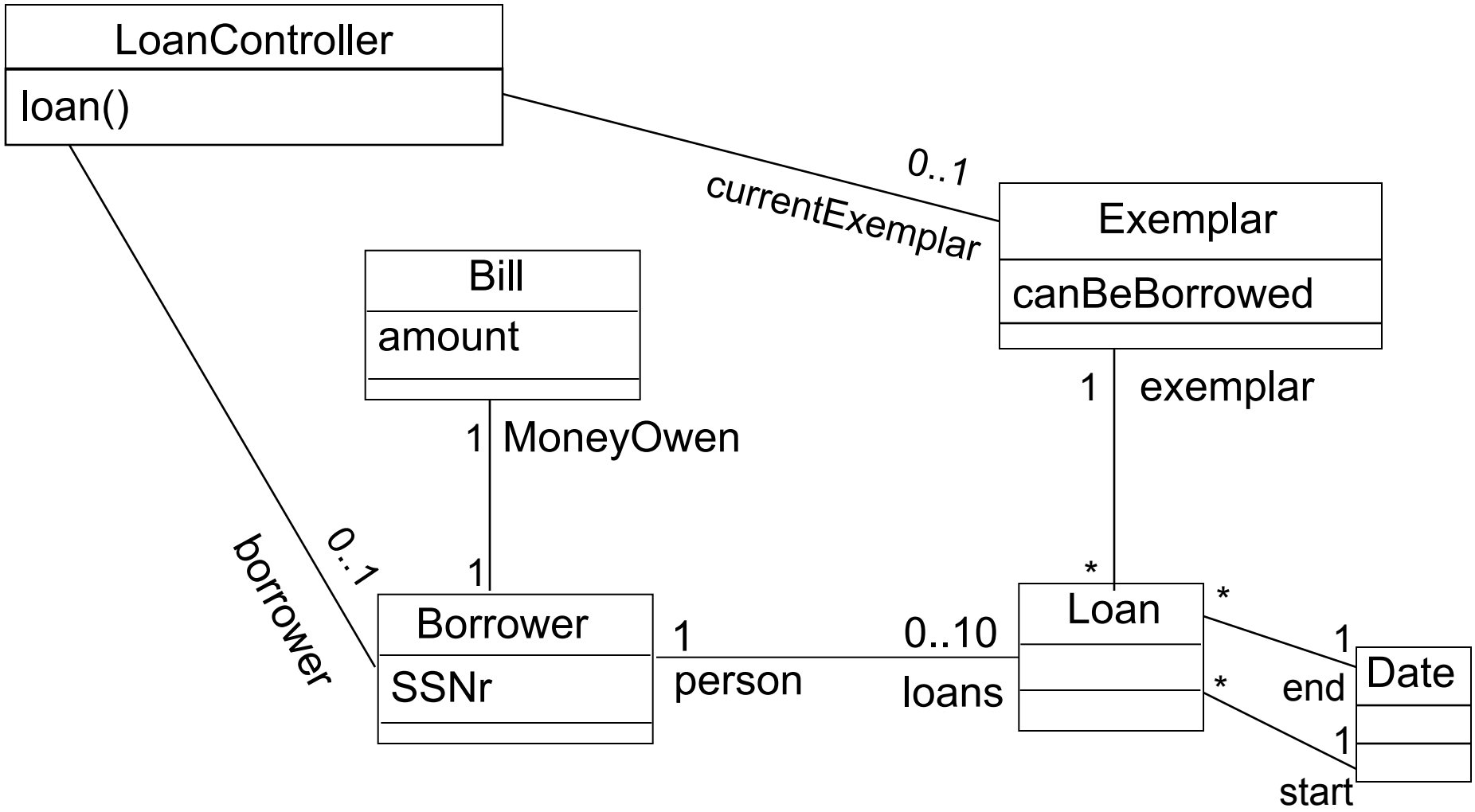


Let Expressions

```
context Person inv:
    let income : Integer = self.employment.salary->sum in
    if isUnemployed then income < 8000
                    else income >= 8000
    endif
```



Larger example

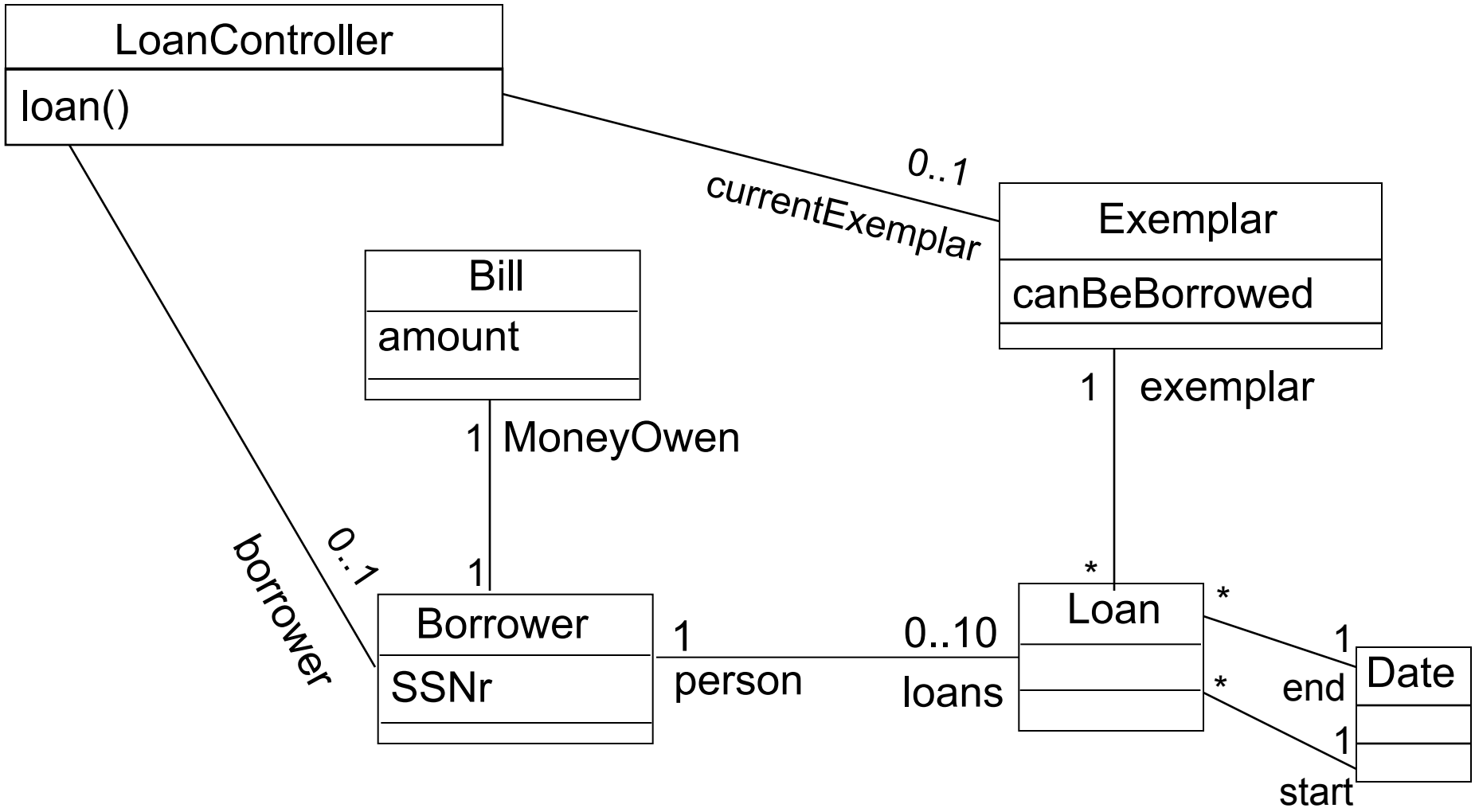

LoanController
loan()
0..1
currentExemplar
Exemplar
canBeBorrowed
Bill
amount
1
exemplar
1
MoneyOwen
0..1
borrower
1
*
Borrower
SSNr
1
person
0..10
loans
Loan
*
*
1
end
Date
1
start

Contract

- **Contract CO4: loan**
- Operation: loan(person)
- Reference: Use case “Loan Book”
- Description: An Exemplar is loaned by a person with the current date as starting date. The return date is one loan period (which depends on the book) later. If person is already loaning too many books, TooManyLoansException is thrown.

Contract

- **Contract CO4: loan() (cont'd)**
- Post-condition:
  - if person was having less than 10 loans and no bill unpaid and book is permitted to be borrowed then
    - new instance of Loan has been created and associated with the person taking the loan.
    - Loan has been associated with start and end date
      - start date is today's date
      - end date is start date plus the loan time
  - else
    - No new loan has been associated with the borrower.


LoanController
loan()
0..1
currentExemplar
Exemplar
canBeBorrowed
Bill
amount
1
exemplar
1
MoneyOwen
0..1
borrower
1
*
Borrower
SSNr
1
person
0..10
loans
Loan
*
*
1
end
Date
1
start

Contract

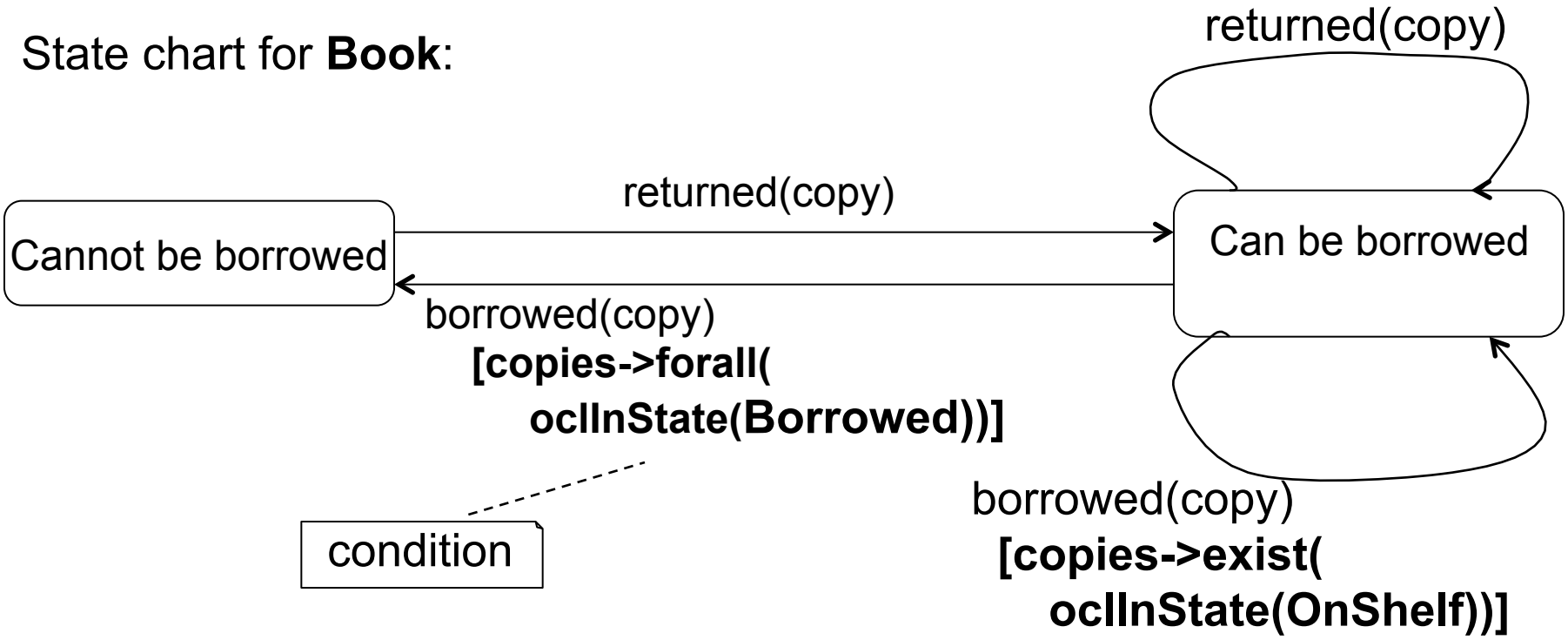
- contex LoanController::loan():Date post:

```
if  borrower.moneyOwen.amount=0 and
    borrower.loans->size() < 10 and
    currentExemplar.canBeBorrowed
then
        borrower.loans->
                exist(loanNew:Loan | loanNew.oclIsNew and
                loanNew.examplar = currentExemplar and
                borrower.loans =
                   borrower.loans@pre->including(loanNew) and
                loanNew.start.isToday() and
                31 = loanNew.end.minus(loanNew.start) and
                result = loanNew.end)
  else
        borrower.loans@pre =  borrower.loans
```

Conditions in state machines

Conditions

State chart for **Book**:



Constraints

- Invariant
- Pre- and post condition
- Guards/Conditions

Appendix

Object Contraint Language

- OCL is a formal declarative specification language, i.e., expressions of the language do not have side effects.
- Can be used for:
  - Specify invariants of classes and types
  - Describe pre- and postconditions of operations and methods
  - Write guards (e.g., for “opt” fragments in sequence diagrams)
  - …

Why constraints?

- First of all, writing constraints makes it necessary to understand a problem in depth; might, e.g., lead to discovering mistakes
- Constraints can be tested in program (dynamically, while program is executed)
- It can be proved that a program does never violate constraints (statically, before running the program)
- …
- A combination of the items above

Basic Data Types of OCL

Type:	Example:
Boolean	false,true
Integer	1,5,333
Real	3.23
String	’hej’
Set	{33,56,45},{’blue’,’green’}
Bag	{67,094,5,2},{13,7,7}
Sequence	{1..10},{3,7,67}

Basic Operations of OCL

- Integer : *, +, -, /, abs, mod ...
- Real: *, +, -, /, floor, ...
- Boolean: and, or, xor, not, if-then-else, implies, ...
- String: toUpper, concat, …
- Set: union, intersection, include, asSequence, asBag …
- Bag: …
- Sequence: first, last, at(i), ...

Infix-operators: +, -,*,/,<,>,<>,<=,>=,and,or,xor
'--' marks comments in OCL

Basic Operations of OCL

Example of OCL expressions:

```
3 + 5 * 111
13 + 12.9                -- implicit type conversion
2.mod(2)
```

Example of incorrect OCL expressions:

```
1 + 'hej'
 true + 1
```

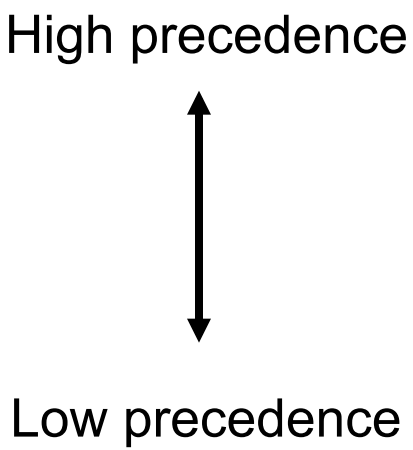
OCL Expressions and Constraints

- Only OCL expressions of type Boolean can be used as constraints! E.g.
  - age >= 0

- Not usable as constraints:
  - ’hej’
  - 3 + 5

Precedence of Operators

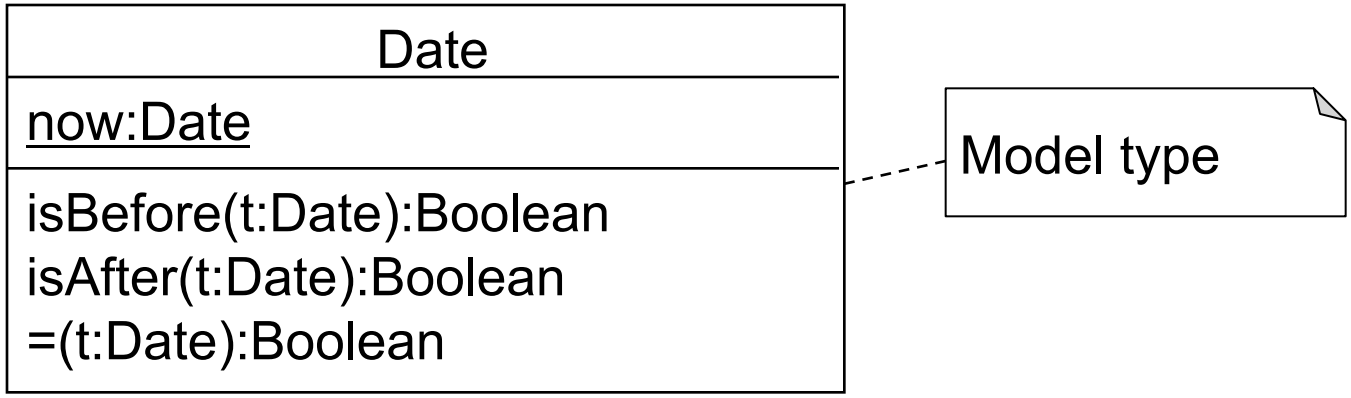
- **::**
- **@pre**
- **.** och **->, ^**
- **not** och **-** **-- unary**
- * och /
- **+** och **-**
- **if-then-else-endif**
- **<,>,<=** och **>= , =** och **<>**
- **and**, **or**, och **xor**
- **implies**



Grouping of operands can be controlled using parentheses

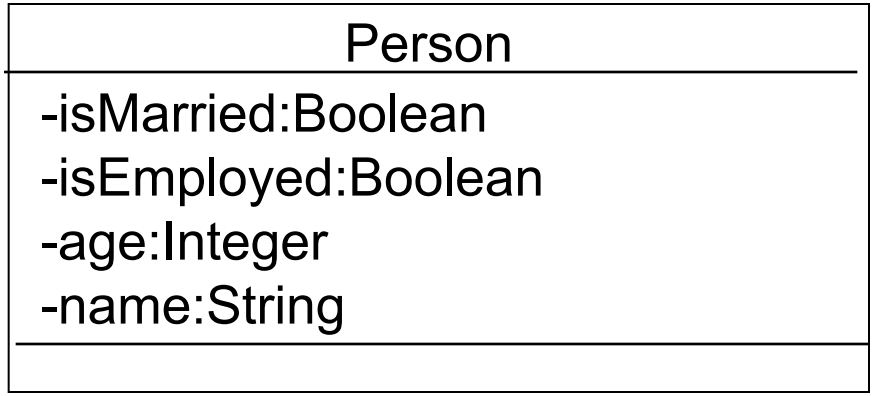
Model Types

- Classes, interfaces, enumerations or other types of a UML model can directly be used in OCL.



Attributes

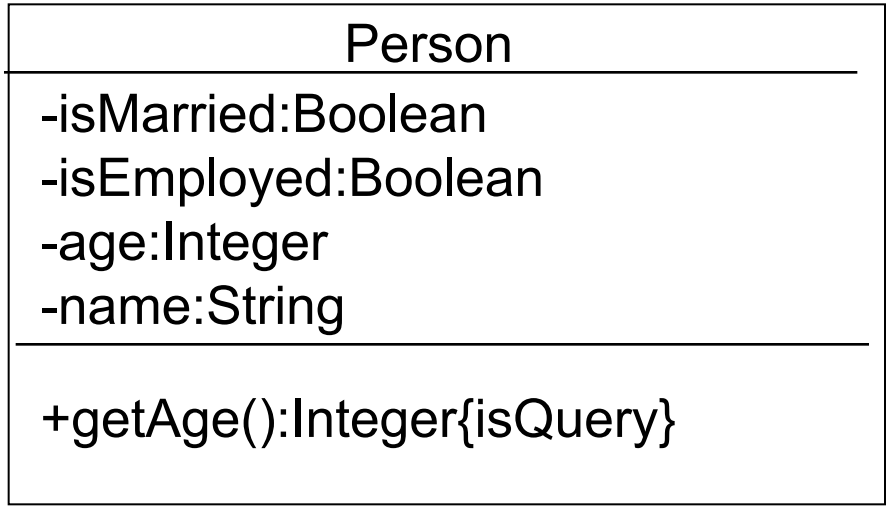
- Attributes of a UML class can be used in OCL expressions like in Java, e.g.,
  - age > 18
  - self.age > 18



Operations

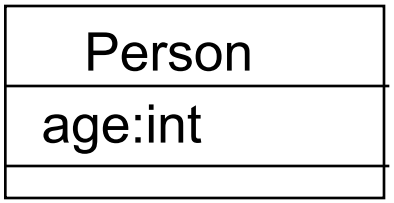
- Operations with the stereotype {isQuery} can be used in OCL expressions. Such operations must not have side effects

- **OCL expressions:**
  - getAge() >= 0
  - self.getAge() >= 0



- Class variables and class operations can be accessed by adding the class name:
  - Data.now

Invariants



- A property that has to hold for all instances of a class/ interface/concept. For example:

- **context** Person **inv:** -- invariant of class Person
  age > 16

- **context** Person **inv:**
  self.age > 16 -- Variable **self** always points to the
  -- instance of Person itself.

Association Ends and Navigation

Navigation from one class to another, along an association, works mostly like accessing attributes. The role name of the association end is used for identifying the target.

context Company **inv:**

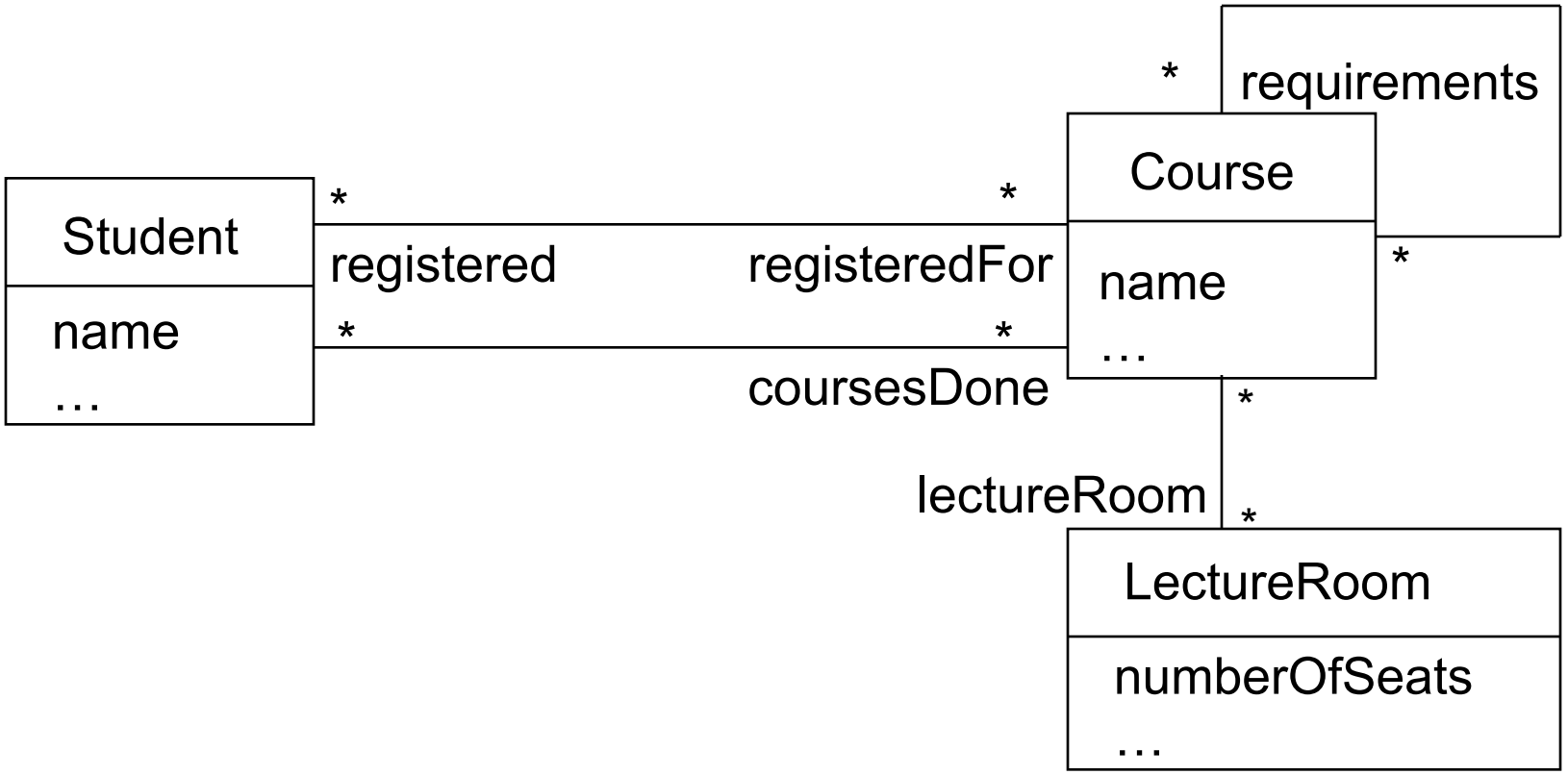
employees->forAll(age > 16)

Choice of Context

- An invariant ”age > 16” in class **Person** ensures that there is no person younger than 17
- An invariant ” employees->forAll(age > 16)” in class **Company** ensures that no employee of a company is younger than 17. Other persons can be young …

Problem

- Number of seats in a lecture room is always more than 10.



Solution

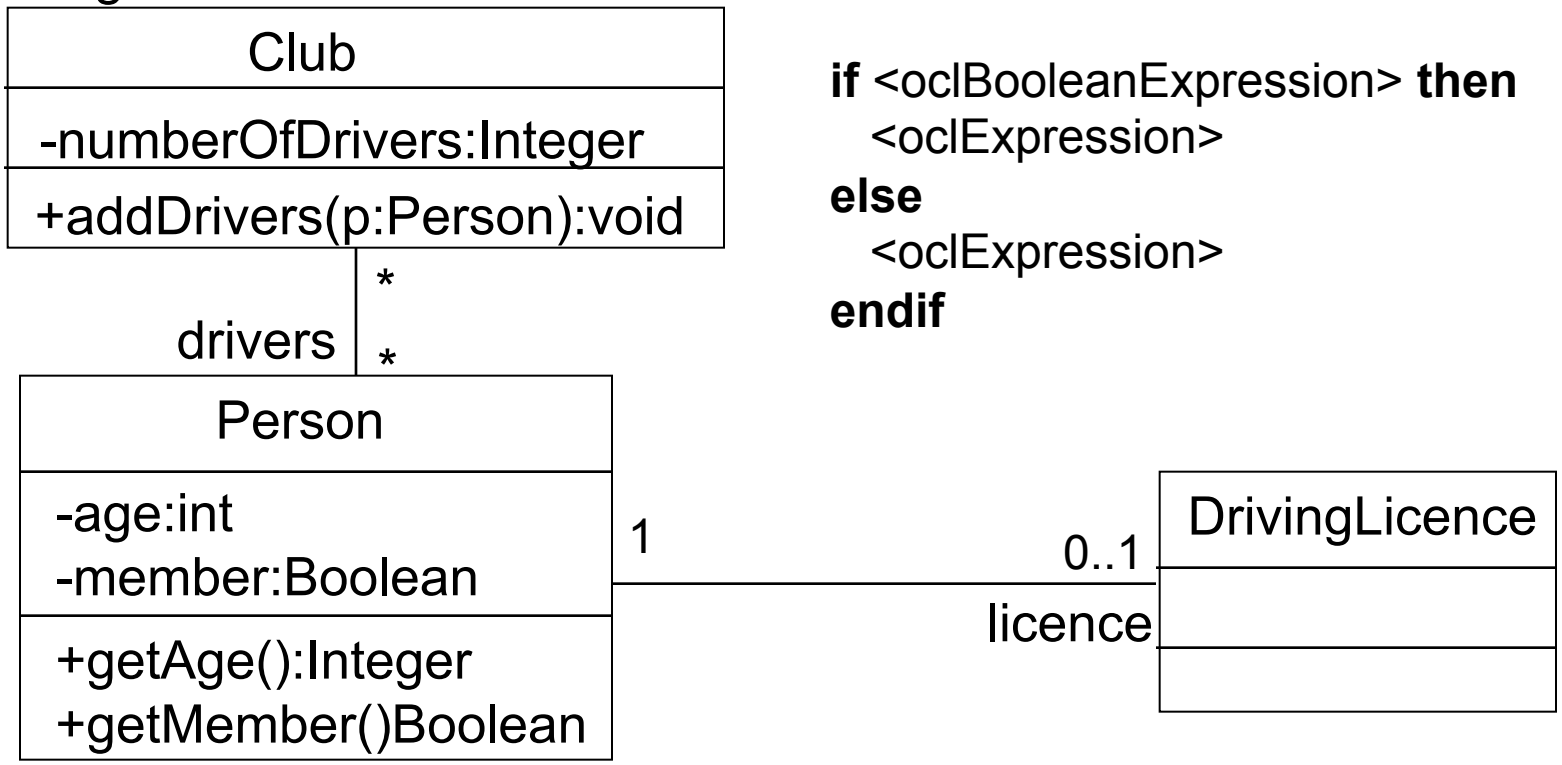
- context LectureRoom inv:
  numberOfSeats > 10

Pre- and Postconditions

- The precondition specifies what has to hold before the call to the operation.
- The postcondition has to specify what has to hold after the execution of the call.

Problem

- Write pre and post-conditions for operation addDrivers in class Club. A pre-condition is that the person needs to be a member of the club. As post-condition a person should be added to ‘drivers’ if the age of the person is more than 20 years and the person has a driving licence.



if <oclBooleanExpression> **then**
  <oclExpression>
else
  <oclExpression>
endif

Solution

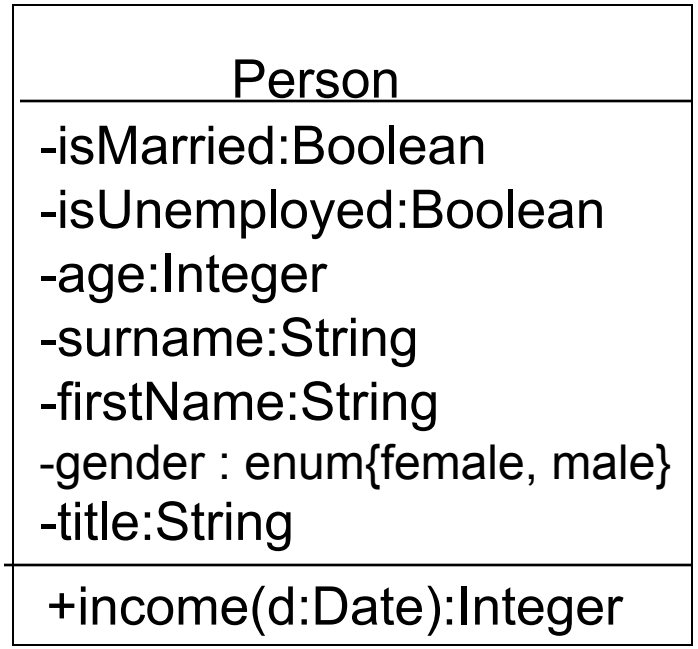
- context Club::addDrivers(p:Person):void
  pre: p.getMember()
  post: if p.age > 20 and p.licence->notEmpty()
  then drivers = drivers@pre->including(p)
  else true endif
- post: (p.age > 20 and p.licence->size() = 1) implies
  drivers = drivers@pre->including(p)

Stronger condition:

- post: if p.age > 20 and p.licence->size() = 1
  then drivers = drivers@pre->including(p)
  else drivers = drivers@pre endif

Boolean

context Person **inv:**
title = (if gender = #male
then 'Herr.'
else 'Fru.' endif)



context Person **inv:**
gender = #male implies title = 'Herr.'

'#' is used to distinguish between attributes and elements of enumerations

Collections

Types Set(X), Bag(X) and Sequence(X) are subtypes of Collection(X).

Lots of operations are defined for collections:

=, size, sum, includes, isEmpty, exists, forAll...

Set Operations

- Operations on collections (sets, bags, sequences) are always invoked with an arrow '->', e.g.

context Company **inv**:
numberOfEmp = employees -> size()

Sets

Set {1, 4, 9, 55}

Operations defined for sets:
union, intersection, -, include, exclude,
select, reject, collect, asBag, asSequence … .

Example: Sets, Bags

```
Set{1, 3, 8, 12} - Set{3,12} = Set{1, 8}          -- Set(Integer)
Set{1, 3}->union(Set{4}) = Set{1, 3, 4}           -- Set(Integer)
```

Bags can be written in the same way:

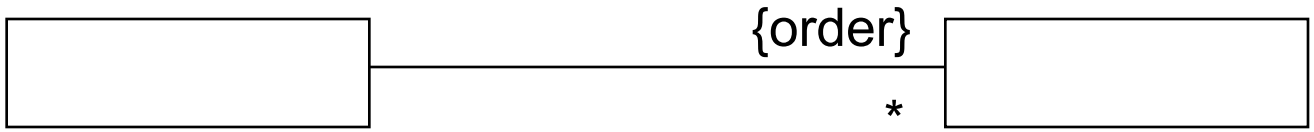
```
Bag{1,2,2,5}
```

Sequences

Sequence{1,8,6,9}

Operations defined for sequences:
union, =, append, prepend, at, first, last, including , exclude, select, reject, collect, asBag, asSet … .

Ordered associations ends are sequences in OCL:

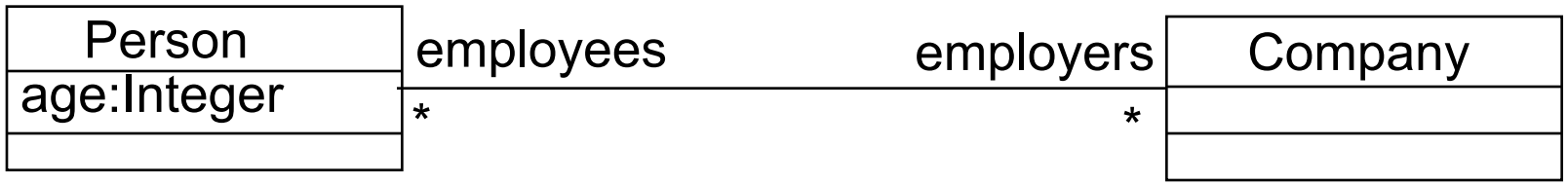


Example: Sequence

```
Sequence{1, 13, 8, 12} ->first = 1          -- Integer
Sequence{1, 13, 8, 12} ->last = 12          -- Integer
Sequence{1, 13, 8, 12} ->at(3) = 8          -- Integer
Sequence{1, 13, 8, 12} ->append(15) =
            Sequence{1,13,8,12,15}   --Sequence(Integer)
```

Example: select

```
context Company inv:
  self.employees->select(age > 45)->notEmpty

context Company inv:
  self.employees->select(p | p.age > 45)->notEmpty

context Company inv:
  self.employees->select(p: Person | p.age > 45)->notEmpty
```



Example: collect

context Company:

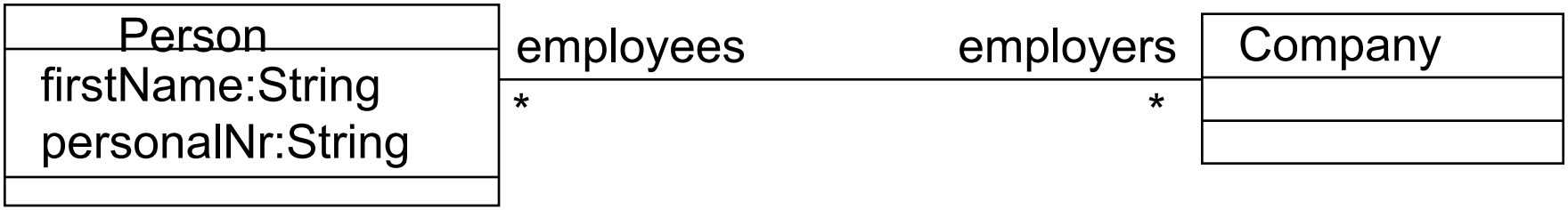
```
self.employees->collect(birthDate)                    -- Bag(Date)
self.employees->collect(p | p.birthDate)
self.employees->collect(p : Person | p.birthDate)

self.employees->collect(birthDate)->asSet
```

Example: ForAll

context Company inv:
    self.employees->forall(firstName = 'Jack')

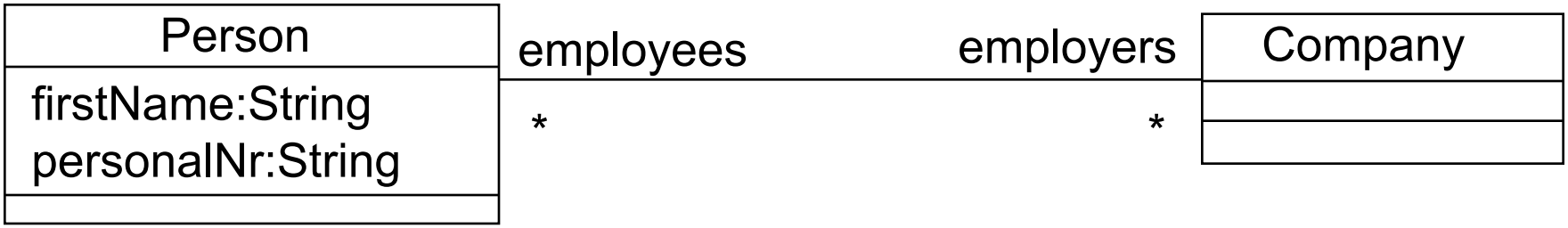
context Company inv:
    self.employees->forall(e_1,e_2:Person |
        e_1 <> e_2 implies e_1.personalNr <> e_2.personalNr)

context Company inv:
    self.employees->forall(e_1| self.employees-> forall (e_2 |
        e_1 <> e_2 implies e_1.personalNr <> e_2.personalNr))



Example: Exists

```
context Company inv:
    self.employee->exists(firstName = 'Jack')
```

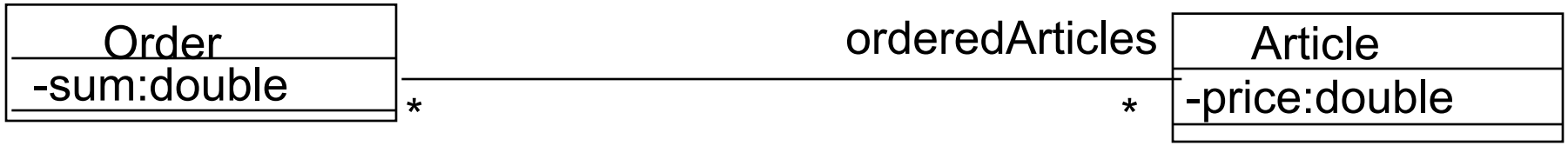


Iterate

- Most powerful and most complicated of all OCL collection operations.

```
collection->iterate(elem : Type;
                    acc : Type = <expression> |
                    expression-with-elem-and-acc)
```

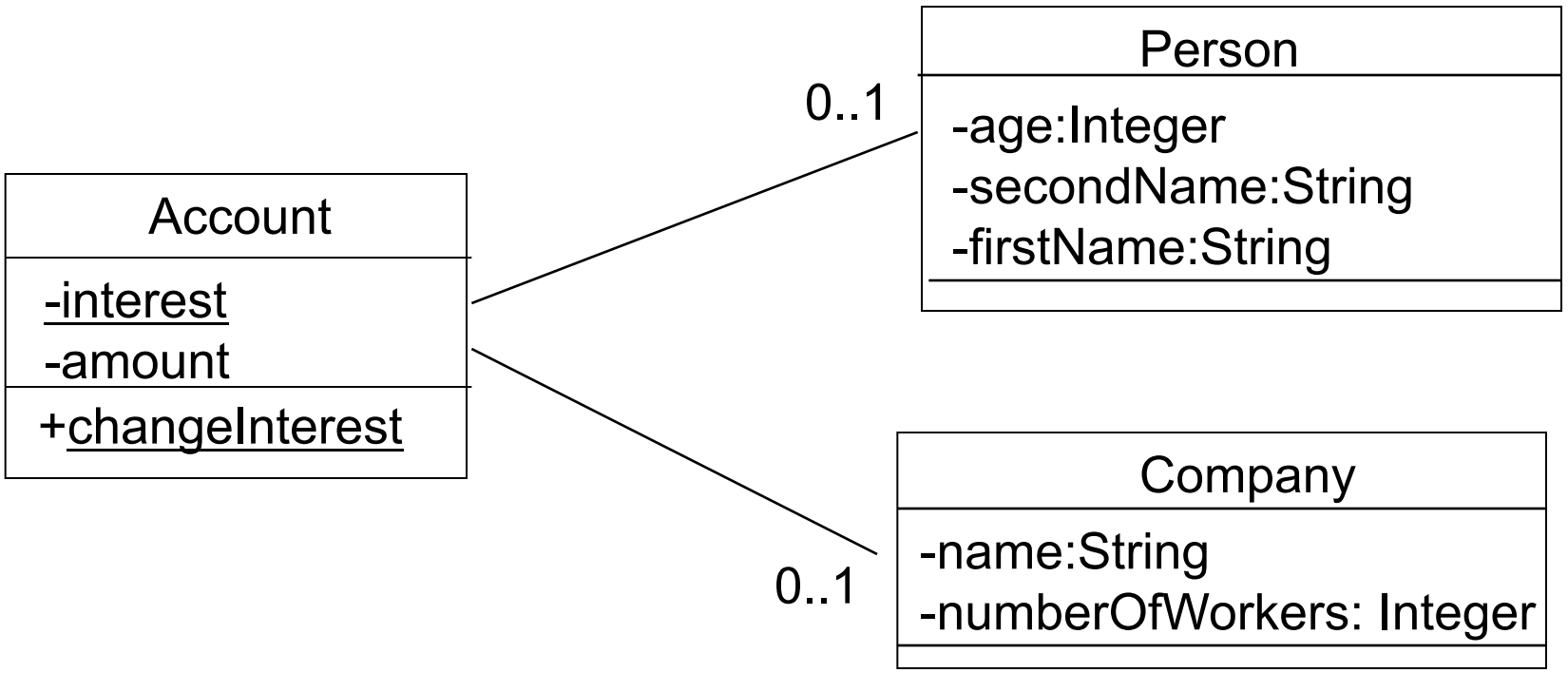
Example:



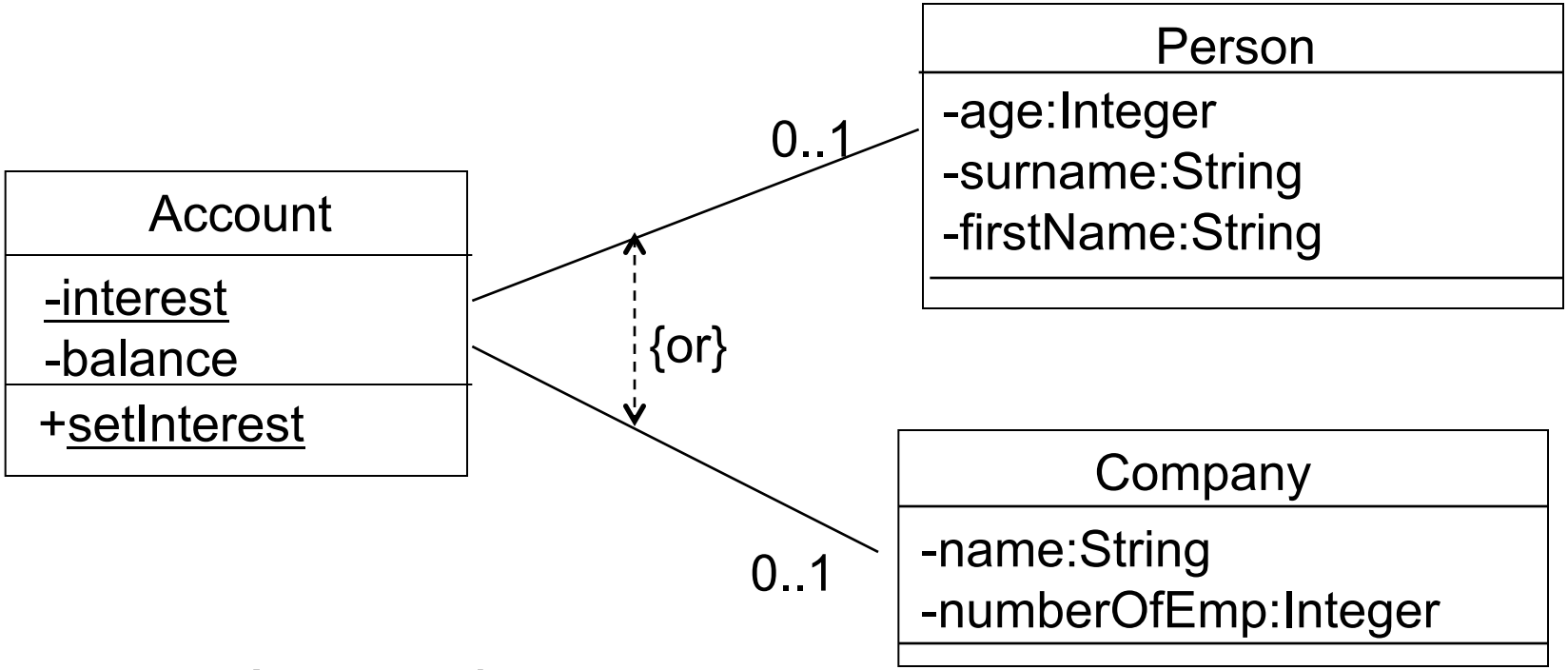

```
context Order inv:
   sum = orderedArticles->iterate(a:Article; result : Real = 0
                  | result + a.price)

context Order inv:
    sum= orderedArticles->collect(price)->sum
```

Problem

Express in OCL that an **Account** can be associated with a **Person** or a **Company** but not with both.



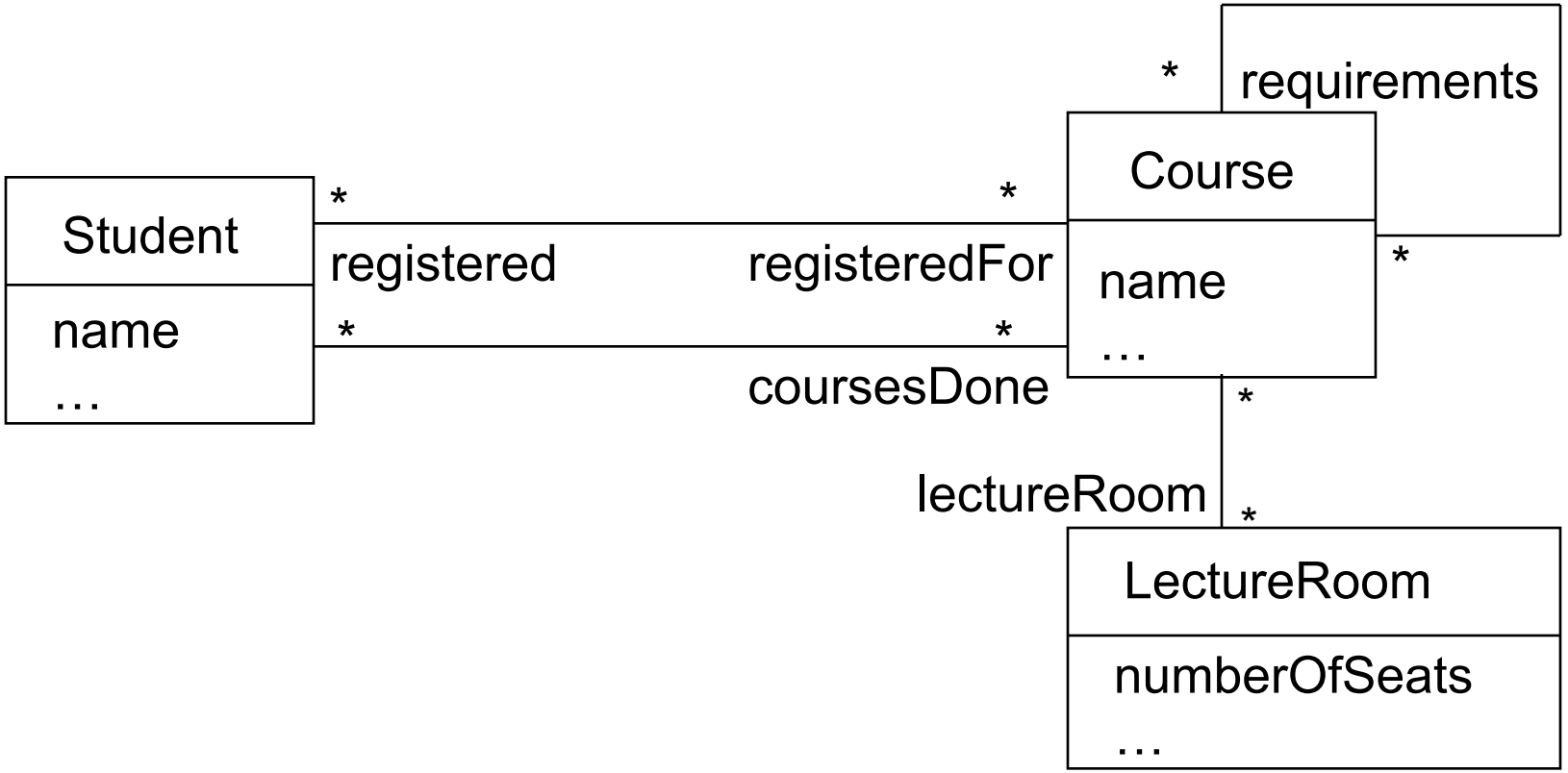
Solution {or}




```
context Account inv:
    person->intersection(company)->isEmpty

context Account inv:
    self.person->isEmpty or self.company->isEmpty
```

Problem

- Write an invariant which does not permit more students to register than there are seats in a lecture room.

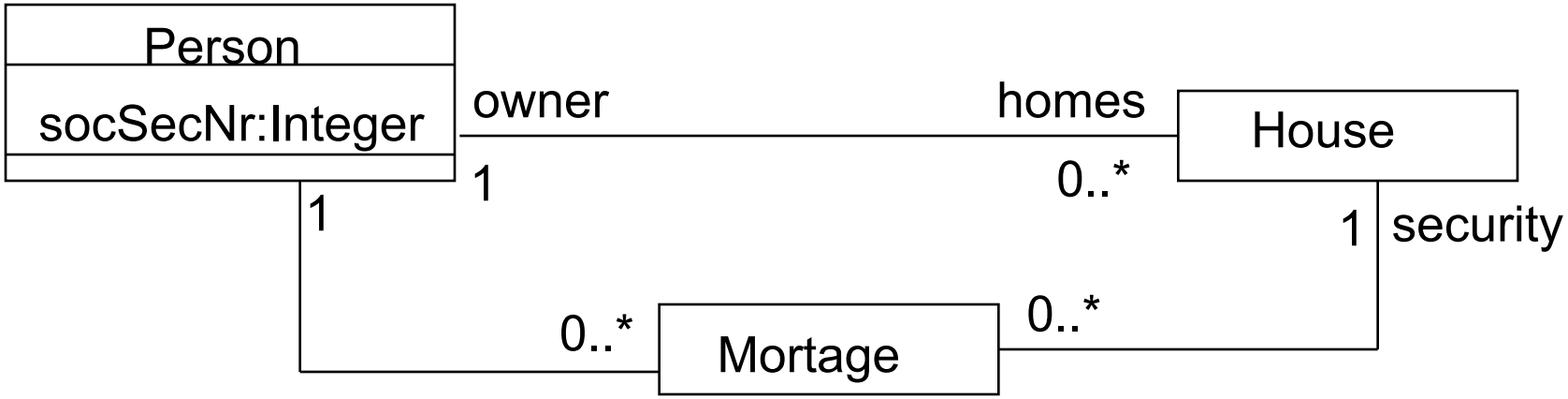


Solution

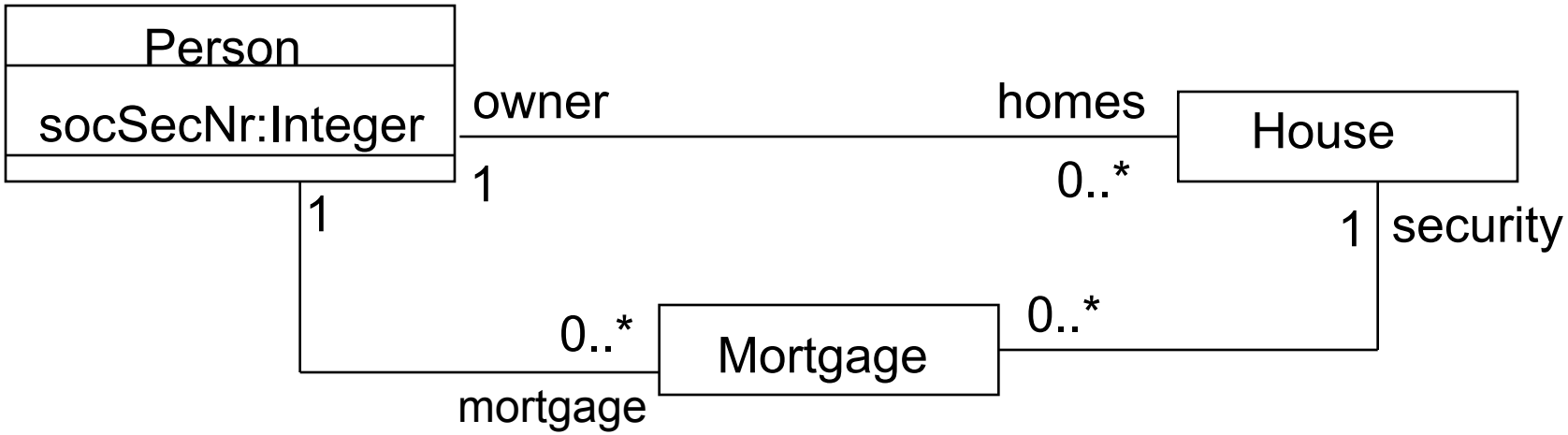
- context Course inv:
  lectureRoom->forAll(self.registered->size() <= numberOfSeats)

Problem

- Express using OCL that if a house is used as security, then one has to own the house.
- Choose context Person.



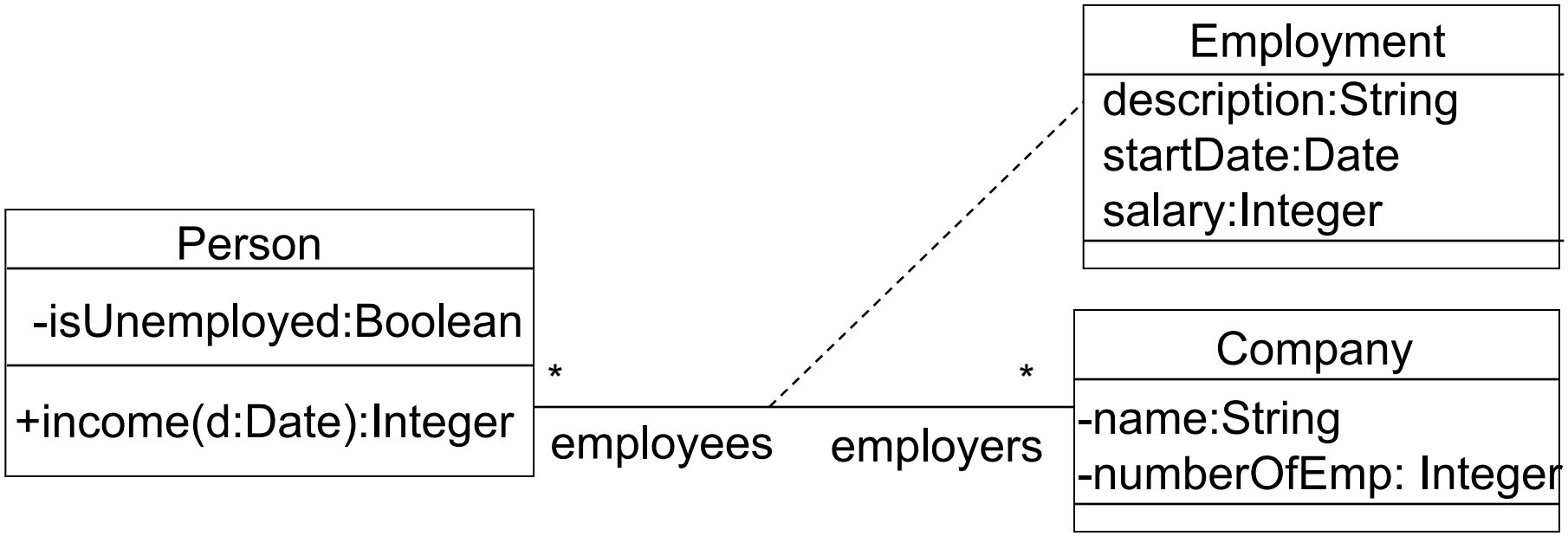
Solution




```
context Person inv:
   mortgage.security.owner = self
```

Let Expressions

```
context Person inv:
    let income : Integer = self.employment.salary->sum in
    if isUnemployed then income < 8000
                    else income >= 8000
    endif
```



Inheritance

Liskov's Substitution Principle:

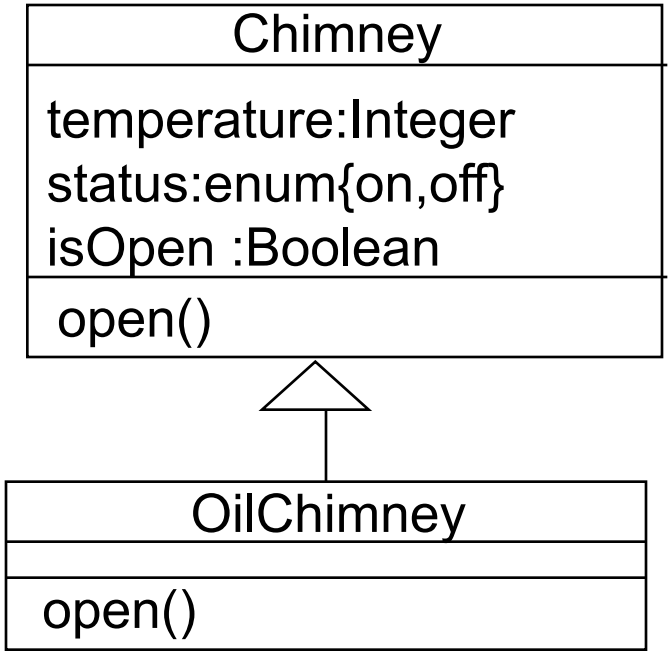
- "Wherever an instance of a class is expected also instances of subclasses can be used"
- This implies the following points:
  - Invariants of superclasses are inherited by subclasses. In subclasses, invariants may be made stronger, but not weaker (or unrelated)
  - Preconditions may be made weaker, but not stronger (or unrelated), if an operation is overridden in a subclass
  - Postconditions may be made stronger, but not weaker (or unrelated), if an operation is overridden in a subclass

Example

context Chimney
inv: temperature <= 300

context OilChimney
inv: temperatur <= 200

~~context OilChimney~~
~~inv: temperatur <= 500~~

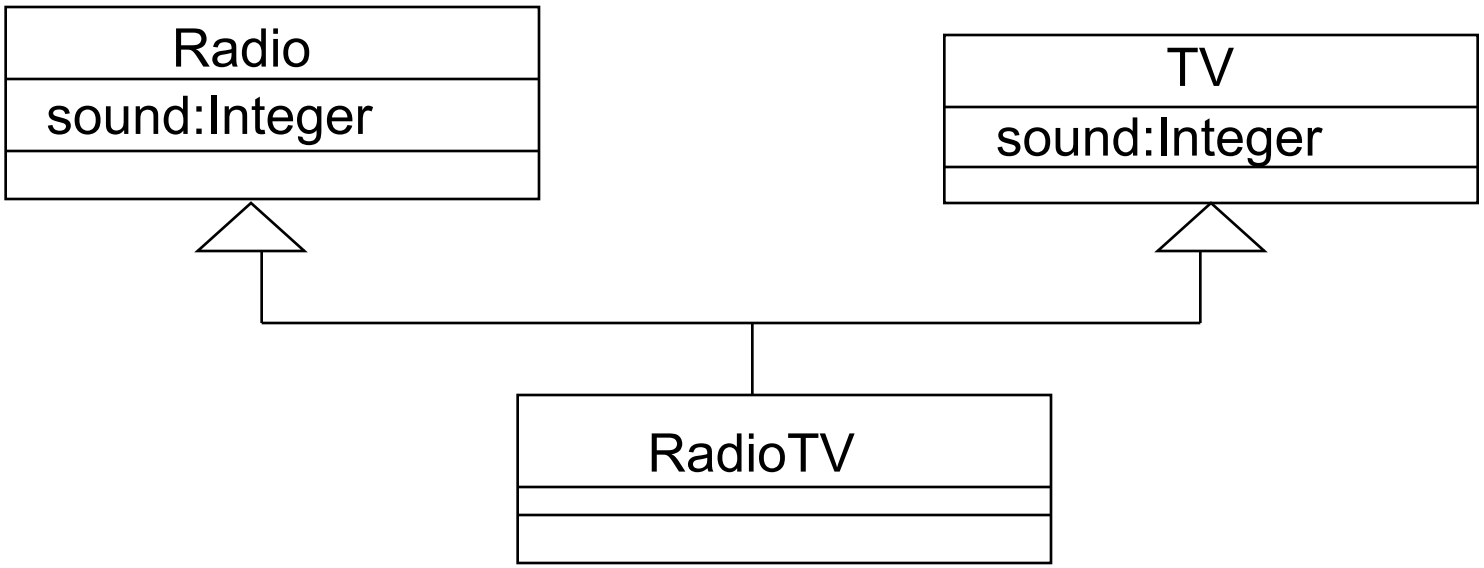


context Chimney::open()
pre : status = #off
post: status = #off and isOpen

context OilChimney::open()
pre : --
post: status = #off and isOpen

~~context OilChimney::open()~~
~~pre : temperature <= 100~~
~~post: isOpen~~

Multiple Inheritance



context RadioTV
inv: Radio::ljud < 12

OclType

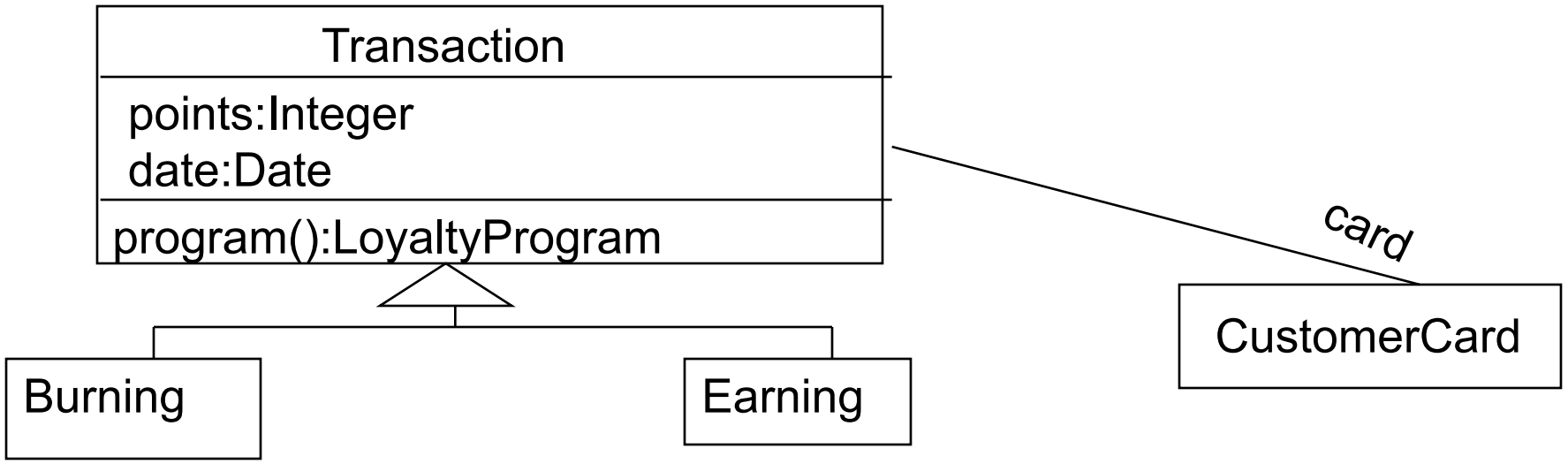
The types of types …

```
sometype.name                -- String
sometype.attributes          -- Set(String)
sometype.operations          -- Set(String)
sometype.supertypes          -- Set(OclType)
sometype.allSupertypes       -- Set(OclType)
sometype.allInstances        -- Set(sometype.oclType)
```

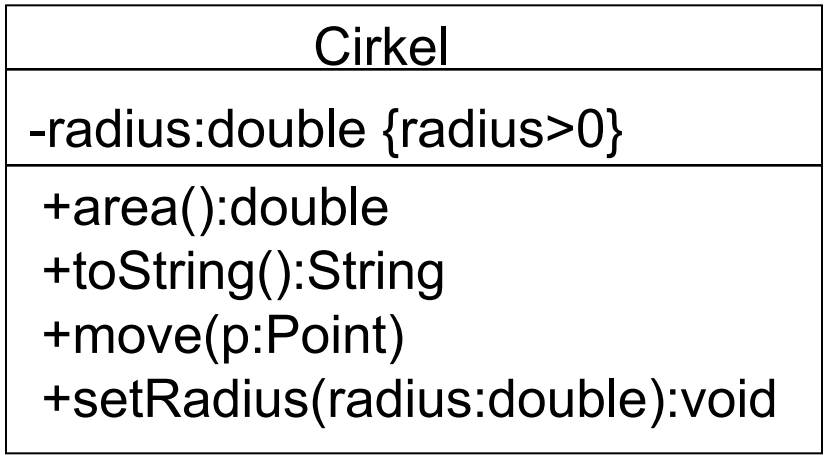
Person.allInstances – give all objects of Person

Example: OclType

Transaction.name = ’Transaction’
Transaction.attributes = Set(’point’,’date’)
Transaction.associationEnds = Set{’card’}
Burning.supertypes = Set{Transaction}
Transaction.operations = Set{’program’}



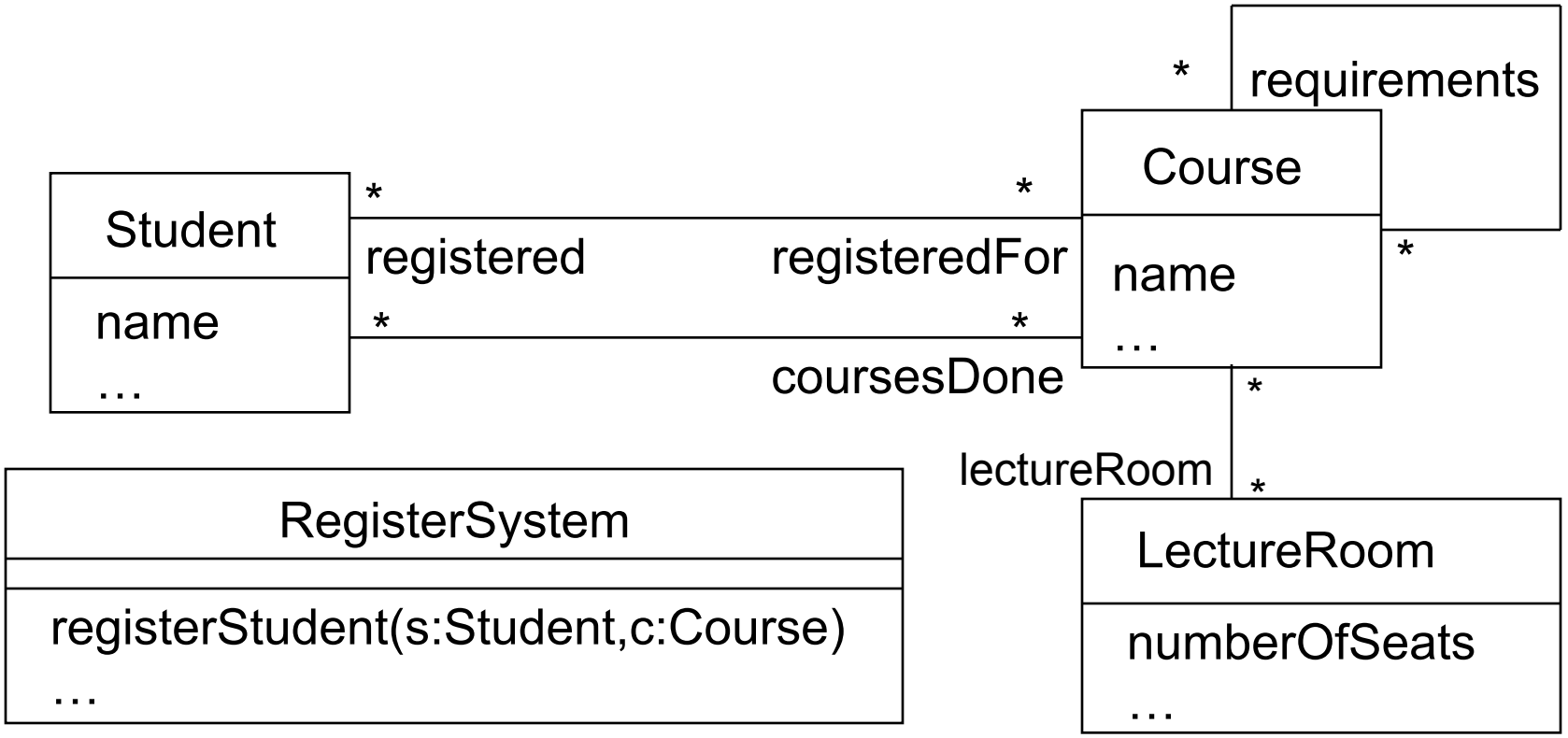
Constraints written in Java




```
class Circle{
    private int radius;

    public void setRadius(int radius){
        if (radius > 0){     // constraint
                this.radius= radius;
        else
                …
        }
    }
    …
}
```

An exception should be thrown if the constraint is violated …

Problem: Contract



Write a post-condition of a contract for operation registerStudent. Should only register students if it has all the right requirements for the course and the lecture room is large enough.

Solution

- context RegisterSystem::
  registerStudent(s:Student,c:Course):void
  post: s. coursesDone -> includesAll
  (s.registeredFor->
  collect(requirements)->flatten())
  and c.lectureRoom->forAll(numberOfSeats
  > c.registered->size())
  implies c.registered = c.registered@pre -> including(s)

Summary

- We have considered how to use OCL in combination with UML to give constraints on the model.