Technische Universität Berlin

Forschungsberichte
der Fakultät IV -
Elektrotechnik und Informatik

Visual OCL: Eine visuelle Notation der Object Constraint Language

Christiane Kiesner, Gabriele Taentzer, Jessica Winkelmann

Bericht–Nr. 2002–23

ISSN 1436 – 9915

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

Inhaltsverzeichnis

1 Überblick **3**

1.1 Kurzbeschreibung von OCL . . . 3
1.2 Kurzbeschreibung von VOCL . . . 4
1.3 Die wesentlichen Anforderungen an die konkrete und abstrakte Syntax . . . 4

2 VOCL Sprachbeschreibung **5**

2.1 Einführung . . . 5
2.1.1 Legende . . . 5
2.1.2 Klassendiagramm . . . 6
2.2 Relationen zum UML-Metamodell . . . 6
2.2.1 Self . . . 6
2.2.2 Spezifiaktion des UML Kontextes . . . 7
2.2.3 Invarianten . . . 8
2.2.4 Die Operationen id und isIn . . . 8
2.2.5 Vor- und Nachbedingungen von Operationen oder Methoden . . . 9
2.2.6 Basiswerte, Basistypen und Enumerationtypen . . . 9
2.2.7 Implies-Ausdrücke . . . 10
2.2.8 If-Then-Else-Ausdrücke . . . 11
2.2.9 Let-Ausdrücke und Definition Constraints . . . 12
2.2.10 Typübereinstimmungen, Re-Typing oder Casting . . . 16
2.3 Objekte und Eigenschaften . . . 16
2.3.1 Objekte . . . 16
2.3.2 Eigenschaften: Attribute . . . 16
2.3.3 Eigenschaften: Operationen . . . 17
2.3.4 Eigenschaften: Assoziationsenden und Navigation . . . 19
2.3.5 Navigation zu Assoziationsklassen . . . 25
2.3.6 Navigation von Assoziationsklassen aus . . . 26
2.3.7 Navigation entlang Assoziationen mit Qualifiern . . . 27
2.3.8 Verwenden von Eigenschaften von Supertypen . . . 28
2.3.9 Vordefinierte Eigenschaften . . . 29
2.3.10 Eigenschaften von Klassen . . . 30
2.3.11 Collections . . . 32
2.3.12 Werte vor der Ausführung einer Operation in der Nachbedingung . . . 34
2.4 Operationen auf Collections . . . 36
2.5 Messages . . . 43

2.6 Tupel . . . 45
2.7 Komposition von Constraints . . . 50

3 Die OCL Standardbibliothek 52
3.1 OclAny und OclVoid . . . 52
3.1.1 OclAny . . . 52
3.1.2 OclMessage . . . 53
3.1.3 OclVoid . . . 54
3.2 ModelElement Typen . . . 54
3.2.1 OclModelElement . . . 54
3.2.2 OclType . . . 55
3.2.3 OclState . . . 56
3.3 Einfache Typen . . . 57
3.3.1 Real, Integer und String . . . 57
3.3.2 Boolean . . . 57
3.4 Generelle Operationen auf Collections . . . 58
3.4.1 Collection . . . 58
3.4.2 Set . . . 61
3.4.3 Bag . . . 65
3.4.4 Sequence . . . 69
3.5 Vordefinierte Ocl-Iteratorbibliothek . . . 73
3.5.1 Collection . . . 73
3.5.2 Set . . . 77
3.5.3 Bag . . . 78
3.5.4 Sequence . . . 80

A Metamodell–Beispiele 82

Kapitel 1

Überblick

Die Object Constraint Language (OCL) ist eine logische Sprache, die als Erweiterung der Unified Modeling Language (UML) [3] eingeführt wurde. Sie ermöglicht es, Invarianten, Vor- und Nachbedingungen für Methodenaufrufe und Zustandsübergänge, etc. zu formulieren. Leider ist die OCL, im Gegensatz zur UML, eine textuelle Sprache und bietet daher eine vollkommen andere Notation von Modelelementen als die UML an. Dieser Text führt in eine Visualisierung von OCL ein, die so weit wie möglich die UML-Notation adaptiert. Diese visuelle Notation von OCL ist als Alternative für die textuelle entwickelt worden. D.h. beide Notationen basieren auf demselben OCL-Metamodell [2], das Teil der UML Version 2.0 sein wird.

Das Kapitel 2 beschreibt anhand zahlreicher Beispiele die Visualisierung von OCL (kurz VOCL). Das Kapitel 3 (ab Seite 52) enthält die visualisierte OCL-Standardbibliothek, d.h. dieses Kapitel stellt für jede Operation deren visuelle Variante vor. Im Anhang werden Metamodellinstanzen zu einigen Constraints aus dem 2. Kapitel gezeigt.

Diese Arbeit entstand im Rahmen des Projektes "Visuelle Sprachen" , das im Sommersemester 2002 an der Technischen Universität Berlin unter der Leitung von Gabriele Taentzer statt fand. Die konzeptionelle Basis zu dieser Sprachbeschreibung wurde von Bottoni, Koch, Parisi-Presicce und Taentzer in [1] gelegt.

1.1 Kurzbeschreibung von OCL

OCL ist eine formale Sprache zur Spezifikation von Invarianten für Klassen und Typen im Klassenmodell oder von Typinvarianten für Stereotypen, zur Beschreibung von Vor- und Nachbedingungen für Operationen, zur Beschreibung von Guards oder für Aussagen über ein UML-Modell. OCL ist die Abkürzung für Object Constraint Language und dient als Erweiterung der Sprache UML. UML selbst hat keine Sprachelemente für die Formulierung von solchen Bedingungen und kann diese nur unzureichend darstellen, z.B. durch natürlichsprachliche Randnotizen, die aber nicht zwingend eingehalten werden müssen. Natürlichsprachliche Texte haben den Nachteil, dass sie meistens mehrdeutig und daher nicht von einer Maschine interpretierbar sind. OCL behebt diesen Mangel durch eine textuelle Syntax, die die gewünschten Constraints adäquat umsetzt. OCL ist objektorientiert und streng typisiert: Für jedes Objekt muss der Typ (seine Klasse) explizit angegeben werden. Dadurch lassen sich die gestellten Bedingungen z.B. durch einen Parser und Constraint Checker einfacher verifizieren. Die Sprache verfügt außerdem über ein Metamodell, das über die abstrakte Syntax von OCL Auskunft

gibt; damit verbunden zeigt sich das theoretische Fundament, auf dem OCL basiert. Das Metamodell ist durch Klassendiagramme formuliert. Darüberhinaus enthält es sogenannte Wohlgeformtheitsregeln, die die Menge von möglichen Ausdrücken in zulässige und unzulässige Notationen trennen. OCL hat den Nachteil, dass die direkte Vereinigung mit der grafischen UML-Welt schwierig herzustellen ist: Der Anwender muss eine weitere Sprache lernen, die die UML-Modelelemente anders darstellt.

1.2 Kurzbeschreibung von VOCL

Visual OCL (VOCL) ist eine grafische Repräsentation der oben beschriebenen Sprache OCL und versucht, den beschriebenen Nachteil der rein textuellen Darstellung von Bedingungen zu beseitigen. Ausgehend vom OCL-Metamodell soll VOCL stark an die UML-Notation angelehnt sein und deren grafische Repräsentation nutzen. Es ergeben sich dadurch Vereinfachungen und ggf. eine direkte Einbindung in UML-Diagramme. Als OCL-Ableger ist auch VOCL eine formale, streng typisierte und objekt-orientierte Sprache und bietet gegenüber OCL den Vorteil, dass der Anwender nicht eine weitere textuelle Sprache erlernen muss.

1.3 Die wesentlichen Anforderungen an die konkrete und abstrakte Syntax

Die abstrakte Syntax von OCL orientiert sich am OCL-Metamodell. Da VOCL dieselbe Sprache beschreiben soll, muss auch VOCL diesem Metamodell genügen. In diesem Zusammenhang wird gefordert, dass eine Transformation von OCL-Text in VOCL-Diagramme und umgekehrt über das Metamodell realisiert werden kann.

Die Visualisierung von OCL soll soweit wie möglich an die von UML angelehnt sein. Wo neue Visualisierungen nötig sind, folgen wir den Empfehlungen des UML Standards und vermeiden z.B. Farben und spezielle Schriftarten, um semantische Bedeutungen auszudrücken. Desweiteren bieten wir Möglichkeiten an, die Grösse eines Diagramms dadurch zu beschränken, dass Unterbedingungen in eigenen Diagrammen formuliert werden können.

Kapitel 2

VOCL Sprachbeschreibung

Dieses Kapitel führt anhand von Beispielen die VOCL, eine visuelle Sprache zur Beschreibung von OCL-Modellen ein. Das Kapitel 3 (Seite 52) enthält die Standardbibliothek, in der die gesamten Operationen visuell dargestellt werden. Aus diesem Grunde werden hier nicht alle Standardoperationen an einem Beispiel visualisiert.

2.1 Einführung

2.1.1 Legende

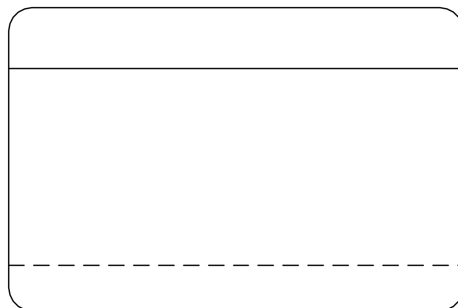


Abbildung 2.1: prinzipielle Darstellung eines Constraints

Ein OCL Constraint wird in einem abgerundeten Rechteck mit zwei Abschnitten, dem Kontextteil und dem Body, der einen Bedingungsteil enthalten kann, dargestellt.
Der Kontextteil(oben) beinhaltet das Schlüsselwort *context* gefolgt vom Typnamen der Klasse, auf die sich der Constraint bezieht, gefolgt vom Typ des Constraints, z.B. *inv*, *pre*, *post* oder *def*.
Im Body wird der Body des Constraints visualisiert.
Im Bedingungsteil können mit Hilfe von im Body definierten Variablen Bedingungen des Constraints angegeben werden. Wenn es einen Bedingungsteil gibt, wird er durch eine gestrichelte Linie vom Rest des Bodys getrennt.

2.1.2 Klassendiagramm

Um die Visualisierung von OCL deutlich zu machen, werden in den folgenden Kapiteln eine Anzahl von Constraints dargestellt. Diese Beispiele beziehen sich auf ein Klassendiagramm, das in der folgenden Abbildung zu sehen ist.

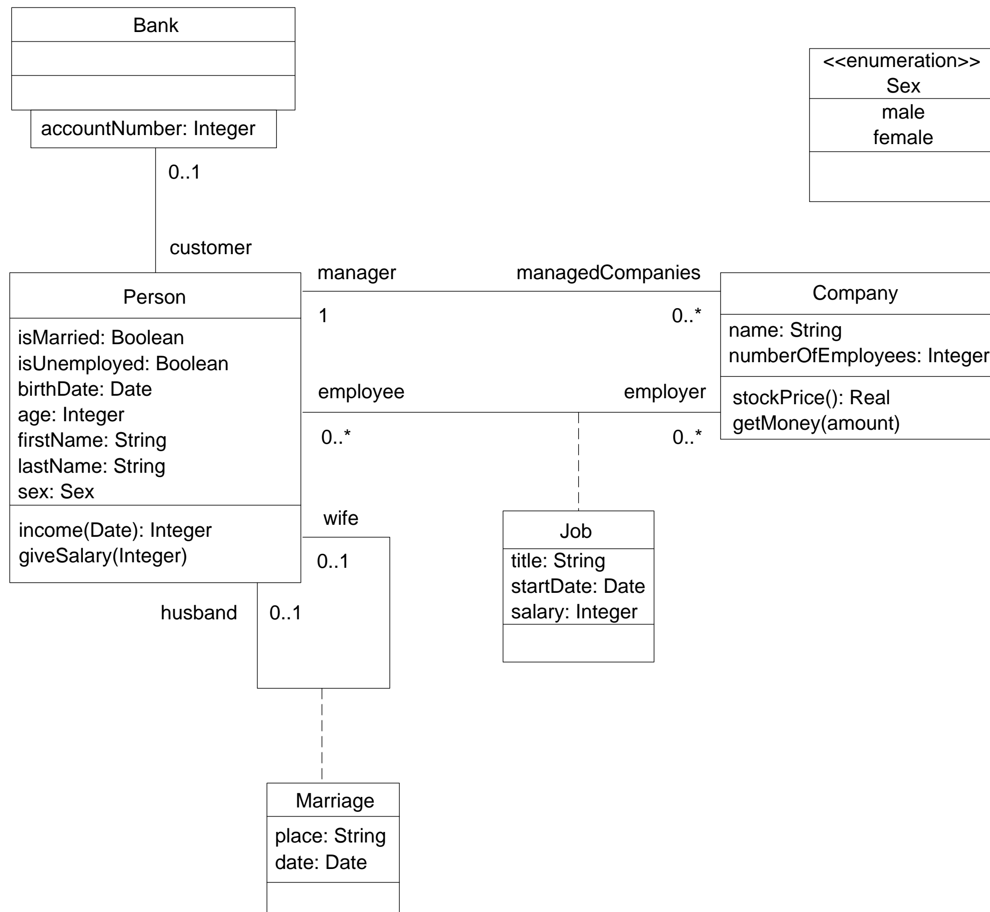


Abbildung 2.2: Klassendiagramm

2.2 Relationen zum UML-Metamodell

2.2.1 Self

Die Variable *self* wird wie in OCL verwendet und ist immer eine Instanz vom Typ des Kontextes. Bei dieser Instanz beginnt der Constraint.
Wenn es eindeutig ist, an welcher Stelle der Constraint beginnt, kann *self* auch weggelassen werden. Das ist genau dann der Fall, wenn es nur eine Instanz vom Typ des Kontextes gibt.

2.2.2 Spezifiaktion des UML Kontextes

Der Kontext kann wie in OCL definiert werden.

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

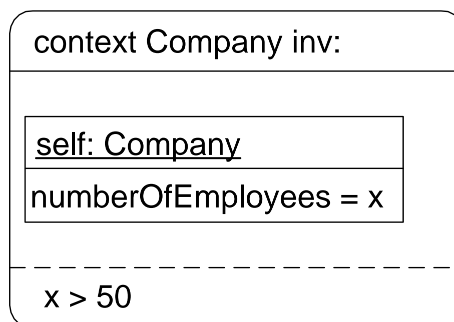


Abbildung 2.3: Kontextdefinition

Dieser Constraint spezifiziert, dass eine Firma mehr als 50 Mitarbeiter hat.
Der Kontext kann auch mit einem Namen definiert werden:

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

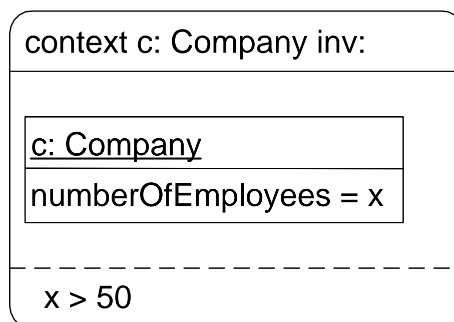


Abbildung 2.4: Constraint mit Kontextname

Im Body wird dann dieser Name anstelle von *self* verwendet, in diesem Fall beginnt der Constraint bei diesem Objekt.

Einem Constraint kann ein Name gegeben werden, dieser wird im Kontextteil hinter der Angabe des Kontextes notiert.

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

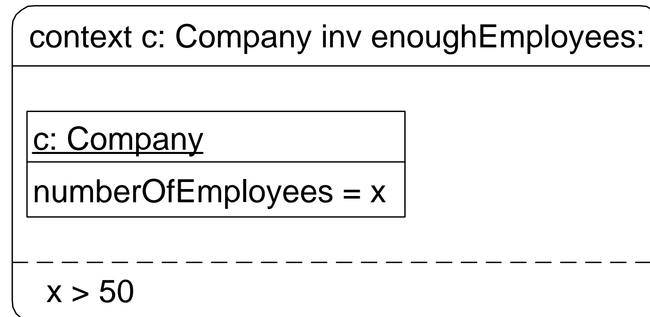


Abbildung 2.5: Constraint mit einem Namen

Abstrakte Syntax

Die abstrakte Syntax eines Constraints ist ein *OclConstraint*, dessen Attribute *name* und *kind* dem Namen und Typ des Constraints entsprechen. (In Abbildung 2.5 ist *name*=”enoughEmployees” und *kind*=”inv”.) Jeder Constraint hat einen *body*, der eine *OclExpression* ist, und ein *constraintElement*, das ein *Classifier* mit dem Namen des Constraint-Kontextes ist. (In Abbildung 2.5 ist das *constraint-Element* ein *Classifier* mit dem Namen ”Person” und der *body* ist eine *OperationCallExp*, die auf einen *type* und eine *referredOperation* verweist, und ein Argument hat. Der *type* ist der Rückgabetyp der Operation (im Beispiel also ”Boolean”), der Name der *referredOperation* ist der Operationsname(im Beispiel ”>”) und die Argumente sind *OclExpressions*). Eine ausführliche Darstellung der abstrakten Syntax des Constraints aus Abbildung 2.5 ist im Metamodell auf Seite 82 zu sehen.

2.2.3 Invarianten

In den vorangegangenen Beispielen war der Typ des Constraints *inv*, also beschreiben diese Constraints Invarianten. Weitere Typen sind *pre* und *post*, welche Vor- und Nachbedingungen von Operationen spezifizieren, sowie *def*, wodurch Definitionen von Variablen oder Operationen eingeführt werden.

2.2.4 Die Operationen id und isIn

Die Operation *id* beschreibt die Identität zweier Objekte und ist eine Hilfsoperation, die es in VOCL, aber nicht in OCL gibt. Sie vereinfacht die Visualisierung gleicher Instanzen von Objekten. Die Darstellung erfolgt mit einem Link, der den Ausdruck *id* enthält.
Ein Beispiel hierfür ist in Abbildung 2.8 auf Seite 11 zu sehen.
Zwei Instanzen sind auch identisch, wenn sie den gleichen Namen haben. Siehe *self:Person* in Abbildung 2.10 auf Seite 14.
Die Operation *isIn* wird auf Collections angewendet und liefert true, wenn das Objekt in der Collection enthalten ist.Auch diese Operation gibt es nicht in OCL.
Hierzu ist ein Beispiel in Abbildung 2.32 auf Seite 31 zu sehen. *p1* und *p2* sind in der Collection *Person* enthalten. Das wird durch einen Link mit dem Ausdruck *isIn* dargestellt.

2.2.5 Vor- und Nachbedingungen von Operationen oder Methoden

Wenn Vor- oder Nachbedingungen einer Methode oder Operation visualisiert werden, ist *self* eine Instanz des Typen, der die Methode oder Operation zur Verfügung stellt. Ein Methodenaufruf wird wie ein Methodenaufruf in einem Kollaborationsdiagramm visualisiert.
Hat die Operation einen Rückgabetyp, der kein primitiver Datentyp ist, z.B. eine Collection, dann wird dies durch einen Pfeil von der Instanz, auf die die Operation angewendet wird, zur Instanz, die zurückgeliefert wird, visualisiert.
Im folgenden Constraint wird die Nachbedingung der Operation *income* einer Person spezifiziert, der Rückgabewert der Operation vom Typ Integer ist 5000.

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

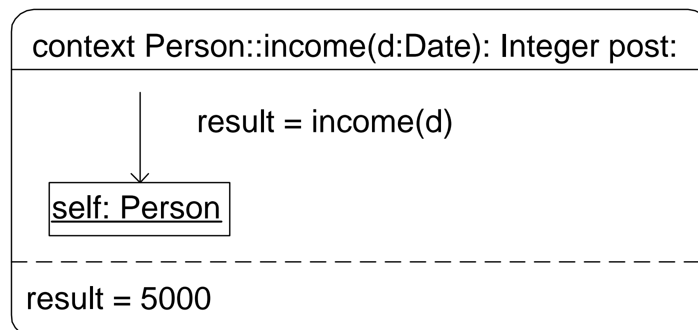


Abbildung 2.6: Nachbedingung einer Operation

result ist ein vordefinierter Wert, in dem der Rückgabewert der Operation steht, sofern es einen gibt. Von daher kann die Zuweisung an die Variable *result* am Operationspfeil auch weggelassen werden.

Abstrakte Syntax

Bei Nachbedingungen ist das Attribut *kind* des *OclConstraints* in der abstrakten Syntax "post". Der Aufruf einer Operation entspricht einer *OperationCallExp*. Eine ausführliche Darstellung der abstrakten Syntax des Constraints aus Abbildung 2.6 ist im Metamodell auf Seite 83 zu sehen.

2.2.6 Basiswerte, Basistypen und Enumerationtypen

Die Basistypen sind wie in OCL Boolean, Integer, Real und String. Die vordefinierten Operationen auf diesen Typen werden in der Standard-Bibliothek beschrieben.
Aufzählungstypen wie *male* oder *female* vom Datentyp *Sex* können folgendermaßen verwendet werden:

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

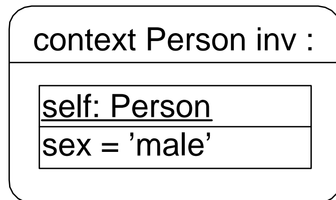


Abbildung 2.7: Aufzählunstypen

Der obige Constraint sagt aus, dass das Geschlecht einer Person männlich ist.

Abstrakte Syntax

Die Basistypen Boolean, Integer, Real und String werden abstrakt auf *BooleanLiteralExp*, *IntegerLiteralExp*, *RealLiteralExp* und *StringLiteralExp* abgebildet. Werden Elemente eines Aufzählungstyps in einem Constraint verwendet, ist deren abstrakte Syntax eine *EnumLiteralExp*, die ein *EnumLiteral* referenziert, welches auf eine *Enumeration* verweist. In Abbildung 2.7 ist der Name des *EnumLiterals* "male" und der Name der *Enumeration* "Sex". Eine ausführliche Darstellung der abstrakten Syntax des Constraints aus Abbildung 2.7 ist im Metamodell auf Seite 84 zu sehen.

2.2.7 Implies-Ausdrücke

Ein Implies-Ausdruck wird in einem Implies-Rahmen visualisiert. Alles über dem *implies* beschreibt die Bedingung, die, wenn sie erfüllt ist, den unteren Teil impliziert. Sowohl der obere als auch der untere Teil des *implies* können einen eigenen Bedingungsteil haben.

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

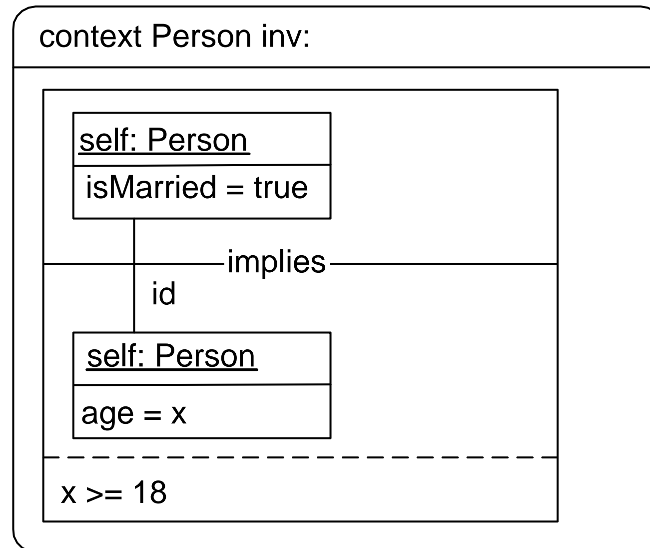


Abbildung 2.8: Implies-Constraint

In diesem Implies-Constraint wird ausgesagt: Wenn eine Person verheiratet ist, ist sie mindestens 18 Jahre alt. Dass es sich hier in den beiden Teilen des Implies-Constraint um dieselbe Person handelt, wird durch den Ausdruck *id* an dem Link verdeutlicht.

Abstrakte Syntax

In der abstrakten Syntax ist implies eine *Operation*. Die Verwendung eines implies-Ausdrucks entspricht also einer *OperationCallExp*, die referenzierte Operation hat den Namen "implies", der referenzierte Typ ist "Boolean". Die Argumente sind die *OclExpressions*, die über und unter dem implies visualisiert wurden. Eine ausführliche Darstellung der abstrakten Syntax des Constraints aus Abbildung 2.8 ist im Metamodell auf Seite 85 zu sehen.

2.2.8 If-Then-Else-Ausdrücke

Der If-Then-Else-Rahmen beinhaltet drei Teile, im If-Teil steht die if-Bedingung, im Then-Teil der then-Body und im Else-Teil der else-Body. Jeder der drei Teile kann einen Bedingungsteil enthalten, wie im Implies-Ausdruck.

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

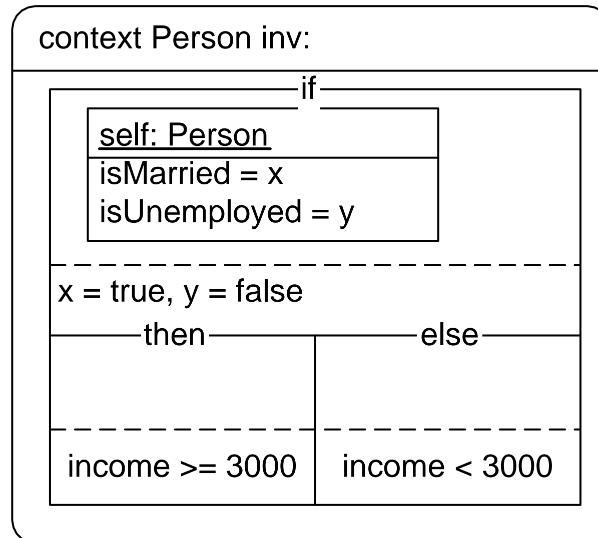


Abbildung 2.9: If-Then-Else-Constraint

Dieser If-Then-Else-Constraint spezifiziert: Wenn eine arbeitetende Person verheiratet ist, ist ihr Einkommen mindestens 3000, sonst kleiner als 3000.

Abstrakte Syntax

Die abstrakte Syntax eines If-Then-Else-Ausdrucks ist eine *IfExp*. Sie verweist auf eine *condition*, eine *thenExpression* und eine *elseExpression*, wenn es eine gibt. Diese drei sind *OclExpressions*. Eine ausführliche Darstellung der abstrakten Syntax des Constraints aus Abbildung 2.9 ist im Metamodell auf Seite 86 zu sehen. Hier wurde allerdings die zweite der beiden If-Bedingungen weggelassen.

2.2.9 Let-Ausdrücke und Definition Constraints

Ein Let-Ausdruck definiert eine Variable oder eine Operation, die danach in einem Constraint verwendet werden kann. Es gibt einen Let-Rahmen und einen In-Rahmen, im Let-Rahmen werden ein oder mehrere Variablen und Operationen definiert. Für eine Variable gibt es einen Rahmen, in dem oben links der Name der neu definierten Variablen steht und darunter die Definition des Wertes der Variablen. Für Operationen wird das analog durchgeführt. Falls mehrere Let-Ausdrücke dargestellt werden sollen, erhält jeder Ausdruck einen gesonderten Rahmen. Bei einem Ausdruck ist der Rahmen optional, kann also weggelassen werden.
Innerhalb des In-Rahmens steht ein normaler Constraint, der die zuvor definierten Variablen oder Operationen benutzt. Ein Let-Ausdruck ist nur innerhalb des Constraints bekannt, in dem er definiert wurde.

- ```
  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
```

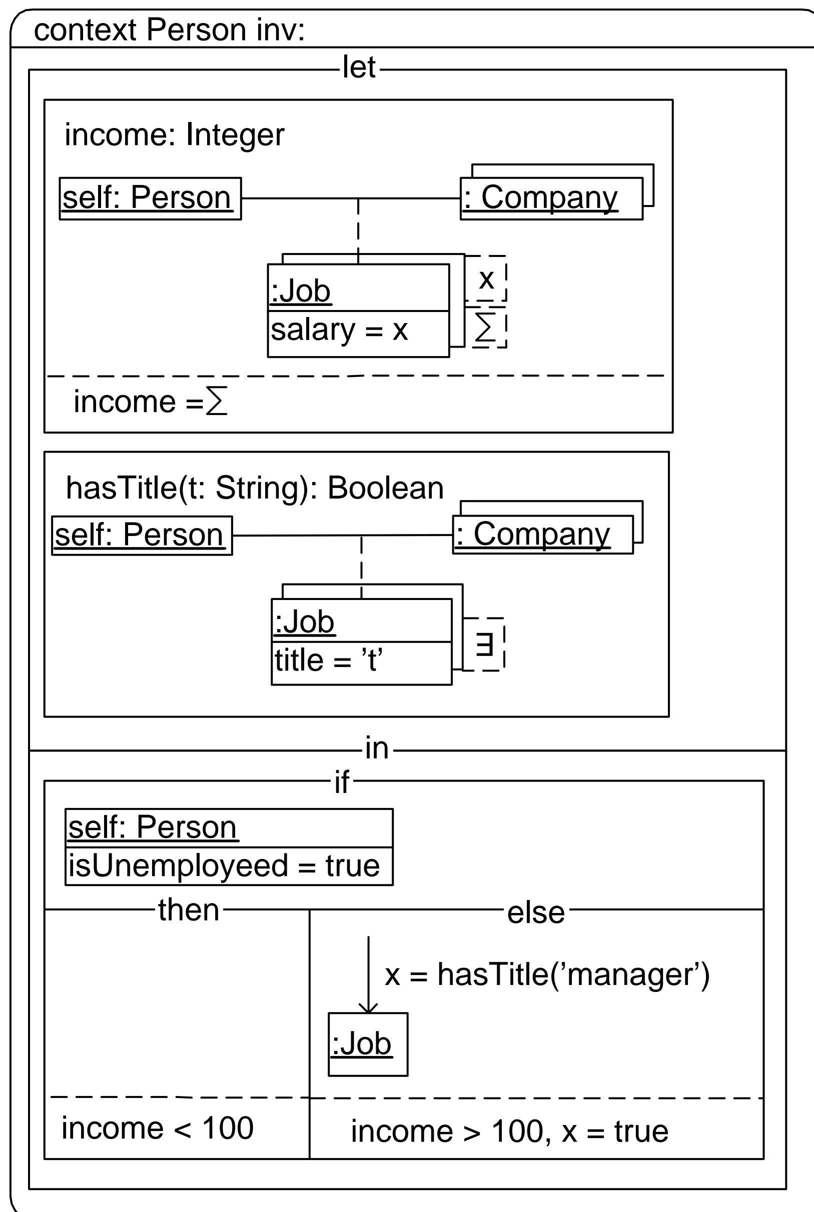


Abbildung 2.10: Let-Constraint

Im oben dargestellten Let-Ausdruck werden eine Variable *income* der Klasse *Person*, welches die Summe der Einkommen aller Jobs der Person ist, und eine Operation *hasTitle*, die einen String übergeben bekommt und einen booleschen Wert zurückgibt, definiert. *hasTitle* liefert true, wenn die Person einen Job mit dem übergebenen Titel ausführt. Im In-Teil werden die Definitionen verwendet. Darin wird ausgesagt, dass eine Person, die nicht arbeitet, ein Einkommen besitzt, das kleiner als 100 ist, ansonsten ist das Einkommen über 100 und diese Person ist ein Manager. Es werden die Operationen *sum* und *exists* verwendet, die im Abschnitt Collections beschrieben werden.
Ein Definition-Constraint besteht nur aus Let-Ausdrücken. Die so definierten Variablen oder Operationen sind auch über den Constraint hinweg bekannt und verwendbar.

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

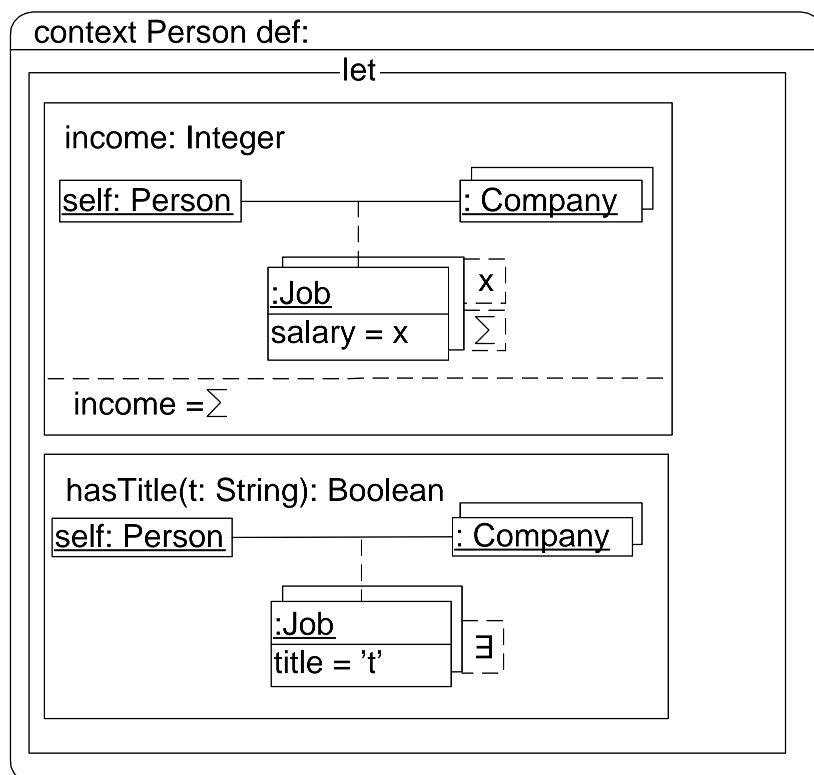


Abbildung 2.11: Ein Definition-Ausdruck

Abstrakte Syntax

Die abstrakte Syntax eines Let-Ausdrucks ist eine *LetExp*, die auf die definierte Variable, eine *VariableDeclaration* und auf den In-Teil, welcher eine *OclExpression* ist, verweist. Die *VariableDeclaration* hat
  ```

eine *initExpression*, auch diese ist eine *OclExpression*. Die Definition von Operationen und Methoden sowie die Definition mehrerer Variablen innerhalb einer *LetExp* wird in der abstrakten Syntax nicht unterstützt. Eine Darstellung der abstrakten Syntax des Constraints aus Abbildung 2.10 ist im Metamodell auf Seite 87 zu sehen. In diesem Metamodell wurde allerdings die Definition der Operation hasTitle nicht berücksichtigt, da sie in der abstrakten Syntax von Ocl nicht unterstützt wird.

2.2.10 Typübereinstimmungen, Re-Typing oder Casting

Für die Basistypen von VOCL gilt dieselbe Typhierarchie wie für die Basistypen in OCL.
Re-Typing oder Casting geschieht durch Anwendung einer Operation *oclAsType(Type2)*.

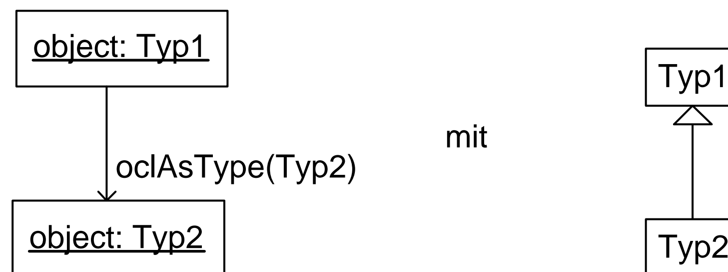


Abbildung 2.12: Re-Typing oder Casting

Die Operation *oclAsType* wandelt ein Objekt vom Typ *Typ1* in ein Objekt vom Typ *Typ2* um, wobei *Typ1* eine Unterklasse von *Typ2* sein muss.

Abstrakte Syntax

Die abstrakte Syntax von OclAsType(Type2) ist eine *OperationCallExp* mit dem Argument Type2.

2.3 Objekte und Eigenschaften

2.3.1 Objekte

Die Visualisierung einer Instanz eines Objektes entspricht der Visualisierung von Objekten im Kollaborationsdiagramm.

2.3.2 Eigenschaften: Attribute

Auf den Wert eines Attributes eines Objektes wird mit Hilfe einer Variablen referiert.

- `context Person inv : self.age`

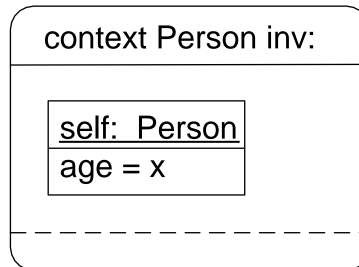


Abbildung 2.13: Attribute von Objekten

In der Variablen x steht der Wert des Alters einer Person, im Bedingungsteil können nun Aussagen über den Wert von x gemacht werden, z.B. Das Alter einer Person ist immer größer als 0:

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

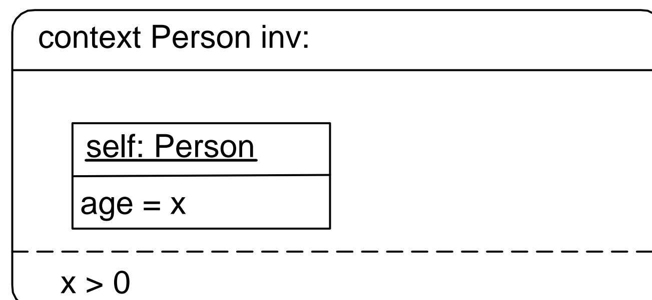


Abbildung 2.14: Attribute von Objekten

Abstrakte Syntax

Die Verwendung der Werte von Attributen eines Objektes entspricht abstrakt einer *AttributeCallExp*. Diese referenziert das Attribut mit dem entsprechenden Namen. Dieses Attribut verweist auf seinen Typ. Eine *AttributeCallExp* kann eine Quelle(*source*) haben, von wo aus sie aufgerufen wird. Die Quelle ist eine *OclExpression*. In Abbildung 2.14 ist der Name des referenzierten Attributs "age", *source* der *AttributeCallExp* ist eine *VariableExp*, die die Variable *self* referenziert. Eine ausführliche Darstellung der abstrakten Syntax des Constraints aus Abbildung 2.14 ist im Metamodell auf Seite 88 zu sehen.

2.3.3 Eigenschaften: Operationen

Die Ausführung einer Operation auf einem Objekt wird folgendermaßen visualisiert:

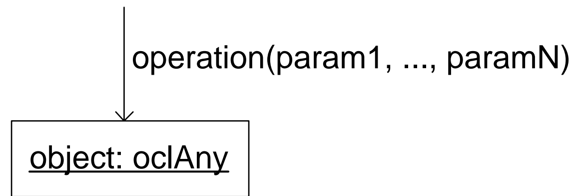


Abbildung 2.15: Prinzipieller Aufruf einer Operation

Die Operation *operation* mit den Parametern *param1,...,param N* wird auf ein Objekt *object* irgendeines Typs angewendet. Der Rückgabewert dieser Operation steht in der dafür reservierten Variable *result*, über die dann Aussagen im Bedingungsteil gemacht werden können.
Der Rückgabewert kann auch einer anderen Variablen zugewiesen werden:

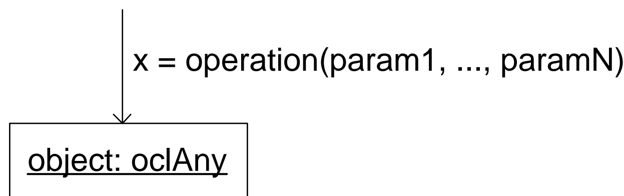


Abbildung 2.16: Prinzipieller Aufruf einer Operation mit Zuweisung an eine Variable

Ein Beispiel für eine Operation wäre:

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

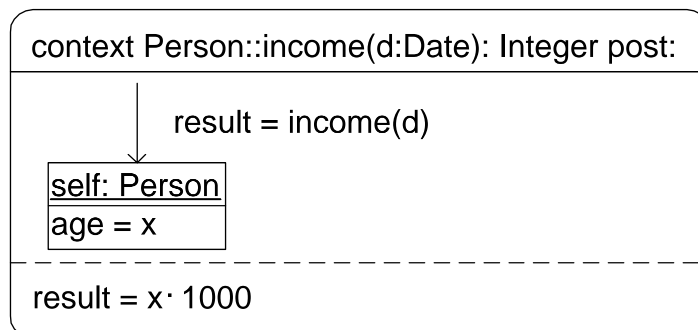


Abbildung 2.17: Aufruf einer Operation mit Zuweisung an eine Variable

Diese Nachbedingung sagt aus, dass das Einkommen einer Person gleich dem Alter der Person multipliziert mit 1000 ist.
Wenn die Operation keine Parameter benötigt, werden leere Klammern anstelle der Parameterliste geschrieben.

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

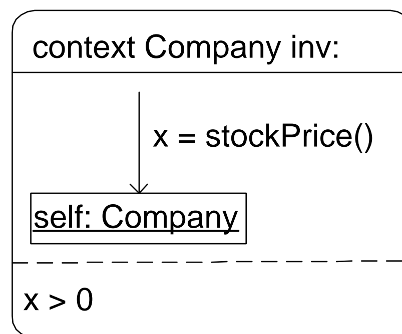


Abbildung 2.18: Aufruf einer Operation ohne Parameter

Dieser Constraint definiert, dass die Aktienpreise einer Company größer als 0 sein müssen.

Hat die Operation einen Rückgabetyp, der kein primitiver Datentyp ist, z.B. eine Collection, dann wird dies durch einen Pfeil von der Instanz, auf die die Operation angewendet wird, zur Instanz, die zurückgeliefert wird, visualisiert.
Ein Beispiel hierfür ist die Operation *including(x)*, die in Abbildung 2.38 zu sehen ist.

Abstrakte Syntax

Der Aufruf einer Operation ist in der abstrakten Syntax eine *OperationCallExp*, die auf die entsprechende Operation und den entsprechenden Rückgabetyp verweist und Argumente hat. Eine ausführliche Darstellung der abstrakten Syntax des Constraints aus Abbildung 2.17 ist im Metamodell auf Seite 89 zu sehen.

2.3.4 Eigenschaften: Assoziationsenden und Navigation

Die Navigation entlang einer Assoziation wird, wie in UML, durch einen Link zwischen den Klasseninstanzen dagestellt. Zur Navigation kann der Rollenname des entgegengesetzten Assoziationsendes verwendet werden. Wenn eindeutig ist, um welche Navigation es sich handelt, kann der Name der Navigation weggelassen werden; das ist genau dann der Fall, wenn es nur eine Navigation zwischen den jeweiligen Objekten gibt. Das Ergebnis dieses Ausdrucks hat die Kardinalität, die im Klassendiagramm definiert wurde. Es können nun Aussagen über die Eigenschaften des Objektes/der Objekte, zu

dem/denen navigiert wurde, gemacht werden. Die Navigation entlang mehrerer Assoziationen beginnt immer beim Objekt *self*, wenn es eines gibt. Ansonsten beginnt die Navigation bei dem im Kontextteil definierten Objekt oder, wenn *self* weggelassen wurde bei dem einzigen Objekt vom Typ des Kontextes. Existieren mehrere Inv-, Post- oder Pre-Ausdrücke, so werden sie als einzelne Contraints visualisiert.

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

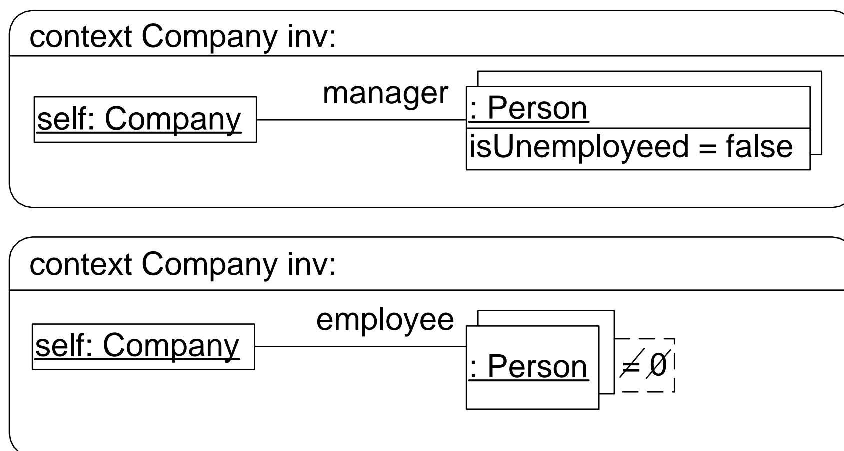


Abbildung 2.19: Assoziationsenden und Navigation

Der oben dargestellte Constraint sagt aus: Der Manager einer Company ist in einer Company angestellt und eine Company hat mindestens einen Angestellten.
In diesem Constraint wird die Operation *notEmpty()* visualisiert, die auf eine Collection angewendet werden kann. An der Collection wird das Zeichen $\neq \emptyset$ in Anlehnung an die Überprüfung einer Menge notiert.
Auf Collections gibt es vordefinierte Operationen, wie z. B. die Operation *size()*, die wie folgt visualisiert wird :

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

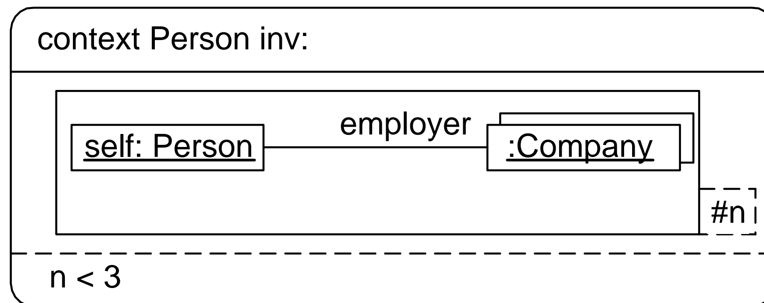


Abbildung 2.20: Assoziationsenden und Navigation

Dieser Constraint spezifiziert: Eine Person hat weniger als 3 Arbeitgeber.
Die Operation *size()* wird auf eine Collection (in diesem Beispiel ein Set) von Arbeitgebern angewendet, in der Variablen n steht die Anzahl der Elemente, die in dieser Collection sind.

Eine weitere Operation ist das Überprüfen, ob eine Collection leer ist oder nicht, mit der Operation *isEmpty()* bzw. *notEmpty()*.

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

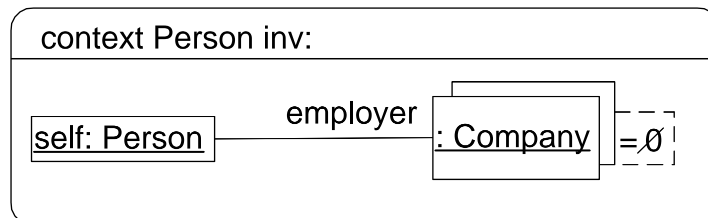


Abbildung 2.21: Die Operation isEmpty

Dieser Constraint spezifiziert, dass eine Person keine Arbeitgeber hat.

Abstrakte Syntax

Die Navigation entlang einer Assoziation zu einem Assoziationsende ist in der abstrakten Syntax eine *AssociationEndCallExp*, die das Assoziationsende und den Assoziationsanfang referenziert. Eine ausführliche Darstellung der abstrakten Syntax des Constraints aus Abbildung 2.20 ist im Metamodell auf Seite 90 zu sehen.

Navigation über Assoziationsenden mit Kardinalität 0..1

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

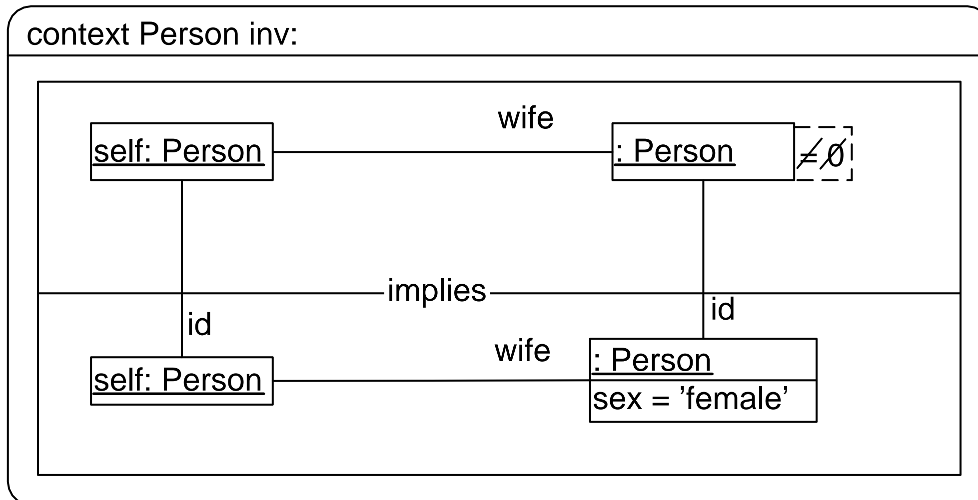


Abbildung 2.22: Assoziationsenden mit Kardinalität 0..1

Dieser Constraint sagt aus: Dass eine Person eine Frau hat, impliziert, dass diese Frau weiblich ist. Aus der Tatsache, dass die Kardinalität der Assoziation *wife* 0 oder 1 ist, lässt sich schließen, dass *wife* nur ein Element enthält, was dann auch so visualisiert werden kann.

Kombination von Eigenschaften

Die Verknüpfung mehrerer Teilaussagen durch *and* wird nicht extra visualisiert, nebeneinander oder untereinander stehende Teilaussagen werden automatisch durch *and* verknüpft.

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

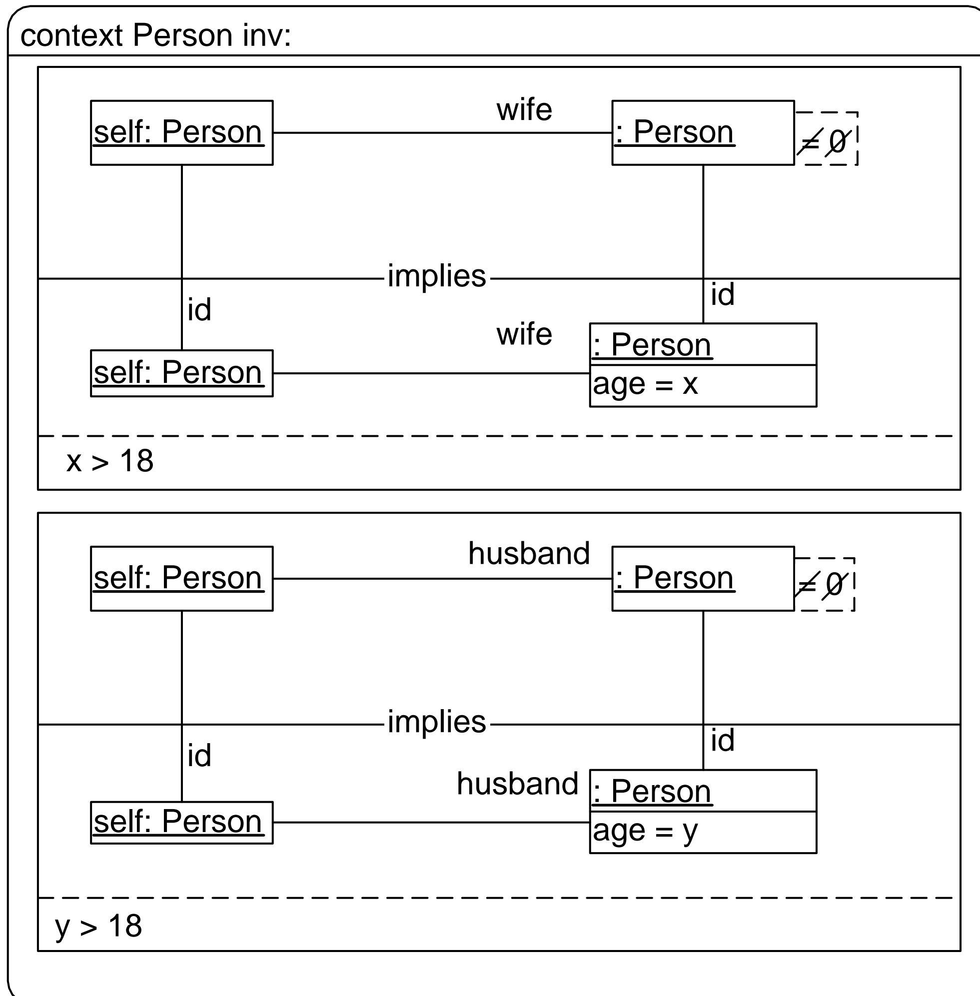


Abbildung 2.23: Kombination von Eigenschaften durch and

Durch diesen Constraint wird ausgesagt: Verheiratete Personen sind mindestens 18.

Eine Verknüpfung durch *or* wird durch einen eigenen Or-Rahmen dargestellt, die Aussagen links und rechts vom *or* (oder überhalb und unterhalb des *or*) werden miteinander verodert.

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

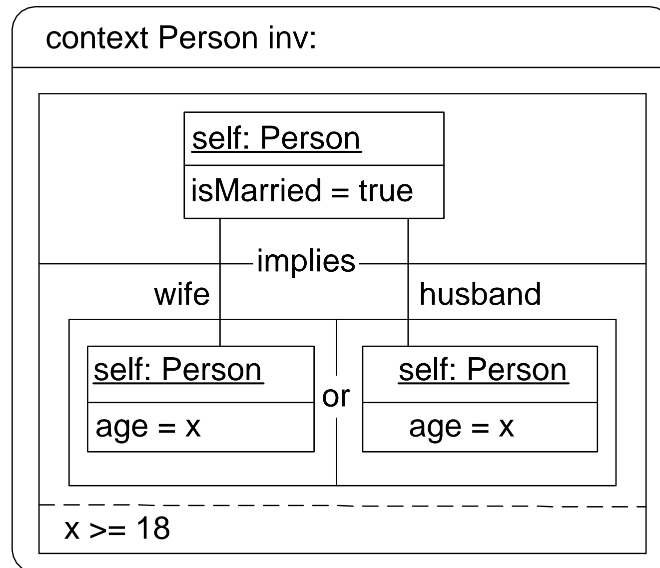


Abbildung 2.24: Kombination von Eigenschaften durch or

Der obige Constraint spezifiziert: Wenn eine Person verheiratet ist, dann ist der Ehepartner mindestens 18.

Abstrakte Syntax

Das Kombinieren von Eigenschaften mit Hilfe von "and" oder "or" ist abstrakt eine *OperationCallExp*. Die von ihr referenzierte Operation hat dann den Namen "and" oder "or", der Typ der *OperationCallExp* ist "Boolean". Eine ausführliche Darstellung der abstrakten Syntax des Constraints aus Abbildung 2.24 ist im Metamodell auf Seite 91 zu sehen.

2.3.5 Navigation zu Assoziationsklassen

Die Navigation zu Assoziationsklassen wird genauso dargestellt wie die Navigation zu anderen Klassen (fehlt ein Rollenname kann der kleingeschriebene Klassenname für die Navigation verwendet werden).

- `context Person inv: self.birthDate < self.job.startDate`

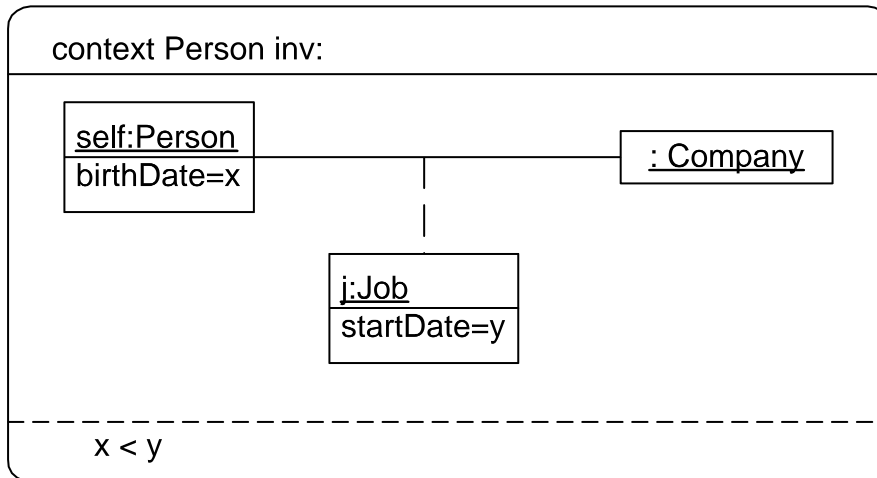


Abbildung 2.25: Navigation zu Assoziationsklassen

Dieser Constraint spezifiziert, dass der Beginn eines Jobs einer Person nach ihrem Geburtstag ist.

Abstrakte Syntax

Die Navigation zu Assoziationsklassen wird in der abstrakten Syntax auf eine *AssociationClassCallExp* abgebildet, die eine *AssociationClass* mit dem Namen der Assoziationsklasse referenziert. Eine ausführliche Darstellung der abstrakten Syntax des Constraints aus Abbildung 2.25 ist im Metamodell auf Seite 92 zu sehen.

2.3.6 Navigation von Assoziationsklassen aus

Die Navigation ausgehend von Assoziationsklassen wird analog zur Navigation von normalen Klassen aus visualisiert.

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

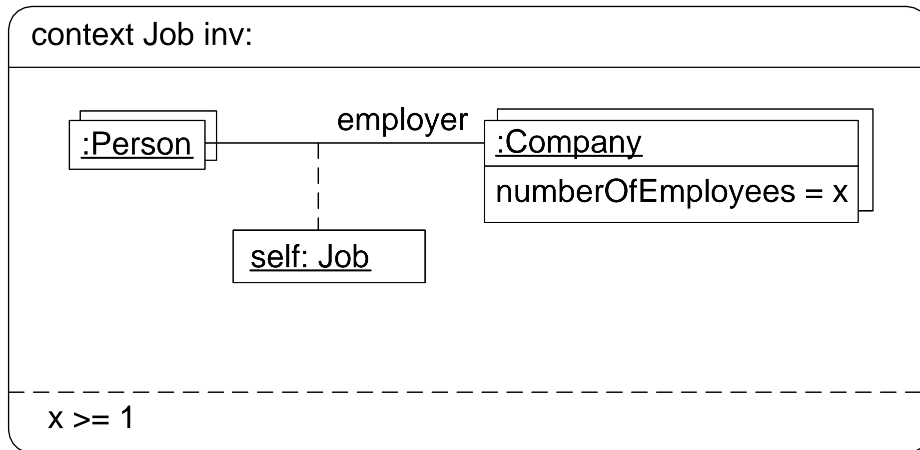


Abbildung 2.26: Navigation von Assoziationsklassen aus

Der Constraint definiert, dass ein Job durch mindestens einen Angestellten einer Firma ausgeführt wird.

Abstrakte Syntax

Die abstrakte Syntax einer Navigation von Assoziationsklassen aus unterscheidet sich nicht von der abstrakten Syntax einer "normalen" Navigation. Eine ausführliche Darstellung der abstrakten Syntax des Constraints aus Abbildung 2.26 ist im Metamodell auf Seite 93 zu sehen.

2.3.7 Navigation entlang Assoziationen mit Qualifiern

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

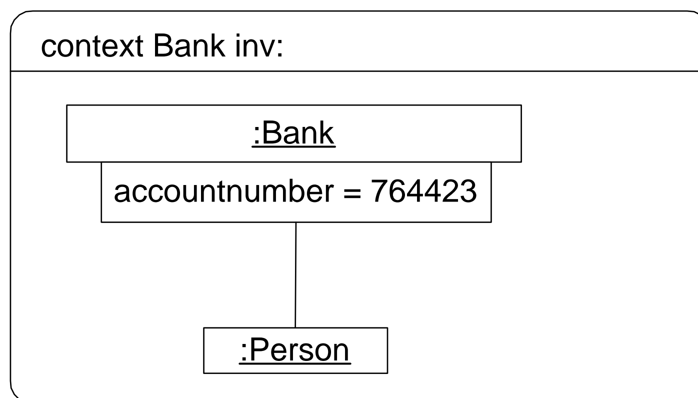


Abbildung 2.27: Navigation mit Qualifiern

Mit Hilfe des Qualifiers *accountnumber* ergibt die Navigation in diesem Constraint genau eine Person, und zwar die, die die *accountnumber* 8764423 hat. Der Qualifier und sein Wert werden unterhalb der Klasse notiert.

Abstrakte Syntax

Die Navigation entlang einer Assoziation zu einem Assoziationsende mit Qualifiern ist in der abstrakten Syntax eine *AssociationEndCallExp*, die neben dem Assoziationsende und -anfang auch auf einen Qualifier verweist. Dieser ist eine *OclExpression*. Eine ausführliche Darstellung der abstrakten Syntax des Constraints aus Abbildung 2.27 ist im Metamodell auf Seite 94 zu sehen.

2.3.8 Verwenden von Eigenschaften von Supertypen

Für das nächste Beispiel wird ein zusätzliches Klassendiagramm benötigt.

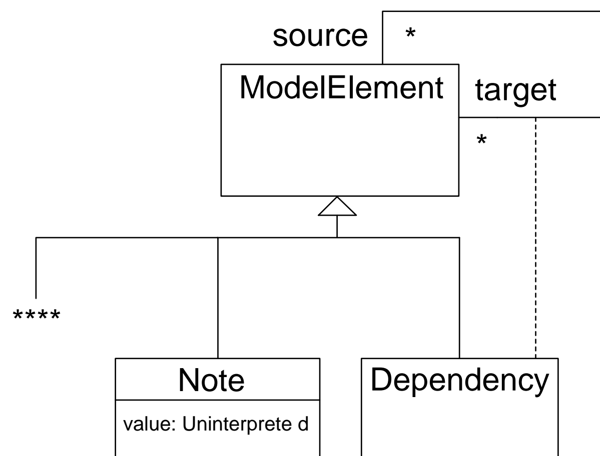


Abbildung 2.28: verwendetes Klassendiagramm

Das Verwenden von Eigenschaften von Supertypen mit der vordefinierten Operation *oclAsType(Type2)* wird folgendermaßen visualisiert:

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

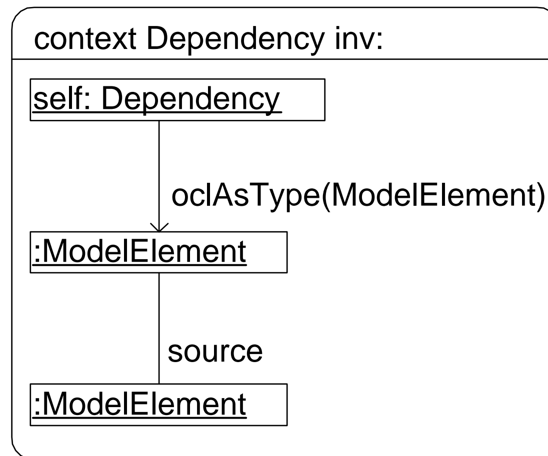


Abbildung 2.29: Eigenschaften von Supertypen

In diesem Constraint kann durch das Umwandeln eines Objektes vom Typ *Dependency* in ein Objekt des Supertyps *ModelElement* über die Assoziation *source* des Supertyps navigiert werden.

Abstrakte Syntax

Die abstrakte Syntax von oclAsType(Type2) ist eine *OperationCallExp* mit dem Argument Type2.

2.3.9 Vordefinierte Eigenschaften

Die vordefinierten Eigenschaften aller Objekte werden wie folgt in VOCL dargestellt:

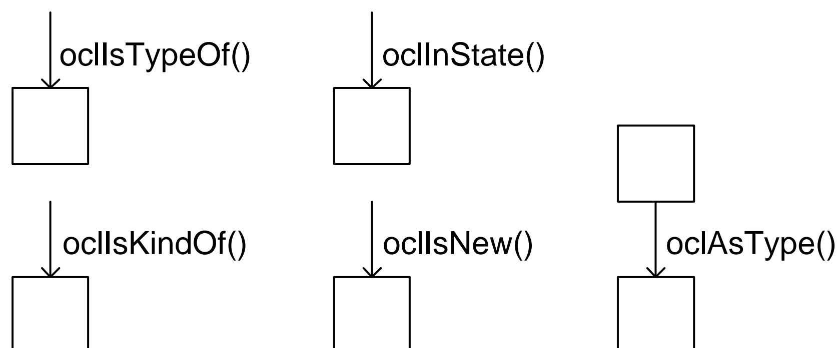


Abbildung 2.30: Vordefinierte Eigenschaften

Dass jede Instanz der Klasse Person vom Typ *Person* ist und nicht vom Typ *Company*, wird so

spezifiziert:

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

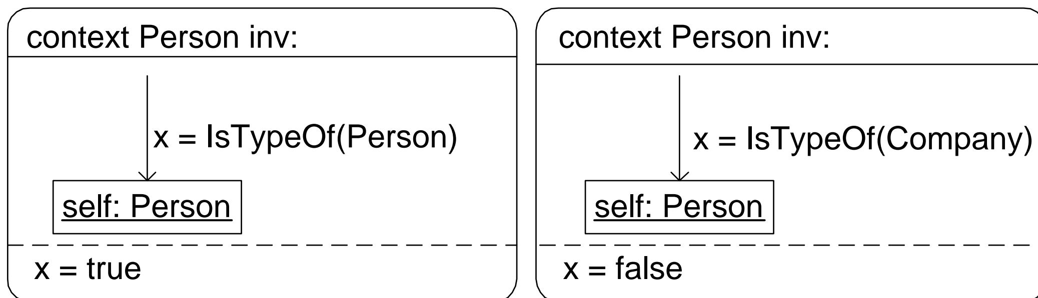


Abbildung 2.31: Vordefinierte Eigenschaften

Abstrakte Syntax

Die abstrakte Syntax dieser vordefinierten Eigenschaften ist eine *OperationCallExp*. Eine ausführliche Darstellung der abstrakten Syntax des linken Teilconstraints aus Abbildung 2.31 ist im Metamodell auf Seite 95 zu sehen.

2.3.10 Eigenschaften von Klassen

Die bisher behandelten Eigenschaften waren Eigenschaften von Instanzen von Klassen.
Es gibt auch Eigenschaften, die auf Klassen selbst definiert sind, ein Beispiel hierfür ist die Eigenschaft *allInstances*.

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

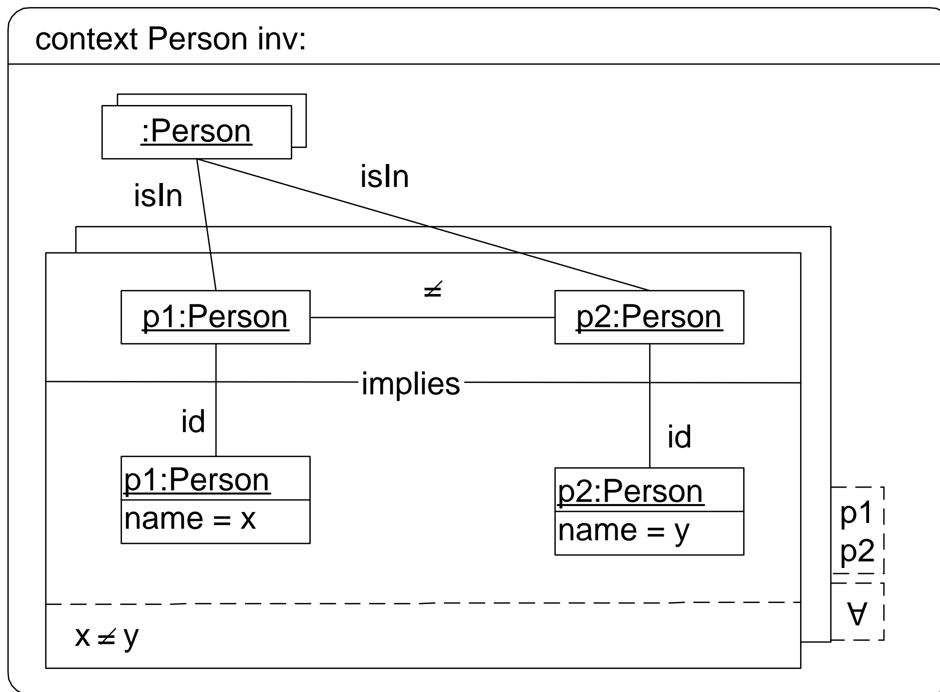


Abbildung 2.32: Eigenschaften von Klassen

Die Eigenschaft *allInstances()* kann nur auf die Klasse, die der Kontext des Constraints ist, angewendet werden und wird durch das Set mit dem Klassennamen dargestellt. In diesem Fall startet die Navigation von der Menge aller Instanzen dieser Klasse, da es keine Instanz der Klasse (*self*) gibt, von der gestartet werden sollte. Danach wird auf die Menge aller Instanzen der Klasse Person eine Forall-Operation und darauf eine geschachtelte Implies-Operation angewendet. Die Forall-Operation ist auf Collections definiert und hat einen oder mehrere Iteratoren. Es gibt einen Forall-Rahmen, der die geforderten Eigenschaften der Forall-Operation umschließt. Dieser muss für alle Elemente der Collection erfüllt sein. Durch die Operation *isIn* kann auf die einzelnen Instanzen des Sets aller Personen zugegriffen werden. Diese Instanzen entsprechen dem Iterator oder den Iteratoren der Operation.

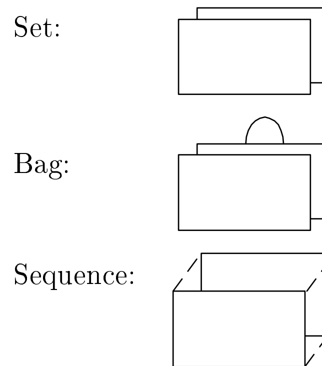
Der Constraint spezifiziert, dass die Namen aller Instanzen vom Typ Person unterschiedlich sind. Dass einzelne Objekte in einer Collection enthalten sein sollen, wird durch den Ausdruck *isIn* an dem entsprechenden Link verdeutlicht.

Abstrakte Syntax

Die Verwendung von Klasseneigenschaften entspricht in der abstrakten Syntax einer *OperationCall-Exp*. Im Metamodell auf Seite 96 ist die abstrakte Syntax des Constraints in Abbildung 2.32 abgebildet.

2.3.11 Collections

Es gibt eine Menge vordefinierter Operationen auf Collections, eine vollständige Liste dieser Operationen ist in der VOCL Standard-Bibliothek zu finden (Kapitel 3 auf Seite 52).
Der Collection-Type ist ein abstrakter Typ, von dem drei konkrete Typen erben: *Set*, *Bag* und *Sequence*. Diese Typen werden folgendermaßen visualisiert:



Einfache Navigationen resultieren in einem *Set*,

- `context Company inv: self.employee`

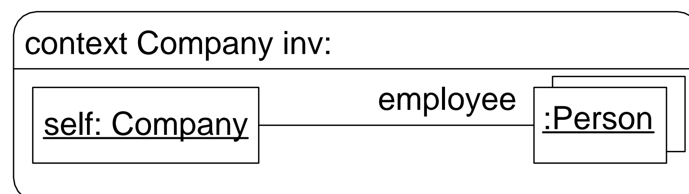


Abbildung 2.33: einfache Navigationen

kombinierte Navigationen ergeben einen *Bag*,

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

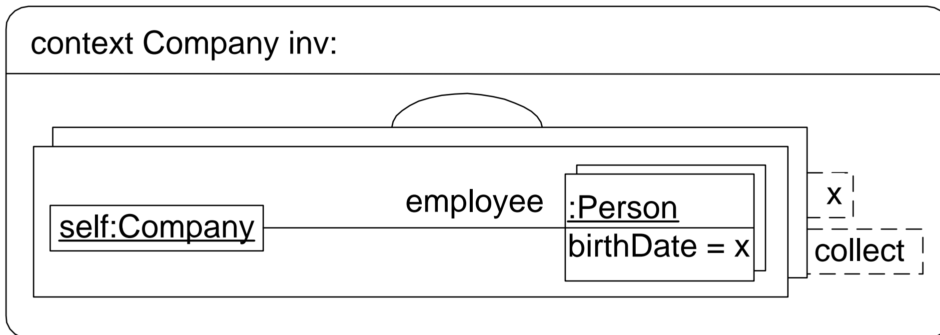


Abbildung 2.34: kombinierte Navigationen

und Navigationen mit *sortedBy* ergeben eine *Sequence*.

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

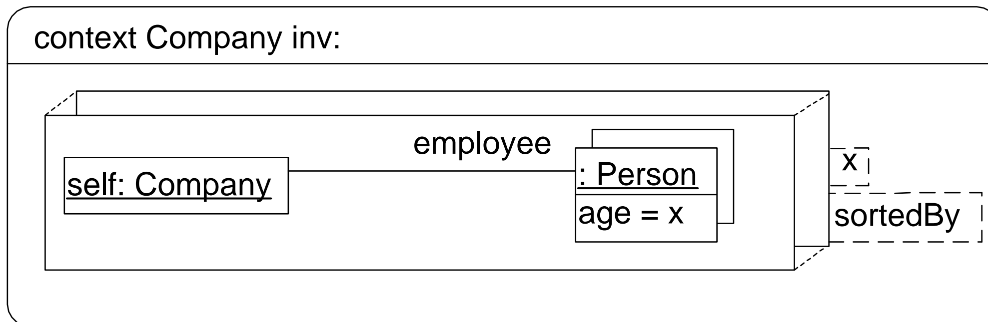


Abbildung 2.35: Navigationen mit *sortedBy*

Operationen auf Collections können neue Collections als Resultat haben., wie z.B. die Vereinigung zweier Collections:

- `collection1->union(collection2)`

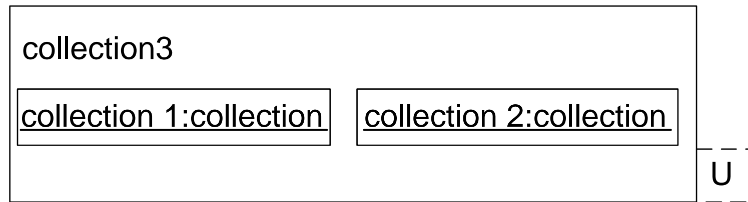


Abbildung 2.36: Operationen auf Collections

Die resultierende Collection kann einen Namen bekommen, der oben links im Vereinigungsrahmen steht, das mathematische Vereinigungszeichen steht rechts am Vereinigungsrahmen.

Abstrakte Syntax

Die abstrakte Syntax einer Navigation zu einer Collection unterscheidet sich nicht von der Navigation zu anderen Klassen, ist also eine *NavigationCallExp* mit zwei Navigationsenden(*navigationSource* und je nachdem, um was für eine Klasse es sich handelt, ein *referredAssociationEnd* oder eine *referredAssociationClass*).

2.3.12 Werte vor der Ausführung einer Operation in der Nachbedingung

Auf den Wert eines Attributes eines Objektes vor der Ausführng einer Operation wird mit dem Attributnamen gefolgt vom Postfix *@pre* referiert.
Dass sich das Alter einer Person an ihrem Geburtstag um 1 erhöht, wird als Constraint so dargestellt:

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

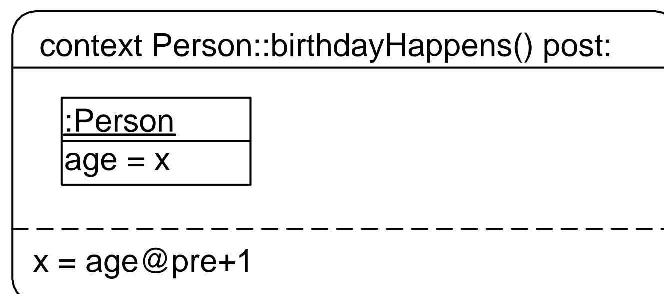


Abbildung 2.37: Attributwerte vor der Ausfühung einer Operation

Bei der Navigation entlang einer Assoziation zu der Menge der Objekte vor der Operation wird der Rollenname gefolgt vom Postfix *@pre* verwendet.

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

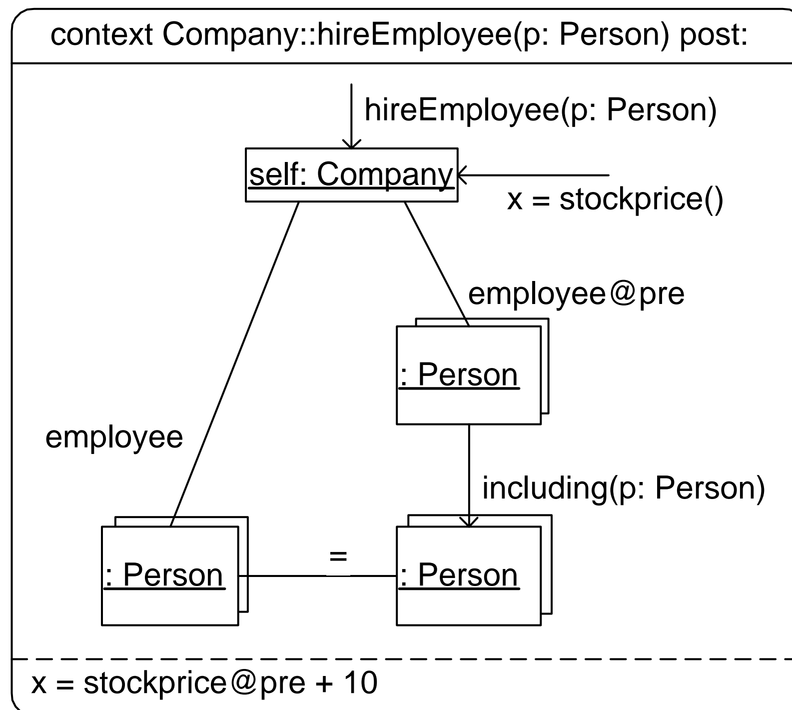


Abbildung 2.38: Navigation zu Objekten vor der Ausführung einer Operation

In diesem Beispiel wird die auf Collections vordefinierte Operation *including* dargestellt. *including* vereinigt die Menge der Arbeitnehmer einer Company vor der Einstellung eines neuen Arbeitnehmers mit dem neuen Arbeitnehmer. Dies wird durch die Operation *including*, die den Namen des Objektes übergeben bekommt und ein neues Set als Ergebnis hat, visualisiert.
Der Constraint spezifiziert, dass nach dem Einstellen eines neuen Arbeitnehmers dieser in der Menge der Arbeitnehmer einer Company enthalten ist und sich die Aktienpreise der Company um 10 erhöht haben.

Abstrakte Syntax

Der Wert eines Attributes vor der Ausführung einer Operation ist abstrakt eine *AtributeCallExp*, die Navigation zu Objekten vor der Ausführung einer Operation ist eine *NavigationCallExp*. Eine ausführliche Darstellung der abstrakten Syntax des Constraints aus Abbildung 2.37 ist im Metamodell auf Seite 97 zu sehen.
  ```

2.4 Operationen auf Collections

In diesem Kapitel wird nur eine Auswahl der Operationen auf Collections aufgeführt. In der Bibliothek (Kap.3 ab Seite 52) sind alle Operationen dargestellt.

Die Select-Operation

Die Select-Operation liefert eine Untermenge einer Collection, deren Elemente bestimmte Eigenschaften haben. Am Select-Rahmen stehen der Iterator und das Schlüsselwort *select*, im Rahmen eine Visualisierung der Selektionseigenschaften. Dieser beinhaltet einen booleschen Wert. Werden mehrere Operationen ausgeführt, werden die durchzuführenden Operationen unter ihren jeweiligen Iteratoren (wenn sie welche haben) der Reihenfolge nach von oben nach unten am Rahmen notiert.

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

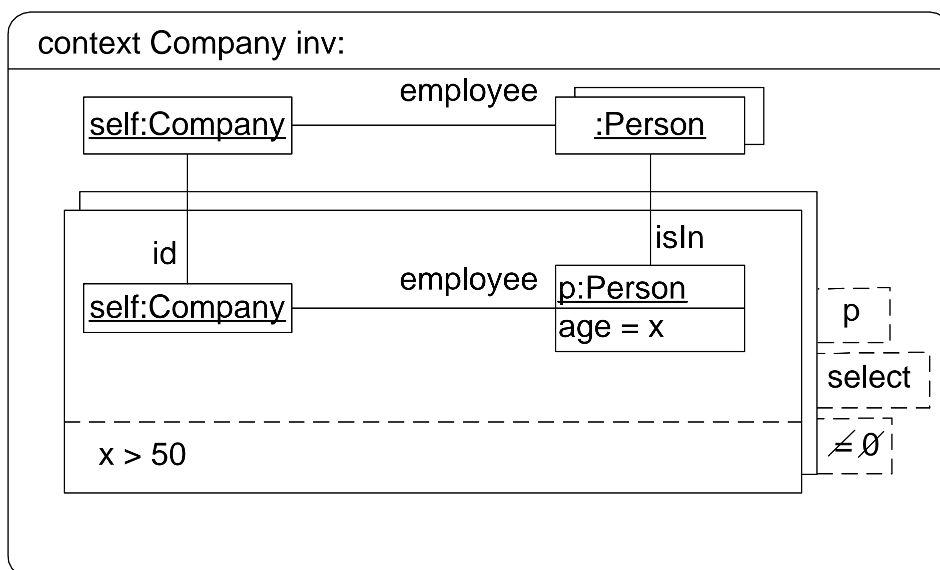


Abbildung 2.39: Eine Select-Operation

Dieser Constraint spezifiziert, dass mindestens ein Angestellter in einer Firma älter als 50 ist.
Der Iterator ist vom Typ Person. Die Selektionseigenschaft ist hier, dass das Alter einer Person grösser als 50 ist.

Die Reject-Operation

Die Reject-Operation liefert eine Untermenge einer Collection, deren Elemente bestimmte Eigenschaften nicht erfüllen. Am Reject-Rahmen stehen der Iterator und das Schlüsselwort *reject*, im Rahmen die Rejecteigenschaft der Reject-Operation. Dieser Rahmen beinhaltet einen booleschen Wert.

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

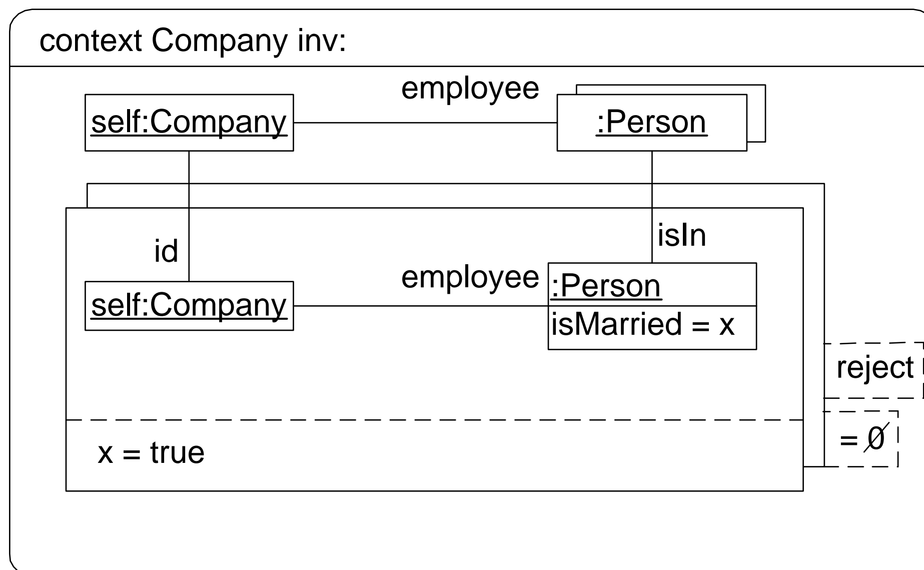


Abbildung 2.40: Eine Reject-Operation

Diese Reject-Operation liefert alle Personen, die nicht verheiratet sind. Der Constraint spezifiziert, dass die Collection von Angestellten, die nicht verheiratet sind, leer ist. Der Iterator ist vom Typ Person, das Attribut *isMarried* wird überprüft. Wenn der Iterator offensichtlich ist, kann er weggelassen werden.
  ```

Die Collect-Operation

Die Select- und Reject-Operationen liefern eine Untermenge der Collection, auf die sie angewendet werden.
Die Collect-Operation liefert eine neue Collection von einem anderen Typ. Im Collect-Rahmen der Collect-Operation wird eine Variable definiert, die den Typ der neuen Collection hat und über dem *collect* notiert wird.
Aus der Menge aller Arbeitnehmer einer Firma wird das Set ihrer Geburtstage gebildet:

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

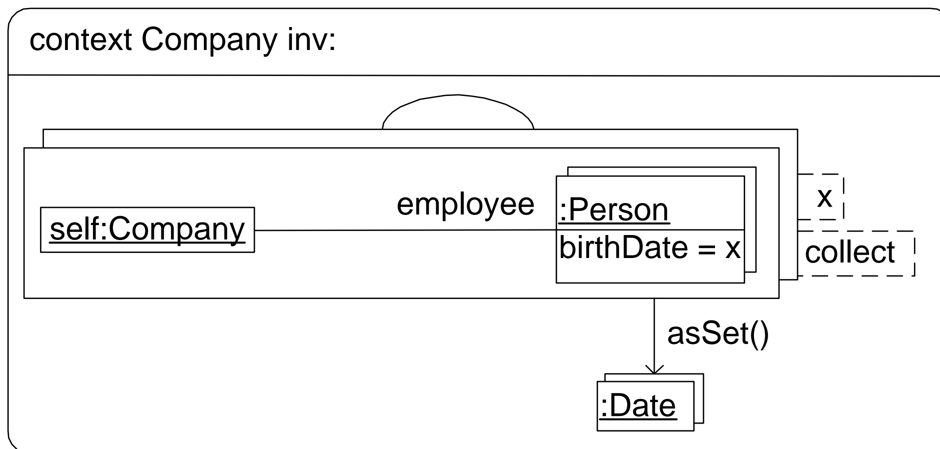


Abbildung 2.41: Eine Collect-Operation

Die Anwendung der Operation *asSet()* auf einen Bag liefert ein Set, aus dem die doppelten Elemente des Bags entfernt wurden.

Die Forall-Operation

Die Forall-Operation wendet einen Constraint auf alle Elemente einer Collection an und hat einen oder mehrere Iteratoren. Es gibt einen Forall-Rahmen, der die Eigenschaften der Forall-Operation umschließt. Diese müssen für alle Elemente der Collection erfüllt sein. An diesem Rahmen werden sowohl der ∀-Operator als auch die Iteratoren notiert.

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

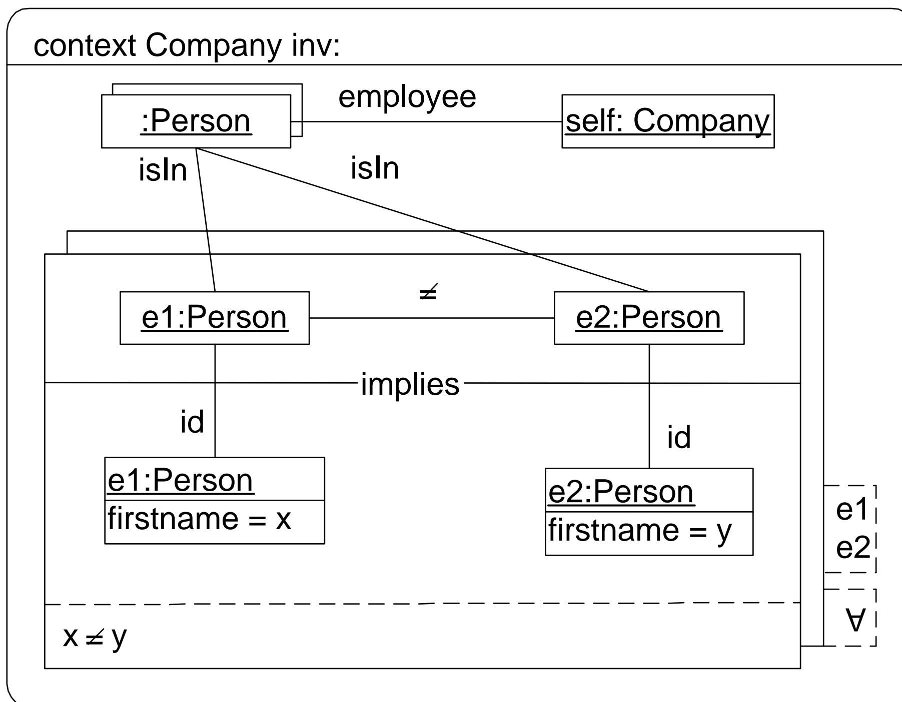


Abbildung 2.42: Eine Forall-Operation

Der Constraint spezifiziert, dass alle Arbeitnehmer einer Firma verschiedene Vornamen haben.
  ```

Die Exists-Operation

Die Exists-Operation wird auf eine Collection angewendet. Auch diese Operation hat einen oder mehrere Iteratoren und einen Rahmen, in dem die Eigenschaften, die für mindestens ein Element der Collection erfüllt sein muss, visualisiert werden. Am Rahmen stehen Iteratoren und das ∃- Zeichen.

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

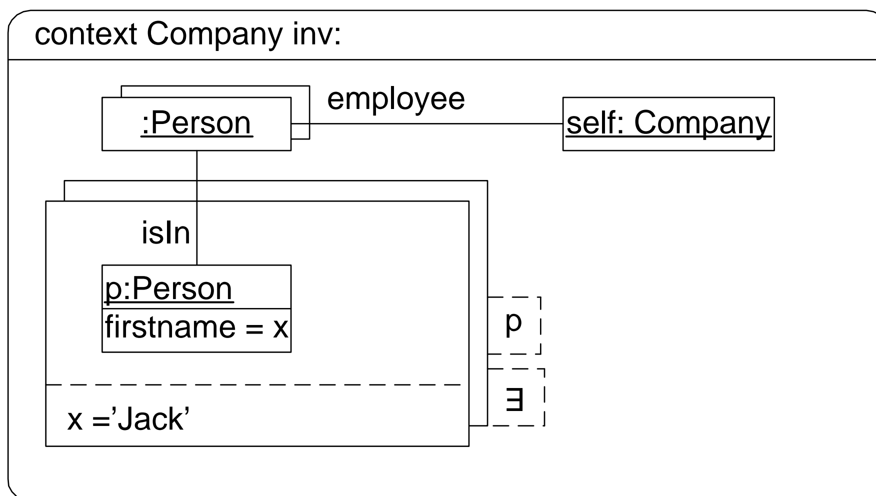


Abbildung 2.43: Eine Exists-Operation

Der Constraint spezifiziert, dass ín einer Firma mindestens ein Angestellter mit dem Vornamen Jack existiert.

Die Iterate-Operation

Auch die Iterate-Operation hat einen Rahmen, in dem Eigenschaften visualisiert werden, und einen Iterator. Hinzu kommt ein Accumulator, der einen Initialwert hat. Die Defintion des Accumulators wird durch die Operation *isNew()* visualisiert.

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

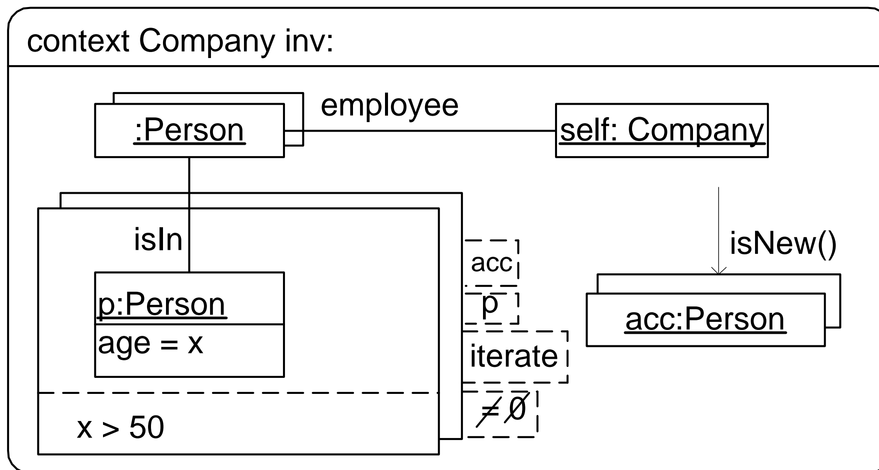


Abbildung 2.44: Eine Iterate-Operation

Dieser Constraint spezifiziert, dass in einer Company die Menge der Angestellten, deren Alter grösser als 50 ist, nicht leer ist.
  ```

Die sum-Operation

Auch die sum-Operation hat einen Rahmen, in dem das Element, über welches summiert werden soll, visualisiert wird. Dieses wird rechts am Rahmen über dem Summenzeichen notiert. Zur Darstellung des Ergebnisses der Summierung wird dann im Bedingungsteil das Summenzeichen verwendet.

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

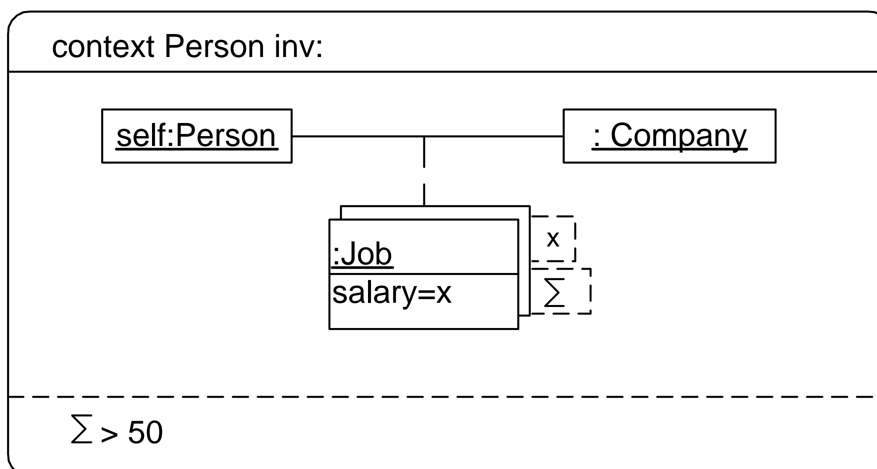


Abbildung 2.45: Eine sum-Operation

Dieser Constraint spezifiziert, dass die Summe der Gehälter der Jobs, die eine Person ausführt, grösser als 50 ist.

Abstrakte Syntax

Der Aufruf der Operationen select, reject, forall, exists und sum wird in der abstrakten Syntax auf eine *IteratorExp* mit dem jeweiligen Namen abgebildet. Der Typ dieser *IteratorExp* ist bei select und reject der entsprechende CollectionType (Set, Bag oder Sequence), bei forall und exists "Boolean" und bei sum der Typ der aufsummierten Elemente. Eine *IteratorExp* verweist auf einen Iterator, welcher eine *VariableDeclaration* mit dem gewählten Namen des Iterators ist. Der Typ, auf den sie verweist, ist der Typ des Iterators. Außerdem hat eine *IteratorExp* einen Verweis auf den Body der Iteration, welcher eine *OclExpression* ist. Werden mehrere Iteratoren verwendet, entspricht dies in der abstrakten Syntax einer mehrfach geschachtelten Iteration.
Die abstrakte Syntax der iterate-Operation ist eine *IterateExp*, die wie bei der *IteratorExp* einen Iterator referenziert und zusätzlich noch einen Akkumulator, eine *VariableDeclaration*, besitzt. Diese Akkumulatorvariable wird durch eine *OclExpression* initialisiert. Die Darstellung der abstrakten Syntax der Select-Operation aus Abbildung 2.39 befindet sich auf Seite 98. Für die Abbildung 2.32 (Seite 31) ist die abstrakte Syntax auf Seite 96 dargestellt.

2.5 Messages

•

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

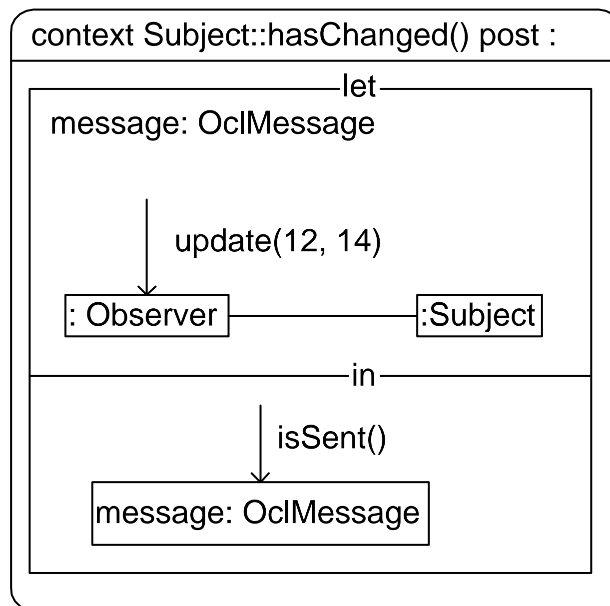


Abbildung 2.46: Messages

Das Anwenden der Operation update auf ein Objekt vom Typ *observer* hat eine Message als Ergebnis, die im Let-Teil des Constraints definiert wird. Dass diese Message gesendet wird, wird durch den Aufruf der Operation *isSent()* in gewohnter Weise visualisiert.

•

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

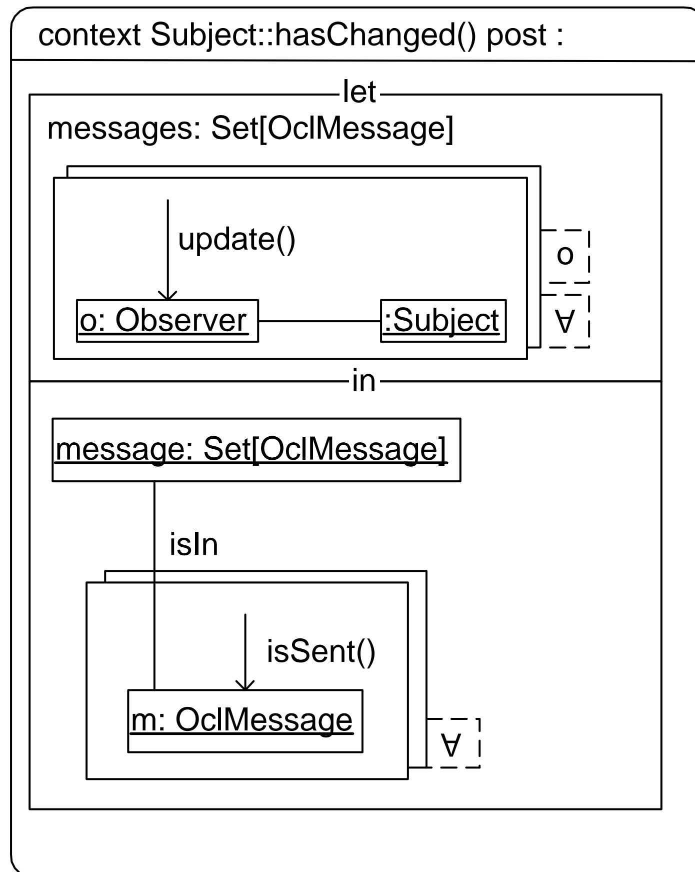


Abbildung 2.47: Messages

Das Anwenden der Operation *update* auf alle Objekte vom Typ *obserser* resultiert in einem Set von Messages, welches im Let-Teil definiert wird. Im In-Teil des Constraints wird spezifiziert, dass alle Messages aus diesem Set gesendet werden.

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

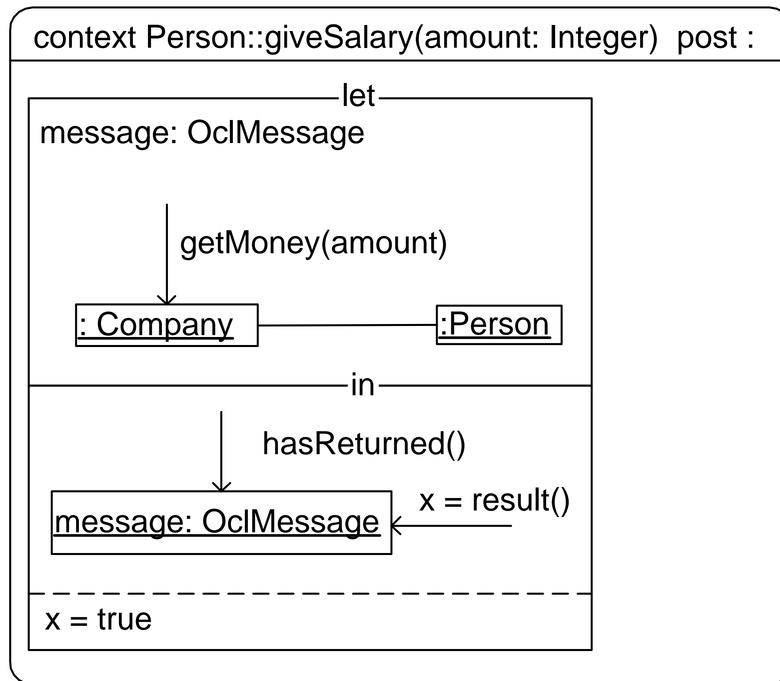


Abbildung 2.48: Messages

Das Anwenden der Operation *getMoney* der Klasse *Company* liefert eine Message, die im Let-Teil als *message* definiert wird. Der In-Teil des Constraints spezifiziert, dass *hasReturnd* und *result* true sind. Wenn der boolsche Rückgabewert nicht weiterverwendet wird, kann er im Bedingungsteil weggelassen werden. In diesem Fall muss er den Wert true besitzen.

Abstrakte Syntax

Die abstrakte Syntax einer Message ist eine *OclMessageExp*. Diese verweist entweder auf eine *SendAction* oder eine *CallAction*, welche wiederum ein *Signal* oder eine *Operation* referenzieren. Eine ausführliche Darstellung der abstrakten Syntax des Constraints aus Abbildung 2.46 ist im Metamodell auf Seite 99 zu sehen.

2.6 Tupel

Für die Definition eines Tupels wird ein Rahmen verwendet, in dem oben links der Name des Tupels steht. Wenn nur ein Tupel definiert wird, kann dieser Rahmen auch weggelassen werden. Innerhalb dieses Tupel-Rahmens wird für jedes Tupelelement ein eigener Rahmen angelegt, in dem die Definition des Wertes des Elements visualisiert wird, der Elementname sowie der Elementtyp und der Rückgabewert stehen oben links in diesem Rahmen.

- ```
  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()}
  )
  ```
  ```

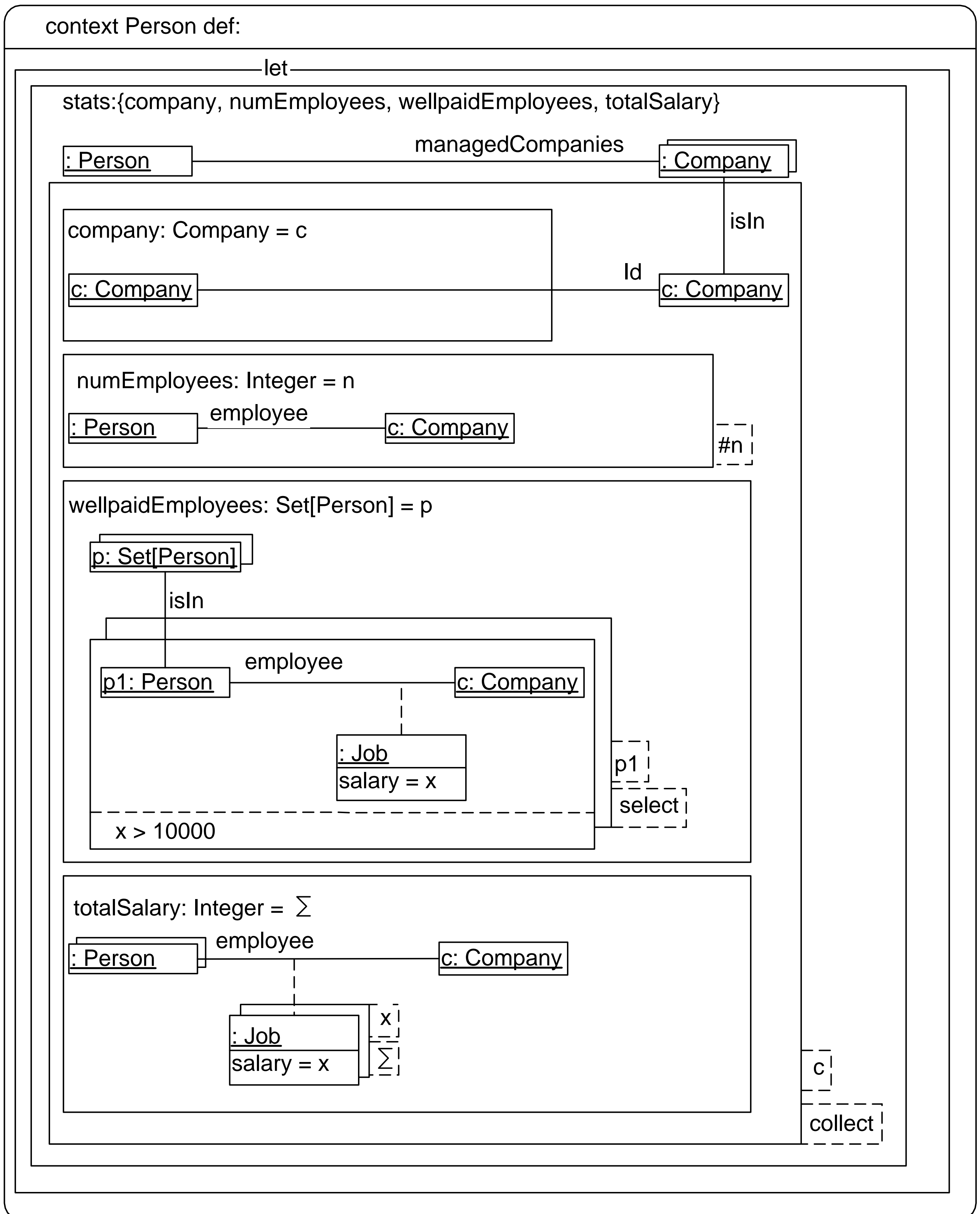


Abbildung 2.49: Eine Tupel-Definition

Im Kontext einer Person wird ein Tupel *stats* definiert, das aus vier Elementen besteht: einer Firma *company*, die von der Person gemanagt wird, der Anzahl Angestellter in dieser Firma *numEmployees*, der Menge der gutbezahlten Angestellten *wellpaidEmployees* und dem Gesamtgehalt *totalSalary*, das die Firma an all ihre Angestellten bezahlt. Dieses Tupel wird im Let-Teil des Def-Constraints definiert und ist somit über den Bereich des Constraints hinaus bekannt und kann für den folgenden Constraint verwendet werden.
In diesem werden alle Tupel vom Typ *stats* nach dem Gesamtgehalt aufsteigend sortiert, aus dieser Menge wird das letzte Element mittels der Operation last, die auf eine Collection angewendet werden kann und ein Objekt liefert, ermittelt. Das Ergebnis ist ein Tupel vom Typ *stats*. Die Menge *wellpaidEmployees* dieses Tupels soll die Person selbst enthalten, was durch die Operation includes sichergestellt wird. Includes liefert dann true, wenn das Element in der Collection enthalten ist. Wenn der Rückgabewert dieser Operation nicht im Bedingungsteil verwendet wird, dann ist er true. Hierbei kann man innerhalb des Tupels die jeweils verwendeten Tupelelemente notieren. Wird ein Tupelelement für eine Aussage nicht verwendet, kann es weggelassen werden.

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

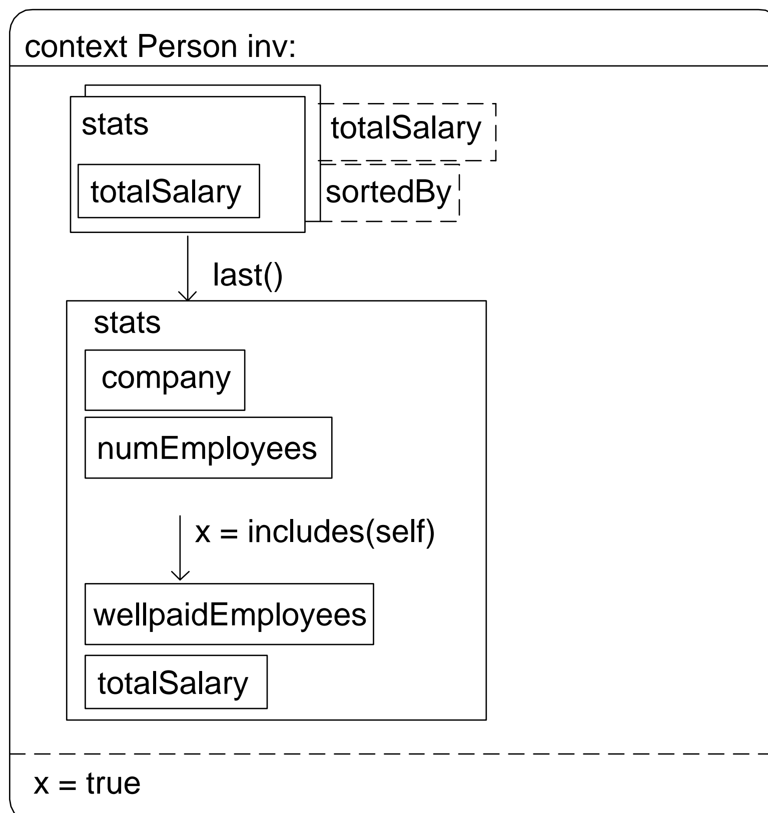


Abbildung 2.50: Verwendung eines Tupels

Der Constraint spezifiziert, dass ein Manager einer Firma, in der Firma, die das meiste Gesamtgehalt aller Firmen, die er managt, hat, zu den gutbezahlten Angestellten gehört.

Abstrakte Syntax

Der Typ eines Tupelelements in der abstrakten Syntax ist *TupleType*. Dieser verweist auf die definierten Attribute, welche einen Namen haben und einen Typ referenzieren.

2.7 Komposition von Constraints

Ein Constraint kann aus der Komposition von vorher definierten Constraints bestehen. Er spezifiziert die Verundung aller Constraints innerhalb der Komposition. Die verwendeten Constraints müssen vorher mit einem Namen definiert worden sein, da mit Hilfe dieses Constraintnamens auf sie referenziert wird. Die einzelnen Constraints stehen in einem Rahmen innerhalb des Kompositionsrahmens.

Möchte man spezifizieren, dass alle Arbeitnehmer einer Firma verschiedene Vornamen haben und einer von ihnen den Vornamen Jack hat, kann das durch die Komposition der folgenden zwei Constraints ausgedrückt werden:

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

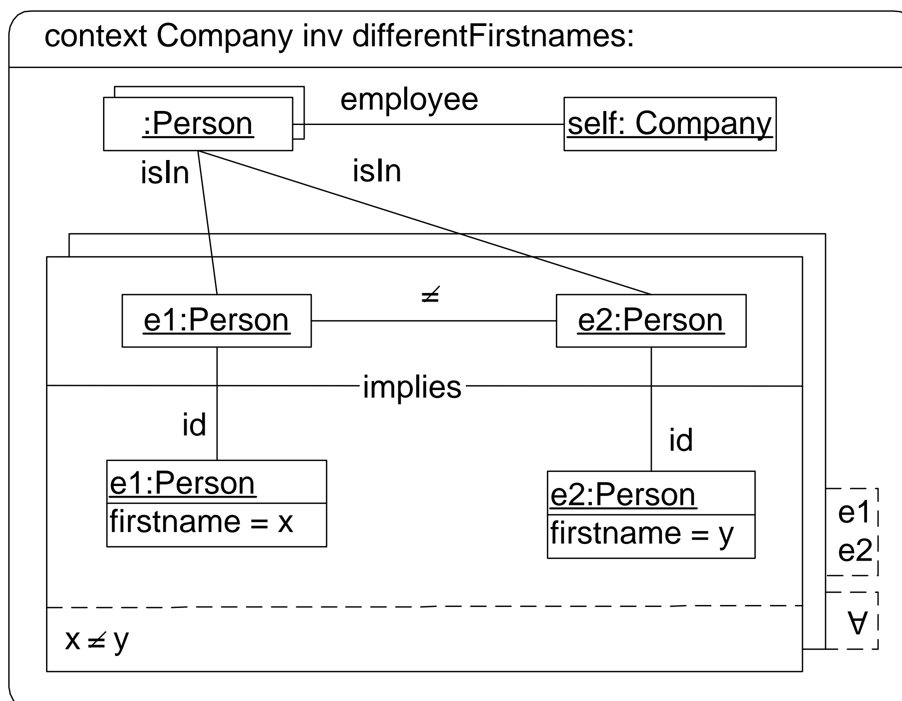


Abbildung 2.51: erster Constraint
  ```

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

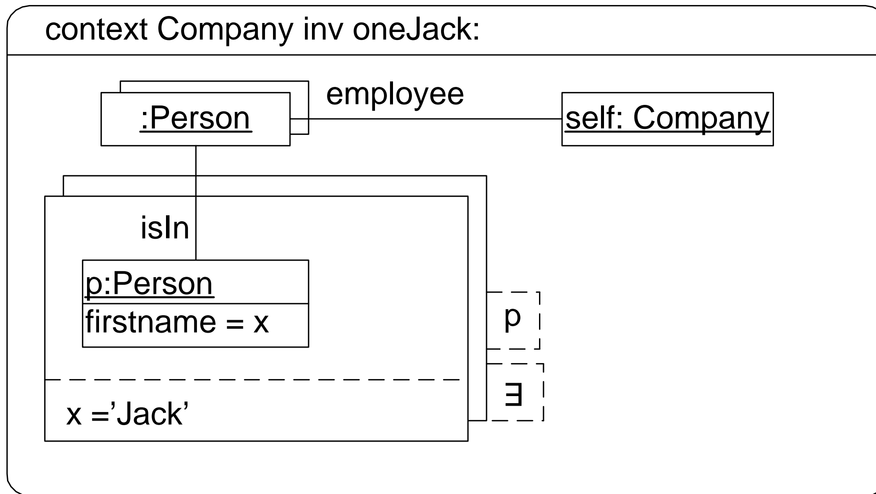


Abbildung 2.52: zweiter Constraint

Der komponierte Constraint sieht dann so aus:

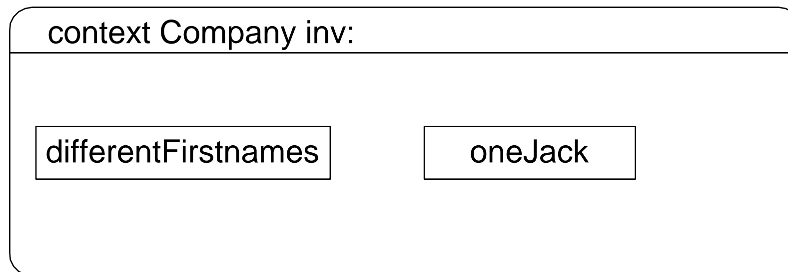


Abbildung 2.53: komponierter Constraint

Abstrakte Syntax

Die abstrakte Syntax von kombinierten Constraints ist eine Verknüpfung der bodys der Einzelconstraints durch die Operation "and". Eine ausführliche Darstellung der abstrakten Syntax des Constraints aus Abbildung 2.34 ist im Metamodell auf Seite 100 zu sehen.
  ```

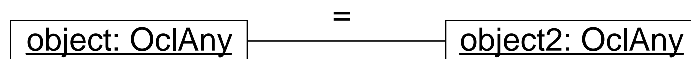
Kapitel 3

Die OCL Standardbibliothek

3.1 OclAny und OclVoid

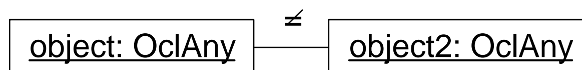
3.1.1 OclAny

- object = (object2: OclAny): Boolean



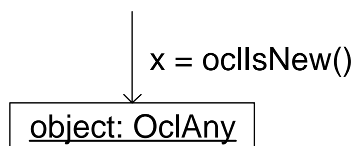
Das Überprüfen der Gleichheit zweier Objekte vom Typ OclAny wird durch einen Link mit einem Gleichheitszeichen zwischen den Objekten visualisiert. Da OclAny ein abstrakter Typ ist, muss die Visualisierung des entsprechenden Typs an dieser Stelle stehen.

- object <> (object2: OclAny): Boolean



Das Überprüfen der Ungleichheit zweier Objekte vom Typ OclAny wird durch einen Link mit einem $\neq$-Zeichen zwischen den Objekten visualisiert.

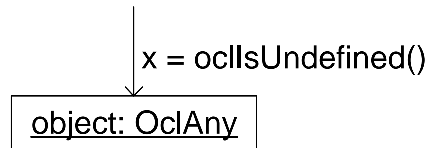
- object.oclIsNew(): Boolean



oclIsNew() ist eine Operation auf ein Objekt des Typs OclAny. Sie kann nur in einer Nachbedingung verwendet werden. Die Operation liefert true, wenn das Object während der Ausführung der Operation entstanden ist. Die Operation wird durch einen Pfeil auf das Objekt visualisiert, der als

Beschriftung x=oclIsNew()enthält, wobei in x der Rückgabewert enthalten ist. Wird der Rückgabewert einer Operation nicht weiterverwendet, so kann er weggelassen werden und wird dann als true angesehen.

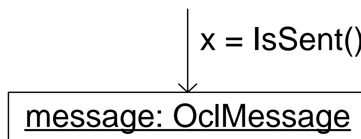
- object.oclIsUndefined(): Boolean



oclIsUndefined() ist eine Operation auf ein Objekt des Typs OclAny. Sie liefert true, wenn das Objekt gleich OclUndefined ist. Die Operation wird durch einen Pfeil auf das Objekt visualisiert, der als Beschriftung oclIsUndefined() enthält.

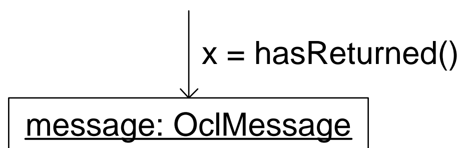
3.1.2 OclMessage

- message.isSent(): Boolean



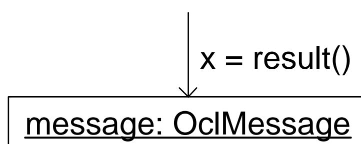
isSent() wird wie eine Operation auf ein Objekt visualisiert. Der Rückgabewert ist true, wenn eine Message zum Ziel gesendet wurde.

- message.hasReturned(): Boolean



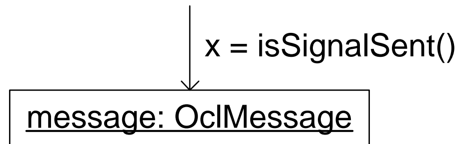
hasReturned() liefert true, wenn die Message eine Operation aufgerufen und diese einen Wert zurückgegeben hat.

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



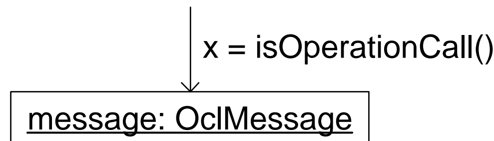
result() liefert das Ergebnis der aufgerufenenen Operation zurück.

- message.isSignalSent(): Boolean



isSignalSent() liefert true, wenn die OclMessage ein gesendetes UML Signal repräsentiert. Das Signal wird wie eine Operation visualisiert.

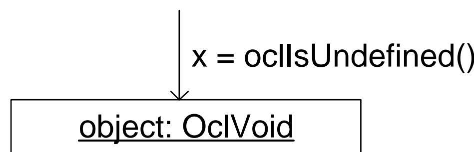
- message.isOperationCall(): Boolean



isOperationCall() liefert true, wenn die OclMessage einen gesendeten UML Operationsaufruf repräsentiert. Dies wird wie eine Operation visualisiert.

3.1.3 OclVoid

- OclUndefined.oclIsUndefined(): Boolean

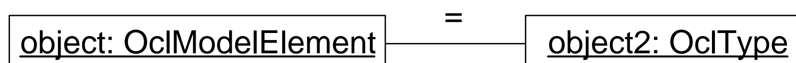


oclIsUndefined() ist eine Operation auf einem Objekt des Typs OclVoid. Sie liefert true, wenn der Typ des Objekts äquivalent zu OclUndefined ist. Die Operation wird durch einen Pfeil auf das Objekt visualisiert, der als Beschriftung oclIsUndefined() enthält.

3.2 ModelElement Typen

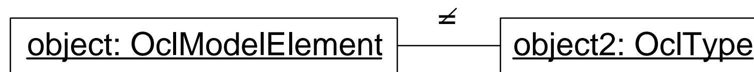
3.2.1 OclModelElement

- object = (object2: OclType): Boolean



Das Überprüfen der Gleichheit eines Objektes vom Typ OclModelElement und eines vom Typ OclType wird durch einen Link mit einem Gleichheitszeichen zwischen den Objekten visualisiert.

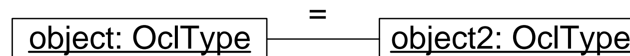
- object <> (object2: OclType): Boolean



Das Überprüfen der Ungleichheit eines Objektes vom Typ OclModelElement und eines vom Typ OclType wird durch einen Link mit einem ≠-Zeichen zwischen den Objekten visualisiert.

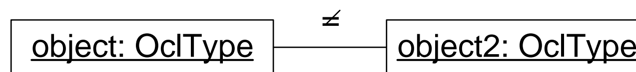
3.2.2 OclType

- object = (object2: OclType): Boolean



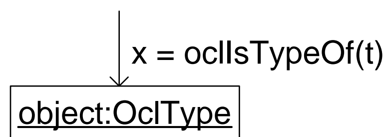
Das Überprüfen der Gleichheit zweier Objekte des Typs OclType wird durch einen Link mit einem Gleichheitszeichen zwischen den Objekten visualisiert.

- object <> (object2: OclType): Boolean



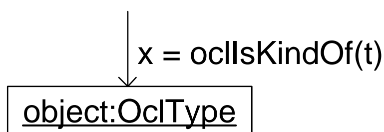
Das Überprüfen der Ungleichheit zweier Objekte des Typs OclType wird durch einen Link mit einem ≠-Zeichen zwischen den Objekten visualisiert.

- oclIsTypeOf(t: OclType): Boolean



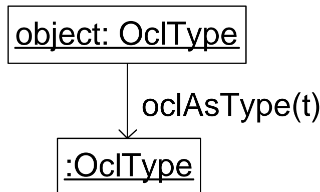
Die Operation *oclIsTypeOf()* liefert ein True, wenn *t* und *object* vom gleichen Type sind. Die Operation wird durch einen Pfeil auf das Objekt visualisiert, der als Beschriftung oclIsTypeOf() enthält.

- oclIsKindOf(t: OclType): Boolean



Die Operation *oclIsKindOf* bestimmt, ob t entweder vom dem direkten oder vom Sypertypen dieses Objektes ist. Die Operation wird durch einen Pfeil auf das Objekt visualisiert, der als Beschriftung oclIsKindOf() enthält.

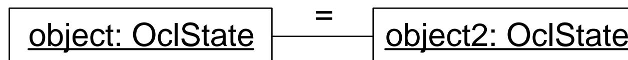
- oclAsType(t: OclType): instance of OclType



Wenn Eigenschaften innerhalb eines Types neu zu definieren sind, kann durch die Verwendung der Operation *oclAsType()* auf die eigenschaften von Sypertypen zugegriffen werden. Die Operation wird durch einen Pfeil auf das Objekt visualisiert, der als Beschriftung oclAsType() enthält.

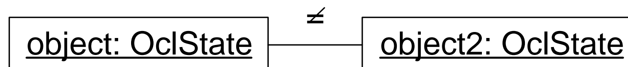
3.2.3 OclState

- object = (object2: OclState): Boolean



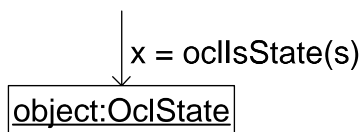
Das Überprüfen der Gleichheit zweier Objekte des Typs OclState wird durch einen Link mit einem Gleichheitszeichen zwischen den Objekten visualisiert.

- object <> (object2: OclState): Boolean



Das Überprüfen der Ungleichheit zweier Objekte des Typs OclState wird durch einen Link mit einem $\neq$-Zeichen zwischen den Objekten visualisiert.

- oclInState(s: OclState):Boolean



Die Operation *oclInState(object)* liefert ein True wenn das Objekt im Zusatand s ist. s ist der Name eines Zustandes in einem Zusatandsdiagramm. Die Operation wird durch einen Pfeil auf das Objekt visualisiert, der als Beschriftung oclInState() enthält.

3.3 Einfache Typen

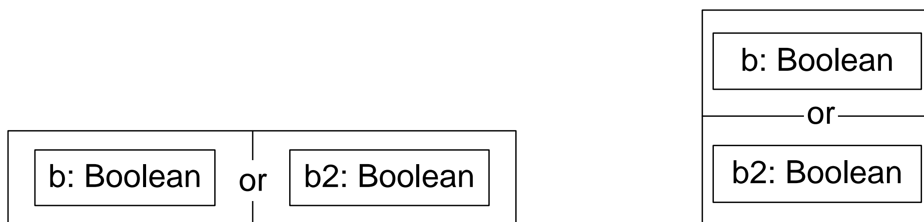
3.3.1 Real, Integer und String

Hier werden die mathematischen Konzepte der reellen, der natürlichen Zahlen und der Strings repräsentiert. Da sie nur textuell in der Bedingung eines Constraints verwendet werden, werden sie nicht visualisiert.

3.3.2 Boolean

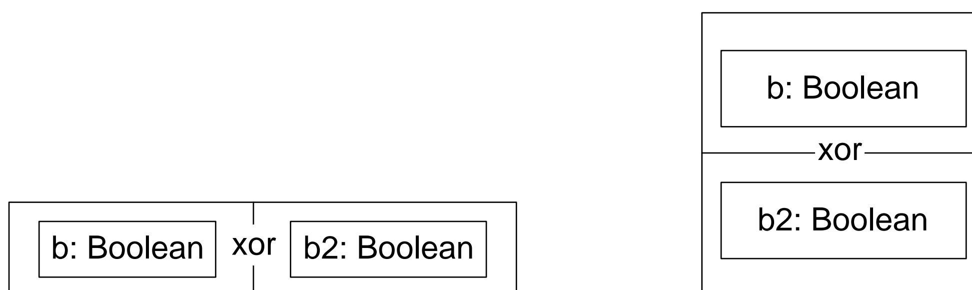
Dieser Typ kann sowohl visualiert werden, als auch textuell in dem Bedingungsteil eines Constraints verwendet werden. Im folgenden wird immer die textuelle Darstellung, gefolgt von der visuellen, angegeben.

- b or (b2: Boolean): Boolean



Eine Verknüpfung durch *or* wird durch einen eigenen Or-Rahmen dargestellt, die Aussagen links und rechts, bzw. ober- und unterhalb vom *or* werden miteinander verodert.

- b xor (b2: Boolean): Boolean

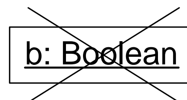


Eine Verknüpfung durch *xor* wird durch einen eigenen Xor-Rahmen dargestellt, die Aussagen links und rechts, bzw. ober- und unterhalb vom *xor* werden exclusiv miteinander verodert.

- b and (b2: Boolean): Boolean

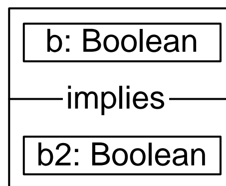
Die Verknüpfung mehrerer Teilaussagen durch *and* wird nicht extra visualisiert, zusammengruppierte-Teilaussagen werden automatisch durch *and* verknüpft.

- not b: Boolean



Die Negierung einer Aussage wird visualisiert, indem die Aussage durchgestrichen wird.

- b implies (b2: Boolean): Boolean



Ein Implies-Ausdruck wird in einem Implies-Rahmen visualisiert. Alles über dem *implies* beschreibt die Bedingung, die, wenn sie erfüllt ist, den unteren Teil impliziert.

3.4 Generelle Operationen auf Collections

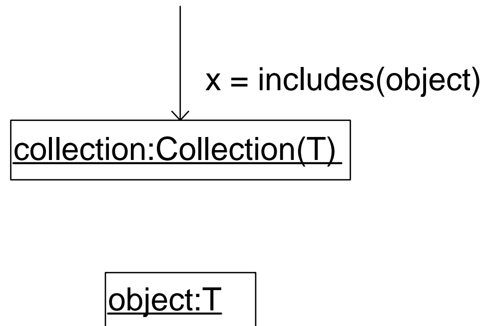
3.4.1 Collection

Dieser Abschnitt enthält alle Operationen, die auf jeden der Collection-Typen, d.h. Set, Bag und Sequence angewendet werden können. In dem jeweiligen Fall muss der Rahmen für Collection durch den Rahmen des jeweiligen Typs ersetzt werden.

- collection->size(): Integer

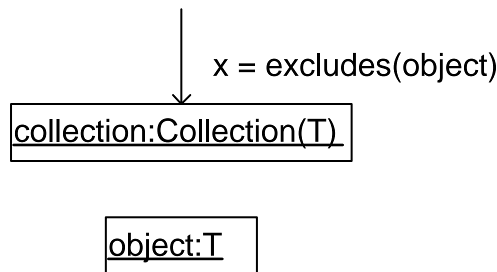
Der *size*-Operator wird auf eine Collection angewendet. In der Varablen n steht die Anzahl der Elemente, die sich in der Collection befinden.

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



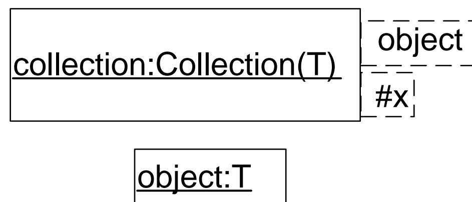
Die Operation *includes()* gibt einen booleschen Wert zurück, der ausdrückt, ob in der Collection collection das Objekt object enthalten ist. Dies wird durch einen Operationaufruf mit dem Text x=includes(object) visualisiert. In x steht dann true oder false.

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



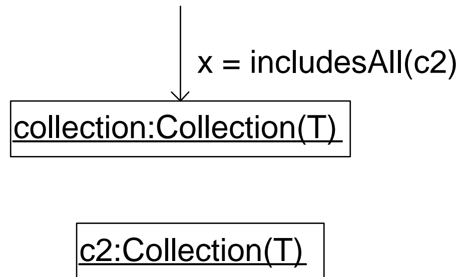
excludes() gibt einen booleschen Wert zurück, der ausdrückt, ob die Collection collection das Objekt object nicht enthält. Dies wird durch einen Operationaufruf mit dem Text x=excludes(object) visualisiert. In x steht dann true oder false.

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



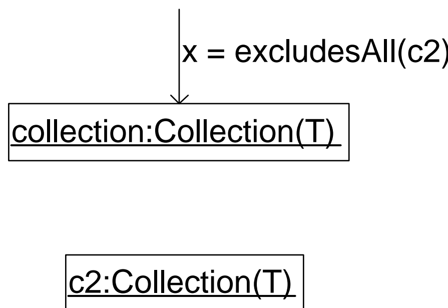
Die Operation *count()* gibt zurück, wie oft das Objekt object in colletion enthalten ist. In der Varablen *x* steht die Anzahl der Elemente, die sich in der Collection befinden. Dies wird durch einen Count-Rahmen, an dem #x und darüber das zu zählende Object steht, dargestellt.

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



Der Rückgabewert von *includesAll()* ist true, wenn alle Elemente der Collection c2 in der Collection collection enthalten sind.

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



Der Rückgabewert von *excludesAll()* ist true, wenn kein Element der Collection c2 in der Collection collection enthalten ist.

- collection->isEmpty(): Boolean

Die *isEmpty()*-Operation wird auf eine Collection angewendet und heißt, dass die Collection kein Element enthält. Die Visualisierung orientiert sich an der mathematischen Mengendarstellung (und überprüft, ob eine Collection gleich der leeren Menge ist).

- collection->notEmpty(): Boolean

Die *notEmpty()*-Operation heißt, dass die Collection mindestens ein Element enthält. Die Visualisierung orientiert sich an der mathematischen Mengendarstellung (und überprüft, ob eine Collection ungleich der leeren Menge ist).

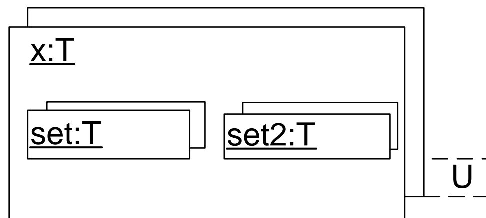
- collection->sum(): T

Die *sum()*-Operation summiert die Werte aller Elemente in einer Collection auf. Hierzu wird über dem mathematischen Summenzeichen das Element(x), über das summiert werden soll, notiert.

In den folgenden Kapiteln sind die Operationen, die für den jeweiligen Collection-Typ typisch sind bzw. die nicht für alle drei Typen gelten, visualisiert.

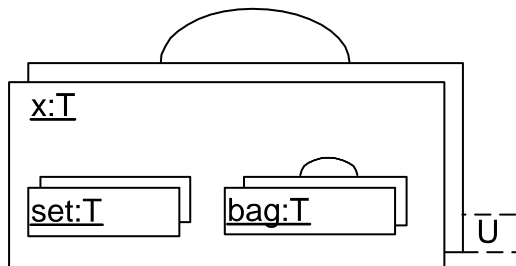
3.4.2 Set

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



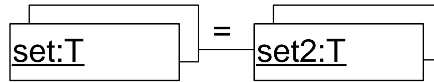
Bei der Vereinigung zweier Sets wird der Name des resultierenden Sets oben links in den Vereinigungs-Rahmen geschrieben. Die beiden Sets innerhalb des Rahmens werden miteinander vereinigt, was durch das mathematische Vereinigungszeichen dargestellt wird.

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



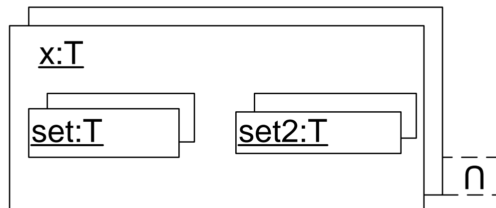
Bei der Vereinigung eines Sets mit einem Bag wird der Name des resultierenden Bags oben links in den Vereinigungs-Rahmen geschrieben. Das Set und der Bag innerhalb des Rahmens werden miteinander vereinigt, was durch das mathematische Vereinigungszeichen dargestellt wird.

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



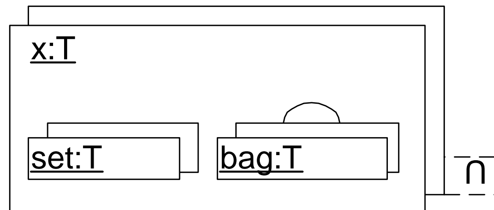
Das Überprüfen der Gleichheit zweier Sets wird durch einen Link mit einem Gleichheitszeichen zwischen den Sets visualisiert.

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



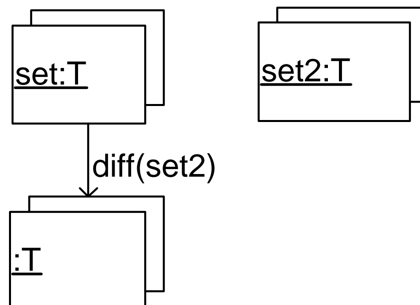
Bei der Bildung der Schnittmenge zweier Sets wird der Name des resultierenden Sets oben links in den Schnittmengen-Rahmen geschrieben. Von den Sets innerhalb des Rahmens wird die Schnittmenge gebildet, was durch das mathematische Schnittmengenzeichen dargestellt wird.

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



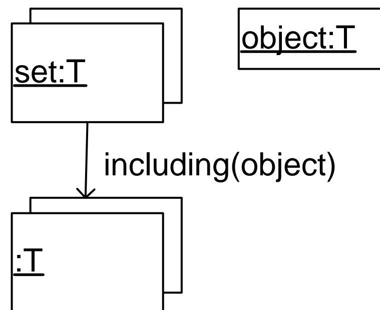
Bei der Bildung der Schnittmenge eines Sets und eines Bags wird der Name des resultierenden Sets oben links in den Schnittmengen-Rahmen geschrieben. Von dem Set und dem Bag innerhalb des Rahmens wird die Schnittmenge gebildet, was durch das mathematische Schnittmengenzeichen dargestellt wird.

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



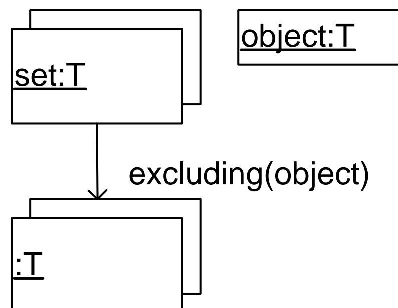
Das Bilden der Differenz zweier Sets resultiert in einem neuen Set, was durch einen Pfeil zwischen dem Set set, von dem das zweite Set set2 abgezogen werden soll, und dem Ergebnisset dargestellt wird.

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



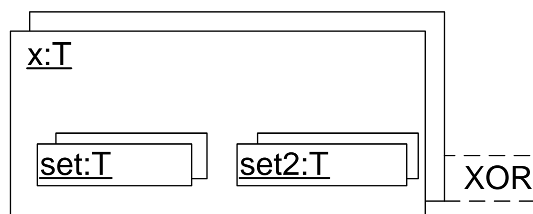
including() vereinigt das Set mit einem Objekt, dies wird durch einen Operationpfeil mit dem Text including(object) zwischen dem Ursprungs-Set und dem Ergebnis-Set visualisiert.

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



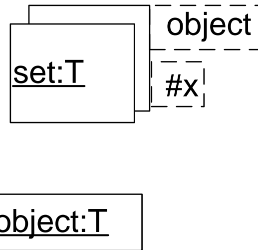
excluding() entfernt ein Objekt aus einem Set, dies wird durch einen Operationspfeil mit dem Text excluding(object) zwischen dem Ursprungs-Set und dem Ergebnis-Set visualisiert.

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



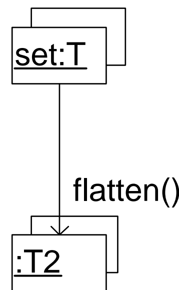
Bei der Bildung der symmetrischen Differenz zweier Sets wird der Name des resultierenden Sets oben links in den Rahmen geschrieben. Von dem linken Set innerhalb des Rahmens wird das rechte Set abgezogen, was durch XOR für die symmetrische Differenz dargestellt wird.

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



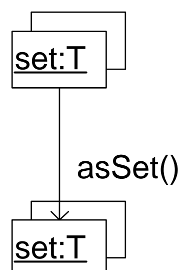
Die Operation *count()* gibt zurück, wie oft das Objekt object in dem Set enthalten ist. In der Variablen x steht die Anzahl der Elemente, die sich in dem Set befinden. Dies wird durch Count-Rahmen, an dem #x und darüber das zu zählende Objekt steht.

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



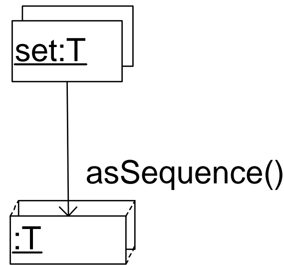
Die Operation *flatten()* auf ein Set angewandt ergibt ein Set, das eventuell Elemente eines anderen Typs enthält als das Set selbst und wird durch einen Link zwischen dem Set und dem Ergebnis-Set mit einem Pfeilende zum Ergebnis-Set visualisiert.

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



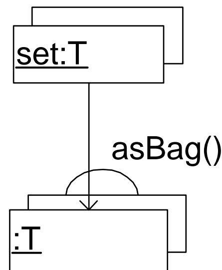
Die Operation *asSet()* auf ein Set angewandt ergibt dasselbe Set und wird durch einen Link zwischen dem Set und dem Ergebnis-Set mit einem Pfeilende zum Ergebnis-Set visualisiert.

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



Die Operation *asSequence()* auf ein Set angewandt ergibt eine Sequence und wird durch einen Link zwischen dem Set und der Ergebnis-Sequence mit einem Pfeilende zur Ergebnis-Sequence visualisiert.

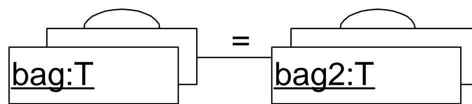
- set->asBag(): Bag(T)



Die Operation *asBag()* auf ein Set angewandt ergibt einen Bag und wird durch einen Link zwischen dem Set und dem Ergebnis-Bag mit einem Pfeilende zum Ergebnis- Bag visualisiert.

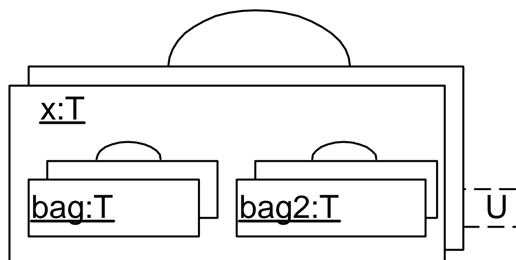
3.4.3 Bag

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



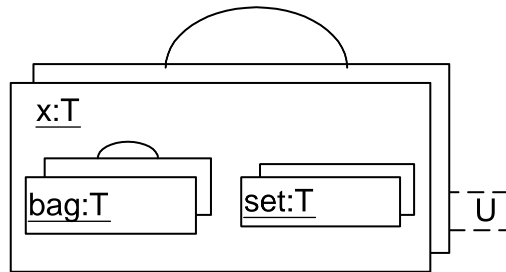
Das Überprüfen der Gleichheit zweier Bags wird durch einen Link mit einem Gleichheitszeichen zwischen den Bags visualisiert.

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



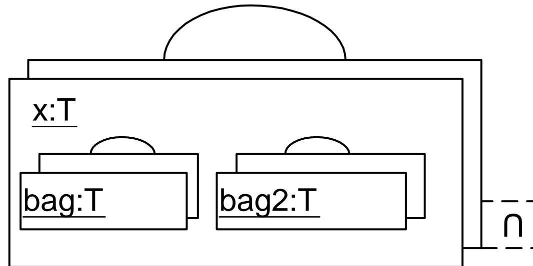
Bei der Vereinigung zweier Bags wird der Name des resultierenden Bags oben links in den Vereinigungs-Rahmen geschrieben. Die beiden Bags innerhalb des Rahmens werden miteinander vereinigt, was durch das mathematische Vereinigungszeichen dargestellt wird.

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



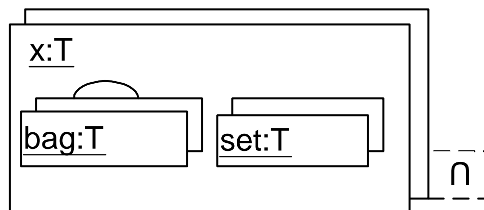
Bei der Vereinigung eines Bags mit einem Set wird der Name des resultierenden Bags oben links in den Vereinigungs-Rahmen geschrieben. Der Bag und das Set innerhalb des Rahmens werden miteinander vereinigt, was durch das mathematische Vereinigungszeichen dargestellt wird.

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



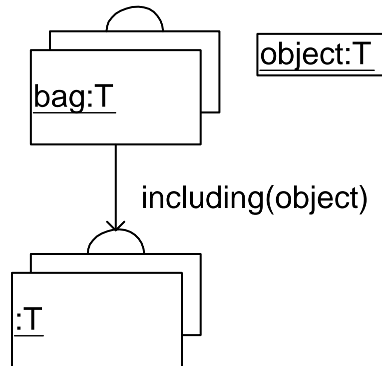
Bei der Bildung der Schnittmenge zweier Bags wird der Name des resultierenden Bags oben links in den Schnittmengen-Rahmen geschrieben. Von den Bags innerhalb des Rahmens wird die Schnittmenge gebildet, was durch das mathematische Schnittmengenzeichen dargestellt wird.

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



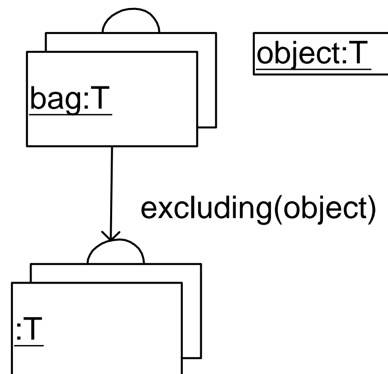
Bei der Bildung der Schnittmenge eines Bags mit einem Set wird der Name des resultierenden Sets oben links in den Schnittmengen-Rahmen geschrieben. Von dem Bag und dem Set innerhalb des Rahmens wird die Schnittmenge gebildet, was durch das mathematische Schnittmengenzeichen dargestellt wird.

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



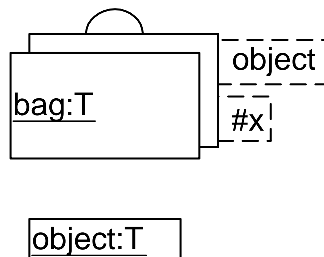
including() vereinigt den Bag mit einem Objekt, dies wird durch einen Operationspfeil mit dem Text including(object) zwischen dem Ursprungs-Set und dem Ergebnis-Set visualisiert.

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



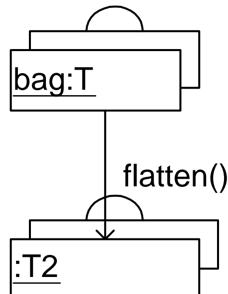
excluding() entfernt ein Objekt aus einem Bag, dies wird durch einen Operationspfeil mit dem Text excluding(object) zwischen dem Ursprungs-Set und dem Ergebnis-Set visualisiert.

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



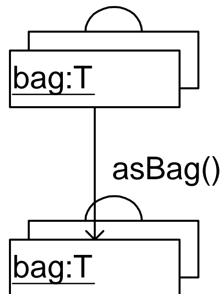
Die Operation *count()* gibt zurück, wie oft das Objekt object in dem Bag enthalten ist. In der Variablen x steht die Anzahl der Elemente, die sich in dem Bag befinden. Dies wird durch Count-Rahmen, an dem #x und darüber das zu zählende Objekt steht.

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



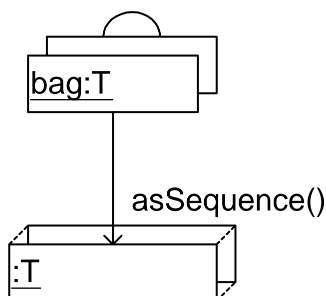
Die Operation *flatten()* auf einen Bag angewandt ergibt einen Bag, der eventuell Elemente eines anderen Typs enthält als der Bag selbst und wird durch einen Link zwischen dem Bag und dem Ergebnis-Bag mit einem Pfeilende zum Ergebnis-Bag visualisiert.

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



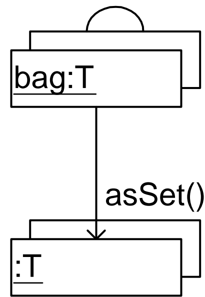
Die Operation *asBag()* auf einen Bag angewandt, ergibt denselben Bag und wird durch einen Link zwischen dem Bag und dem Ergebnis-Bag mit einem Pfeilende zum Ergebnis-Bag visualisiert.

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



Die Operation *asSequence()* auf einen Bag angewandt ergibt eine Sequence und wird durch einen Link zwischen dem Set und der Ergebnis-Sequence mit einem Pfeilende zur Ergebnis-Sequence visualisiert.

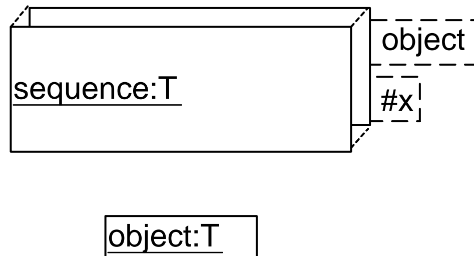
- bag->asSet(): Set()



Die Operation *asSet()* auf einen Bag angewandt ergibt ein Set und wird durch einen Link zwischen dem Bag und dem Ergebnis-Set mit einem Pfeilende zum Ergebnis-Set visualisiert.

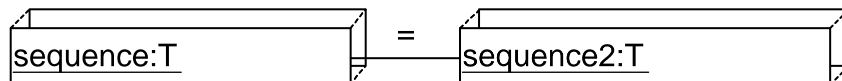
3.4.4 Sequence

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



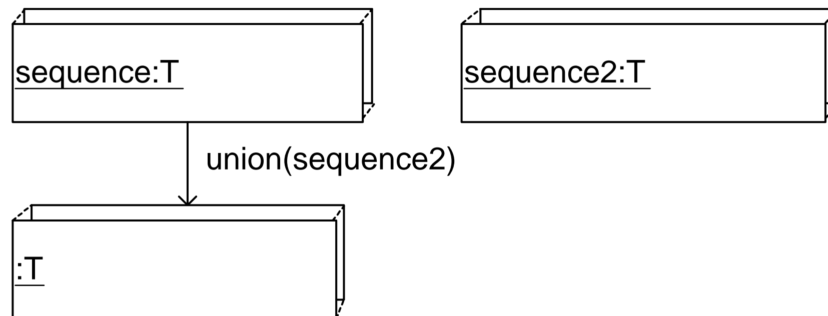
Die Operation *count()* gibt zurück, wie oft das Objekt object in der Sequence enthalten ist. In der Variablen x steht die Anzahl der Elemente, die sich in der Sequence befinden. Dies wird durch Count-Rahmen, an dem #x und darüber das zu zählende Objekt steht.

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



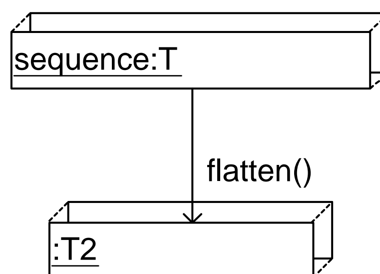
Das Überprüfen der Gleichheit zweier Sequences wird durch einen Link mit einem Gleichheitszeichen zwischen den Sequences visualisiert.

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



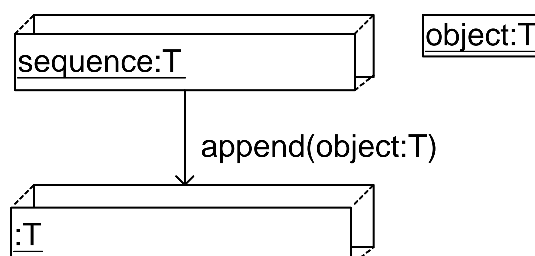
Bei der Vereinigung einer Sequence sequence mit einer anderen Sequence sequence2 wird die Operation union(sequence2) auf die Sequence sequence angewendet. Das Resultat ist eine neue Sequence, die durch das Anhängen von sequence2 an sequence entsteht.

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



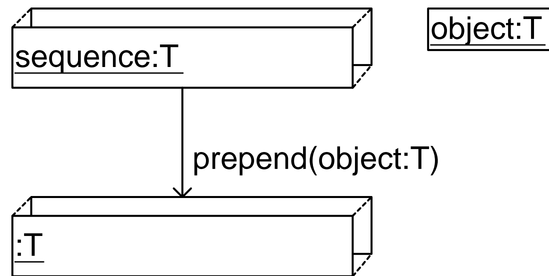
Die Operation *flatten()* auf eine Sequence angewandt ergibt eine Sequence, die eventuell Elemente eines anderen Typs enthält als die Sequence selbst und wird durch einen Link zwischen der Sequence und der Ergebnis-Sequence mit einem Pfeilende zur Ergebnis-Sequence visualisiert.

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



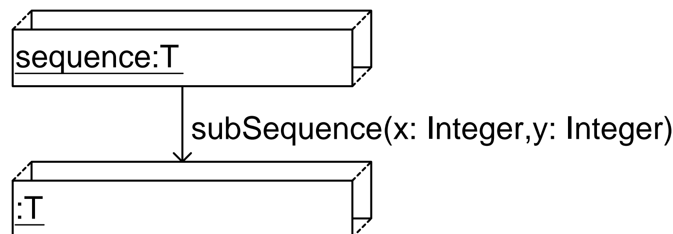
Die Operation *append()* hängt das Objekt object an das Ende der Sequence an. Dies wird durch einen Operationspfeil mit dem Text append(object:T) zwischen der Ursprungs-Sequence und der Ergebnis-Sequence visualisiert.

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



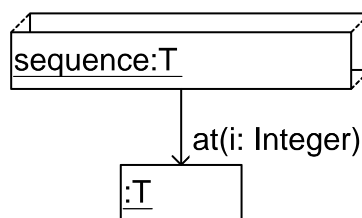
Die Operation *prepend()* hängt das Objekt object an den Anfang der Sequence an. Dies wird durch einen Operationspfeil mit dem Text prepend(object:T) zwischen der Ursprungs-Sequence und der Ergebnis-Sequence visualisiert.

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



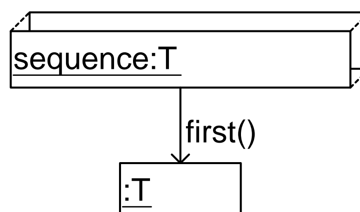
Die Operation *subSequence(x, y)* ergibt eine Sequence, die alle Objekte zwischen den Indizes x und y aus der Ursprungs-Sequence beinhaltet, was durch einen Link mit einem Pfeilende zur Ergebnis-Sequence dargestellt wird.

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



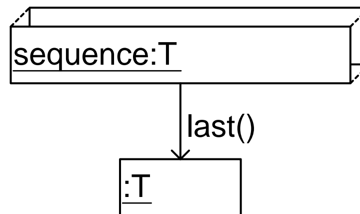
Die Operation *at(i)* wird auf eine Sequence angewendet und gibt das Element am Index i zurück, was durch einen Link mit einem Pfeilende zum Ergebnis-Objekt dargestellt wird.

- sequence->first(): T



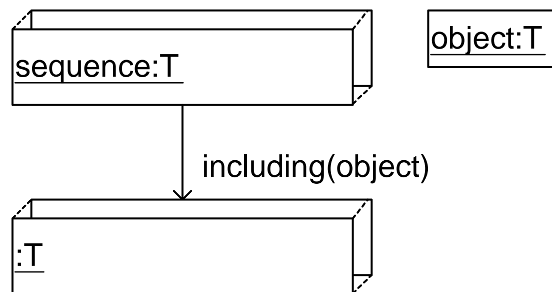
Die Operation *first()* gibt das erste Element der Sequence zurück, was durch einen Link mit einem Pfeilende zum Ergebnis-Objekt dargestellt wird.

- sequence->last(): T



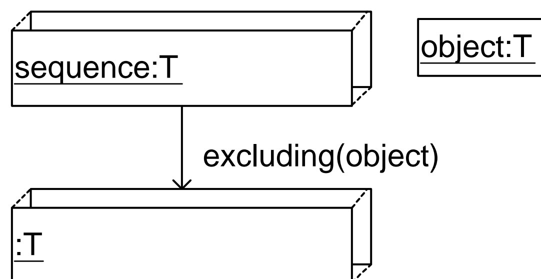
Die Operation *last()* gibt das letzte Element der Sequence zurück, was durch einen Link mit einem Pfeilende zum Ergebnis-Objekt dargestellt wird.

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



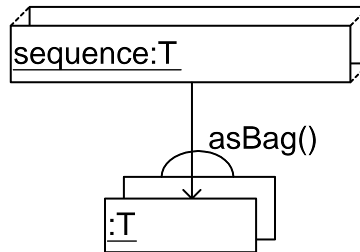
including() vereinigt die Sequence mit einem Objekt, dies wird durch einen Operationspfeil mit dem Text including(object:T) zwischen der Ursprungs-Sequence und der Ergebnis-Sequence visualisiert

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



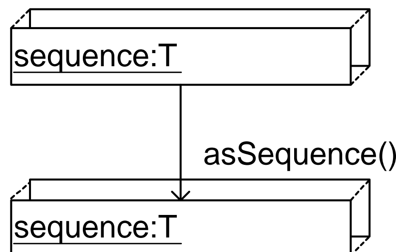
excluding() entfernt ein Objekt aus einer Sequence, dies wird durch einen Operationspfeil mit dem Text excluding(object:T) zwischen der Ursprungs-Sequence und der Erbegnis-Sequence visualisiert.

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



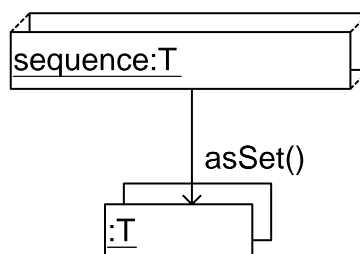
Die Operation *asBag()* auf eine Sequence angewandt ergibt einen Bag und wird durch einen Link zwischen der Sequence und dem Ergebnis-Bag mit einem Pfeilende zum Ergebnis-Bag visualisiert.

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



Die Operation *asSequence()* auf eine Sequence angewandt ergibt dieselbe Sequence und wird durch einen Link zwischen der Sequence und der Ergebnis-Sequence mit einem Pfeilende zur Ergebnis-Sequence visualisiert.

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



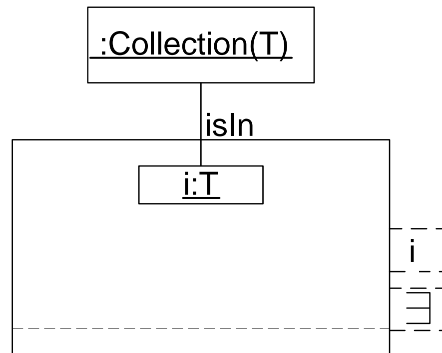
Die Operation *asSet()* auf eine Sequence angewandt ergibt ein Set und wird durch einen Link zwischen der Sequence und dem Ergebnis-Set mit einem Pfeilende zum Ergebnis-Set visualisiert.

3.5 Vordefinierte Ocl-Iteratorbibliothek

3.5.1 Collection

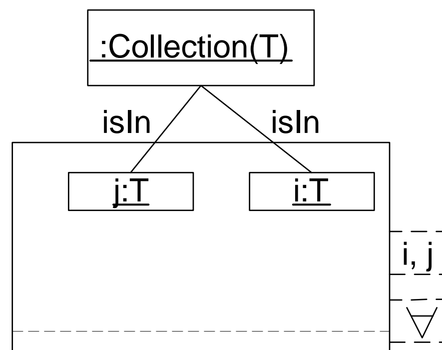
Die folgenden Operation gelten für alle Collection-Typen und werden aus diesem Grunde zusammengefasst. Der Typ der Elemente der jeweiligen Collection ist T. Der Collection-Rahmen muss durch einen Rahmen des jeweiligen Typs ersetzt werden.

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



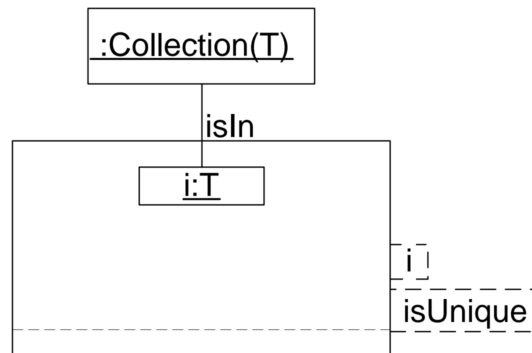
Die *Exists*-Operation wird auf eine Collection angewendet, diese Operation hat einen Iterator und einen Body. Der Ausdruck des Bodys wird in einem Rahmen visualisiert, an dessen rechten Rand der/die Iterator/en und der ∃-Operator notiert werden. Der Rückgabewert dieser Operation ist vom Typ Boolean, die Bedingungen im Body müssen für mindestens ein Element in der Collection erfüllt sein. Dass der Ausdruck im Body auf ein Element aus der Collection angewendet wird, wird mit Hilfe der Operation *isIn* dargestellt. Innerhalb des Rahmens können dann weitere Bedingungen für die Objekte der Collection visualisiert werden, er kann einen Bedingungsteil enthalten.

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



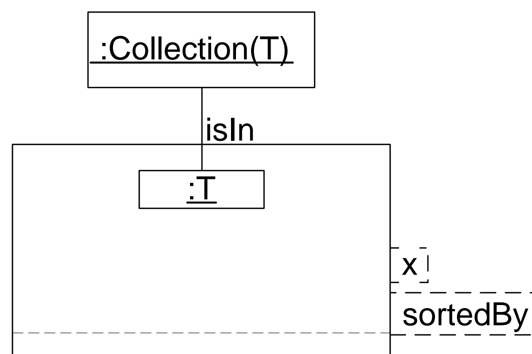
Die *Forall*-Operation wird auf eine Collection angewendet, diese Operation hat einen oder zwei Iteratoren und einen Body. Der Ausdruck des Bodys steht in einem Rahmen, an dessen rechten Rand der/die Iterator/en und der ∀-Operator notiert werden. Der Rückgabewert dieser Operation ist vom Typ Boolean, die Bedingungen im Body müssen für alle Elemente in der Collection erfüllt sein. Dass der Ausdruck im Body auf alle Element aus der Collection angewendet wird, wird mit Hilfe der Operation *isIn* dargestellt. Innerhalb des Rahmens können dann weitere Bedingungen für die Objekte der Collection visualisiert werden, er kann einen Bedingungsteil enthalten.

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



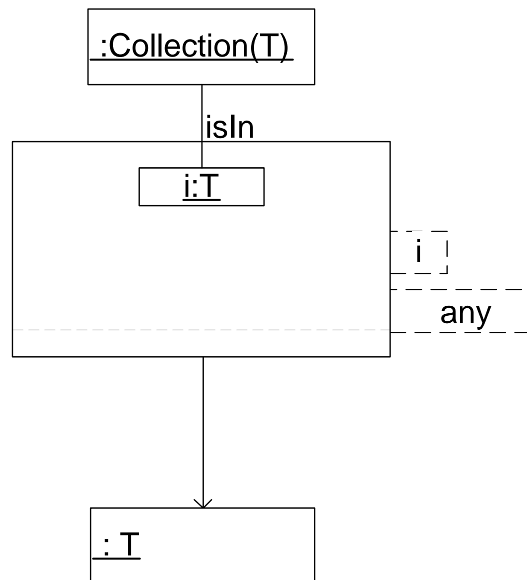
Die *isUnique*-Operation wird auf eine Collection angewendet, diese Operation hat einen oder mehrere Iteratoren und einen Body. Der Ausdruck des Bodys wird in einem Rahmen visualisiert, an dessen rechten Rand der Iterator und isUnique notiert werden. IsUnique ergibt true, wenn alle Elemente in der Collection verschiedene Auswertungen des Ausdrucks im Body haben. Auch hier wird mit der Operation *isIn* gearbeitet.

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



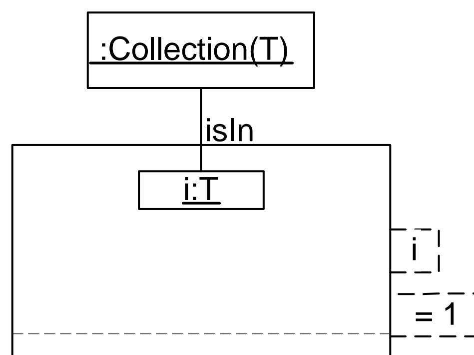
Die *sortedBy*-Operation wird auf eine Collection angewendet, diese Operation hat einen Iterator und einen Body. Der Ausdruck des Bodys wird in einem Rahmen visualisiert, an dessen rechten Rand der Iterator und sortedBy notiert werden. Die Elemente der Collection werden nach dem Iterator sortiert und die sortierte Collection wird zurückgegeben. Im Allgemeinen ist der Rückgabetyp eine Sequence. Auch hier wird mit der Operation *isIn* gearbeitet.

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



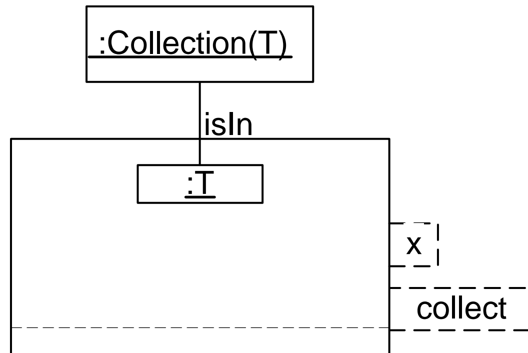
Die *any*-Operation wird auf eine Collection angewendet, diese Operation hat einen Iterator und einen Body. Der Ausdruck des Bodys wird in einem Rahmen visualisiert, an dessen rechten Rand der Iterator und "any" notiert werden. Any liefert irgendein Objekt der Collection, das die Bedingungen des Bodys erfüllt oder ein Objekt vom Typ OclUndefined, falls keins existiert. Auch hier wird mit der Operation *isIn* gearbeitet.

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



Die *one*-Operation wird auf eine Collection angewendet, diese Operation hat einen Iterator und einen Body. Der Ausdruck des Bodys wird in einem Rahmen visualisiert, an dessen rechten Rand der Iterator und "=1" notiert werden. One ergibt true, wenn genau ein Element in der Collection die Bedingungen des Bodys erfüllt. Auch hier wird mit der Operation *isIn* gearbeitet.

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

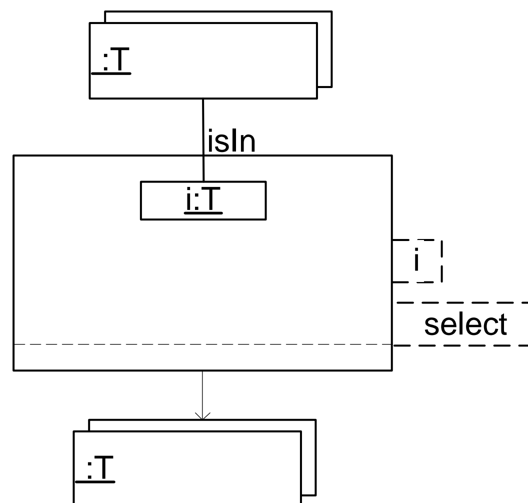


Die *collect*-Operation wird auf eine Collection angewendet, diese Operation hat einen oder mehrere Iteratoren und einen Body. Der Ausdruck des Bodys wird in einem Rahmen visualisiert, an dessen rechten Rand der Iterator und ein collect notiert werden. Alle Elemente, die den Body erfüllen, werden ausgewählt und als Collection des Typs des Iterators zurückgegeben.

Nun folgen die Operationen, die für den jeweiligen Typ gelten, bzw. voneinander abweichende Ergebnisse besitzen.
Alle diese Operationen machen Gebrauch von der *isIn*-Operation, um die Eigenschaften des Bodys zu visualisieren.

3.5.2 Set

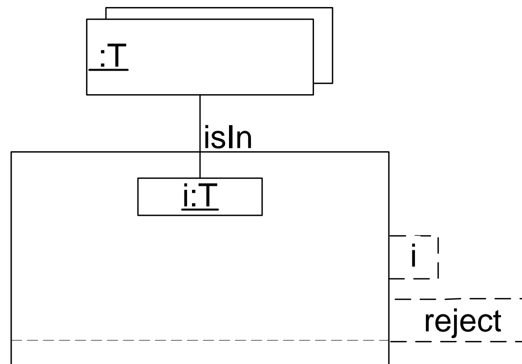
- Set(T)->select(expression: OclExpression)->Set(T)



Die *Select*-Operation auf ein Set angewendet liefert eine Untermenge dieses Sets, deren Elemente bestimmte Eigenschaften haben. Diese Selektionseigenschaften werden im Select-Rahmen spezifiziert, an dessen Rand stehen der Iterator und das Schlüsselwort select. Die Auswertung des Selektionsausdrucks liefert eine Menge vom Typ T.
In obiger Darstellung wurde auch die Rückgabe-Collection (in diesem Fall ein Set) visualisert, worauf in

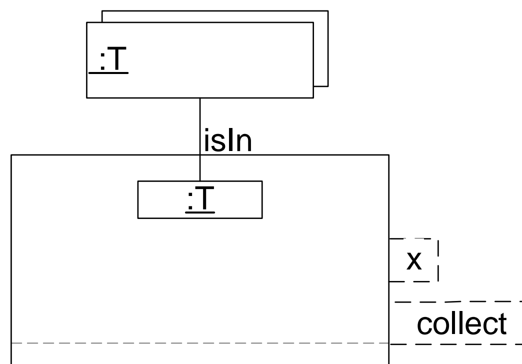
den vorangegangenen und folgenden Visualisierungen zwecks besserer Übersichtlichkeit und leichterem Verständnis der grafischen Darstellungen verzichtet wurde.

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



Die *Reject*-Operation auf ein Set angewendet liefert eine Untermenge dieses Sets, deren Elemente bestimmte Eigenschaften nicht haben. Diese Eigenschaften werden im Reject-Rahmen spezifiziert, an dessen Rand stehen der Iterator und das Schlüsselwort reject. Die Auswertung des Reject-Ausdrucks liefert eine Menge vom Typ T.

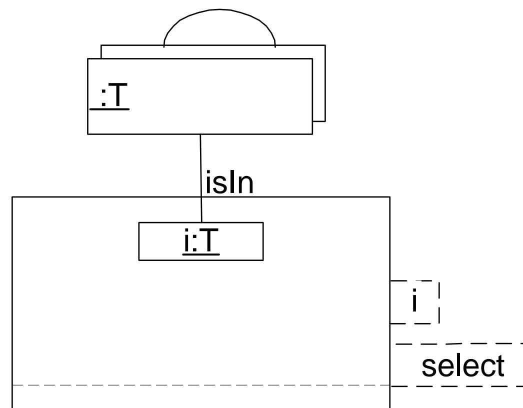
- Set(T)->collectNested(expression: OclExpression)->Bag(T2)



Die *collectNested*-Operation wird genauso visualisiert wie die collect Operation. Der Rückgabewert ist bei der Anwendung auf ein Set ein Bag.

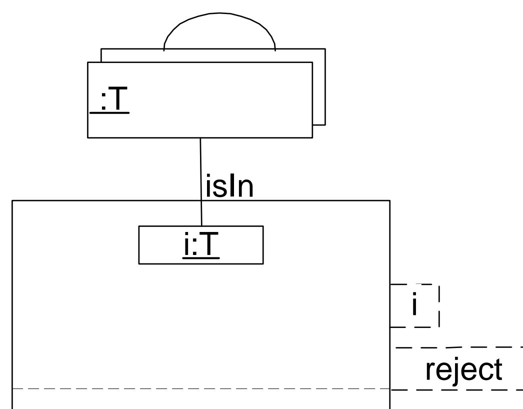
3.5.3 Bag

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



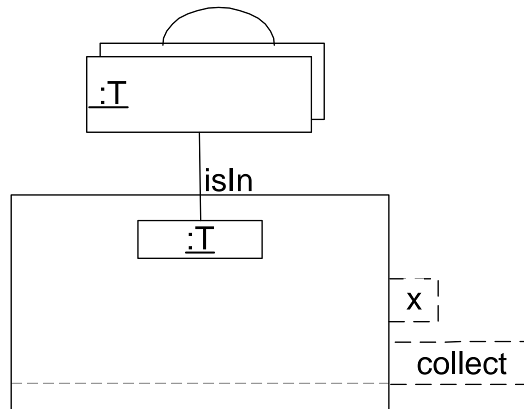
Die *Select*-Operation auf einen Bag angewendet liefert eine Untermenge dieses Bags, deren Elemente bestimmte Eigenschaften haben. Diese Selektionsigenschaften werden im Select-Rahmen spezifiziert, an dessen Rand stehen der Iterator und das Schlüsselwort select. Die Auswertung des Selektionsausdrucks liefert eine Menge vom Typ T.

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



Die *Reject*-Operation auf einen Bag angewendet liefert eine Untermenge dieses Bags, deren Elemente bestimmte Eigenschaften nicht haben. Diese Eigenschaften werden im Reject-Rahmen spezifiziert, an dessen Rand stehen der Iterator und das Schlüsselwort reject. Die Auswertung des Reject-Ausdrucks liefert eine Menge vom Typ T.

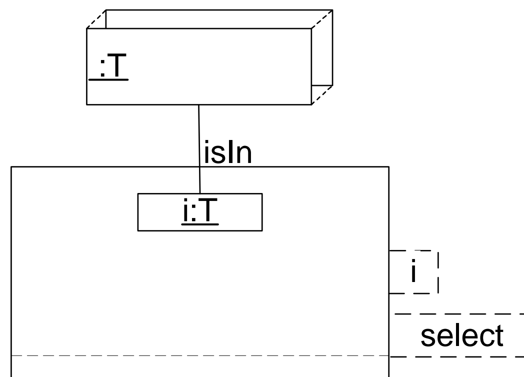
- Bag(T)->collectNested(expression: OclExpression):Bag(T2)



Die *collectNested*-Operation wird genauso visualisiert wie die collect Operation. Der Rückgabewert ist bei der Anwendung auf einen Bag ein Bag.

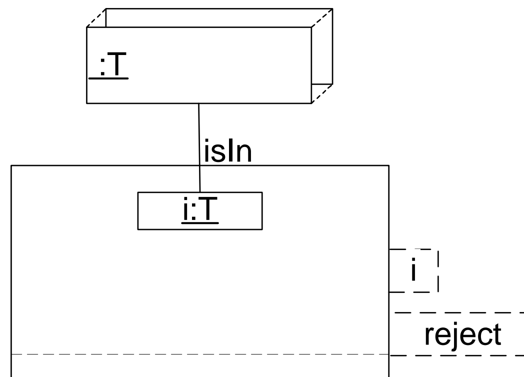
3.5.4 Sequence

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



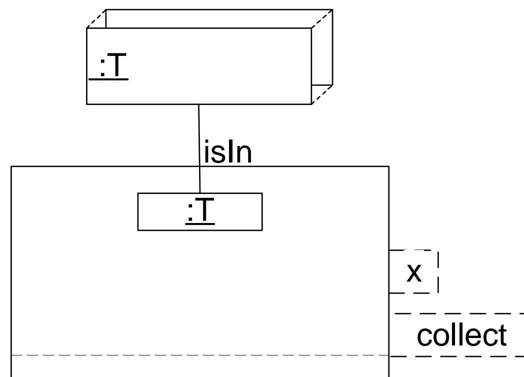
Die *Select*-Operation auf eine Sequence angewendet liefert eine Untermenge dieser Sequence, deren Elemente bestimmte Eigenschaften haben. Diese Selektionsigenschaften werden im Select-Rahmen spezifiziert, an dessen Rand stehen der Iterator und das Schlüsselwort select. Die Auswertung des Selektionsausdrucks liefert eine Menge vom Typ T.

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



Die *Reject*-Operation auf eine Sequence angewendet liefert eine Untermenge dieser Sequence, deren Elemente bestimmte Eigenschaften nicht haben. Diese Eigenschaften werden im Reject-Rahmen spezifiziert, an dessen Rand stehen der Iterator und das Schlüsselwort reject. Die Auswertung des Reject-Ausdrucks liefert eine Menge vom Typ T.

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



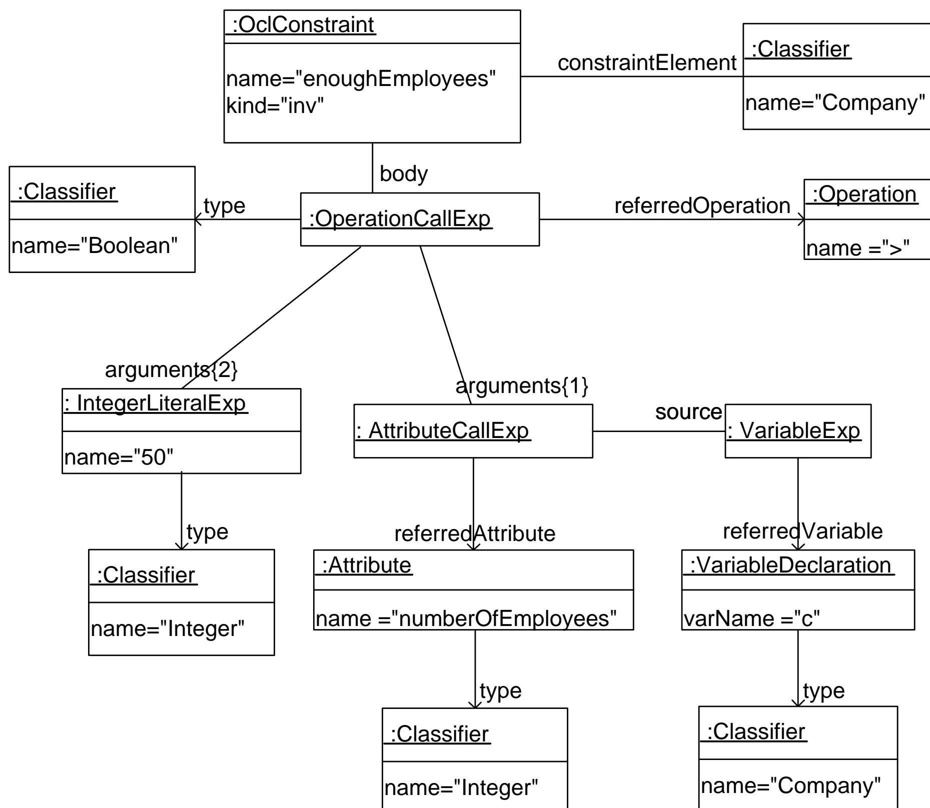
Die *collectNested*-Operation wird genauso visualisiert wie die collect Operation. Der Rückgabewert ist bei der Anwendung auf eine Sequence eine Sequence.

Anhang A

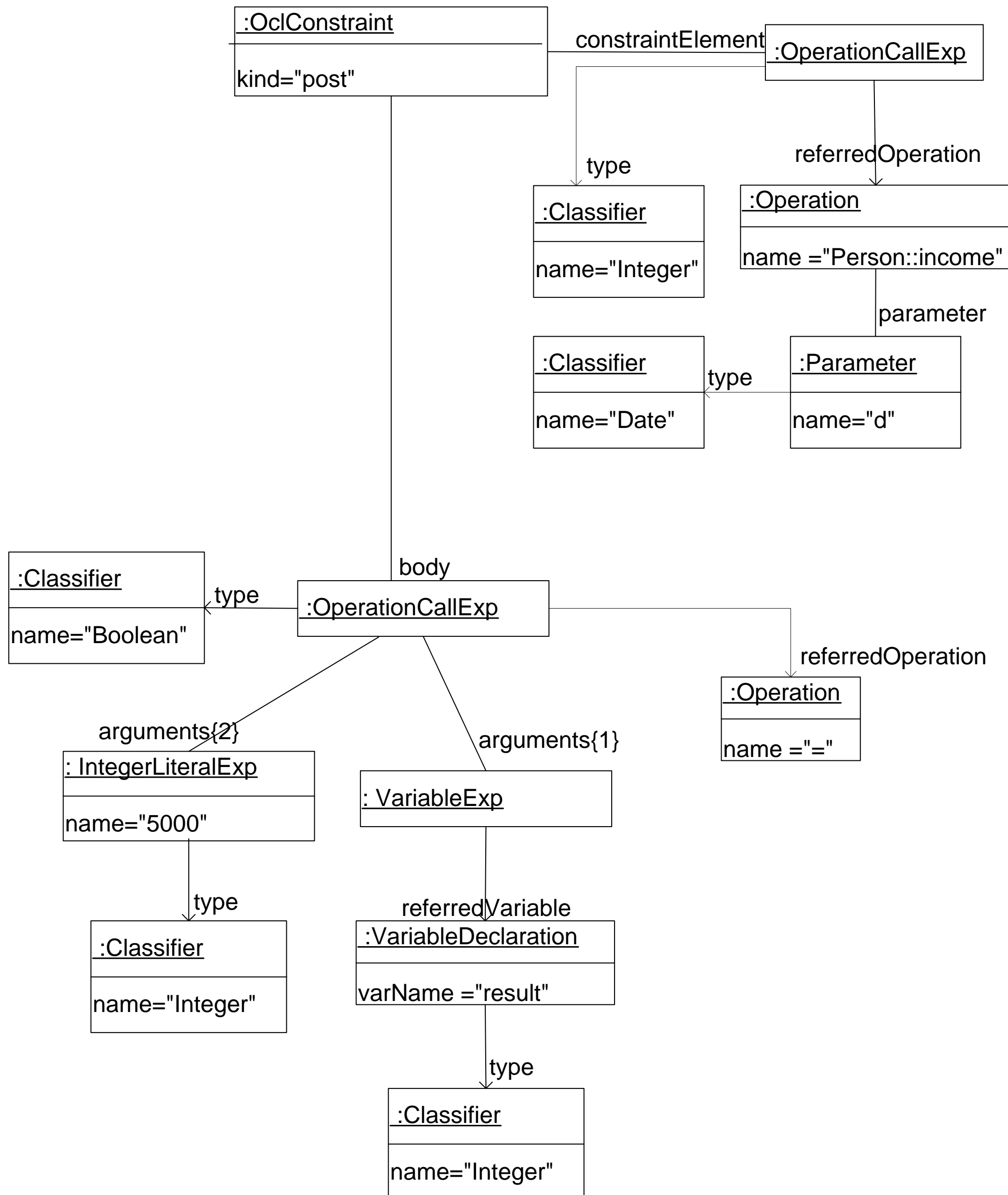
Metamodell–Beispiele

In diesem Abschnitt werden die Metamodelle für einige Constraints aus dem 2.Kapitel dargestellt. Teilweise wurden diese zum besseren Verständnis ein wenig abgeändert, daher steht vor jedem Metamodell noch einmal der dazugehörie textuelle Constraint.
Jede OclExpression benötigt einen Typ. Aus Übersichtsgründen wurden die Angaben zu den Typen in den folgenden Metamodell–Beispielen z.T. weggelassen.

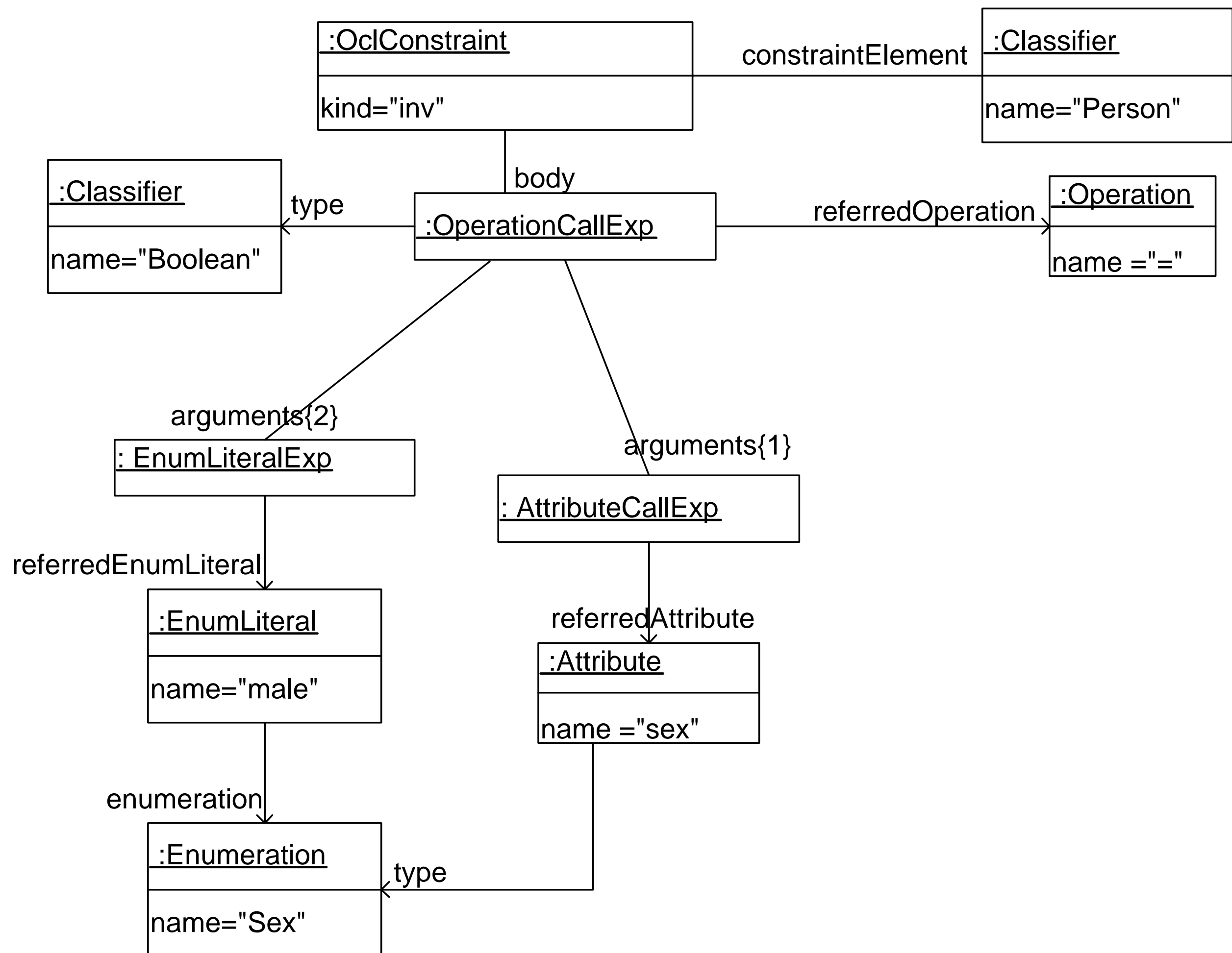
- `context c: Company inv: c.numberOfEmployees > 50`



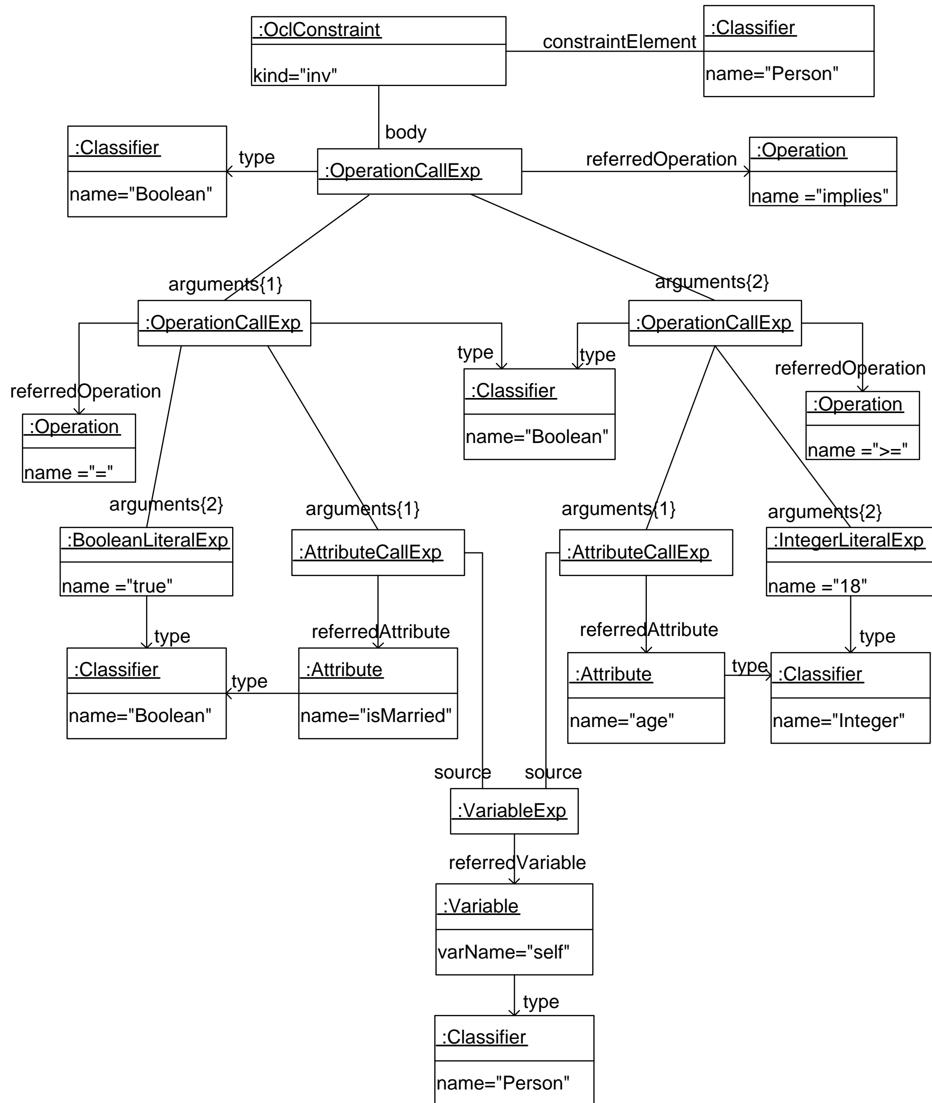
- `context Person::income(d:Date):Integer post: result = 5000`



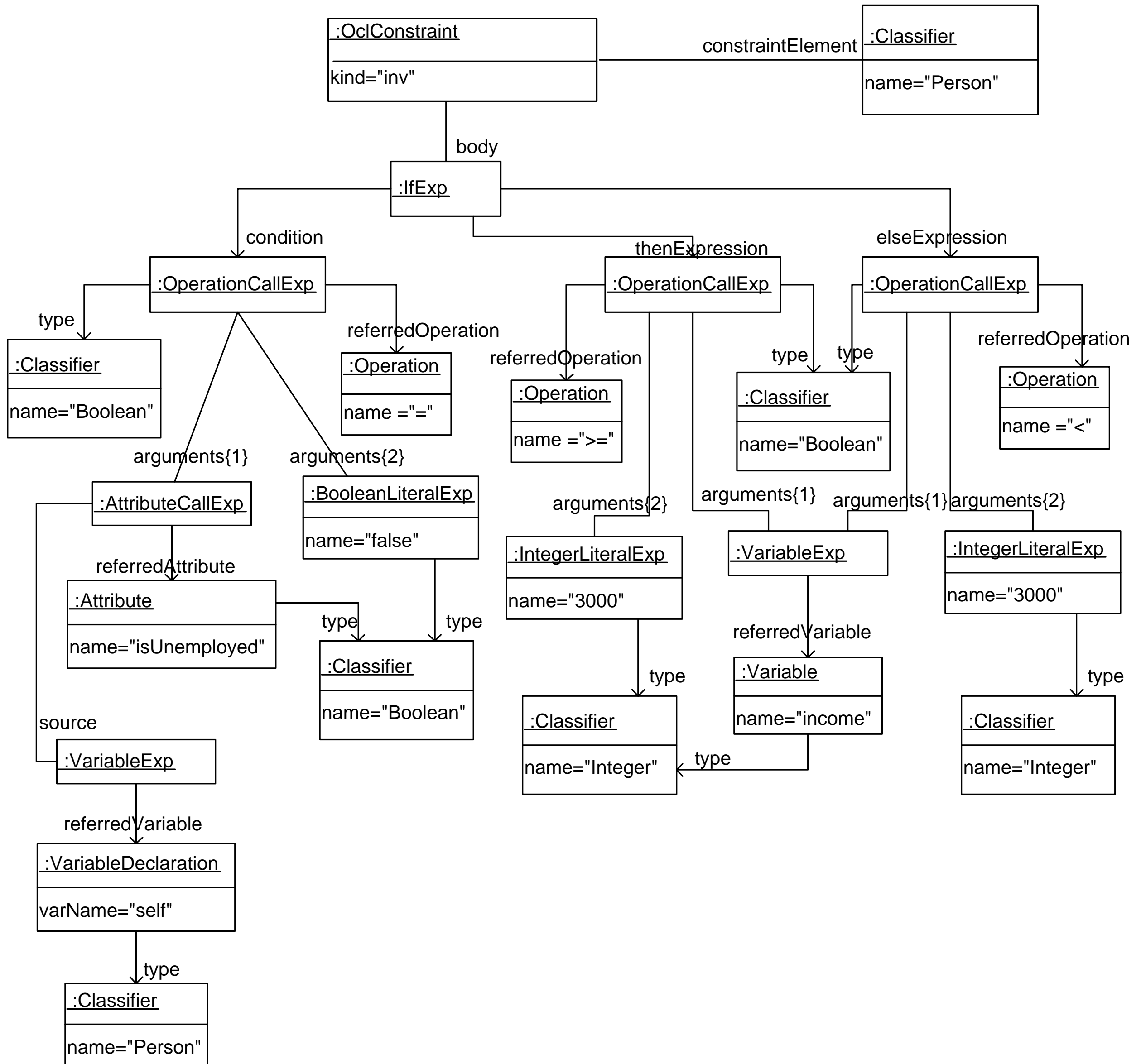
- `context Person inv: sex = Sex::male`



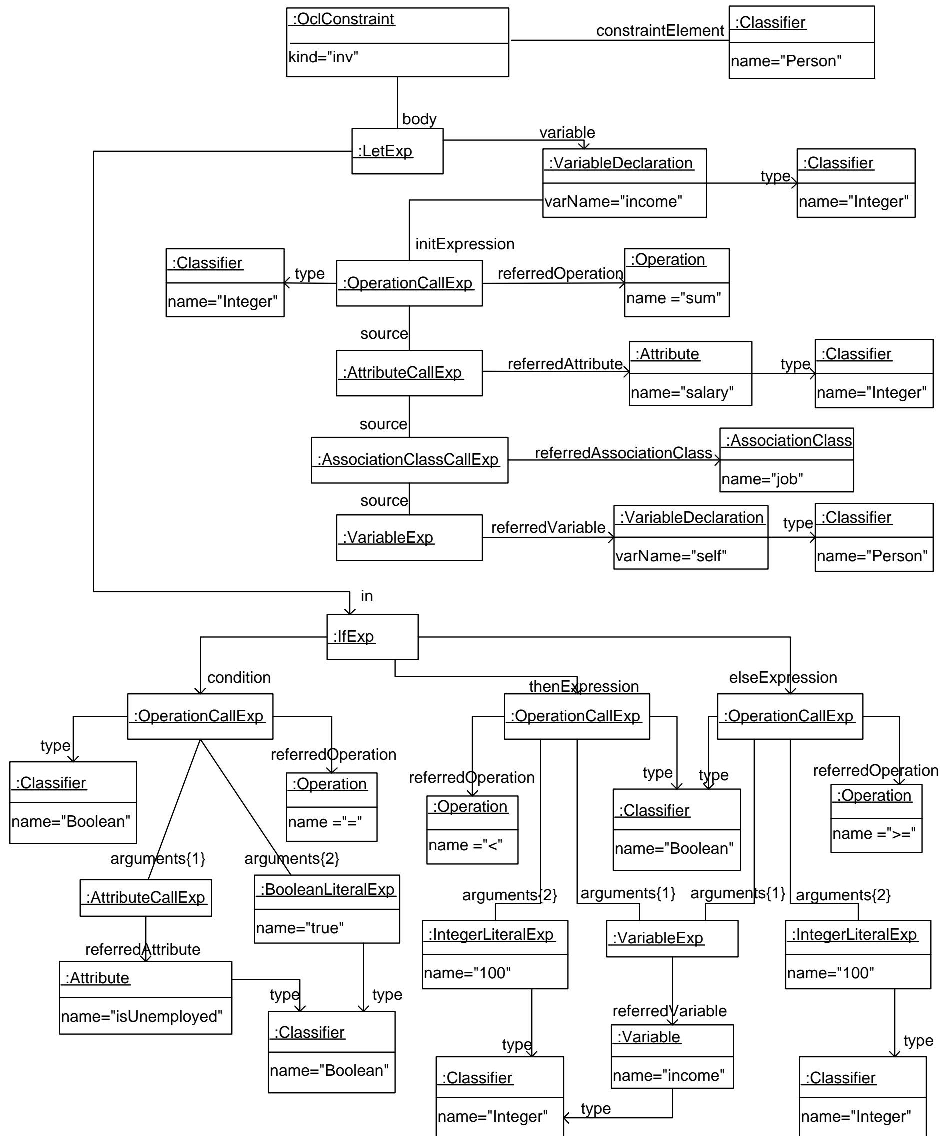
- `context Person inv:`
  `self.isMarried = true implies self.age >= 18`



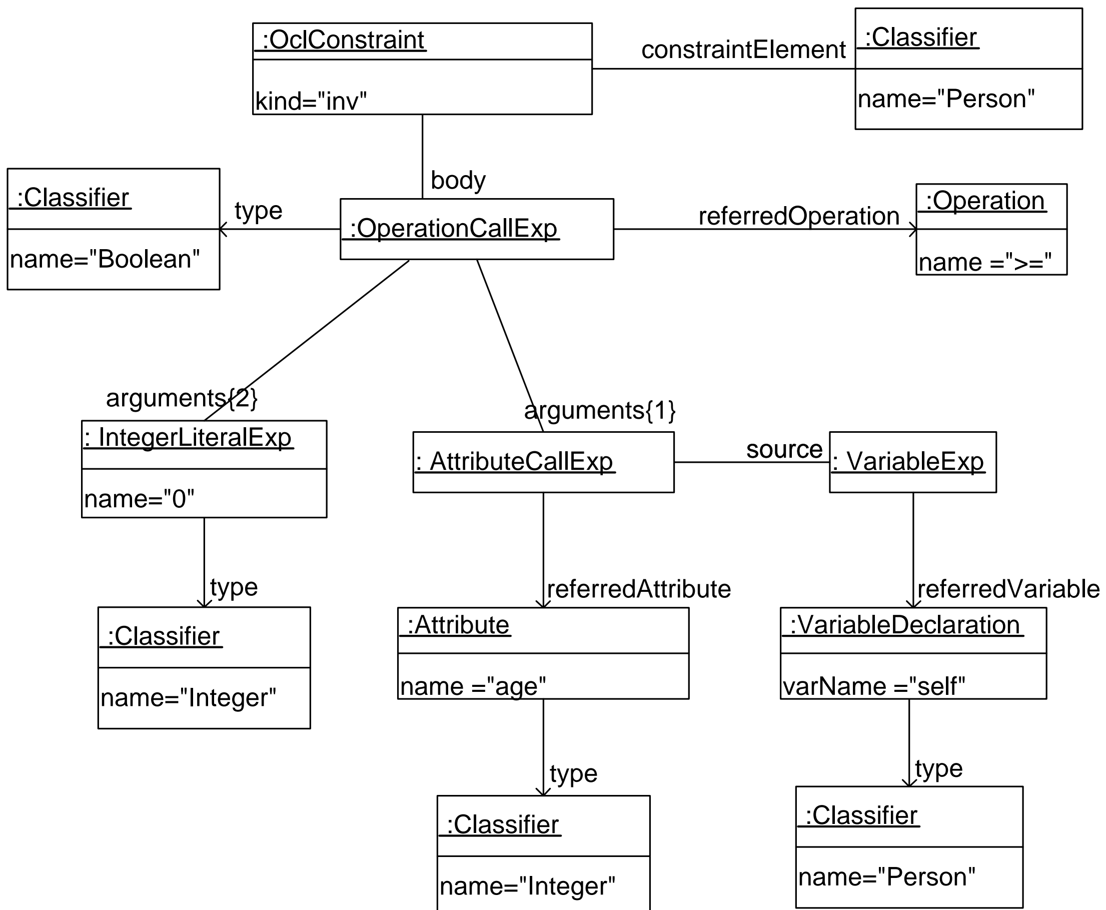
- `context Person inv:`
  `if (self.isUnemployed =false ) then income >= 3000 else income < 3000`



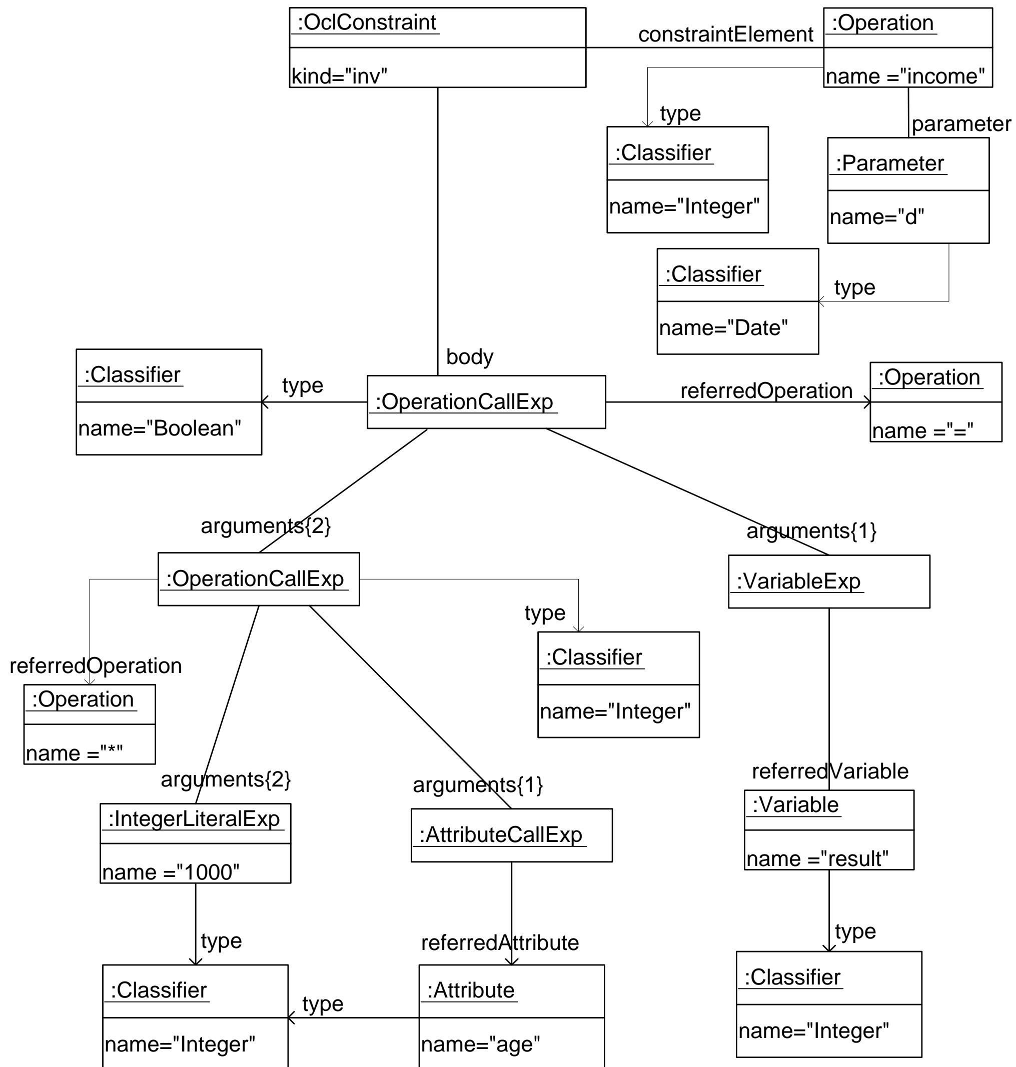
- ```
  context Person inv: let income:Integer=self.job.salary->sum()
  in if isUnemployed then income < 100 else income > 100
  ```


  ```

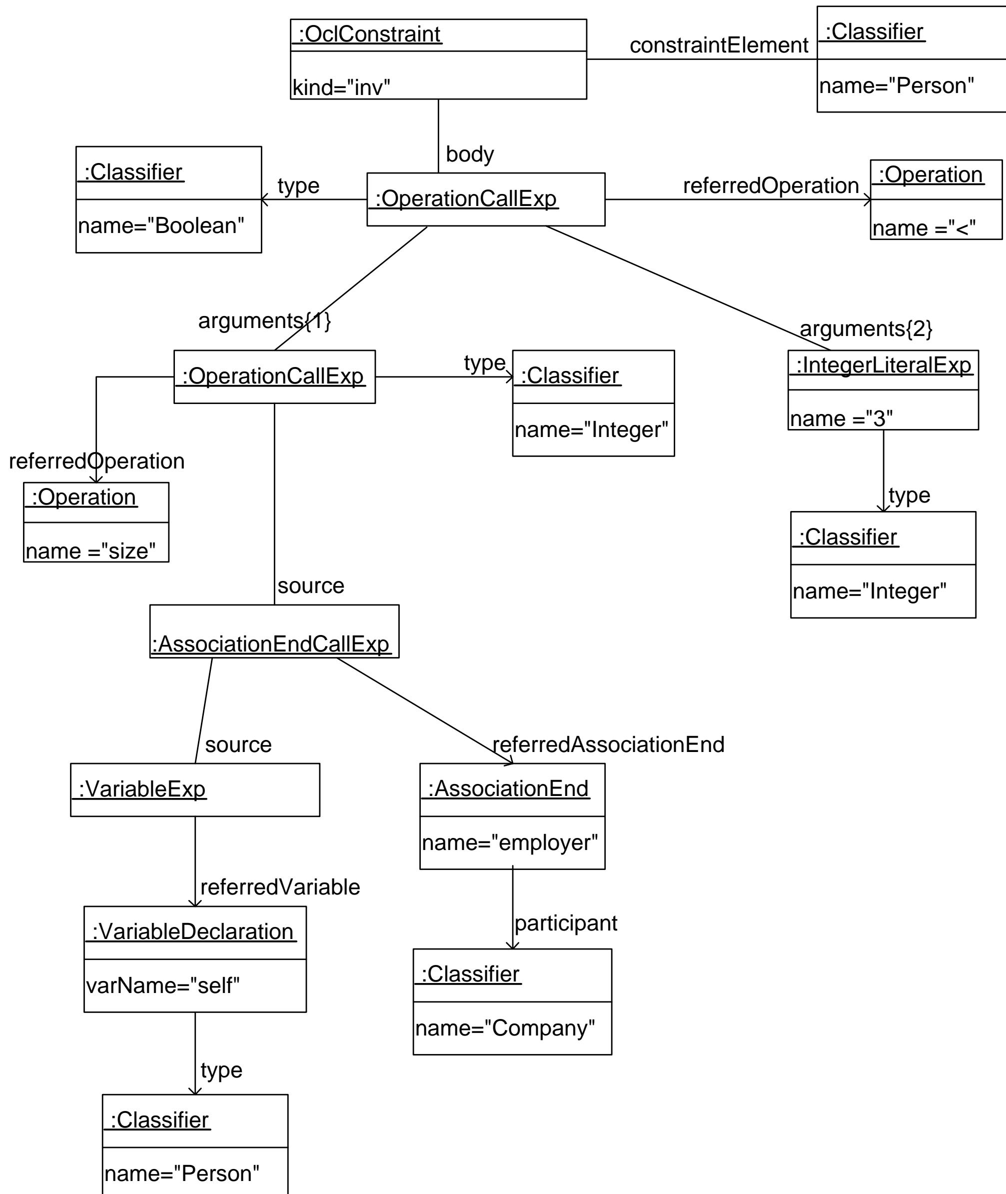
- `context Person inv: self.age >= 0`



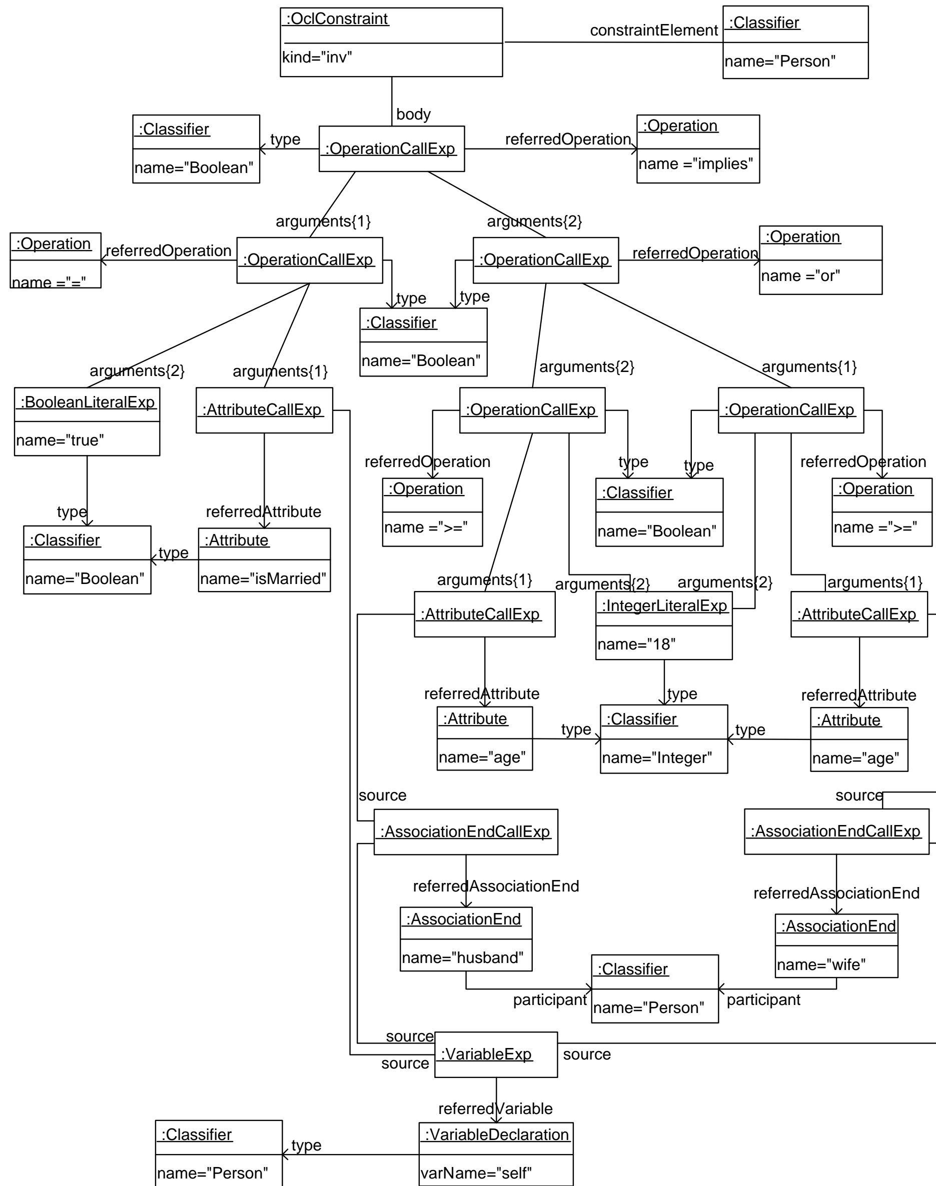
- `context Person::income(d:Date):Integer post:`
  `result = age * 1000`



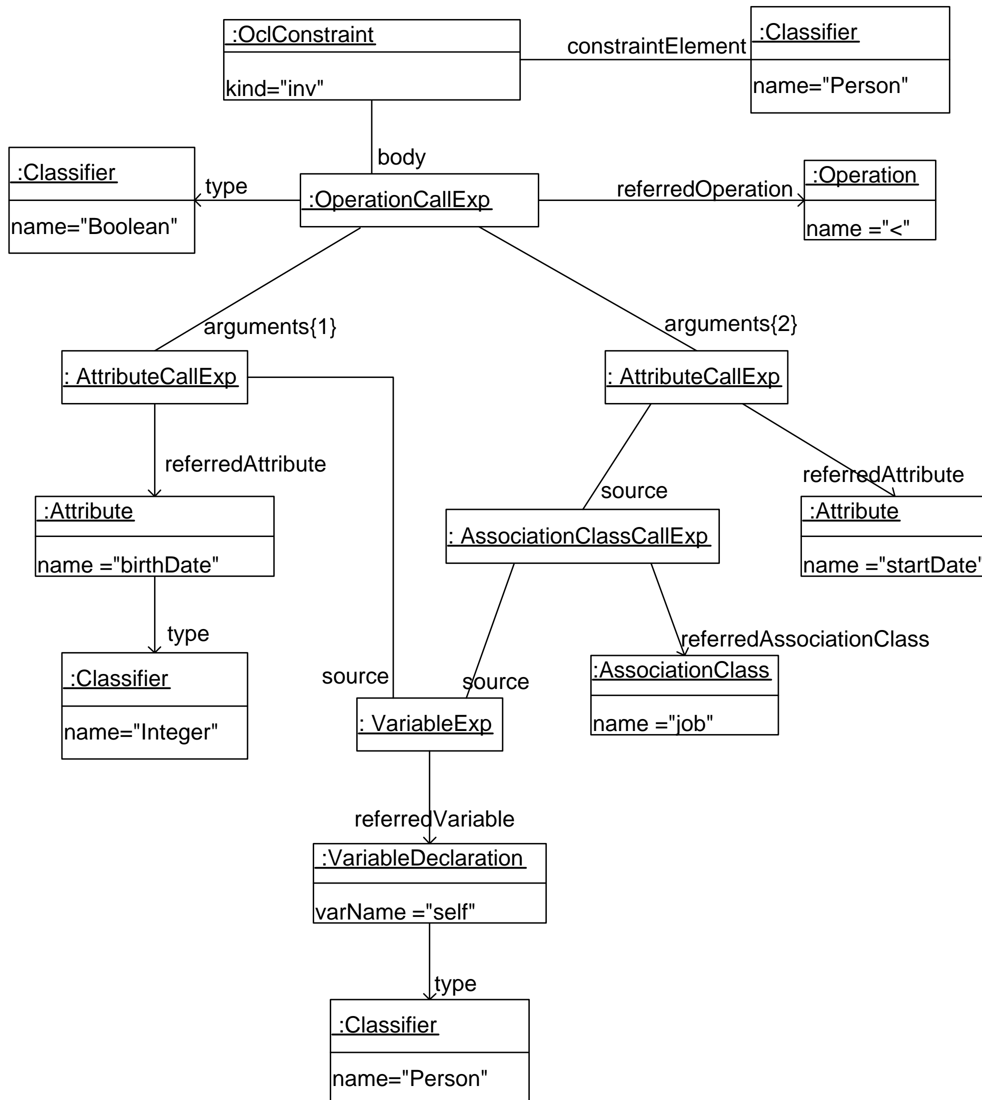
- `context Person inv: self.employer->size() < 3`



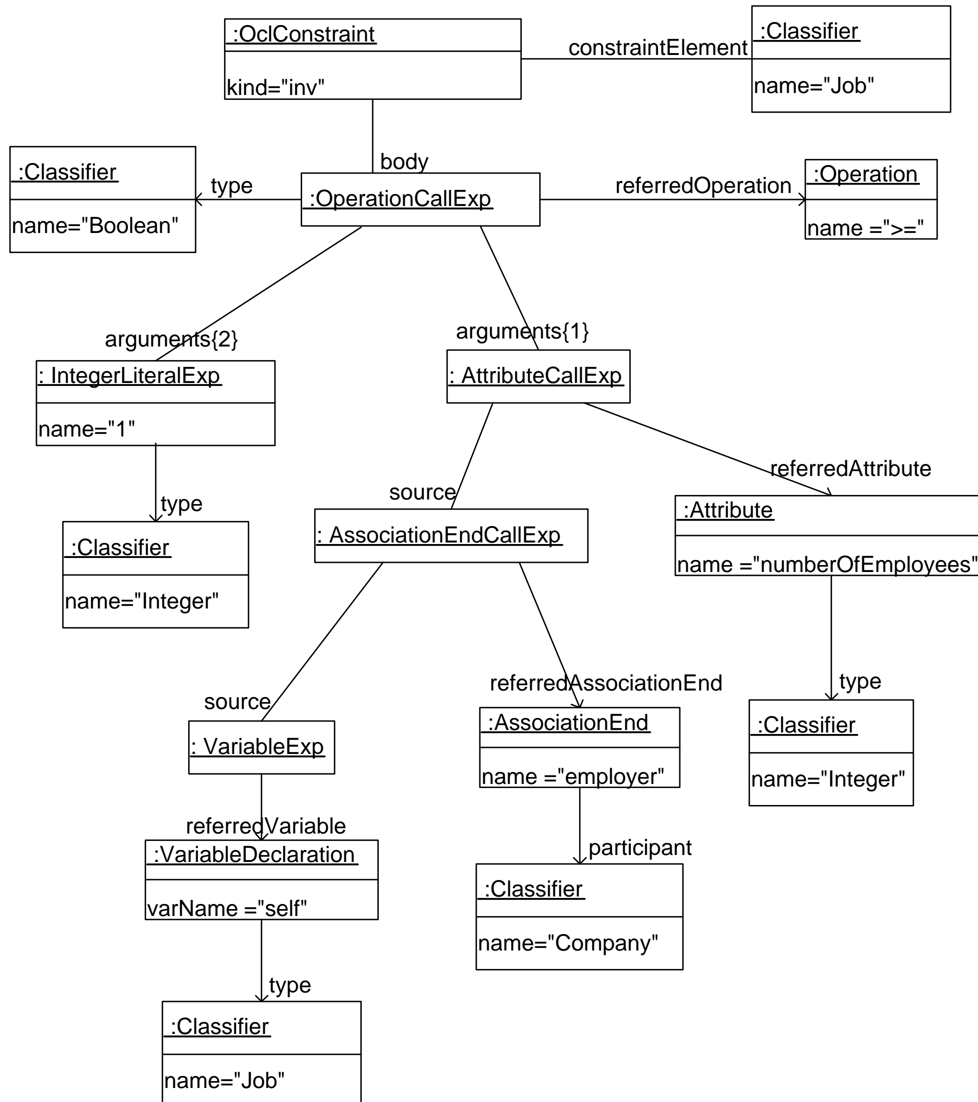
- `context Person inv: self.isMarried = true implies ((self.wife.age >= 18) or (self.husband.age >= 18))`



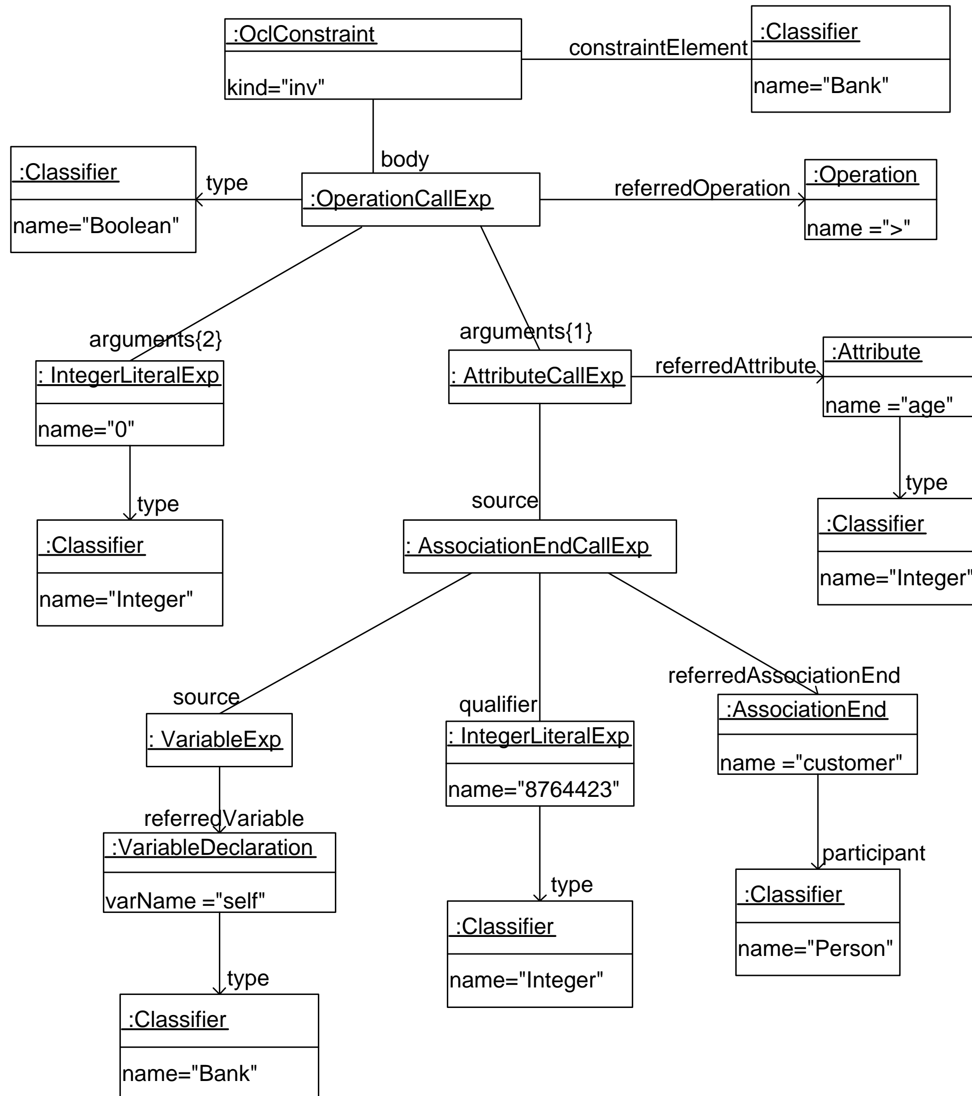
- `context Person inv: self.birthDate < self.job.startDate`



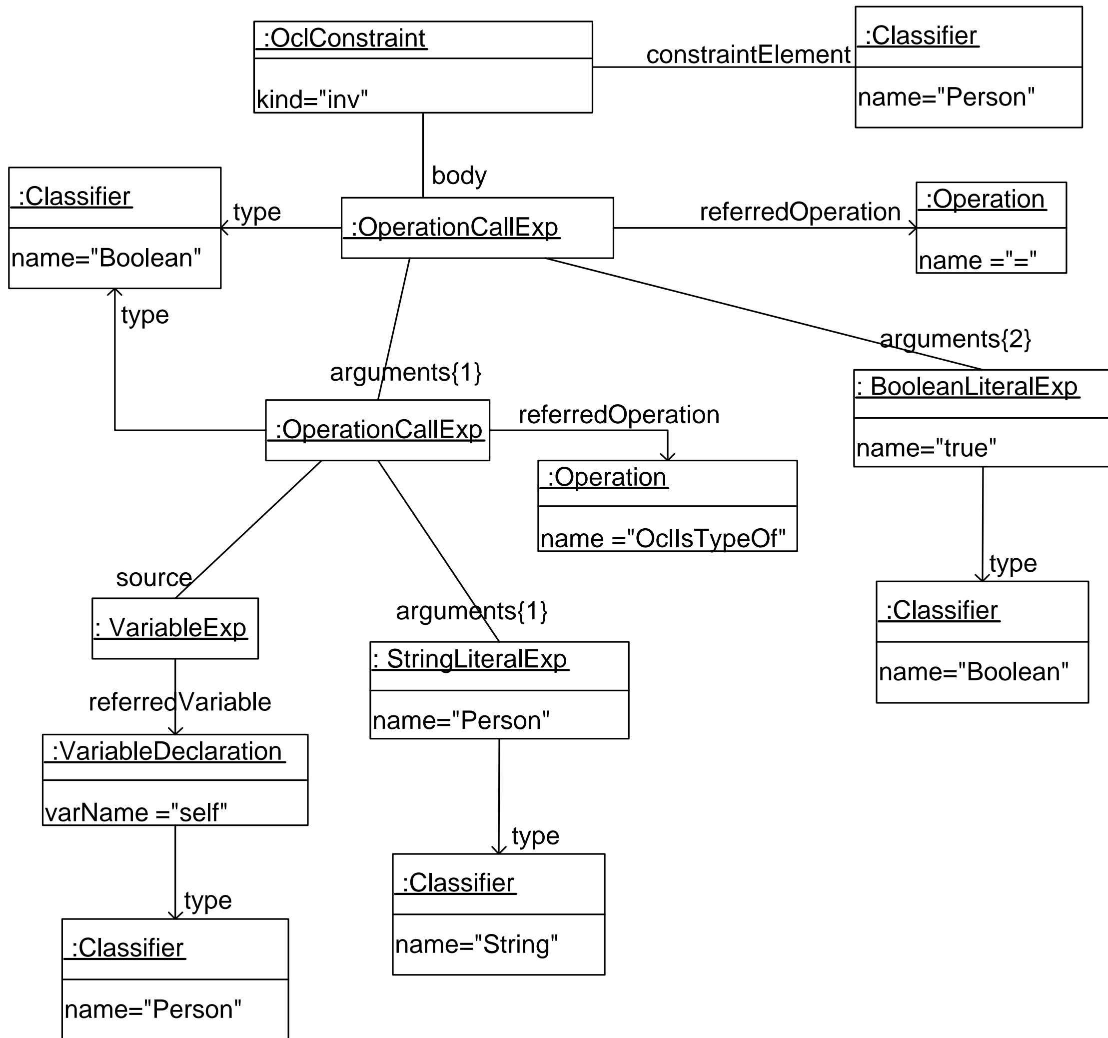
- `context Job inv: self.employer.numberOfEmployees >=1`



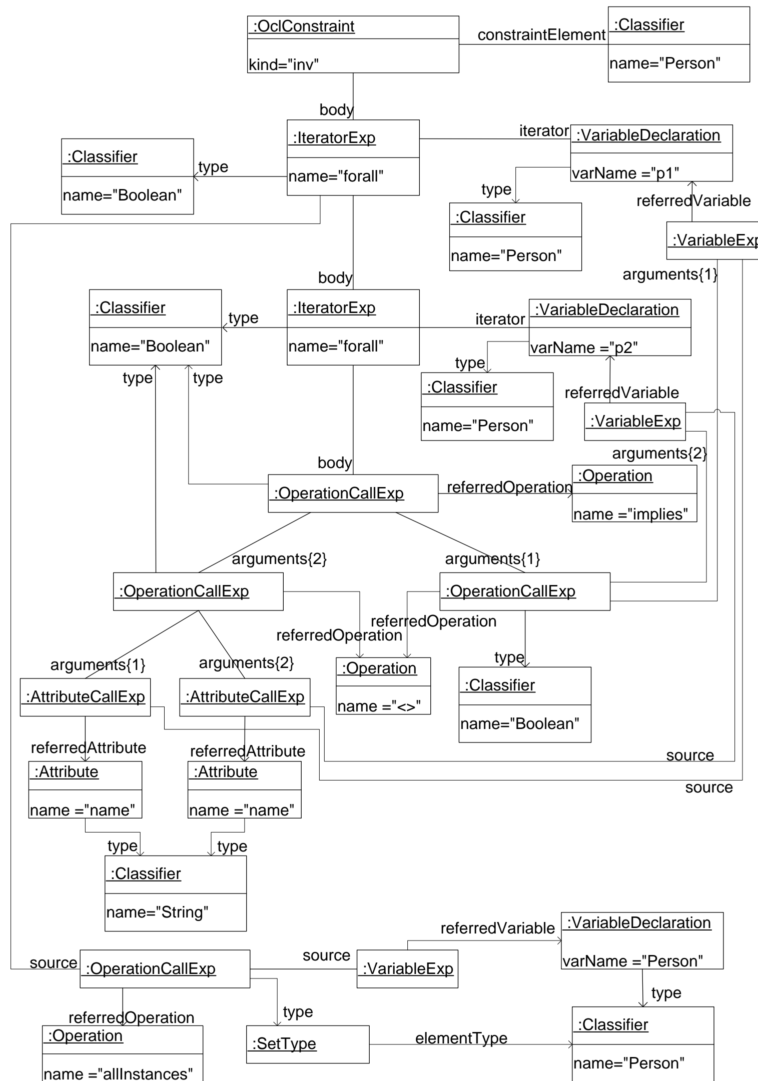
- `context Bank inv: self.customer[8764423].age > 0`



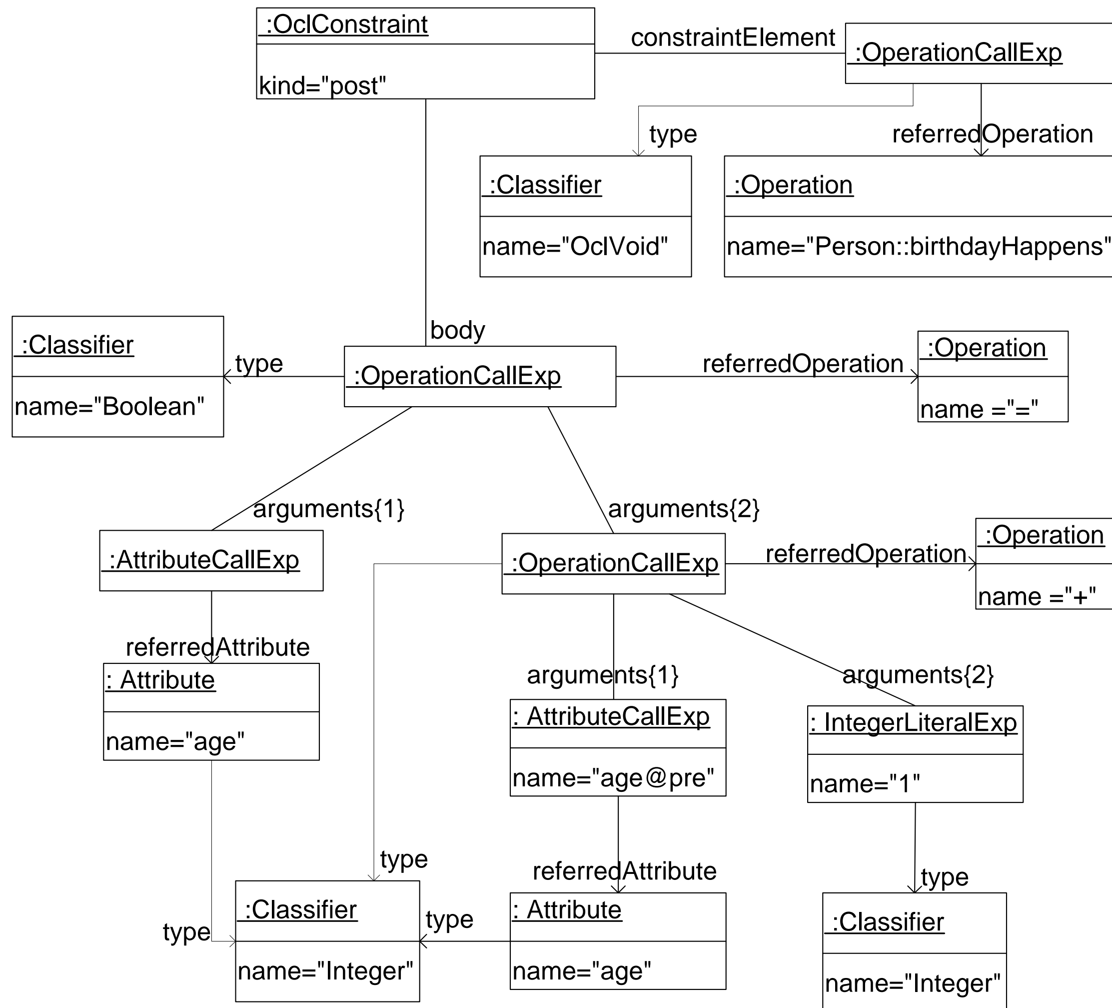
- `context Person inv: self.oclIsTpeOf(Person) = true`



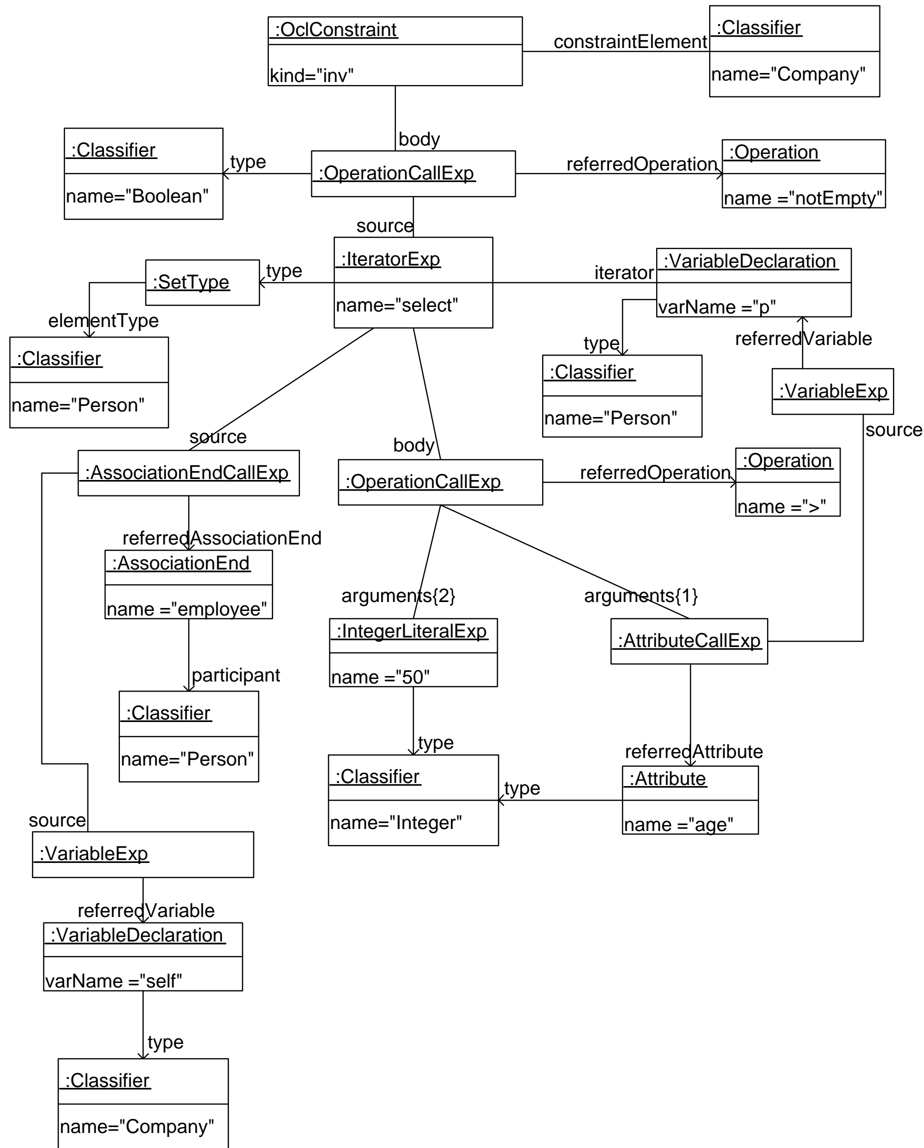
-
```
context Person inv:
Person.allInstances()->forall(p1, p2 | p1<>p2 implies p1.name<>p2.name)
```



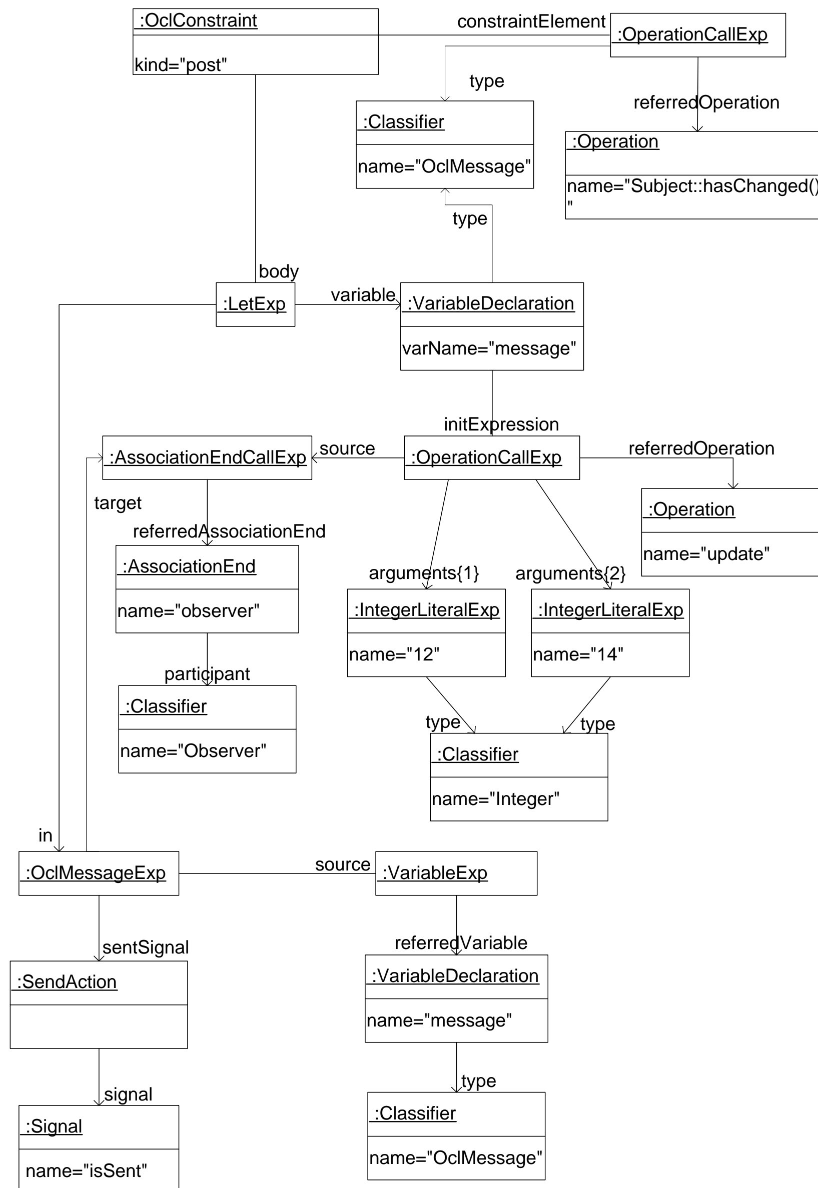
- `context Person::birthdayHappens() post:`
  `age = age@pre + 1`



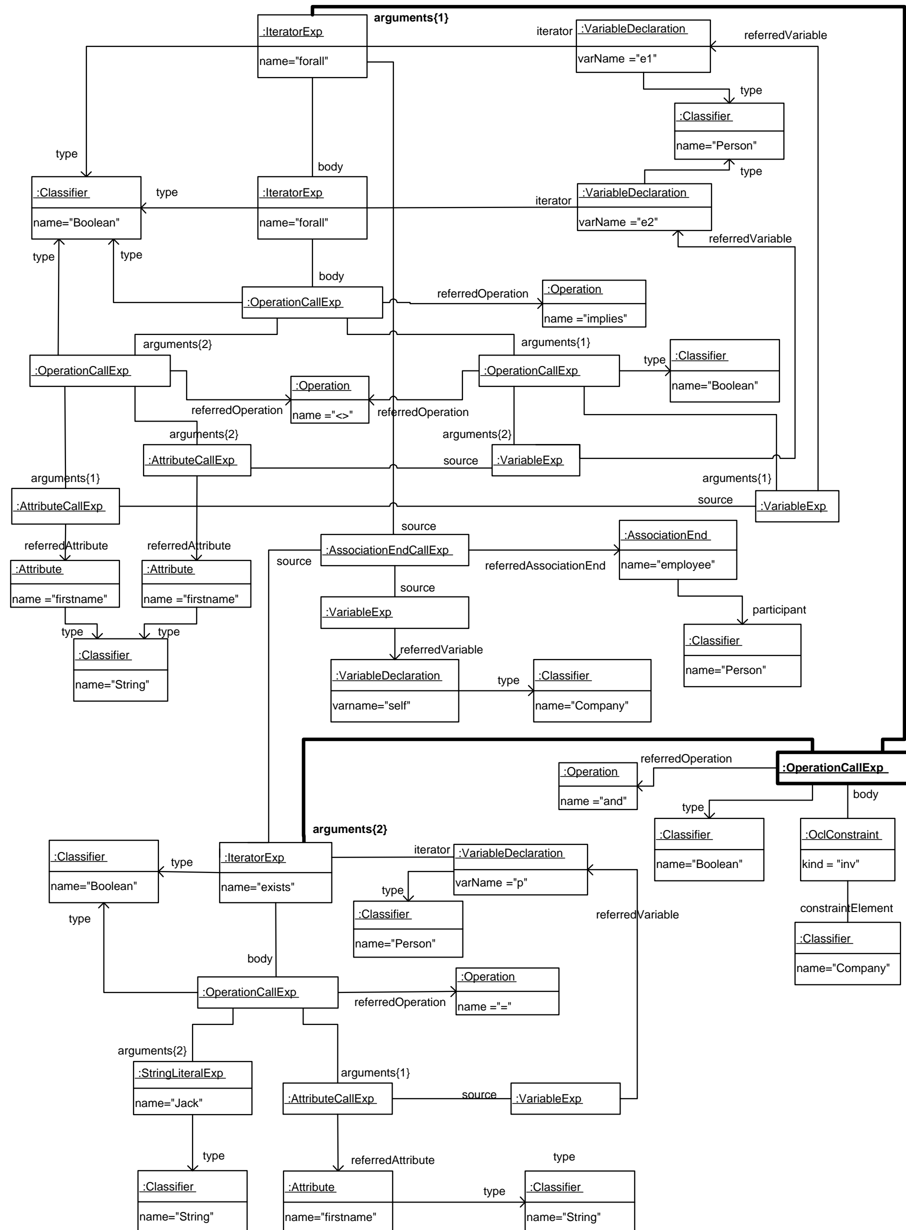
- `context Company inv:`
  `self.employee->select(p | p.age > 50)->notEmpty()`



- `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´)
  ```


  ```

Literaturverzeichnis

[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`.