A presentation of OCL 2

Object Constraint Language

Christian Hein, Fraunhofer FOKUS

April 2006

Context of this work

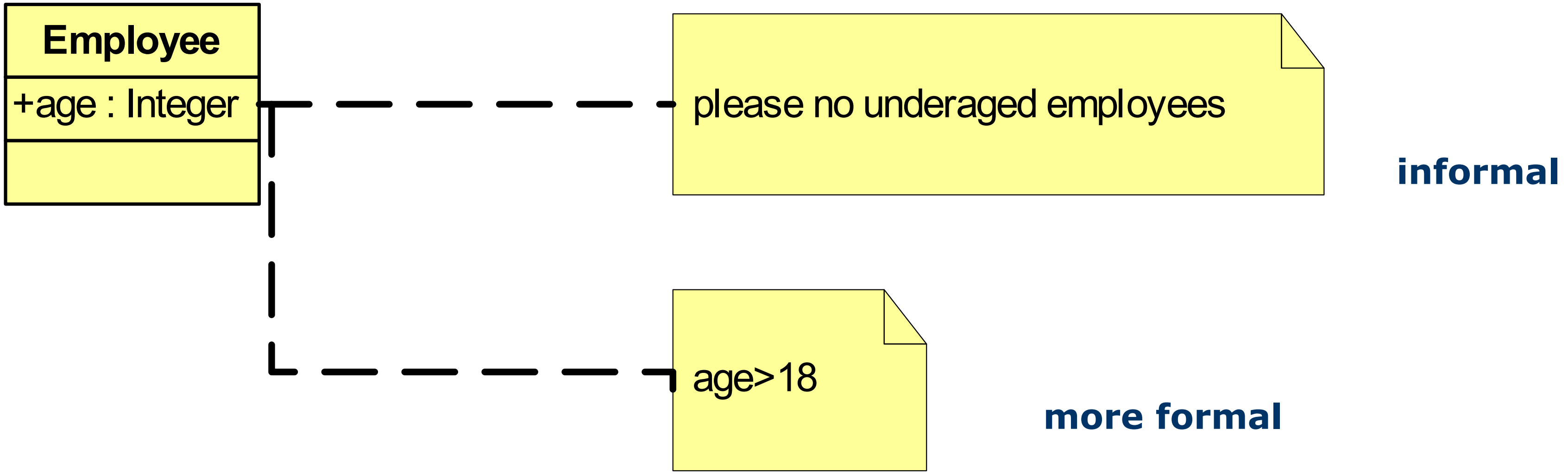
- The present courseware has been elaborated in the context of the MODELWARE European IST FP6 project (http://www.modelware-ist.org/).
- Co-funded by the European Commission, the MODELWARE project involves 19 partners from 8 European countries. MODELWARE aims to improve software productivity by capitalizing on techniques known as Model-Driven Development (MDD).
- To achieve the goal of large-scale adoption of these MDD techniques, MODELWARE promotes the idea of a collaborative development of courseware dedicated to this domain.
- The MDD courseware provided here with the status of open source software is produced under the EPL 1.0 license.

Overview

- Motivation and short history
- OCL
  - structure of an OCL constraint
  - basic types
  - accessing objects and their properties
  - collections

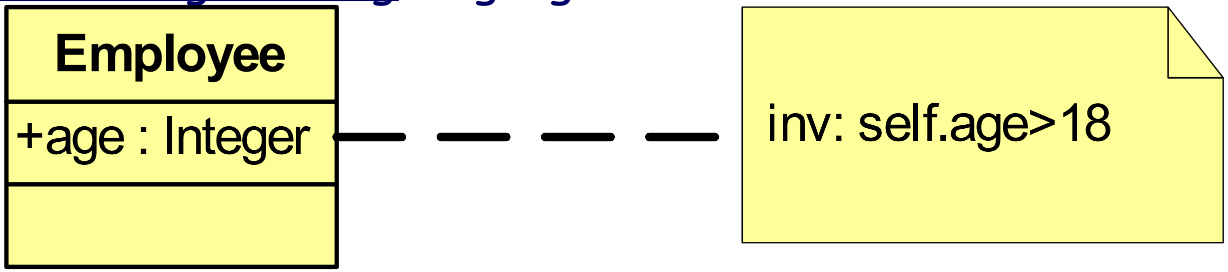
Motivation

- Graphic specification languages such as UML can describe often only partial aspects of a system
- Constraints are often (if at all) described as marginal notes in natural language
  - almost always ambiguous
  - imprecise
  - not automatically realizable/checkable
- Formal Languages are better applicable



Motivation 2

- Traditional formal languages (e.g. Z) require good mathematical understanding from users
  - mostly applied in academic world, not in industry
  - hard to learn, to complex in application
- The Object Constraint Language (OCL) has been developed to achieve the following goals:
  - formal, precise, unambiguous
  - applicable for a large number of users (business or system modeler, programmers)
  - Specification language
  - not a Programming language



History

- Developed in 1995 from IBM's Financial Division
  - original goal: business modeling
  - Insurance department
  - derived from S. Cook's „Syntropy"
- Belongs to the UML Standard since Version 1.1 (1997)
- OCL 2.0 Final Adopted Specification (ptc/ 03-10-14) October 2003
- developed parallel to UML 2.0 and
  - core OCL (basic or essential OCL)

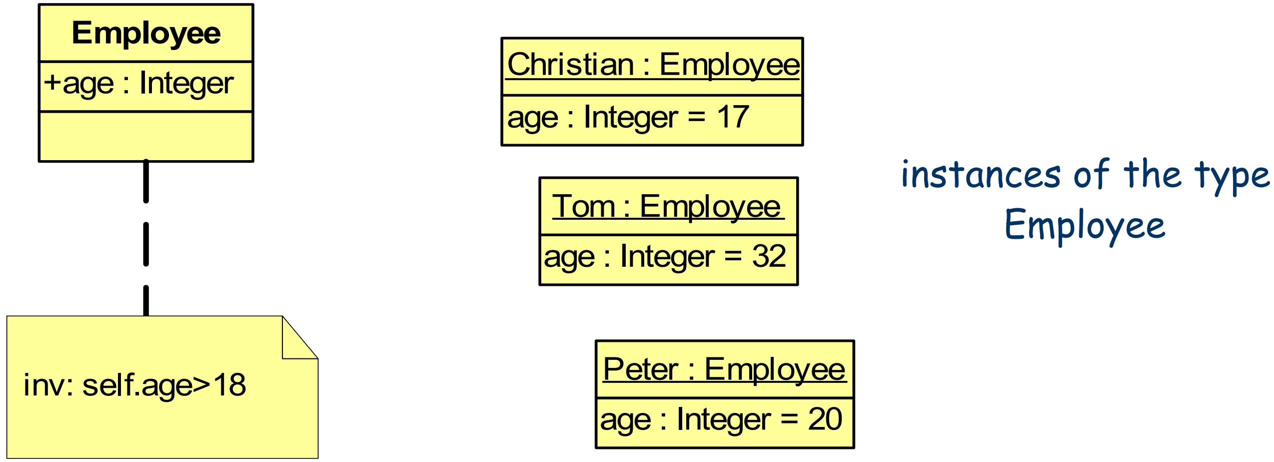
Language features

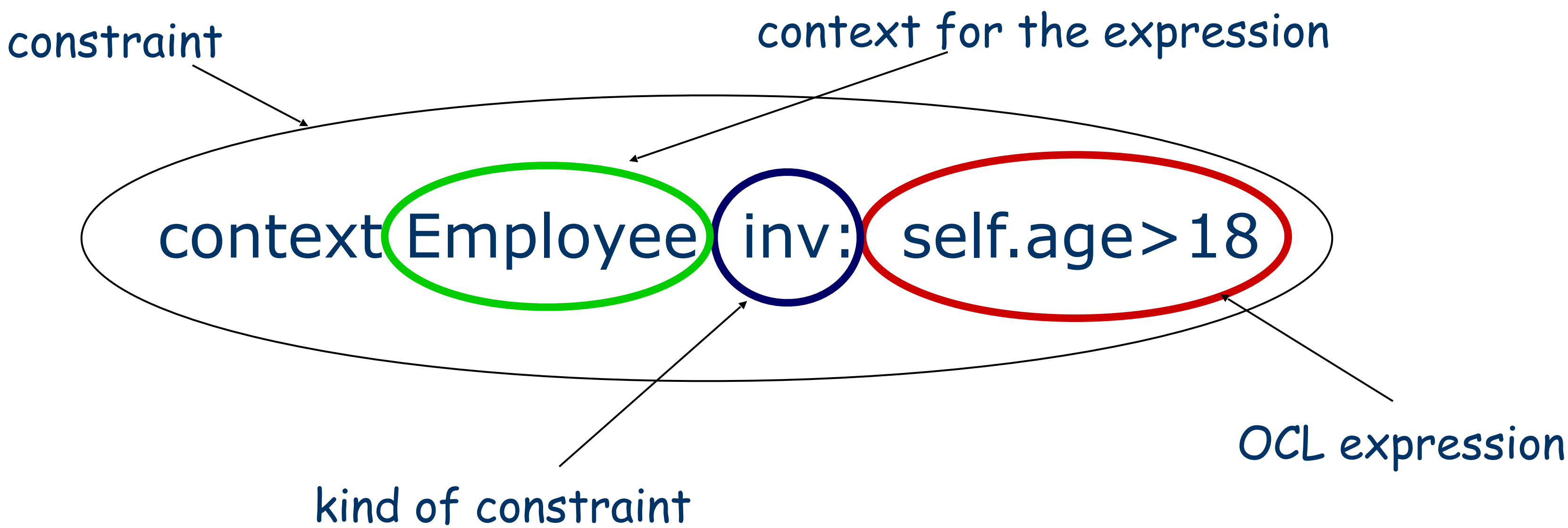
- Specification language without side effects
- Evaluation of an OCL expression returns a value – the model remains unchanged! (even though an OCL expression maybe used to specify a state change (e.g., post-condition) the state of the system will never change)
- OCL is not a programming language (no program logic or flow control, no invocation of processes or activation of non-query operations, only queries)
- OCL is a typed language, each OCL expression has a type. It is not allowed to compare Strings and Integers
- Includes a set of predefined types
- The evaluation of an OCL expression is instantaneous, the states of objects in a model cannot change during evaluation

Where to use OCL

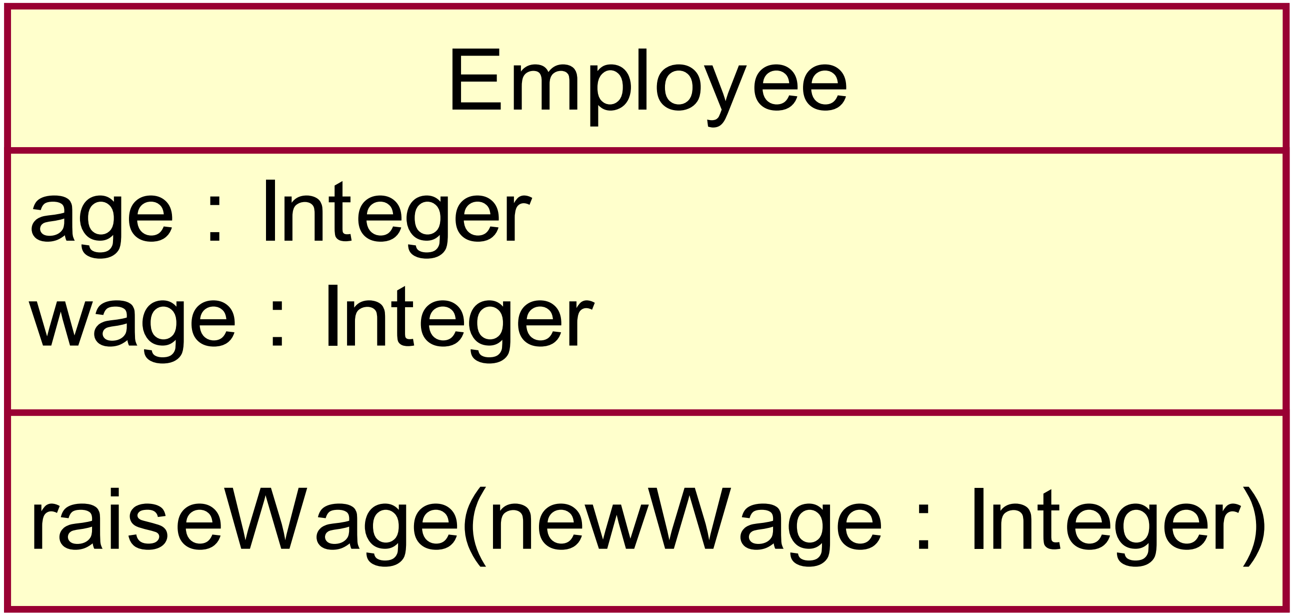
- Constraints specification for model elements in UML models
    - Invariants
    - Pre- and post conditions (Operations and Methods)
    - Guards
    - Specification of target (sets) for messages and actions
    - initial or derived values for attributes & association ends
- As „query language"
- Constraints specification in metamodels based on MOF or Ecore
    - metamodels are also models
    - possible kinds of constraints
        - invariants, pre- and post conditions, initial or derived values

OCL Constraint





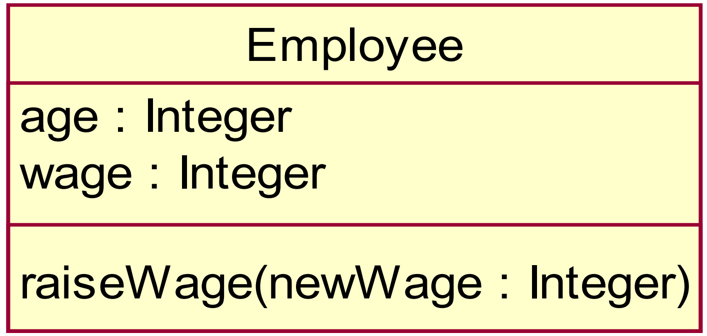
kind of constraints (Invariants)



- **inv** invariant: constraint must be true
  - for all instances of constrained type at any time
  - Constraint is always of the type Boolean

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

kind of constraints 2 (Pre- and Postconditions)



- **pre** precondition: constraint must be true, before execution of an Operation
- **post** postcondition: constraint must be true, after execution of an Operation
  - `self` refers to the object on which the operation was called
  - `return` designates the result of the operation (if available)
  - The names of the parameters can also be used

```
context Employee::raiseWage(newWage:Integer)
 pre: newWage > self.wage
 post: wage = newWage
```

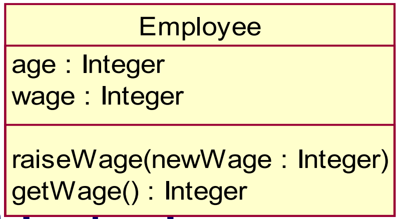
kind of constraints 3 (others)

- **body** specifies the result of a query operation
  - The expression has to be conformed to the result type of the operation

```
context Employee::getWage() : Integer
body: self.wage
```

- **init** specifies the initial value of an attribute or association end
  - Conformity to the result type + Mulitiplicity

```
context Employee::wage
init: wage = 900
```



- **derive** specifies the derivation rule of an attribute or association end

```
context Employee::wage
derive : wage = self.age * 50
```

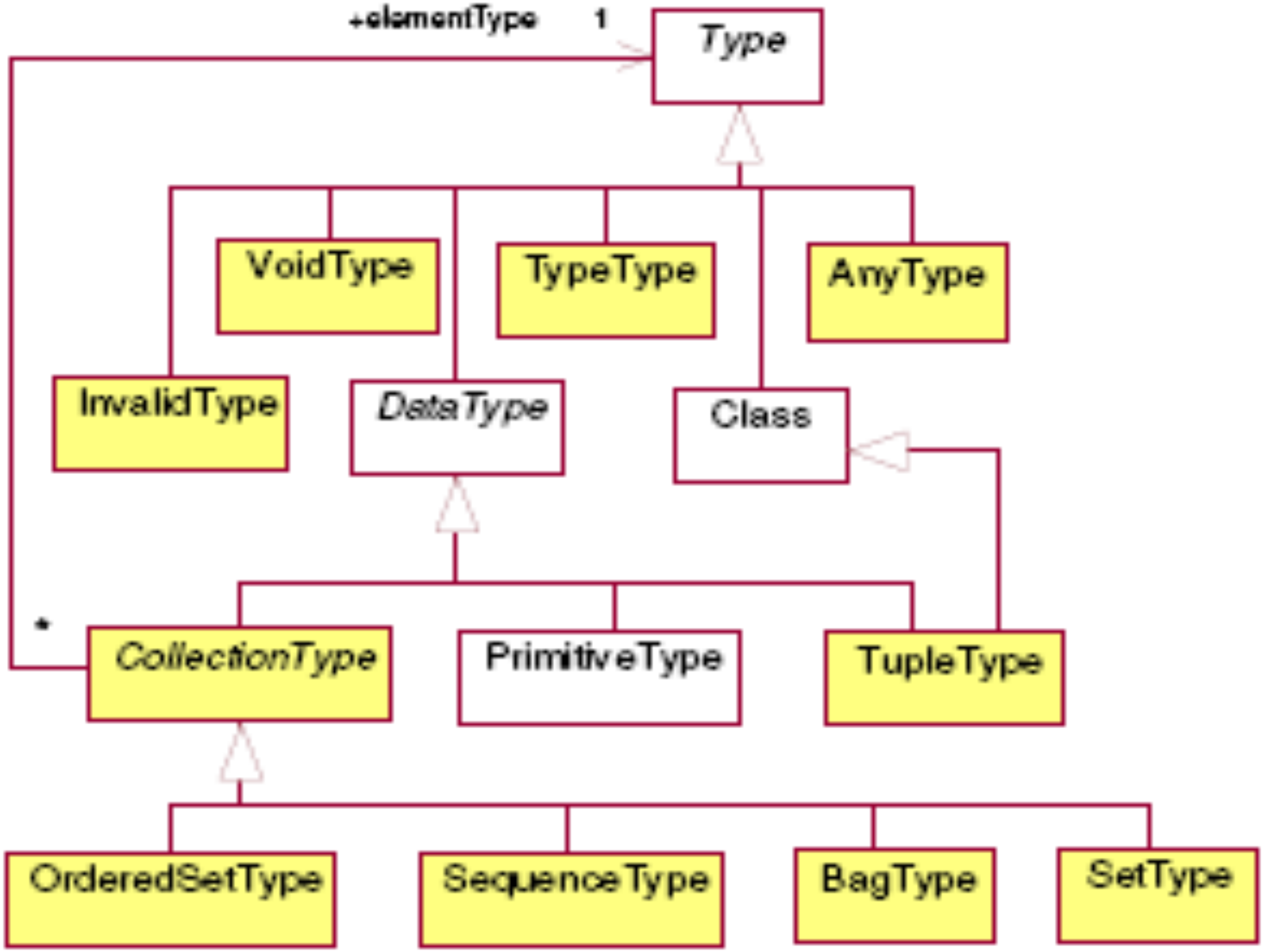
- **def** enables reuse of variables/operations over multiple OCL expressions

```
context Employee
def: annualIncome : Integer = 12 * wage
```

OCL Metamodel

- OCL 2.0 has MOF Metamodel
- The Metamodel reflects OCL's abstract syntax
- Metamodel for OCL Types
  - OCL is a typed language
    - each OCL expression has a type
    - OCL defines additional to UML types:
      - CollectionType, TupleType, OclMessageType,....
- Metamodel for OCL Expressions
  - defines the possible OCL expressions

OCL Types Metamodel



OCL Types

- Primitive Types
  - Integer, Real, Boolean, String
  - OCL defines a number of operations on the primitive types
  - + , - , * , / , min() , max() , … , for Integer or Real
  - concat(), size(), substring() , … , for String
- OCLModelElementTypes
  - All Classifiers within a model, to which OCL expression belongs, are types
- Collection Types
  - CollectionType is abstract, has an element type, which can be CollectionType again
  - Set: contains elements without duplicates, no ordering
  - Bag: may contain elements with duplicates, no ordering
  - Sequence: ordered, with duplicates
  - OrderedSet: ordered, without duplicates

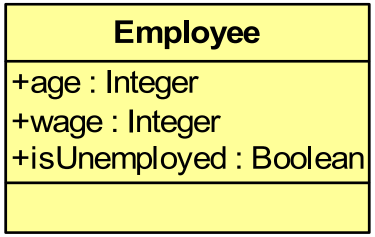
OCL Types 2

- TupleType
  - Is a “Struct” (combination of different types into a single aggregate type)
  - is described by its attributes, each having a name and a type
- VoidType
  - Is conform to all types

Basic constructs for OCL expressions

- **Let, If-then-else**

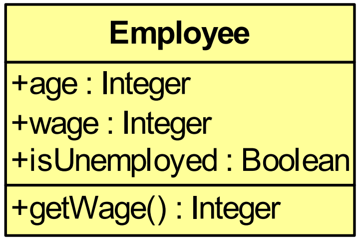
```
context Employee inv:
let annualIncome : Integer = wage * 12 in
if self.isUnemployed then
   annualIncome < 5000
else
   annualIncome >= 5000
endif
```



- **Let expression allows to define a (local) variable**
- **If-then-else construct (complete syntax)**

```
if <boolean OCL expression>
then <OCL expression>
else <OCL expression>
endif
```

Accessing objects and their properties (Features)



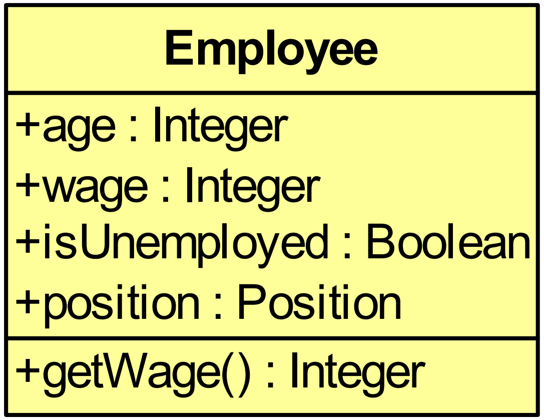
- Attribute:

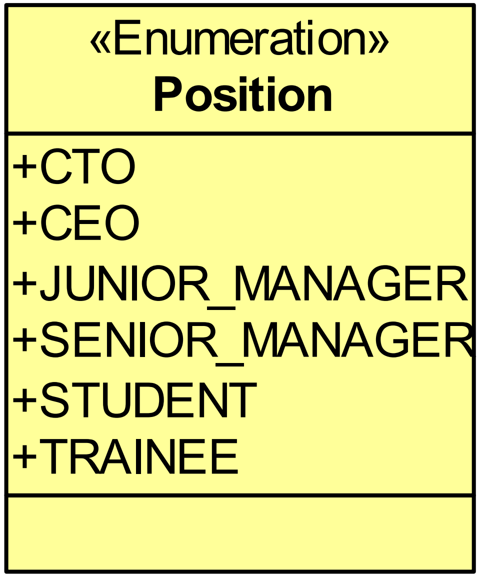
```
context Employee inv: self.age > 18
context Employee inv: self.wage < 10000
context Employee inv: self.isUnemployed
```

- Operations:

```
context Employee inv: self.getWage() > 1000
```

Accessing objects and their properties (Features) 2

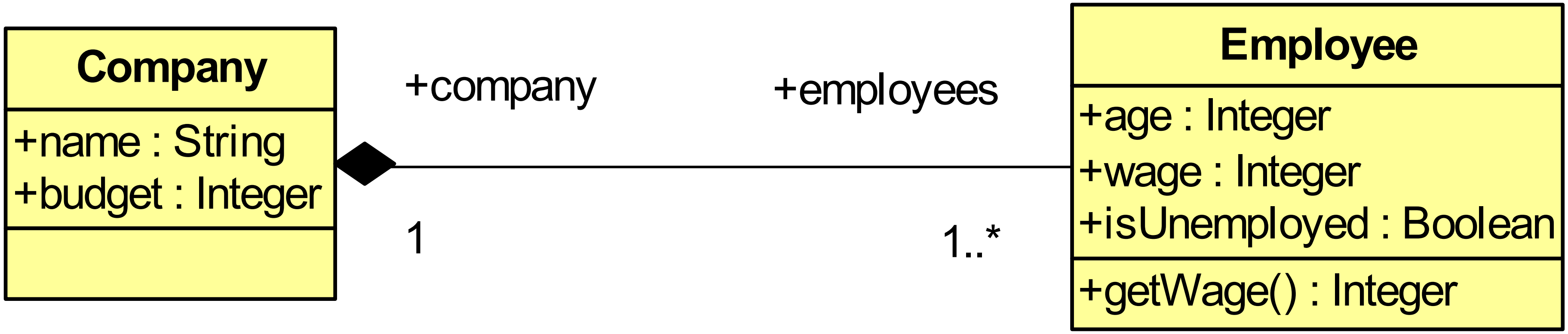




- Accessing enumerations with ´::´

```
context Employee inv:
self.position=Position::TRAINEE implies self.wage<500
```

Accessing objects and their properties (Features) 3



- Association ends:
  - allow navigation to other objects
  - result in `Set`
  - result in `OrderedSet`, when association ends are ordered

```
context Company inv: if self.budget<50000
then self.employees->size() < 31
else true
endif
```

Collections Operations (Iterations)

- some defined operations for collections
  - isEmpty(), size(), includes(),....
- Iteration operations
  - Select/Reject
  - Collect
  - ForAll
  - Exists
  - Iterate

Collections Operations (Iterations) 2

- **select** and **reject** create a subset of a collection
  - (result: Collection)

```
context Company inv:
   self.employees->select(age < 18) -> isEmpty()
```

  - Expression will be applied to all elements within the collection, context is then the related element

```
context Company inv:
   self.employees->reject(age>=18)-> isEmpty()
```

Collections Operations (Iterations) 3

- `collect` specifies a collection which is derived from some other collection, but which contains different objects from the original collection (resulttype: Bag or Sequence)

```
context Company inv: self.employees->collect(wage)
  ->sum()<self.budget
-- collect returns a Bag of Integer
```

- Shorthand notation

```
self.employees.age
```

- Applying a property to a collection of elements will automatically be interpreted as a collect over the members of the collection with the specified property

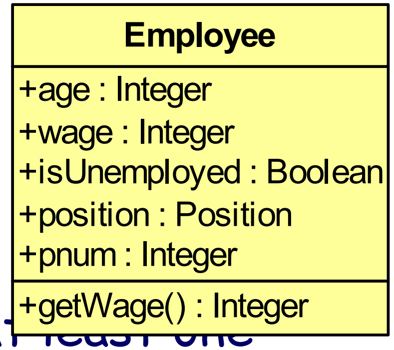
Collections Operations (Iterations) 4

- **forAll** specifies expression, which must hold for all objects in a collection (resulttype: Boolean)

```
context Company inv: self.employees->forAll(age > 18)
```

  - Can be nested

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



- **exists** `returns` true if the expression is true for at least one element of collection (resulttype: Boolean)

```
context Company inv:
self.employees->exists(e|e.pnum=1)
```

Collections Operations (Iterations) 5

- **iterate** is the general form of the Iteration, all previous operations can be described in terms of iterate

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

  - elem is the iterator, variable acc is the accumulator, which gets an initial value <expression>.
  - Example SELECT operation:

```
collection-> select(iterator | body)
-- is identical to:
collection->iterate(iterator; result : Set(T) = Set{} |
if body
then result->including(iterator)
else result
endif )
```

Predefined Operations

- OCL defines several Operations that apply to all objects
- **oclIsTypeOf(t:OclType):Boolean**
  - results is true if the type of self and t are the same

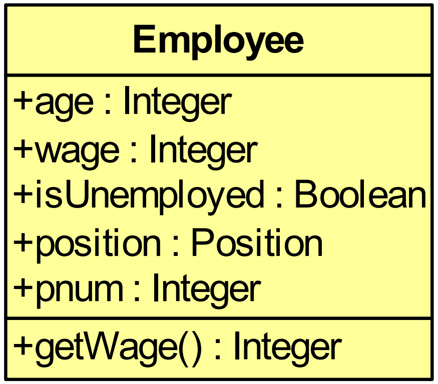
```
context Employee inv:
self.oclIsTypeOf(Employee) -- is true
self.oclIsTypeOf(Company)  -- is false
```

- **oclIsKindOf(t:OclType):Boolean**
  - determines whether t is either the direct type or one of the supertypes of an object
- **oclIsNew():Boolean**
  - only in postcondition: results is true if the object is created during performing the operation

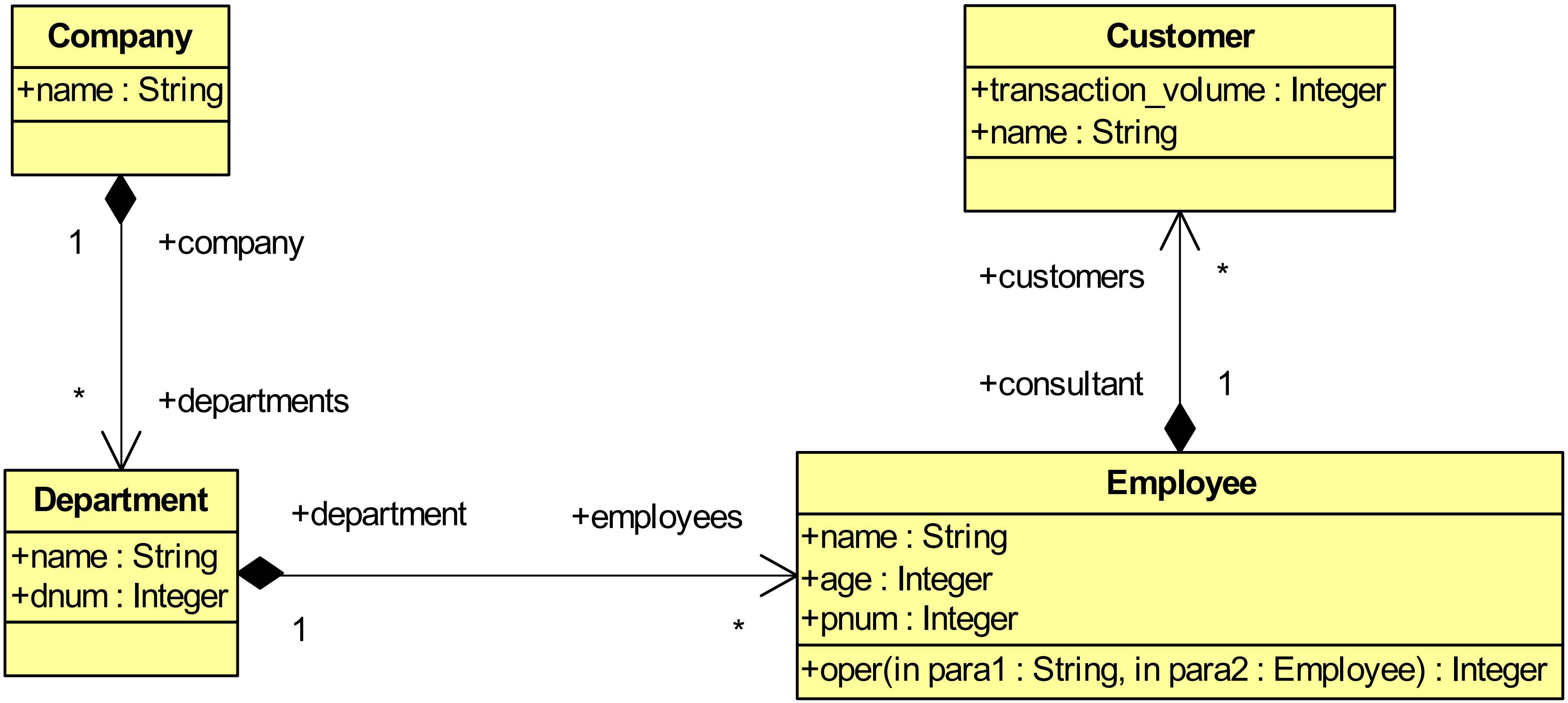
Predefined Operations 2

- **`oclAsType(t:OclType):T`**
  - results in the same object, but the known type is the OclType
- **`allInstances`**
  - predefined feature on classes, interfaces and enumerations
  - results in the collection of all instances of the type in existence at the specific time when the expression is evaluated

```
context Company inv:
Employee.allInstances()->forAll(p1|
    Employee.allInstances()->forAll(p2|
    p1 <> p2 implies p1.pnum <> p2.pnum)
```



example model



Tips & Tricks to write better OCL (1/5)

- Keep away from complex navigation expressions!
  - a customer bonusprogram have to be funded if a customer exists which have a transaction volume more than 10000

```
context Company
inv: departments.employees.customers->exists(c|c.volume>10000)
   implies bonusprogram.isfunded
```

⬇

```
context Department
def: reachedVolume:Boolean = employees.customers-> exists(c|
   c.volume>10000)

context Company
inv: departments->exists(d|d.reachedVolume) implies
   bonusprogram.isfunded
```

Tips & Tricks to write better OCL (2/5)

- Choose context wisely (attach an invariant to the right type)!

  - two persons who are married are not allowed to work at the same company:

```
context Person
inv: wife.employers>intersection(self.employers)
->isEmpty() and husband.employers
->intersection(self.employers)->isEmpty()
```

⬇

```
context Company

inv: employees.wife->intersection(self.employees)->isEmpty()
```

Tips & Tricks to write better OCL (3/5)

- Avoid allInstances operation if possible!
  - results in the set of all instances of the modeling element and all its subtypes in the system
  - problems:
    - the use of allInstances makes (often) the invariant more complex
    - in most systems, apart from database systems, it is difficult to find all instances of a class

```
context Person
inv: Person.allInstances->
forAll(p| p. parents->size <= 2)
```

```
context Person
inv: parents->size <= 2
```

Tips & Tricks to write better OCL (4/5)

- Split complicated constraint into several separate constraints !
  - Some advantages:
    - each invariant becomes less complex and therefore easier to read and write
    - the simpler the invariant, the more localized the problem
    - maintaining simpler invariants is easier

```
context Company inv: self.employees.wage-> sum()<self.budget and
self.employees->forAll (e1 | self.employees ->forAll (e2 | e1 <> e2
implies e1.pnum <> e2.pnum)) and self.employees->forAll(e|e.age>20)
```

```
context Company
inv: self.employees.wage->sum()<self.budget
inv: self.employees->forAll (e1 | self.employees->forAll (e2|e1<>
    e2 implies e1.pnum <> e2.pnum))
inv: self.employees->forAll(e|e.age>20)
```

Tips & Tricks to write better OCL (5/5)

- Use the collect shorthand on collections!

```
context Person
inv: self.parents->collect(brothers) -> collect(children)->notEmpty()
```

↓

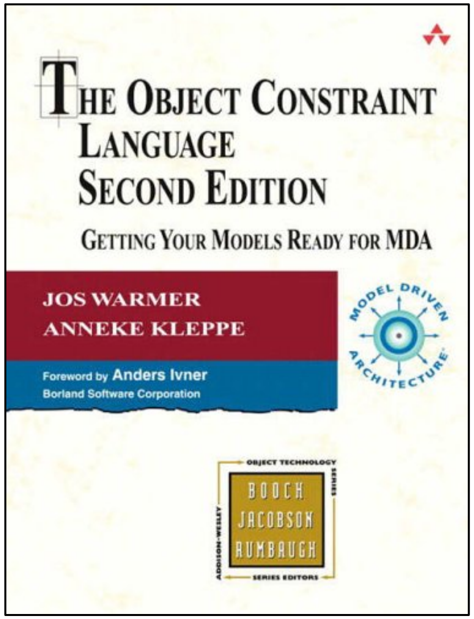
```
context Person inv: self.parents.brothers.children->notEmpty()
```

- Always name association ends!
  - indicates the purpose of that element for the object holding the association
  - helpful during the implementation: the best name for the attribute (or class member) that represents the association is already determined

Summary

- focus was on the "core" part of OCL
- core OCL can be used for UML2 as well as MOF metamodels
- constraint for metamodels can be used for computing metrics or check design guidelines
- additional courseware about some of these topics is available

References



- Jos Warmer and Anneke Kleppe – The Object Constraint Language – Second Edition
- OCL 2.0 Final Adopted Specification (ptc/03-10-14)
- UML 2.0 Infrastructure Specification: (formal/05-07-05)