Technische Universität Berlin

Forschungsberichte
der Fakultät IV -
Elektrotechnik und Informatik

# Visual OCL: A Visual Notation of the Object Constraint Language

Christiane Kiesner, Gabriele Taentzer, Jessica Winkelmann

Bericht–Nr. 2002/23
ISSN

Forschungsberichte
Fakultät IV - Elektrotechnik und Informatik
Franklinstraße 28/29 • D-10587 Berlin

# Contents

# Chapter 1

# Overview

The Object Constraint Language (OCL) is a formal language which was developed as extension of the Unified Modeling Language (UML) [3]. OCL is used to specify invariants, pre- and post conditions of operations, methods or state changes. In contrast to UML, OCL is a pure expression language; therefore, the notation of an OCL expression has a completely different notation of model elements as the UML. This document introduces a visualization of OCL based on the OCL meta model which will be part of the UML version 2.0. So the notation of Visual OCL(VOCL) is meant as an alternative solution to the textual OCL.

Chapter 2 describes the visualisation of OCL considering a variety of examples. Chapter 3 (page 45) includes the visualized OCL standard library. For each operation in OCL this chapter describes the visualized notation. The appendix represents instances of the meta model describing some VOCL constraints used in Chapter 2.

This work is done in the context of a student's project on visual languages which took place at Technische Universität Berlin under the guidance of Gabriele Taentzer in summer 2002. This document is based conceptually on the language description of Bottoni, Koch, Parisi-Presicce und Taentzer in [1].

## 1.1 Brief Description of OCL

OCL is an abbreviation for Object Constraint Language and is an extension of the language UML. OCL is a formal language which can be used to specify invariants of classes and types in an UML class diagram or invariants of stereotypes, to describe pre- and post conditions of operations and methods, to describe guards or to denote any expression in an UML model. UML has no language elements to formulate such conditions other than in OCL and therefore in UML you can only add textual annotations in natural language to a diagram to describe additional constraints. These annotations are often ambiguous and cannot be interpreted by a machine. OCL removes this deficiency by a textual syntax which formulates the constraints adequately. In OCL you can write unambiguous constraints, since it is an object oriented and typed language (i.e. every expression has a type with well-defined semantics). Thereby it is easier to verify the expressed conditions by a parser or a constraint checker. OCL is based on the OCL meta model which describes the abstract syntax. The OCL meta model is defined in class diagrams and contains well-formedness rules which divide the set of all possible expressions in valid notations and invalid ones. The union of UML and OCL leads to the difficulty that the user has to learn two different languages to represent common model elements such as objects, links, etc.

## 1.2 Brief Description of VOCL

VOCL is a graphical representation of OCL and is trying to dispose the handicap described above. Based on the OCL meta model, VOCL follows the UML notation and its graphical representation as far as possible. This makes a direct integration of OCL in UML diagrams easier. Like OCL, VOCL is a formal, typed and object oriented language. The user doesn't need to learn another textual language, an advantage over the textual OCL. New data types and operations such as collections and operations like forall, select, union,etc. are represented by simple but meaningful graphics. Logical expressions are dentoted as Peircian graphs using different kinds of box to express disjunctions and conjunctions.

## 1.3 Basic Requirements on the Concrete and Abstract Syntax

The abstract syntax of OCL is oriented at the OCL meta model. Since VOCL describes the same language, VOCL is based on the same meta model. Thus, a transformation from textual OCL to VOCL and back is possible using the OCL meta model.

The visualization of OCL should follow as much as possible the visualization of UML. Where new visualizations are necessary we are following the recommendations of the UML standard and avoid e.g. colors and special types, to express semantic meaning. Furthermore, we offer the options to restrict the size of a diagram in the way that sub-conditions can be formulated in own diagrams.

# Chapter 2

# VOCL Language Description

This chapter introduces VOCL, a visual language to describe OCL models, conceptually and by examples. Chapter 3 (page 45) contains the standard library in which all operations of OCL are visualized. Therefore this chapter describes not all of the operations OCL offers.

## 2.1 Introduction

### 2.1.1 Legend

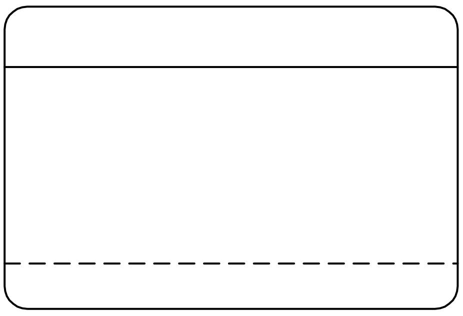


Figure 2.1: Principle representation of a constraint

An OCL constraint is visualized as a rounded rectangle with two sections, the section of the context and the section of the body which can contain a condition.
The context section(above) contains the keyword *context* followed by the typename of the model element (mostly a class or method) of the constraint followed by the kind of the constraint e.g. *inv*, *pre*, *post* or *def*.
In the body section the body of the constraint is visualized.
In the condition section are the conditions of the constraint declared, using variables defined in the body. If there is a condition section it is separated from the rest of the body by a dashed line.

### 2.1.2 Class Diagram

In the following examples of this chapter, the class diagram below is used.

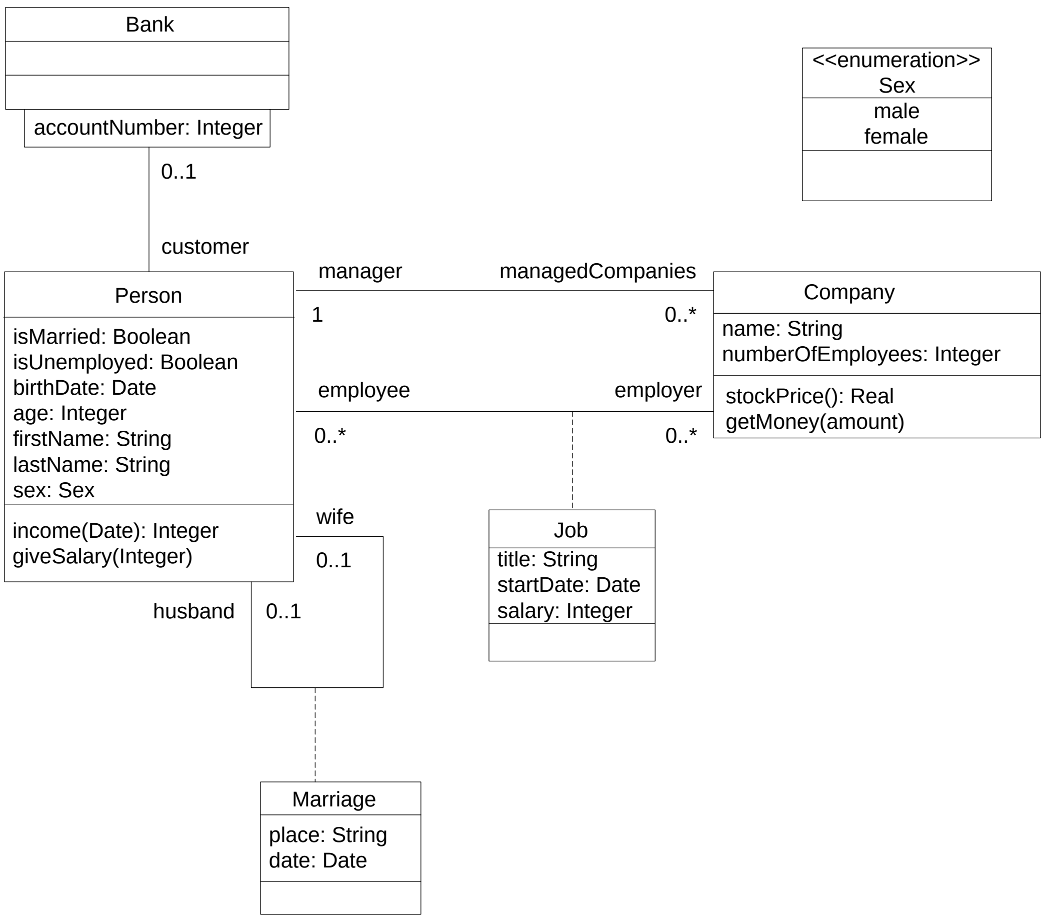


Figure 2.2: Class diagram

## 2.2 Relation to the UML Meta Model

### 2.2.1 Self

The variable *self* is used like in OCL and is always an instance of the type of the context. At this instance the constraint starts.
If it is clear where the constraint starts, *self* could be left out. This is the case exactly if there is only one instance of the type of the context.

### 2.2.2 Specifying the UML Context

The context can be specified like in OCL.

- `context Company inv: self.numberOfEmployees > 50`

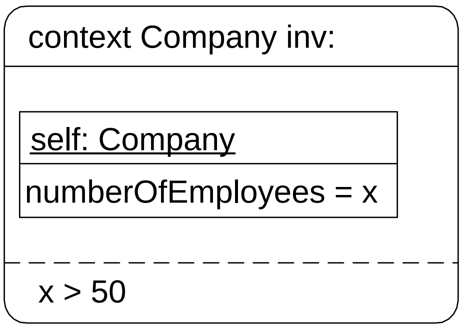


Figure 2.3: Specification of the context

This constraint specifies that the number of employees must always exceed 50.

In the context, a different name can be defined as an alternative to self.

- `context c:Company inv: c.numberOfEmployees > 50`

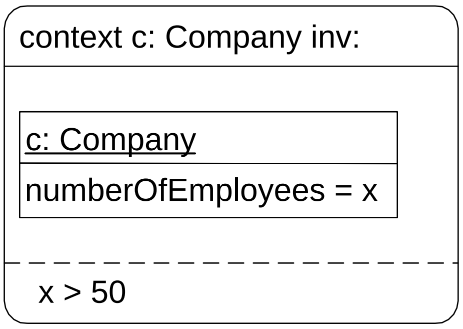


Figure 2.4: Constraint with a context name

In the body, this name is used instead of *self.* In this case, the constraint starts at this object.

A constraint can be given a name, this name occurs behind the declaration of the constraint kind.

- `context c:Company inv enoughEmployees : c.numberOfEmployees > 50`

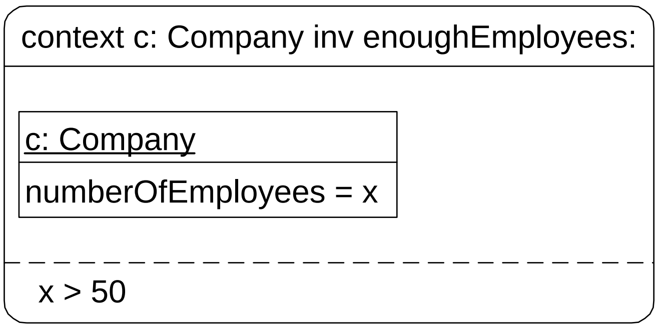


Figure 2.5: Constraint with a name

**Abstract Syntax**

The abstract syntax of a constraint is an *OclConstraint* whose attributes *name* and *kind* match the name and kind of the constraint. (In Figure 2.5, *name*="enoughEmployees" and *kind*="inv".) Each constraint has a *body* which is an *OclExpression* and a *ConstraintElement* which is a *Model Element* with the name of the constraint context. (In Figure 2.5 the *constrainedElement* is a *Class* with the name "Person" and the *body* is an *OperationCallExp* which refers to a *type* and a *referredOperation* which has an argument. The *type* is the return type of the operation (in the example "Boolean"), the name of the *referredOperation* is the name of the operation (in the example ">") and the arguments are *OclExpressions*). A detailed description of the abstract syntax of the constraint in Figure 2.5 is shown in the meta model instance on page 75.

### 2.2.3 Invariants

In the previous examples, the constraint kind always was *inv*, hence these constraints describe invariants. Other kinds are *pre* and *post* which specify pre- and post conditions of operations as well as *def* whereby definitions of variables or operations are introduced.

### 2.2.4 Operations id and isIn

Operation *id* describes the identity of two objects. It is a helper operation which exists only in VOCL and not in OCL. It simplifies the visualization of identical instances of objects. The representation is a link labeled by *id*. See Figure 2.8 on page 10 for an example.
Two instances are also identical if they have the same name. See Figure 2.10 on page 12.
The operation *isIn* is applied to a collection and returns true if the collection contains the object. This operation doesn't exist in OCL, too. See Figure 2.32 on page 26. The collection *Person* contains *p1* and *p2*. This is represented by an isIn link.

### 2.2.5 Pre- and Post Conditions of Operations or Methods

If a pre- or post condition of a method or operation is visualized, *self* is an instance of the type that provides this method or operation. An operation call is visualized as in collaboration diagrams.

If the operation has a return type which is not a primitive data type (e.g. a collection), this is visualized by an arrow from the instance on which the operation is called to the instance which is returned.
The following constraint specifies the post condition of operation *income* of a person, the return type of this operation is of type Integer. Its value is 5000.

- `context Person::income(d : Date) : Integer post: result = 5000`

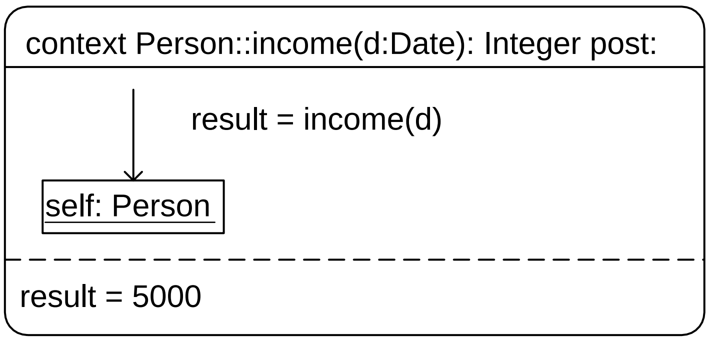


Figure 2.6: Post condition of an operation

*result* is a predefined value, it is the return value of an operation if there is one. Therefore, the assignment to the variable *result* can also be left out.

**Abstract Syntax**

At post conditions the attribute *kind* of an *OclConstraint* in the abstract syntax is ”post”. The operation call corresponds to an *OperationCallExp*. A detailed description of the abstract syntax of the constraint in Figure 2.6 is shown in the meta model instance on page 76.

### 2.2.6 Basic Values, Basic Types and Enumeration Types

As in OCL the basic types are Boolean, Real, Integer and String. The predefined operations on these types are described in the standard library.
Enumeration types like *male* or *female* of data type *Sex* can be used as follows:

- `context Person inv: sex = Sex::male`

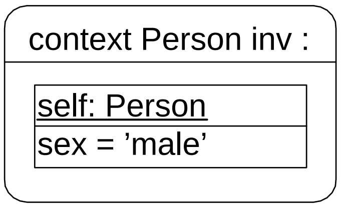


Figure 2.7: Enumeration types

The constraint above states that the sex of a person is male.

### Abstract Syntax

The basic types Boolean, Real, Integer and String are abstractly mapped on *BooleanLiteralExp*, *RealLiteralExp*, *IntegerLiteralExp* and *StringLiteralExp*. The elements of an enumeration type are *EnumLiteralExps* abstractly which reference *EnumLiteral* refering to an *Enumeration*. At Figure 2.7, the name of the *EnumLiteral* is ”male” and the name of the *Enumeration* is ”Sex”. A detailed description of the abstract syntax of the constraint in Figure 2.7 is shown in the meta model instance on page 77.

## 2.2.7 Implies Expressions

An *implies* expression is visualized in an *implies* frame. Anything above the keyword *implies* describes the premise. When this premise is true, it implies the inclusion denoted below *implies*. Both sections can have their own condition section.

- `context Person inv: self.isMarried = true implies self.age >= 18`

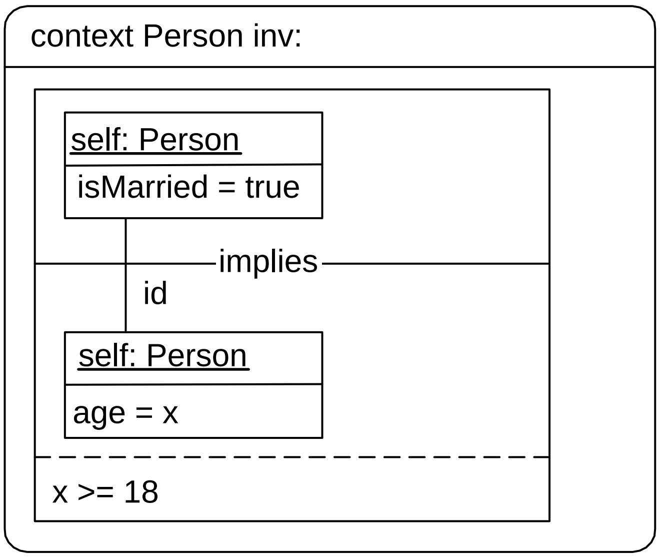


Figure 2.8: Implies constraint

This *implies* constraint specifies: If a person is married, the person is elder than 18 years.
The *id* link describes that the persons above and below the keyword *implies* are the same.

### Abstract Syntax

In the abstract syntax an *implies* is an *Operation*. The usage of an *implies* expression is an *OperationCallExp*, the referenced *Operation* has the name ”implies”, the referenced type is ”Boolean”. The arguments are *OclExpressions* which are visualized above and below *implies*. A detailed description of the abstract syntax of the constraint in Figure 2.8 is shown in the meta model instance on page 78.

### 2.2.8 If-Then-Else Expressions

The *If-Then-Else* frame contains three sections, the *if* section describes the *if* condition, the *then* section describes the *then* part and the *else* section describes the *else* part. Each of these sections can have a condition section.

-
  ```
  context Person inv:
  if(self.isUnemployed = false and self.isMarried = true)
  then income >= 3000
  else income < 3000
  ```

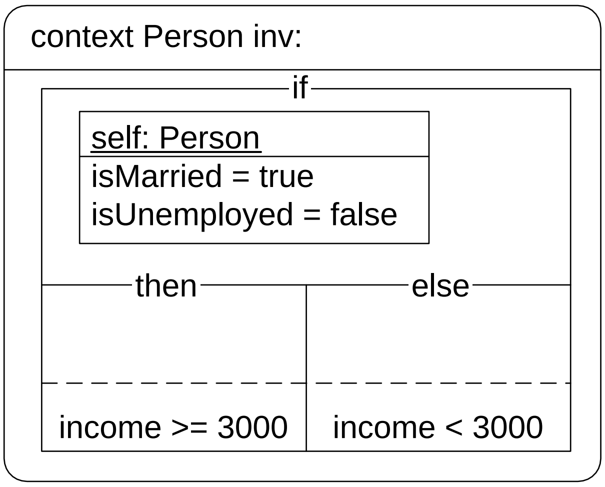


Figure 2.9: If-Then-Else constraint

This *If-Then-Else* constraint specifies: If a working person is married, its income is at least 3000 else the income is less than 3000.

#### Abstract Syntax

The abstract syntax of an *If-Then-Else* expression is an *IfExp.* It refers to a *condition*, a *thenExpression* and an *elseExpression*, if there is one. These three are *OclExpressions.* A detailed description of the abstract syntax of the constraint in Figure 2.9 is shown in the meta model instance on page 79. In this the second of the *if* conditions is left out.

### 2.2.9 Let Expressions and Definition Constraints

A *let expression* defines a variable or an operation which can be used in a constraint after its definition. There are two frames, a *let* frame and an *in* frame. The *let* frame contains the visualized definition of the variables and operations. Each variable is defined in an own frame where the name of the variable is depicted in the upper left corner and below the definition of the variable value follows; for operations analogously. If more than one *let expression* has to be visualized, then each *let expression* has its own frame. If just one *let expresssion* has to be visualized, the frame is optional.
Inside the *in* frame a normal constraint is described which uses the variables and operations defined above. A *let expression* is only known in the constraint in which it was defined.

-
  ```
  context Person inv:
  let income : Integer = self.job.salary->sum()
  let hasTitle(t : String) : Boolean = self.job->exists(title = t) in
  if isUnemployed = true  then
  income < 100
  else income >= 100 and hasTitle (´manager´)
  endif
  ```

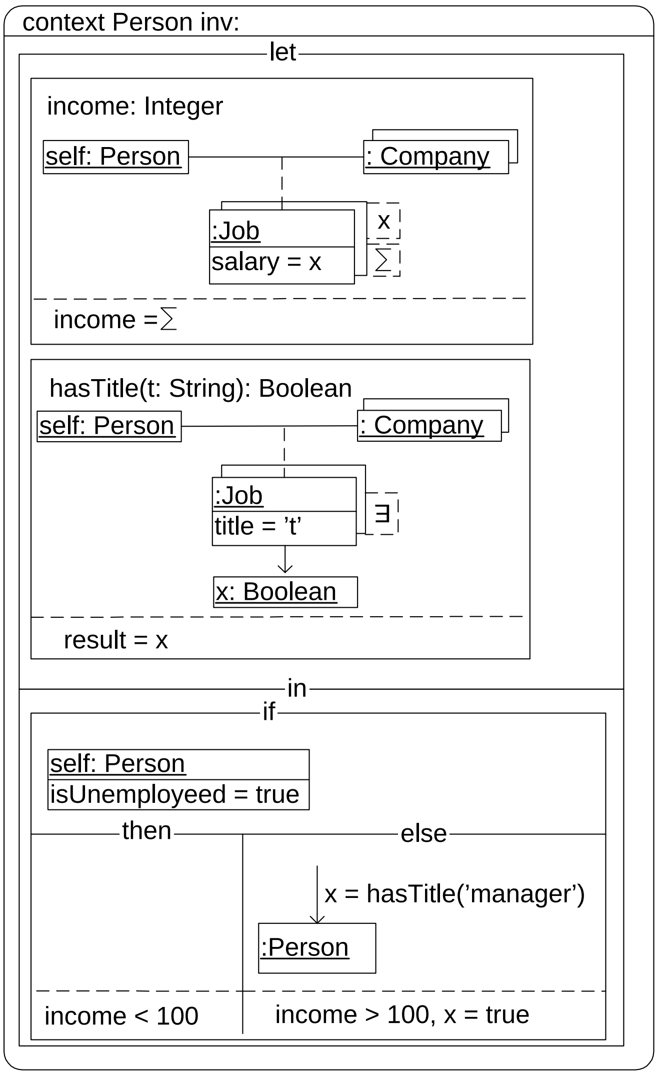


Figure 2.10: A let constraint

In the *let* constraint above a variable *income* which is the sum of incomes of all the jobs one person has, and an operation *hasTitle* which has a String as input and returns a Boolean value, are defined. *hasTitle* returns true if the person has a job with the given title.
The *in* section of the constraint specifies: If a person doesn't work he/she has an income less than 100 else the person has an income of at least 100 and is a manager.
In this constraint operations *sum* and *exists* are used which are described in the Section 2.3.11.

A *definition* constraint contains only *let expressions*. Variables and operations which are defined in a *definition* constraint, are also known and usable in other constraints.

-
  ```
  context Person def:
  let income : Integer = self.job.salary->sum()
  let hasTitle(t : String) : Boolean = self.job->exists(title = t)
  ```

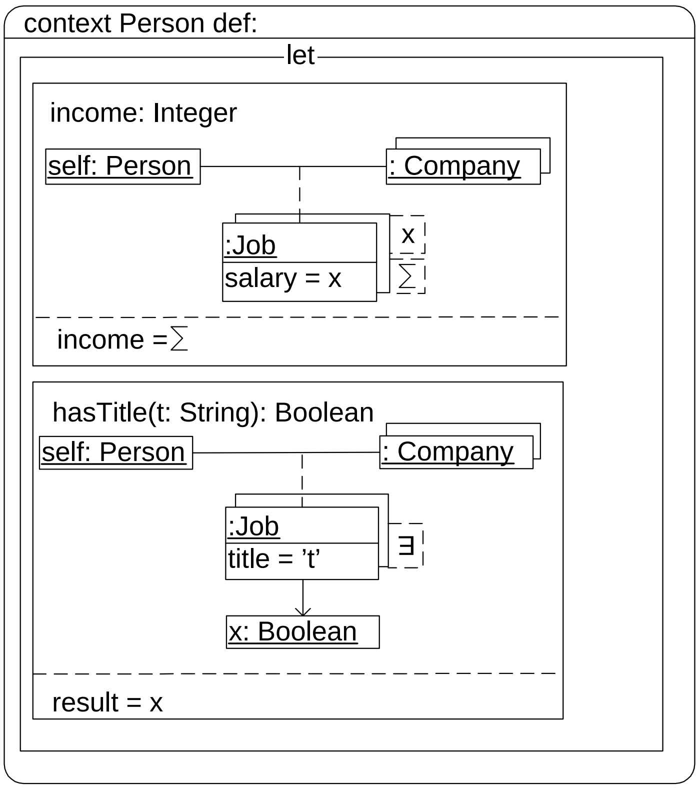


Figure 2.11: A definition constraint

**Abstract Syntax**

The abstract syntax of a *let expression* is a *LetExp* that refers to the defined variable, a *VariableDeclaration*, and to the *in* section which is an *OclExpression*. The *VariableDeclaration* has an *initExpression* which is also an *OclExpression*. The definition of more than one variable, operations or methods is not supported by the abstract syntax of OCL. A detailed description of the abstract syntax of the constraint in Figure 2.10 is shown in the meta model instance on page 80. In this meta model instance, the operation *hasTitle* is not been considered, because it is not supported in the abstract syntax.

### 2.2.10 Type Conformance, Re-typing or Casting

The basic value types of VOCL are organized in the same type hierarchy as the basic value types in OCL.
Re-typing or casting is done by using the operation *oclAsType(Type2)*.

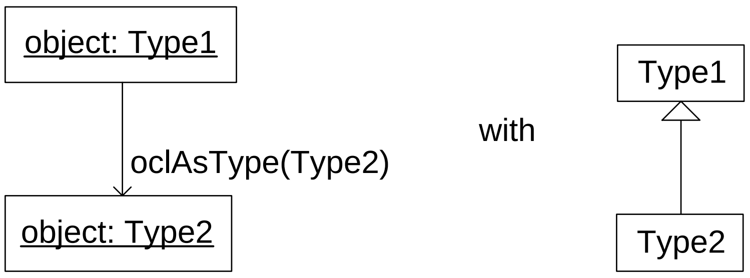


Figure 2.12: Re-typing or casting

The operation *oclAsType* re-types an object of type *Type1* to an object of type *Type2*. An object can only be re-typed to one of its subtypes: therefore, *Type2* must be a subtype of *Type1*.

#### Abstract Syntax

The abstract syntax of *OclAsType(Type2)* is an *OperationCallExp* with argument *Type2*.

## 2.3 Objects and Properties

### 2.3.1 Objects

The visualization of an object is the same as the visualization of objects in collaboration diagrams.

### 2.3.2 Properties: Attributes

The attribute value of an object is referred by a variable.

- `context Person inv : self.age`

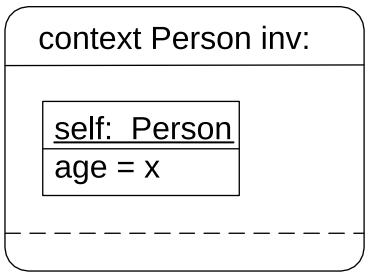


Figure 2.13: Attributes of objects

The variable $x$ refers to the age of a person. In the condition section, expressions about the value of $x$ can be stated, e.g. the age of a person is always greater than zero:

- `context Person inv : self.age > 0`

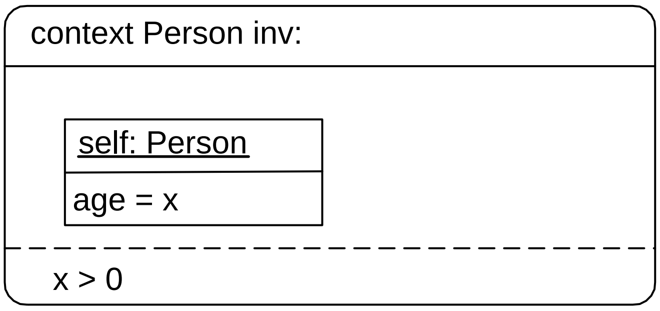


Figure 2.14: Properties of an objects attribute

### Abstract Syntax

In the abstract syntax the attribute value of objects is an *AttributeCallExp* which refers to an *Attribute* with the same name. This *Attribute* refers to its type. An *AttributeCallExp* can have a source, from which it was called. This source is an *OclExpression.* In Figure 2.14, the name of the referred attribute is "age", *source* of the *AttributeCallExp* is a *VariableExp*, which refers to the variable *self.* A detailed description of the abstract syntax of the constraint in Figure 2.14 is shown in the meta model instance on page 81.

### 2.3.3 Properties: Operations

An operation is visualized as follows:

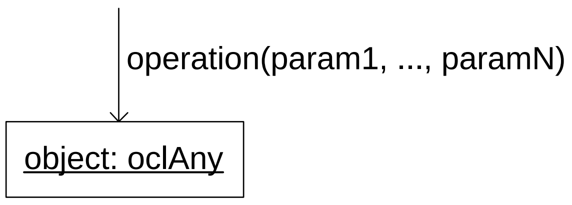


Figure 2.15: An abstract operation call

Operation *operation* with the parameters *param1,...,param N* is applied to an object *object* of any type. The object that is returned, can be referred to by the reserved variable *result*. Expressions about the value of *result* can be made in the condition section of a constraint.

The return value can also be named differently:

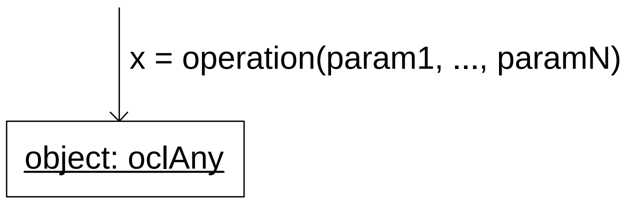


Figure 2.16: An abstract operation call, with x as return value

An example for an operation call:

- `context Person::income (d: Date) : Integer post: result = age ·1000`

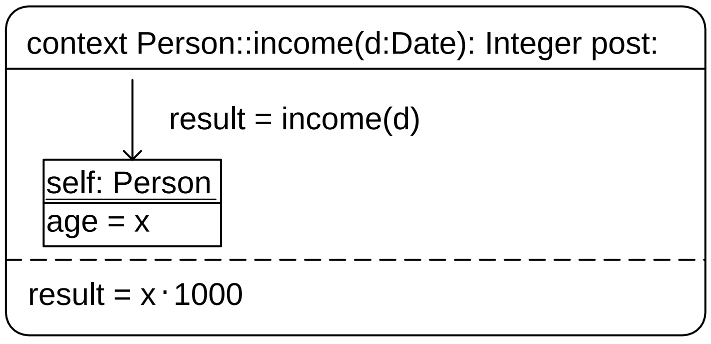


Figure 2.17: Operation call with assignment

This post condition specifies that the income of a person is equal to the age of the person times 1000. To refer to an operation without parameters, parentheses with an empty argument list are mandatory:

- `context Company inv: self.stockPrice() > 0`

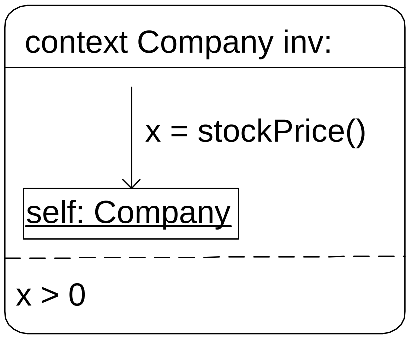


Figure 2.18: Operation without parameters

This constraint defines that the stock price of a company is always greater than zero.
If the operation has a return type, which is not a primitive type, e.g. a collection, this is visualized by an arrow starting at the instance on which the operation is called, and goes to the returned instance. An example is the operation *including(x)*, which is shown in Figure 2.38 on page 30.

**Abstract Syntax**

The abstract syntax of an operation call is an *OperationCallExp* that refers to the return type, to the arguments of the operation which are *OclExpressions*, and to the *Operation*. A detailed description of the abstract syntax of the constraint in Figure 2.17 is shown in the meta model instance on page 82.

### 2.3.4 Properties: Association Ends and Navigation

The navigation on an association is visualized by a link between the instances of the classes (as in UML). If the name of a navigation is left out, the role name of the opposite association end can be used. In the case of unambiguous navigation, the name of the navigation can be left out. This is exactly then the case, if there exists only one navigation between the classes. The expression result is the set of objects on the opposite end of the association and has the multiplicity defined in the class diagram. The navigation on any numbered associations always starts at object *self* if it exists, or otherwise at the object defined in the context section. Else it starts at the only object of the context type.

-
  ```
  context Company
  inv: self.manager.isUnemployed = false
  inv: self.employee->notEmpty()
  ```

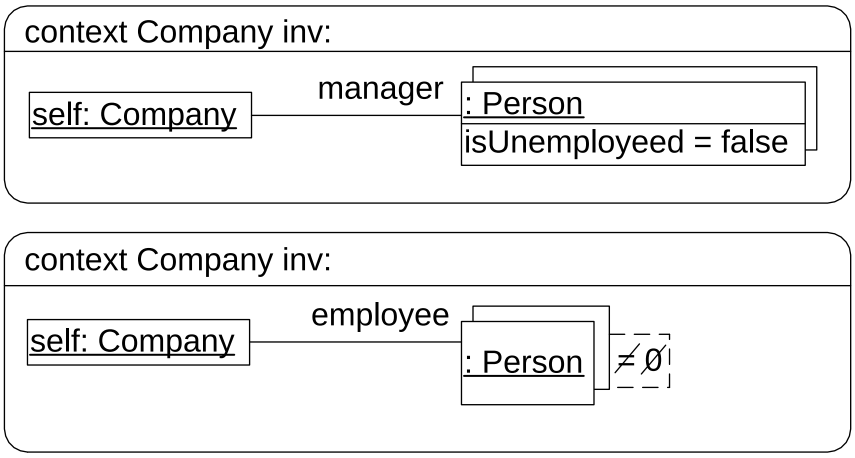


Figure 2.19: Association ends and navigation

(If several inv-, post-, or pre expressions have to be visualized within one context, they are visualized as single constraints.)
The upper constraint in Figure 2.19 specifies: The manager of a company is not unemployed. The lower one says: A company has at least one employee.
In this constraint, the operation *notEmpty()* which is applied to an collection, is visualized. The operation is denotated by $\neq \emptyset$ at the collection which should be known as an operation on sets.
Collections, like sets, bags, and sequences, are predefined types in OCL. They have a large number of predefined operations, e.g. operation *size()*, which is visualized as follows:

- `context Person inv: self.employer->size() < 3`

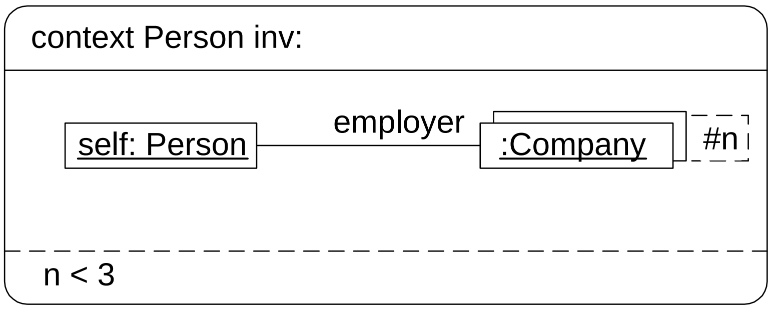


Figure 2.20: Association ends and navigation

The constraint specifies: A person has less than 3 employers.
Operation *size()* is applied to a collection (in this example a set) of employers. The variable $n$ contains the number of elements in that collection.

Other operations are *isEmpty()* which tests if a collection is empty, and operation *notEmpty()* which returns true, if a collection is not empty.

- `context Person inv: self.employer->isEmpty()`

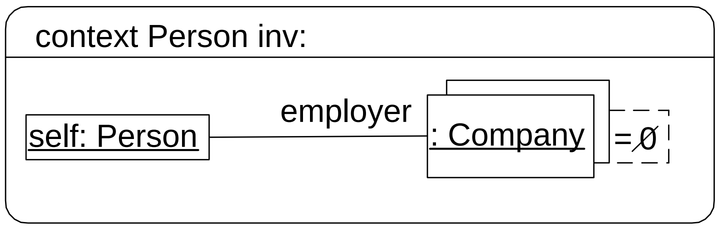


Figure 2.21: Operation isEmpty

This constraint specifies that a person has no employers.

**Abstract Syntax**

The abstract syntax of a navigation along an association is an *AssociationEndCallExp* which refers to both association ends. A detailed description of the abstract syntax of the constraint in Figure 2.20 is shown in the meta model instance on page 83.

**Navigation over Associations with Multiplicity Zero or One**

- `context Person inv: self.wife->notEmpty() implies self.wife.sex = Sex::female`

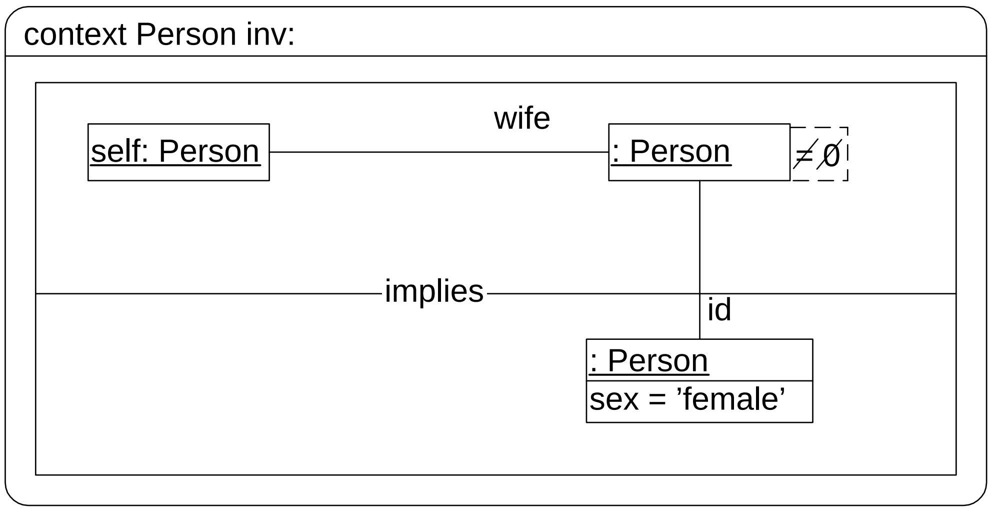


Figure 2.22: Navigation along associations with multiplicity zero or one

This constraint defines: A person has a wife implies that the wife is female.
Since the multiplicity of association end *wife* is zero or one, a navigation along this end results in only one element which is visualized in that way.

### Combining Properties

Subexpressions which are visualized side by side or below each other are automatically combined by *and*.

-
  ```
  context Person inv:
  self.wife->notEmpty() implies self.wife.age >= 18
  and self.husband->notEmpty() implies self.husband.age >= 18
  ```

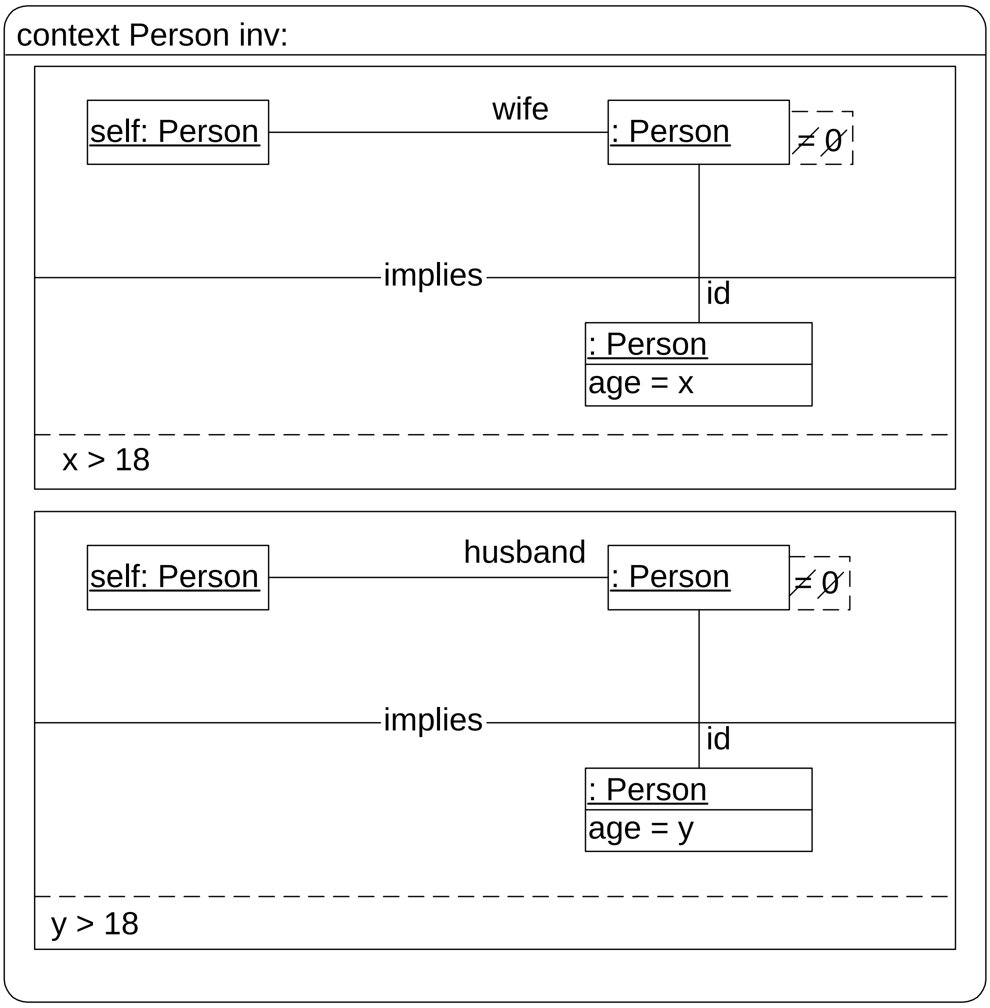


Figure 2.23: Combining properties by "and"

The constraint expresses: Married persons are at least 18 years old.

Combining properties by *or* is visualized by an *or* frame. The expressions left and right (or above and below) of *or* are combined by *or*.

- `context Person inv: self.isMarried = true implies [(self.wife.age > = 18) or (self.husband.age >= 18)]`

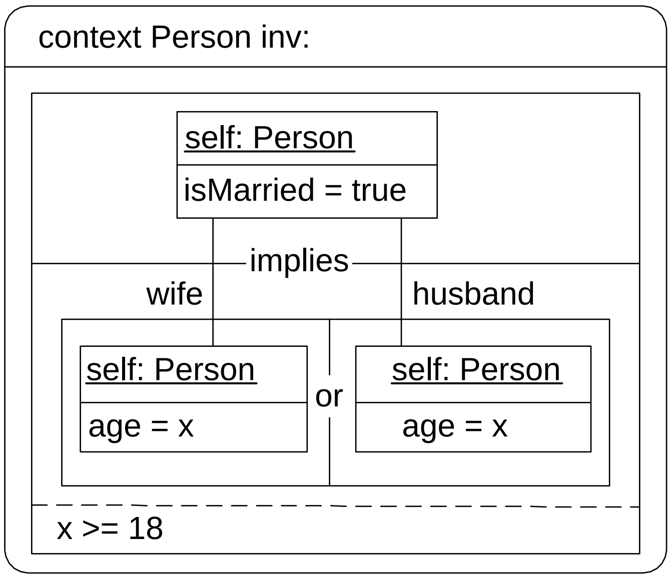


Figure 2.24: Combining properties by ”or”

The constraint above specifies: If a person is married, then the spouse is at least 18.

### Abstract Syntax

Considering the abstract syntax, combining properties by *and* or *or* is done by an *OperationCallExp*. The referred operation has the name ”and” or ”or” and the *OperationCallExp* is of ”Boolean” type. A detailed description of the abstract syntax of the constraint in Figure 2.24 is shown in the meta model instance on page 84.

## 2.3.5 Navigation to Association Classes

The navigation to association classes is visualized like the navigation to other classes (if the role name is missing the class name can be used as well).

- `context Person inv: self.birthDate < self.marriage.date`

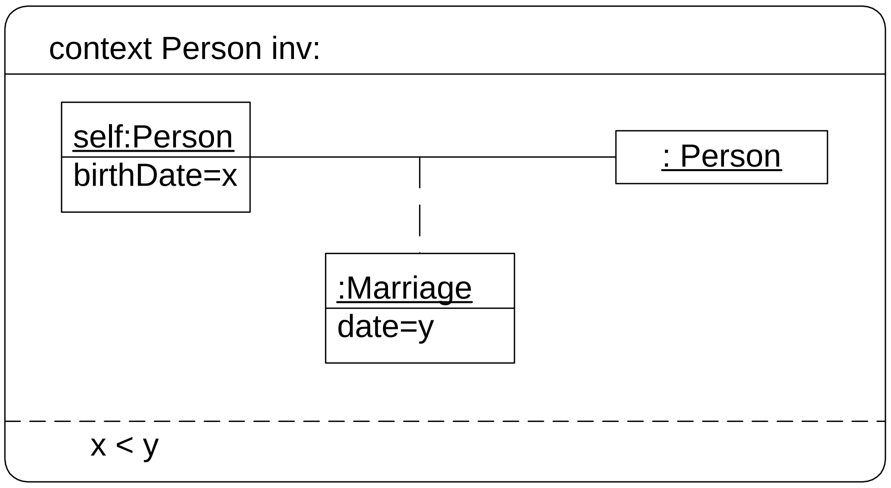


Figure 2.25: Navigation to association classes

The constraint specifies that a person's marriage is after the person's birthdate.

**Abstract Syntax**

The navigation to association classes is abstracted to an *AssociationClassCallExp* that refers to an *AssociationClass* with the name of the association class. A detailed description of the abstract syntax of the constraint in Figure 2.25 is shown in the meta model instance on page 85.

### 2.3.6 Navigation from Association Classes

The navigation from association classes is visualized analogously to the navigation from normal classes.

- `context Job inv:  self.employer.numberOfEmployees >= 1`

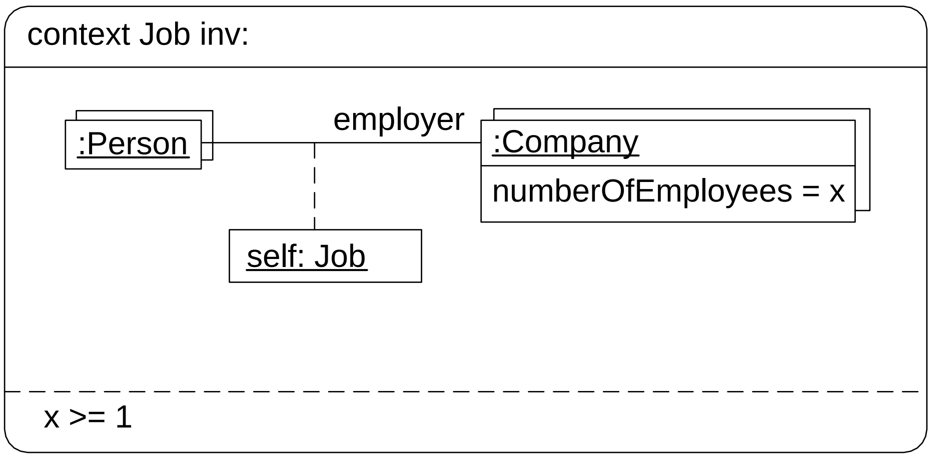


Figure 2.26: Navigation from association classes

The constraint defines that a job is done by at least one employee of the company.

### Abstract Syntax

The abstract syntax of a navigation from an association class does not differ from the abstract syntax of a ”normal” navigation. A detailed description of the abstract syntax of the constraint in Figure 2.26 is shown in the meta model instance on page 86.

## 2.3.7 Navigation through Qualified Associations

- `context Bank inv: self.customer[8764423]`

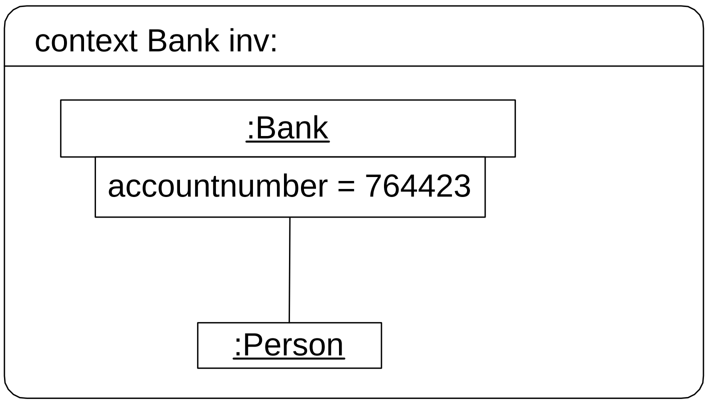


Figure 2.27: Navigation through qualified associations

The navigation through qualifier *accountnumber* results in one person having *accountnumber* 8764423. The qualifier and its value are denoted as in UML object diagrams.

### Abstract Syntax

The abstract syntax of a navigation through qualified associations is an *AssociationEndCallExp* which refers to the association end, an *AssociationEnd* and to the qualifier. The qualifier is an *OclExpression* (in Figure 2.27 an *IntegerLiteralExp*). A detailed description of the abstract syntax of the constraint in Figure 2.27 is shown in the meta model instance on page 87.

## 2.3.8 Accessing overridden Properties of Supertypes

The next example applies to the following class diagram.

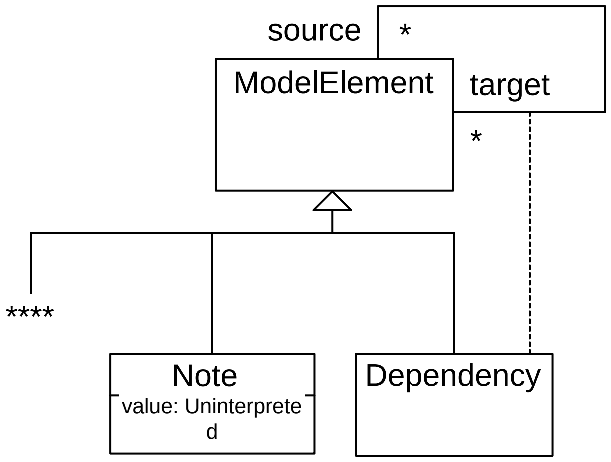


Figure 2.28: Class diagram

A property of a supertype can be accessed using operation *oclAsType(Type2)* which is visualized as follows:

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

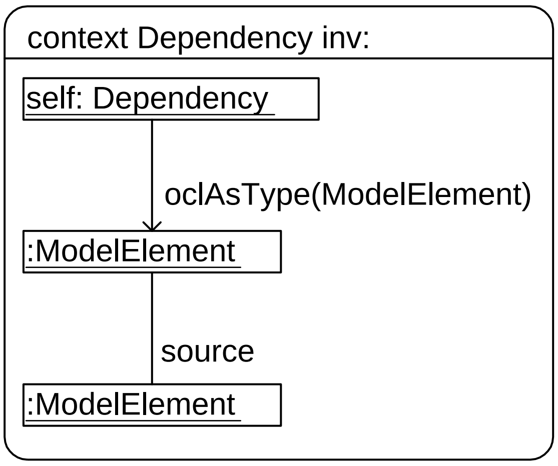


Figure 2.29: Properties of supertypes

After re-typing an object of type *Dependency* to an object of supertype *ModelElement* the navigation along association *source* can be used.

**Abstract Syntax**

The abstract syntax of *oclAsType(Type2)* is an *OperationCallExp* with argument *Type2*.

### 2.3.9 Predefined Properties on All Objects

Predefined properties on all objects are visualized as follows:

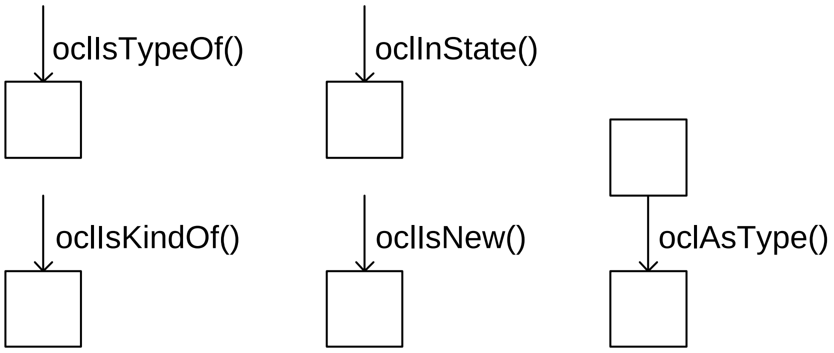


Figure 2.30: Predefined properties on all objects

If we want to specify that every instance of class Person is of type *Person* and not of type *Company* we write:

-
  ```
  context Person
  inv: self.oclIsTypeOf(Person) = true
  inv: self.oclIsTypeOf(Company) = false
  ```

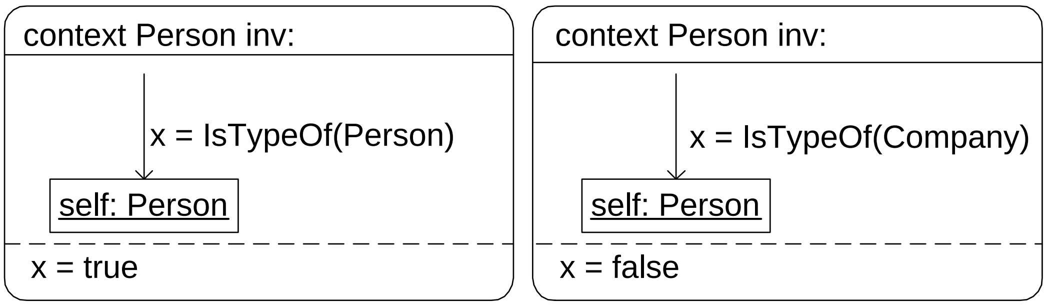


Figure 2.31: Predefined properties on all objects

**Abstract Syntax**

In the abstract syntax, a predefined property is an *OperationCallExp*. A detailed description of the abstract syntax of the left sub-constraint in Figure 2.31 is shown in the meta model instance on page 88.

## 2.3.10 Features on Classes Themselves

All properties discussed until now, are properties on class instances.
It is possible to use features defined on the types/classes themselves. One example is feature *allInstances*.

-
  ```
  context Person inv:
  Person.allInstances()->forall(p1, p2 | p1 <> p2 implies p1.name <> p2.name)
  ```

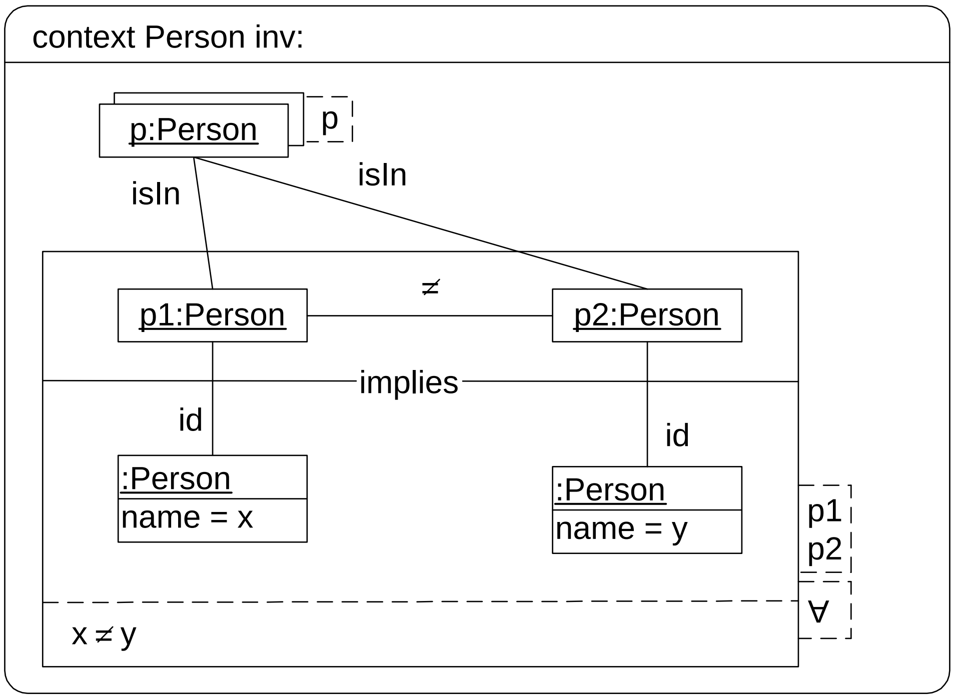


Figure 2.32: Features on classes themselves

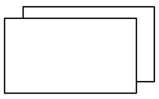
Feature *allInstances()* results in a set of all instances of the type which exists at the specific time when the expression is evaluated. It is visualized by a set of type of the class name with an iterator variable depicted on the right of the set inside a dashed box.
In the constraint above the navigation starts at this set, since there is no instance *self* from which could be started. After this a *forall* operation is applied to that set and inside the *forall* an *implies* operation. *forall* is defined on collections and has one or two iterators. It has a frame which contains the expression which has to be true for each collection element. By *isIn* all instances of the set of all persons can be accessed. These instances corresponds to the iterator/iterators.
The constraint specifies that all instances of type *Person* have unique names.
The link labeled by *isIn* visualizes that single objects are contained in the collection.

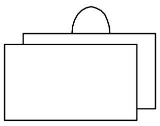
**Abstract Syntax**

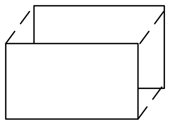
The usage of class features is abstracted to an *OperationCallExp*. In the meta model instance on page 89 the abstract syntax of the constraint in Figure 2.32 is shown.

### 2.3.11 Collections

The collection type defines a large number of predefined operations, a complete reference of this operations is in the VOCL standard library (in Chapter 3 starting on page 45).
The collection type is an abstract type with three concrete collection types as its subtypes: *Set*, *Bag* and *Sequence*. These types are visualized as follows:

Set:


Bag:


Sequence:


A simple navigation results in a *Set*,

- `context Company inv: self.employee`

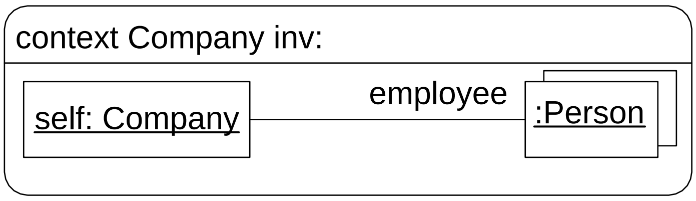


Figure 2.33: Simple navigation

a combined navigation in a *Bag*,

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

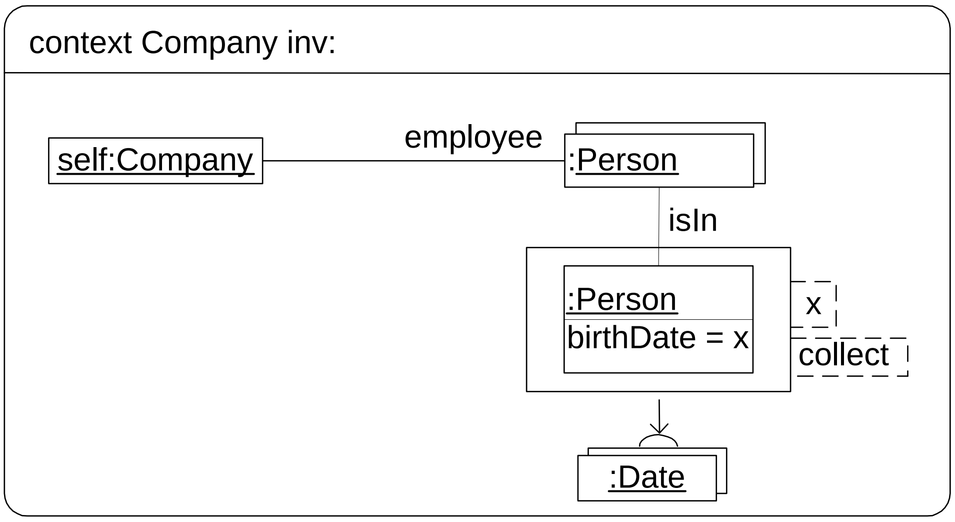


Figure 2.34: Combined navigation

and a navigation along an association adorned with *sortedBy* results in a *Sequence*.

- `context Company inv: self.employee->sortedBy(age)->asSequence()`

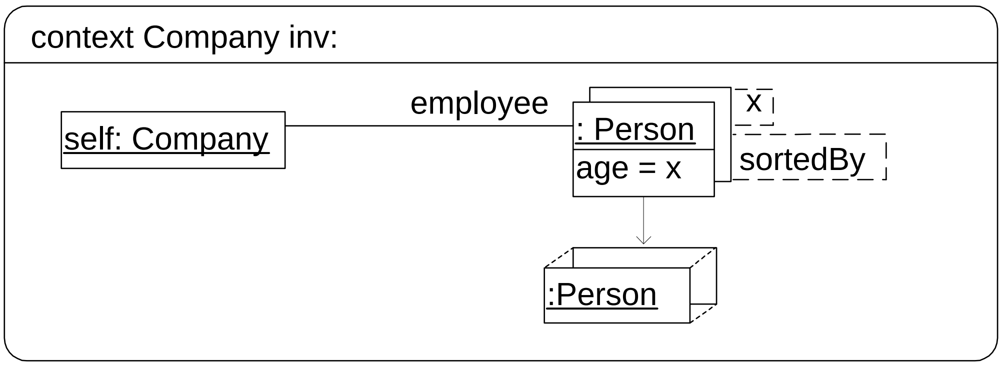


Figure 2.35: Navigation along an association adorned with *sortedBy*

Operations on collections may result in new collections, as e.g. the union of two collections:

- `collection1->union(collection2)`

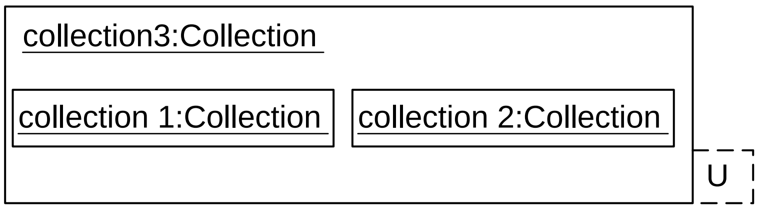


Figure 2.36: Operations on collections

The resulting collection can get a name which is denoted at the upper left corner of the union frame, the mathematical union symbol is put to the right of the union frame.

### Abstract Syntax

The abstract syntax of a navigation to a collection does not differ from the navigation to other classes, so it is a *NavigationCallExp* with two navigation-ends (*navigationSource* and *referredAssociationEnd* or *referredAssociationClass*).

## 2.3.12 Previous Values in Post Conditions

To refer to a value of an attribute at the start of the operation, the attribute name with the postfix *@pre* is used.
The following visualization specifies, that the age of a person is incremented by 1 on its birthday.

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

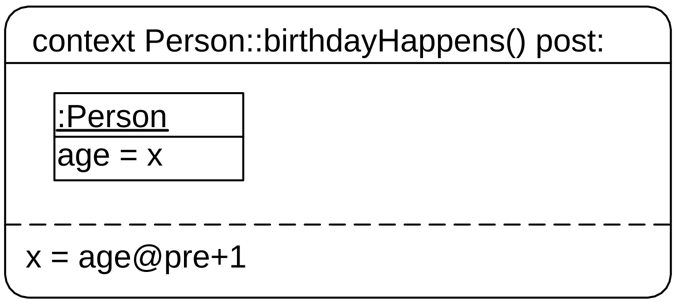


Figure 2.37: Previous values in post conditions

To refer to a value of a navigation at the start of an operation, the role name with the postfix *@pre* is used.

-
  ```
  context Company::hireEmployee(p: Person)
  post: employees = employee@pre->including(p)
  and stockprice() = stockprice@pre() + 10
  ```

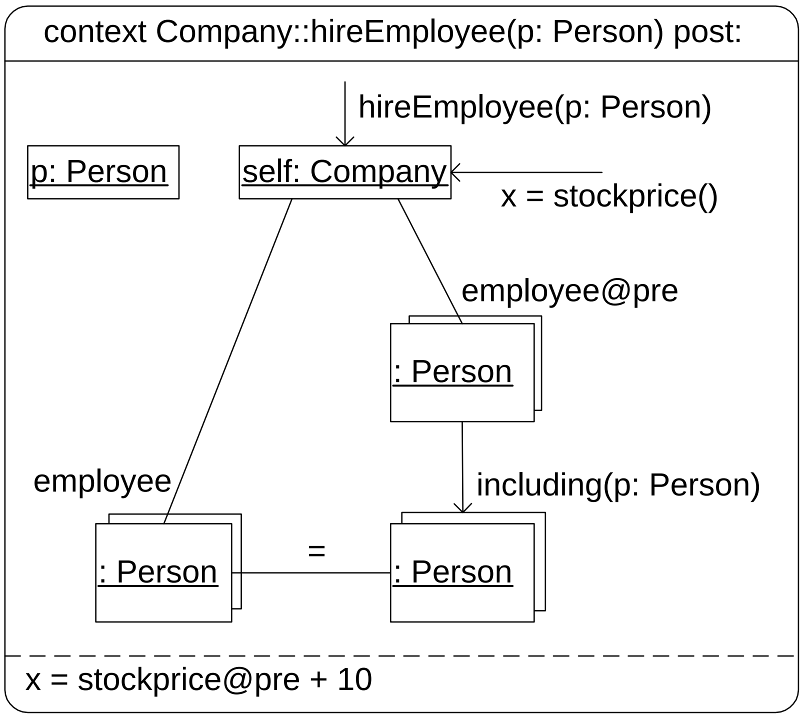


Figure 2.38: Navigation to previous objects in a post condition

In this example the predefined operation *including* is demonstrated. *including* unifies the set of employees of a company before the employment of a new employee with the new employee. The operation *including* is applied to a collection and gets an object name. It returns a new collection.
The constraint specifies the following: After the employment of a new employee the set of employees contains the new employee and the stock price of the company is incremented by 10.

**Abstract Syntax**

The value of an attribute at the start of an operation is accessed by an *AttributeCallExp*, the navigation to objects which exist at the start of the operation is a *NavigationCallExp*. A detailed description of the abstract syntax of the constraint in Figure 2.37 is shown in the meta model instance on page 90.

## 2.4 Collection Operations

In this chapter only some of the collection operations are described. All operations are described in the standard library (Chapter 3 on page 45).

### The Select Operation

The operation *select* specifies a subset of a collection. All elements in this subset have the properties visualized in the *select* frame. The evaluation of the *select* frame results in a Boolean value. On the right of the *select* frame the iterator and the keyword *select* are depicted.
If several operations are executed, the names of the operations and the iterators are denoted sequentially (top down) on the right of the frame.

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

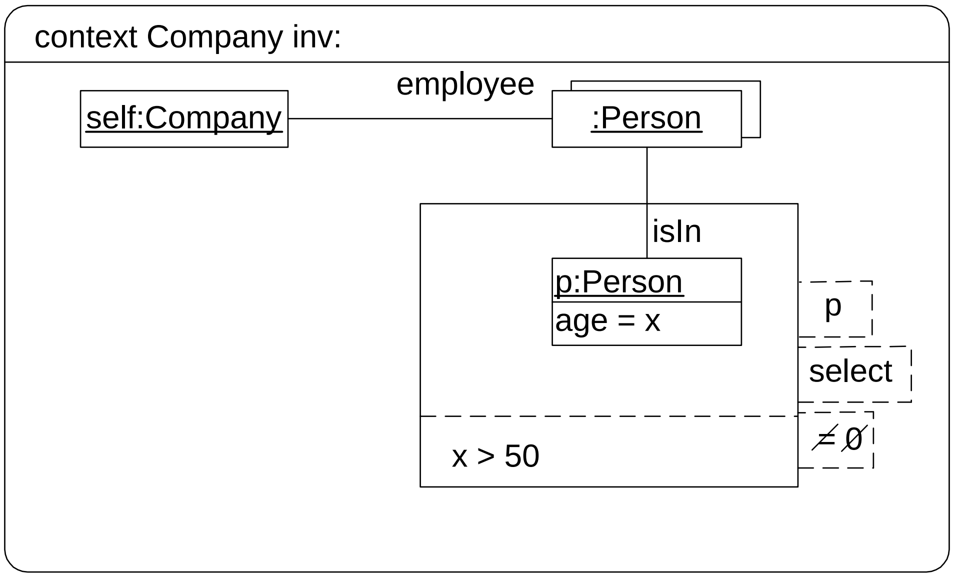


Figure 2.39: A select operation

The constraint specifies that at least one employee of a company is elder than 50.
The iterator is of type *Person*, the select property states that the age of a person has to be greater than 50.

### The Reject Operation

The operation *reject* specifies a subset of a collection. All elements in this subset have not the properties visualized in the *reject* frame. The evaluation of the expression in the *reject* frame results in a Boolean value. On the right of the *reject* frame the iterator and the keyword *reject* are depicted.

- `context Company inv:`
  `self.employee->reject(iMarried)->isEmpty()`

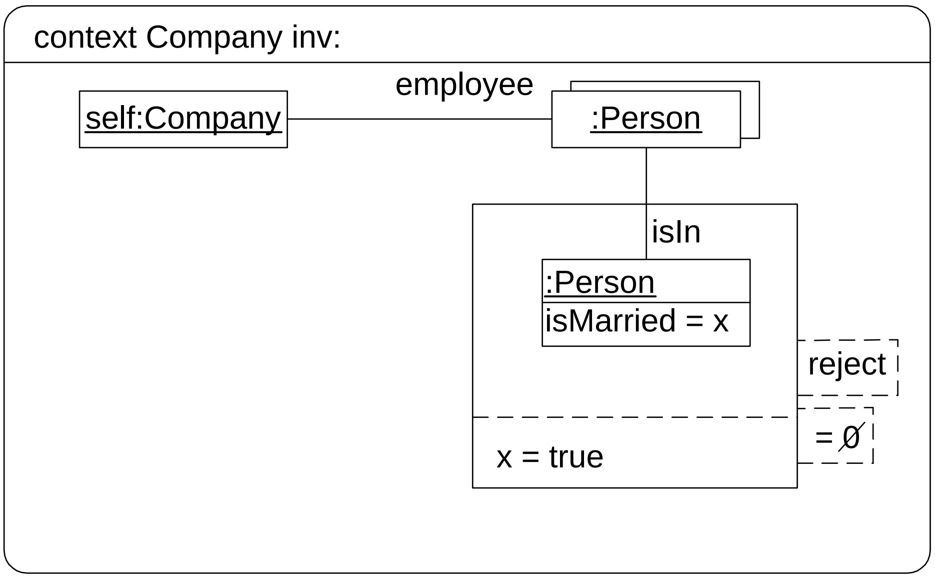


Figure 2.40: A reject operation

This *reject* operation returns all persons that are not married. The constraint specifies that the collection of employees not married, is empty.
The iterator is of type *Person*, the attribute *isMarried* is checked. If the iterator is obvious, it can be omitted.

## The Collect Operation

The *select* and *reject* operations always result in a sub-collection of the original one.
The *collect* operation results in a collection which is derived from some other collection, but which contains different objects from the original collection (i.e., it is not a sub-collection). In the *collect* frame of the *collect* operation, a new variable is defined, which has the type of the new collection. This variable is denoted on the right of the *collect* frame above the keyword *collect*.
The set of birthdates of all employees of a company is specified as follows:

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

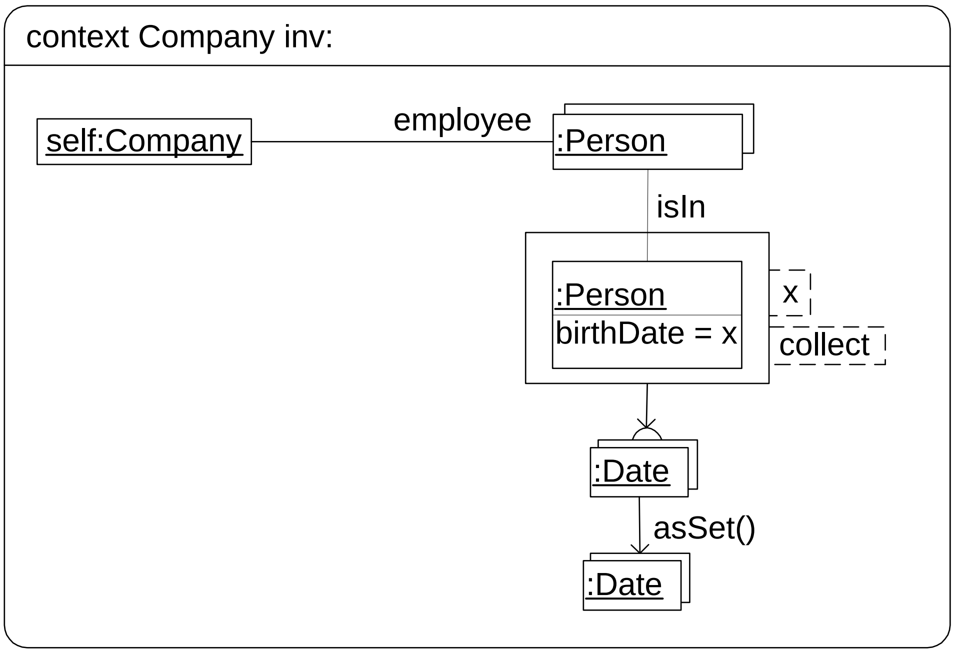


Figure 2.41: A collect operation

The operation *asSet()* applied to a bag results in a set. All multiple objects in the bag are removed.

### The Forall Operation

The operation *forall* checks a constraint for all elements of a collection. It has one or two iterators and a *forall* frame in which the property of the *forall* operation is visualized. The *forall* operation returns a Boolean value. All elements of the collection must fullfil this property. On the right of the frame the ∀-operator and the iterators are depicted.

-
  ```
  context Company inv:
  self.employee->forall (e1, e2: Person | e1 <> e2 implies
  e1.firstname <> e2.firstname)
  ```

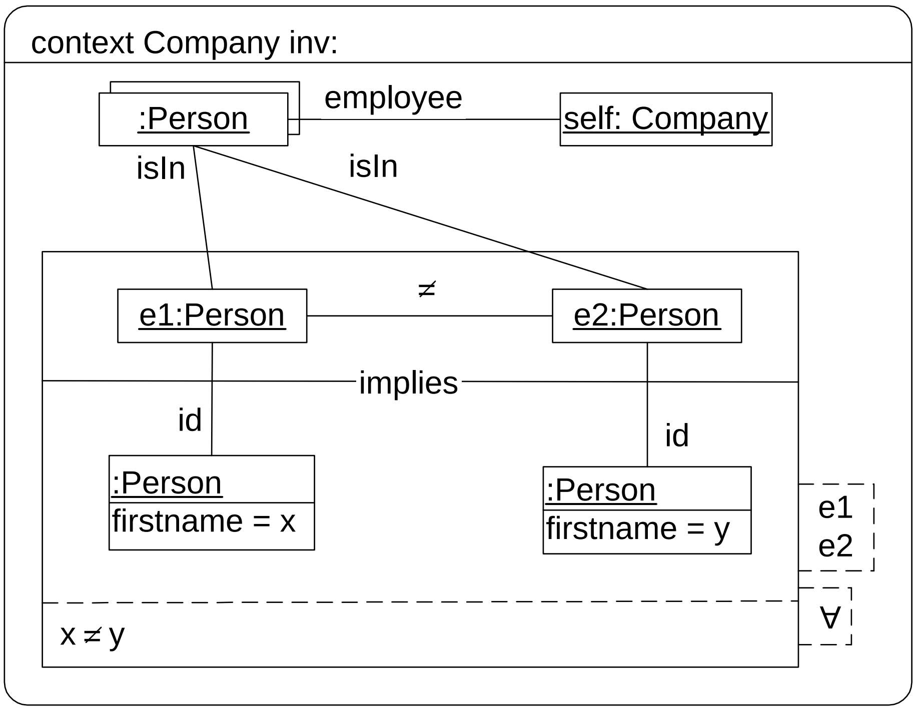


Figure 2.42: A forall operation

The constraint specifies that the firstnames of all employees of a company have to be different.

## The Exists Operation

The operation *exist* checks if a constraint is satisfied for at least one element of a collection. It has one iterator and an *exist* frame in which the property of the *exist* operation is visualized. The *exists* operation returns a Boolean value. On the right of the frame the ∃-operator and the iterator are depicted.

-
  ```
  context Company inv:
  self.employee->exists (p: Person | p.firstname = ´Jack´)
  ```

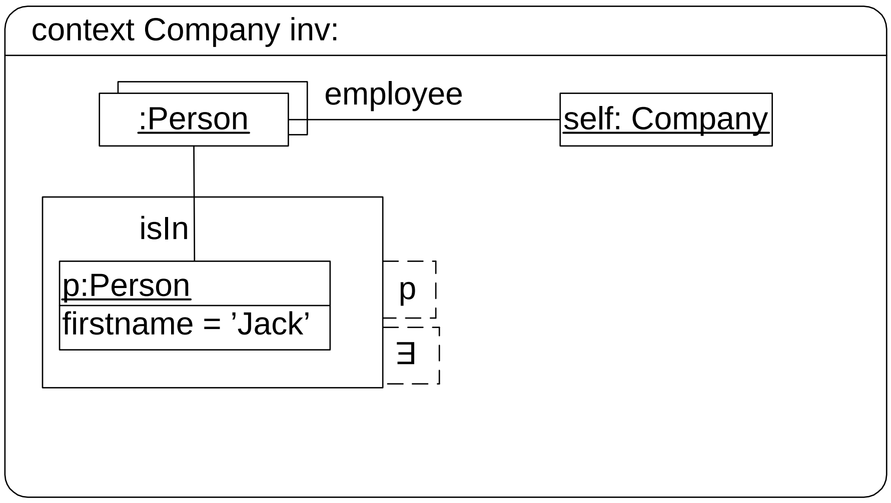


Figure 2.43: An exists operation

The constraint specifies that at least one employee of a company has the firstname ”Jack”.

## The Iterate Operation

The operation *iterate* has a frame in which properties are visualized, and an iterator. Additionally it has an accumulator with an initial value. The definition of the accumulator is visualized by the operation *isNew()*.
The operations *reject*, *select*, *forall*, *exists*, *collect*, can all be described in terms of *iterate*.

-
  ```
  context Company inv:
  self.employee->iterate (p: Person, acc: Set= Set()|
  acc->including(p| p.age > 50))->notEmpty
  ```

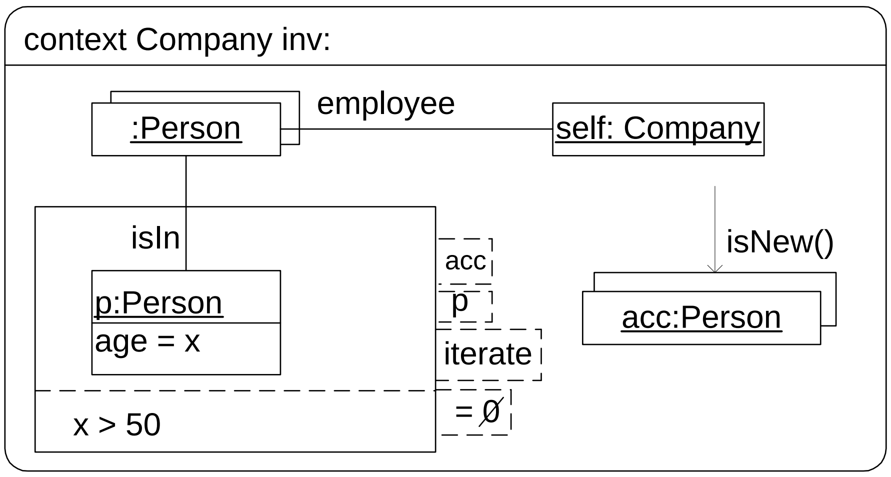


Figure 2.44: An iterate operation

This constraint specifies that in a company, the set of employees which are elder than 50, is not empty.

## The Sum Operation

The operation *sum* has a frame in which the element whose values are summed up, is visualized. This element is depicted at the frame above the $\sum$ symbol. To use the result of the *sum* in the condition section, the $\sum$ symbol is used.

- `context Person inv:`
  `self.job.salary->sum() > 50`

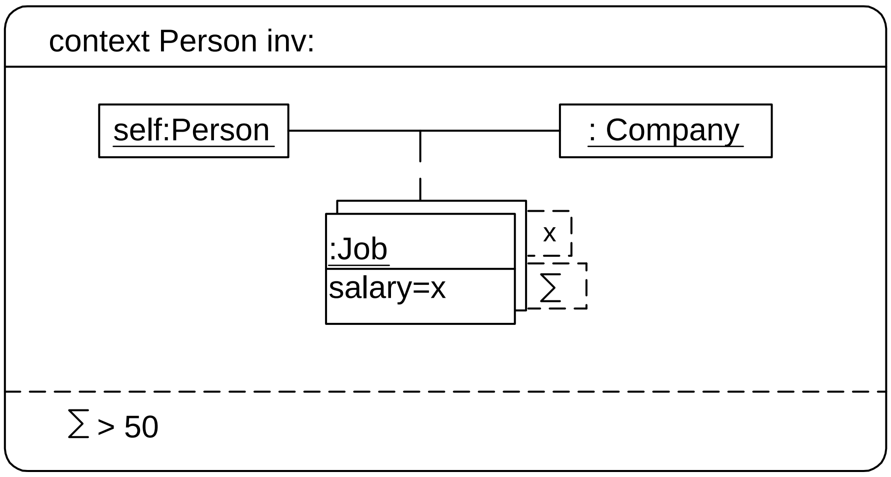


Figure 2.45: A sum operation

This constraint specifies that the sum of all job salaries of a person, is greater than 50.

### Abstract Syntax

Calls of the operations *select*, *reject*, *forall* and *exists* are abstracted to *IteratorExps* with corresponding names. In the case of *select* or *reject* the type of this *IteratorExp* is the corresponding collection type (Set, Bag or Sequence), in the case of *forall* and *exists* it is "Boolean" and in the case of sum it is the type of elements summed up. An *IteratorExp* refers to an iterator which is a *VariableDeclaration* with the name of the iterator as variable name. The referred type is the type of the iterator. Moreover, an *IteratorExp* has a reference to the body of the iteration which is an *OclExpression*. The use of several iterators has to be done by nested iterations in the abstract syntax.
The abstract syntax of an *iterate* operation is an *IterateExp* which refers to an iterator like the *IteratorExp* and additionally it refers to an accumulator, a *VariableDeclaration*. This accumulator variable is initialized by an *OclExpression*.
The abstract syntax of the *select* operation in Figure 2.39 (page 31) is shown on page 91, the abstract syntax of Figure 2.32 (page 26) is shown on page 89.

## 2.5 Messages

-
  ```
  context Subject::hasChanged() post:
  let message : OclMessage = observer^update(12, 14) in
  message.isSent()
  ```

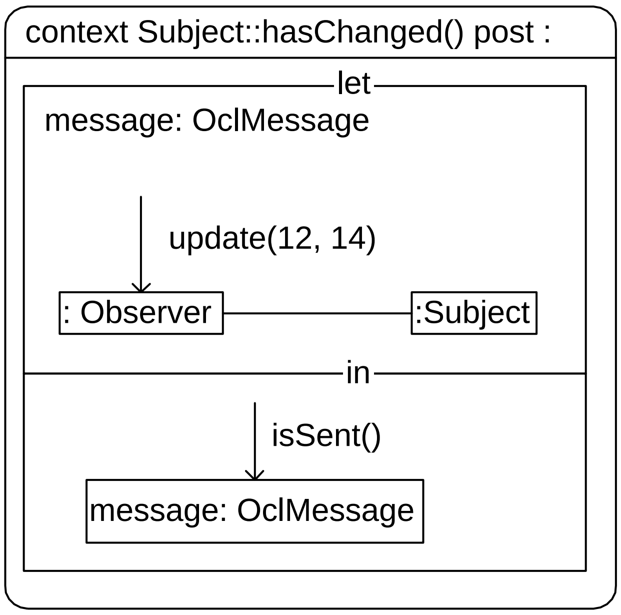


Figure 2.46: Messages

*observer^update(12, 14)* results in an instance of *OclMessage* which is defined in the let section of the constraint. *isSent()* is a standard operation on objects of type *OclMessage* and is visualized like any other operation.

-
  ```
  context Subject::hasChanged() post:
  let messages : Set[OclMessage] = observer->forall(o| o^update() ) in
  messages->forall(isSent())
  ```

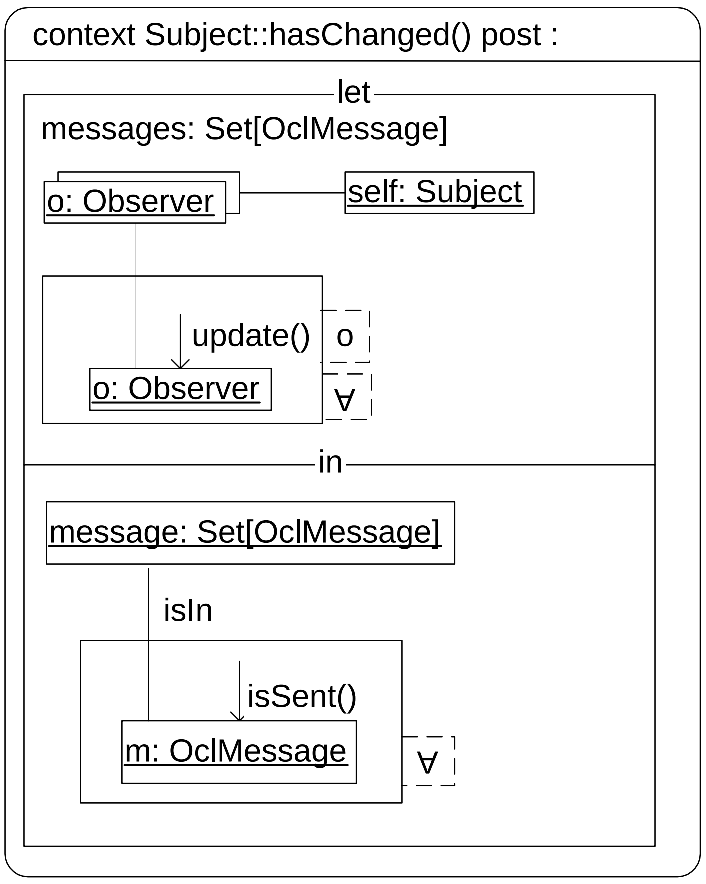


Figure 2.47: Messages

*update()* applied to all objects of type *observer* results in a set of *OclMessages*, defined in the *let* section. The *in* section of the constraint specifies that all of these messages are sent.

-
  ```
  context Person::give Salary(amount: Integer) post:
  let message: OclMessage = company^getMoney(amount) in
  message.hasReturned()
  and message.result() = true
  ```

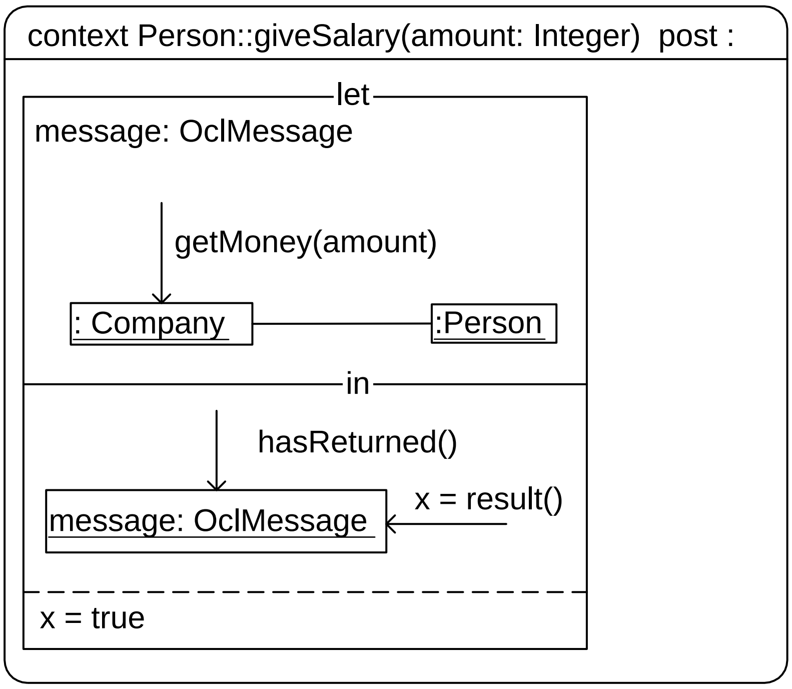


Figure 2.48: Messages

Operation *getMoney(amount)* of the class *Company* results in an *OclMessage*. The *in* section defines that *hasReturned()* and *result()* return *true*. Please note that if an operation returns a Boolean value and this value is left out in the visualization of the constraints body, it has to be true.

**Abstract Syntax**

The abstract syntax of a *message* is an *OclMessageExp* which either refers to a *SendAction* or to a *CallAction*. Again these action refers to a *Signal* or to an *Operation*. A detailed description of the abstract syntax of the constraint in Figure 2.46 is shown in the meta model instance on page 92.

## 2.6 Tuple

The definition of a tuple is visualized by a frame, the tuple name is depicted in the upper left corner of the frame. If only one tuple is defined the frame is optional. Inside this tuple frame, every tuple element gets its own frame (a rounded rectangle), where the element name, the element type, and the element value are denoted in the upper left corner. Inside the frame, the definition of the element value is represented.

-
  ```
  context Person def:
  let stats = managedCompanies->collect(c |
  {company: Company = c,
  numEmployees: Integer = c.employee->size(),
  wellpaidEmployees: Set(Person) =
  c.job->selection(salary > 10000).employee,
  totalSalary: Integer = c.job.salary->sum()}
  )
  ```

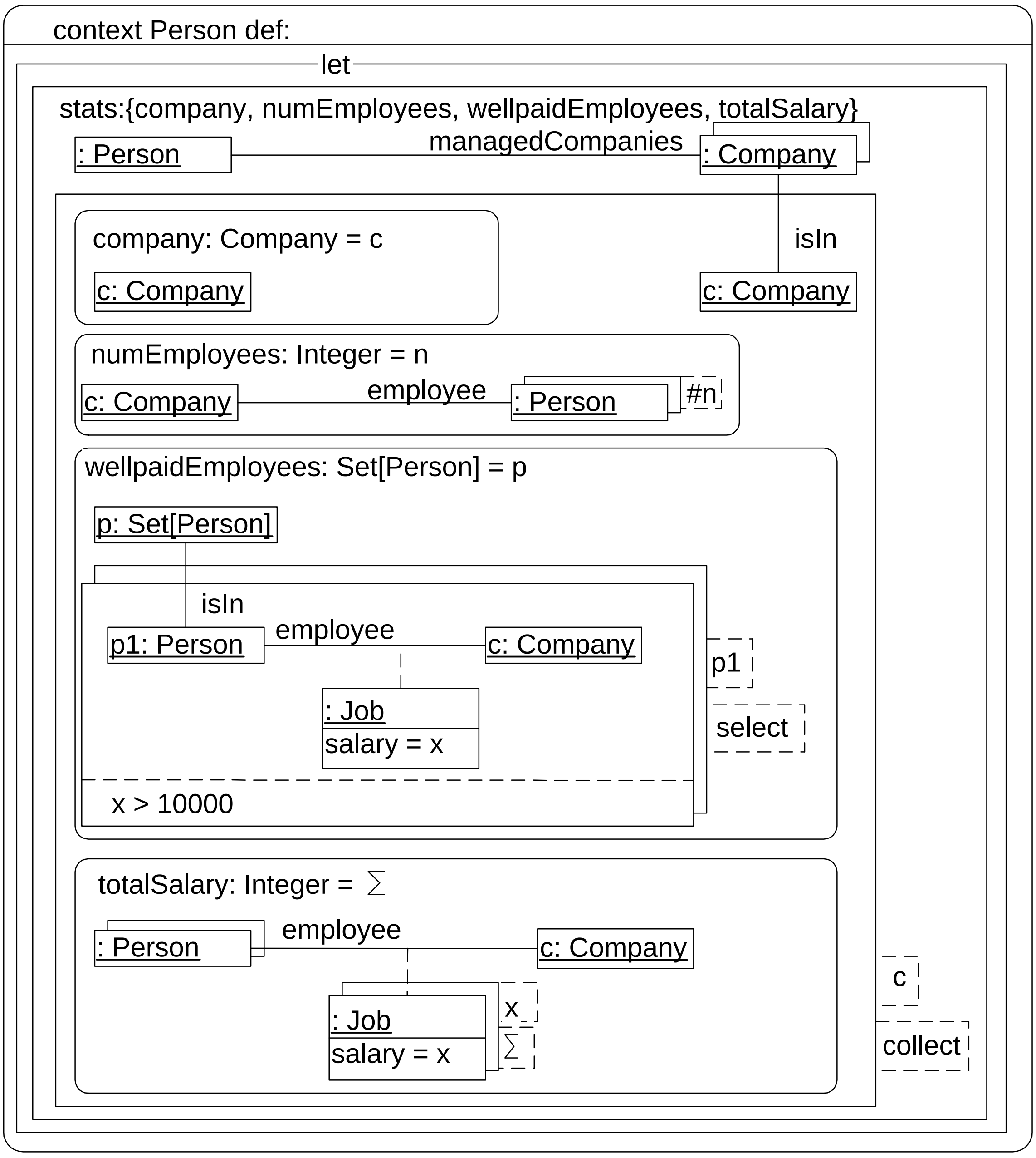


Figure 2.49: A tuple definition

A tuple *stats* is defined in the context Person. It contains four elements: the company *company*, the number of employees *numEmployees*, the best paid employees *wellpaidEmployees* and the total salary costs *totalSalary* of each company a person manages. This tuple is defined in a constraint of kind *def*, it is also known in the next constraint.
In this constraint, all tuples of type *stats* are sorted by its total salary which results in a collection. The operation *last()* returns the last element of a collection which is the tuple of type *stats* with the highest total salary. The tuple element *wellpaidEmployees* contains the person that is visualized by the operation *includes(self)*. *includes* returns true if the collection contains the assigned element. It is enough to represent in the tuple only the elements used, the others could be left out.

- `context Person inv:`
  `stats->sortedBy(totalSalary)->last().wellpaidEmployees->includes(self)`

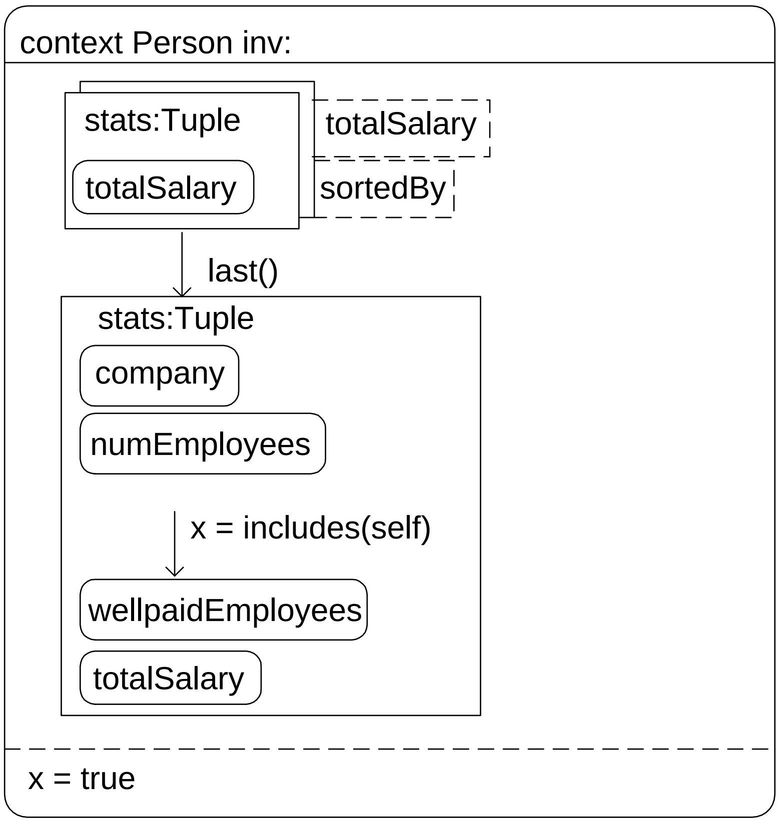


Figure 2.50: A tuple

The constraint specifies that each person being an instance of class Person is one of the best-paid employees of the company with the highest total salary.

**Abstract Syntax**

The abstract type of a tuple is *TupleType* which refers to its defined attributes which are *Attributes*. Those have a name and refer to the corresponding type.

## 2.7 Composition of Constraints

A constraint can be composed of other constraints defined before. It specifies the composition of all constraints inside a composition frame by "and". The constraints inside the frame are refered by their names, thus the constraints used have to be defined first.
The constraint that all employees of a company have different names and one of them has the firstname "Jack", can be expressed by the composition of the following two constraints:

- `context Company inv differentFirstnames:`
  `self.employee->forall (e1, e2: Person | e1 <> e2 implies`
  `e1.firstname <> e2.firstname)`

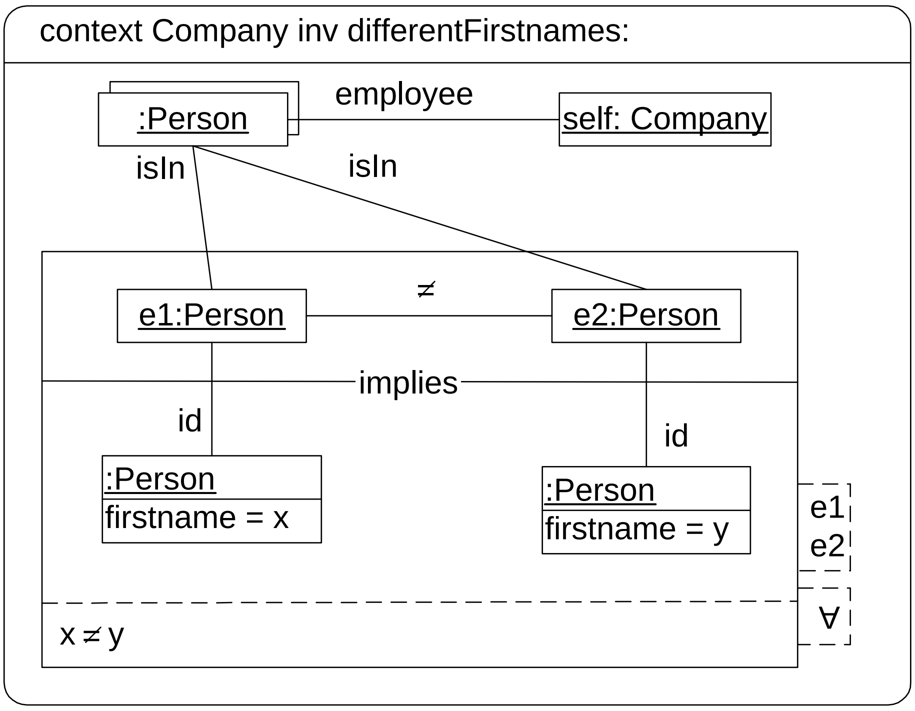


Figure 2.51: First constraint

-
  ```
  context Company inv oneJack:
  self.employee->exists (p: Person | p.firstname = ´Jack´)
  ```

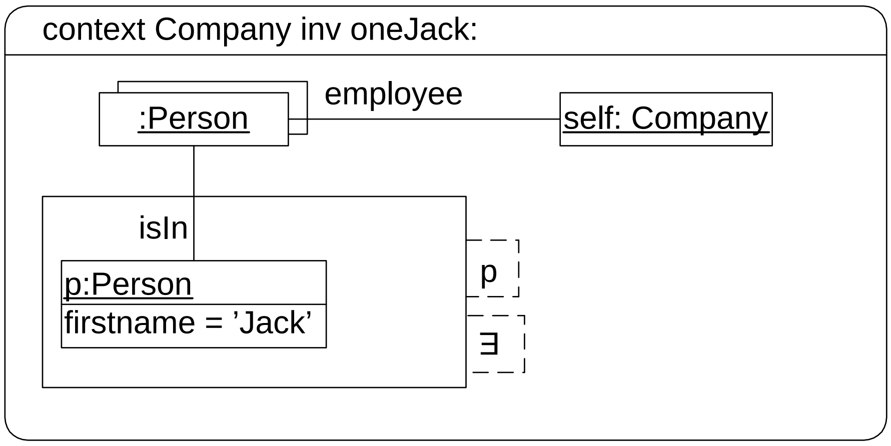


Figure 2.52: Second constraint

The composed constraint:

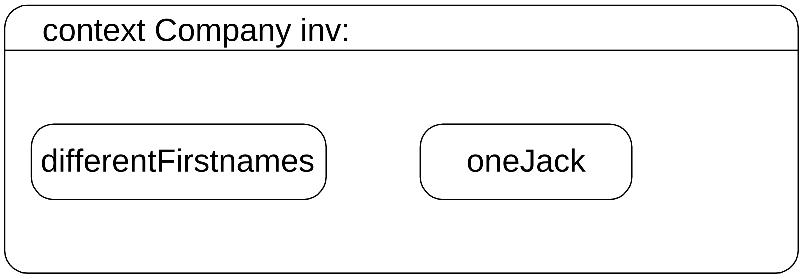


Figure 2.53: Composition constraint

### Abstract Syntax

The abstract syntax of a composition constraint is a combining of the bodys of the single constraints by the operation "and". A detailed description of the abstract syntax of the constraint in Figure 2.53 is shown in the meta model instance on page 93.

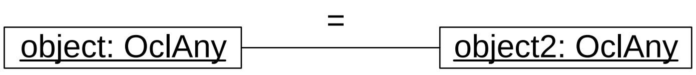
Chapter 3

# The OCL Standard Library

## 3.1 OclAny and OclVoid

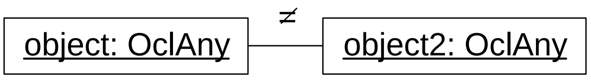
### 3.1.1 OclAny

- object = (object2: OclAny): Boolean



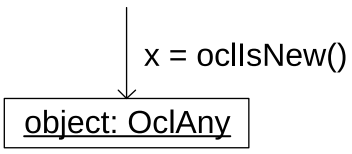
The test on equality of two objects of type *OclAny* is visualized by a link labeled by a = sign between the objects. *OclAny* is an abstract type, so the objects have to be substituted by the visualization of the corresponding type.

- object <> (object2: OclAny): Boolean



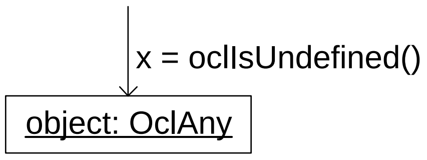
The test on inequality of two objects of type *OclAny* is visualized by a link labeled by a ≠ sign.

- object.oclIsNew(): Boolean



*oclIsNew()* is an operation that can only be used in a post condition. It evaluates to true if the object is created during performing the operation. The operation is visualized by an arrow labeled by *x=oclIsNew()* heading to the object; *x* contains the return value. The return value can be left out (what means that it is true in that case) if it is not used again.

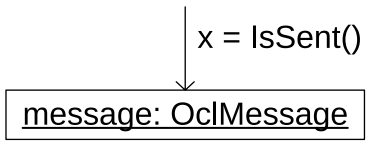
- object.oclIsUndefined(): Boolean



Applying *oclIsUndefined()* to an object of type *OclAny* returns true, if the object is equal to *OclUndefined*. The operation is visualized by an arrow labeled by *x=oclIsUndefined()*.

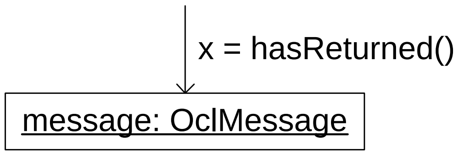
### 3.1.2 OclMessage

- message.isSent(): Boolean



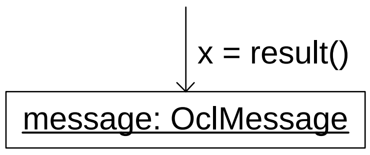
*isSent()* is visualized like an operation on an object. It is true if *message* has been sent to the target.

- message.hasReturned(): Boolean



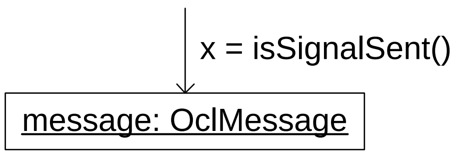
*hasReturned()* returns true if *message* is an operation call and the called operation has returned a value. This implies the fact that the message has been sent.

- message.result(): <<The return type of the called operation>>



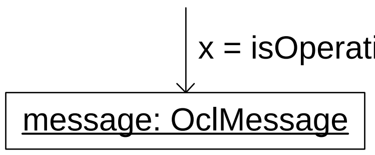
*result()* returns the result of the called operation.

- message.isSignalSent(): Boolean



*isSignalSent()* returns true if the OclMessage *message* represents the sending of an UML signal. The signal is visualized like an operation.

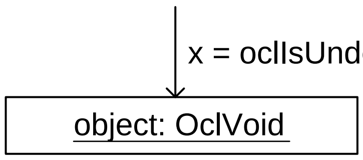
- message.isOperationCall(): Boolean



*isOperationCall()* returns true if the *OclMessage message* represents the sending of an UML operation call.

### 3.1.3 OclVoid

- OclUndefined.oclIsUndefined(): Boolean

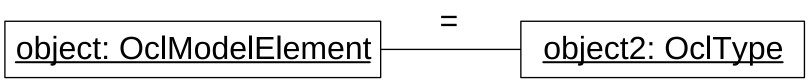


Applying *OclIsUndefined()* to an object of type *OclAny* returns true if *object* is equal to *OclUndefined*. This operation is visualized by an arrow labeled by *OclIsUndefined()*.

## 3.2 ModelElement Types

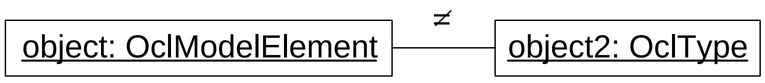
### 3.2.1 OclModelElement

- object = (object2: OclType): Boolean



The test on identity of an object of type *OclModelElement* and an object of type *OclAny* is visualized by a link labeled by a = sign between the objects.

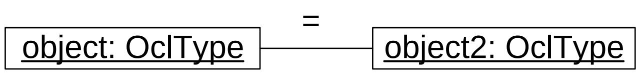
- object <> (object2: OclType): Boolean



The test on inequality of an object of type *OclModelElement* and an object of type *OclType* is visualized by a link labeled by a $\neq$ sign between the objects.

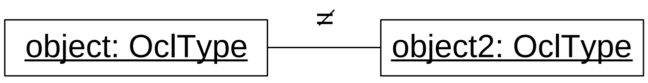
### 3.2.2 OclType

- object = (object2: OclType): Boolean



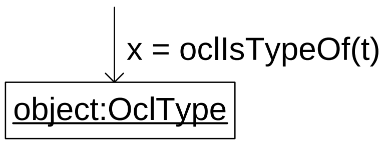
The test on equality of two objects of type *OclType* is visualized by a link labeled by a = sign between the objects.

- object <> (object2: OclType): Boolean



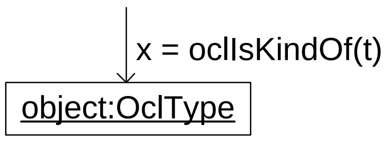
The test on inequality of two objects of type *OclType* is visualized by a link labeled by a ≠ sign between the objects.

- oclIsTypeOf(t: OclType): Boolean



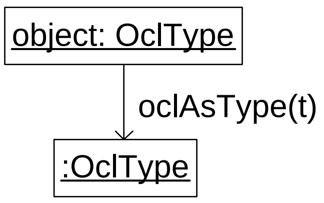
*oclIsTypeOf()* returns true if *t* and *object* have the same type. The operation is visualized by an arrow labeled by *oclIsTypeOf()* ending at *object*.

- oclIsKindOf(t: OclType): Boolean



*oclIsKindOf()* returns true if type *t* is equal to the object's type or a supertype of the object's type. The operation is visualized by an arrow labeled by *oclIsKindOf()* heading to *object*.

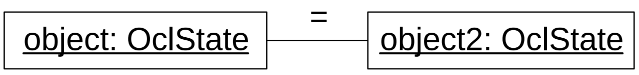
- oclAsType(t: OclType): instance of OclType



A property of a supertype can be accessed using *oclAsType(Type2)* which is visualized by an arrow labeled by *oclAsType()* from *object* to the resulting object.

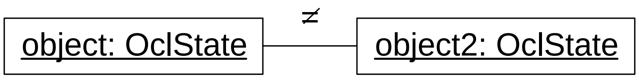
### 3.2.3 OclState

- object = (object2: OclState): Boolean



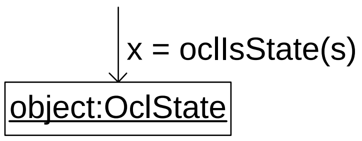
The test on equality of two objects of type *OclState* is visualized by a link labeled by a = sign between the objects.

- object <> (object2: OclState): Boolean



The test on inequality of two objects of type *OclState* is visualized by a link labeled by a ≠ sign between the objects.

- oclInState(s: OclState):Boolean



*oclIsInState(s)* returns true if the object is in state *s*. *s* is the name of a state in a state diagram. The operation is visualized by an arrow labeled by *oclIsInState(s)* heading to *object*.

## 3.3 Primitive Types

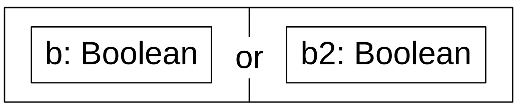
### 3.3.1 Real, Integer and String

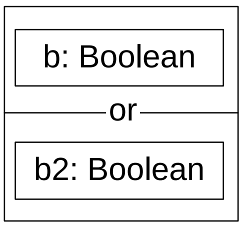
The standard types *Real* and *Integer* represent the mathematical concepts of real and integer and their properties. *String* represents strings and their properties. They are only used textual in the condition section of a constraint, so they are not visualized here.

### 3.3.2 Boolean

This type can be visualized and can also be used textually in the condition section of a constraint. In the following subsection, the textual followed by the visual representation is shown.

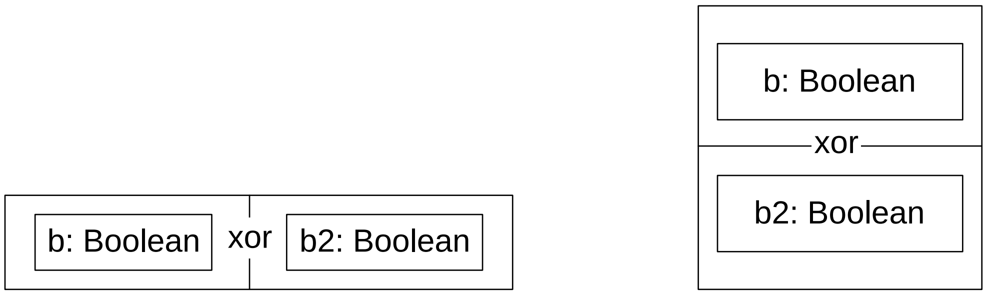
- b or (b2: Boolean): Boolean





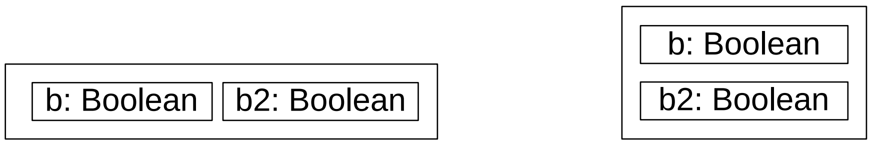
Combining sub expressions by ”or” is visualized by an ”or” frame, the expressions left and right or above and below the keyword *or* respectively, are combined by ”or”.

- b xor (b2: Boolean): Boolean



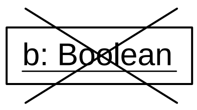
Combining sub expressions by ”xor” is visualized by an ”xor” frame, the expressions left and right or above and below the keyword *xor* respectively, are combined by an exclusive or.

- b and (b2: Boolean): Boolean



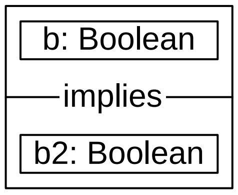
Combining sub expressions by ”and” is not explicitly visualized. Drawing expressions within one frame, they are automatically combined by ”and”.

- not b: Boolean



The negation of an expression is visualized by crossing out the expression.

- b implies (b2: Boolean): Boolean



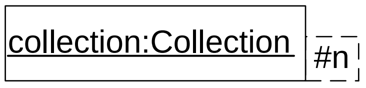
An *implies* expression is visualized in an *implies* frame. Anything above the keyword *implies* describes the premise. When this premise is true it implies the conclusion denoted below *implies*.

## 3.4 Collection-Related Types

### 3.4.1 Collection

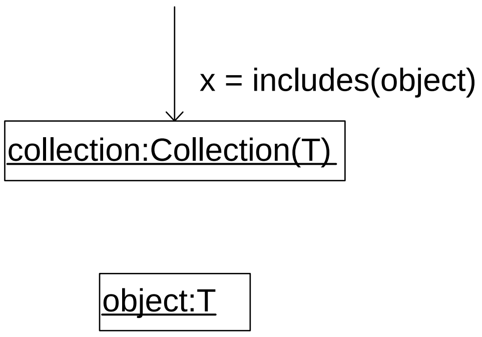
This section describes all general operations on collections that can be applied to each of the collection types, i.e. *Set*, *Bag* and *Sequence*. The frame containing *collection:Collection* must be replaced by the corresponding frame of the used subtype.

- collection->size(): Integer



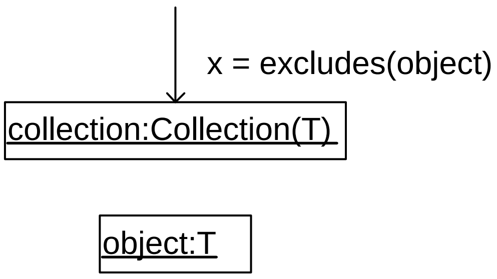
*size* is applied to a collection, the variable $n$ contains the number of elements in the collection. This operation is visualized by a frame, $\#n$ is depicted on the right of the frame inside a dashed box.

- collection->includes(object: T): Boolean



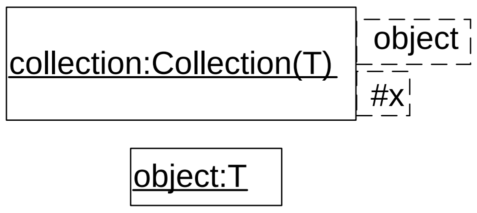
*includes(object)* returns a Boolean value. It is true if the object *object* is an element of the collection *collection*. This is visualized by an arrow labeled by *x=includes(object)* heading to *collection*. $x$ is *true* or *false*.

- collection->excludes(object: T): Boolean



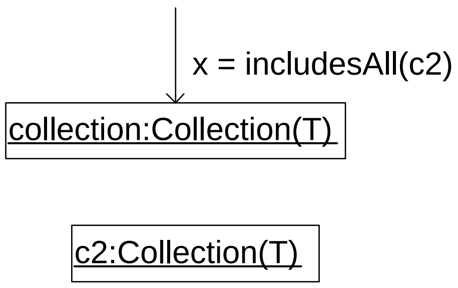
*excludes(object)* returns a Boolean value. It is true if the collection *collection* does not contain the object *object*. This is visualized by an arrow labeled by *x=excludes(object)* heading to *collection*. $x$ is *true* or *false*.

- collection->count(object: T): Integer



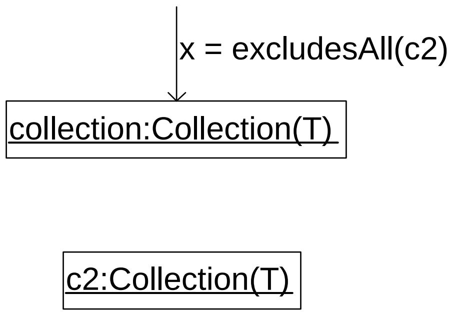
*count(object)* returns the number of occurrences of *object* in *collection*. $x$ is an integer. This operation is visualized by a frame, the counted object and $\#x$ are depicted on the right of the frame, each inside a dashed box.

- collection-> includesAll(c2: Collection(T)): Boolean



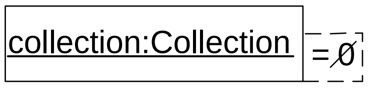
*includesAll(c2)* returns true if all elements of collection *c2* are elements of collection *collection*. It is visualized by an arrow labeled by *x=includes(object)* heading to *collection*.

- collection->excludesAll(c2: Collection(T)): Boolean



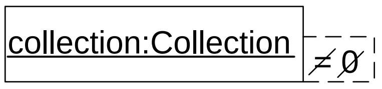
*excludesAll(c2)* returns true if the collection *collection* contains no elements of collection *c2*. It is visualized by an arrow labeled by *x=excludes(object)* heading to *collection*.

- collection->isEmpty(): Boolean



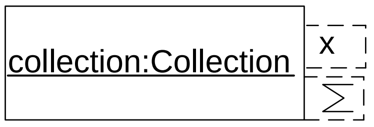
*isEmpty()* is applied to a collection and expresses that the collection contains no element. This operation is visualized by a frame, $= \emptyset$ is depicted on the right of the frame inside a dashed box. The visualization uses the set-theoretic representation of an empty set.

- collection->notEmpty(): Boolean



*notEmpty()* is applied to a collection and expresses that the collection is not empty. This operation is visualized by a frame, $\neq \emptyset$ is depicted on the right of the frame inside a dashed box.

- collection->sum(): T

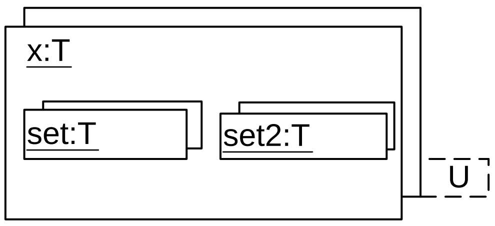


*sum()* sums up the values of attributes of all elements in the collection *collection*. This operation is visualized by a frame, the attribute $x$ which is summed up, and the $\sum$ sign are depicted on the right of the frame, each inside a dashed box.

The following sections contain the operations that are typical for particular collection types and are not defined for the other collection types.

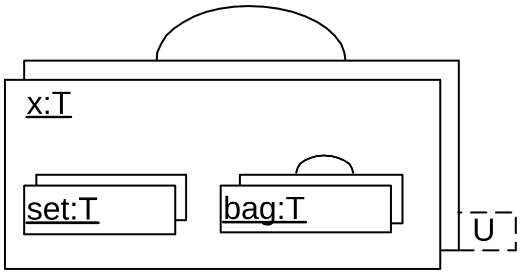
### 3.4.2 Set

- set->union(set2: Set(T)): Set(T)



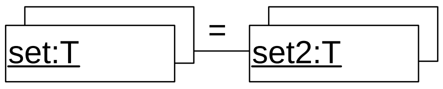
The union of two sets is represented by a frame that contains the name of the resulting set in the upper left corner. The sets that are unified are also placed into the frame. On the right of the frame the mathematical union sign is depicted inside a dashed box.

- set->union(bag: Bag(T)): Bag(T)



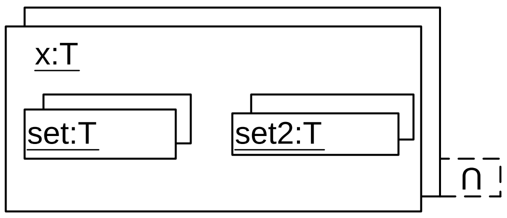
The union of a set and a bag is represented by a frame that contains the name of the resulting bag in the upper left corner. The set and bag that are unified are also placed into the frame. On the right of the frame the mathematical union sign is depicted inside a dashed box.

- set = (set2: Set(T)): Boolean



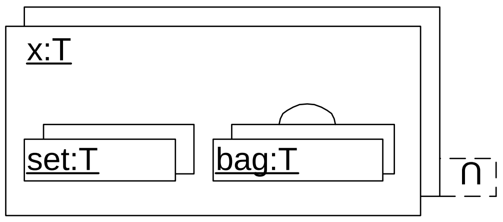
The test on equality of two sets is visualized by a link labeled by a = sign between the sets.

- set->intersection(set2: Set(T)): Set(T)



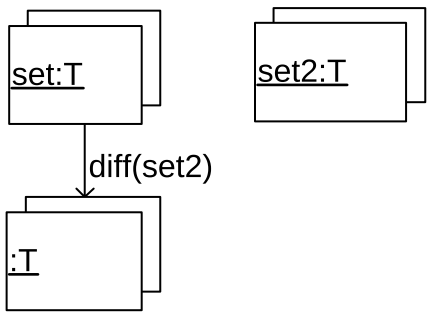
The intersection of two sets is represented by a frame that contains the name of the resulting set in the upper left corner. The two sets are also placed into the frame. On the right of the frame the mathematical intersection sign is depicted inside a dashed box.

- set->intersection(bag: Bag(T)): Set(T)



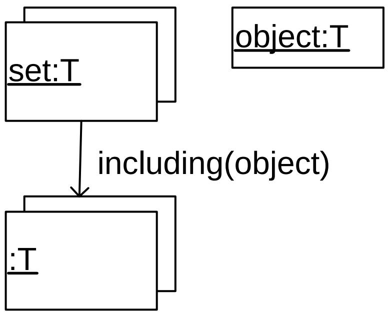
The intersection of a set and a bag is represented by a frame that contains the name of the resulting set in the upper left corner. The set and bag are also placed into the frame. On the right of the frame the mathematical intersection sign is depicted inside a dashed box.

- set-(set2: Set(T)): Set(T)



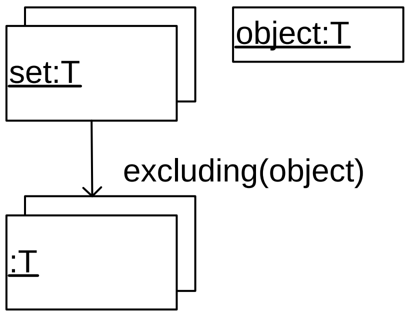
The difference of two sets results in a new set. This is represented by an arrow labeled by *diff(set2)* from set *set* to the resulting set. The set *set2* is subtracted from *set*.

- set->including(object: T): Set(T)



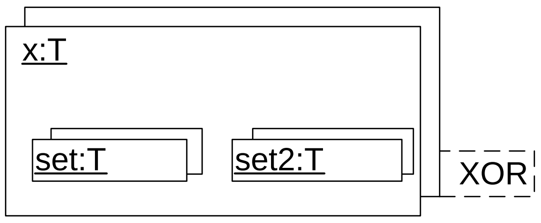
*including(object)* unifies a set *set* with the object *object*, represented by an operation arrow labeled by *including(object)* between the original set *set* and the resulting set.

- set->excluding(object: T): Set(T)



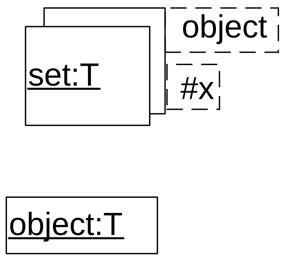
*excluding(object)* returns a set containing all elements of *set* without *object*, represented by an operation arrow labeled by *excluding(object)*, between the original set *set* and the resulting set .

- set->symmetricDifference(set2: Set(T)): Set(T)



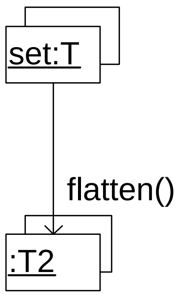
The symmetric difference of two sets is represented by a frame that contains the name of the resulting set in the upper left corner. The set and the bag are also placed into the frame. On the right of the frame *XOR* is depicted inside a dashed box.

- set->count(object: T): Integer



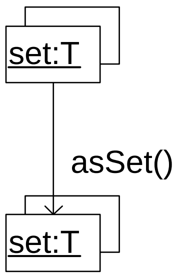
*count(object)* returns the number of occurrences of object *object* in set *set*. On the right of the frame the object to be counted and $\#x$ is depicted, each inside a dashed box. $x$ contains the number of counted objects.

- set->flatten(): Set(T2)



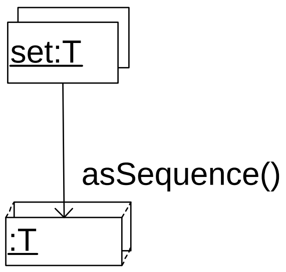
Applying *flatten()* to a set *set* returns a set that may contain elements of another type as the set type $T$ and is represented by an arrow labeled by *flatten()* from the original set *set* to the resulting set.

- set->asSet(): Set(T)



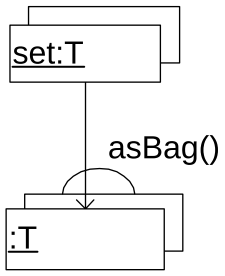
Applying *asSet()* to a set returns the same set and is represented by an arrow labeled by *asSet()* from the original set *set* to the resulting set *set*.

- set->asSequence(): Sequence(T)



Applying *asSequence()* to a set returns a sequence that contains all elements of the set. This is represented by an arrow labeled by *asSequence()* from the set *set* to the resulting sequence.

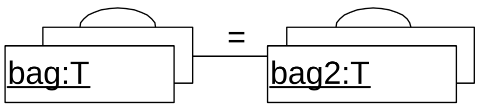
- set->asBag(): Bag(T)



Applying *asBag()* to a set returns a bag that contains all elements of the set. This is represented by an arrow labeled by *asBag()* from the set *set* to the resulting bag.

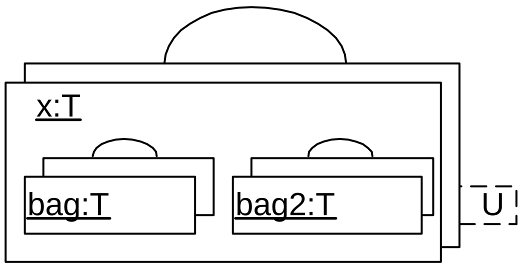
### 3.4.3 Bag

- bag = (bag2: Bag(T)): Boolean



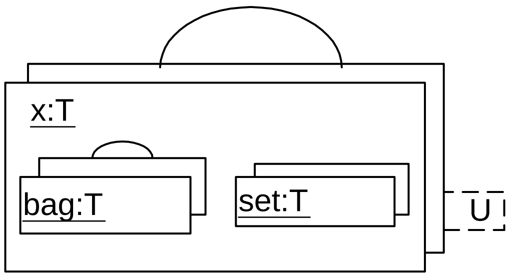
The test on equality of two bags is visualized by a link labeled by a = sign between the bags.

- bag->union(bag2: Bag(T)): Bag(T)



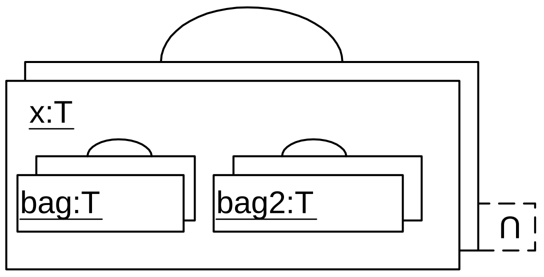
The union of two bags is represented by a frame that contains the name of the resulting bag in the upper left corner. The bags that are unified are also placed into the frame. On the right of the frame the mathematical union sign is depicted inside a dashed box.

- bag->union(set: Set(T)): Bag(T)



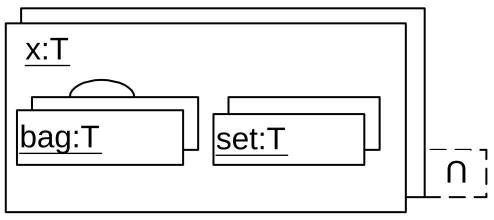
The union of a bag and a set is represented by a frame that contains the name of the resulting bag in the upper left corner. The bag and the set that are unified are also placed into the frame. On the right of the frame the mathematical union sign is depicted inside a dashed box.

- bag->intersection(bag2: Bag(T)): Bag(T)



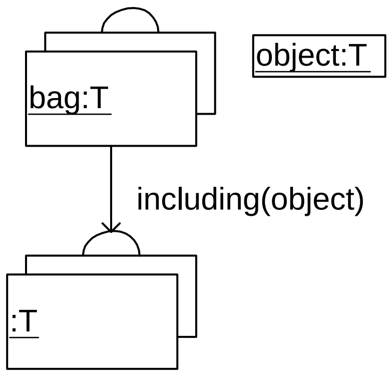
The intersection of two bags is represented by a frame that contains the name of the resulting bag in the upper left corner, The two bags are also placed into the frame. On the right of the frame the mathematical intersection sign is depicted inside a dashed box.

- bag->intersection(set: Set(T)): Set(T)



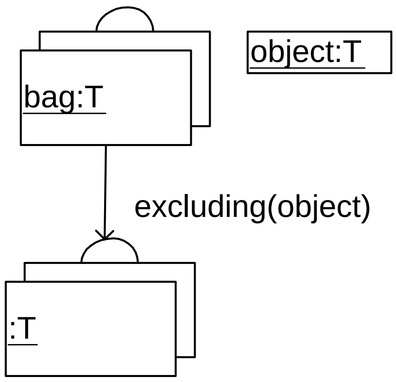
The intersection of a bag and a set is represented by a frame that contains the name of the resulting set in the upper left corner. The bag and set are also placed into the frame. On the right of the frame the mathematical intersection sign is depicted inside a dashed box.

- bag->including(object: T): Bag(T)



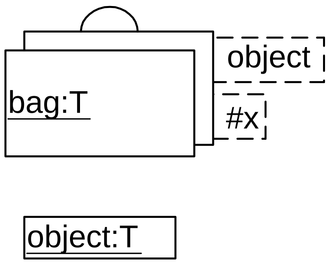
*including(object)* unifies a bag *bag* with an object *object*, represented by an arrow labeled by *including(object)* from the original bag *bag* to the resulting bag.

- bag->excluding(object: T): Bag(T)



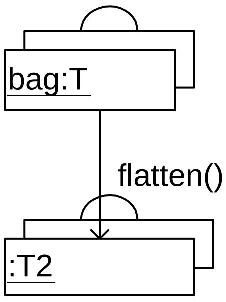
*excluding(object)* returns a bag containing all elements of *bag* without *object*, represented by an arrow labeled by *excluding(object)* from the original bag *bag* to the resulting bag.

- bag->count(object: T): Integer



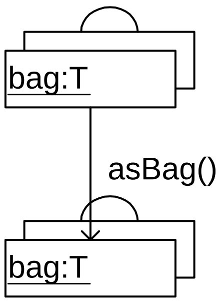
*count(object)* returns the number of occurrences of the object *object* in the bag *bag*. On the right of the frame the object to be counted and $\#x$ are depicted, each inside a dashed box. $x$ contains the resulting number.

- bag->flatten(): Bag(T2)



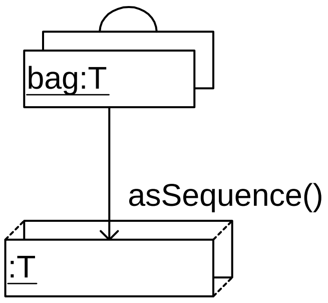
Applying *flatten()* to a bag *bag* returns a bag that may contain elements of another type as the bag type *T* and is represented by an arrow labeled by *flatten()* from the original bag *bag* to the resulting bag.

- bag->asBag(): Bag(T)



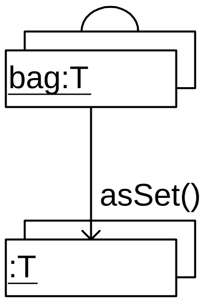
Applying *asBag()* to a bag *bag* returns the same bag and is represented by an arrow labeled by *asBag()* from the original bag *bag* to the resulting bag *bag*.

- bag->asSequence(): Sequence(T)



Applying *asSequence()* to a bag *bag* returns a sequence that contains all elements of the bag. This is represented by an arrow labeled by *asSequence()* from the bag *bag* to the sequence.

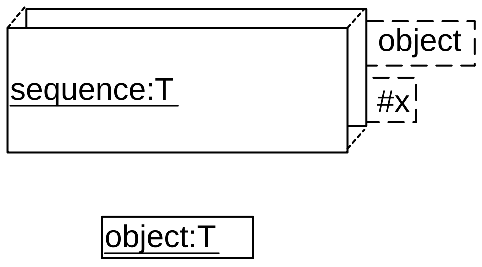
- bag->asSet(): Set()



Applying *asSet()* to a bag *bag* returns a set that contains all elements of the bag with duplicates removed. This is represented by an arrow labeled by *asSet()* from the bag *bag* to the resulting set.

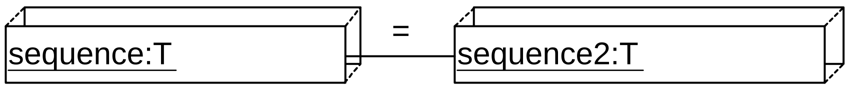
### 3.4.4 Sequence

- sequence->count(object: T): Integer



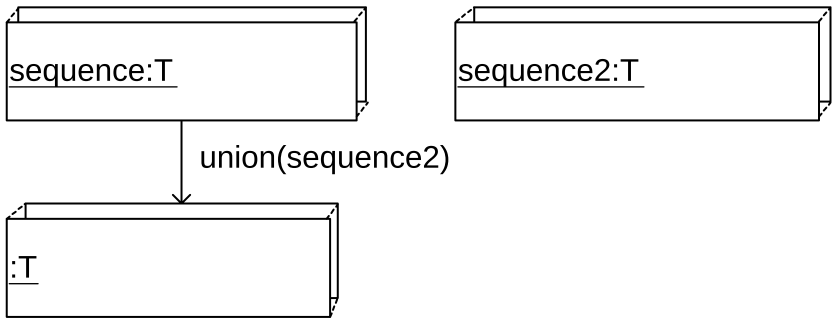
*count(object)* returns the number of occurrences of the object *object* in the sequence *sequence*. On the right of the frame, the object to be counted and $\#x$ are depicted, each inside a dashed box. $x$ contains the resulting number.

- sequence = (sequence2: Sequence(T)): Boolean



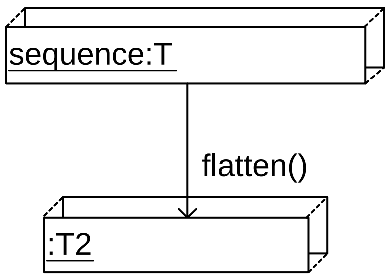
The test on equality of two sequences is visualized by a link labeled by a = sign between the sequences.

- sequence->union(sequence2: Sequence(T)): Sequence(T)



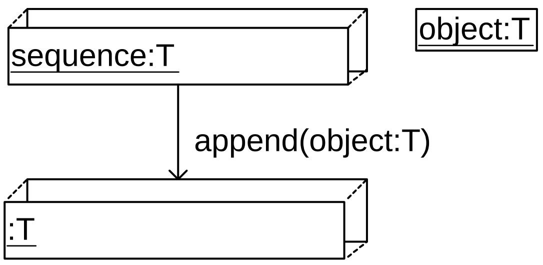
*union(sequence2)* returns a sequence consisting of all elements in the sequence *sequence* followed by all elements in the sequence *sequence2*, represented by an arrow labeled by *union(sequence2)* from the original sequence *sequence* to the resulting sequence *sequence2*.

- sequence->flatten(): Sequence(T2)



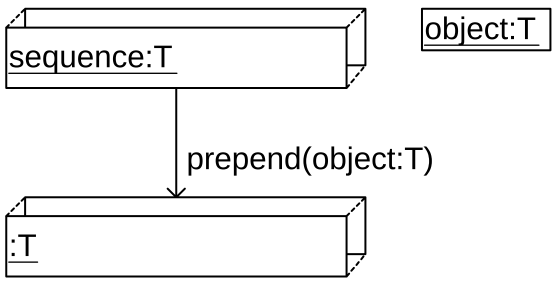
Applying *flatten()* to a sequence *sequence* returns a sequence that may contain elements of another type as the sequence type $T$ and is represented by an arrow labeled by *flatten()* from the original sequence *sequence* to the resulting sequence.

- sequence->append(object: T): Sequence(T)



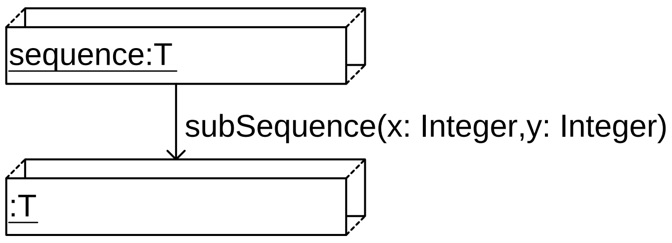
*append(object)* appends an object to a sequence *sequence*, represented by an arrow labeled by *append(object)* from the original sequence *sequence* to the resulting sequence.

- sequence->prepend(object: T): Sequence(T)



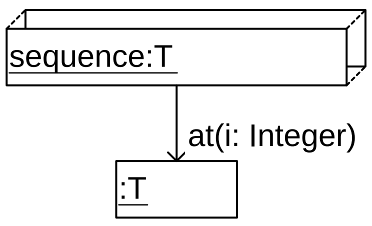
*prepend(object)* prepends an object to a sequence *sequence*, represented by an arrow labeled by *prepend(object)* from the original sequence *sequence* to the resulting sequence.

- sequence->subSequence(lower: Integer, upper: Integer): Sequence(T)



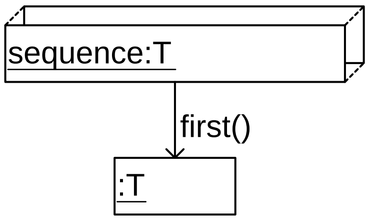
Applying *subSequence(x, y)* to a sequence *sequence* returns a subsequence containing the objects of the sequence starting at number $x$, up to and including element number $y$. This is represented by an arrow labeled by *subSequence(x, y)* from the original sequence *sequence* to the resulting sequence.

- sequence->at(i: Integer): T



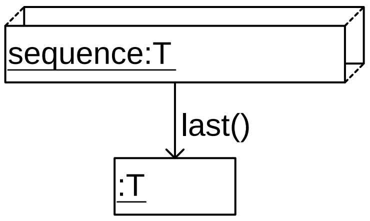
Applying *at(i)* to a sequence *sequence* returns the i-th element of the sequence, represented by an arrow labeled by *at(i)* from the sequence *sequence* to the returned object.

- sequence->first(): T



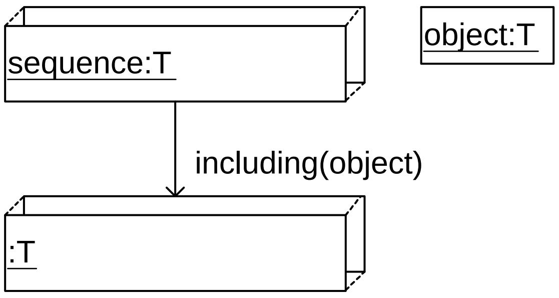
*first()* returns the first element of the sequence *sequence*, represented by an arrow labeled by *first()* from the sequence *sequence* to the returned object.

- sequence->last(): T



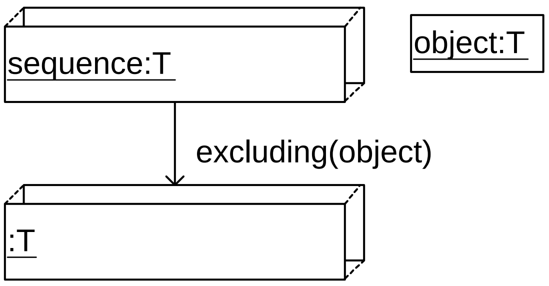
*last()* returns the last element of the sequence *sequence*, represented by an arrow labeled by *last()* from the sequence *sequence* to the returned object.

- sequence->including(object: T): Sequence(T)



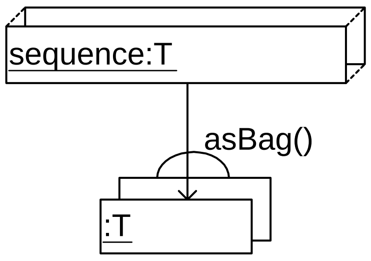
*including(object)* unifies a sequence *sequence* with an object, represented by an arrow labeled by *including(object)* from the original sequence *sequence* to the resulting sequence.

- sequence->excluding(object: T): Sequence(T)



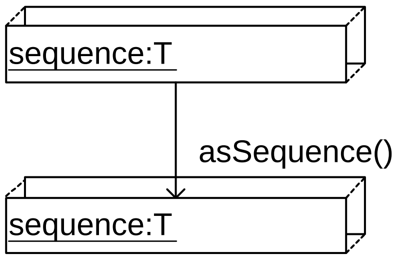
*excluding(object)* returns the sequence containing all elements of *sequence* without *object*, represented by an arrow labeled by *excluding(object)* from the original sequence *sequence* to the resulting sequence.

- sequence->asBag(): Bag(T)



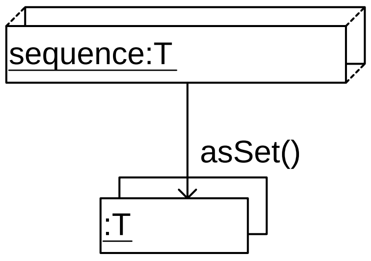
Applying *asBag()* to a sequence *sequence* returns a bag that contains all elements of the sequence. This is represented by an arrow labeled by *asBag()* from the sequence *sequence* to the resulting bag.

- sequence->asSequence(): Sequence(T)



Applying *asSequence()* to a sequence *sequence* returns the same sequence and is represented by an arrow labeled by *asSequence()* from the original sequence *sequence* to the resulting sequence.

- sequence->asSet(): Set(T)



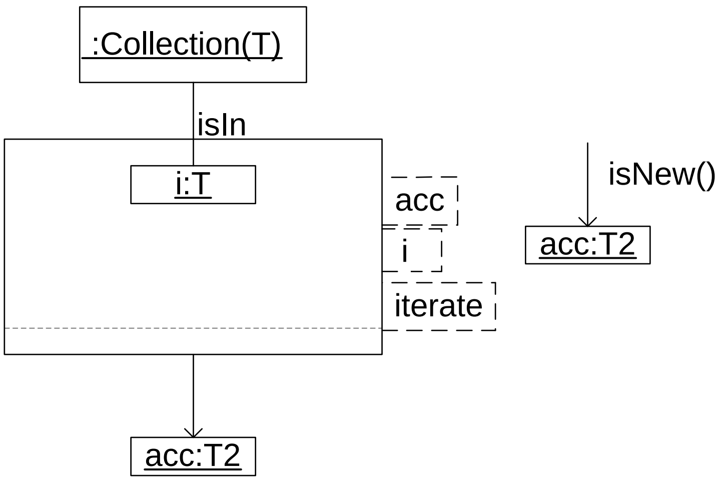
Applying *asSet()* to a sequence *sequence* returns a set that contains all elements of the sequence, where duplicates are removed. This is represented by an arrow labeled by *asSet()* from the sequence *sequence* to the set.

## 3.5 Predefined OclIterator Library

### 3.5.1 Collection

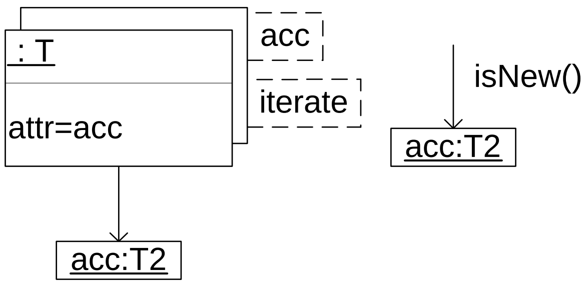
The operations in the following section are defined on all collection types. The collection elements have type *T*. The collection frame has to be substituted by the frame of the corresponding collection subtype.

- Collection(T)->iterate(i: T; acc: T2; expression: OclExpression):Boolean



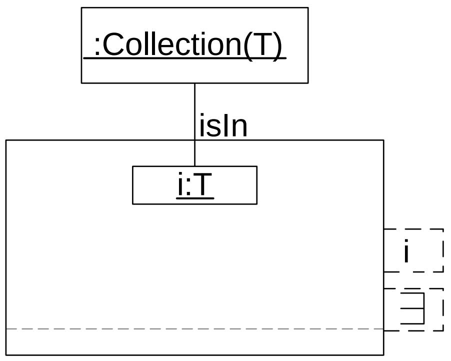
The *iterate* operation is applied to a collection and has an iterator, an accumulator and a body. The expression inside the body is visualized in a frame at which the accumulator, the iterator and the keyword *iterate* are depicted, each inside a dashed box. The accumulator gets an initial value. In the figure above *acc* is initialized by a new created instance of type *T2* (generally it can be an initial value of a variable of any type). The *iterate* operation returns the accumulator *acc*. The operations *exists*, *forall*, *select*, *reject*, *collect* can all be described in terms of iterate.
If the value over which is iterated is just an attribute of a collection element, the following shortcut for iterator operations can be used:

- Collection(T)->iterate(i: T; acc: T2; expression: OclExpression):Boolean



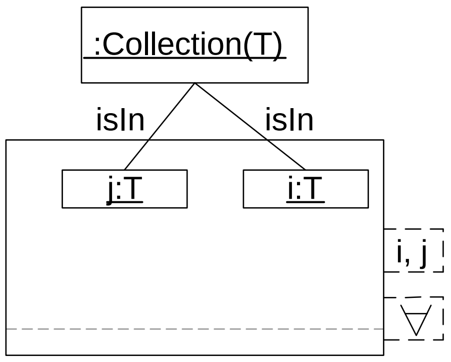
In the figures above the result which may be used in further subexpressions, is represented. It can be a collection, a Boolean value etc. If it is not further used, it can be left out, e.g. it is not needed if just the number of elements is interesting using operation *size()*. (In the previous and in the following visualizations, the resulting collection is left out for the purpose of clearness and better understanding of the representations.)

- Collection(T)->exists(expression: OclExpression):Boolean



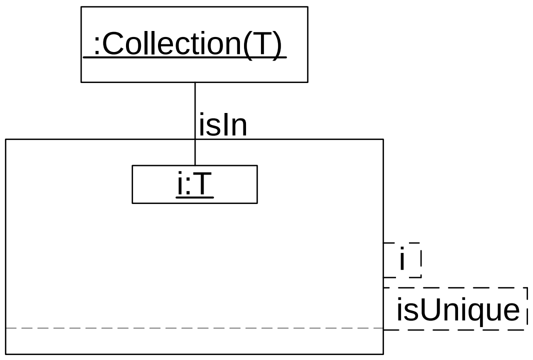
The *exists* operation is applied to a collection and has an iterator and a body. The expression inside the body is visualized in a frame at which the iterator and the ∃ operator are depicted, each inside a dashed box. The return value of *exists* is of type Boolean and the expression inside the body has to be satisfied for at least one element in the collection. The application of the expression inside the body to a collection element is represented by the helper operation *isIn*. Inside the frame all conditions one element has to satisfy, can be visualized; the frame can have its own condition section.

- Collection(T)->forall(expression: OclExpression):Boolean



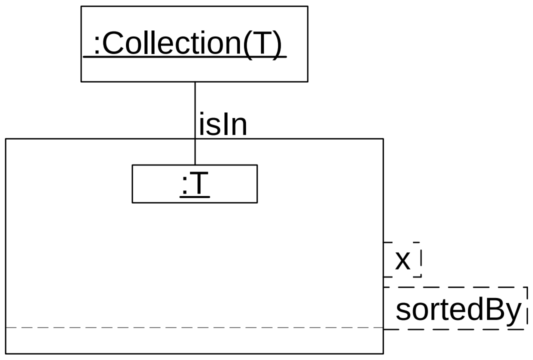
The *forall* operation is applied to a collection and has one or two iterators and a body. The expression inside the body is visualized in a frame at which the iterator/iterators and the ∀ operator are depicted, each inside a dashed box. The return value of *forall* is of type Boolean and the expression inside the body has to be satisfied for all elements in the collection. The application of the expression inside the body to each collection element is represented by the helper operation *isIn*. Inside the frame all conditions each collection element has to satisfy, can be visualized; the frame can have its own condition section.

- Collection(T)->isUnique(expression: OclExpression):Boolean



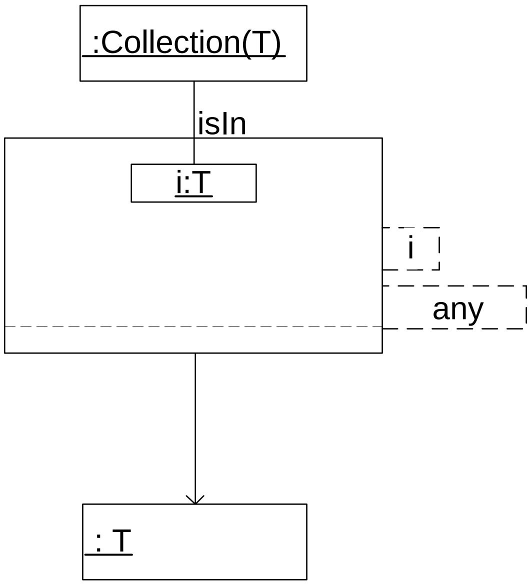
The *isUnique* operation is applied to a collection and has one iterator and a body. The expression inside the body is visualized in a frame at which the iterator and the keyword *isUnique* are depicted, each inside a dashed box. The return value of *isUnique* is of type Boolean and is true if the expression inside the body evaluates to a different value for each element in the collection. The application of the expression inside the body to each collection element is represented by the helper operation *isIn*. Inside the frame all conditions can be visualized, the frame can have its own condition section.

- Collection(T)->sortedBy(x:T2):Collection(T)



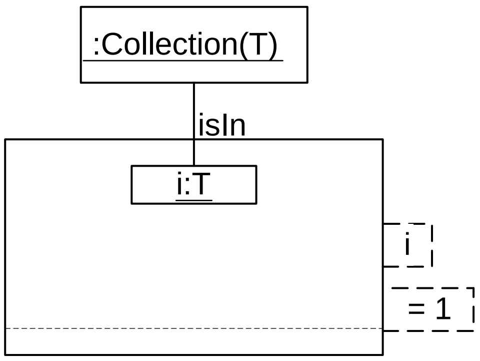
The *sortedBy* operation is applied to a collection and has an iterator and a body. The expression inside the body is visualized in a frame at which the iterator and the keyword *sortedBy* are depicted, each inside a dashed box. The collection elements are sorted by the iterator and the sorted collection is returned. Generally the return value of *sortedBy* is a sequence. *isIn* is used here, too.

- Collection(T)->any(expression: OclExpression):T



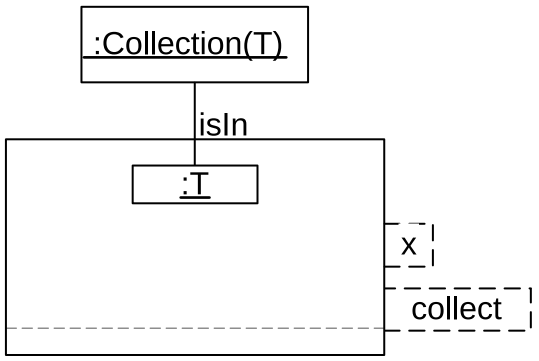
The *any* operation is applied to a collection and has an iterator and a body. The expression inside the body is visualized in a frame at which the iterator and the keyword *any* are depicted, each inside a dashed box. *any* returns one collection element which satisfies the condition in the body, or an object of type *OclUndefined* if no such element exists. *isIn* is used here, too.

- Collection(T)->one(expression: OclExpression):Boolean



The *one* operation is applied to a collection and has an iterator and a body. The expression inside the body is visualized in a frame at which the iterator and *=1* are depicted, each inside a dashed box. *one* returns true if exactly one collection element satisfies the condition inside the body. *isIn* is used here, too.

- Collection(T)->collect(expression: OclExpression):Collection(T2)

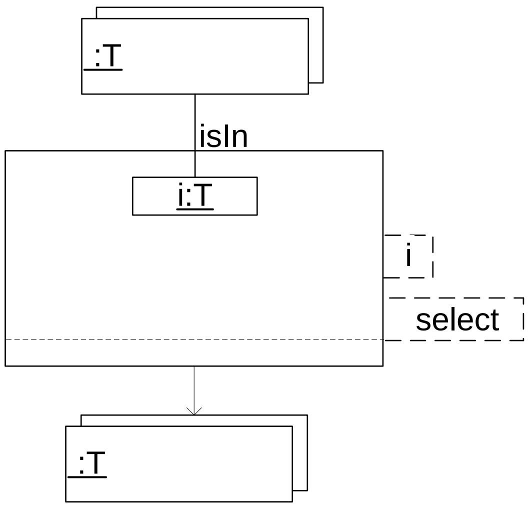


The *collect* operation is applied to a collection and has one or two iterators and a body. The expression inside the body is visualized in a frame at which the iterator/iterators and the keyword *collect* are depicted, each inside a dashed box. All elements that satisfy the condition in the *collect* frame are returned as collection of the iterator type.

The following operations are defined for each collection subtype and have different results.
All these operations use the *isIn* operation to visualize the properties of the body.

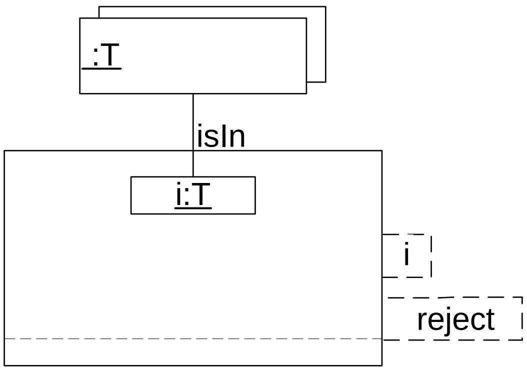
### 3.5.2 Set

- Set(T)->select(expression: OclExpression)->Set(T)



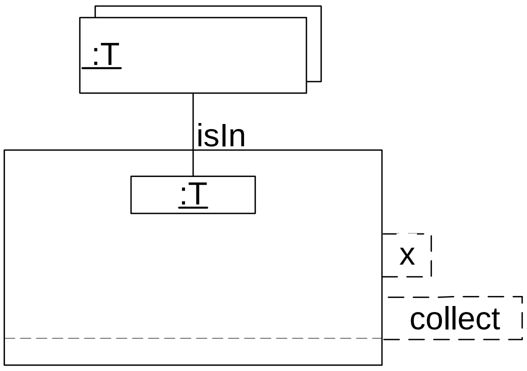
Applying operation *select* to a set returns a subset with elements having the specified properties. These selecting properties are visualized in the *select* frame. On the right of the frame, the iterator and the keyword *select* are depicted, each inside a dashed box. The evaluation of the select expression results in a set of type *T*.

- Set(T)->reject(expression: OclExpression)->Set(T)



Applying operation *reject* to a set returns a subset with elements not having the specified properties. These rejecting properties are visualized in the *reject* frame. On the right of the frame, the iterator and the keyword *reject* are depicted, each inside a dashed box. The evaluation of the reject expression results in a set of type *T*.

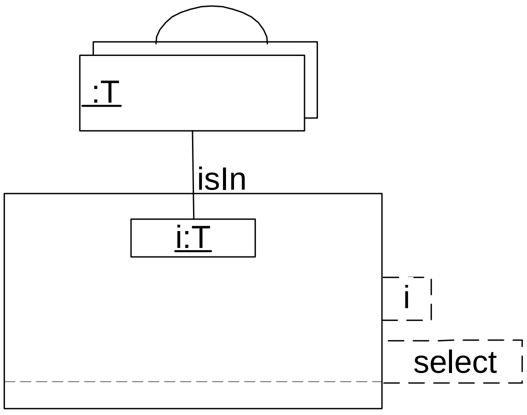
- Set(T)->collectNested(expression: OclExpression)->Bag(T2)



Operation *collectNested* is visualized as the operation *collect*. Applyin *collectNested* to a set results in a bag.

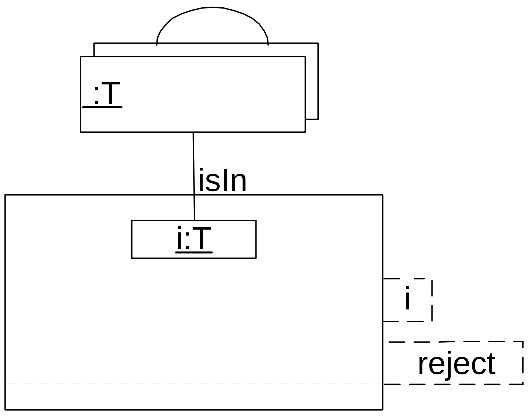
### 3.5.3 Bag

- Bag(T)->select(expression: OclExpression):Bag(T)



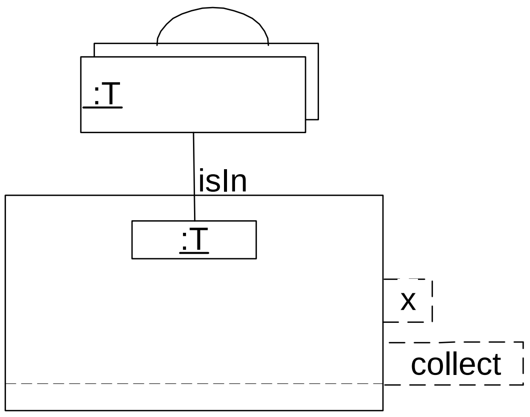
Applying operation *select* to a bag returns a subbag with elements having the specified properties. These selecting properties are visualized in the *select* frame. On the right of the frame, the iterator and the keyword *select* are depicted, each inside a dashed box. The evaluation of the select expression results in a bag of type *T*.

- Bag(T)->reject(expression: OclExpression):Bag(T)



Applying operation *reject* to a bag returns a subbag with elements not having the specified properties. These rejecting properties are visualized in the *reject* frame. On the right of the frame, the iterator and the keyword *reject* are depicted, each inside a dashed box. The evaluation of the reject expression results in a bag of type *T*.

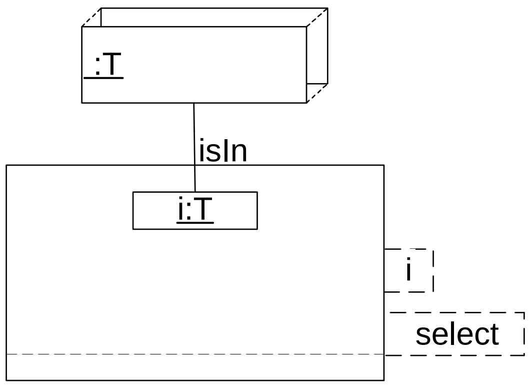
- Bag(T)->collectNested(expression: OclExpression):Bag(T2)



Operation *collectNested* is visualized as the operation *collect*. *collectNested* applied to a bag results in a bag.

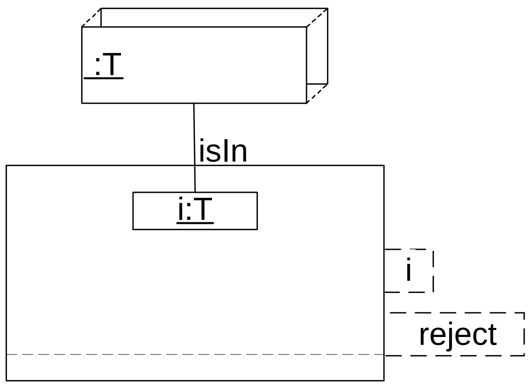
### 3.5.4 Sequence

- Sequence(T)->select(expression: OclExpression): Sequence(T)



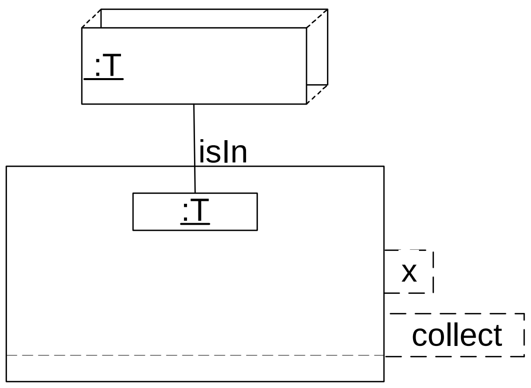
Applying operation *select* to a sequence returns a subsequence with elements having the specified properties. These selecting properties are visualized in the *select* frame. On the right of the frame, the iterator and the keyword *select* are depicted, each inside a dashed box. The evaluation of the select expression results in a sequence of type *T*.

- Sequence(T)->reject(expression: OclExpression): Sequence(T)



Applying operation *reject* to a sequence returns a subsequence with elements not having the specified properties. These rejecting properties are visualized in the *reject* frame. On the right of the frame, the iterator and the keyword *reject* are depicted, each inside a dashed box. The evaluation of the reject expression results in a sequence of type *T*.

- Sequence(T)->collectNested(expression: OclExpression): Sequence(T2)



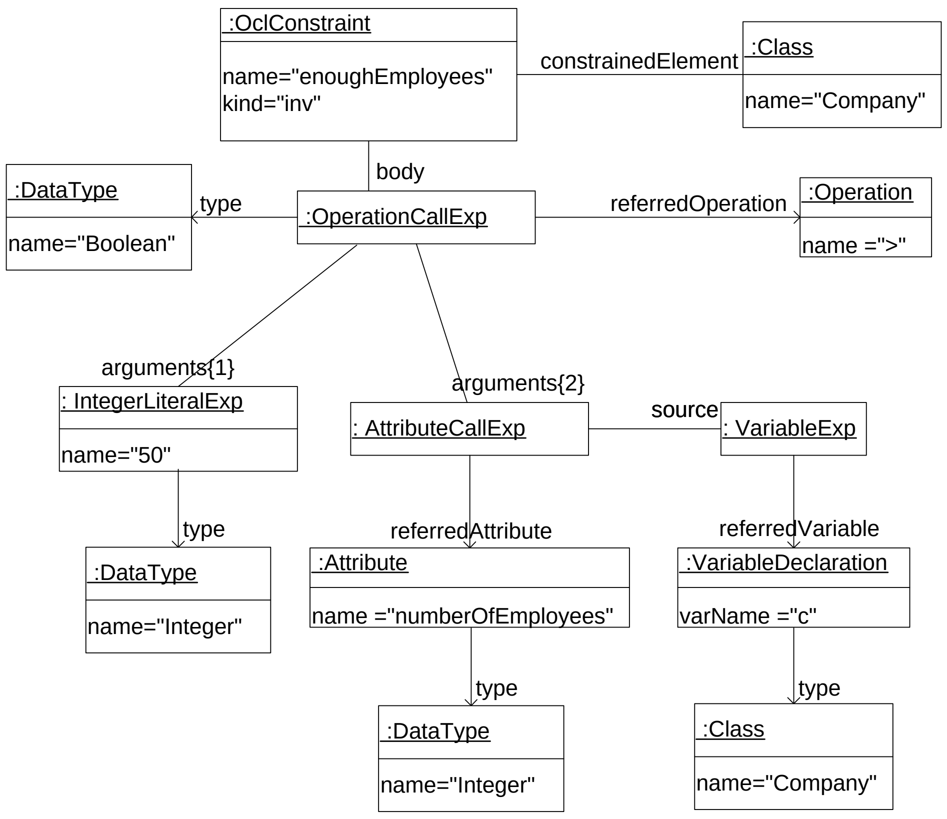
Operation *collectNested* is visualized as the operation *collect*. *collectNested* applied to a sequence results in a sequence.

# Appendix A

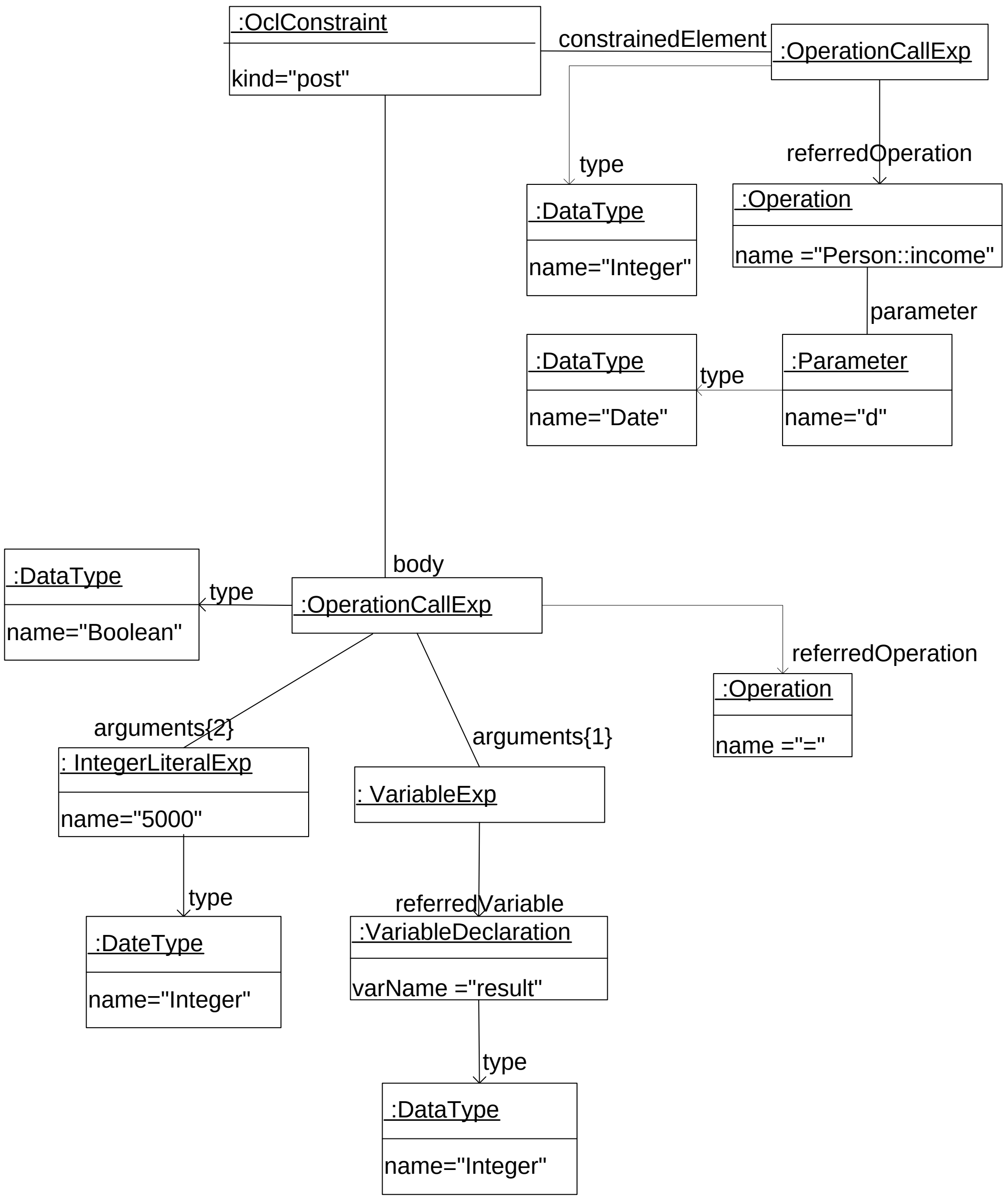
# Meta Model Instances

In this section, some meta model instances of constraints of Chapter 2 are represented. For better understanding some of them are changed a little. Therefore, the textual constraint is depicted above each meta model instance.
Each *OclExpression* has a type. It is partially left out in the following representations for reasons of clearness.

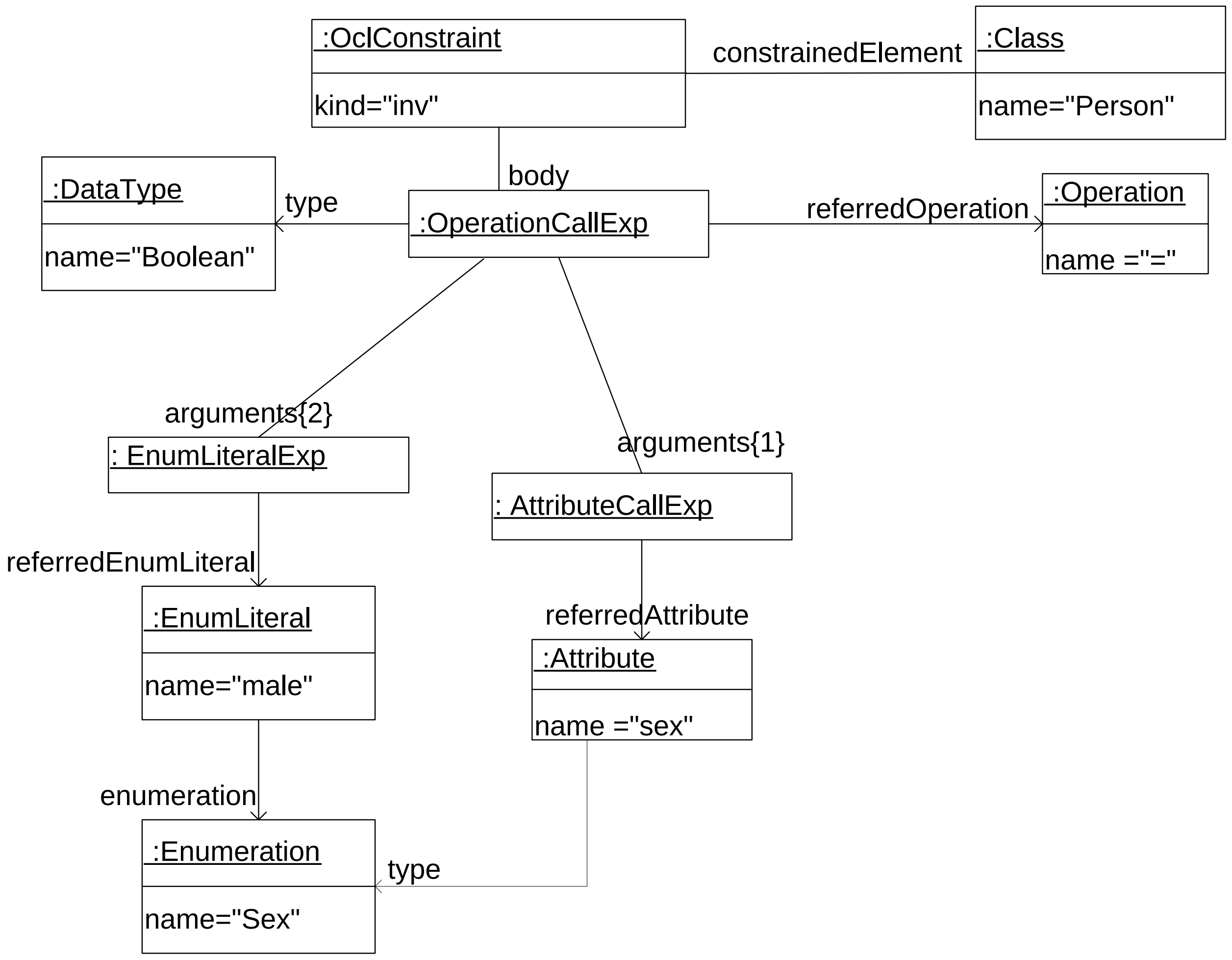
- `context c: Company inv enoughEmployees: c.numberOfEmployees > 50`



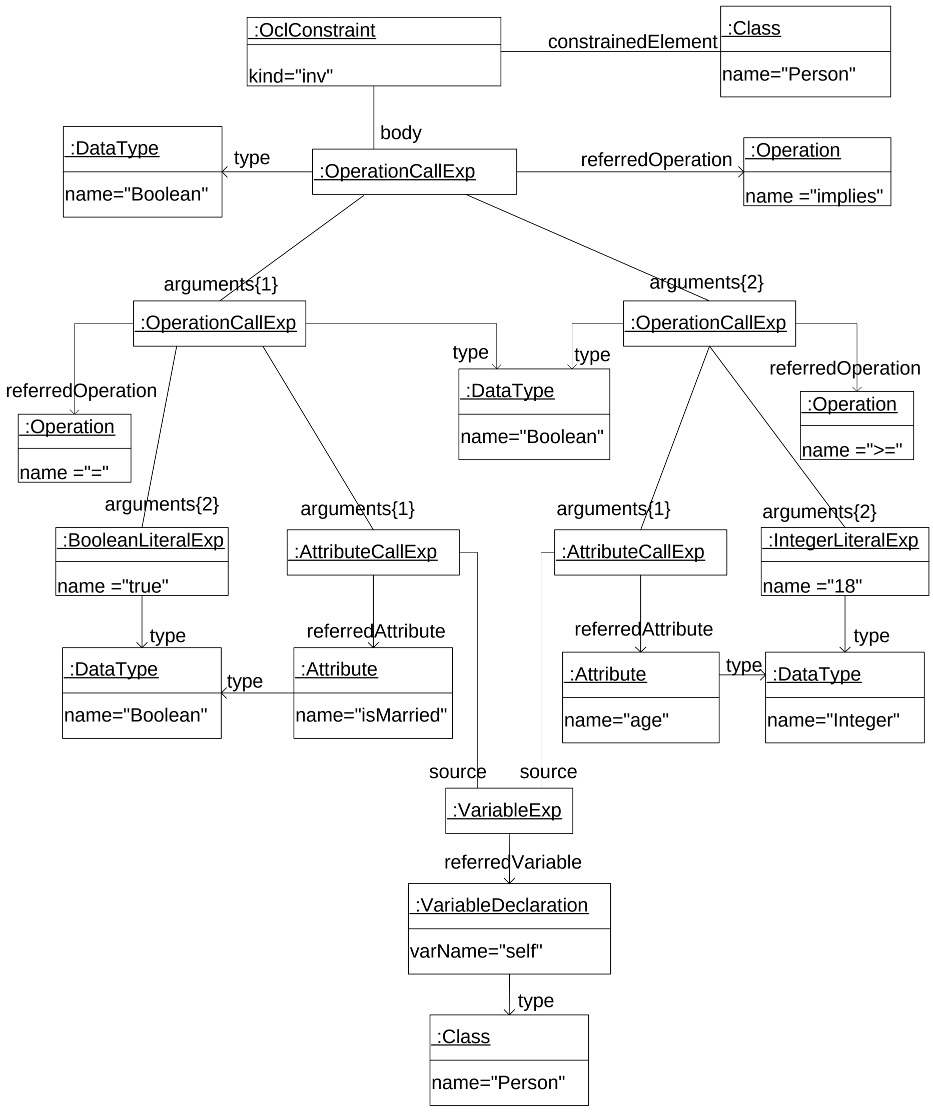
- `context Person::income(d:Date):Integer post: result = 5000`



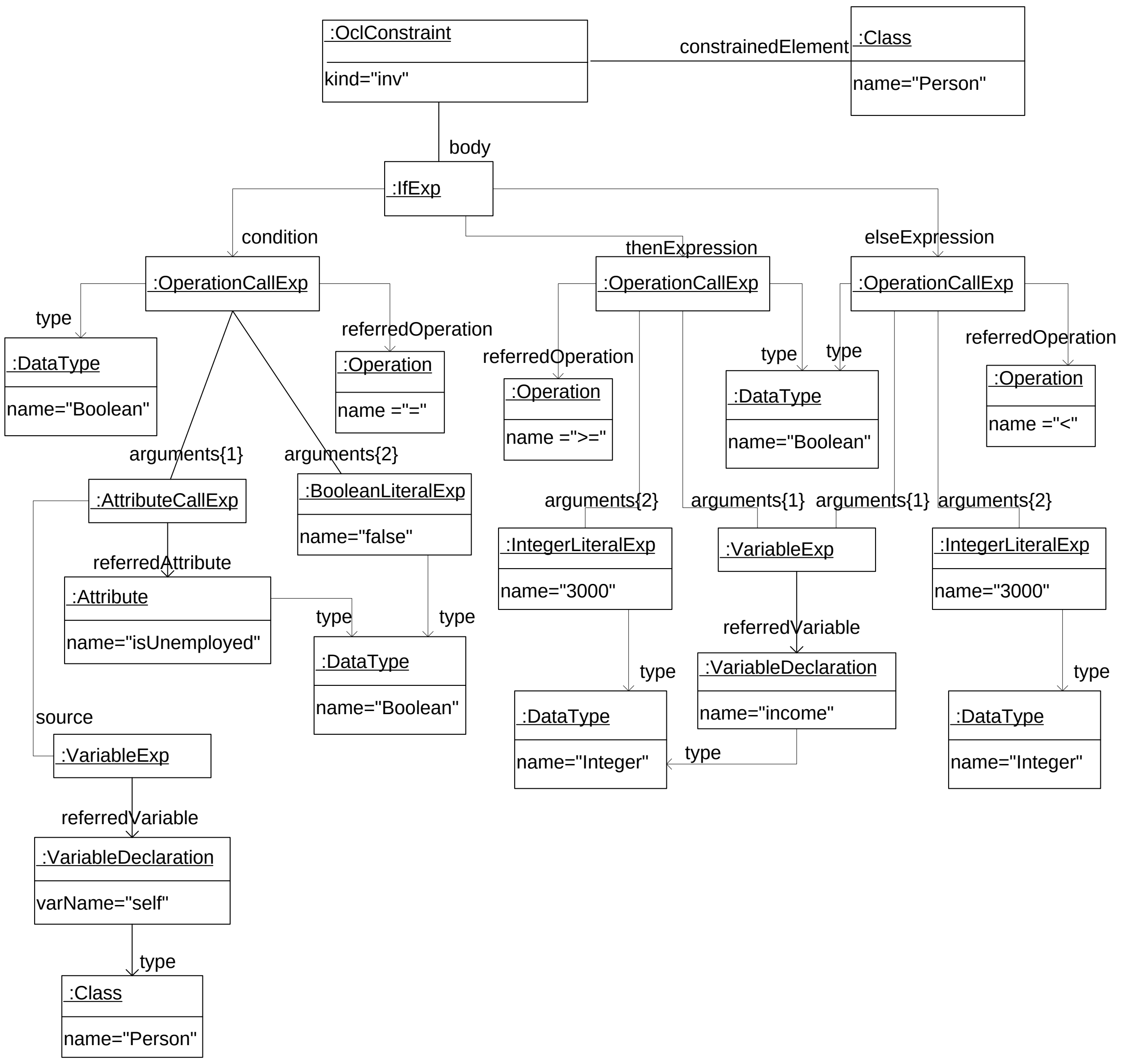
- `context Person inv: sex = Sex::male`



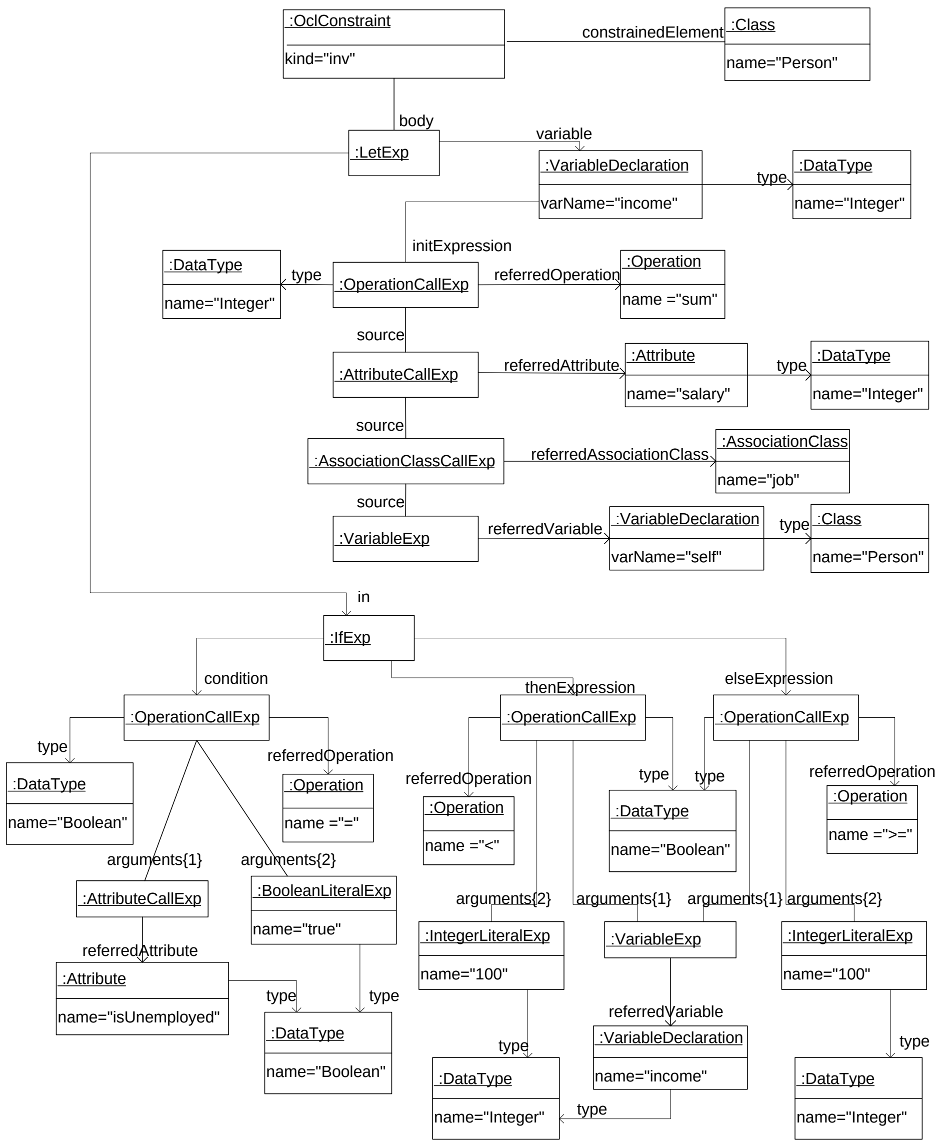
- `context Person inv:`
  `self.isMarried = true implies self.age >= 18`



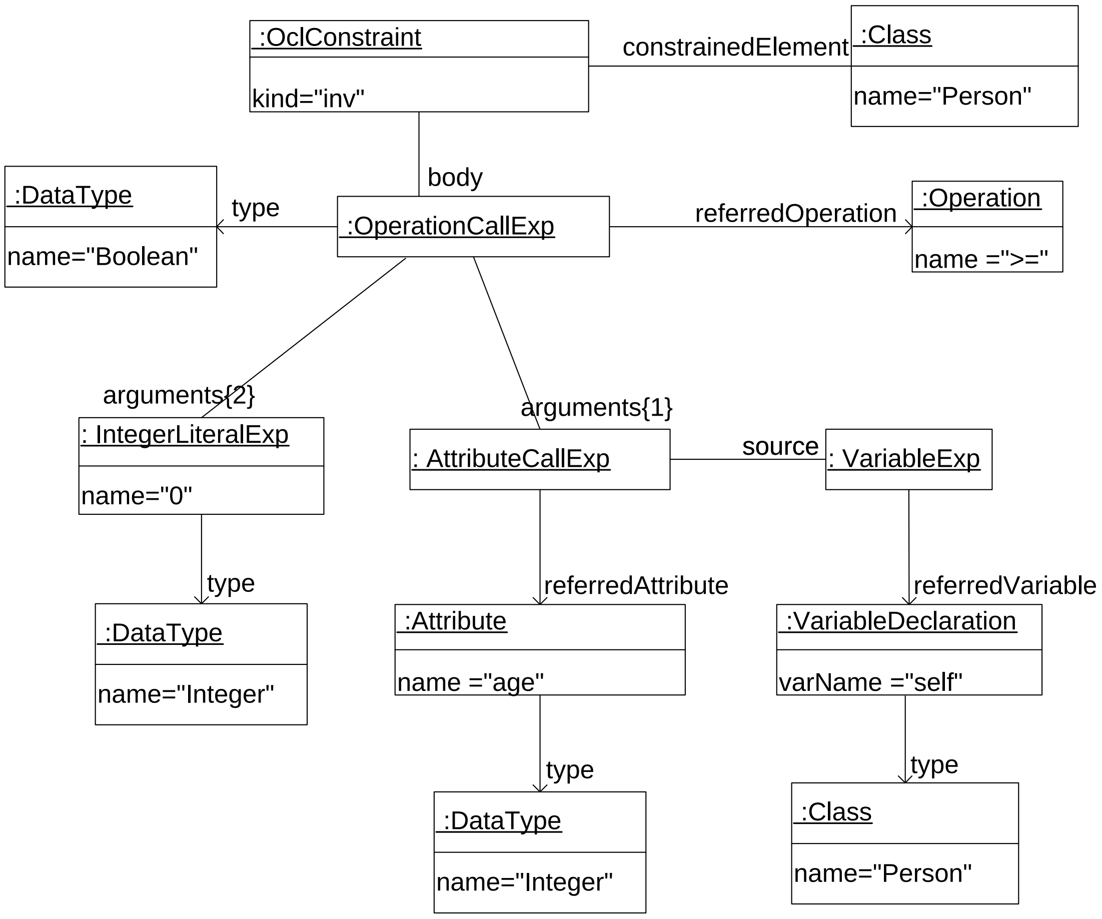
- `context Person inv:`
  `if (self.isUnemployed =false ) then income >= 3000 else income < 3000`



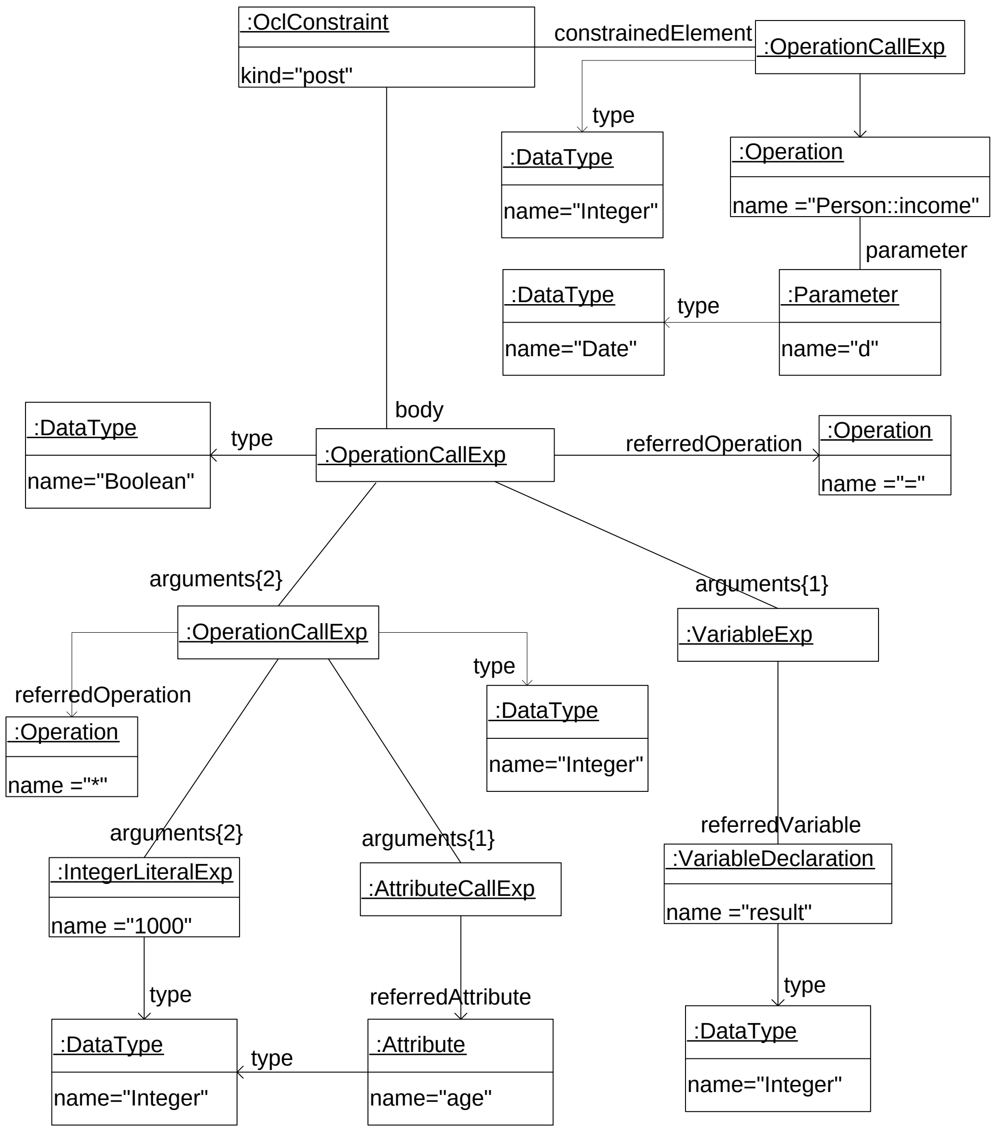
- `context Person inv: let income:Integer=self.job.salary->sum()`
  `in if isUnemployed then income < 100 else income > 100`



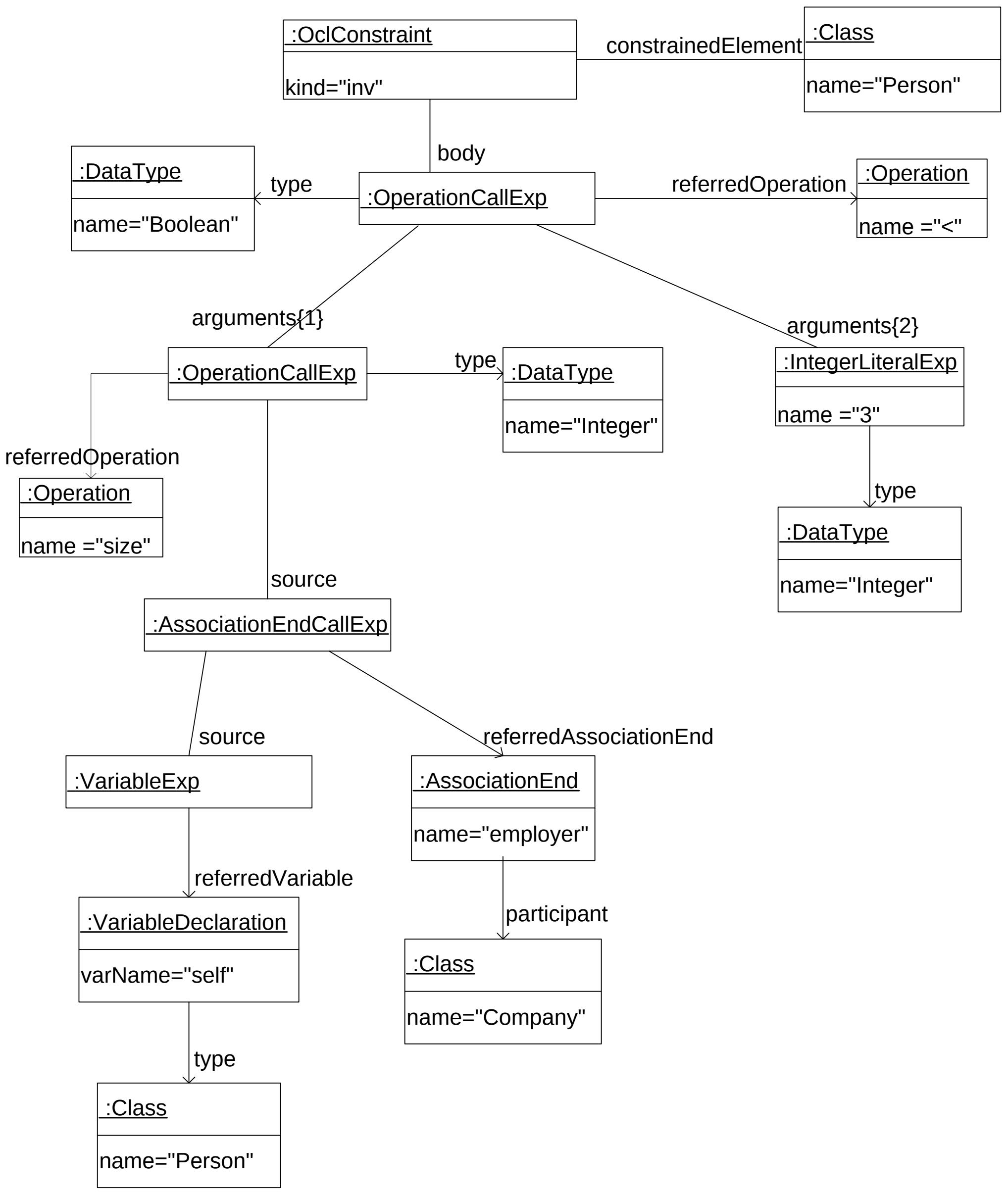
- `context Person inv: self.age >= 0`



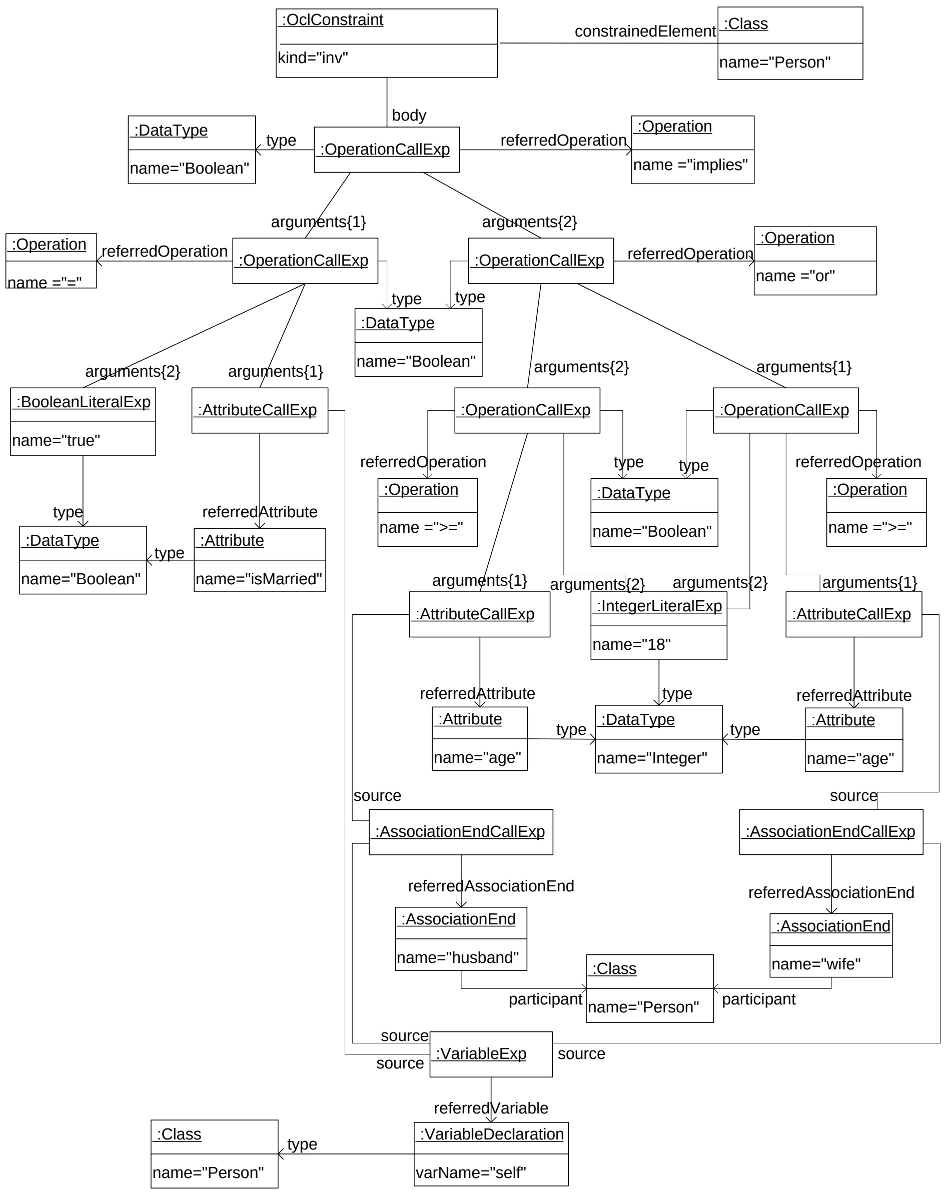
- `context Person::income(d:Date):Integer post:`
  `result = age * 1000`



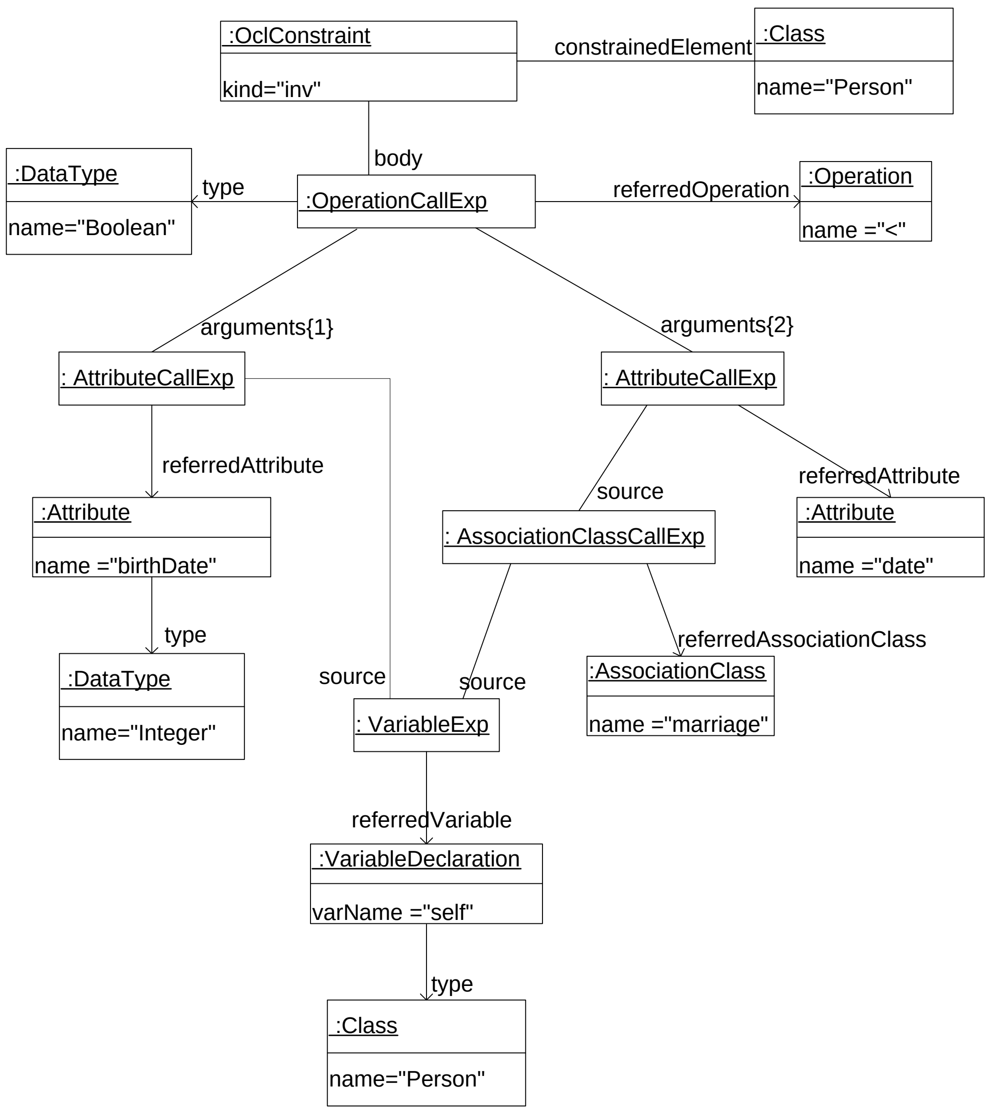
- `context Person inv: self.employer->size() < 3`



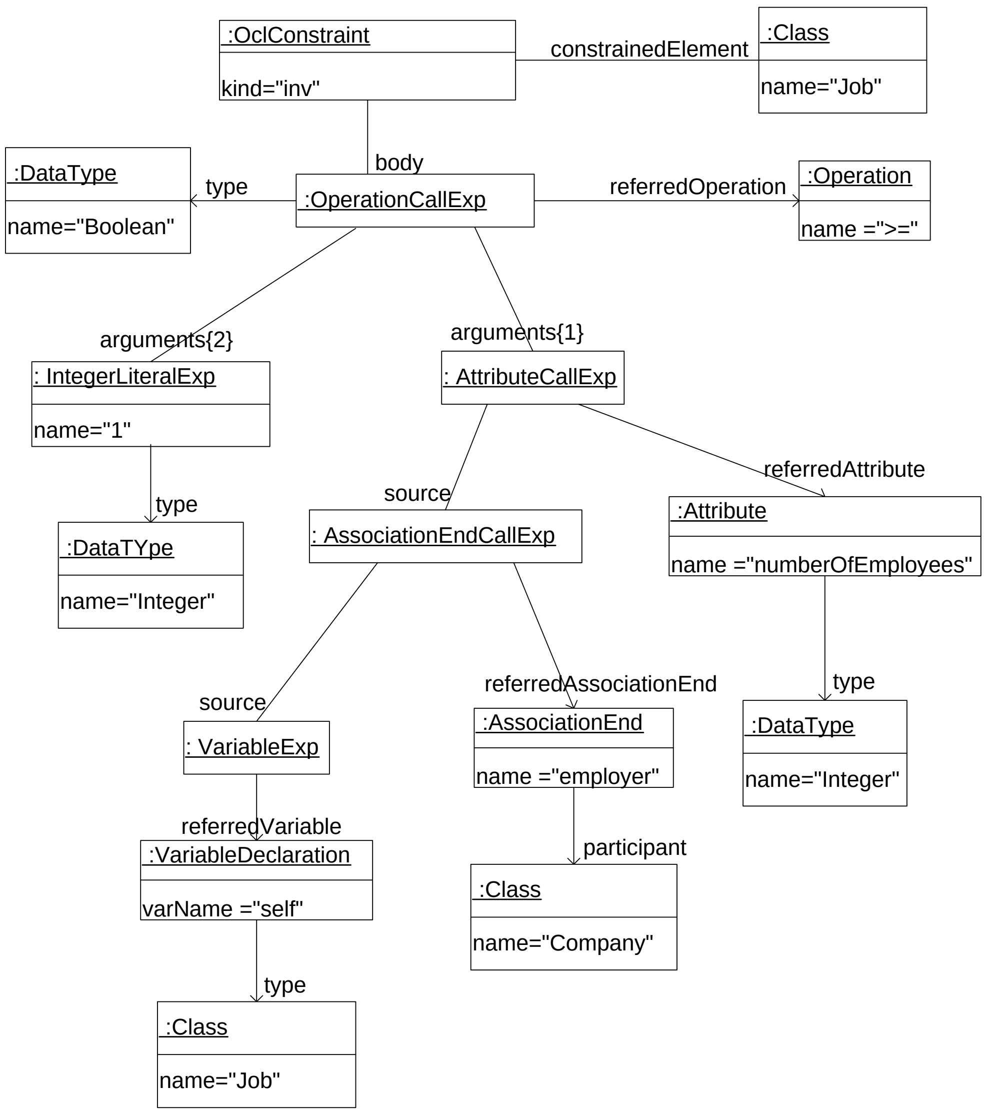
- `context Person inv: self.isMarried = true implies ((self.wife.age >= 18) or (self.husband.age >= 18))`



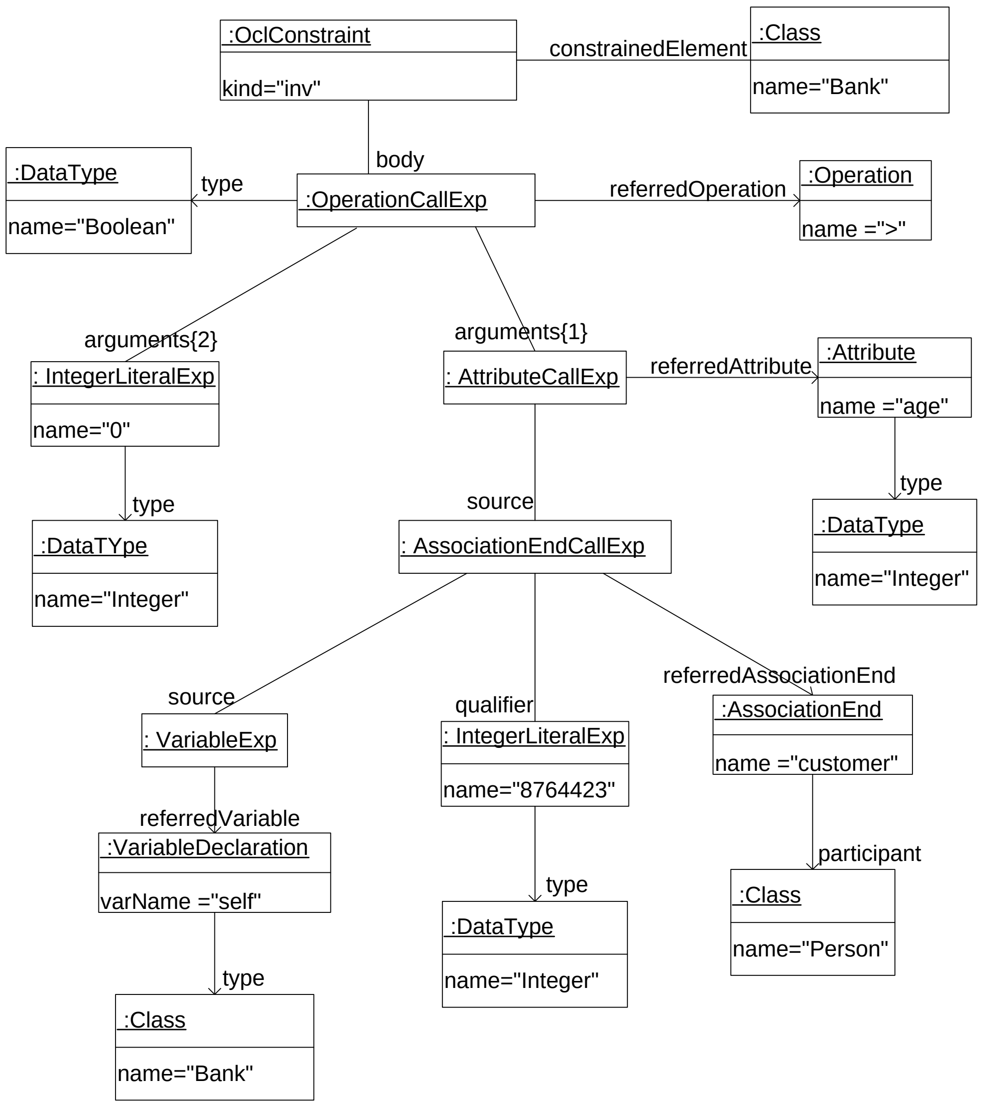
- `context Person inv: self.birthDate < self.marriage.date`



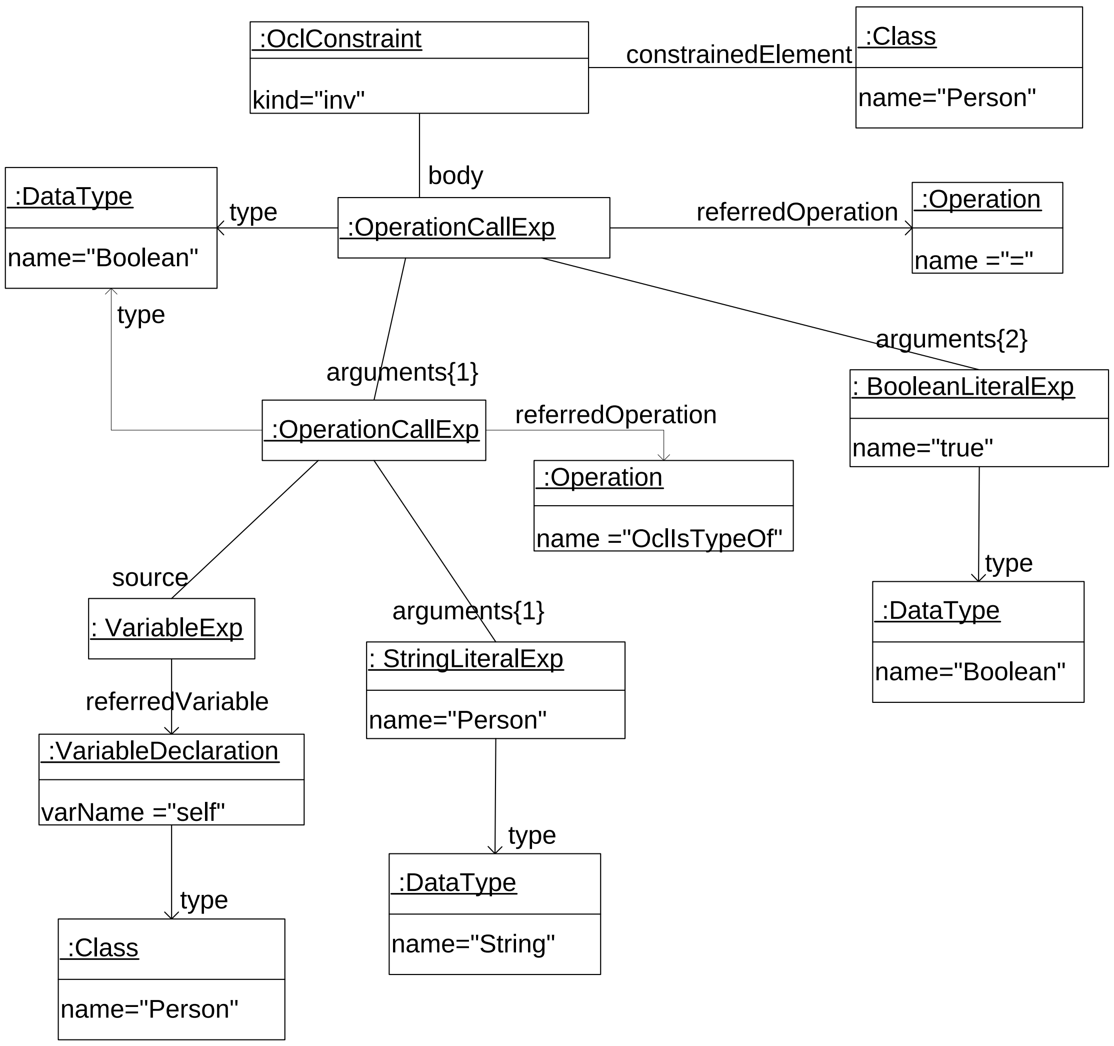
- `context Job inv: self.employer.numberOfEmployees >=1`



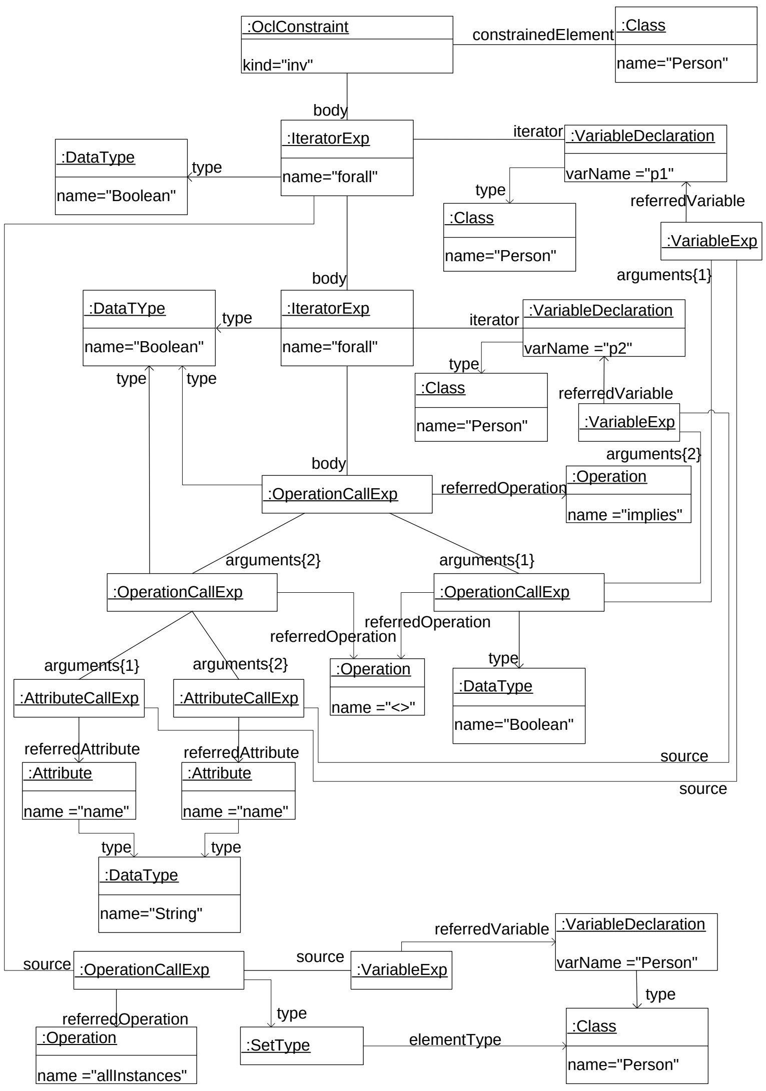
- `context Bank inv: self.customer[8764423].age > 0`



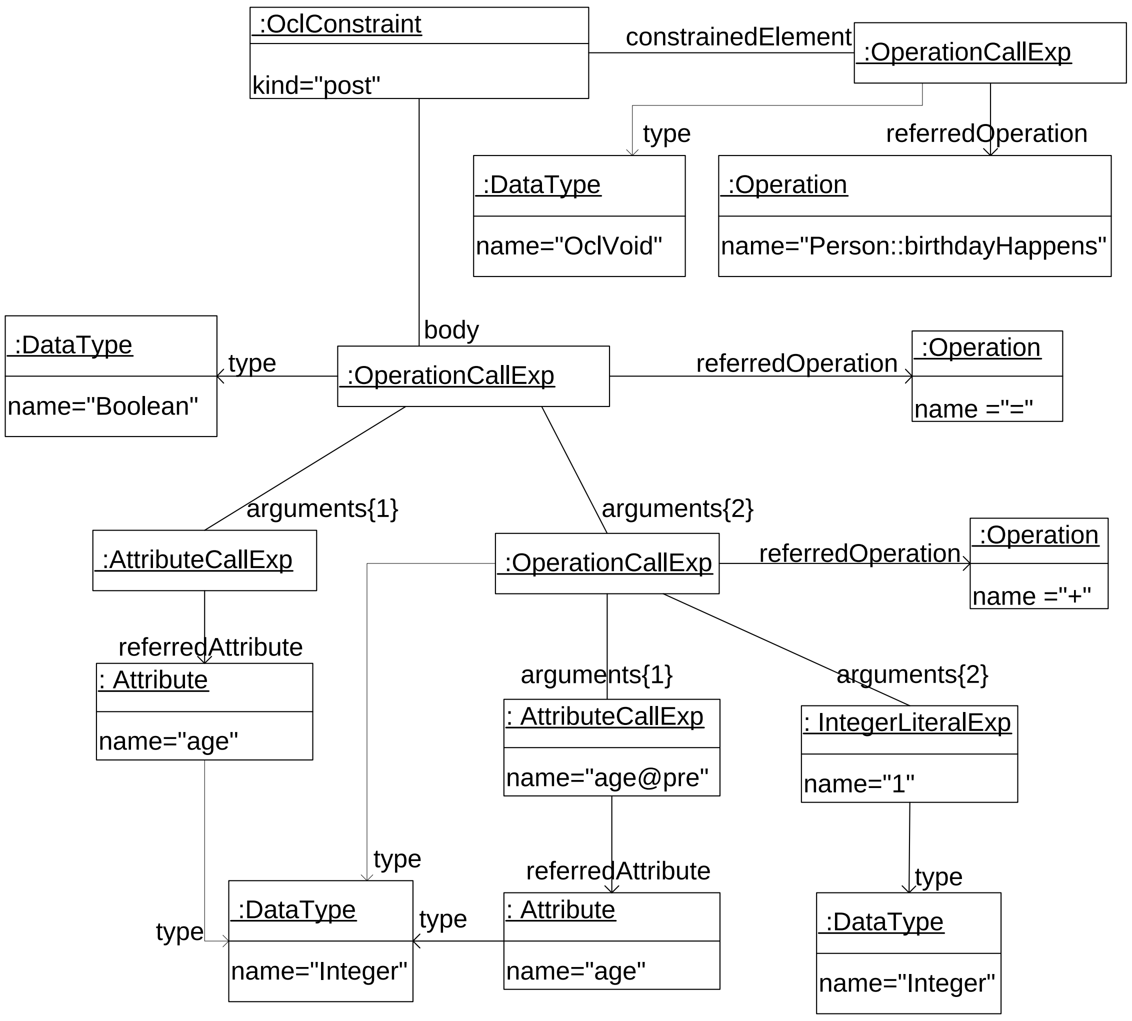
- `context Person inv: self.oclIsTpeOf(Person) = true`



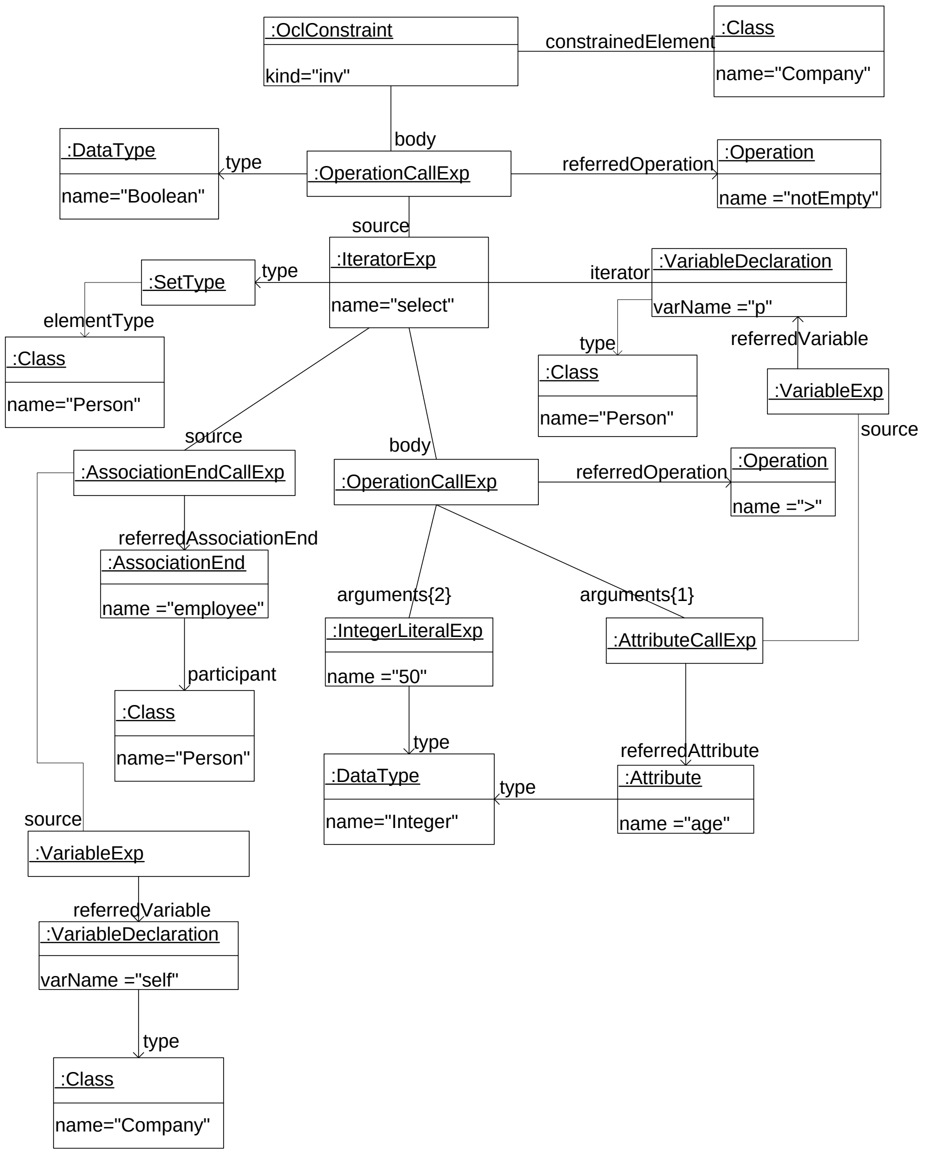
- `context Person inv:`
  `Person.allInstances()->forall(p1, p2 | p1<>p2 implies p1.name<>p2.name)`



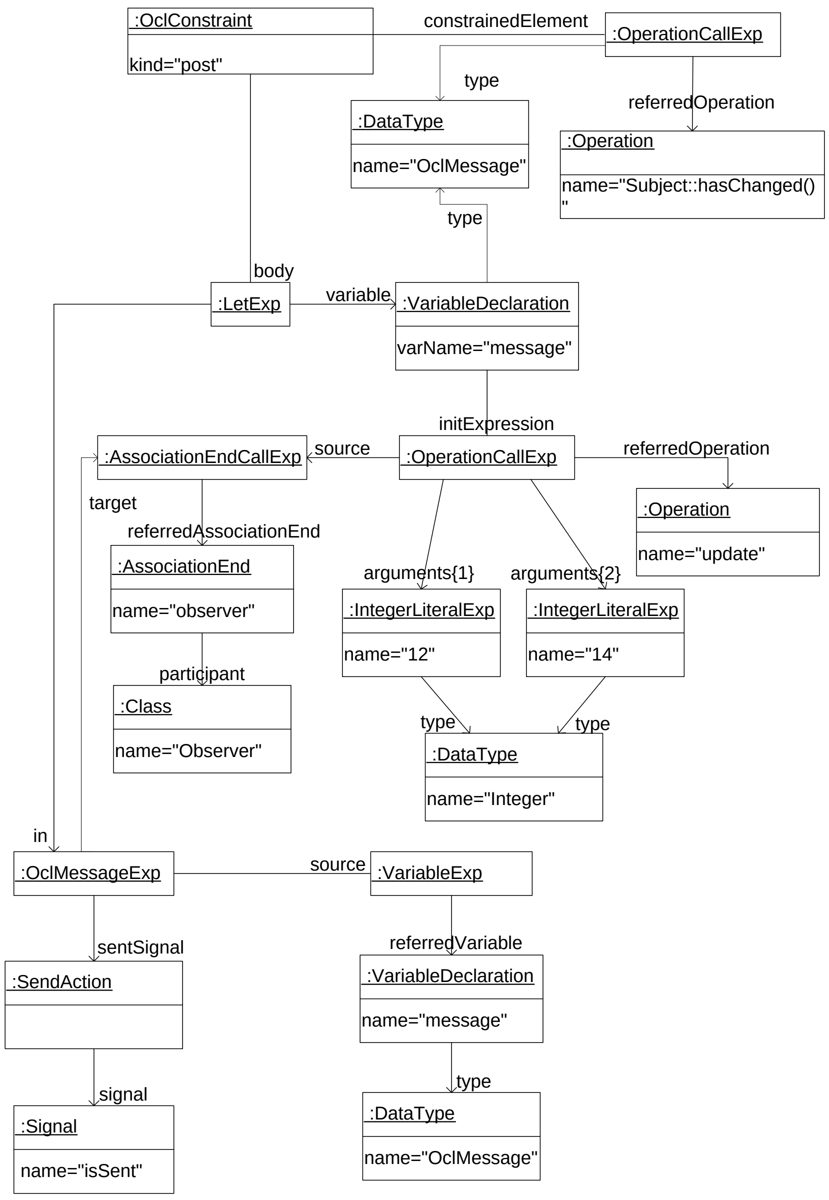
- `context Person::birthdayHappens() post:`
  `age = age@pre + 1`



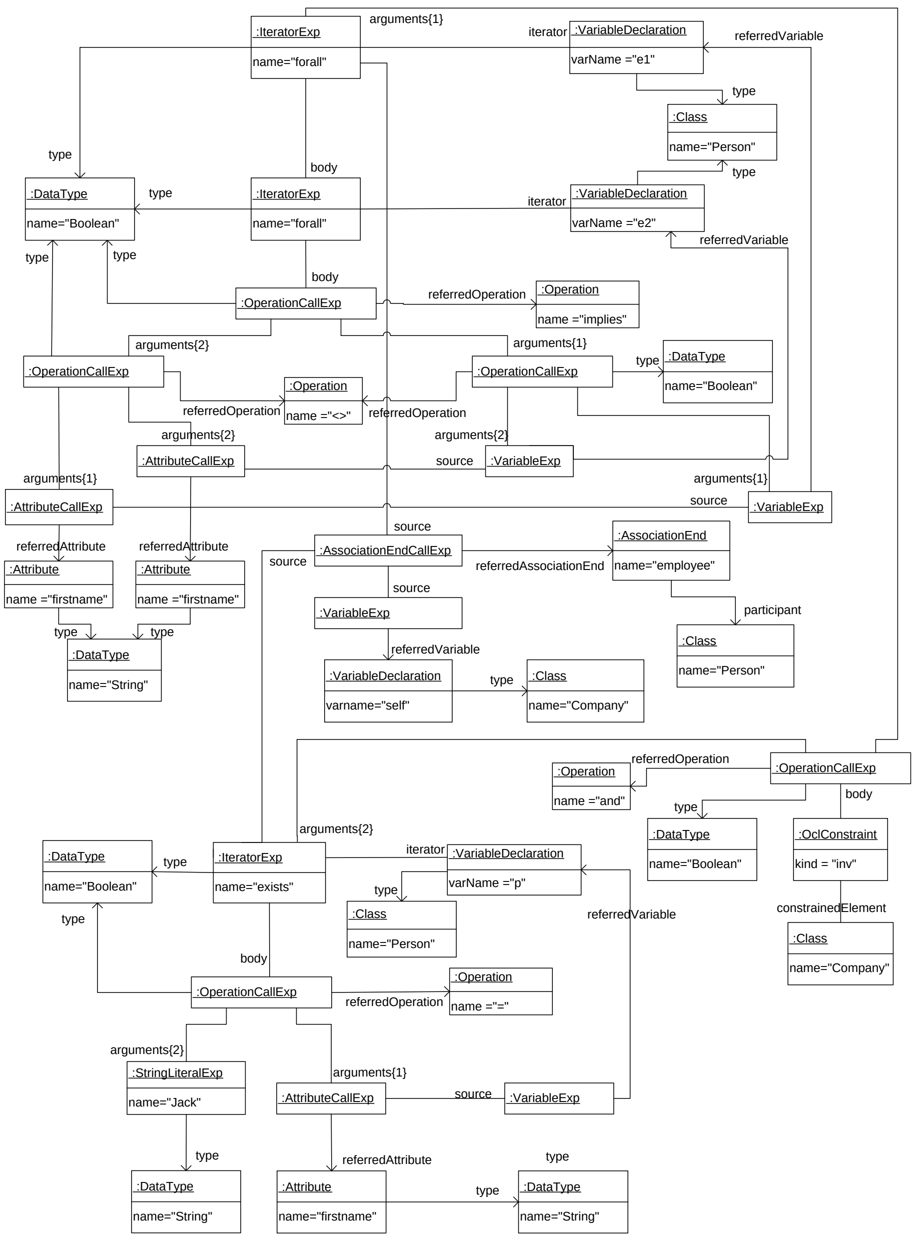
- `context Company inv:`
  `self.employee->select(p | p.age > 50)->notEmpty()`



- `context Subject::hasChanged() post:`
  `let message : OclMessage = observer^update(12, 14) in`
  `message.isSent()`



-
  ```
  context Company
  inv: self.employee->forall(e1, e2 | e1<>e2 implies e1.firstname<>e2.firstname)
  inv: self.employee->exists (p: Person | p.firstname = ´Jack´)
  ```



# Bibliography

[1] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A Visualization of OCL using Collaborations. In *UML 2001 – The Unified Modeling Language*, LNCS 2185, pages 257 – 271. Springer, 2001.

[2] *OCL 2.0 http://www.klasse.nl/ocl*, 2002. OMG.

[3] *Unified Modeling Language – version 1.4*, 2002. Available at `http://www.omg.org/technology/documents/formal/uml.htm`.