EDOM - Engenharia de Domínio

Mestrado em Engenharia Informática

Lecture 07.1

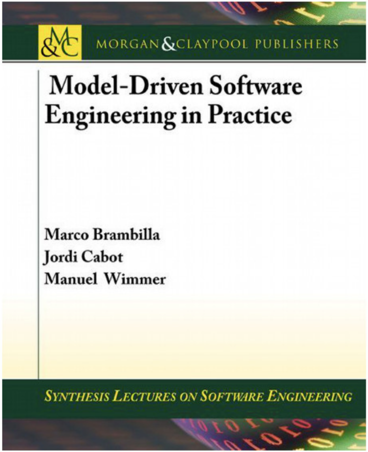
OCL

Alexandre Bragança atb@isep.ipp.pt

Dep. de Engenharia Informática – ISEP

2017/2018

Acknowledgement



"Model-Driven Software Engineering in Practice", Marco Brambilla et al., Morgan & Claypool Publishers, 2012

- These slides are based on the contents of this book.

In model-driven engineering many examples of domain-specific languages may be found like OCL, a language for decorating models with assertions or QVT, a domain-specific transformation language. However languages like UML are typically general purpose modeling languages. [1]

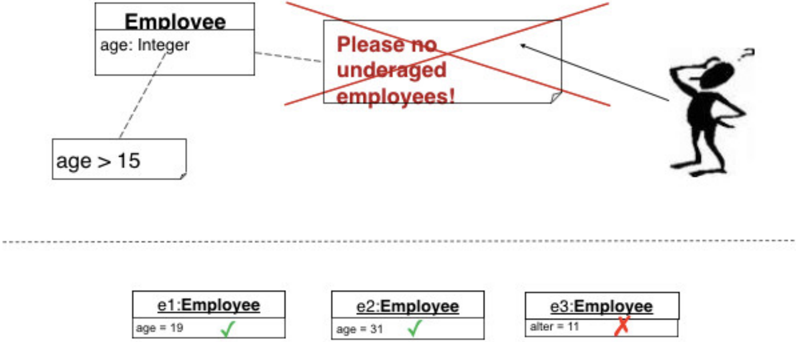
[1] From https://en.wikipedia.org/wiki/Domain-specific_language

OCL Topics

- Introduction
- OCL Core Language
- OCL Standard Library
- Further Reading

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).



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

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

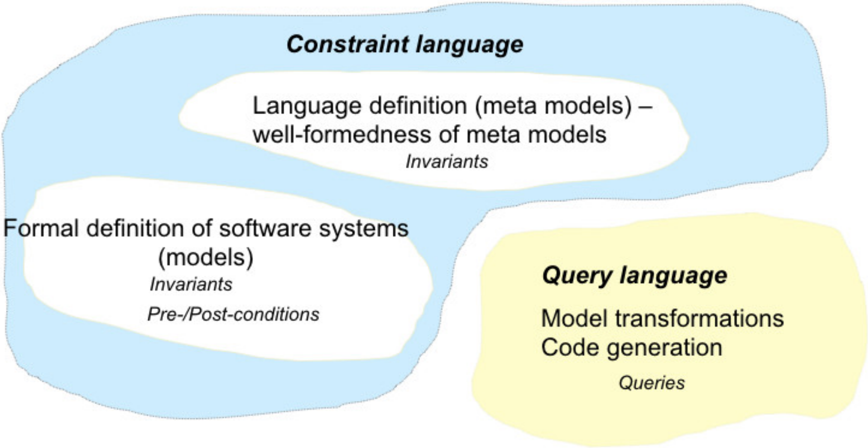
Tabela: OCL field of application

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

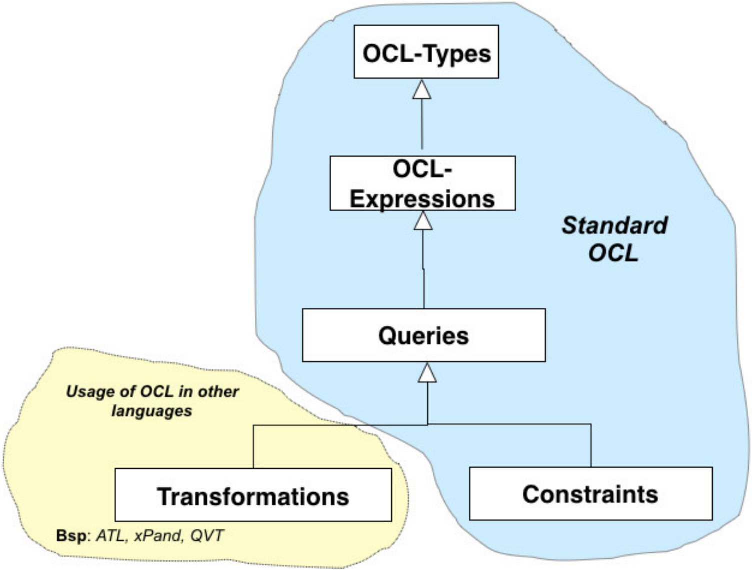
Caution: Side effects are not allowed!

- Operation allowed in OCL: **C::getAtt : String body: att**
- Operation **not allowed** in OCL: **C::setAtt(arg) : T body: att = arg**

Field of application of OCL in model driven engineering

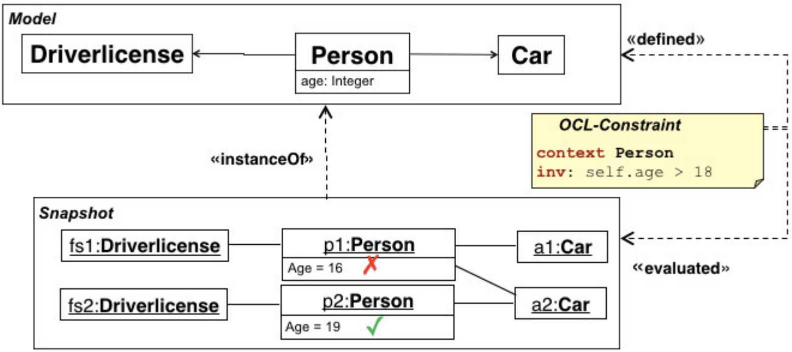


OCL Usage

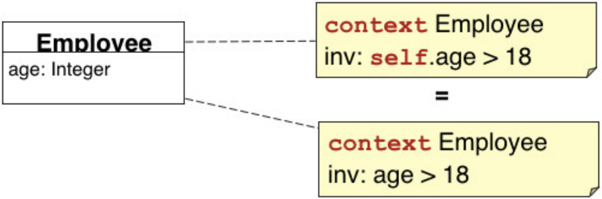


OCL-Types
OCL-Expressions
Standard OCL
Queries
Usage of OCL in other languages
Transformations
Constraints
Bsp: ATL, xPand, QVT

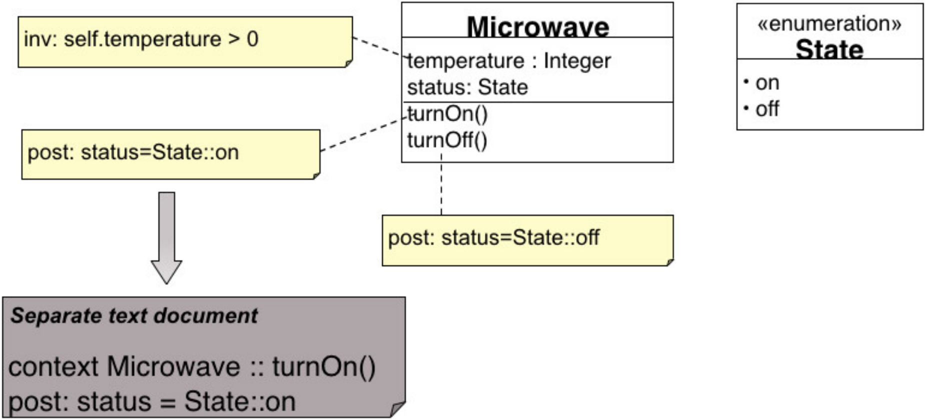
- **Constraints** are defined at the modeling level
  - Basis: Classes and their properties
- Information of the **object graph** are **queried**
  - Represents system status, also called **snapshot**
- **Analogy** to XML query languages
  - XPath/XQuery query XML-documents
  - Scripts are based on XML-schema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



- 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:



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



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 supertype: 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

Basic Types

- true, false : *Boolean*
- -17, 0, 1, 2 : *Integer*
- -17.89, 0.01, 3.14 : *Real*
- "Hello World" : *String*

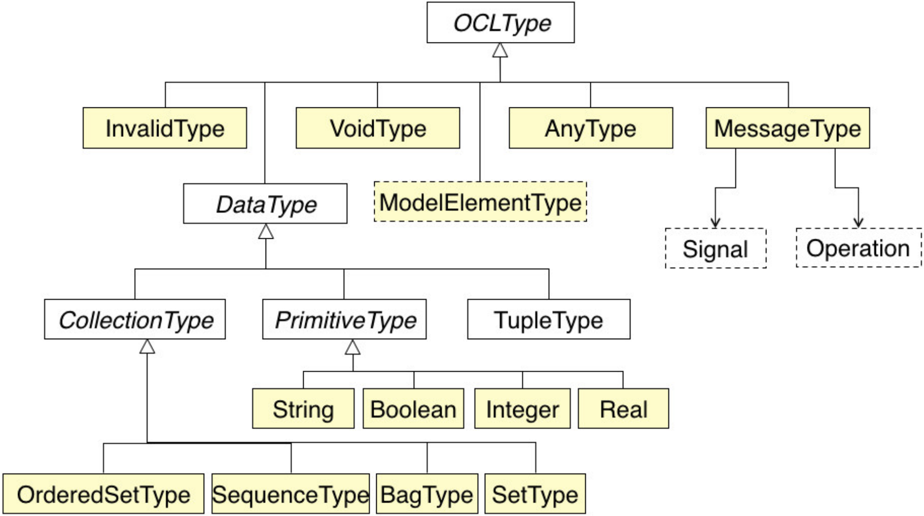
Set-valued, parameterized types

- 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)



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**

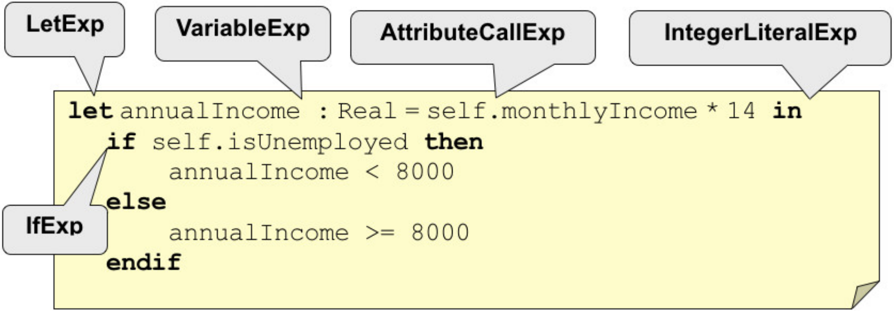
Simple OCL expressions

- *LiteralExp, IfExp, LetExp, VariableExp, LoopExp*

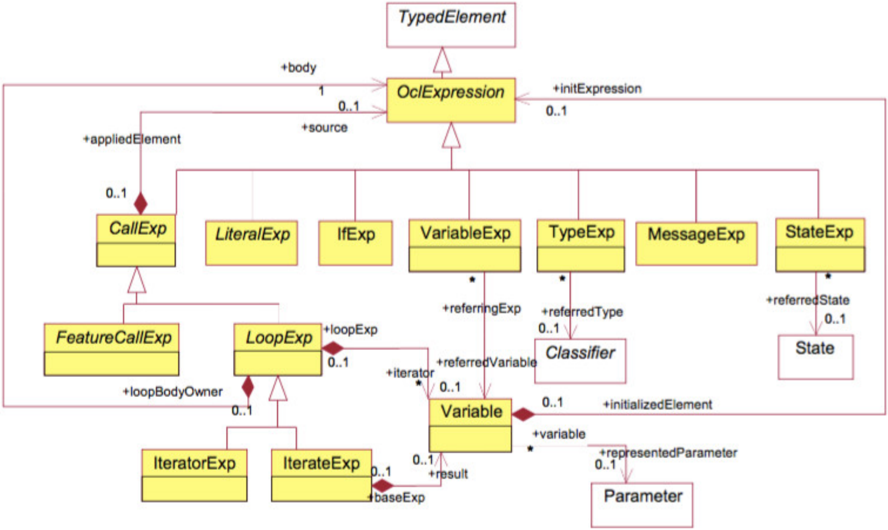
OCL expressions for querying model information

- *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!

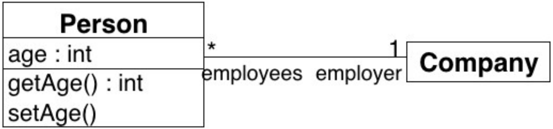
Example for *LetExp, IfExp, VariableExp, AttributeCallExp, IntegerLiteralExp*



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



LiteralExp: *CollectionLiteralExp, PrimitiveLiteralExp, TupleLiteralExp, EnumLiteralExp*



Context instance

- `context` **Person**

AttributeCallExp

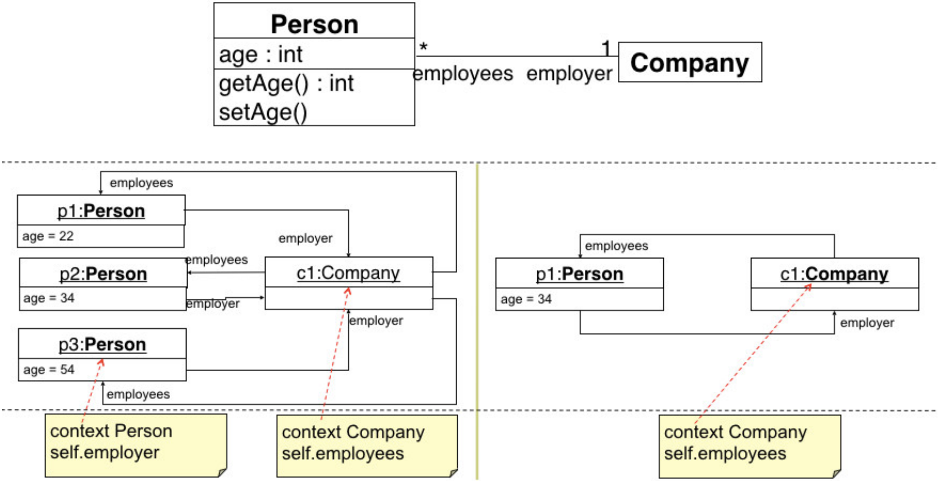
- `self.age : int`

OperationCallExp

- Operations must not have **side effects**
- Allowed: `self.getAge() : int`
- **Not allowed**: `self.setAge()`

AssociationEndCallExp

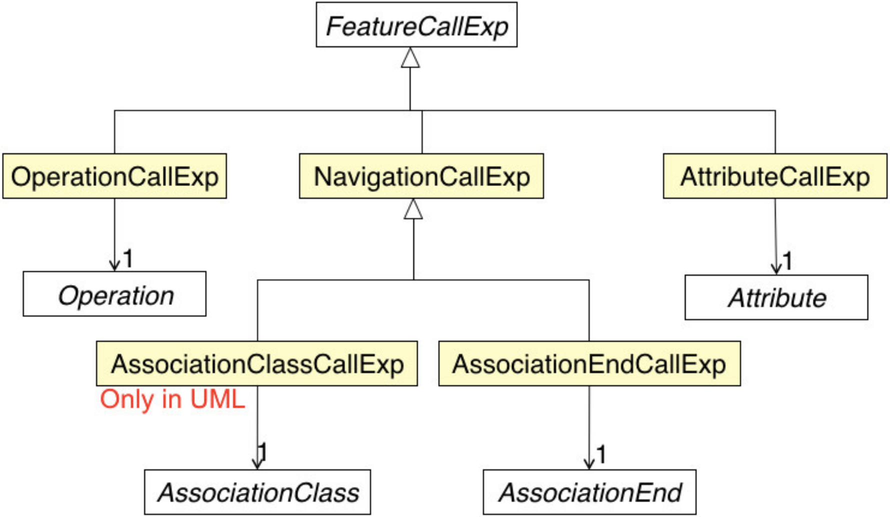
- Navigate to the opposite association end using role names
  - `self.employer` – Return value is of type ***Company***
- Navigation often results into a set of objects – Example
  - `context Company`
  - `self.employees` – Return value is of type **Set (Person)**



c1 : Company

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

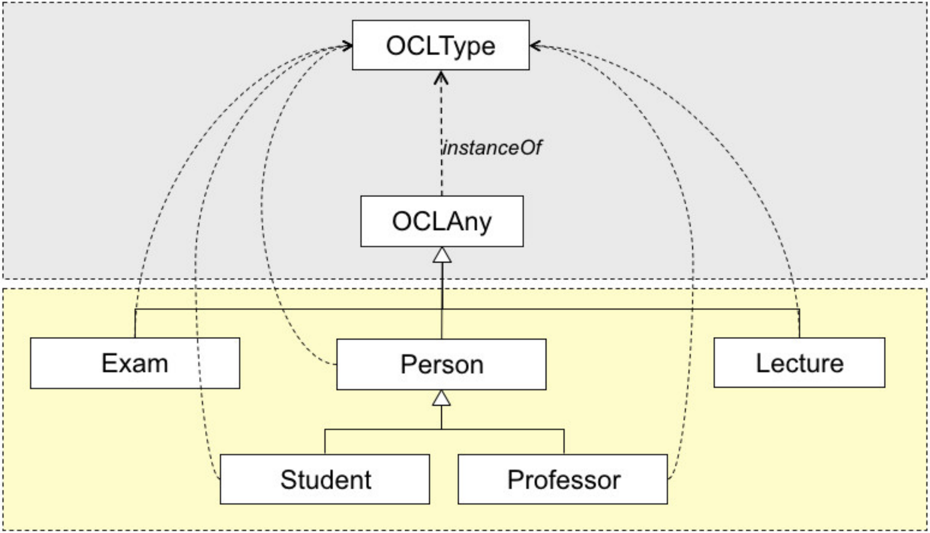
Set{p1} : Set(Person)


FeatureCallExp
OperationCallExp
NavigationCallExp
AttributeCallExp
1
Operation
1
Attribute
AssociationClassCallExp
AssociationEndCallExp
Only in UML
1
1
AssociationClass
AssociationEnd

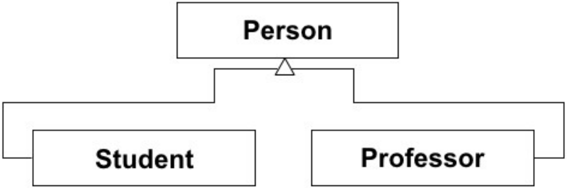
- *OclAny* - **Supertype** of all other types in OCL
  - **Operations** are **inherited** by all other types.
- **Operations** of *OclAny* (extract)
  - Receiving object is denoted by *obj*

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*

- **Predefined** simple types
  - Integer Z
  - Real R
  - Boolean true, false
  - String ASCII, Unicode
- Each simple type has predefined operations [2]

Simple Type	**Predefined Operations**
Integer	*, +, -, /, abs(), . . .
Real	*, +, -, /, floor(), . . .
Boolean	and, or, xor, not, implies
String	concat(), size(), substring(), . . .

[2] **implies** semantic: A **implies** B = **not** A **or** B

- **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: Semantic

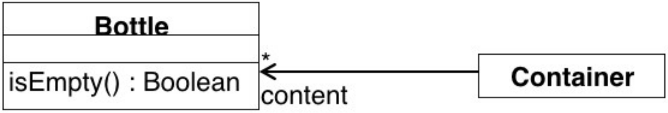
- OCL is based on a **three-valued (trivalent) logic**
  - Expressions are mapped to the three values `{true, false, undefined}`
- ***Undefined***: Return value if an expression fails
  1. Access on the first element of an empty set
  2. Error during Type Casting
  3. ...

- 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`

- 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*

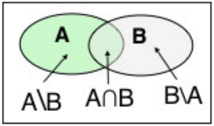
Operation	**Explanation of Result**
size():Integer	Number of elements in *coll*
includes(obj:OclAny):Boolean	True, if *obj* exists in *coll*
isEmpty:Boolean -> Boolean	True, if *coll* contains no elements
sum:T	Sum of all elements in *coll* Elements have to be of type Integer or Real

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

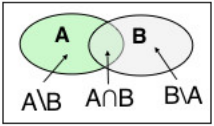
- *Set* and *Bag* define additional operations
  - Generally based on **theory of set concepts**



- **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 exist 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

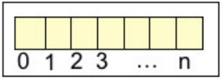
- *Set* and *Bag* define additional operations
  - Generally based on **theory of set concepts**



- **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*

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



- **Operations** of **OrderedSet** (extract)
  - Receiving object is denoted by *orderedSet*
  - Note: Operations on Sequence are analogous to the operations of 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 element on position *lower* and 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*

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(. . .)

- **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 l booleanExp(v) )
collection -> select( v l booleanExp(v) )
collection -> select( booleanExp )
```

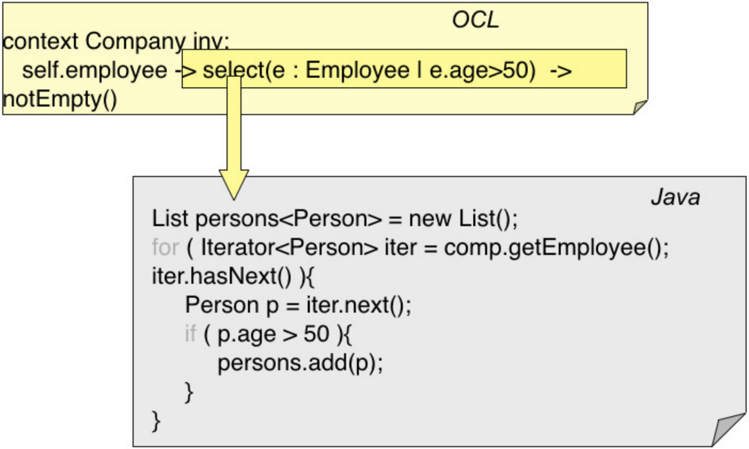
- Reject
  - **Result:** Subset of collection, including elements where *booleanExpr* is **false**
  - Just *Syntactic* Sugar, because each *reject-Operation* can be defined as a *select-Operation* with a negated expression

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

=

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

- Semantic of the *Select-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*

- 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()
```

Bag
(with duplicates)

Set
(without
duplicates)

- **ForAll** checks, if all elements of a collection evaluate to true
- **Example:** self.employees -> forAll(age > 18)

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

- **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
- Example: employees -> exists(e: Employee | e.isManager = true)

- **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))

- Semantic of the *Iterate-Operator*

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

Java
```
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

Further reading...

OCL is used in diverse tools

- OCL is used in ATL
  - See https://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language
  - See https://wiki.eclipse.org/ATL/EMFTVM for specifics about the EMFTVM mode of ATL (the mode we use in this course)
- OCL is used by EMF
  - OCL is used in OCLInEcore
  - OCL is used in interactive OCL consoles in eclipse
  - See http://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FOverviewandGettingStarted.html