The Object Constraint Language (OCL)

Dan CHIOREAN

"Babes-Bolyai" University CLUJ-NAPOCA

chiorean@cs.ubbcluj.ro

The Object Constraint Language
UML & OCL

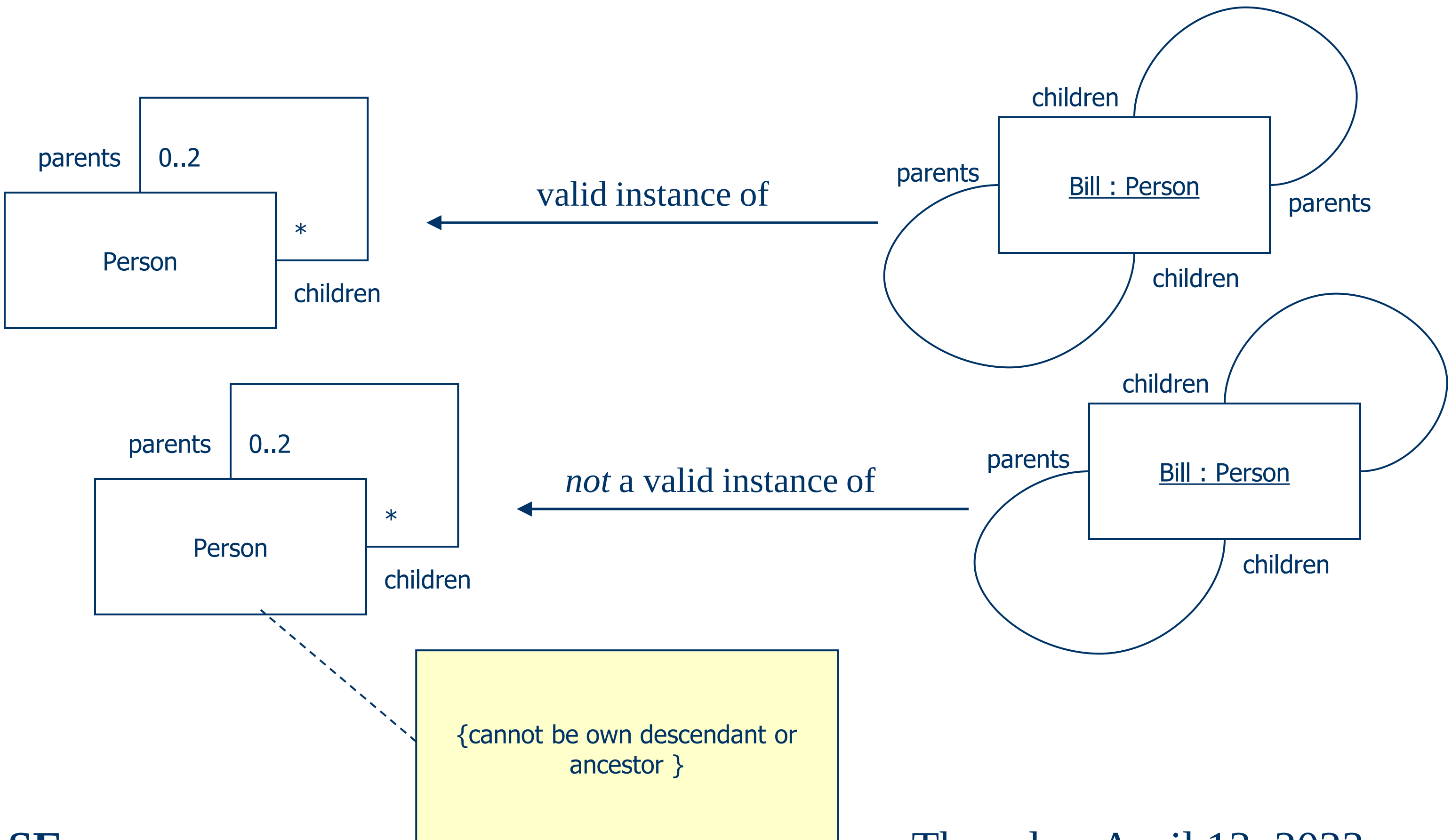
"In theory there is no difference between theory and practice; in practice there is.“

Yogi Bera

Berra was known for his impromptu pithy comments, malapropisms, and seemingly unintentional witticisms, known as "Yogi-isms". These often took the form of either an apparent tautology or a contradiction, but often with underlying humor and wisdom.

The Object Constraint Language

Only using UML diagrams can't tell the whole story



The Object Constraint Language

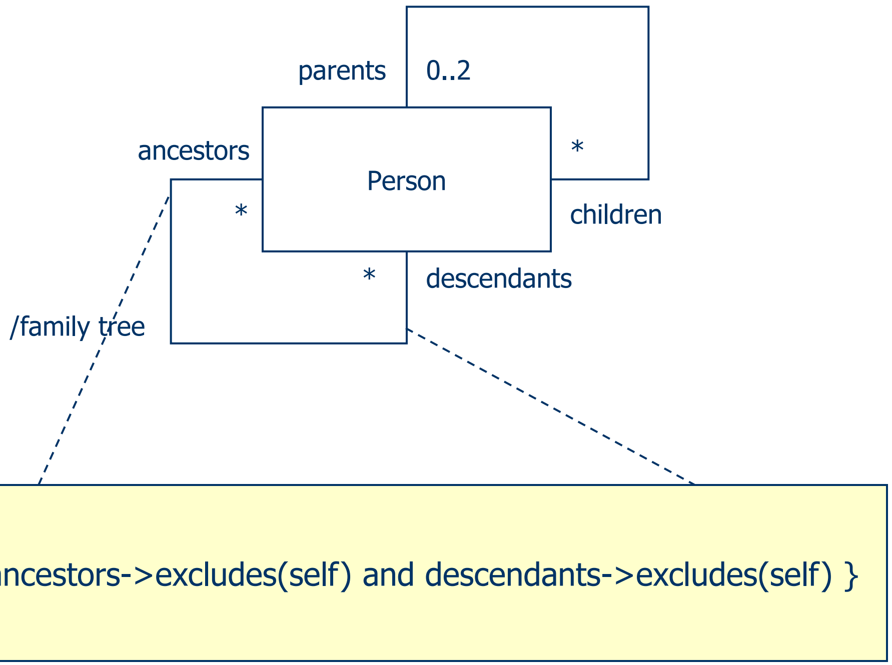
OCL supports the unambiguous constraints specifications

{ancestors = parents->union(parents.ancestors->asSet())}

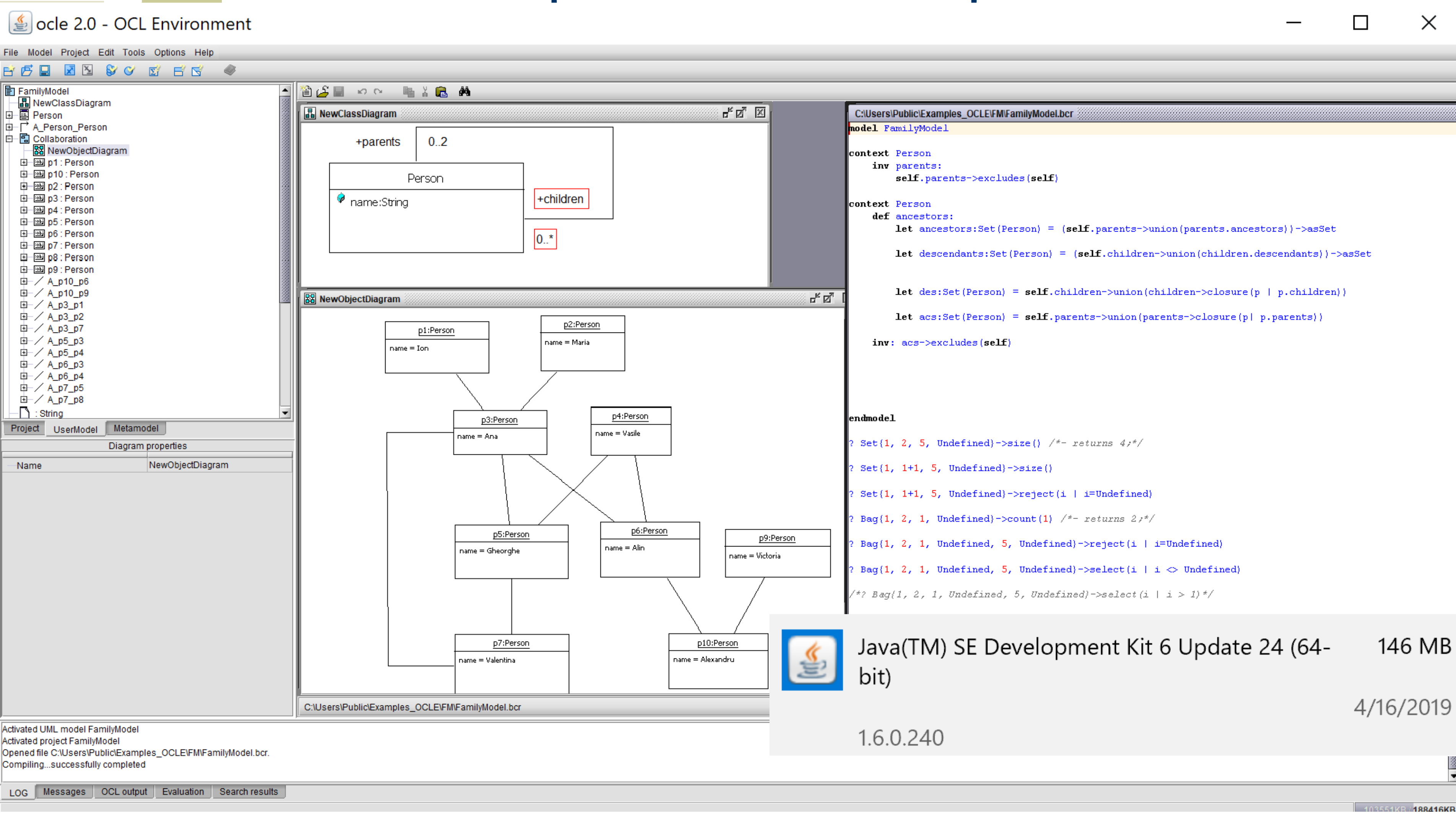
acs:Set(Person) = self.parents->union(parents->closure(p| p.parents))

{descendants = children->union(children.descendants->asSet())}

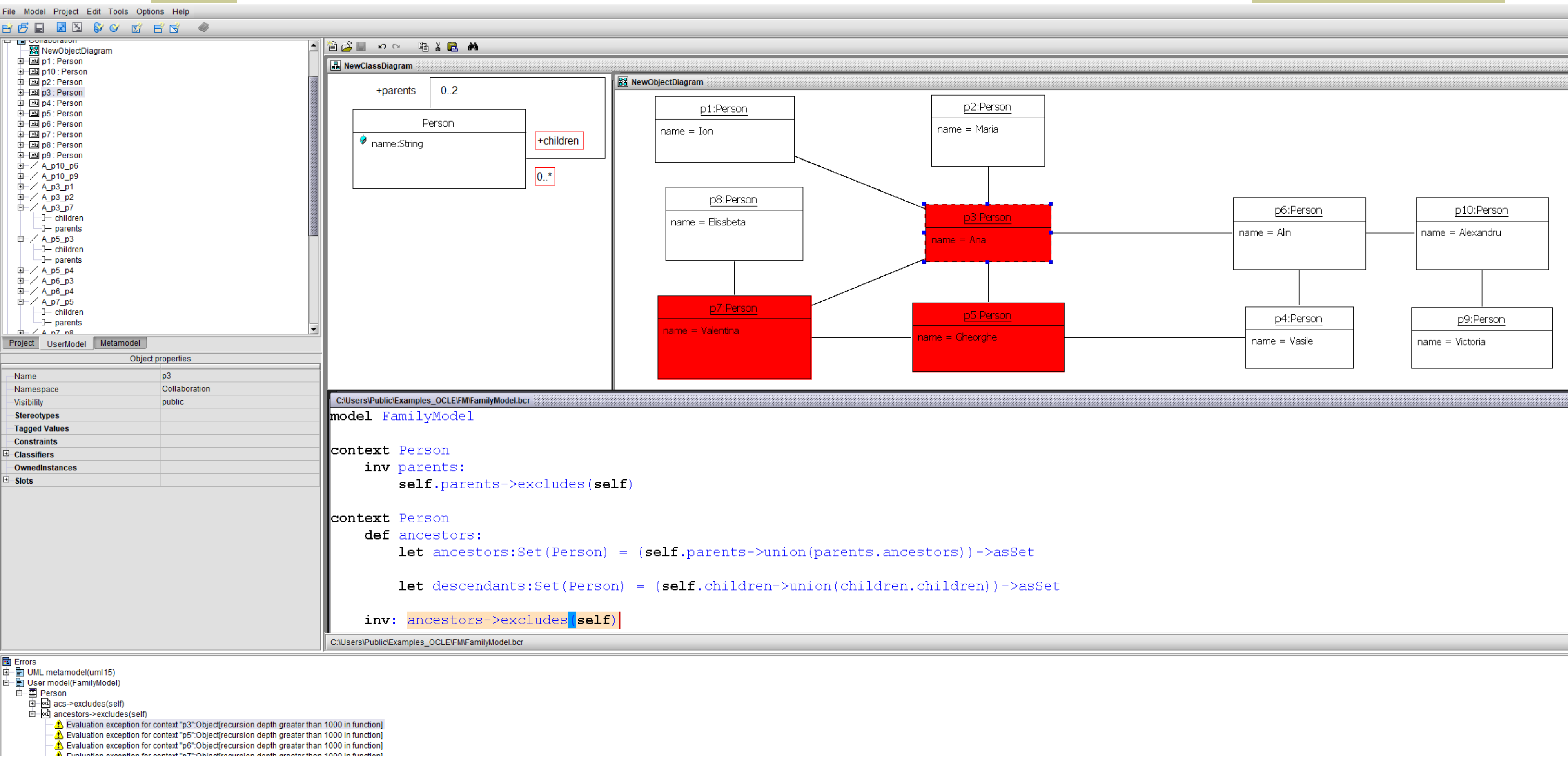
des:Set(Person) = self.children->union(children->closure(p | p.children))



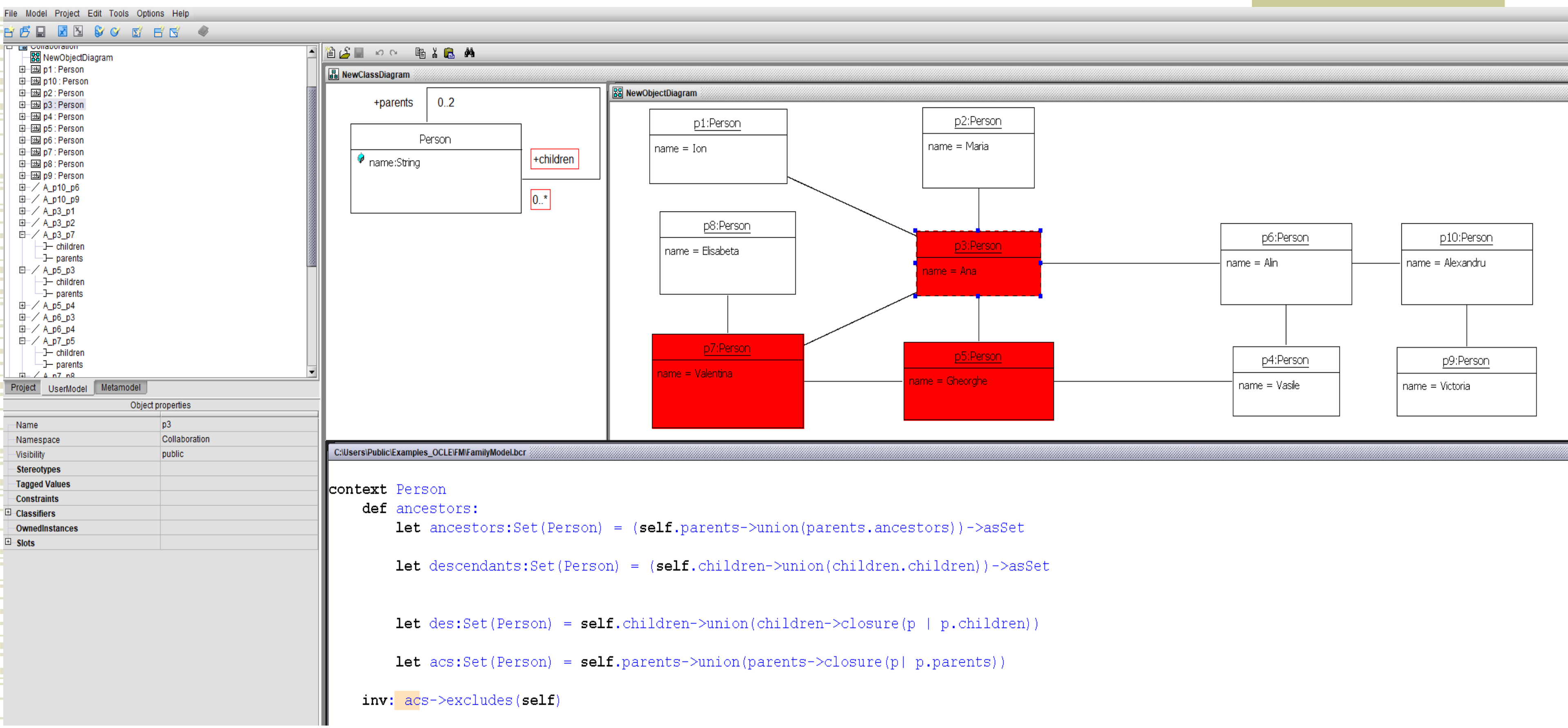
The Object Constraint Language
OCLE 2.0 http://lci.cs.ubbcluj.ro/ocle/



The Object Constraint Language Recursive Specification Drawback



The Object Constraint Language hidden specification errors



The Object Constraint Language

The context and main features

- OCL has arise from the necessity of bridging an expressivity gap of UML
  - diagrams are unable to capture most of the constraints imposed by nontrivial software systems

- => OCL is not a standalone language, but a language meant to complement UML. OCL expressions are meaningful in the context of a valid UML/(MOF based) model [except OCL literal expressions].

- the current OCL standard is a side free effects (declarative) language. So, the evaluation of an OCL specification does not change the UML model

The Object Constraint Language

The context and main features 2

- OCL is a typed language – all elements of OCL expressions are typed => the type of any OCL expression can be obtained by inference
- OCL supports first order logic
- **the language supports main features of OOP**. OCL specifications are inherited in descendants, where they may be overwritten. Constraint redefinition complies with the rules established in Design By Contract. The language supports type-casting, including upper type-casting

The Object Constraint Language

The purposes of using OCL

- **to support:**
- *model navigation* - querying the information in a model through (repeated) navigation of its association relationships using role names;
- *specification of assertions* - explicit definition of pre/post-conditions and invariants, as promoted by Design by Contract;
- *definitions of behavior* - body specifications for the query operations included in the model, derivation rules for existing attributes and references, as well as the definition of new operations and attributes for the model;
- *specification of guards*, of *type invariants for stereotypes*, etc.

The Object Constraint Language

The purposes of using OCL 2

- The objectives of using OCL go beyond the purpose of accomplishing a complete and unambiguous description of the problem solution by means of models.

- The final target is **to produce high quality software by using models**. Translating model-level OCL specifications into code must be done in a natural an unequivocal manner. Moreover, the results obtained when evaluating OCL specifications on model instantiations should be equivalent to the results obtained at run time, when evaluating their corresponding code on similar configurations of objects.

The Object Constraint Language

OCL Types

- Predefined types:
  - Basic Types: Boolean, Integer, Real, and String
  - Collections Types: Collection, Set, Bag, OrderedSet, and Sequence
- User-defined types:
  - Enumeration Type, Tuple Type
- Types defined in the UML diagrams
- OclAny, OclVoid, OclInvalid

The Object Constraint Language

OCL Types

- Precedence order for operations, starting with highest:
  - `@pre`
  - `dot and arrow operations: '.' and '->'`
  - `unary 'not' and unary minus '-'`
  - `'*' and '/'`
  - `'+' and binary '-'`
  - `'if-then-else-endif'`
  - `'<', '>', '<=', '>='`
  - `'=', '<>'`
  - `'and', 'or' and 'xor'`
  - `'implies'`

- Parentheses '(' and ')' can be used to change precedence
  - Often useful simply to make it clearer

The Object Constraint Language
OCL specification

- OCL specifications are in a context like:

  - **context** ClassName/InterfaceName
    inv [nameOfInv]: | def DefName:/DefSignature:=oclExpr
    - **let varName:Type = oclExpr**

  - **context** ClassName::OpSignature
    - **pre** [nameOfPre]: **|** **post** [nameOfPost]:

  - **package** PackageName
    …
  - **endpackage**

  - **context** StateMachineName
    inv [nameOfInv]:

The Object Constraint Language

Boolean values

Operation	Notation	Result Type
or	a or b	Boolean
and	a and b	Boolean
exclusive or	a xor b	Boolean
negation	not a	Boolean
equals	a = b	Boolean
not equals	a <> b	Boolean
implies	a implies b	Boolean

The Object Constraint Language

Boolean values 2

a	b	**not** a	a **or** b	a **and** b	a **implies** b	a **xor** b
false	false	true	false	false	true	false
false	true	true	true	false	true	true
false	ε	true	⊥	false	true	⊥
false	⊥	true	⊥	false	true	⊥
true	false	false	true	false	false	true
true	true	false	true	true	true	false
true	ε	false	true	⊥	⊥	⊥
true	⊥	false	true	⊥	⊥	⊥
ε	false	⊥	⊥	false	⊥	⊥
ε	true	⊥	true	⊥	⊥	⊥
ε	ε	⊥	⊥	⊥	⊥	⊥
ε	⊥	⊥	⊥	⊥	⊥	⊥
⊥	false	⊥	⊥	false	⊥	⊥
⊥	true	⊥	true	⊥	⊥	⊥
⊥	ε	⊥	⊥	⊥	⊥	⊥
⊥	⊥	⊥	⊥	⊥	⊥	⊥

Four-valued logic(ε invalid value, ⊥ undefined value)

The Object Constraint Language

Evaluating expressions containing undefined values

```
Set{1, 2, 5, Undefined}->size()

Set{1, 1+1, 5, Undefined}->size()

Set{1, 1+1, 5, Undefined}->reject(i | i=Undefined)

Bag{1, 2, 1, Undefined}->count(1)

Bag{1, 2, 1, Undefined, 5, Undefined}->reject(i | i=Undefined)

Bag{1, 2, 1, Undefined, 5, Undefined}->select(i | i <> Undefined)

Bag{1, 2, 1, Undefined, 5, Undefined}->select(i | i > 1)
```

The Object Constraint Language

Type system – important features

OclAny

- `OclAny` is the supertype of all types in UML models and is an instance of the metatype `AnyType`. Features of `OclAny` are available on each object. Each class of UML user models inherits all operations defined on `OclAny`. Most of them are basic operations in object-toriented languages, such as:
  - **operations for testing the equality of two objects**
    - `=(object2:OclAny):Boolean`
    - `<>(object2:OclAny):Boolean`
  - **operations inferring the objects type or state**
    - `oclIsTypeOf(type:Classifier):Boolean`
    - `oclIsKindOf(type:Classifier):Boolean`
    - `oclType():Classifier`
    - `oclIsInState(statespec:OclState):Boolean`

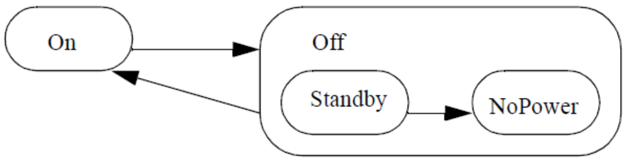
The Object Constraint Language
Type system – important features 2

OclAny

- **operations specifying type casting**
  - `oclAsType(type:Classifier):T`, where T is a classifier
- **other operations**
  - `oclIsNew():Boolean`, conceived to be used in postconditions, to support the identification of objects created after the method starts executing
  - `oclIsUndefined():Boolean` and `oclIsInvalid():Boolean` are two operations returning true when the receiver object or data value is undefined/unknown or when an exception has been triggered, respectively.

The Object Constraint Language

OclAny– oclIsInState



The operation **`oclInState(s)`** results in true if the object is in the state **`s`**. Values for *s* are the names of the states in the statemachine(s) attached to the Classifier of *object*. For nested states the statenames can be combined using the double colon '::' . In the example statemachine above, values for *s* can be **`On, Off, Off::Standby, Off::NoPower`**.

```
object.oclInState(On)
object.oclInState(Off)
object.oclInstate(Off::Standby)
object.oclInState(Off:NoPower)
```

If there are multiple statemachines attached to the object's classifier, then the statename can be prefixed with the name of the statemachine containing the state and the double semicolon ::, as with nested states.

The Object Constraint Language

Accessing Overridden Properties

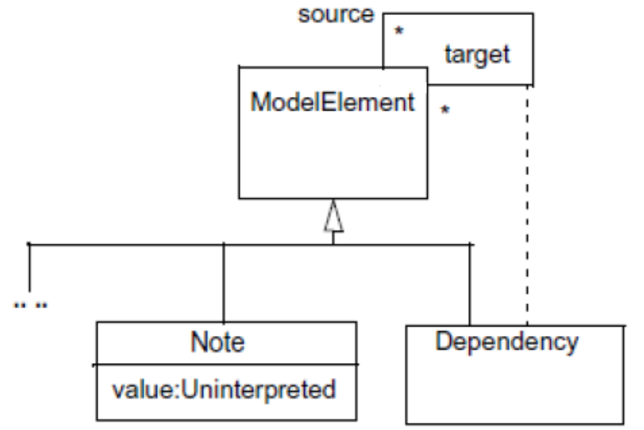


Figure 7.4 - Accessing Overridden Properties Example

```
context Dependency
    inv: self.source <> self
    inv: self.oclAsType(Dependency).source->isEmpty()
    inv: self.oclAsType(ModelElement).source->isEmpty()
```

The Object Constraint Language

pre & post(conditions)

- Constraint that specify the applicability and effect of an operation without stating an algorithm or implementation
- Are attached to an operation in a class diagram
- Allow a more complete specification of a system

Preconditions must be true just prior to the execution of an operation

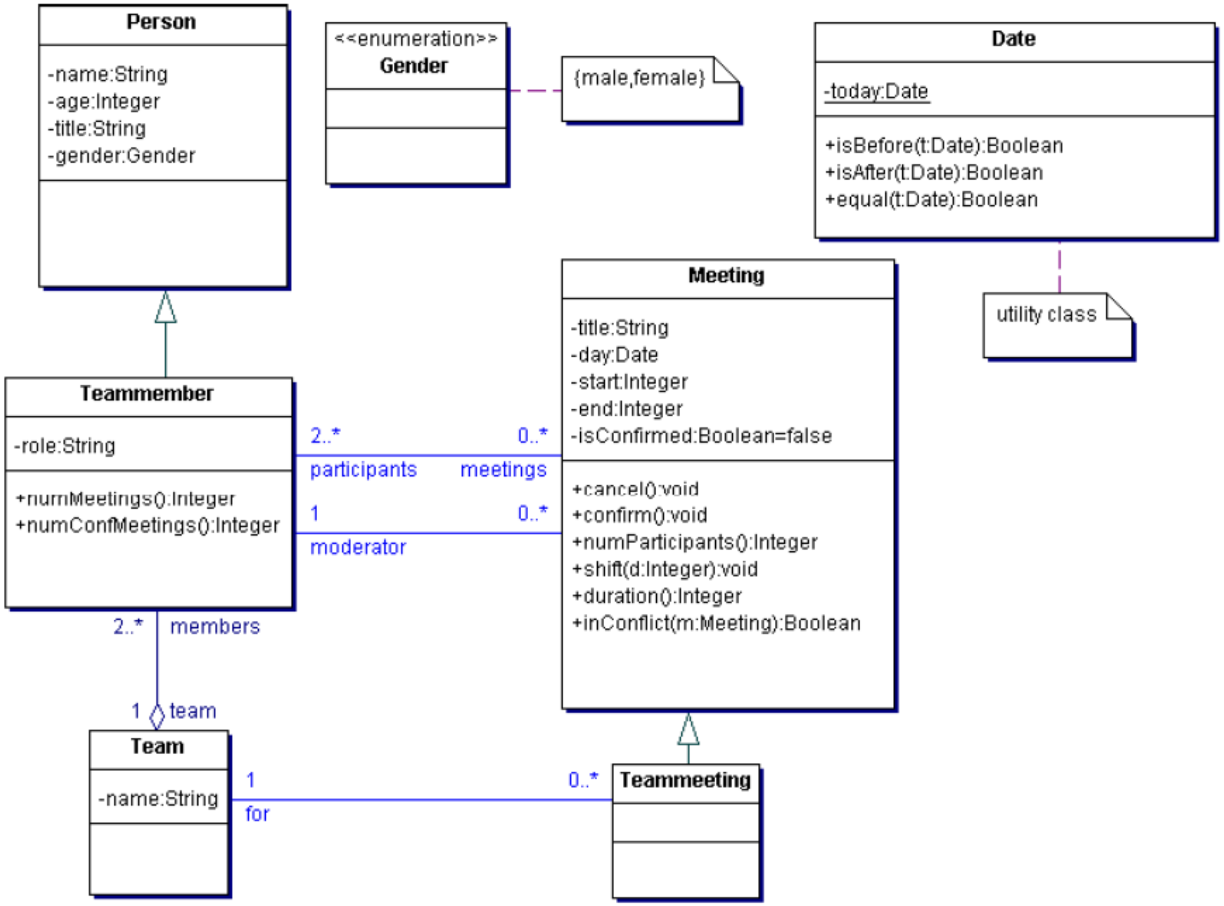
Precondition

Syntax

```
context <classifier>::<operation> (<parameters>)
   pre [<constraint name>]:
        <Boolean OCL expression>
```

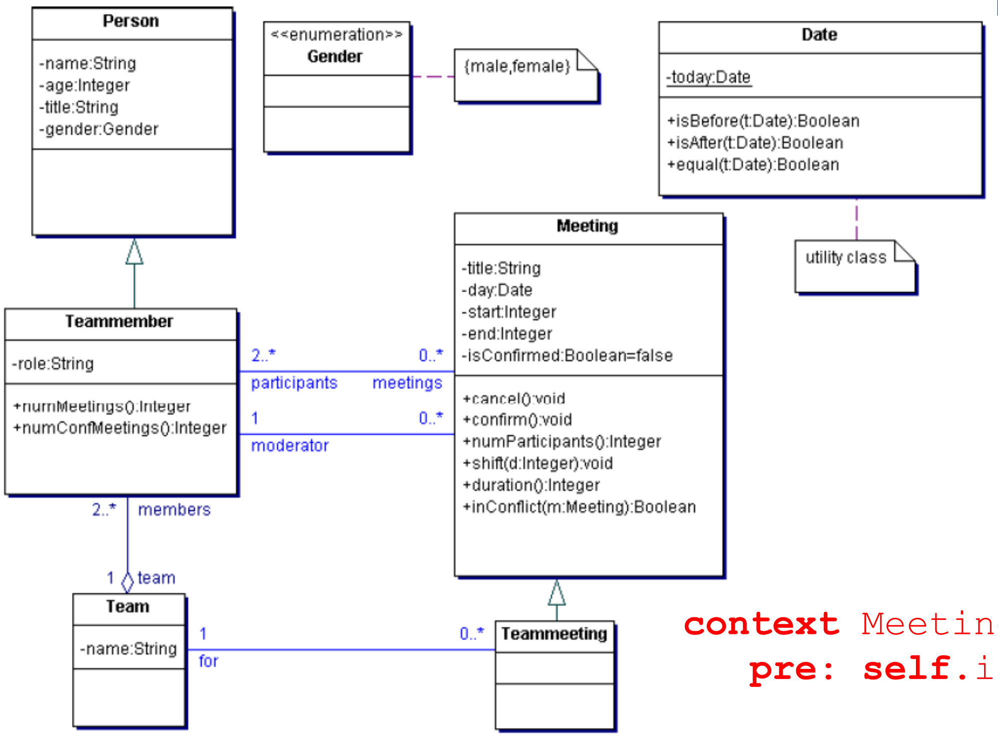
The Object Constraint Language

pre & post(conditions) 2



The Object Constraint Language

pre & post(conditions) 3




```
context Meeting::shift(d:Integer)
   pre: self.isConfirmed = false

context Meeting::shift(d:Integer)
   pre: d>0

context Meeting::shift(d:Integer)
   pre: self.isConfirmed = false and d>0
```

The Object Constraint Language

pre & post(conditions) 4

Postcondition

Constraint that must be true just after to the execution of an operation. Postconditions are the way how the actual effect of an operation is described in OCL.

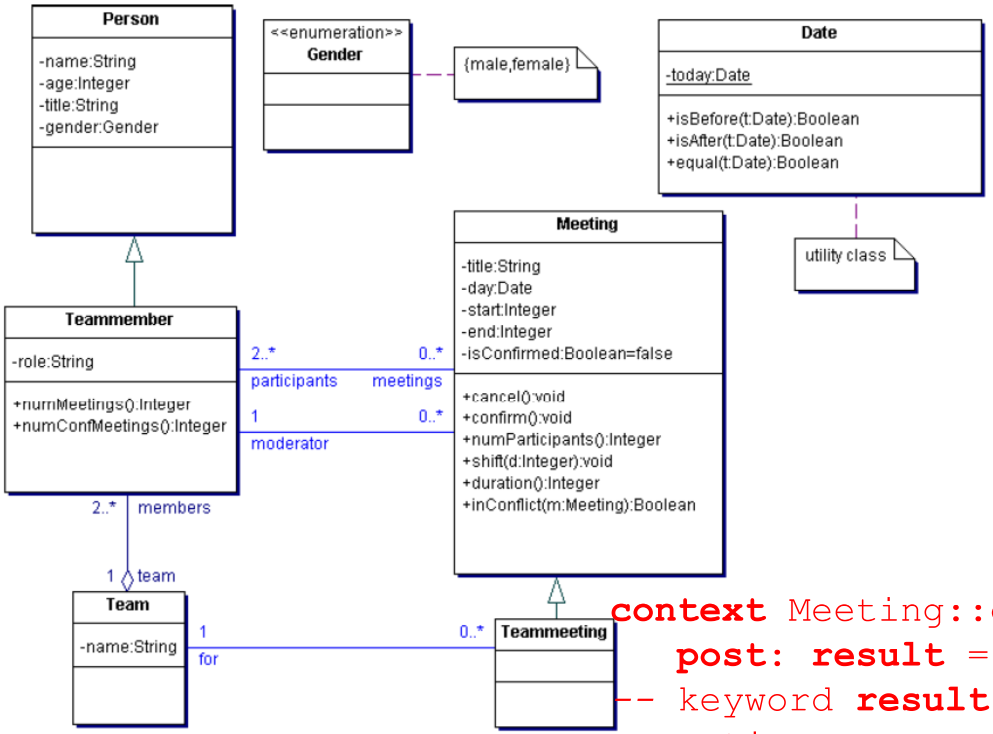
Syntax

```
context <classifier>::<operation> (<parameters>)
   post [<constraint name>]:
        <Boolean OCL expression>
```

- `vizElem@pre` indicates a part of an expression which is to be evaluated in the original state before execution of the operation
- `vizElem` refers to the value upon completion of the operation
- `@pre` is only allowed in postconditions

The Object Constraint Language

pre & post(conditions) 5




```
context Meeting::duration():Integer
   post: result = self.end - self.start
-- keyword result refers to the result of the
operation

context Meeting::confirm()
   post: self.isConfirmed = true

context Meeting::shift(d:Integer)
   post: start = start@pre +d and end = end@pre + d
```

The Object Constraint Language

Collection types

- Models are modeled by graphs => the need to navigate/query the model
- At the class level, this is done by accessing the **appropriate opposite association end**. When the multiplicity of the association end is greater than 1, navigation will result in a **Set**. When the association end is adorned with {ordered}, the result will be an **OrderedSet**. Therefore, the **collection types defined in the OCL Standard Library play an important role in OCL expressions**.
- Apart of the two above mentioned types, collection types also include **Bag**, **Sequence** and **Collection**, the latter being the common parent of the other four collection types.
- The hierarchy and behavior of these types has been inspired by Smalltalk Collection classes.

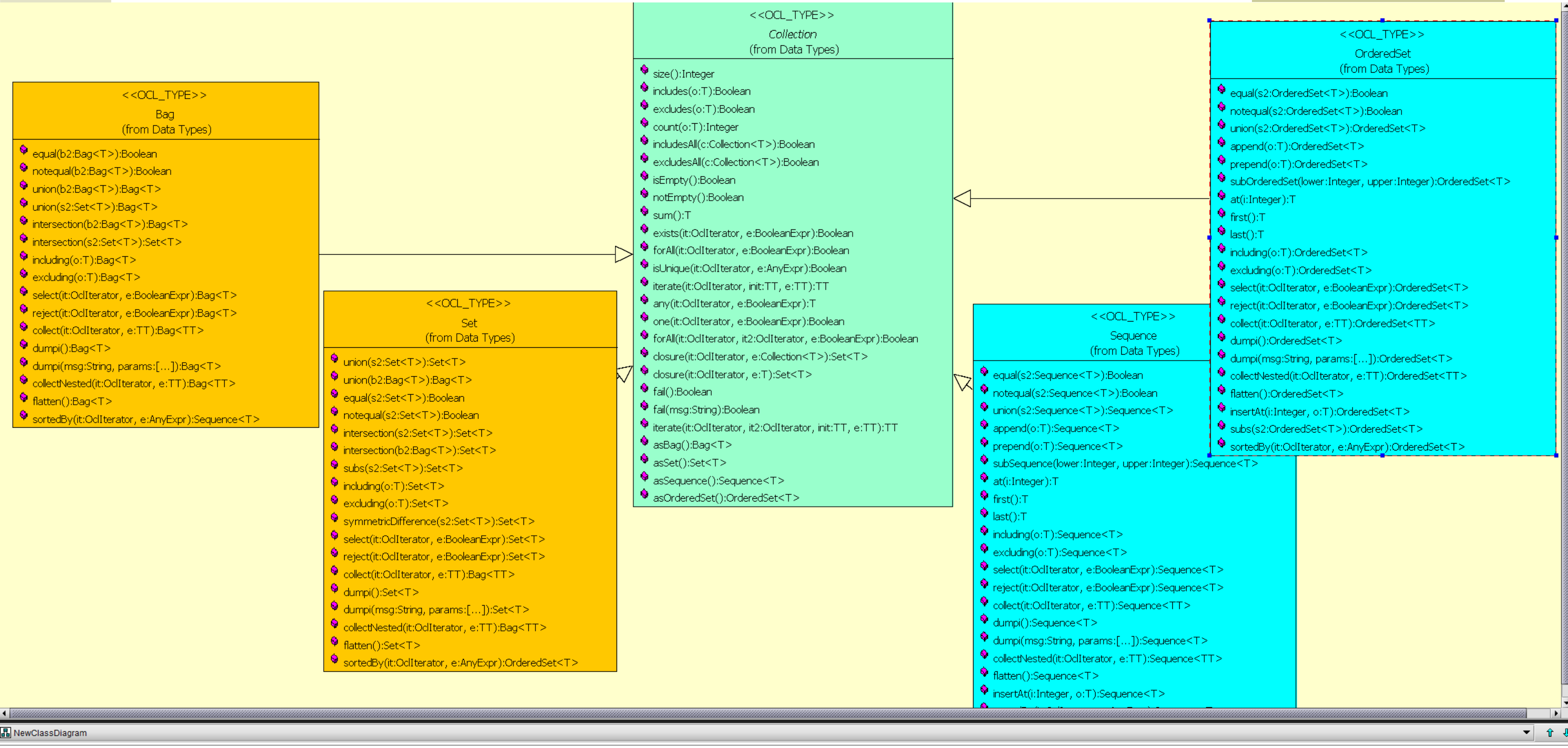
The Object Constraint Language
Collection types 2

- Usually, operations applied to collections are specified by using the arrow operator (->). This is meant to stress that, in case of collections, operations are potentially applied to many receivers (collection elements). However, in case of the **`collect`** operation, an exception (enabling to replace the -> operator with the dot operator (`.`)) is made.

- So, the expression **`aCollection->collect(property)`** is equivalent to **`aCollection.property`**. This exception, known as ”shorthand collect”, is very used in practice, since it leads to slightly shorter specifications.

The Object Constraint Language

Collection hierarchy in OCL



The Object Constraint Language

Collection – the **`iterate`** operation

- The **`iterate`** operation is very generic. All the operations: **`reject, select, forAll, exists, collect`** can all be described in terms of **`iterate`**.

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

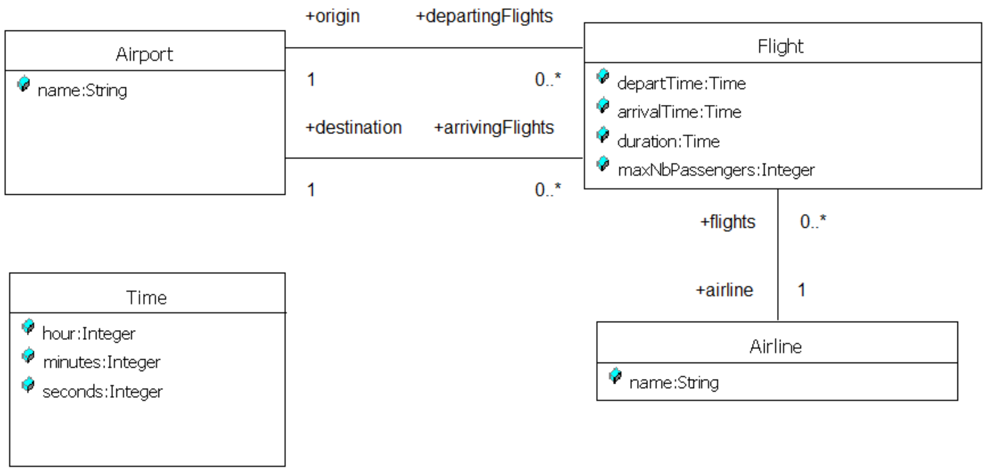
- When the iterate is evaluated, *elem* iterates over the *collection* and the *expression-with elem-and-acc* is evaluated for each *elem*. After each evaluation of *expression-with elem-and-acc*, its value is assigned to *acc*. In this way, the value of *acc* is built up during the iteration of the collection.

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

-- is identical to:

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

The Object Constraint Language

Collection types 3 - Navigating associations

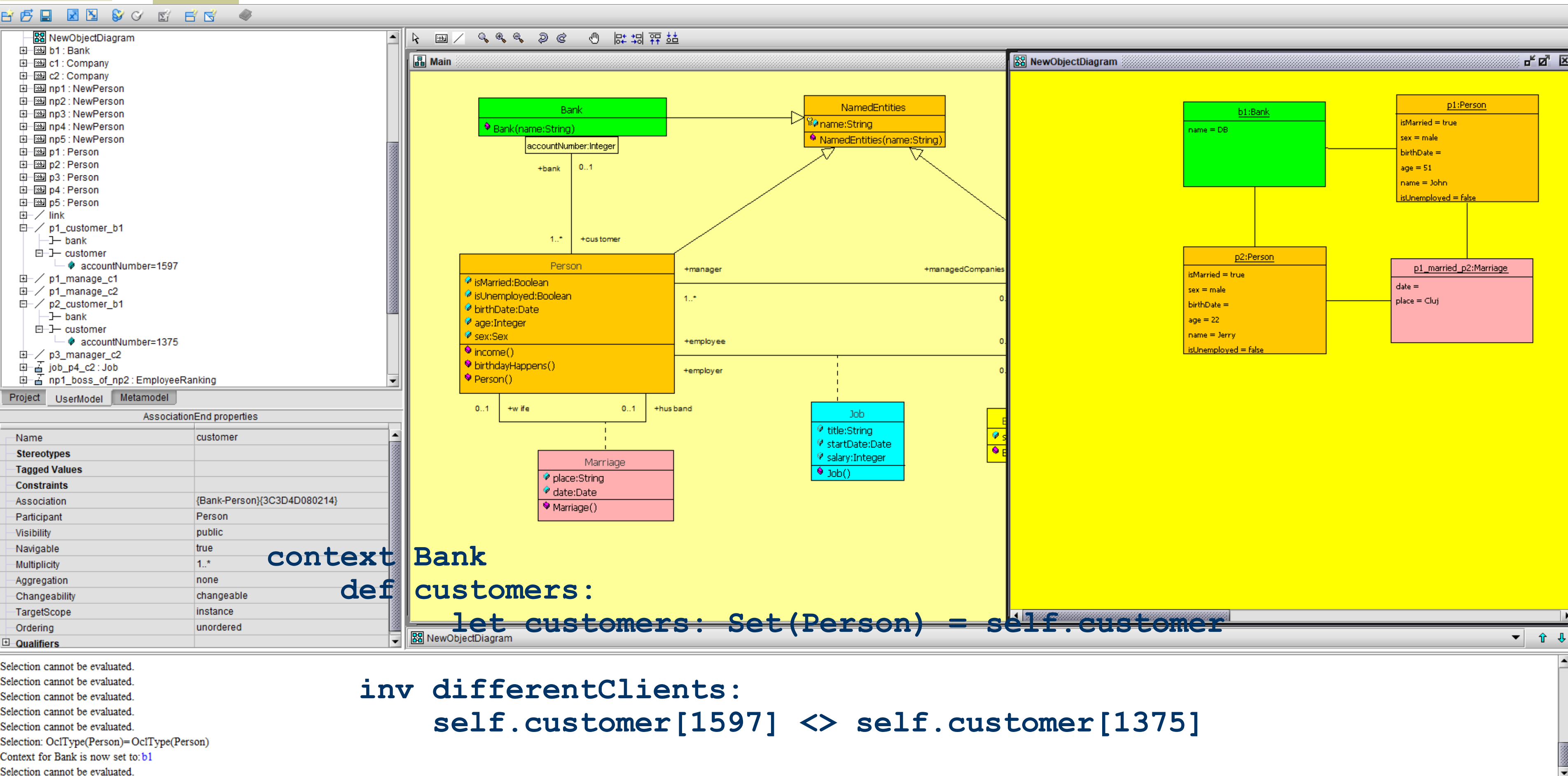
- Every association in the model is a navigation path.
- The context of the expression is the starting point.
- Role names are used to identify the navigated association.




```
context Flight

    inv: origin <> destination
    inv: origin.name = "Amsterdam"
```

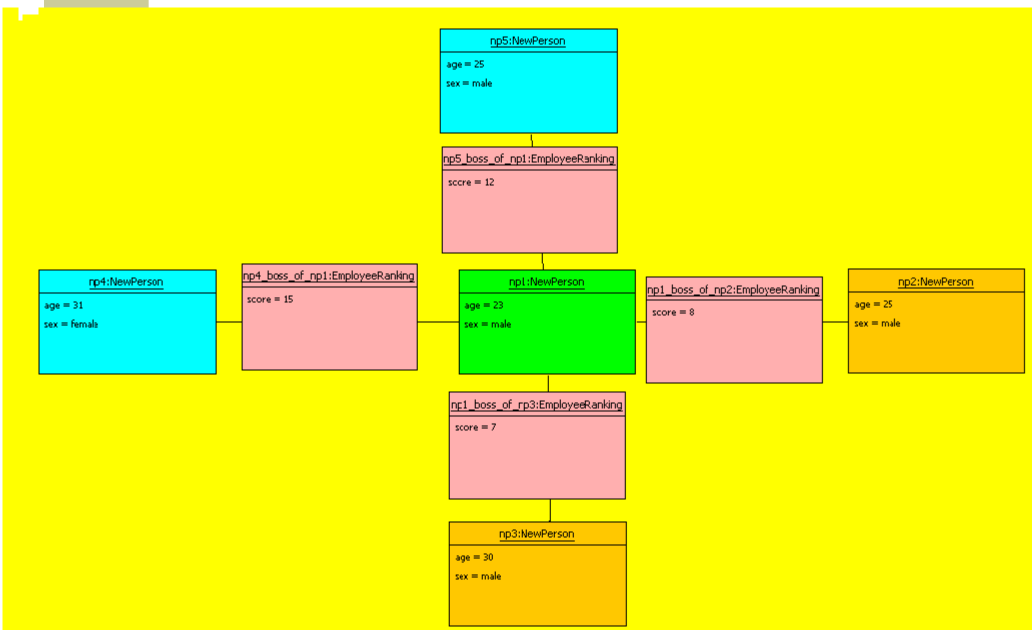
The Object Constraint Language

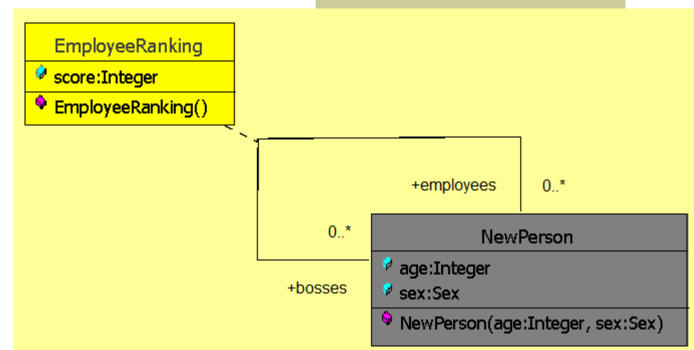
Qualified associations




```
context Bank
  def customers:
      let customers: Set(Person) = self.customer

  inv differentClients:
      self.customer[1597] <> self.customer[1375]

inv differentClients_2:
      self.customer[1597] <> self.customer[1598]
```

The Object Constraint Language
Association classes






```
context NewPerson
    inv bosse_s_objectivy:
          let nB: Integer = self.bosses->size
          let nE: Integer = self.employees->size in
          if nB*nE = 0
              then true
              else
    (self.employeeRanking[bosses].score->sum / nB) <=
    (self.employeeRanking[employees].score->sum / nE)
          endif
```

The Object Constraint Language

Collection types 4 – the `select` & `forAll` operations

Syntax:

```
collection->select(elem : T | expression)
collection->select(elem | expression)
collection->select(expression)
```

The *select and reject* operations does not change the type of the collection
The *select* operation results in the subset of all elements for which *expression* is true

Syntax:

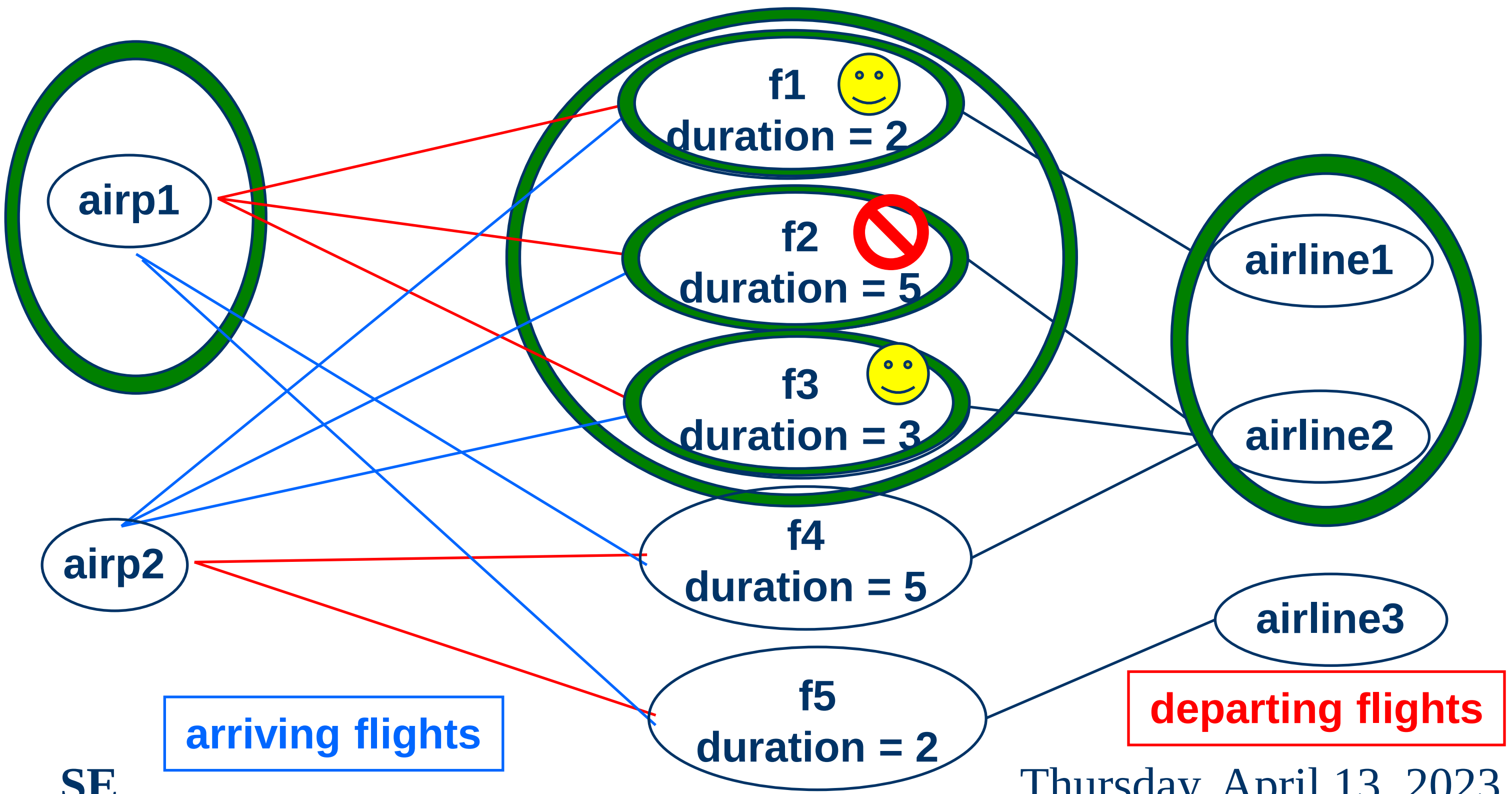
```
collection->forAll(elem : T | expr)
collection->forAll(elem | expr)
collection->forAll(expr)
```

The `forAll` operation results in `true` if expr is `true` for all elements of the collection. In case of medium and large size collections it's difficult to find element(s) violating the invariant

The Object Constraint Language

Collection types 5 – the `select` operation

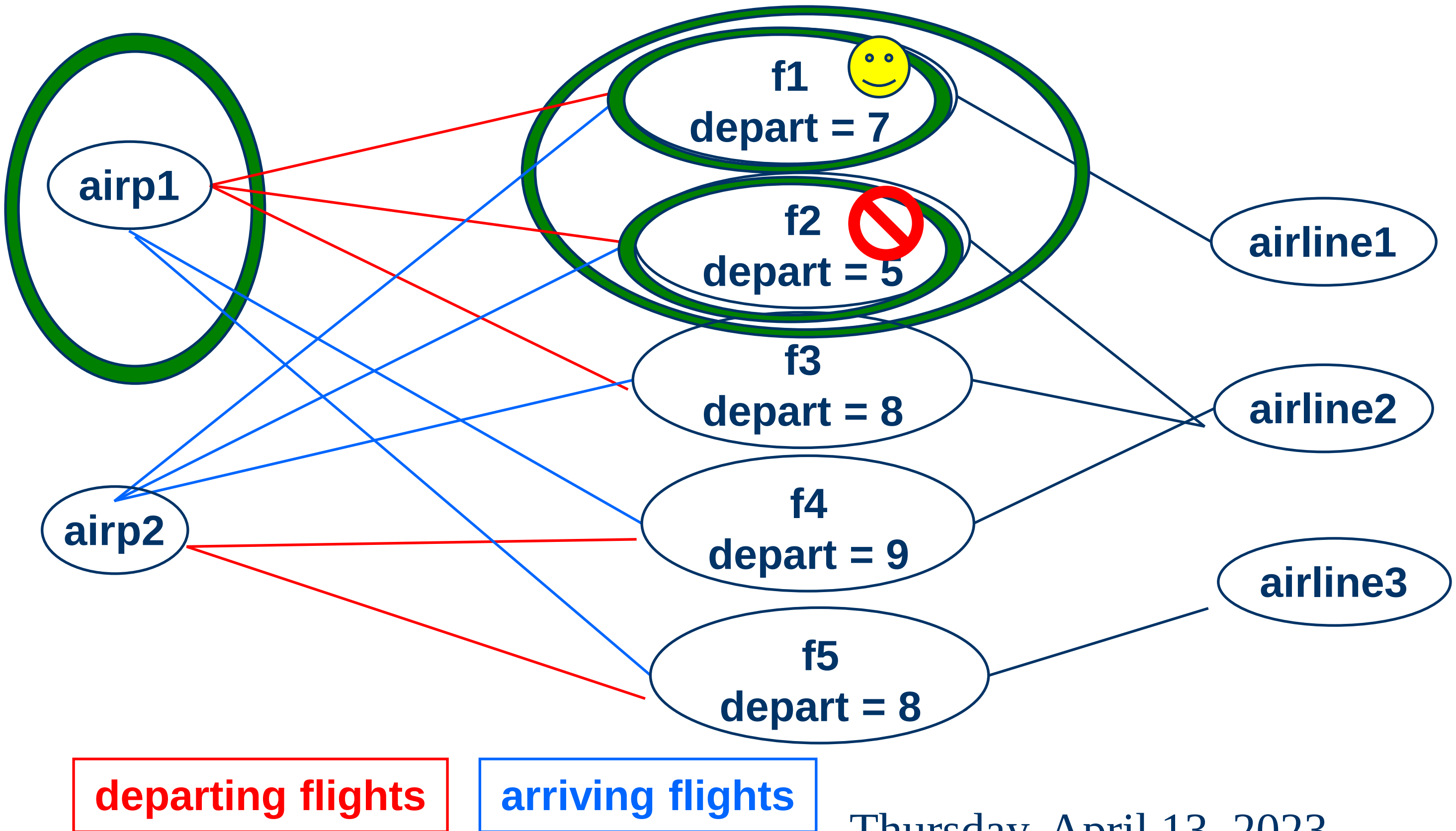
```
context Airport inv:
  self.departingFlights->select(duration<4)->notEmpty
```



The Object Constraint Language

Collection types 6 – the forAll operation

```
context Airport inv:
    self.departingFlights->forAll(departTime.hour>6)
```



The Object Constraint Language

Collection types 7 – the `exists` operation

Syntax:

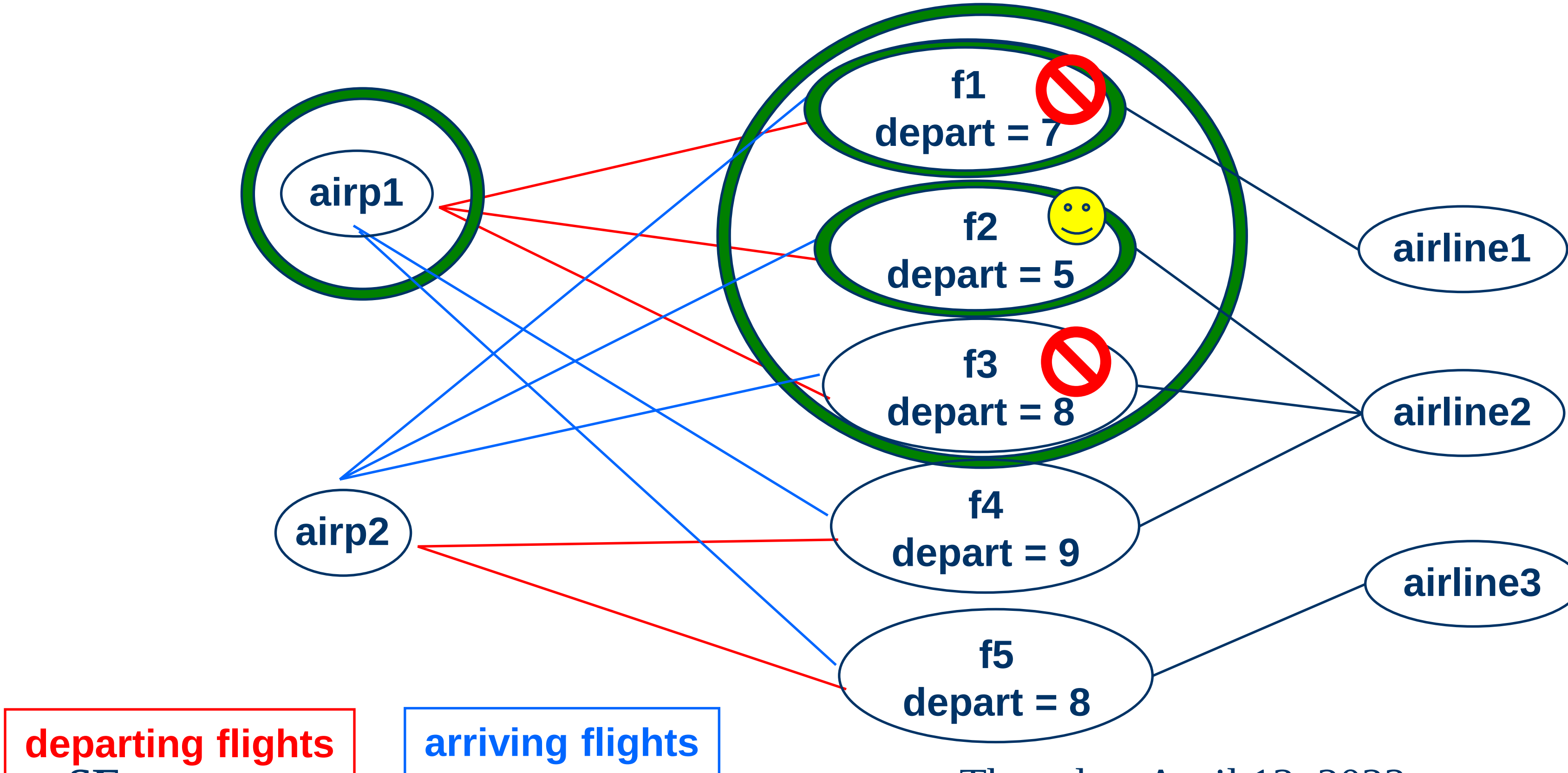
```
collection->exists(elem : T | expr)
collection->exists(elem | expr)
collection->exists(expr)
```

The `exists` operation results in `true` if there is at least one element in the collection for which the expression *expr* is `true`.

The Object Constraint Language

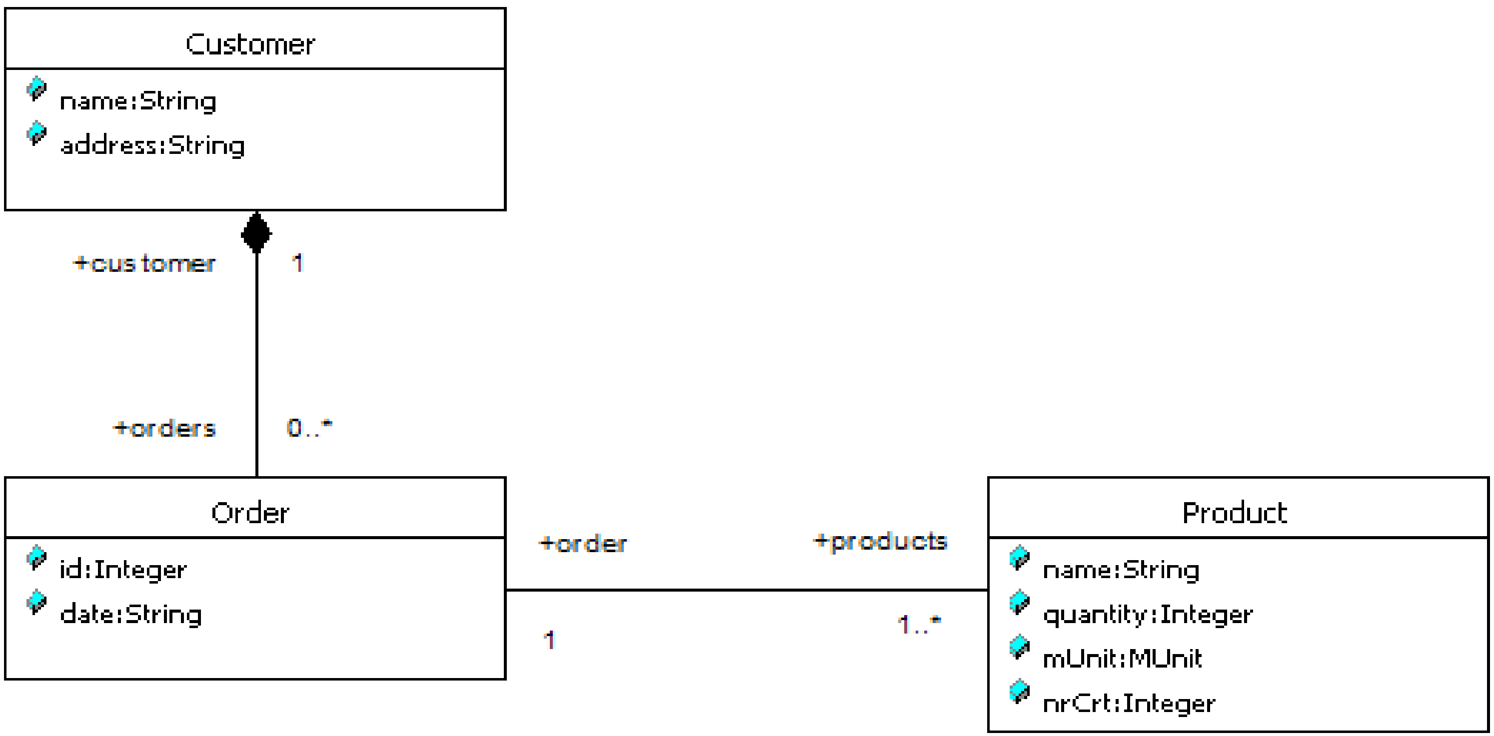
Collection types 8 – the `exists` operation

```
context Airport inv:
    self.departingFlights->exists(departTime.hour<6)
```



The Object Constraint Language

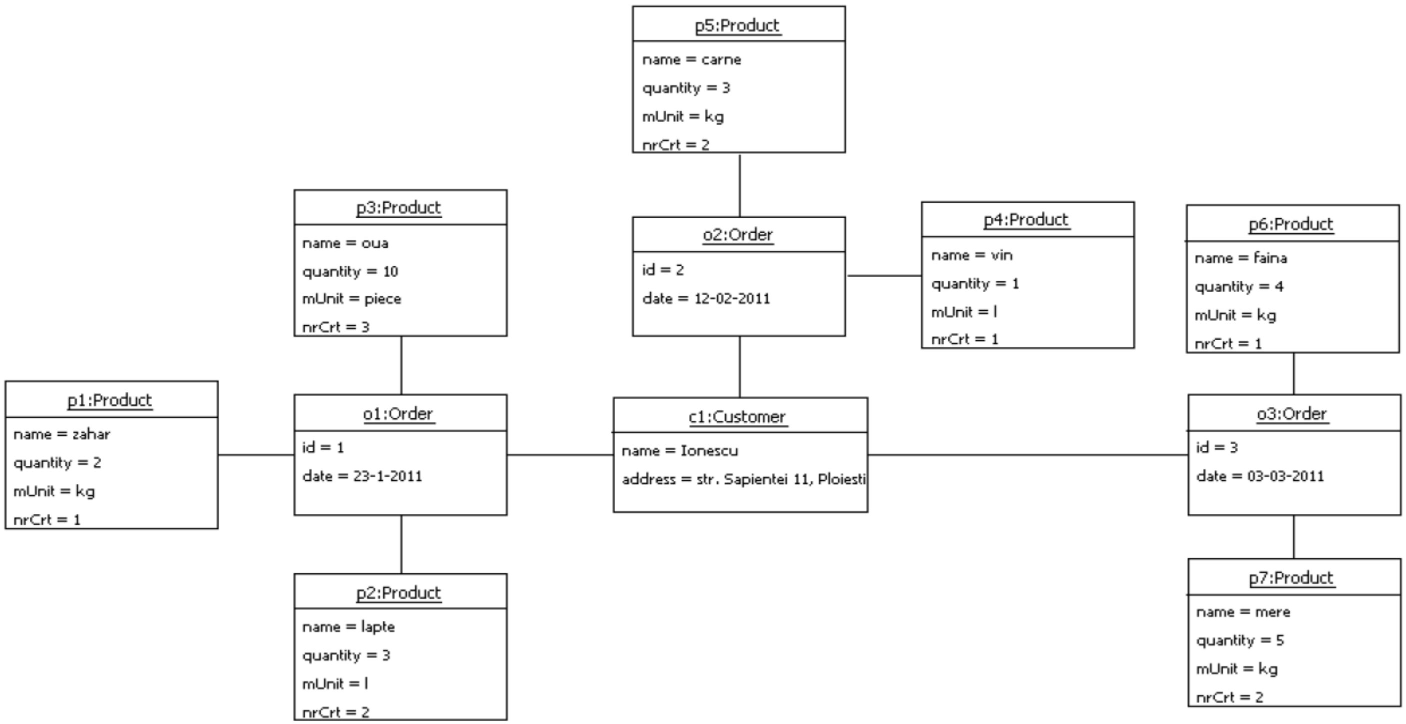
Collection types 9



Usually, users interested in knowing the products ordered by a customer evaluate the following OCL specification **`self.orders->collect(products)`**.

The Object Constraint Language

Collection types 10




```
context Customer
def productsOrderedByClient:Bag(Product)=self.orders.products
```

The result is: `Bag{p1, p2, p3, p4, p5 , p6, p7}.`

The Object Constraint Language

Collection types 11

- More information can be obtained by using the **`collectNested`** operation:

```
context Customer

def productsOrderedByClient_cn:Sequence(Sequence(Product)) =
self.orders->collectNested(products)
```

- The result returned by the OCL evaluator will be:

```
Sequence{Sequence{p1, p2, p3}, Sequence{p4, p5}, Sequence{p6, p7}}.
```

- Compared with the result of the previous query, this time we know the products grouped by order, but we do not know the id or the reference of each order.

The Object Constraint Language

Collection types 12

- Compared with the result of the previous query, this time we know the products grouped by order, but we do not know the id or the reference of each order. Slightly detailing the previous **collectNested** operation will support us in obtaining the missing information.

```
context Customer
def productsOrderedByClient_cnd:Sequence(Sequence(OclAny)) =
self.orders->collectNested(o| Sequence{o, o.products})
```

- The result returned will be:

```
Sequence{ Sequence{o1, Sequence{p1, p2, p3}}, Sequence{o2,
Sequence{p4, p5}}, Sequence{o3, Sequence{p6, p7}}}.
```

The Object Constraint Language

Collection types 13

- Moreover, if one is interested in knowing the date of each order, the query can be refined as follows:

```
context Customer
def productsOrderedByClient_cndd:Sequence(Sequence(OclAny)) =
self.orders->collectNested(o| Sequence{o, o.date, o.products})
```

- This returns:

```
Sequence{Sequence{o1,'23-1-2011',Sequence{p1, p2, p3}},
Sequence{o2, '12-02-2011', Sequence{p4, p5}}, Sequence{o3,
'03-03-2011', Sequence{p6, p7}}}.
```

The Object Constraint Language

Collection types 14

- Using a TupleType, the specification could be rewritten as:

```
context Customer
def clientProducts_nsct:Sequence(TupleType(date:String,
id:Integer, ordProds:Sequence(Product))) =
self.orders->collectNested(o| Tuple{date=o.date, id=o.id,
ordProds = o.products})
```

- Returning:

```
Sequence{Tuple{'23-1-2011', 1, Sequence{p1, p2, p3}},
Tuple{'12-02-2011', 2, Sequence{p4, p5}}, Tuple{'03-03-2011',
3, Sequence{p6, p7}}}.
```

The Object Constraint Language

Collection types 15

- Each operation on collections can be specified by using the iterate operation. In our case, the last specification is equivalent to:

```
context Customer
def clientProducts_nsctI:Sequence(TupleType(date:String,
id:Integer, ordProds:Sequence(Product))) =
self.orders->iterate(o:Order; acc: Sequence(TupleType(date:String,
id:Integer, ordProds: Sequence(Product))) =
oclEmpty(Sequence(TupleType(date: String, id:Integer,
ordProds:Sequence(Product))))| acc->including(Tuple{date=o.date,
id=o.id, ordProds = o.products }))
```

- This returns the same result. In case of using the `iterate` operation, the OCL specification is more complex than its equivalent using `collectNested`. This is because the iterate operation is the most generic operation on collections.

The Object Constraint Language

Collection types 15 - other operations on collections

- *isEmpty*: **true** if collection has no elements
- *notEmpty()*: **true** if collection has at least one element
- *size*: number of elements in collection
- *count(elem)*: number of occurences of elem in collection
- *includes(elem)*: **true** if elem is in collection
- *including(elem)*: **returns** a collection a similar collection including *elem*
- *excludes(elem)*: **true** if elem is not in collection
- *includesAll(coll)*: **true** if all elements of coll are in collection