A presentation of OCL 2

Object Constraint Language

Fraunhofer FOKUS

Context of this work

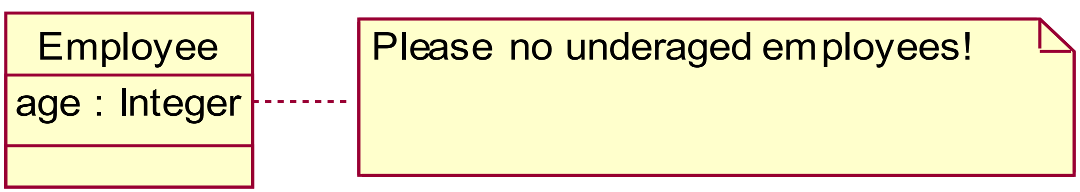
- The present courseware has been elaborated in the context of the MODELWARE European IST FP6 project (http://www.modelware-ist.org/).
- Co-funded by the European Commission, the MODELWARE project involves 19 partners from 8 European countries. MODELWARE aims to improve software productivity by capitalizing 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
- Introduction and short History
- Applying OCL
    - Relation to the UML Metamodel
    - Basic types
    - Objects und their Properties
    - Collections
    - Messages
- Tools

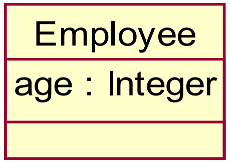
Motivation

- 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 suitable



Motivation 2

- Traditional formal languages (e.g. Z) require good mathematical understanding from users
  - Applying and distribution only in academic world, not in industry
  - hard to learn, to complex in application
  - Problem: „large“ systems
- 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
  - tool support




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

History

- Developed in 1995 from IBM's Financial Division
    - original goal: business modelling
    - 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
- Aligned with UML 2.0 and MOF 2.0

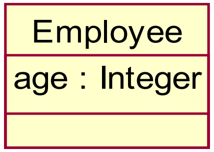
Language features

- Specification language without side effects
- Evaluation of an OCL expression returns a value – the model remains unchanged! (even though an OCL expression is 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 typed language, each OCL expression has a type. It is not allowed to compare Strings and Integers
- Each Classifier defined in model represents a distinct OCL type
- 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 (MOF) – MOF models are also models

Relation to the UML Model

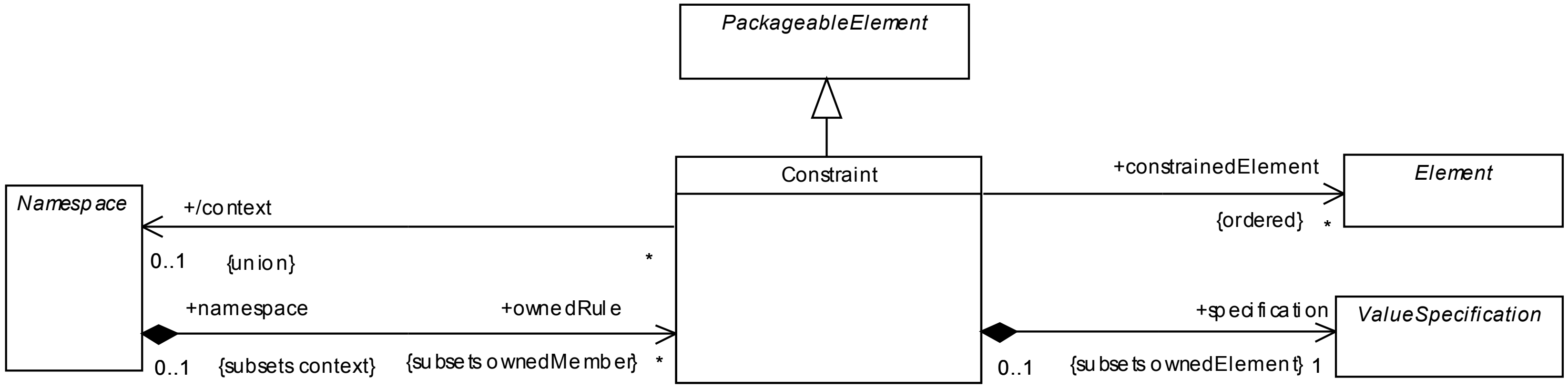



```
context Employee
inv: self.age > 18
```

- Each OCL expression is related to an instance of a model element
- Context declaration is used to determine the model element
    - In a diagram: dashed line to the element, which the defined OCL constraint refer to
- **self** refers to the contextual instance

Relation to the UML Model 2

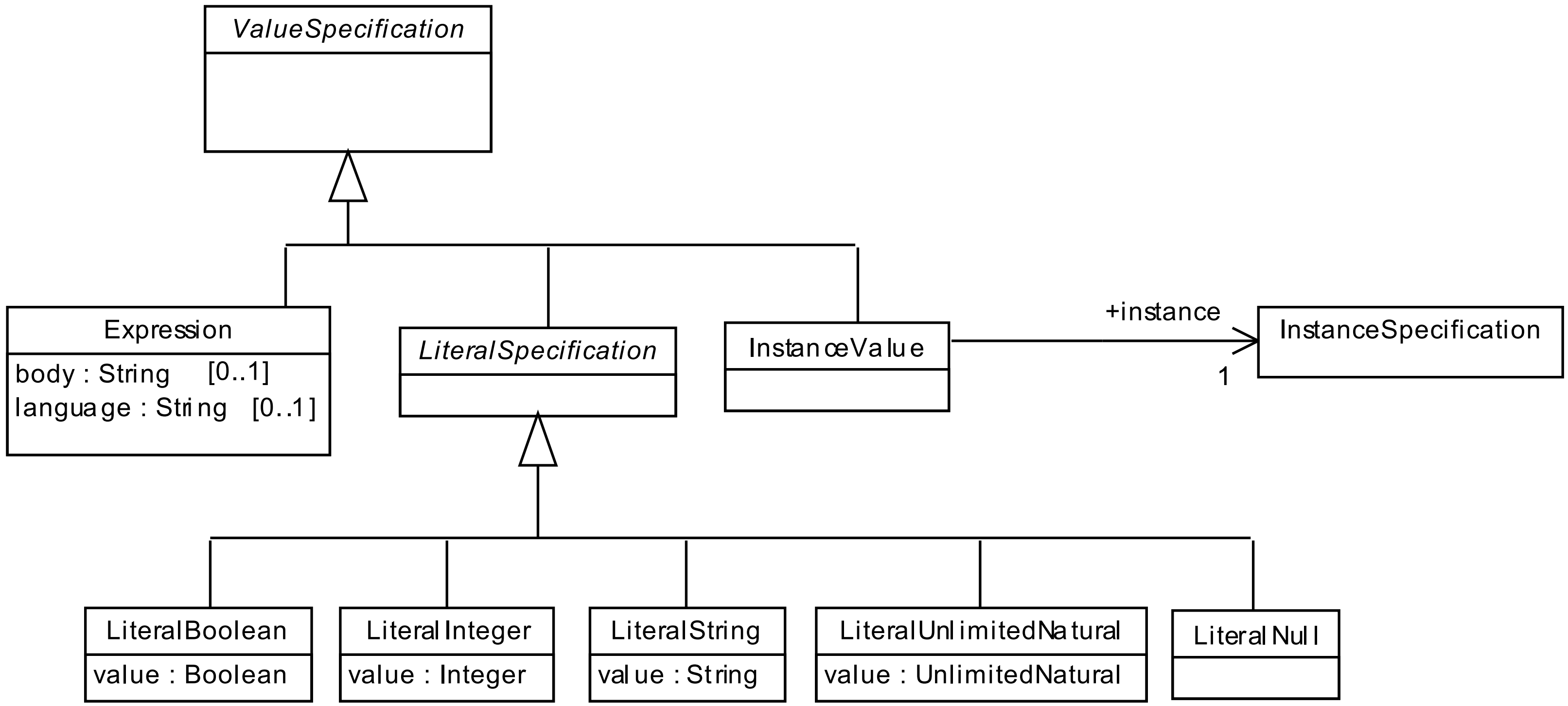
- Constraint is an element of the UML2 metamodel
  - part of the Kernel-Package
  - Describes additional semantic of one or more model elements
  - Language is not predefined: natural language, OCL, Java etc.



- The „Owner" of a constraint determines the time of the constraint evaluation (e.g.: Owner: Operation, time of the evaluation : pre or post)
- Constrained Elements: set of elements referenced by the Constraint.

Relation to the UML Model 3

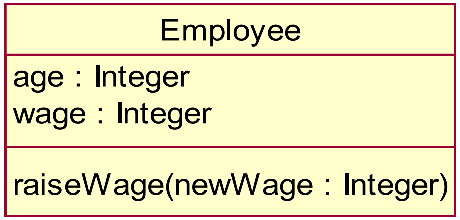
- Value Specification
  - In case of OCL: Expression with Language == „OCL"



Relation to the UML Model 4

- OCL notation in UML model
  - constraint ::= ‘{‘ [<name> ‘:’] <expression>’ }’
  - may follow the element directly (e.g. Attribute)
  - may be placed near the symbol for the element, preferably near the name, if any (e.g. Association End)
  - may be shown as a dashed line between two elements (if a Constraint applies to two elements) labeled by the constraint string (in braces)
  - may be placed in a note symbol

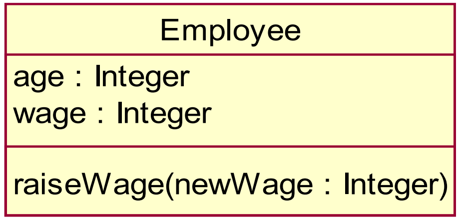
Stereotypes (Constraint types)



- **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
context Employee
  inv age_18: self.age >18
context c : Employee
  inv: c.age > 18
```

Stereotypes (Constraint types) 2



- **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 my_post: wage = newWage
```

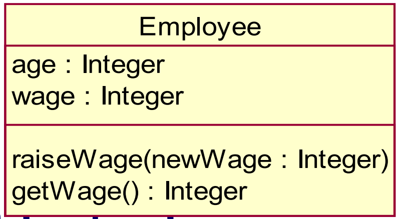
Stereotypes (Constraint types) 3

- **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 : Integer
init: wage = 900
```



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

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

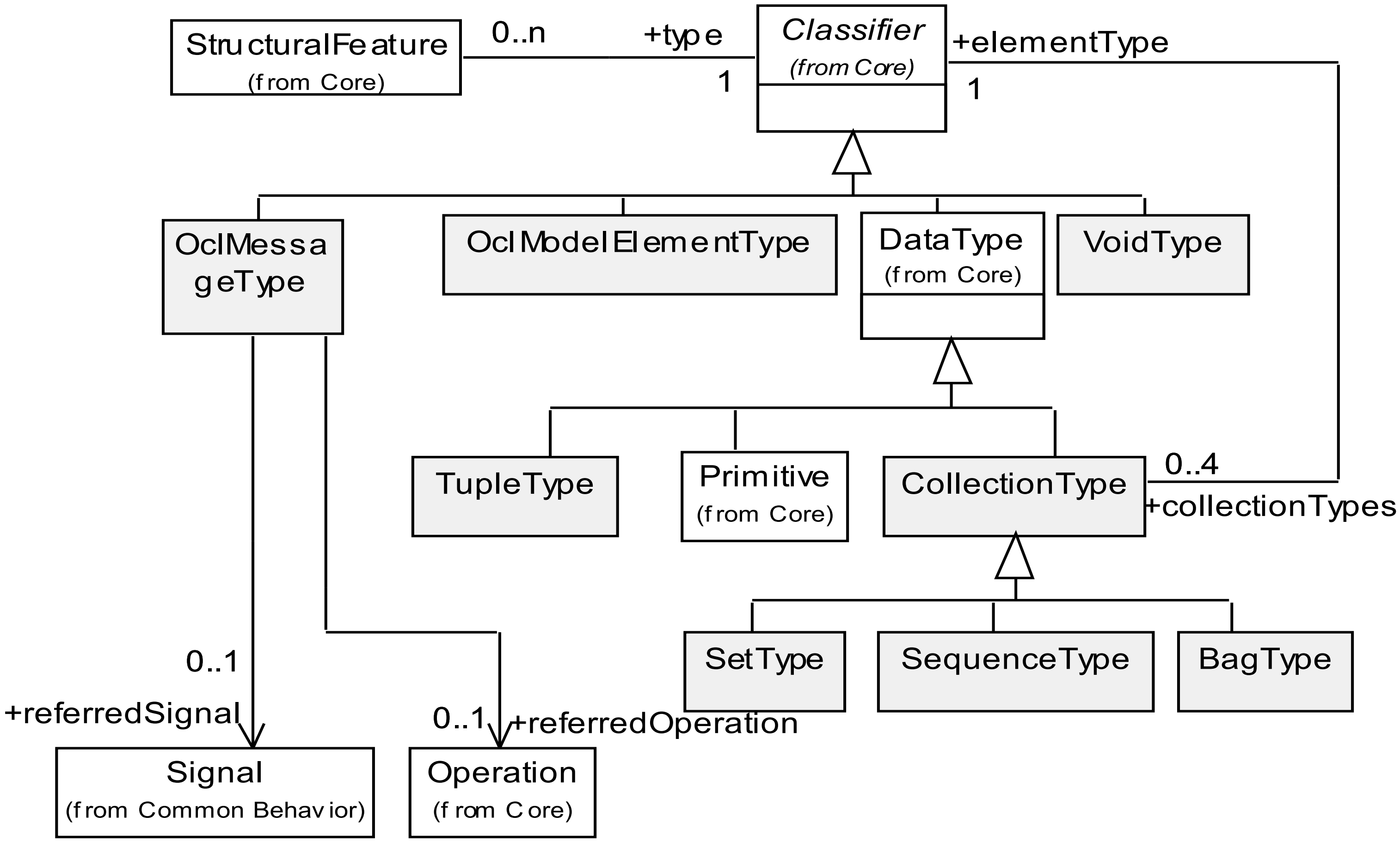
- **def** enables reuse of variables/operations over multiple OCL expressions

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

OCL Metamodel

- OCL 2.0 has (of course ;-)) 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



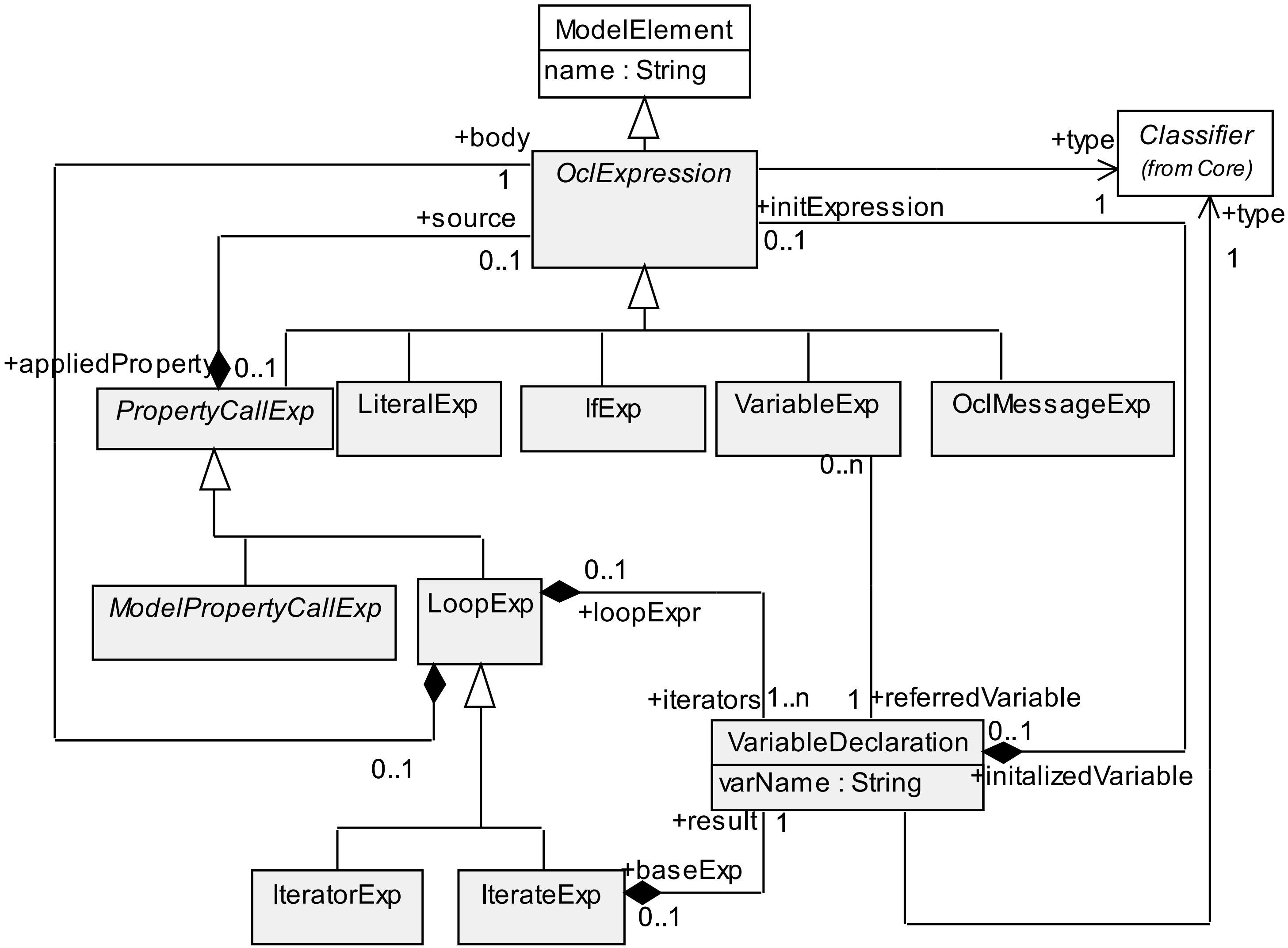
OCL Types

- All Classifier within a UML model, to which OCL expression belongs, are types
  - OCLModelElementType
  - e.g. for `oclIsKindOf`
- Collection Types
  - Are not defined in the Metamodel, exist only implicitly, if they are used (otherwise, infinite since recursive application possible)
  - 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)
  - Has Attribute with a name and a type
- OCLMessageType
  - is used for an access to messages of an operation or signal
  - Statements about the possible sending/receiving of signal/ operations
- VoidType
  - Has only an instance `oclUndefined`
  - Is conform to all types

OCL Expression Metamodel – Basis elements



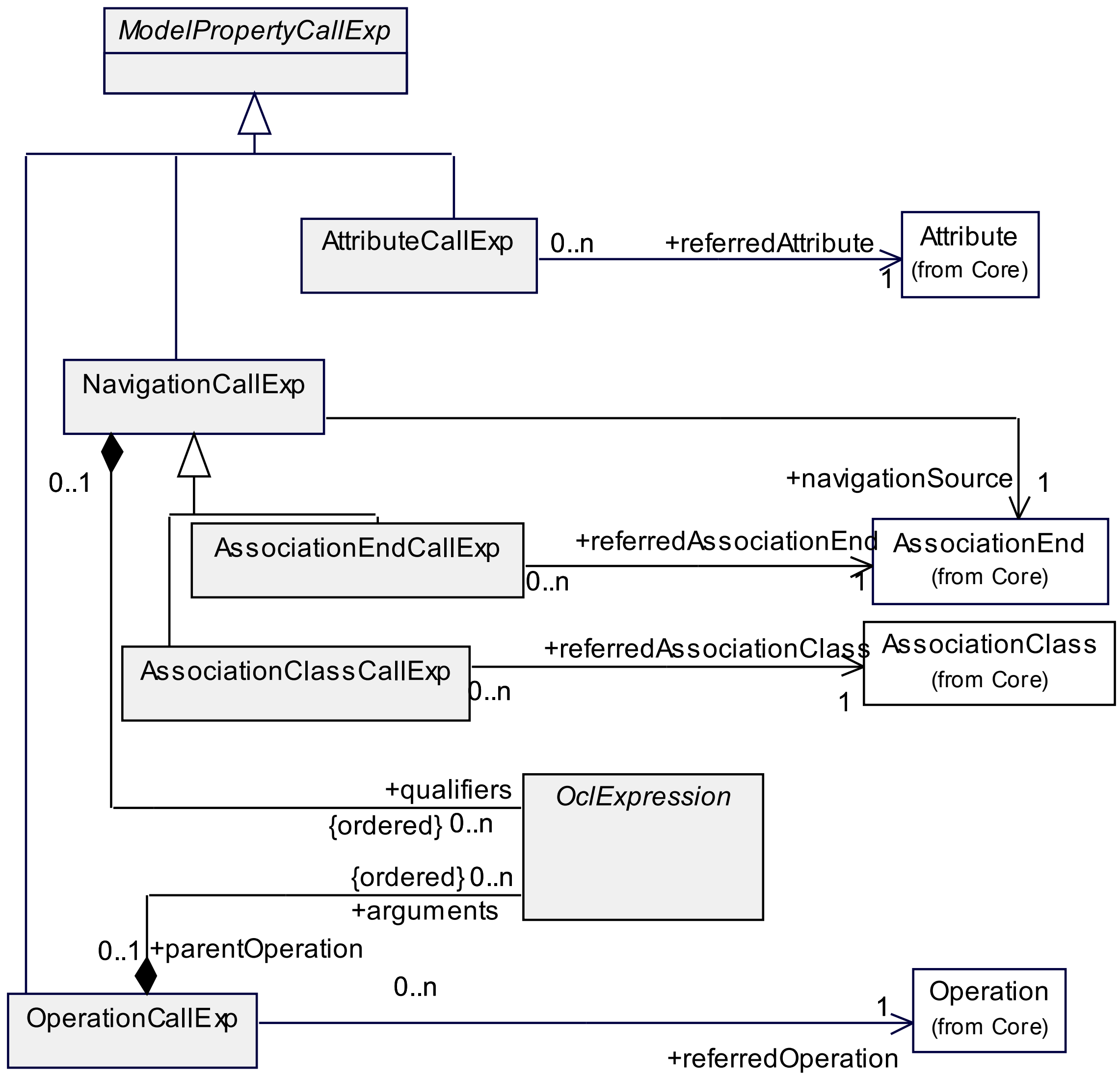
Basic constructs

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

```
let annualIncome : Integer = wage * 12 in
if self.isUnemployed then
   annualIncome < 5000
else
   annualIncome >= 5000
Endif
```

- **Standard Library**
  - Similar to C++, Java, mostly template-based defined
  - Primitive Types:
    - Integer, Real, Boolean, String
    - Operations: +, -, min(), max(), concat()...
  - Collection Types, described as Parameterized Classifier (Template):
    - Set<T>, OrderedSet<T>, Bag<T>, Sequence<T>
    - Operations: size(), includes(), append()...

Accessing objects and their properties



Accessing objects and their properties (Features)

- Attribute:
  ```
  self.age > 18
  ```
- Operations:
  - may no have side effects (only when? allowed)
    - isQuery = true
  ```
  self.getWage() > 1000
  ```
- Association ends:
  - allow navigation to other objects
  - result in `Set`, when multiplicity > 1 und unique
  - result in `OrderedSet`, when Multiplicity > 1 and {ordered,unique}
  - ...
  ```
  self.employer->size() > 0
  ```
- Accessing enumerations with ´::´
  ```
  Gender::male
  ```
- Accessing overridden properties with `oclAsType`
  ```
  context B inv:
     self.oclAsType(A).p1     -- accesses the p1 property
                              -- defined in A
  ```

Collections Operations (Iterations)

- Collections result from Navigation
  - OCL allows elements of collections to be collections themselves
  - Multiplicity of navigated Features determines the Collection type
  - Collection Operations: different constructs for enabling a way of projecting new collections from existing ones
  - Collection Operations do not change the model
- Defined Operations
  - Select/Reject
  - Collect
  - ForAll
  - Exists
  - Iterate

Collections Operations (Iterations) 2

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

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

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

```
collection->select(v : Type | boolean-expression-
  with-v)
collection->select(v | boolean-expression-with-v)
collection->select(boolean-expression)
```

Collections Operations (Iterations) 3

- `collect` specify a collection which is derived from some other collection, but which contains different objects from the original collection (result: Bag)

```
self.employees->collect(age)
-- returns a Set of Integer
```

  - Complete syntax

```
collection->collect( v : Type | expression-with-v )
collection->collect( v | expression-with-v )
collection->collect( expression )
```

  - 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 (result: Boolean)

```
self.employees->forAll(age > 18)

collection->forAll( v : Type | boolean-expression-with-
  v )
collection->forAll( v | boolean-expression-with-v )
collection->forAll( boolean-expression )
```

  - Can be nested

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

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

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:

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

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 in true if the object is created during performing the operation
- **oclIsInState(t:OclState):Boolean**
  - results in true if the object is in the state t

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 Employee inv:
Employee.allInstances()->forAll(p1, p2 |
   p1 <> p2 implies p1.name <> p2.name)
```

Properties in Postconditions

- In a Postcondition property values can be accessed at two times:
  - value at precondition time (before operation execution)
  - value after operation execution
- the **"@pre"** mark can be used in a Postcondition to refer to properties of the previous state

```
context Employee::birthdayHappens()
post: age = age@pre + 1
```

Old values in Postconditions

- If the property (accessed with @pre) is an object, then all further accesses refer to the new value

```
a.b@pre.c -- takes the old value of property b
   of a, say x
-- and then the new value of c of x.
```

- If the object is destroyed, the access result to the current value is oclUndefined
- If the object is created, the access result to the old value is oclUndefined

Messages

- New in OCL 2.0
- Operator hasSent ('^') is used for specifying that during the execution of an operation communication has taken place:

```
context Subject::hasChanged()
post: self.observer^update(8, 15)
```

  - True if an update message with arguments 8 and 15 was sent to observer
  - update() is an Operation or a Signal defined in the UML model
- If the actual arguments of the operation/signal are not specified, operator '?' can be used
- Extra: Type declaration, in order to be able to address operations exactly

```
context Subject::hasChanged()
post: observer^update(? : Integer, ? : Integer)
```

Messages 2

- Message Operator ´^^´ results in the Sequence of messages sent (each element in the Sequence is an instance of OclMessage type)
- Afterwards access to the parameters of the sent Operation/ Signal with the formal parameter names of their definition is possible

```
post:
let messages : Sequence(OclMessage) =
  observer^^update(? : Integer, ? : Integer)
in
messages->notEmpty() and
messages->exists( m | m.i > 0 and m.j >= m.i )
```

Messages 3

- Access to an operation return value (if the sent message is an operation call) is possible whith **`message.result()`**
- **`message.hasReturned()`** : results true, if the operation has already returned (asynchronous operation call)

```
context Person::giveSalary(amount : Integer)
post: let message : OclMessage =
  company^getMoney(amount) in
message.hasReturned() -- getMoney was sent and
                                returned
  and
message.result() = true -- getMoney call returned
                         true
```

Tips & Tricks to write good OCL 1

- Keep away from complex navigation expressions!
  - a Membership does not have a loyaltyAccount if you cannot earn points in the program:

```
context Membership
inv noEarnings:programs.partners.deliveredServices->
forAll(pointsEarned = 0) implies account >isEmpty()
```

⬇

```
context LoyaltyProgram
def: isSaving : Boolean = partners.deliveredServices
->forAll(pointsEarned = 0)

context Membership
inv noEarnings: programs.isSaving implies account->isEmpty()
```

Tips & Tricks to write good OCL 2

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

  - two persons who are married to each other 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 good OCL 3

- 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 good OCL 4

- Split and 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 LoyaltyProgram
inv: partners.deliveredServices
->forAll(pointsEarned = 0) and Membership.card
->forAll(goodThru = Date.fromYMD(2000,1,1)) and
participants->forAll(age() > 55)
```

⬇

```
context LoyaltyProgram
inv: partners.deliveredServices->forAll(pointsEarned = 0)
inv: Membership.card->forAll(goodThru = Date::fromYMD(2000,1,1))
inv: participants->forAll(age() > 55)
```

Tips & Tricks to write good OCL 5

- Use the collect shorthand on collections!

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

⬇

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

Tools

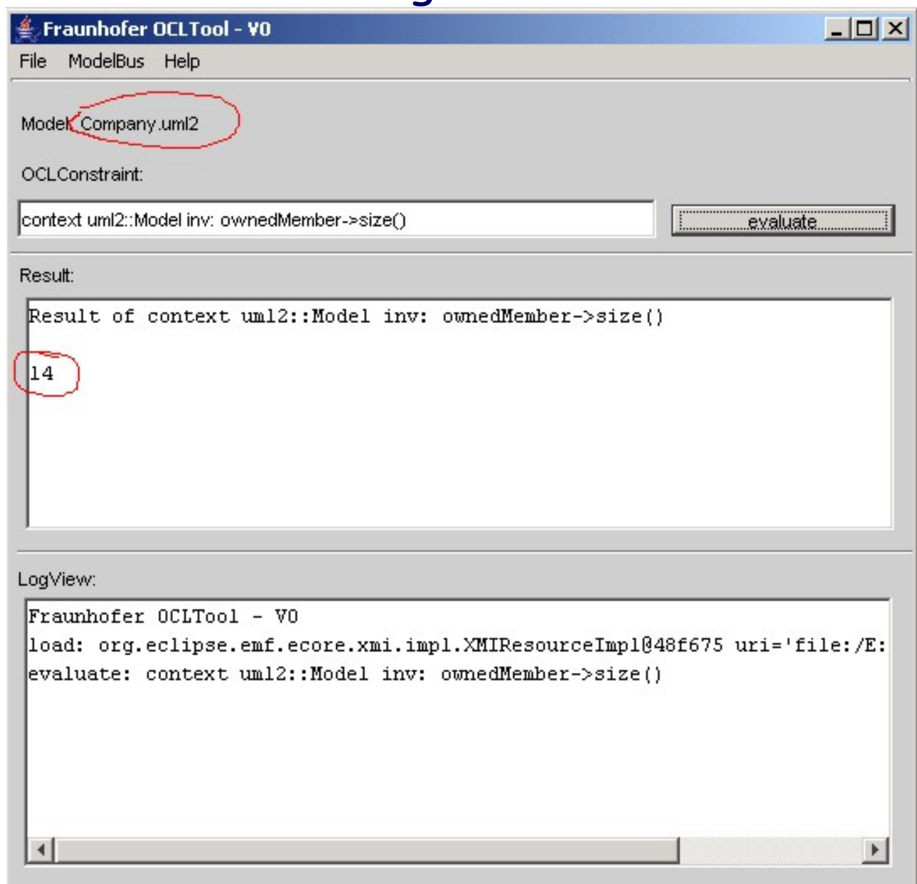
- Some OCL Parser are available, which can check syntax and evaluate OCL expressions (IBM and others)
- Dresden OCL Toolkit 2.0
  - Generates java code from OCL Constraints
  - Can be integrated into Argo/UML and its code generation:
    - Constraints from the model are included into the program
- LCI OCL Evaluator OCLE 2.0.4
  - Support for dynamic semantic validation: allows execution of OCL expressions directly from UML models
  - UML model checking against Rules defined at the metamodel level

Tools 2

- Fraunhofer OCLTool
  - Based on Kent OCL Library
  - Uses EMF libraries
  - Syntactic/semantic analyze and check of OCL expressions
    - supports evaluation at runtime
  - Supports any models based on EMF
    - dynamic and some static metamodels are supported
    - in this manner also UML2 models (EMF-based UML 2.0 Metamodel implementation)
  - Ability to use it as query tool

Tools 3

- Fraunhofer OCLTool (screenshot)
  - check OCL constraint against an UML2 model



Tools 4

- Octopus OCL (Eclipse Plug-In)
  - Required Eclipse 3.0
  - Analyze and check of OCL expressions
  - Java Code generation
  - Works on UML models
    - supports XMI import (with limitations and workarounds)
  - Open source software – distributed under a public (BSD) license
  - www.klasse.nl/english/research/octopus-intro.html