OCL – The Object Constraint Language

Gábor Bergmann, Ákos Horváth, Dániel Varró, István Majzik and Gergely Pintér

Model Driven Software Development

Lecture 4b

Budapest University of Technology and Economics
Department of Measurement and Information Systems

OCL Motivation

How to capture restrictions / constraints of domain classes?

Motivation

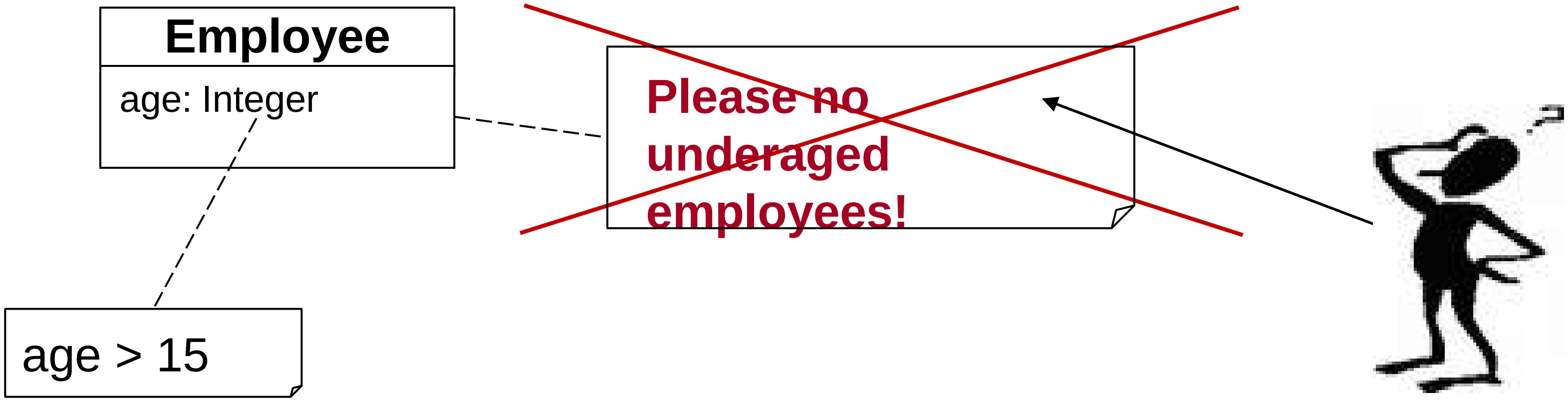
- Graphical modeling languages are generally not able to describe all facets of a problem description
  - *MOF, UML, ER, …*

- Special **constraints** are often (if at all) added to the diagrams in **natural language**
  - Often **ambiguous**
  - Cannot be validated **automatically**
  - No **automatic** code generation

- Constraint definition also crucial in the definition of new modeling languages (DSLs).

Motivation

- Example 1

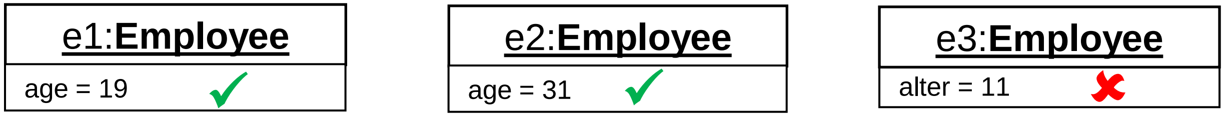


Employee

age: Integer

age > 15

~~**Please no underaged employees!**~~



e1:**Employee**

age = 19 ✓

e2:**Employee**

age = 31 ✓

e3:**Employee**

alter = 11 ✗

Additional question: How do I get all Employees younger than 30 years old?

Motivation

- **Formal specification languages** are the solution
  - Mostly based on **set theory** or **predicate logic**
  - Requires good mathematical understanding
  - Mostly used in the academic area, but hardly used in the industry
  - Hard to learn and hard to apply
  - Problems when to be used in big systems

- ***Object Constraint Language (OCL)*:** Combination of modeling language and formal specification language
  - Formal, precise, unique
  - Intuitive syntax is key to **large group of users**
  - No programming language (no algorithms, no technological APIs, …)
  - Tool support: *parser, constraint checker, codegeneration,…*

OCL usage

- Constraints in UML-models
  - Invariants for classes, interfaces, stereotypes, …
  - Pre- and postconditions for operations
  - Guards for messages and state transition
  - Specification of messages and signals
  - Calculation of derived attributes and association ends

- Constraints in meta models
  - Invariants for Meta model classes
  - Rules for the definition of well-formedness of meta model

- Query language for models
  - In analogy to SQL for DBMS, XPath and XQuery for XML
  - Used in transformation languages

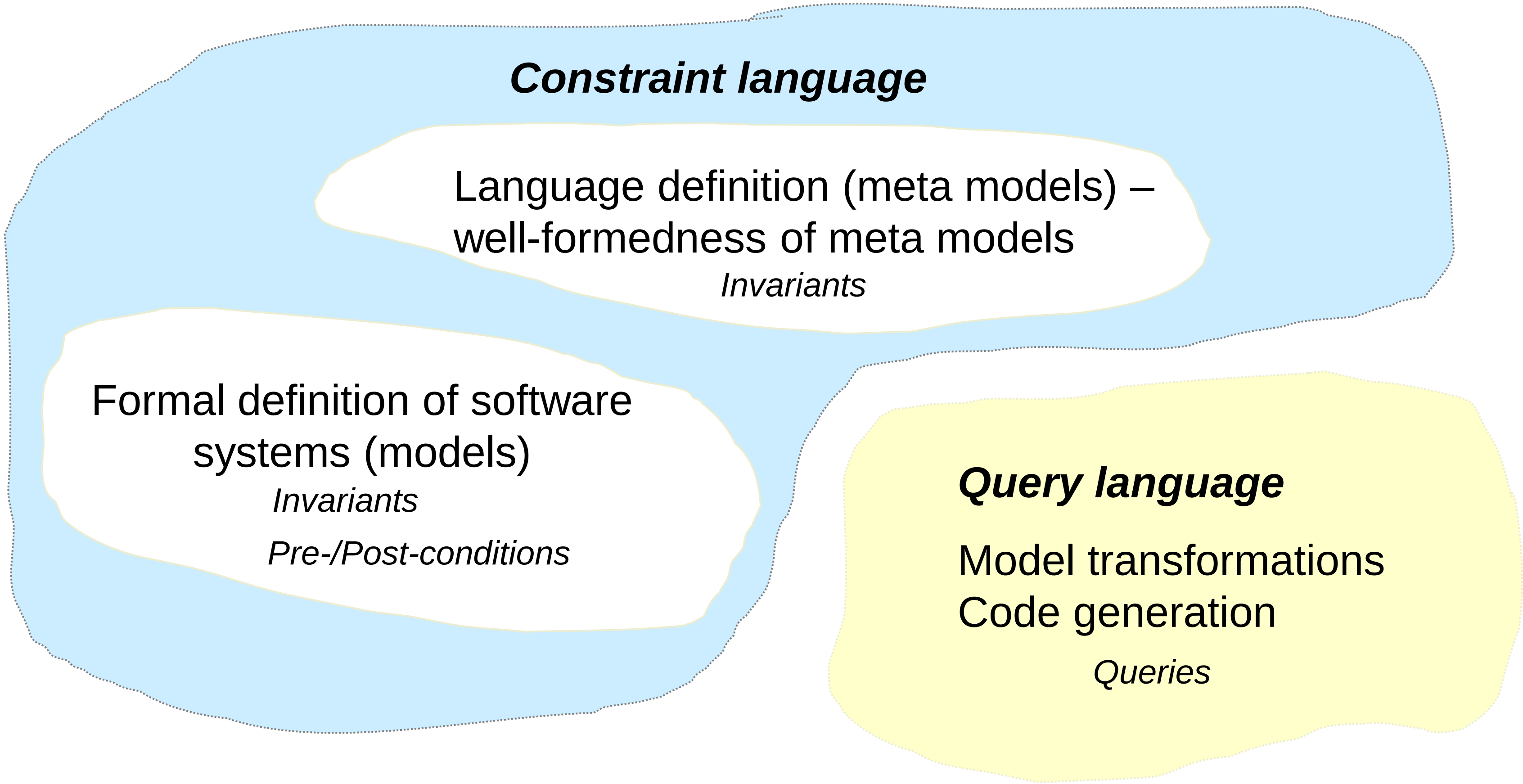
OCL usage

- OCL field of application
  - Invariants **context** C **inv:** *I*
  - Pre-/Postconditions **context** *C::op() : T* **pre:** *P* **post:** *Q*
  - Query operations **context** *C::op() : T* **body:** *e*
  - Initial values **context** *C::p : T* **init:** *e*
  - Derived attributes **context** *C::p : T* **derive:** *e*
  - Attribute/operation definition **context** C **def:** *p : T* = *e*

- Caution: Side effects are not allowed!
  - Operation `C::getAtt : String body: att` allowed in OCL
  - Operation `C::setAtt(arg) : T body: att = arg` **not** allowed in OCL

OCL usage

- **Field of application** of OCL in model driven engineering



Constraint language

Language definition (meta models) –
well-formedness of meta models

Invariants

Formal definition of software
systems (models)

Invariants

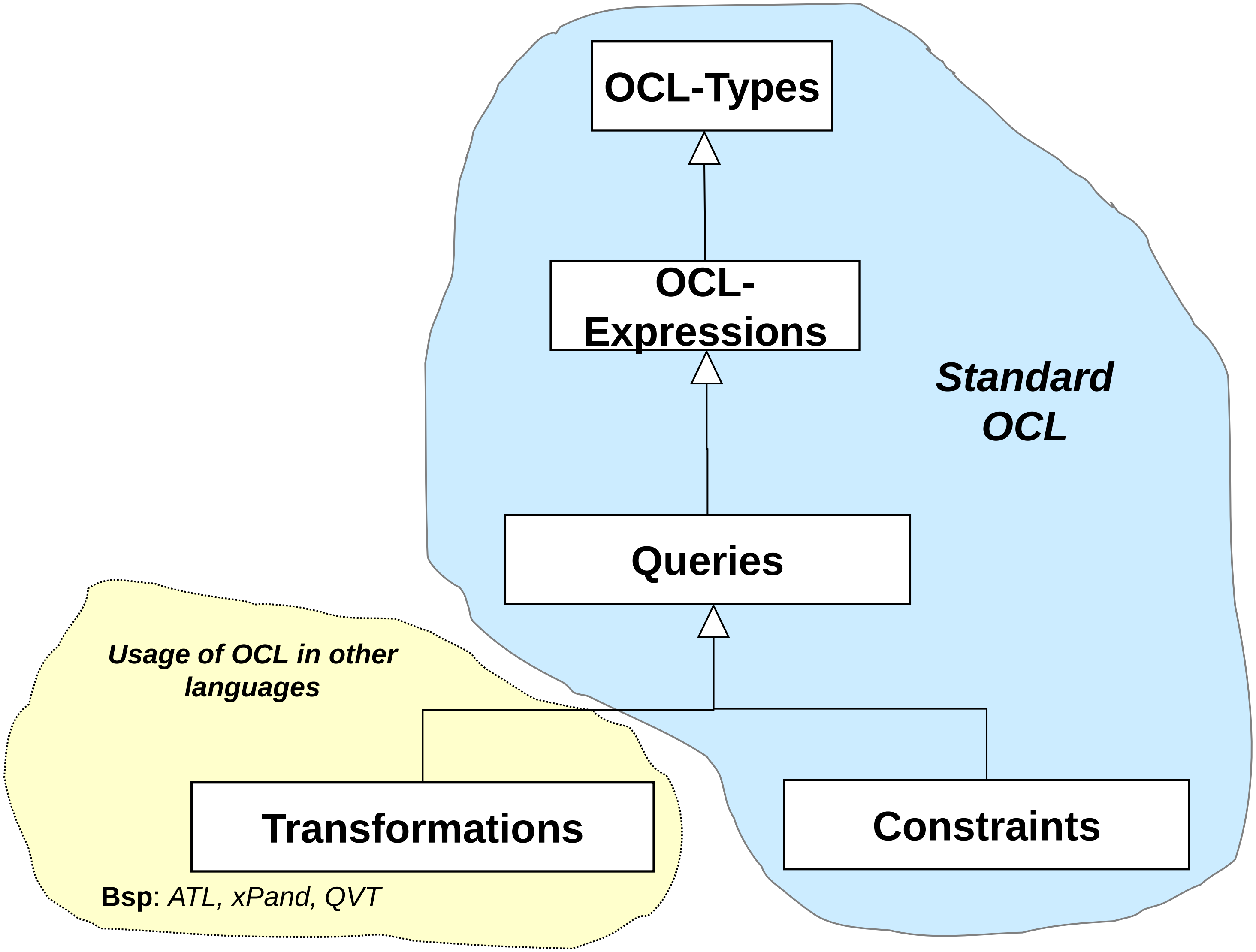
Pre-/Post-conditions

Query language

Model transformations
Code generation

Queries

OCL usage



OCL-Types

OCL-Expressions

Standard *OCL*

Queries

Usage of OCL in other languages

Transformations

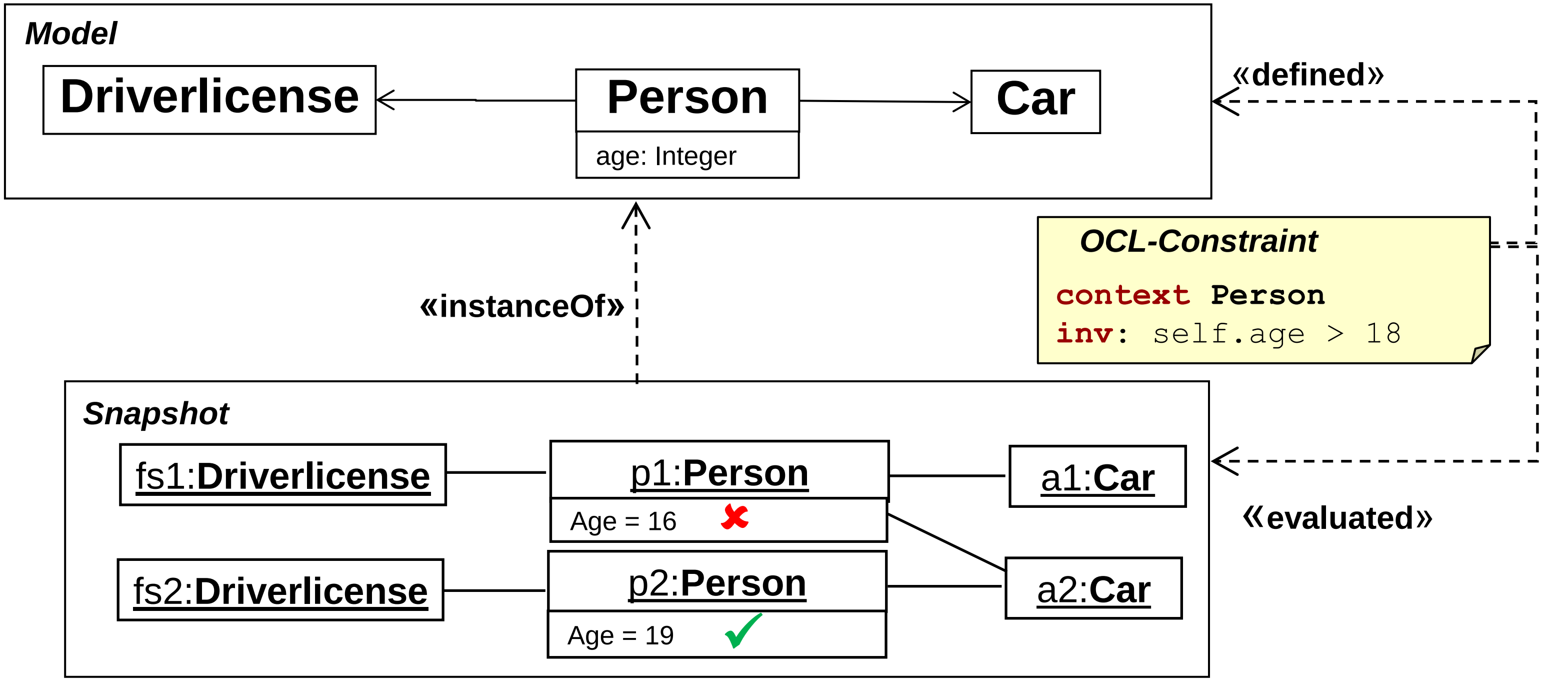
Constraints

Bsp: *ATL, xPand, QVT*

OCL usage

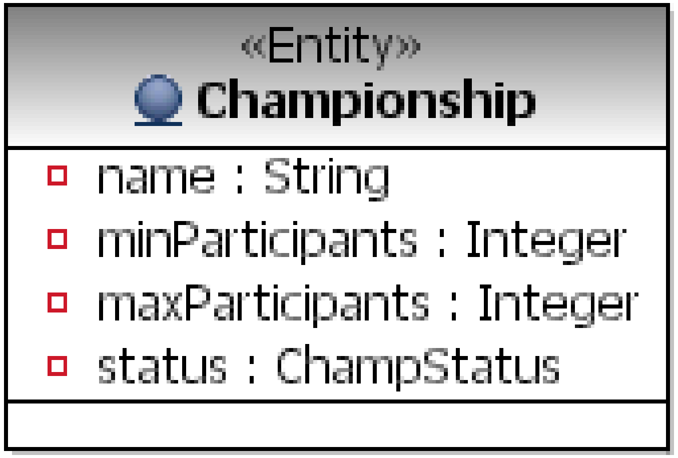
How does OCL work?

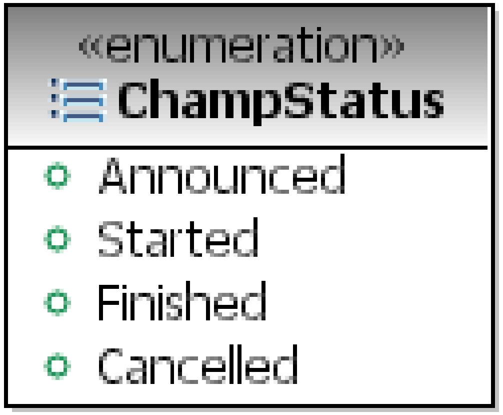
- **Constraints** are defined on the modeling level
  - Basis: Classes and their properties
- Information of the **object graph** are **queried**
  - Represents system status, also called ***snapshot***
- **Anaology** to XML query languages
  - XPath/XQuery query XML-documents
  - Scripts are based on XML-schema information
- Examples



First OCL Examples

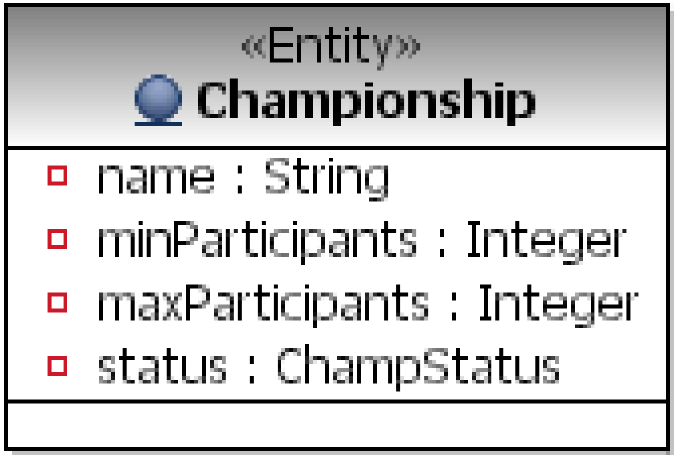
Informal Constraints on Championship

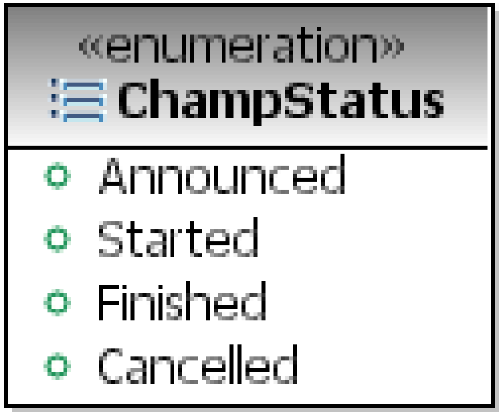

«Entity»
Championship
□ name : String
□ minParticipants : Integer
□ maxParticipants : Integer
□ status : ChampStatus


«enumeration»
ChampStatus
○ Announced
○ Started
○ Finished
○ Cancelled

- What are the restrictions?
  - `name` is not empty
  - `minParticipants ≤ maxParticipants`
  - `minParticipants ≥ 0`
  - `maxParticipants > 0`

First OCL constraints





- Name is not empty

Context

Invariant

```
context Championship inv:
  self.name <> ''
```

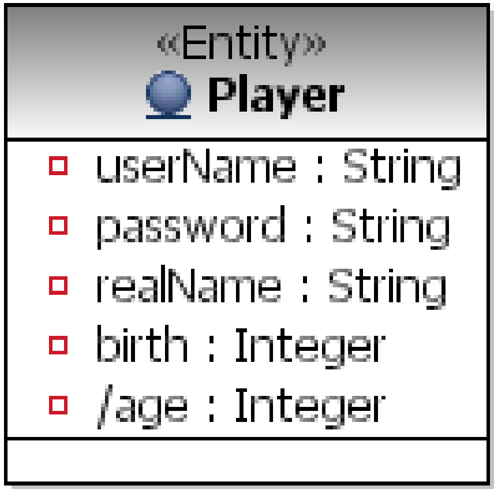
- Constraints on participants

```
context Championship inv:
  self.minParticipants >=
  0
context Championship inv:
  self.maxParticipants >=
  1
context Championship inv:
  self.maxParticipants >=
  self.minParticipants
```

Instance of the class

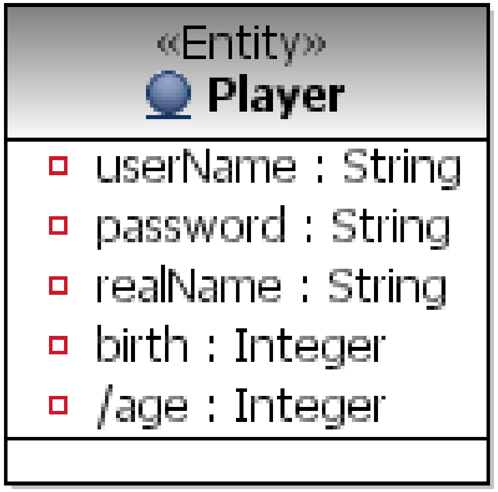
Navigation along attributes

Informal Constraints on Player



- What are the restrictions?
  - `userName` is not empty
  - `userName` is unique
  - `1800 ≤ birth ≤ 3000`
  - `password` is not empty
  - `age = current_year - birth`

Informal Constraints on Player



- 1800 ≤ birth ≤ 3000

```
context Player inv:
  self.birth >= 1800 and
  self.birth <= 3000
```

- Name is unique

```
context Player inv:
  Player.allInstances()->
  forAll(p1, p2 | p1<>p2 implies
  p1.userName <> p2.userName)
```

Get all instances into a collection

Logical AND

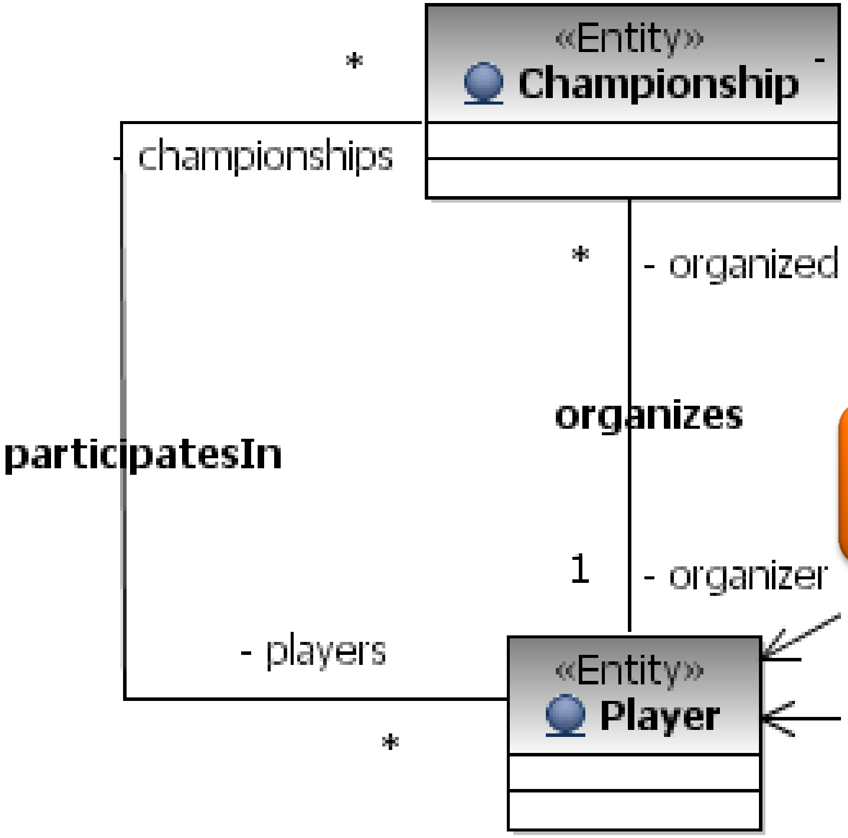
Logical implication

If p1 ≠ p2

Then p1.userName ≠ p2.userName

Universal quantification: For all objects in the collection

Navigation along roles



- Multiplicity 0..1

```
context Championship inv:
  self.organizer.birth >
  1976
```

Only attributes of an **object** can be compared with a value

- Multiplicity * (many)

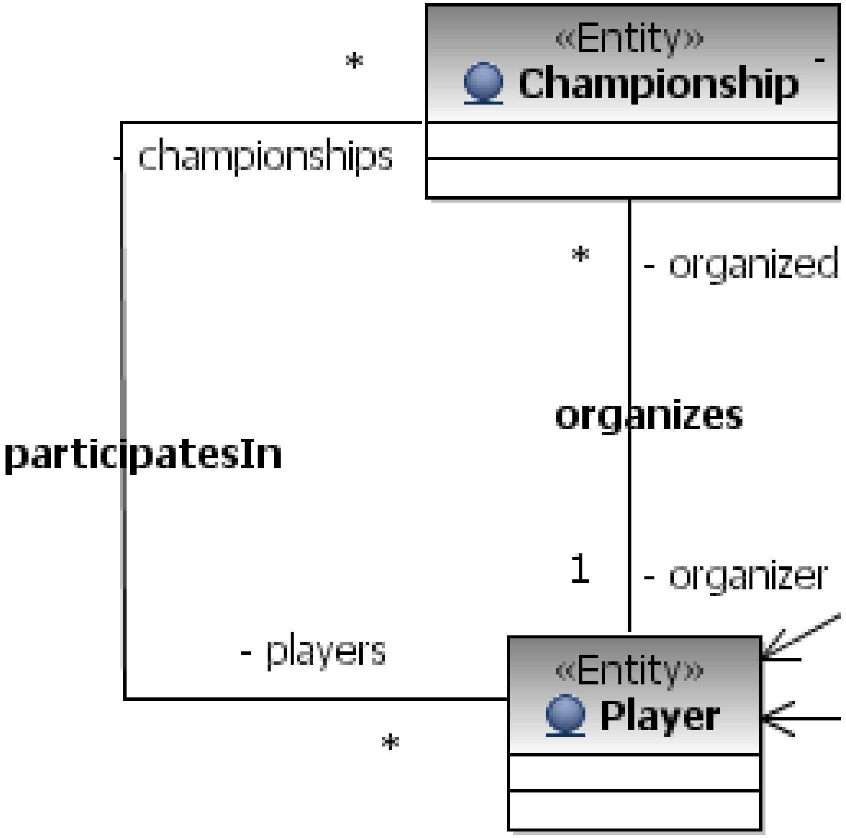
```
~~context Championship inv:
  self.players.birth > 1976~~
```

`self.players` results in a **collection**
`self.players.birth`: the coll. of birth years

```
context Championship inv:
  self.players-> …
  (operations on
  collections)
```

Consistency of bidirectional associations

- If a bidirectional association exists between two objects then it is navigable from both directions



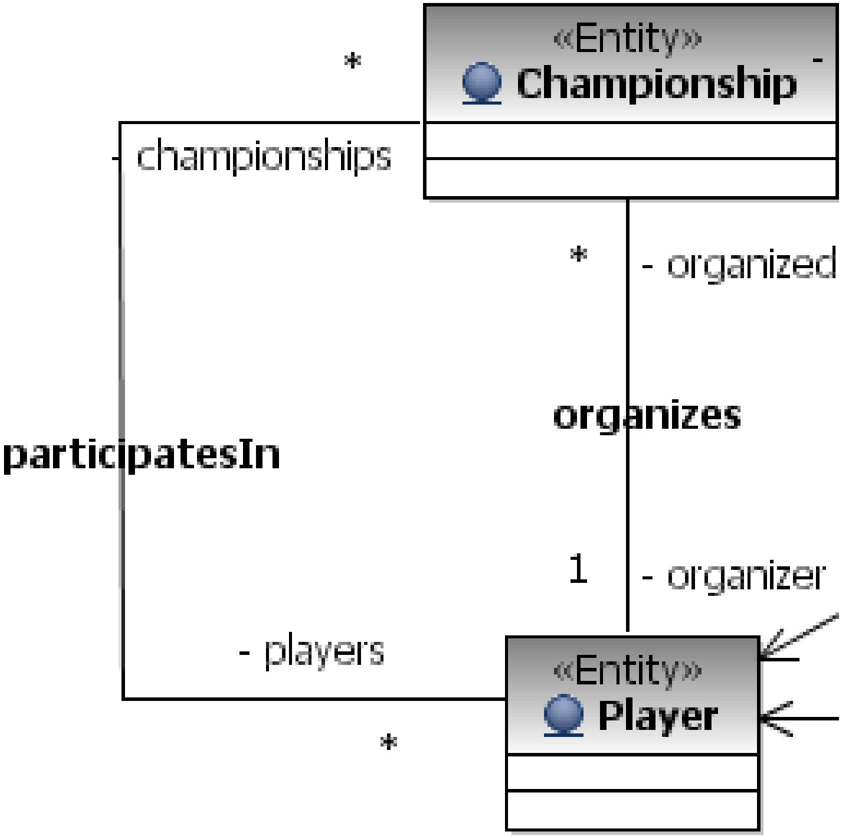
~~context Championship inv:~~
~~self.organizer.organized=self~~

Collection = Single object
Such an equality is invalid

```
context Championship inv:
  self.organizer.organized
  -> includes(self)
```

`Coll->includes(e):` Tests collection membership: **e ∈Coll**

Consistency of bidirectional associations



- If a bidirectional association exists between two objects then it is navigable from both directions

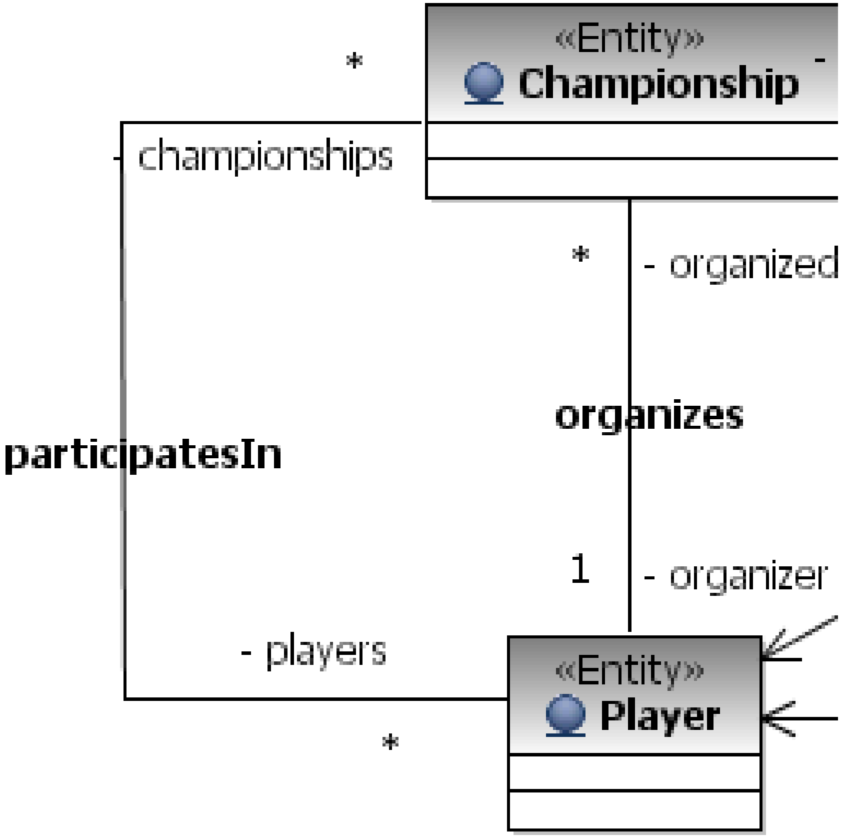
```
~~context Player inv:
  self.organized->exists(
  c | c.organizer = self)~~
```

Incorrect: constraint is prescribed **for all** champs

```
context Player inv:
  self.organized->forAll(
  c | c.organizer = self)
```

`Coll->forAll(e|cond(e))`
Quantifiers can only be applied to collections

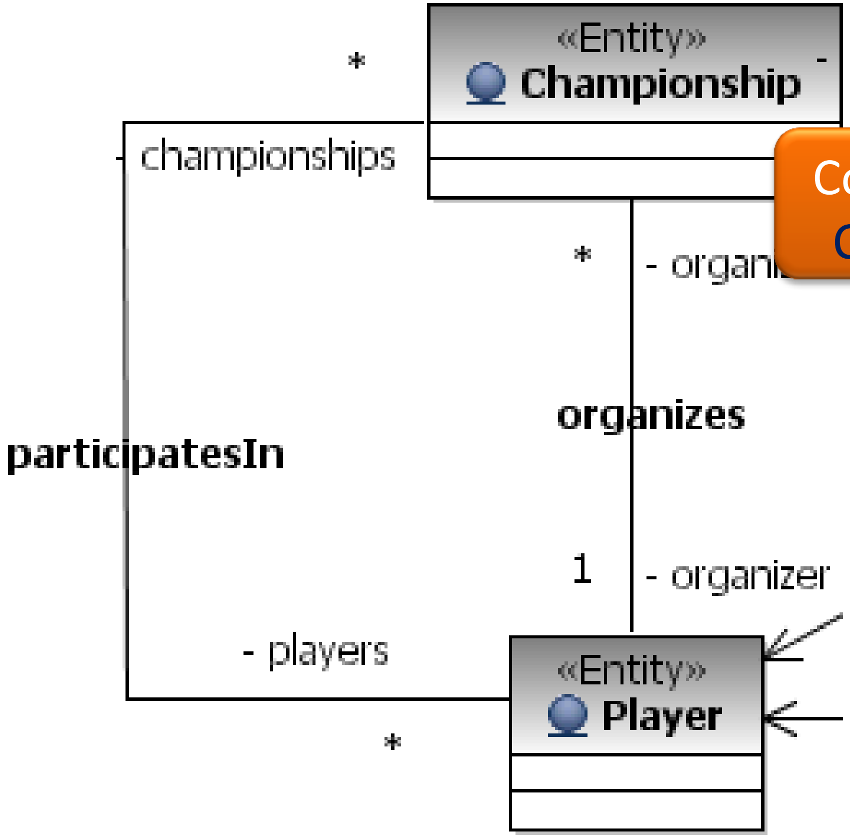
Consistency of bidirectional associations



- If a bidirectional association exists between two objects then it is navigable from both directions

```
context Championship inv:
   self.players->forall(
   p | p.championships->
   includes(self))
context Player inv:
   self.championships->forall(
   c | c.players ->
   includes(self))
```

Consistency of bidirectional associations



- The organizer of the championship organizes at least one championship

~~context Player inv:~~
~~self.organized->size() > 0~~

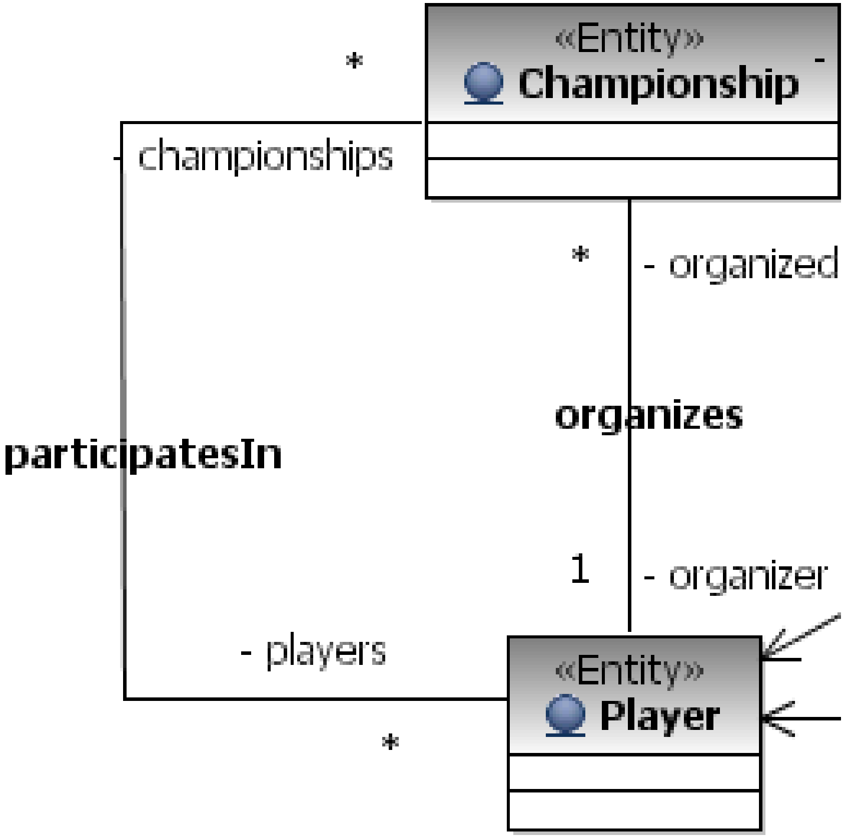
Context should be Championship

No player is forced to organize a champs

```
context Championship inv:
  self.organizer.organized->
  size() > 0
```

```
context Championship inv:
  self.organizer.organized->
  notEmpty()
```

Application specific constraints

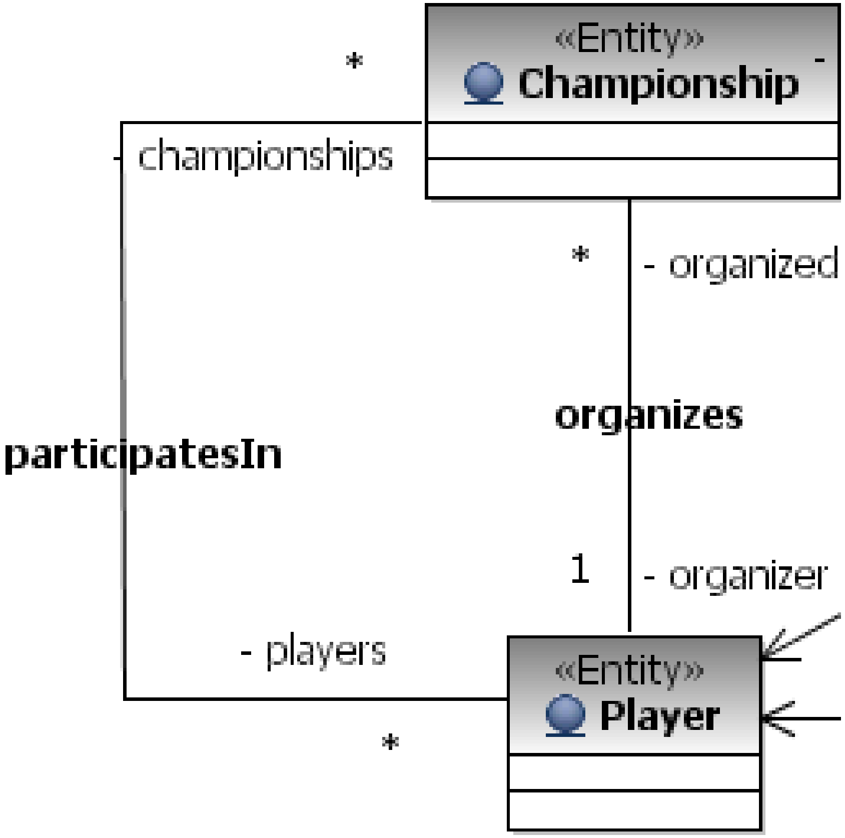


- A player is allowed to organize a single active championship at a time

```
context Player inv:
  self.organized->
  forall(c1, c2 | c1<>c2 implies
  (c1.status = ChS::closed or
   c1.status = ChS::cancelled)
  or
  (c2.status = ChS::closed or
   c2.status = ChS::cancelled))
context Player inv:
  self.organized->select(c |
  c.status = ChS::announced or
  c.status = ChS::started)->
  size() <=1
```

Values of an enumeration

Application specific constraints

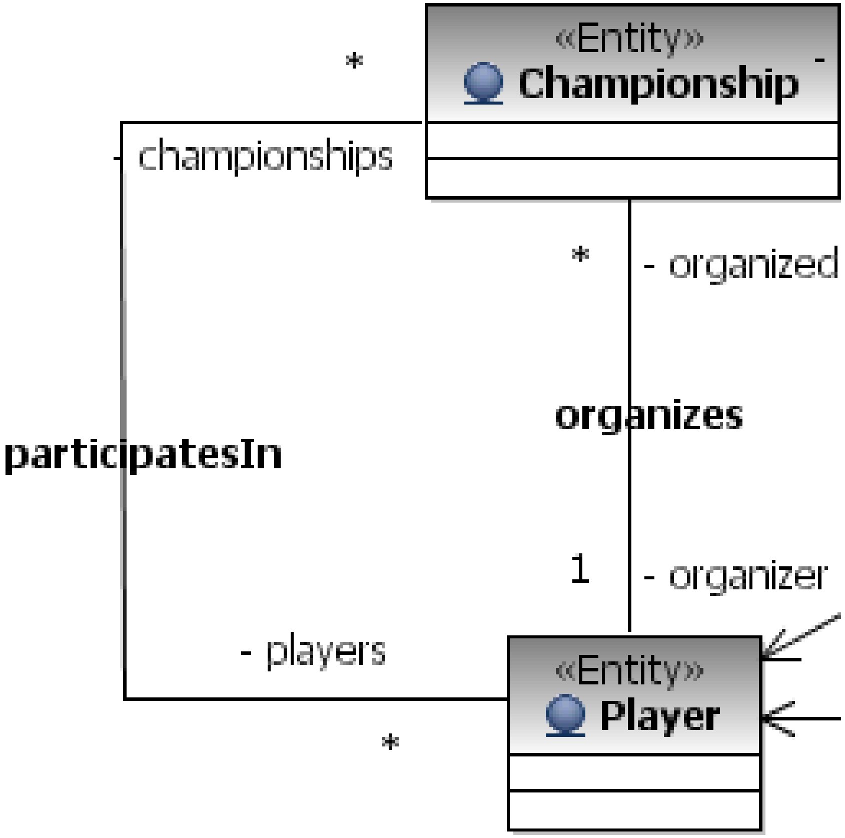


- A championship can only be started when the sufficient number of participants are present.

```
context Championship inv:
   (self.status =
   ChampStatus::started or
   self.status =
   ChampStatus::finished)
   implies
   (self.players->size() >=
    self.minParticipants and
    self.players->size() <=
    self.maxParticipants)
```

Application specific constraints

- Youth championship: the average age of participants is below 21.



players.age is the collection of the age attributes of players

```
context Championship inv:
  self.players.age->sum() /
  self.players->size() < 21
```

sum() can only be applied to a collection that contains numbers

An Overview of OCL Constructs

Types and Boole algebra in OCL

- All OCL expressions are typed
  - `OclAny`:
    The type that includes all others. E.g. `x, y : OclAny`
  - `x = y`
    x and y are the same object.
  - `x <> y`
    `not (x = y).`
  - `x.oclType()`
    The type of x.
  - `x.isKindOf ( T )`
    True if T is a supertype (transitive) of the type of x.
  - `T.allInstances() :`
    Collection

    All the instances of type T.

- Boolean operators:
  - `b and b2, b or b2, b xor b2, not b`
    If any part of a Boolean expression fully determines the result, then it does not matter
    if some other parts of that expression have unknown or undefined results.
  - `b implies b2`
    True if b is false or if b is true and b2 is true.
  - `if b then e1 else e2 endif`
    If b is true the result is the value of e1; otherwise, the result is the value of e2.

Overview of Collection Valued Terms

- Size / aggregation:
  - `c->size()`: Integer
    Number of elements in the collection; for a bag or sequence, duplicates are counted as separate items.
  - `c->sum()`: Integer
    Sum of elements in the collection. Elements must be numbers
  - `c->count(e)`: Integer
    The number of times that e is in c.
  - `c->isEmpty()`: Boolean
    Same as `c->size() = 0`.
  - `c->notEmpty()`: Boolean
    Same as `not c->isEmpty()`.
- Equality
  - `c = c2` : Boolean
- Collection membership
  - `c->includes(e)`: Boolean;
    `c->exists ( x | x = e )`.
  - `c->excludes(e)`: Boolean;
    `not c->includes( e )`.
  - `c->includesAll(c2)`:
    Boolean;
    c includes all the elements in c2.
  - `c->including(e)`: Collection
    The collection that includes all of c as well as e.
  - `c->excluding(e)`: Collection
    The collection that includes all of c except e.

Overview of Collection Valued Terms

- Existential quantifier:
  - `c->exists(x | P )`: Boolean; there is at least one element in c, named x, for which predicate P is true.
  - Equivalent notation is: `c->exists(P)`, `c->exists(x:Type | P(x))`
- Universal quantifier:
  - `c->forAll(x | P)`: Boolean; for every element in c, named x, predicate P is true.
  - Equivalent notation is: `c->forAll(P)` `c->forAll(x:Type | P)`
- Selection:
  - `c->select(x | P)`: Collection The collection of elements in c for which P is true.
  - Equivalent is: `c->select(P)`
- Filtering:
  - `c->reject(x | P)`: Collection `c->select(x | not P)`.
  - Equivalent is: `c->reject(P)`
- Collection:
  - `c->collect(x | E)` : Bag The bag obtained by applying E to each element of c, named x.
  - `c.attribute` : Collection The collection(of type of c) consisting of the attribute of each element of c.

Sets, Bags, Sequences

Literals:

```
Set{ 1, 2, 5, 88 }
Set{ ’apple’, ’orange’,
   ’strawberry’}
Sequence{ 1, 3, 45, 2, 3 }
Sequence{ ’ape’, ’nut’ }
Bag{1, 3, 4, 3, 5 }
Sequence{ 1..(5+4) } =
Sequence{ 1.. 9 } =
Sequence{ 1, 2, 3, 4, 5, 6,
   7, 8, 9 }
```

Traditional operations are defined (union, intersection, etc.)

- Conversion from Collection:
  - `c->asSet()`: Set
    A set corresponding to the collection (duplicates are dropped, sequencing is lost).
  - `c->asSequence()`: Sequence
    A sequence corresponding to the collection.
  - `c->asBag()`: Bag
    A bag corresponding to the collection.
- Comments:
  - `--`

OCL – OBJECT CONSTRAINT LANGUAGE

MORGAN & CLAYPOOL PUBLISHERS

Model-Driven Software Engineering in Practice

Marco Brambilla
Jordi Cabot
Manuel Wimmer

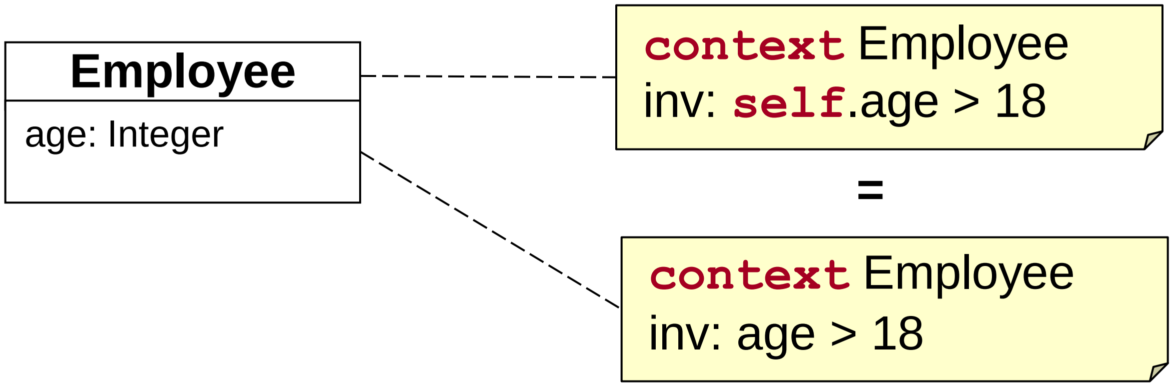
SYNTHESIS LECTURES ON SOFTWARE ENGINEERING

OCL Topics

- Introduction
- OCL Core Language
- OCL Standard Library
- Tool Support
- Examples

Design of OCL

- A context has to be assigned to each OCL-statement
  - **Starting address –** which model element is the OCL-statement defined for
  - Specifies which model elements can be reached using path expressions
- The context is specified by the keyword **`context`** followed by the name of the model element (mostly class names)
- The keyword **`self`** specifies the current instance, which will be evaluated by the invariant (context instance).
  - **`self`** can be omitted if the context instance is unique
- Example:



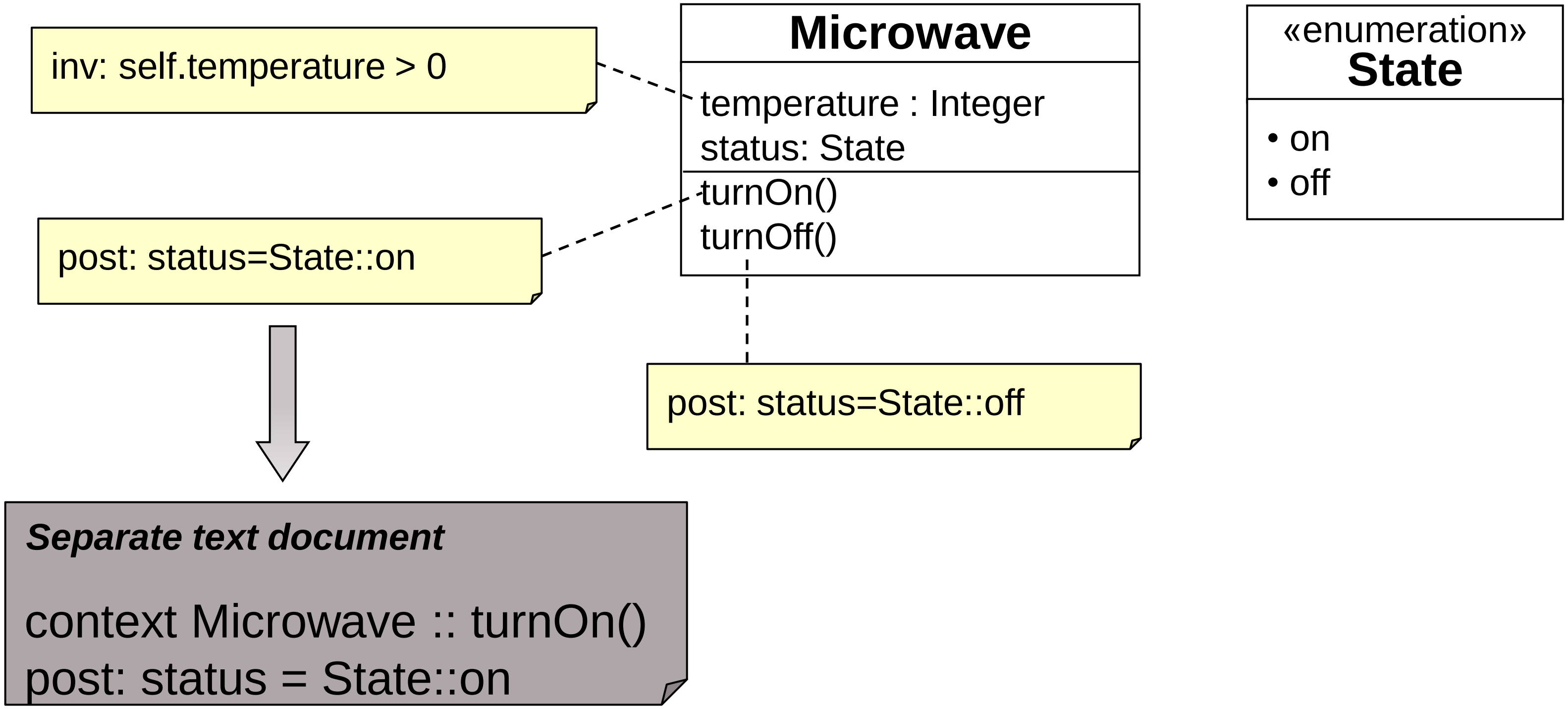
```
context Employee
inv: self.age > 18
```

=

```
context Employee
inv: age > 18
```

Design of OCL

- OCL can be specified in **two** different ways
  - As a comment **directly** in the class diagram (context described by connection)
  - Separate document file



Separate text document

```
context Microwave :: turnOn()
post: status = State::on
```

Types

- **OCL** is a typed language
  - Each **object, attribute,** and **result** of an operation or navigation is assigned to a **range of values** (type)

- **Predefined types**
  - **Basic types**
    - Simple types: *Integer, Real, Boolean, String*
    - OCL-specific types: *AnyType, TupleType, InvalidType, …*
  - **Set-valued, parameterized Types**
    - Abstract supertyp: Collection(T)
    - Set(T) – no duplicates
    - Bag(T) – duplicates allowed
    - Sequence(T) – Bag with ordered elements, association ends {*ordered*}
    - OrderedSet(T) – Set with ordered elements, association ends {*ordered, unique*}

- **Userdefined Types**
  - Instances of *Class* in MOF and indirect instances of *Classifier* in UML are types
  - *EnumerationType* – user defined set of values for defining constants

Types

Examples

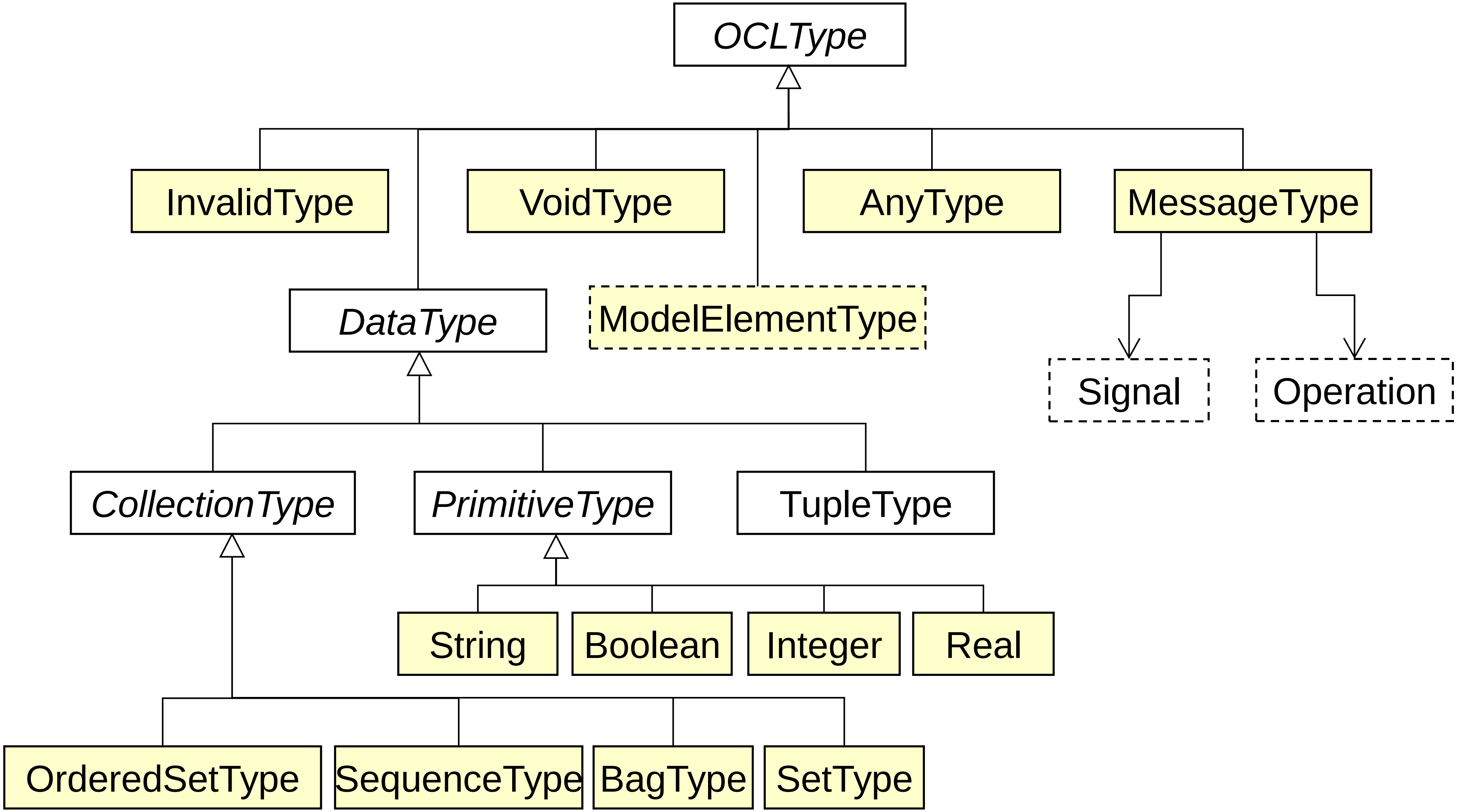
- **Basic types**
  - true, false : *Boolean*
  - -17, 0, 1, 2 : *Integer*
  - -17.89, 0.01, 3.14 : *Real*
  - “Hello World” : *String*

- **Set-valued, parameterized types**
  - Set{ Set{1}, Set{2, 3} } : *Set(Set(Integer))*
  - Bag{ 1, 2.0, 2, 3.0, 3.0, 3 } : *Bag(Real)*
  - Tuple{ x = 5, y = false } : *Tuple{x: Integer, y : Boolean}*

- **Userdefined types**
  - Passenger : *Class*, Flight : *Class*, Provider : *Interface*
  - Status::started - enum Status {started, landed}

Types

OCL meta model (extract)



Expressions

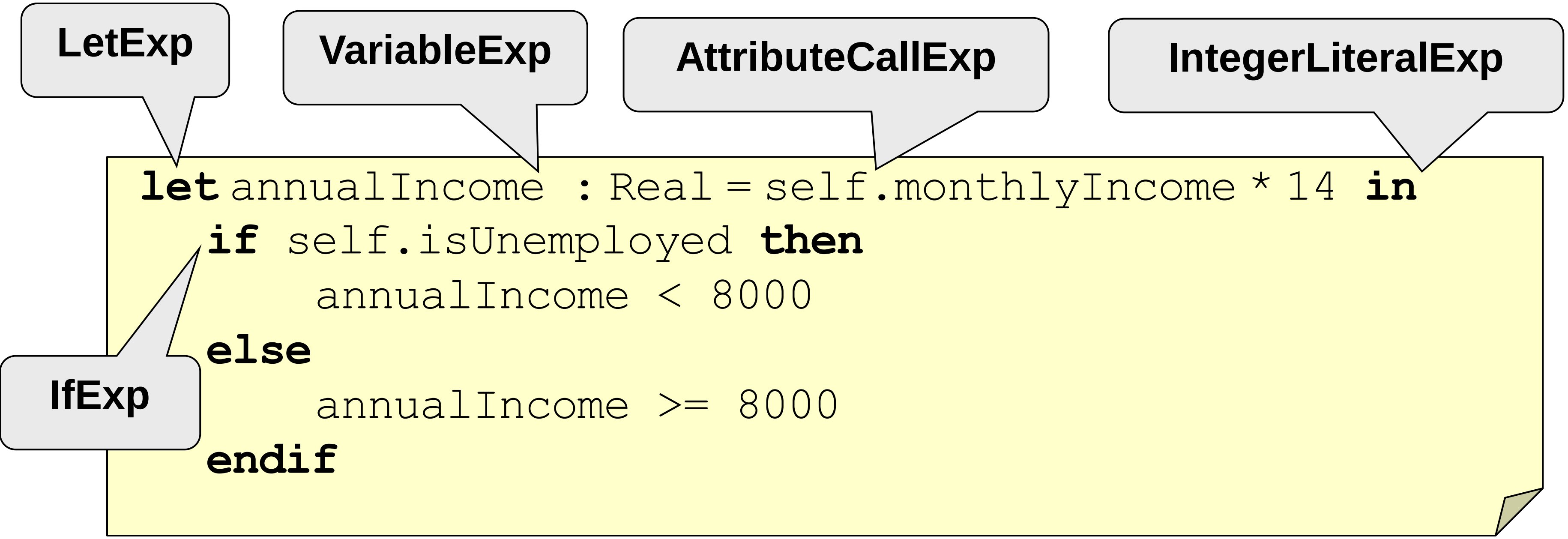
- Each OCL expression is an indirect instance of *OCLExpression*
  - Calculated in certain environment – cf. context
  - Each OCL expression has a **typed return value**
  - **OCL Constraint is an OCL expression with return value Boolean**

- **Simple OCL expressions**
  - *LiteralExp, IfExp, LetExp, VariableExp, LoopExp*

- **OCL expressions for querying model information**
  - *FeatureCallExp* – abstract superclass
  - *AttributeCallExp* – querying attributes
  - *AssociationEndCallExp* – querying association ends
    - Using role names; if no role names are specified, lowercase class names have to be used (if unique)
  - *AssociationClassCallExp* – querying association class (only in UML)
  - *OperationCallExp* – Call of query operations
    - Calculate a value, but do **not** change the system state!

Expressions

- Examples for *LiteralExp, IfExp, VariableExp, AttributeCallExp*

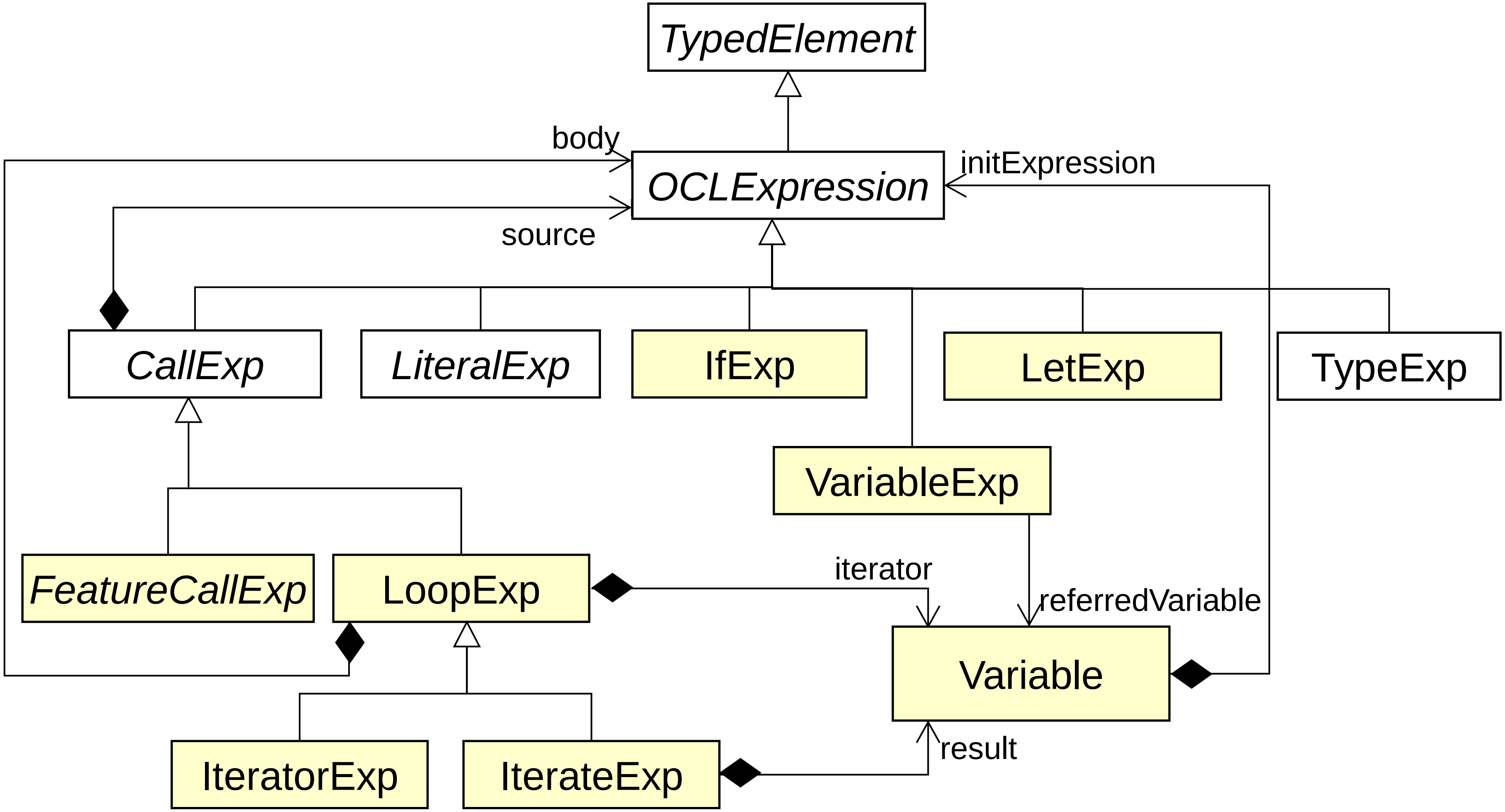


```
let annualIncome : Real = self.monthlyIncome * 14 in
   if self.isUnemployed then
        annualIncome < 8000
   else
        annualIncome >= 8000
   endif
```

- **Abstract syntax** of OCL is described as **meta model**
- **Mapping from abstract syntax to concrete syntax**
  - *IfExp -> **if** Expression **then** Expression **else** Expression **endif***

Expressions

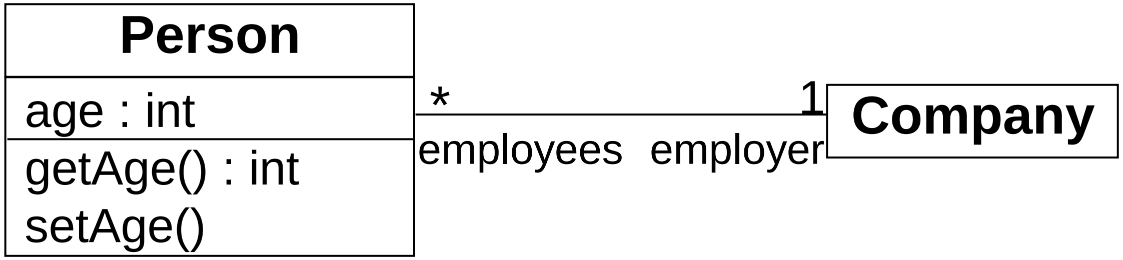
OCL meta model (extract)



LiteralExp: *CollectionLiteralExp*, *PrimitiveLiteralExp*, *TupleLiteralExp, EnumLiteralExp*

Query of model information

- Context instance
  - `context` **`Person`**



- AttributeCallExp
  - `self.age : int`
- OperationCallExp
  - Operations must not have **side effects**
  - Allowed: `self.getAge() : int`
  - **Not allowed**: `self.setAge()`
- AssociationEndCallExp
  - Navigate to the opposite association end using role names

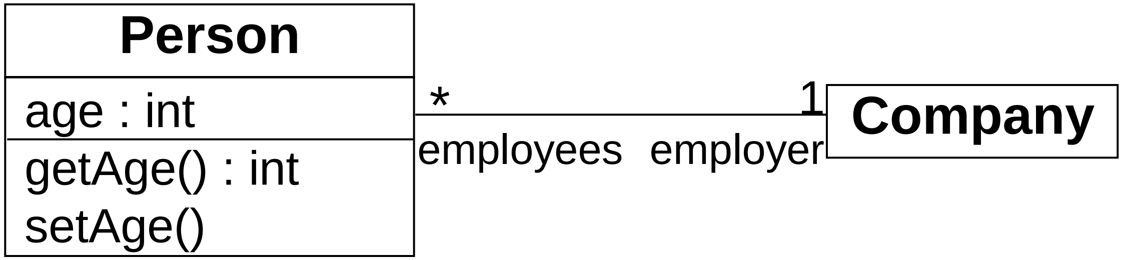
    `self.employer` – Return value is of type ***Company***
  - Navigation often results into a set of objects – Example

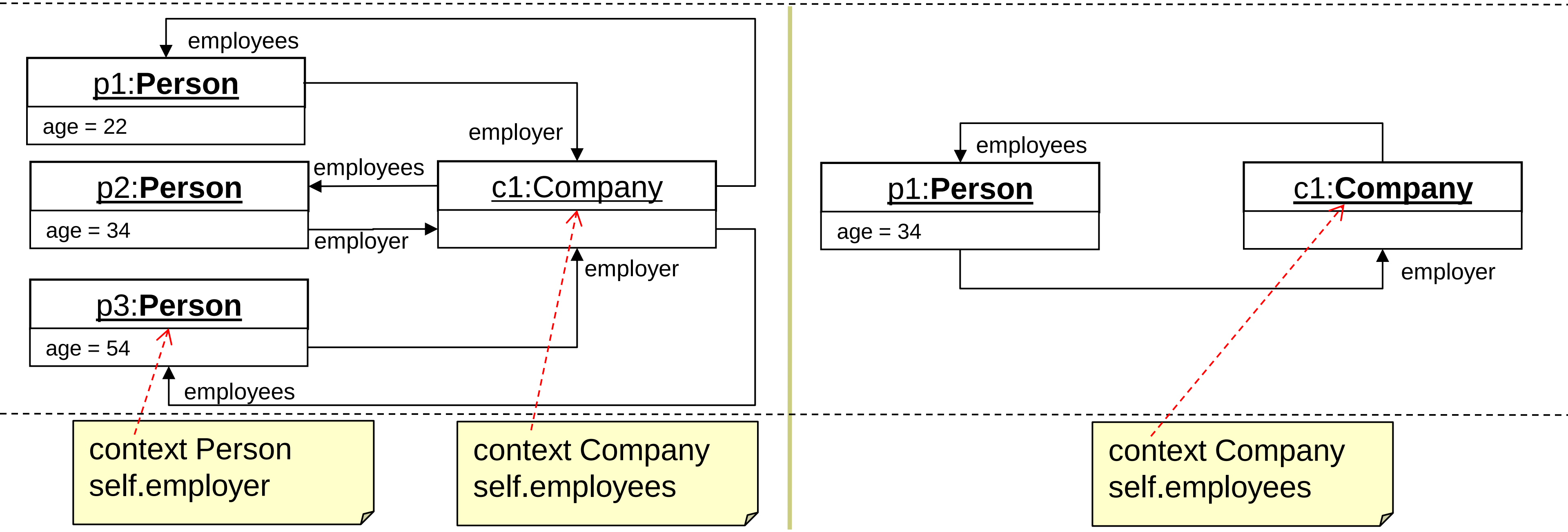
    `context` **`Company`**

    `self.employees` – Return value is of type **Set (Person)**

Query of model information

Example





context Person
self.employer

c1 : Company

context Company
self.employees

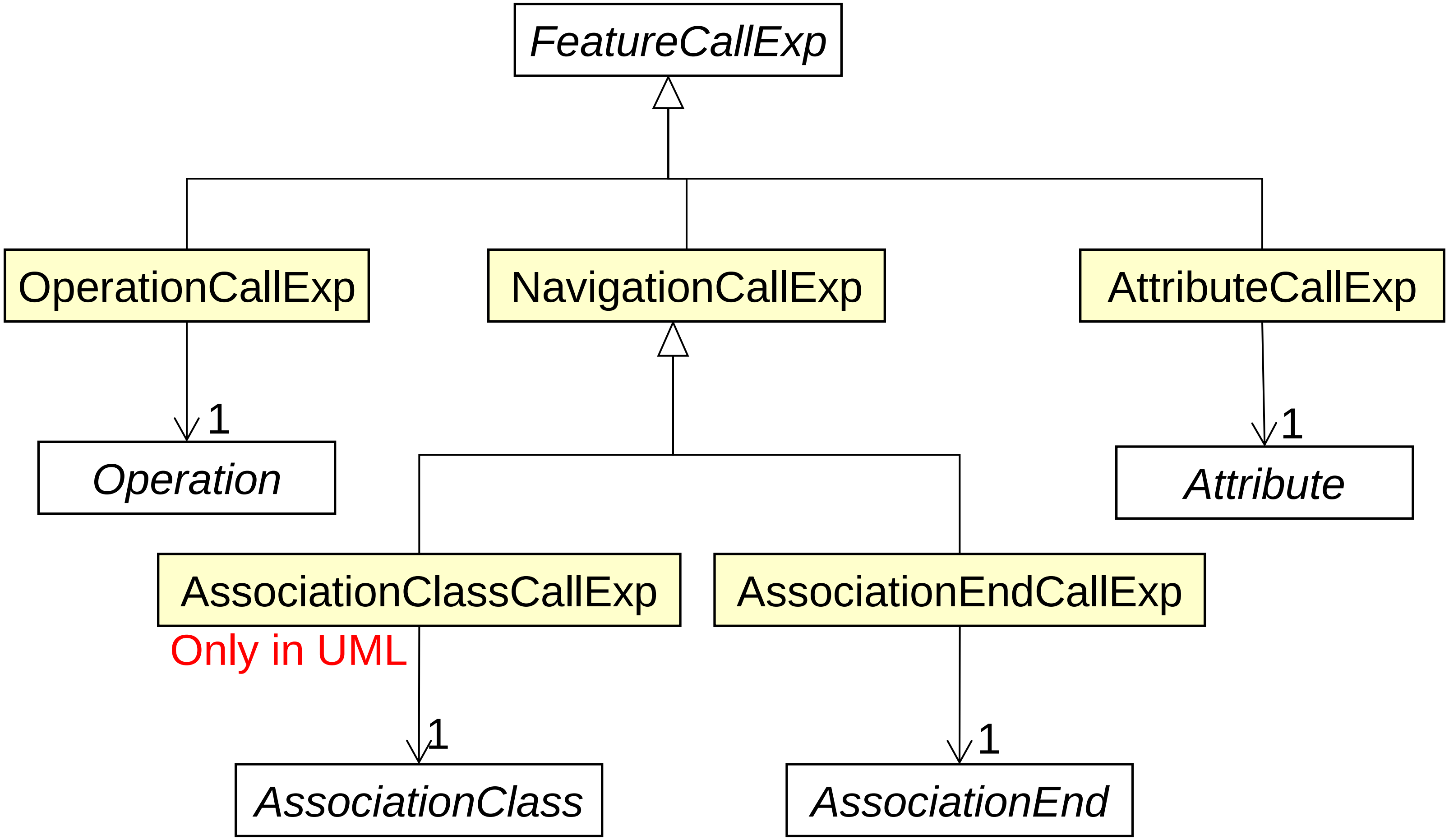
Set{p1,p2,p3} :
Set(Person)

context Company
self.employees

Set{p1} :
Set(Person)

Query of model information

OCL meta model (extract)



Only in UML

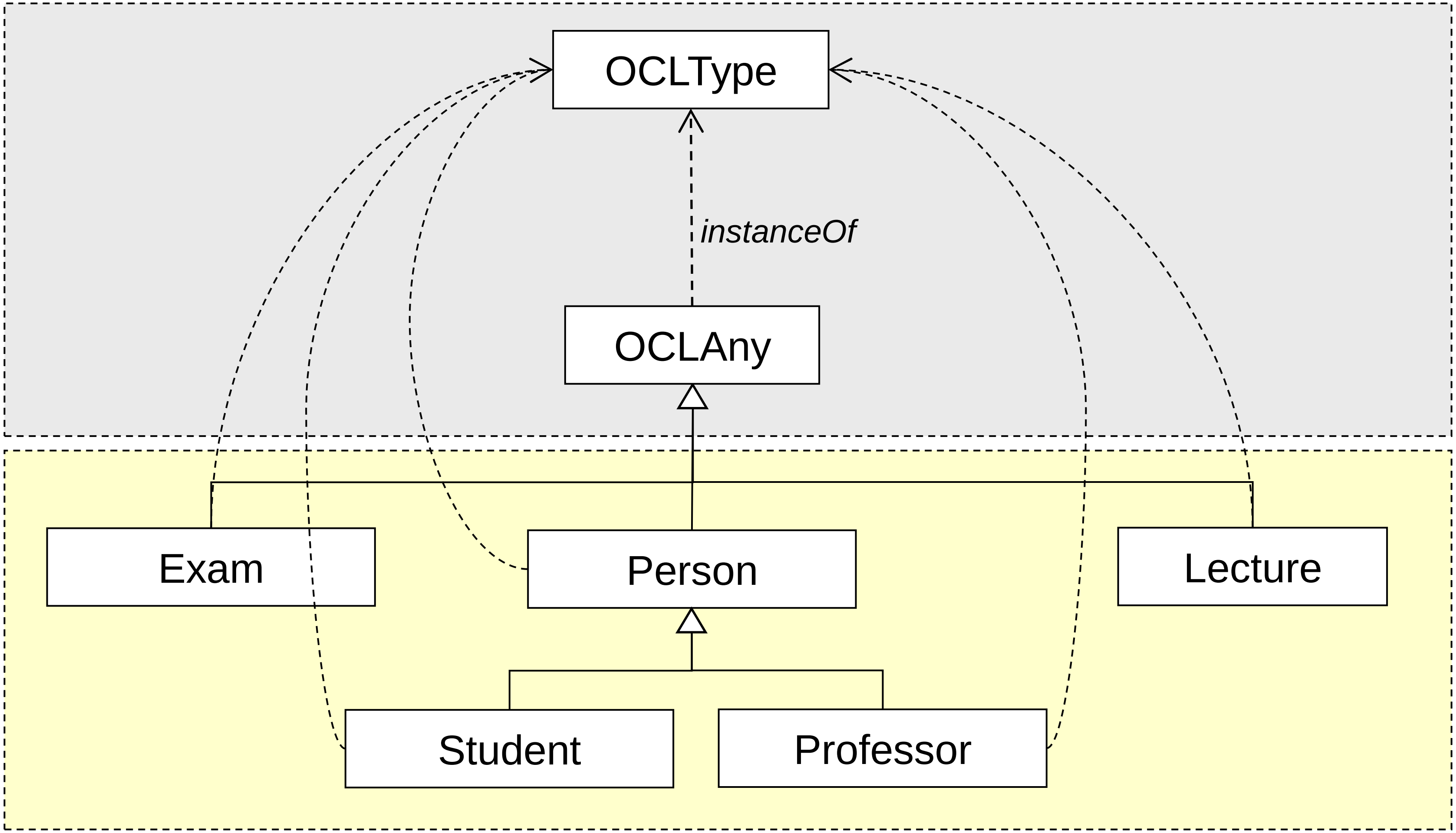
OCL Library: Operations for OclAny

- *OclAny* - **Supertype** of all other types in OCL
  - **Operations** are **inherited** by all other types.
- **Operations** of *OclAny* (extract)
  - Receiving object is denoted by *obj*

Operation	Explanation of result
=(obj2:OclAny):Boolean	True, if *obj2* and *obj* reference the same object
oclIsTypeOf(type:OclType):Boolean	True, if *type* is the type of *obj*
oclIsKindOf(type:OclType): Boolean	True, if *type* is a direct or indirect supertype or the type of *obj*
oclAsType(type:Ocltype): Type	The result is *obj* of type *type*, or *undefined*, if the current type of *obj* is not *type* or a direct or indirect subtype of it (casting)

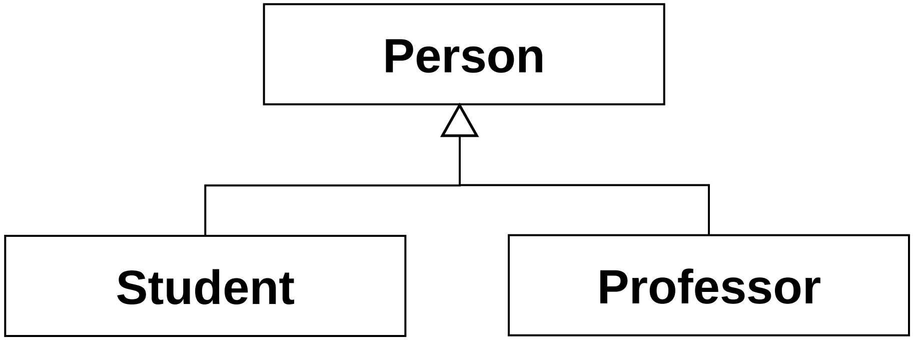
Operations for OclAny

Predefined environment for model types



Operations for OclAny

- ***oclIsKindOf vs. oclIsTypeOf***



context **Person**
self.oclIsKindOf(Person) : *true*
self.oclIsTypeOf(Person) : *true*
self.oclIsKindOf(Student) : *false*
self.oclIsTypeOf(Student) : *false*

context **Student**
self.oclIsKindOf(Person) : *true*
self.oclIsTypeOf(Person) : *false*
self.oclIsKindOf(Student) : *true*
self.oclIsTypeOf(Student) : *true*
self.oclIsKindOf(Professor) : *false*
self.oclIsTypeOf(Professor) : *false*

Operations for simple types

- **Predefined** simple types
  - Integer {Z}
  - Real {R}
  - Boolean {true, false}
  - String {ASCII, Unicode}

- Each simple type has predefined operations

Simple type	**Predefined operations**
Integer	*, +, -, /, abs(), …
Real	*, +, -, /, floor(), …
Boolean	and, or, xor, not, implies
String	concat(), size(), substring(), …

Operations for simple types

- Syntax
  - v.operation(para1, para2, …)
    - Example: “bla”.**concat**(“bla”)
  - Operations without brackets (Infix notation)
    - Example: 1 **+** 2, true **and** false

Signature	**Operation**
Integer **X** *Integer* → *Integer*	{+, -, *}
t1 **X** *t2* → *Boolean*	{<,>,≤,≥}, *t1, t2* typeOf {Integer or Real}
Boolean **X** *Boolean* → *Boolean*	{and, or, xor, implies}

Operations for simple types

Boolean operations - semantic

- OCL is based on a **three-valued (trivalent) logic**
  - Expressions are mapped to the three values {true, false, undefined}
- Semantic of the operations
  - $\mathcal{M}$(I, *exp*)= *I(exp*), if exp not further resolvable
  - $\mathcal{M}$(I, **not** *exp*)= ¬$\mathcal{M}$ (I, *exp*)
  - $\mathcal{M}$(I,(*exp1* **and** *exp2*)) = $\mathcal{M}$(I, *exp1*) $\wedge$ $\mathcal{M}$(I, *exp2*)
  - $\mathcal{M}$(I,(*exp1* **or** *exp2*)) = $\mathcal{M}$(I, *exp1*) $\vee$ $\mathcal{M}$(I, *exp2*)
  - $\mathcal{M}$(I,(*exp1* **implies** *exp2*)) = $\mathcal{M}$(I, *exp1*) $\rightarrow$ $\mathcal{M}$(I, *exp2)*

Undefined: Return value if an expression fails

1. Access on the first element of an empty set
2. Error during *Type Casting*
3. …

- Truth table: true(1), false (0),undefined (?)

¬	
0	1
1	0
?	?

$\wedge$	0	1	?
0	0	0	0
1	0	1	?
?	0	?	?

$\vee$	0	1	?
0	0	1	?
1	1	1	1
?	?	1	?

$\rightarrow$	0	1	?
0	1	1	1
1	0	1	?
?	?	1	?

Operations for simple types

Boolean operations - semantic

- Simple example for an **undefined** OCL expression
  - *1/0*

- **Query** if undefined– OCLAny.oclIsUndefined()
  - *(1 / 0).oclIsUndefined() : true*

- Examples for the evaluation of Boolean operations
  - *(1/0 = 0.0)* ***and*** *false : false*
  - *(1/0 = 0.0)* ***or*** *true : true*
  - *false* ***implies*** *(1.0 = 0.0) : true*
  - *(1/0 = 0.0) implies true : true*

Operations for collections

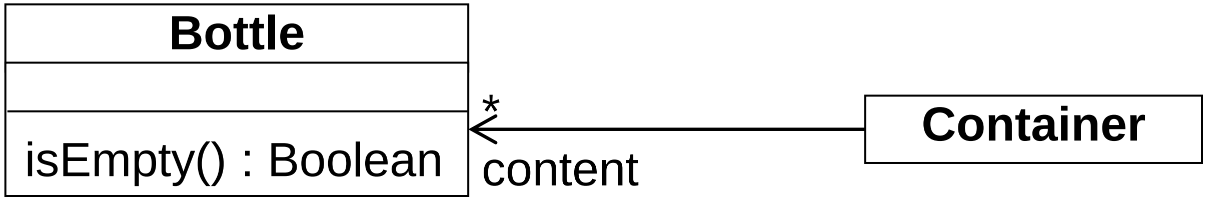
- Collection is an **abstract supertype** for all set types
  - Specification of the **mutual** operations
  - *Set, Bag, Sequence, OrderedSet* inherit these operations
- **Caution:** Operations with a return value of a set-valued type create a new collection (no side effects)
- Syntax: v -> op(…) – Example: {1, 2, 3} -> size()

- Operations of collections (extract)
  - Receiving object is denoted by *coll*

Operation	**Explanation of result**
size():Integer	Number of elements in *coll*
includes(obj:OclAny):Boolean	True, if *obj* exists in *coll*
isEmpty:Boolean	True, if *coll* contains no elements
sum:T	Sum of all elements in *coll* Elements have to be of type Integer or Real

Operations for collections

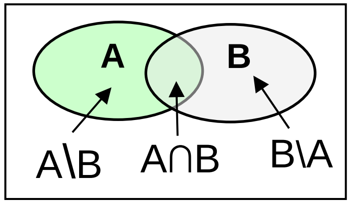
- **Model operations vs. OCL operations**



OCL-Constraint	***Semantic***
context Container inv: self.content -> first().isEmpty()	Operation *isEmpty()* always has to return true
context Container inv: self.content -> isEmpty()	Container instances must not contain bottles

Operationen for Set/Bag

- *Set* and *Bag* define additional operations
  - Generally based on **theory of set concepts**



- **Operations** of **Set** (extract)
  - Receiving object is denoted by set

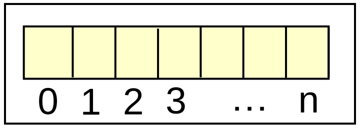
Operation	Explanation of result
union(set2:Set(T)):Set(T)	Union of *set* and *set2*
intersection(set2:Set(T)):Set(T)	Intersection of *set* and *set2*
difference(set2:Set(T)):Set()	Difference set; elements of *set*, which do not consist in *set2*
symmetricDifference(set2:Set(T)): Set(T)	Set of all elements, which are either in *set* or in *set2*, but do not exist in both sets at the same time

- **Operations** of **Bag** (extract)
  - Receiving object is denoted by bag

Operation	Explanation of result
union(bag2:Bag(T)):Bag(T)	Union of *bag* and *bag2*
intersection(bag2:Bag(T)): Bag(T)	Intersection of *bag* and *bag2*

Operations for OrderedSet/Sequence

- *OrderedSet* and *Sequences* define additional operations
  - Allow access or modification through an **Index**



- **Operations** of **OrderedSet** (extract)
  - Receiving object is denoted by *orderedSet*

Operation	Explanation of result
first:T	First element of *orderedSet*
last:T	Last element of *orderedSet*
at(i:Integer):T	Element on index i of *orderedSet*
subOrderedSet(lower:Integer, upper:Integer):OrderedSet(T)	Subset of *orderedSet,* all elements of *orderedSet* including the element on position *lower* and the element on position *upper*
insertAt(index:Integer,object:T) :OrderedSet(T)	Result is a copy of the *orderedSet*, including the element *object* at the position *index*

- **Operations** of **Sequence**
  - Analogous to the operations of OrderedSet

Iterator-based operations

- OCL defines operations for *Collections* using *Iterators*
  - Expression Package: LoopExp
  - **Projection** of new *Collections* out of existing ones
  - Compact **declarative specification** instead of imperative algorithms

- Predefined Operations
  - select(exp) : *Collection*
  - reject(exp) : *Collection*
  - collect(exp) : *Collection*
  - forAll(exp) : *Boolean*
  - exists(exp) : *Boolean*
  - isUnique(exp) : *Boolean*

- iterate(…) – Iterate over all elements of a *Collection*
  - Generic operation
  - Predefined operations are defined with iterate(…)

Iterator-based operations

Select-/Reject-Operation

- **Select** and **Reject** return subsets of collections
  - Iterate over the complete collection and collect elements
- Select
  - **Result**: Subset of collection, including elements where *booleanExpr* is **true**

```
collection -> select( v : Type | booleanExp(v) )
collection -> select( v | booleanExp(v) )
collection -> select( booleanExp )
```

- Reject
  - **Result**: Subset of collection, including elements where *booleanExpr* is **false**
  - Just *Syntactic* Sugar, because each *reject-Operation* can be defined as a *select-Operation* with a negated expression

```
collection-> reject(v : Type | booleanExp(v))
```

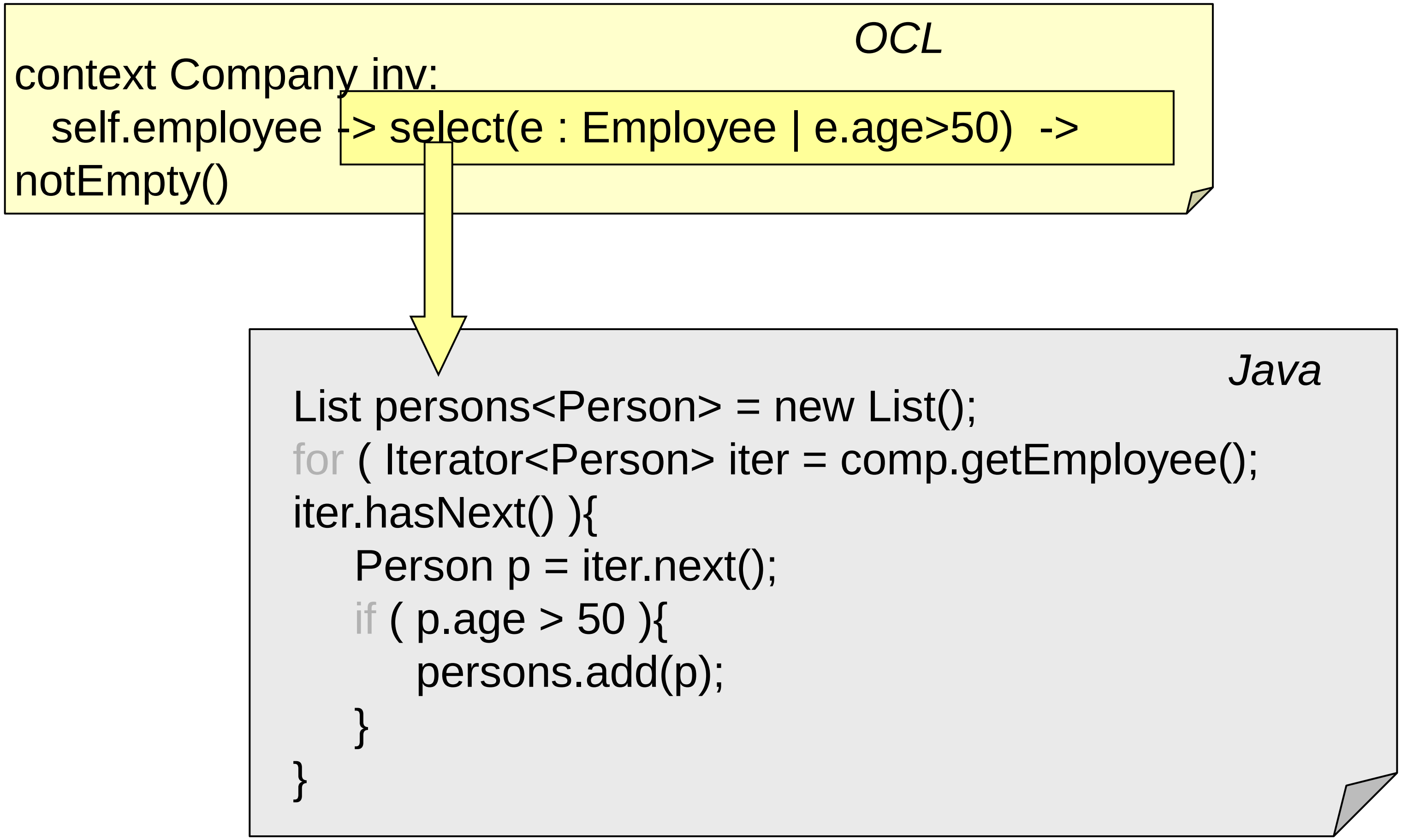
=

```
collection-> select(v : Type | not (booleanExp(v))
```

Iterator-based operations

Select-/Reject-Operation

- Semantic of the *Select-Operation*



```
context Company inv:
   self.employee -> select(e : Employee | e.age>50)  ->
notEmpty()
```

```
List persons<Person> = new List();
for ( Iterator<Person> iter = comp.getEmployee();
iter.hasNext() ){
     Person p = iter.next();
     if ( p.age > 50 ){
          persons.add(p);
     }
}
```

Iterator-based operations

Collect-Operation

- *Collect-Operation* returns a new collection from an existing one. It collects the **Properties** of the objects and not the objects itself.
  - Result of *collect* always **Bag<T>.T** defines the type of the property to be collected

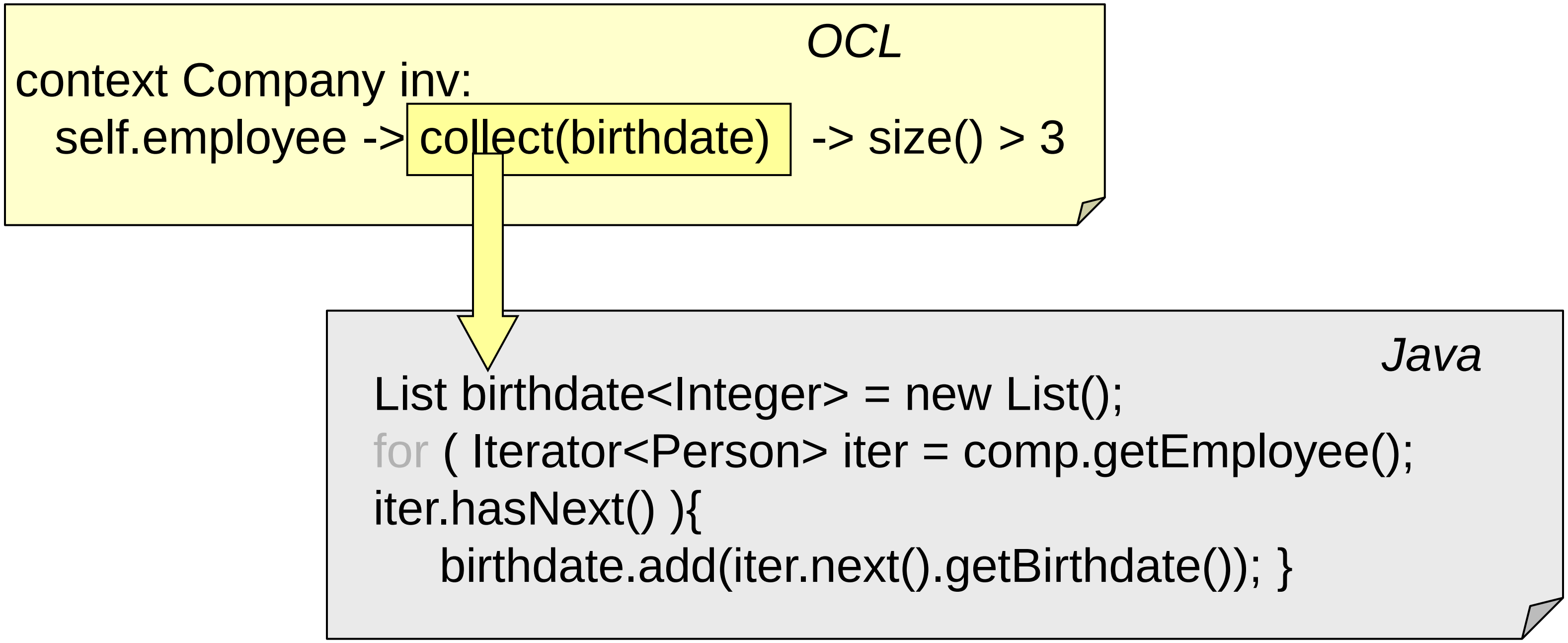
```
collection -> collect( v : Type | exp(v) )
collection -> collect( v | exp(v) )
collection -> collect( exp )
```

- Example
  - *self.employees -> collect(age)* – Return type: Bag(Integer)
- Short notation for collect
  - *self.employees.age*

Iterator-based operations

Collect-Operation

- Semantic of the *Collect-Operator*



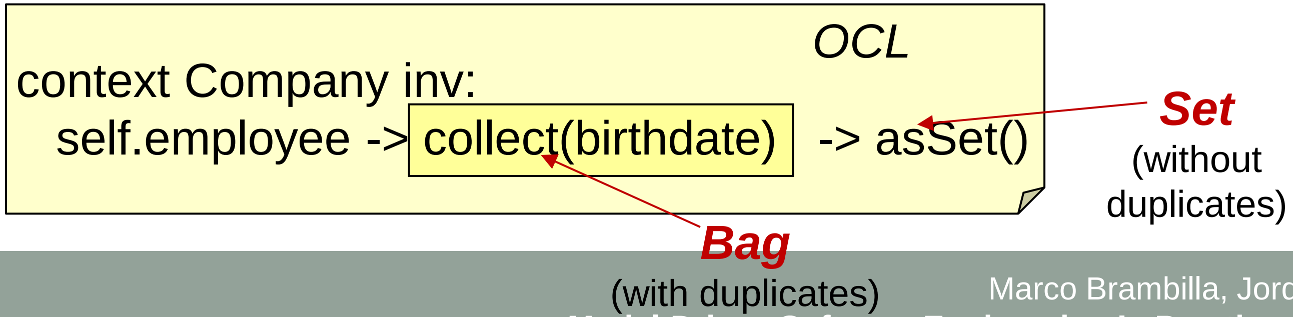
OCL

```
context Company inv:
   self.employee -> collect(birthdate)  -> size() > 3
```

Java

```
List birthdate<Integer> = new List();
for ( Iterator<Person> iter = comp.getEmployee();
iter.hasNext() ){
     birthdate.add(iter.next().getBirthdate()); }
```

- Use of *asSet()* to eliminate duplicates



OCL

```
context Company inv:
   self.employee -> collect(birthdate)  -> asSet()
```

Set
(without duplicates)

Bag
(with duplicates)

Iterator-based operations

ForAll-/Exists-Operation

- **ForAll** checks, if all elements of a collection evaluate to true

```
collection -> forAll( v : Type | booleanExp(v) )
collection -> forAll( v | booleanExp(v) )
collection -> forAll( booleanExp )
```

- **Example:** self.employees -> forAll(age > 18)

- **Nesting** of forAll-Calls (*Cartesian Product*)

```
context Company inv:
self.employee->forAll (e1 | self.employee -> forAll (e2 |
        e1 <> e2 implies e1.svnr <> e2.svnr))
```

- ***Alternative***: Use of multiple iterators

```
context Company inv:
self.employee -> forAll (e1, e2 | e1 <> e2 implies e1.svnr <> e2.svnr))
```

- **Exists** checks, if at least one element evaluates to true
  - Beispiel: employees -> exists(e: Employee | e.isManager = true)

Iterator-based operations

Iterate-Operation

- **Iterate** is the generic form of all iterator-based operations

- **Syntax**
  collection -> iterate(**elem** : Typ; **acc** : Typ =
  <initExp> | **exp(elem**, **acc**))
  - Variable **elem** is a typed ***Iterator***
  - Variable **acc** is a typed ***Accumulator***
    - Gets assigned initial value initExp
  - **exp(elem, acc)** is a function to calculate **acc**

- **Example**

  collection -> collect(x : T | x.property)

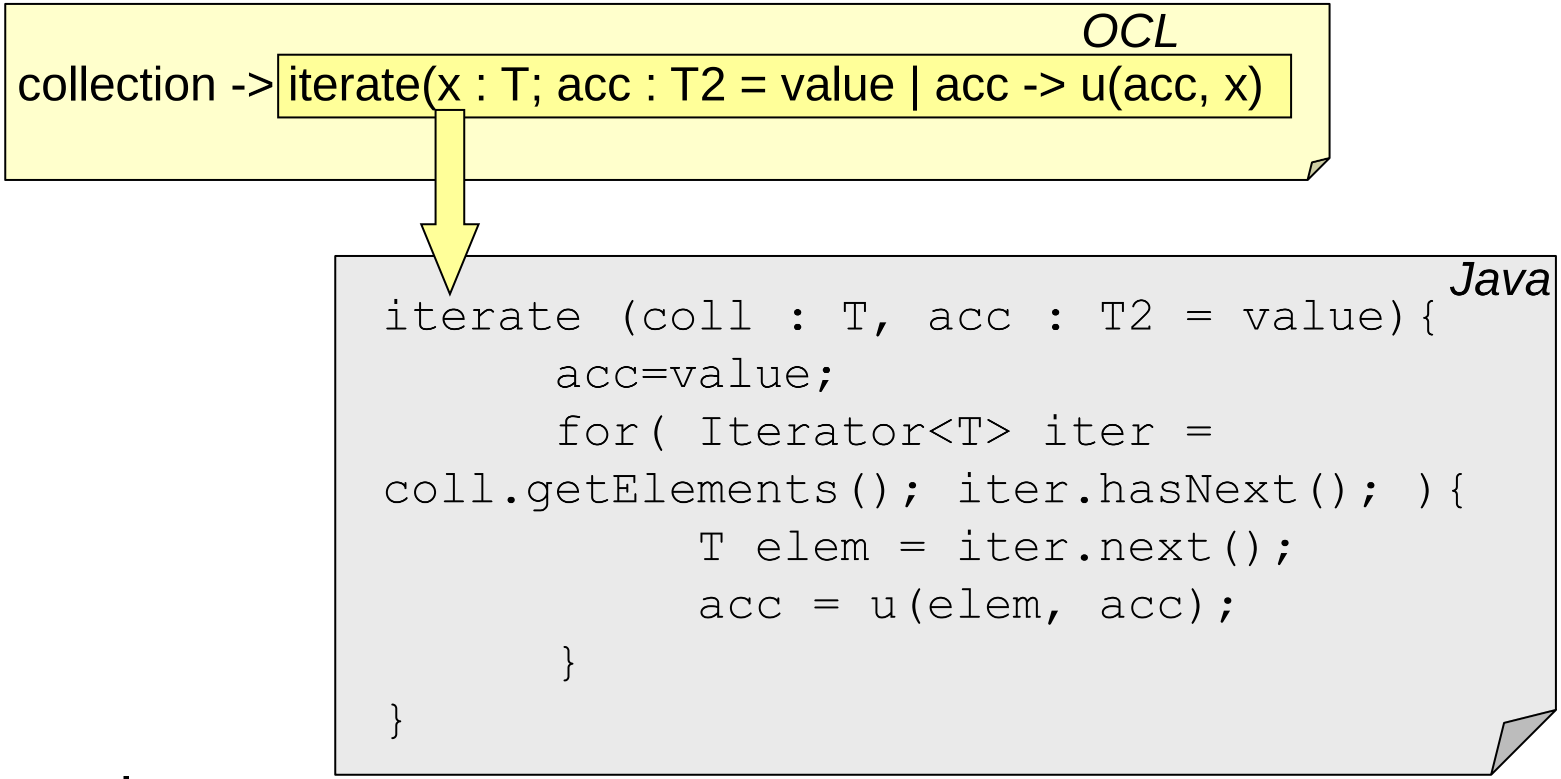
  -- semantically equivalent to:

  collection -> iterate(x : T; acc : T2 = Bag{} | acc -> including(x.property))

Iterator-based operations

Iterate-Operator

- Semantic of the *Iterate-Operator*



collection -> iterate(x : T; acc : T2 = value | acc -> u(acc, x)

```
iterate (coll : T, acc : T2 = value){
      acc=value;
      for( Iterator<T> iter =
coll.getElements(); iter.hasNext(); ){
           T elem = iter.next();
           acc = u(elem, acc);
      }
}
```

- Example
  - Set{1, 2, 3} -> iterate(i:Integer, a:Integer=0 | a+i)
  - Result: 6

Tool Support

- **Wishlist**
  - Syntactic analysis: Editor support
  - Validation of logical consistency (Unambiguous)
  - Dynamic validation of invariants
  - Dynamic validation of Pre-/Post-conditions
  - Code generation and test automation

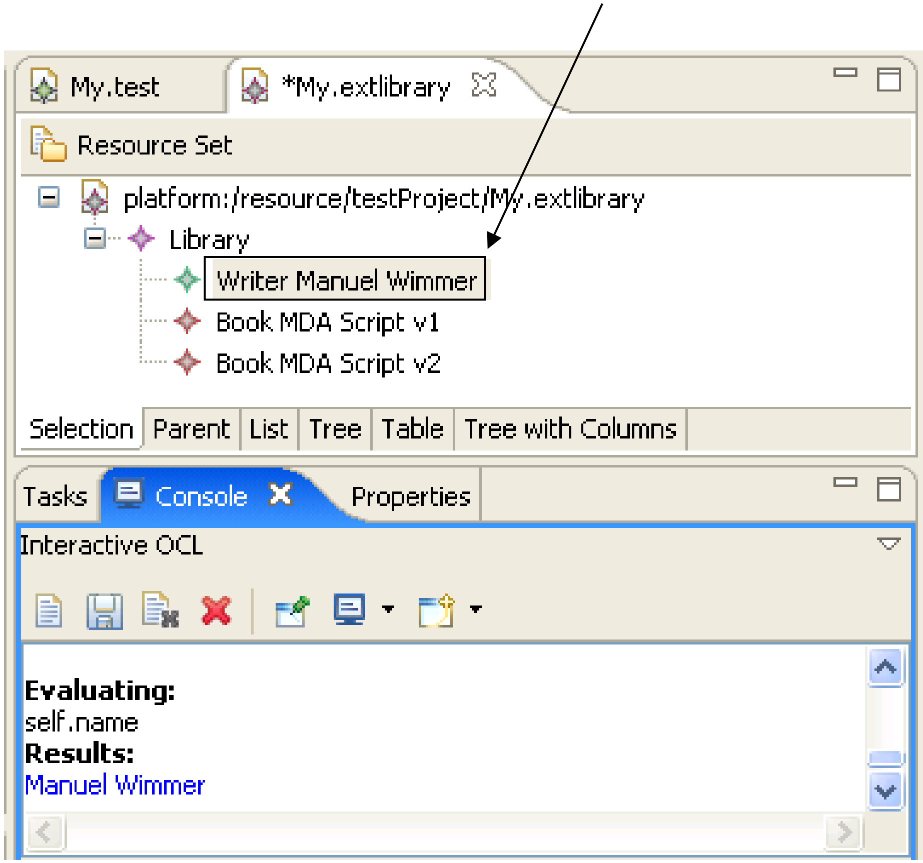
- **Today**
  - UML-tools provide OCL-editors
  - MDA-tools provide code generation of OCL-expressions
  - Meta modeling platforms provide the opportunity to define OCL Constraints for meta models.
    - The editor should dynamically check constraints or restrict modeling, respectively.

OCL Tools

- Some OCL-parsers, which check the syntax of OCL-constraints and apply them to the models, are for free.
  - IBM Parser
- Dresden OCL Toolkit 2.0
  - Generation of Java code out of OCL-constraints
  - Possible integration with ArgoUML
- OCL-frameworks are originated in the areas of EMF and the UML2 project of Eclipse
  - Octopus
  - Frauenhofer Toolkit
  - OSLO
  - EMFT OCL-Framework/Query-Framework

OCL-Tools

- **EMFT OCL-Framework**
  - Based on EMF
  - *OCL-API* – Enables the use of OCL in Java programs
  - *Interactive OCL Console* – Enables the definition and evaluation of OCL-constraints
- **EMFT Query-Framework**
  - **Goal**: SQL-like query of model information
  - **select** exp **from** exp **where** *oclExp*

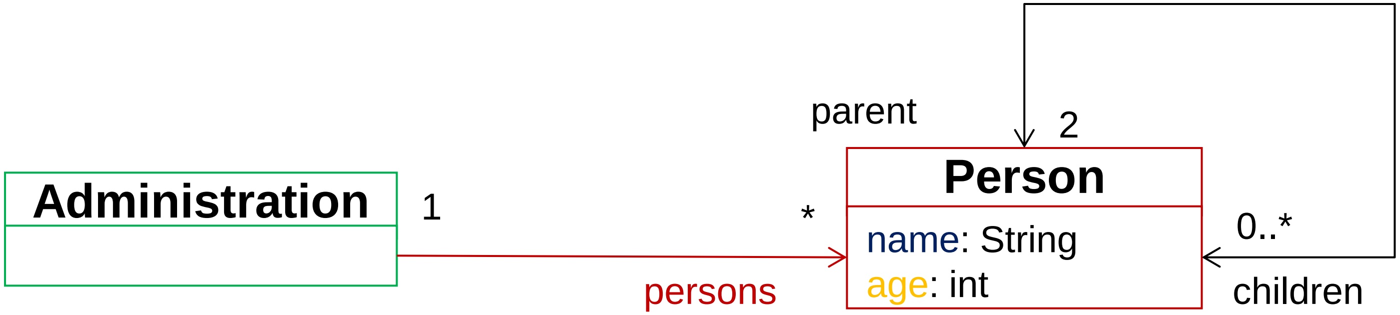


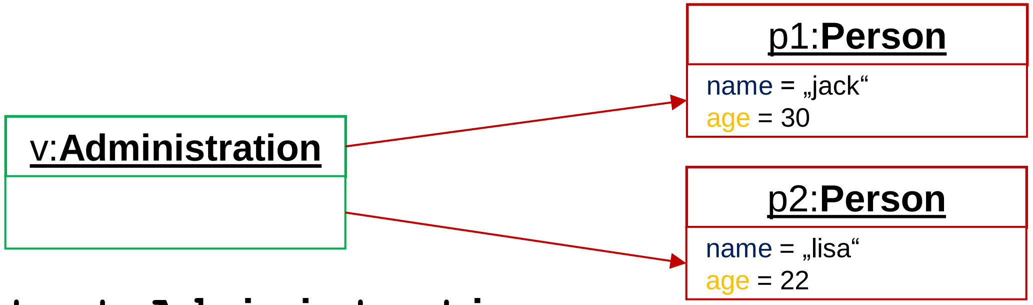
Context

```
Evaluating:
self.name
Results:
Manuel Wimmer
```

TUWEL: Interactive OCL Console Screencast

Example 1: Navigation (1)

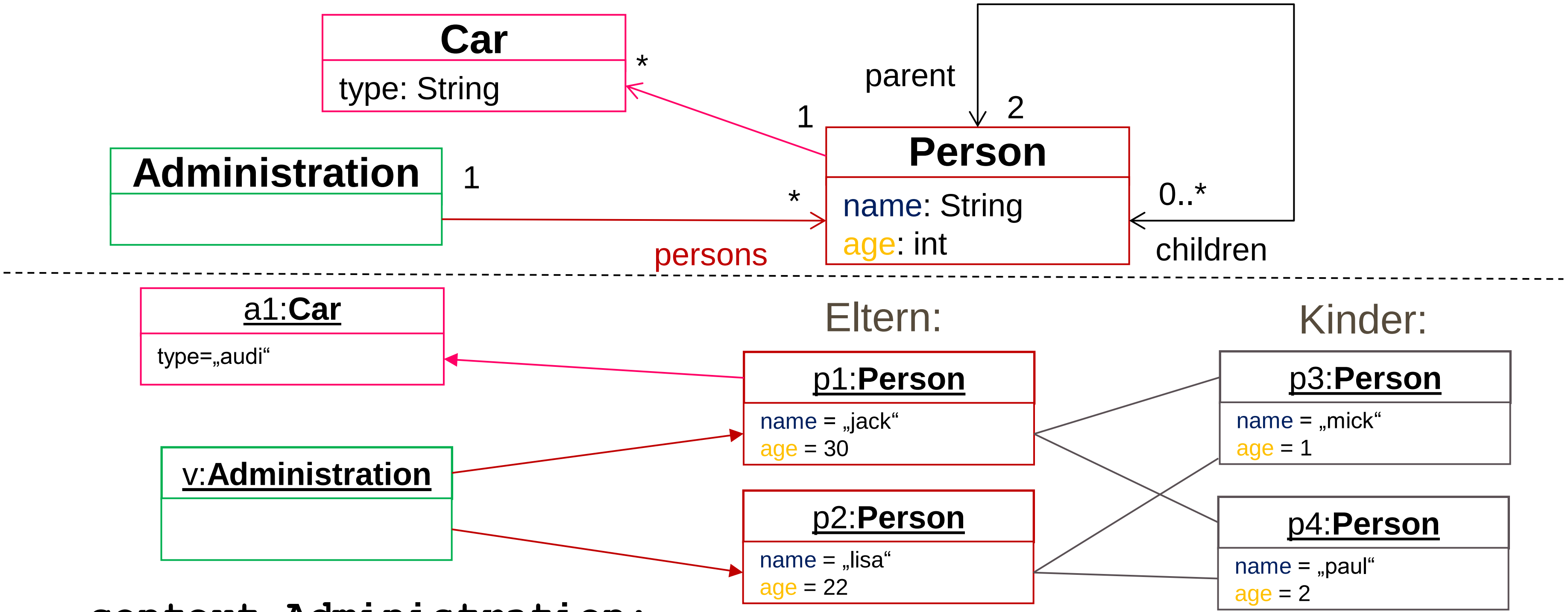




context Administration:

- `self.persons` → {Person p1, Person p2}
- `self.persons.name` → {jack, lisa}
- `self.persons.alter` → {30, 22}

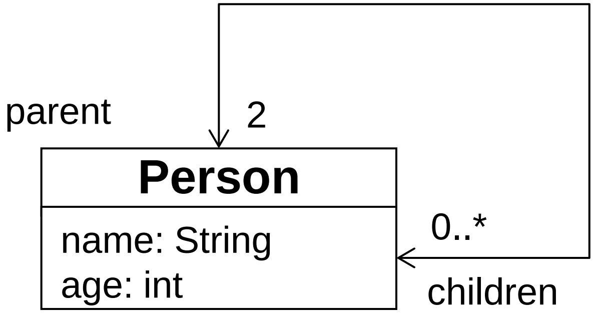
Example 1: Navigation (2)



context Administration:

- `self.persons.children` → {{p3, p4}, {p3, p4}}
- `self.persons.children.parent` → {{{p1, p2}, {p1, p2}}, ...}
- `self.persons.car.type` → {{"audi"}}

Example 2: Invariant (1)



parent

2

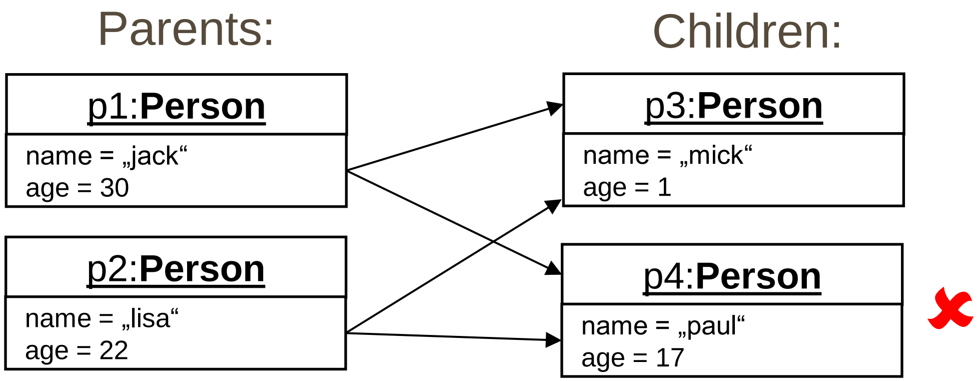
Person

name: String
age: int

0..*

children

Constraint: A child is at least 15 years younger than his parents.



Parents:

Children:

p1:**Person**

name = „jack“
age = 30

p2:**Person**

name = „lisa“
age = 22

p3:**Person**

name = „mick“
age = 1

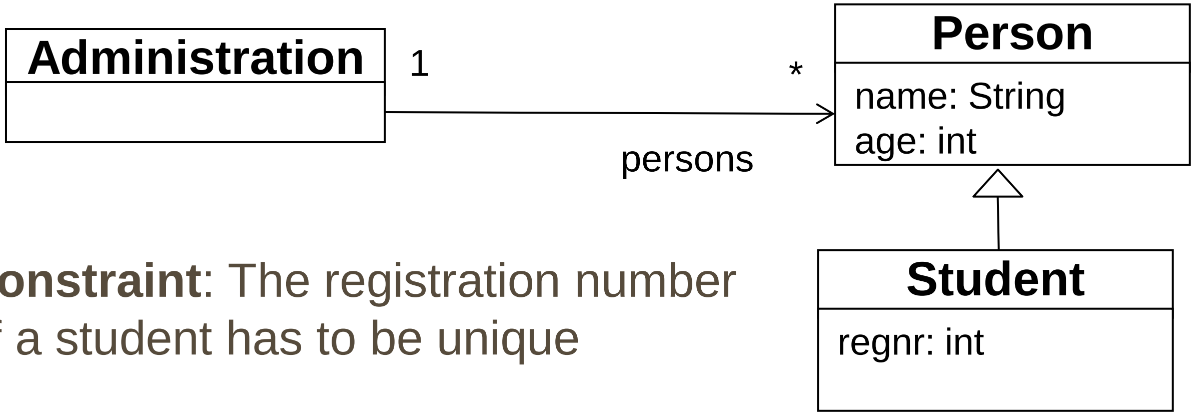
p4:**Person**

name = „paul“
age = 17

✘

```
context Person
inv: self.children->forAll(k : Person | k.age
 < self.age-15)
```

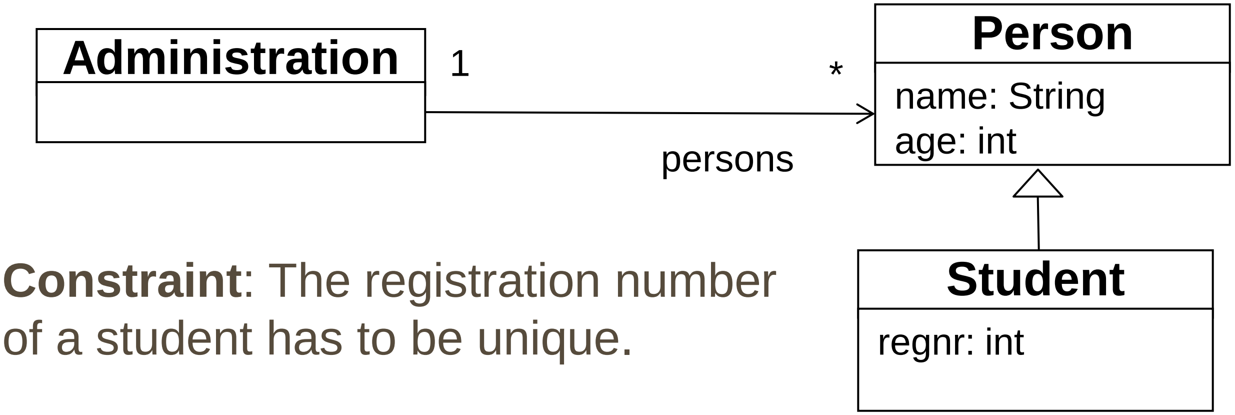
Example 2: Invariant (2)



Constraint: The registration number of a student has to be unique

```
context Administration
inv uniqueRegnr :
  self.persons -> select(e : Person| e.oclIsTypeOf(Student))
                           -> forAll(e1 |
   self.persons -> select(e : Person | e.oclIsTypeOf(Student))
                           -> forAll(e2 |
  e1 <> e2 implies   e1.oclAsType(Student).regnr       <>
                                    e2.oclAsType(Student).regnr))
```

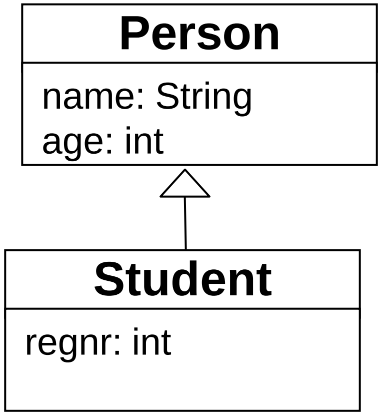
Example 2: Invariant (2) cont.



Constraint: The registration number of a student has to be unique.

```
context Administration
inv uniqueRegnr :
 self.persons -> select(e : Person| e.oclIsTypeOf(Student))
      -> forAll(e1, e1 | e1 <> e2 implies
      e1.oclAsType(Student).regnr <>
      e2.oclAsType(Student).regnr)
      )
```

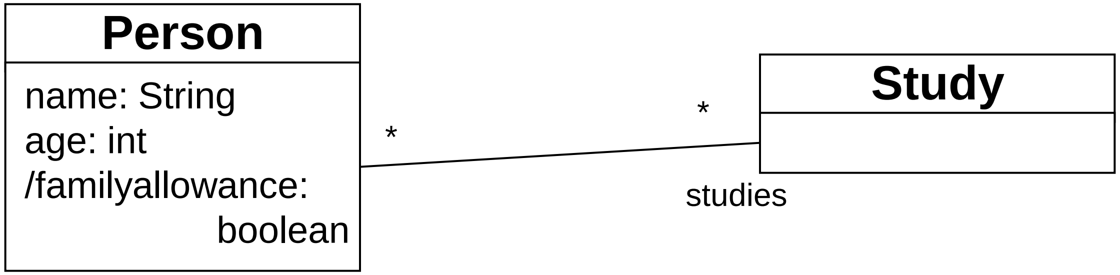
Example 2: Invariant (2) cont.



Constraint: The registration number of a student has to be unique.

```
context Student
inv uniqueRegnr :
 Student.allInstances() -> forAll(e1, e1 | e1 <> e2 implies
      e1.oclAsType(Student).regnr <>
      e2.oclAsType(Student).regnr))
```

Example 3: Inherited attribute



Person

name: String
age: int
/familyallowance:
boolean

Study

*

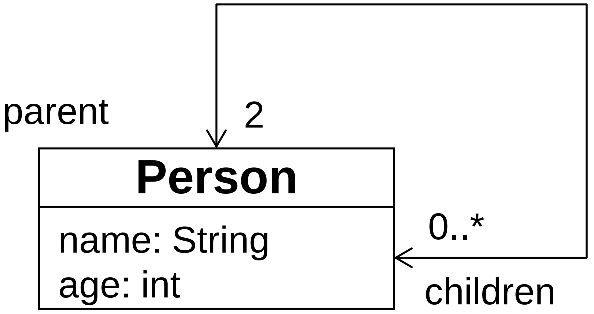
*

studies

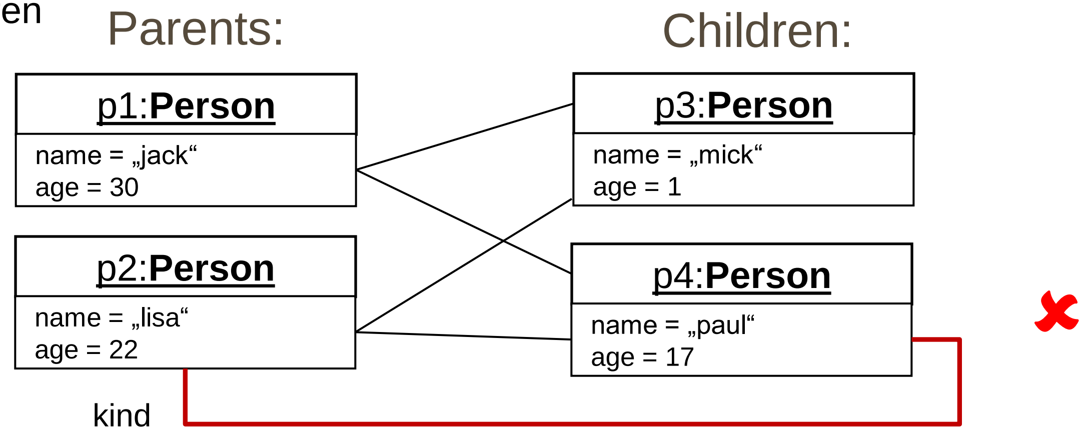
A Person obtains family allowance, if he/she is younger than 18 years, or if he/she is studying and younger than 27 years old.

```
context Person::familyallowance
derive: self.age < 18 or
       (self.age < 27 and self.studies -> size() > 0)
```

Example 4: Definitions



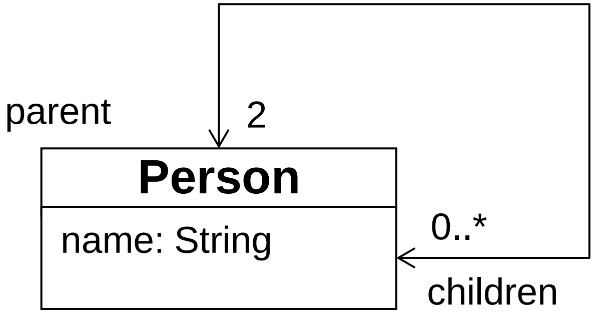
Constraint: A Person is not a relative of itself



```
context Person
def: relative: Set(Person) = children-> union(relative)
inv: self.relative -> excludes(self)
```

Assumption: Fixed-point semantic, otherwise if then else required

Example 5: equivalent OCL-formulations (1)



Constrain: A person is not its own child

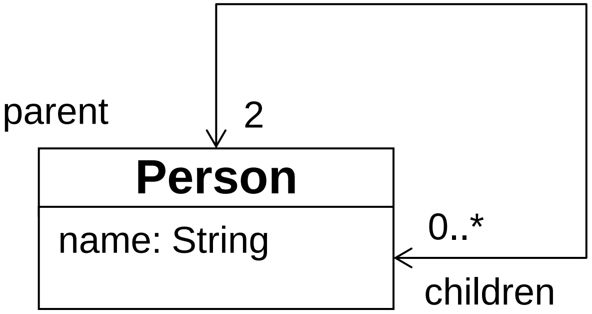
-
  ```
  (self.children->select(k | k = self))->size() = 0
  ```

  The Number of children for each person „self“, where the children are the person „self“, have to be 0.

-
  ```
  (self.children->select(k | k = self))->isEmpty()
  ```

  The set of children for each person „self, where the children are the person „self“, has to be empty.

Example 5: equivalent OCL-formulations (2)



Constrain: A person is not its own child

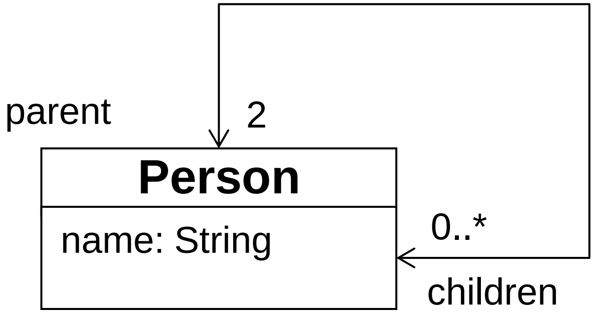
- `not self.children->includes(self)`

  It is not possible, that the set of children of each person „self“ contains the person „self“.

- `self.children->excludes(self)`

  The set of children of each person „self‟ cannot contain „self‟.

Example 5: equivalent OCL-formulations (3)



Constrain: A person is not its own child

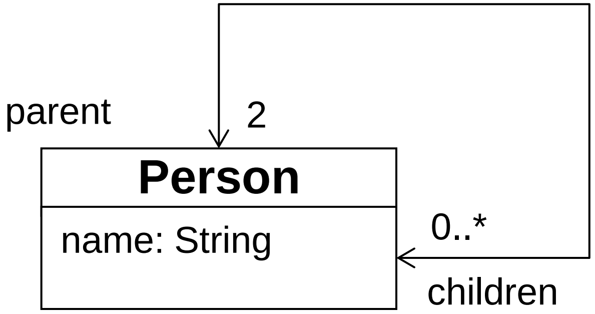
- **`Set{self}->intersection(self.children)->isEmpty()`**

  The intersection between the one element set, which only includes one person „self“ and the set of the children of „self“ has to be empty.

- **`(self.children->reject(k | k <> self))->isEmpty()`**

  The set of children for each person „self“, for whome does not apply, that they are not equal to the person „self“, has to be empty.

Example 5: equivalent OCL-formulations (4)



Constrain: A person is not its own child

- **`self.children->forAll(k | k <> self)`**

  Each child of the person „self“ is not the person „self“.

- **`not self.children->exists(k | k = self)`**

  There is no child for each person „self“, which is the person „self“

References on OCL

- **Literature**
  - Object Constraint Language Specification, Version 2.0
    - http://www.omg.org/technology/documents/formal/ocl.htm
  - Jos Warmer, Anneke Kleppe: The Object Constraint Language - Second Edition, Addison Wesley (2003)
  - Martin Hitz et al: UML@Work, d.punkt, 2. Auflage (2003)
- **Tools**
  - OSLO - http://oslo-project.berlios.de
  - Octopus - http://octopus.sourceforge.net
  - Dresden OCL Toolkit - http://dresden-ocl.sourceforge.net
  - EMF OCL - http://www.eclipse.org/modeling/mdt/?project=ocl